Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators

Mehdi Amini

▶ **To cite this version:**

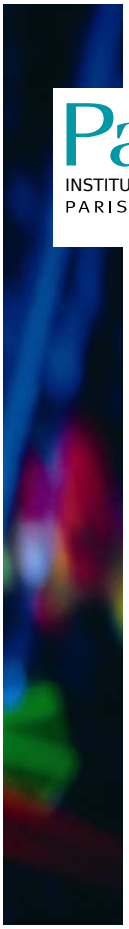
Mehdi Amini. Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators. Other [cs.OH]. Ecole Nationale Supérieure des Mines de Paris, 2012. English. NNT : 2012ENMP0105 . pastel-00958033

HAL Id: pastel-00958033
https://pastel.hal.science/pastel-00958033

Submitted on 11 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 :
Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Mehdi Amini

le 13 décembre 2012

Transformations de programme automatiques et source-à-source pour accélérateurs matériels de type GPU

~ ~ ~

Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators

Directeur de thèse : **François Irigoin**
Co-encadrement de la thèse : **Fabien Coelho**
Co-encadrement de la thèse : **Ronan Keryell**

Jury

M. Cédric Bastoul, Maître de Conférence, Alchemy/LRI/INRIA, Université Paris-Sud	Examinateur
M. Philippe Clauss, Professeur, ICPS/LSIIT, Université de Strasbourg	Examinateur
M. Fabien Coelho, Maître-Assistant, CRI, MINES ParisTech	Co-encadrant de la thèse
M. Albert Cohen, Directeur de Recherche, PARKAS, INRIA Rocquencourt	Rapporteur
M. Alain Darte, Directeur de Recherche, LIP, CNRS	Rapporteur
M. François Irigoin, Directeur de Recherche, CRI, MINES ParisTech	Directeur de thèse
M. Ronan Keryell, Directeur Scientifique, SILKAN	Co-encadrant de la thèse
M. Sanjay Rajopadhye, Professeur, CS/ECE Departments, Colorado State University	Rapporteur

T H È S E

MINES ParisTech
Centre de Recherche en Informatique
35 rue Saint-Honoré, 77305 Fontainebleau, France

To my beloved father.

Remerciements

Avant toute chose, relever ce défi personnel n'aurait pas été possible sans le soutien indéfectible de ma tendre épouse, ces trois dernières années n'ont pas été les plus reposantes et j'admire sa patience et sa tolérance vis à vis de son mari travaillant à 500km de notre domicile et s'absentant régulièrement pour toutes sortes de conférences ou formations.

Mes pensées vont évidement à mon fils Tim, mon moteur principal aujourd'hui, mais aussi celui qui m'épuise à longueur de journées (du moins le peu de celles que j'ai passé avec lui à la maison ces dernières années). A croire qu'il puise son énergie qui semble infinie dans la notre.

J'ai la chance d'avoir des parents formidables à qui j'ai posé beaucoup de difficultés qu'ils ont su surmonter pour me pousser à faire des études. Peut-être que Tim qui me le rend bien me fait mesurer l'ampleur de la tâche. Je les en remercie milles fois.

Une aventure commence souvent avec une rencontre, et pour satisfaire ma nature nostalgique, je vais la raconter. C'était le 4 juin 2009, lorsque mon étoile m'a conduit à assister à la *Journée jeunes chercheurs sur les Multiprocesseurs et Multicoeurs* à Paris. Quelle chance d'être encore là à la fin de la journée lorsqu'un curieux personnage (non, je ne me risquerai pas à une description *à la* Zola) s'est levé pour annoncer qu'il cherchait des candidats passionnés à un projet un peu fou (ou des candidats un peu fous pour un projet passionnant, je ne sais plus très bien...). Il n'en fallait pas plus pour piquer ma curiosité et, après une description plus détaillée, j'étais fasciné par la folie apparente du projet et j'oubliais être le fou potentiel. Je repartais vers mon TGV, une carte de visite dans la poche. Peu après minuit le soir même, sitôt rentré chez moi, j'envoyais mon CV à *Ronan Keryell*. Le doigt dans l'engrenage...

Trois jours plus tard, j'étais invité par *François Irigoin* à lui rendre visite à Fontainebleau. C'est finalement le 1er juillet que je me suis rendu dans les locaux du Centre de Recherche en Informatique (CRI) pour y rencontrer celui qui allait me proposer de diriger cette thèse. Neuf jours plus tard, Ronan me proposait de réaliser ces travaux dans une petite entreprise. Était-ce le goût du risque ? Ou peut-être le nom tellement *cool* de *HPC Project* ? Le challenge de la thèse n'était pas assez difficile à relever en lui même qu'il fallait y associer une entreprise ? A moins que Ronan n'ait simplement su employer les arguments qui font mouche pour un esprit pur: *Nous sommes prêts à te prendre en thèse CIFRE entre HPC Project et CRI, histoire de combiner le fun de la thèse avec une rémunération raisonnable et de voir la vie à 2 endroits différents.*

Le 6 décembre j'étais encore ingénieur à l'Université de Strasbourg, le 7 décembre 2009

l'aventure commençait vraiment. Le jour même je m'envolais pour une *école thématique sur le calcul hautes performances sur accélérateurs matériels* et je rencontrais deux personnes dont j'ignorais l'importance de nos futurs interactions: *Béatrice Creusillet* et *Stéphanie Even*. Je ne pouvais rêver meilleur départ.

A ces rencontres s'en sont succédées de nombreuses autres, et il m'est impossible de toutes les mentionner ici. Certaines ont été plus marquantes, ou ont eu plus d'impact dans le cadre de mes travaux, que ce soit vis-à-vis de l'orientation de mes recherches ou simplement de mes conditions de "travail".

Bien sûr Ronan et François, pour m'avoir supporté, ont la plus grande part de mérites, ils ont été en quelque sorte le Yin et le Yang, le Chaud et le Froid, l'Alpha et l'Omega, enfin bref les éléments essentiels au maintient d'un équilibre pendant ces trois années. J'ai le privilège de continuer à travailler avec Ronan aujourd'hui. *Fabien Coelho*, qui a rejoint l'aventure en cours de route (et il n'avait pas l'excuse de l'ignorance, il avait déjà eu à me supporter sur pipsdev@), a su apporter une couleur différente et compléter efficacement la direction de mes travaux. J'ai beaucoup progressé à leur contact, techniquement et humainement. Leurs relectures attentives et leur nombreux commentaires ont contribué significativement à l'amélioration de ce manuscrit, et j'atteste que tout ce qui peut empêcher ce dernier de répondre à l'exigence du lecteur reste de mon seul fait, que ce soit de ma paresse ou de mon incompétence (je me plais à croire au premier et à douter du second).

Au rang de mes collègues, j'ai eu la chance de côtoyer des gens très brillant. Il est très facile d'être modeste quand on évolue dans un pareil environnement.

Je reste admiratif devant la qualité du travail de Béatrice qui continue à m'impressionner chaque semaine par son efficacité. Je déconseille la lecture de son manuscrit de thèse à tout doctorant: la qualité (et la taille) de l'ouvrage pose une base à décourager de commencer sa propre rédaction. Si elle n'était pas si gentille, dévouée, et tellement agréable, je pourrais peut-être songer à lui trouver un défaut. Au même titre que Ronan, je suis chanceux de pouvoir continuer à la compter parmi mes collègues pour la suite de nos travaux sur Par4All.

Je suis heureux d'avoir pu travailler avec Serge *papillon* Guelton pour la teneur nos discussions techniques, pour sa *fraîcheur*, et pour son enthousiasme communicatif. Nos différences ont été sources d'enrichissement, que j'espère mutuel.

Le choix de m'engager avec Silkan (HPC-Project à cette époque) comportait des risques, mais ce fut payant. En dehors de l'expérience intéressante qui a consister à trouver

l'équilibre entre la recherche et les problématiques *terre-à-terre* d'une jeune entreprise, j'ai côtoyé (et continue à travailler avec) des gens brillants. Parmi eux Pierre Villalon, François-Xavier Pasquier, Thierry Porcher, Janice Onanian McMahon, Onil Nazra Persada Goubier, ou Yannick Langlois ; mais également notre PDG-CEO Pierre Fiorini que je remercie pour la confiance qu'il m'accorde.

Le CRI est un laboratoire accueillant, situé dans un cadre magnifique pour ne rien gâcher. J'ai apprécié d'y rencontrer et d'échanger sur la compilation avec Pierre Jouvelot, Karim Barkati, Antoniu Pop, Amira Mensi, Rachid Habel, ou encore Claude Tadonki ; et d'échanger sur divers autres sujets avec Georges-André Silber, Claire Medrala, Laurent Daverio, ou Jacqueline Altimira.

J'ai découvert la communauté française de compilation, grâce aux immanquable *journées compilation* que je soupçonne de devoir beaucoup à l'implication de Laure Gonnord parmi d'autres. J'y ai fait beaucoup de rencontre, et passé d'excellent moment. J'espère que mon éloignement géographique actuel me laissera des opportunités d'assister et de partager nos travaux lors de quelques journées futures.

Je me dois de dire que je ne me serai pas lancé dans ce projet sans avoir *vécu* à côté l'équipe Informatique et Calcul Parallèle Scientifique (ICPS) de Strasbourg et son ambiance de travail inégalable. Les meilleurs enseignants de mon cursus universitaire en font (ou faisait) partis, parmi eux : Alain Ketterlin, Vincent Loechner, Eric Violard, Catherine Mongenet, Arnaud Giersch, ou encore Benoit Meister. J'ai vraiment apprécié l'informatique en suivant leur enseignement. J'ai une pensée particulière pour *Philippe Clauss* avec qui j'ai découvert la programmation parallèle, et qui, ce 4 juin 2009 dans le TGV qui nous ramenait de Paris, m'a encouragé à postuler sur ce sujet de thèse.

Pour m'avoir montré une autre facette de la recherche, et m'avoir encadré (supporté ?) pendant mon stage de Master à l'Observatoire Astronomique de Strasbourg, je remercie chaleureusement Dominique Aubert.

Je suis très honoré de ne compter dans mon jury que des gens que j'admire pour la qualité de leur travaux, et que j'apprécie en dehors pour tous nos contacts passés et, je l'espère, futurs. je suis reconnaissant à mes rapporteurs d'avoir accepté cette charge, je mesure la quantité de travail qui leur a été demandé, et j'apprécie particulièrement le sérieux de leur lecture et de leurs commentaires.

Il me reste un petit mot pour Benoît Pin qui à partager notre bureau à Fontainebleau, et un autre petit mot pour relever que j'ai apprécié le grand nombre de formations de qualité proposées et surtout la gestion efficace de Régine Molins de l'École des Mines et

d'Alexandrine Jamin de ParisTech.

Enfin, mes travaux des trois dernières années n'auraient été possible sans le soutien financier de (ordre alphabétique des projets): l'Association Nationale de la Recherche et de la Technologie (ANRT) et le dispositif CIFRE, l'Agence Nationale pour la Recherche (ANR) et le projet MediaGPU, le Pôle de Compétitivité SYSTEM@TIC et le projet OpenGPU, et bien entendu Silkan.

Abstract

Since the beginning of the 2000s, the raw performance of processors stopped its exponential increase. The modern graphic processing units (GPUs) have been designed as array of hundreds or thousands of compute units. The GPUs' compute capacity quickly leads them to be diverted from their original target to be used as accelerators for general purpose computation. However programming a GPU efficiently to perform other computations than 3D rendering remains challenging.

The current jungle in the hardware ecosystem is mirrored by the software world, with more and more programming models, new languages, different APIs, etc. But no *one-fits-all* solution has emerged.

This thesis proposes a compiler-based solution to partially answer the three "*P*" properties: Performance, Portability, and Programmability. The goal is to transform automatically a sequential program into an equivalent program accelerated with a GPU. A prototype, Par4All, is implemented and validated with numerous experiences. The programmability and portability are enforced by definition, and the performance may not be as good as what can be obtained by an expert programmer, but still has been measured excellent for a wide range of kernels and applications.

A survey of the GPU architectures and the trends in the languages and framework design is presented. The data movement between the host and the accelerator is managed without involving the developer. An algorithm is proposed to optimize the communication by sending data to the GPU as early as possible and keeping them on the GPU as long as they are not required by the host. Loop transformations techniques for kernel code generation are involved, and even well-known ones have to be adapted to match specific GPU constraints. They are combined in a coherent and flexible way and dynamically scheduled within the compilation process of an interprocedural compiler. Some preliminary work is presented about the extension of the approach toward multiple GPUs.

Résumé

Depuis le début des années 2000, la performance brute des cœurs des processeurs a cessé son augmentation exponentielle. Les circuits graphiques (GPUs) modernes ont été conçus comme des circuits composés d'une véritable grille de plusieurs centaines voir milliers d'unités de calcul. Leur capacité de calcul les a amenés à être rapidement détournés de leur fonction première d'affichage pour être exploités comme accélérateurs de calculs généralistes. Toutefois programmer un GPU efficacement en dehors du rendu de scènes 3D reste un défi.

La jungle qui règne dans l'écosystème du matériel se reflète dans le monde du logiciel, avec de plus en plus de modèles de programmation, langages, ou API, sans laisser émerger de solution universelle.

Cette thèse propose une solution de compilation pour répondre partiellement aux trois "*P*" propriétés : Performance, Portabilité, et Programmabilité. Le but est de transformer automatiquement un programme séquentiel en un programme équivalent accéléré à l'aide d'un GPU. Un prototype, Par4All, est implémenté et validé par de nombreuses expériences. La programmabilité et la portabilité sont assurées par définition, et si la performance n'est pas toujours au niveau de ce qu'obtiendrait un développeur expert, elle reste excellente sur une large gamme de noyaux et d'applications.

Une étude des architectures des GPUs et les tendances dans la conception des langages et cadres de programmation est présentée. Le placement des données entre l'hôte et l'accélérateur est réalisé sans impliquer le développeur. Un algorithme d'optimisation des communications est proposé pour envoyer les données sur le GPU dès que possible et les y conserver aussi longtemps qu'elle ne sont pas requises sur l'hôte. Des techniques de transformations de boucles pour la génération de code noyau sont utilisées, et même certaines connues et éprouvées doivent être adaptées aux contraintes posées par les GPUs. Elles sont assemblées de manière cohérente, et ordonnancées dans le flot d'un compilateur interprocédural. Des travaux préliminaires sont présentés au sujet de l'extension de l'approche pour cibler de multiples GPUs.

Table of Contents

Remerciements . . . iii
Abstract . . . vii
Résumé . . . ix

1 Introduction **1**
1.1 The Prophecy . . . 2
1.2 Motivation . . . 3
1.3 Outline . . . 7

2 General-Purpose Processing on GPU : History and Context **11**
2.1 History . . . 12
2.2 Languages, Frameworks, and Programming Models . . . 14
2.2.1 Open Graphics Library (OpenGL) . . . 15
2.2.2 Shaders . . . 15
2.2.3 Brook and BrookGPU . . . 17
2.2.4 Nvidia Compute Unified Device Architecture (CUDA) . . . 18
2.2.5 AMD Accelerated Parallel Processing, *FireStream* . . . 20
2.2.6 Open Computing Language (OpenCL) . . . 20
2.2.7 Microsoft DirectCompute . . . 21
2.2.8 C++ Accelerated Massive Parallelism (AMP) . . . 21
2.2.9 ΣC and the MPPA Accelerator . . . 23
2.2.10 Directive-Based Language and Frameworks . . . 23
2.2.11 Automatic Parallelization for GPGPU . . . 30
2.3 Focus on OpenCL . . . 30
2.3.1 Introduction . . . 31
2.3.2 OpenCL Architecture . . . 31
2.3.2.1 Platform Model . . . 32
2.3.2.2 Execution Model . . . 32
2.3.2.3 Memory Model . . . 35
2.3.2.4 Programming Model . . . 37
2.3.3 OpenCL Language . . . 38
2.3.3.1 Conclusion . . . 39

2.4 Target Architectures . . . 39
2.4.1 From Specialized Hardware to a Massively Parallel Device . . . 40
2.4.2 Building a GPU . . . 40
2.4.3 Hardware Atomic Operations . . . 42
2.4.4 AMD, from R300 to Graphics Core Next . . . 43
2.4.5 Nvidia Computing Unified Device Architecture, from G80 to Kepler 48
2.4.6 Impact on Code Generation . . . 52
2.4.7 Summary . . . 54
2.5 Conclusion . . . 55

3 Data Mapping, Communications and Consistency 61
3.1 Case Study . . . 63
3.2 Array Region Analysis . . . 64
3.3 Basic Transformation Process . . . 68
3.4 Region Refinement Scheme . . . 70
3.4.1 Converting Convex Array Regions into Data Transfers . . . 73
3.4.2 Managing Variable Substitutions . . . 74
3.5 Limits . . . 76
3.6 Communication Optimization Algorithm . . . 77
3.6.1 A New Analysis: Kernel Data Mapping . . . 78
3.6.2 Definitions . . . 79
3.6.3 Intraprocedural Phase . . . 80
3.6.4 Interprocedural Extension . . . 81
3.6.5 Runtime Library . . . 82
3.7 Sequential Promotion . . . 84
3.7.1 Experimental Results . . . 86
3.8 Related Work . . . 87
3.8.1 Redundant Load-Store Elimination . . . 88
3.8.1.1 Interprocedural Propagation . . . 89
3.8.1.2 Combining Load and Store Elimination . . . 89
3.9 Optimizing a Tiled Loop Nest . . . 90
3.10 Conclusion . . . 93

4 Transformations for GPGPU 95
4.1 Introduction . . . 96

4.2 Loop Nest Mapping on GPU . . . 98
4.3 Parallelism Detection . . . 101
4.3.1 Allen and Kennedy . . . 102
4.3.2 Coarse Grained Parallelization . . . 103
4.3.3 Impact on Code Generation . . . 104
4.4 Reduction Parallelization . . . 105
4.4.1 Detection . . . 105
4.4.2 Reduction Parallelization for GPU . . . 109
4.4.3 Parallel Prefix Operations on GPUs . . . 111
4.5 Induction Variable Substitution . . . 111
4.6 Loop Fusion . . . 112
4.6.1 Legality . . . 113
4.6.2 Different Goals . . . 115
4.6.3 Loop Fusion for GPGPU . . . 116
4.6.4 Loop Fusion in PIPS . . . 118
4.6.5 Loop Fusion Using Array Regions . . . 124
4.6.6 Further Special Considerations . . . 126
4.7 Scalarization . . . 127
4.7.1 Scalarization inside Kernel . . . 128
4.7.2 Scalarization after Loop Fusion . . . 128
4.7.3 Perfect Nesting of Loops . . . 130
4.7.4 Conclusion . . . 131
4.8 Loop Unrolling . . . 132
4.9 Array Linearization . . . 133
4.10 Toward a Compilation Scheme . . . 134

5 Heterogeneous Compiler Design and Automation 137
5.1 Par4All Project . . . 138
5.2 Source-to-Source Transformation System . . . 140
5.3 Programmable Pass Managers . . . 141
5.3.1 PyPS . . . 142
5.3.1.1 Benefiting from Python: *on the shoulders of giants* . . . 142
5.3.1.2 Program Abstractions . . . 143
5.3.1.3 Control Structures . . . 143
5.3.1.4 Builder . . . 146

5.3.1.5 Heterogeneous Compiler Developements . . . 146
5.3.2 Related Work . . . 149
5.3.3 Conclusion . . . 150
5.4 Library Handling . . . 151
5.4.1 Stubs Broker . . . 152
5.4.2 Handling Multiple Implementations of an API: Dealing with External Libraries . . . 153
5.5 Tool Combinations . . . 155
5.6 Profitability Criteria . . . 156
5.6.1 Static Approach . . . 157
5.6.2 Runtime Approach . . . 157
5.6.3 Conclusion . . . 158
5.7 Version Selection at Runtime . . . 158
5.8 Launch Configuration Heuristic . . . 159
5.8.1 Tuning the Work-Group Size . . . 159
5.8.2 Tuning the Block Dimensions . . . 162
5.9 Conclusion . . . 163

6 Management of Multi-GPUs 165
6.1 Introduction . . . 166
6.2 Task Parallelism . . . 166
6.2.1 The StarPU Runtime . . . 166
6.2.2 Task Extraction in PIPS . . . 167
6.2.3 Code Generation for StarPU . . . 168
6.3 Data Parallelism Using Loop Nest Tiling . . . 170
6.3.1 Performance . . . 171
6.4 Related Work . . . 173
6.5 Conclusion . . . 175

7 Experiments 177
7.1 Hardware Platforms Used . . . 178
7.2 Benchmarks, Kernels, and Applications Used for Experiments . . . 179
7.3 Parallelization Algorithm . . . 180
7.4 Launch Configuration . . . 180
7.5 Scalarization . . . 181

7.5.1 Scalarization inside Kernel . . . 191
7.5.2 Full Array Contraction . . . 191
7.5.3 Perfect Nesting of Loops . . . 191
7.6 Loop Unrolling . . . 195
7.7 Array Linearization . . . 195
7.8 Communication Optimization . . . 197
7.8.1 Metric . . . 197
7.8.2 Results . . . 199
7.8.3 Comparison with Respect to a Fully Dynamic Approach . . . 199
7.8.4 Sequential Promotion . . . 201
7.9 Overall Results . . . 202
7.10 Multiple GPUs . . . 205
7.10.1 Task Parallelism . . . 205
7.10.2 Data Parallelism . . . 206
7.11 Conclusions . . . 208

8 Conclusion **211**

Personal Bibliography **217**

Bibliography **219**

Acronyms **253**

Résumé en français **257**
1 Introduction . . . 258
2 Calcul généraliste sur processeurs graphiques : histoire et contexte . . . 265
3 Placement des données, communications, et cohérence . . . 280
4 Transformations pour GPGPU . . . 288
5 Conception de compilateurs hétérogènes et automatisation . . . 299
6 Gestion de multiples accélérateurs . . . 310
7 Expériences . . . 313
8 Conclusion . . . 314

List of Figures

1.1 von Neumann architecture 3

1.2 More than three decades of prosperity, the misquoted Moore's law (source [Sutter 2005], updated 2009, reprinted here with the kind permission of the author). 4

1.3 The free lunch is over. Now welcome to the hardware jungle (source [Sutter 2011], reprinted here with the kind permission of the author). 5

1.4 The three *P* properties. 6

2.1 Performance evolution for single-precision floating point computation, for both Nvidia GPUs and Intel CPUs between 2003 and 2012, computed from vendors' datasheets. 14

2.2 Example of a `saxpy` OpenGL 4.4 compute shader (adapted from [Kilgard 2012]). 16

2.3 Example of a trivial *pass-through* GLSL geometry shader, which emits a vertex directly for each input vertex (source wikipedia [Wikipedia 2012b]). 17

2.4 Example of a simple `saxpy` using BrookGPU (taken from [Buck *et al.* 2004]). 18

2.5 Example of a Cg/HLSL shader for DirectCompute (source Microsoft [Deitz 2012]). 22

2.6 Rewriting a C++ computation using C++ AMP. The example shows the use of a lambda function and a `parallel_for_each` construct to express the parallelism (source Microsoft [Microsoft Corporation 2012b]). 24

2.7 A sample matrix multiplication code with hiCUDA directives (source [Han & Abdelrahman 2009]). 26

2.8 Simple example for HMPP directive-based code writing (source wikipedia [Wikipedia 2012c]). 27

2.9 Example of a PGI Accelerator code using data movement optimization (source PGI Insider [Wolfe 2011]). 28

2.10 A simple JCUDA example. Note the `IN`, `OUT`, and `INOUT` attributes in the kernel declaration that drive automatic memory transfers (source [Yan *et al.* 2009]). 29

2.11 Simplified view of the OpenCL abstraction model. A host is connected to multiple devices (GPUs, FPGAs, DPSs, . . .). OpenCL platforms are vendors' implementations that target some types of devices. A context is created for a given platform and a set of devices. Memory objects and events are created context-wise. Devices are then controlled in a given context using command queues. There can be multiple command queues per device, and a device can be associated with queues from multiple contexts and platforms. 33

2.12 UML representation of the OpenCL abstraction model (see Figure 2.11) taken from the Standard [Khronos OpenCL Working Group 2011]. 34

2.13 A mapping example of a two-dimensional loop nest iteration set into an OpenCL index range. The mapping is the simplest possible; one work-item executes one iteration of the original loop nest. The work-group size used as an illustration on figure *c* is a two-dimensional square with an edge of five. Values for `get_global_id()` and `get_local_id()` OpenCL primitives are exhibited for a particular work-group. 35

2.14 Visual example of the OpenCL memory model. Two possible mappings are illustrated: data caches are optional, and private, local, and constant memories are not necessarily dedicated. On the right the simplest mapping, for instance a CPU, merges all memory spaces onto the same piece of hardware. 37

2.15 High-level simplified GPGPU-oriented view of generic GPU architecture. . 41

2.16 Instruction Level Parallelism (ILP) versus Thread Level Parallelism (TPL), two different ways of extracting parallelism in GPUs. 42

2.17 AMD R600 Compute Unit (CU) is built on top of 5-way VLIW instructions set. Four Processing Elements (PE) and a Special Function Unit (SFU) are grouped together in a Processing Unit (PU) to process instructions. These PUs are organized in a 16-wide SIMD array. 43

2.18 Table summarizing the ALU occupation and the VLIW packing ratio for some computing kernels, taken from [Zhang *et al.* 2011b] (©2011 IEEE). . 46

2.19 The 2012 AMD architecture Graphics Core Next. No longer VLIW, the four separate SIMD pipelines are independent. A new integer scalar unit is introduced. The scheduler feeds each SIMD every four cycles (one per cycle) with a 64-wide virtual SIMD instruction. 46

2.20 Evolution of Processing Element (PE) grouping across AMD architectures. 48

2.21 The GeForce 8800 architecture (G80) introduced unified shaders where shader programmable processors can be used to replace multiple stages of the classic graphic pipeline. There are still specialized units for some graphics operations. (Source: Nvidia) 49
2.22 GT200 compute unit (CU) on the left, Fermi's on the right. Processing Elements (PE) upgrade from eight to sixteen per pipeline, but the logical SIMD width is unchanged, threads are scheduled by groups of thirty-two (source Nvidia). 50
2.23 Influence of runtime parameters on performance for different kernels and different architectures, Nvidia Fermi and AMD Evergreen. On the left the launch configuration for different kernels shows that there is no universal work-group size. On the right a matrix multiply kernel without local data store optimization is used with one to four elements processed in each thread. The upper part shows the impact on performance for both architectures while the lower part shows the occupancy of the AMD GPU and the VLIW packing ratio. Taken from [Zhang *et al.* 2011b] (©2011 IEEE). 53
2.24 Performance of two different versions of matrix multiply kernel, a *horizontal* scheme and a vertical scheme, without local memory usage for a given architecture (Fermi), depending on the input size and the activation or not of the L1 cache. Taken from [Zhang *et al.* 2011b] (©2011 IEEE). 54
2.25 Google trends for the word GPU during last decade. 55
2.26 Source-to-source compilation scheme for GPU (source [Guelton 2011a]). . . 57
2.27 Overview of the global compilation scheme. 58

3.1 Stars-PM is a N-body cosmological simulation. Here a satellite triggers a bar and spiral arms in a galactic disc. 63
3.2 Simplified global scheme commonly used in numerical simulations. 64
3.3 Outline of one time step in the Stars-PM cosmological simulation code. . . 65
3.4 Array regions on a code with a function call. 66
3.5 Array regions on a code with a `while` loop. 66
3.6 Array regions on a code with a `switch` case. 67
3.7 Basic process for mapping data to the accelerator (source [Yan *et al.* 2009], ©2011 Springer-Verlag)g. 69
3.8 Sequential source code for function `discretization`, the first step of each Stars-PM simulation main iteration. 70

3.9 Code for function `discretization` after automatic GPU code generation. . 71
3.10 Isolation of the irregular `while` loop from Figure 3.5 using array region analysis. . . . 72
3.11 Code with a `switch` case from Figure 3.6 after isolation. . . . 75
3.12 Interprocedural isolation of the outermost loop of a Finite Impulse Response. 75
3.13 Isolated version of the `KERNEL` function of the Finite Impulse Response (see Figure 3.4). . . . 76
3.14 Bandwidth for memory transfers over the PCI-Express 2.0 bus as a function of block size. Results are shown for transfers from the host to the GPU (H-TO-D) and in the opposite direction (D-TO-H), each for pinned or standard allocated memory. . . . 77
3.15 Illustration of set construction using the intraprocedural analysis on the function `iteration`. The different calls to `step` functions use and produce data on the GPU via kernel calls. Sometimes in the main loop, array `a` is read to display or to checkpoint. The interprocedural translation exploits at call site the summary computed on function `iteration`. A fix point is sought on the loop. . . . 83
3.16 Simplified code for functions `discretization` and `main` after interprocedural communication optimization. . . . 84
3.17 `gramschmidt` example taken from Polybench suite. The first part of the loop body is sequential while the following are parallel loop nests. The sequential promotion on the GPU avoids costly memory transfers. . . . 85
3.18 Illustration of the redundant load-store elimination algorithm. . . . 90
3.19 Code with communication for FIR function presented in Figure 3.4. . . . 91

4.1 Example of a short Scilab program with the generated C file. . . . 97
4.2 Example from Baghdadi et al. [Baghdadi *et al.* 2010] that illustrates how to tile a loop nest to map the GPU execution. . . . 98
4.3 Illustration of the successive steps performed to map a loop nest on the GPU. 100
4.4 The iteration set is over-approximated with a rectangular hull; a guard is added to clamp it. . . . 100
4.5 Example of Allen and Kennedy algorithm as implemented in PIPS: loops are distributed and parallelism is expressed using OpenMP pragmas. . . . 102

4.6 The impact of the two parallelization schemes on a example of code performing a correlation. Allen and Kennedy algorithm results to three different parallel loop nests expressing the maximum parallelism, while the coarse grained algorithm detects only one parallel loop leading to less synchronization but also less exposed parallelism. 105
4.7 Example of reduction detection and interprocedural propagation in PIPS. . 108
4.8 An example of reduction parallelization of an histogram using hardware atomic operations. 110
4.9 Example of induction variable substitution to enable loop nest parallelization. 112
4.10 Example of loop fusion. 113
4.11 Example of two parallel loops that can be legally fused, but the resulting loop nest would be sequential. 115
4.12 Example of a loop fusion scheme to extend the iteration set of a loop nest. 118
4.13 Example of manual kernel fusion using Thrust library and a SAXPY example. The first version is expressed using native Thrust operators and requires temporary arrays, the second version fuses the three steps in one user-defined kernel (source [Hoberock & Bell 2012]). 119
4.14 On the left, a sequence of statements, in the middle the associated Dependence Graph (DG), and on the right the corresponding Reduced Dependence Graph (RDG) obtained after clustering the vertices that belong to the same loop. In solid red the flow dependences, in dashed blue the anti-dependence, and in dotted green the special dependences that model the declaration. The DG view showed here is simplified for the sake of clarity, for instance output dependences and the loop carried dependences are omitted. 120
4.15 The algorithm begins with a pruning phase. For each direct edge between two vertices it ensures that there is no other path between them. 122
4.16 Heuristic algorithm `FUSE_RDG` to traverse the RDG and apply fusion. The graph is modified as side effect. 124
4.17 Merging two vertices in the graph while enforcing pruning as introduced in Figure 4.15. 125
4.18 The resulting code after applying the loop-fusion algorithm on the code presented in Figure 4.14a. 125
4.19 Sample code showing that inner loops have to be fused first in order to be able to fuse the outer loops without breaking the perfect nesting. 127

4.20 Only perfectly nested loops are labeled parallel to avoid GPU unfriendly loop fusion. . . . 127
4.21 Processing of example in Figure 4.1. A Scilab script compiled to C code offers good opportunities for loop fusion and array scalarization. . . . 129
4.22 Simple matrix multiplication example to illustrate the impact of scalarization. 129
4.23 Array scalarization can break the perfect nesting of loop nests, thus limiting potential parallelism when mapping on the GPU. . . . 131
4.24 Example of loop unrolling with a factor four. . . . 133
4.25 Simple matrix multiplication example to illustrate array linearization interest. 134

5.1 Source-to-source cooperation with external tools. . . . 141
5.2 PyPS class hierarchy (source [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso)]). . . . 144
5.3 Conditionals in PyPS. . . . 144
5.4 For loop is a control structure commonly involved in PyPS 144
5.5 Using exceptions to adapt the compilation process. . . . 145
5.6 Searching fix point. . . . 145
5.7 Terapix compilation scheme. . . . 147
5.8 SAC compilation scheme extract. . . . 147
5.9 OpenMP compilation scheme. . . . 148
5.10 The brokers infrastructure and compilation flow in Par4All. . . . 154
5.11 The original source code for the `potential` step that involves two calls to FFTW library. . . . 155
5.12 FFTW library requires that a *plan* is initialized. Here in the original source code, two plans are initialized for the `potential` code presented in Figure 5.11. 155
5.13 Computing the block size that maximizes the occupancy for a given kernel and a given GPU. . . . 161
5.14 The main loop nest in `syrk` benchmark from the Polybench suite and the resulting kernel. . . . 163

6.1 Task transformation process. `3mm` example from the Polybench suite automatically transformed with tasks. . . . 168

6.2 Code generated automatically with pragmas to drive the StarPU runtime. The tasks have been declared with attributes in order to drive StarPU, specifying the suitable target platform for each implementation. Task parameters are declared with a `const` qualifier when used as read-only, and with an `__attribute__((output))` when used as write-only. 169
6.3 Execution timeline when using parallel tasks on one GPU (upper part) and on two GPUs (lower part) for the `3mm` code presented in Figure 6.2. 170
6.4 Tiling transformation process illustrated on a simple vector scaling multiplication example. 172

7.1 Kernel execution times in ms (best over twenty runs) and speedups between the Coarse Grained and the Allen and Kennedy parallelization algorithms using CUDA for different Nvidia GPUs. The example used here is the correlation loop nest shown in Figure 4.6 with $m = 3000$. 180
7.2 Impact of work-group size on GT200 and Fermi architectures, speedup normalized with respect to a size of 512 work-items per work-group. The CUDA API is used in this experiment. 182
7.3 Impact of block dimensionality on performance for different block sizess expressed in µs for G80 architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163. . . 183
7.4 Impact of block dimensionality on performance for different block sizes expressed in µs for GT200 architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163. . . 184
7.5 Impact of block dimensionality on performance for different block sizes expressed in µs for Fermi architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163. . . 185
7.6 Impact of block dimensionality on performance for different block sizes expressed in µs for Kepler architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163. . . 186
7.7 Impact of block dimensionality on performance for different block sizes expressed in µs for G80 architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129. 187
7.8 Impact of block dimensionality on performance for different block sizes expressed in µs for GT200 architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129. 188

7.9 Impact of block dimensionality on performance for different block sizes expressed in µs for Fermi architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129. 189

7.10 Impact of block dimensionality on performance for different block sizes expressed in µs for Kepler architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129. 190

7.11 Execution times in ms (best over twenty runs) and speedups for scalarization using CUDA and OpenCL for different AMD and Nvidia GPUs. The example is the `matmul` kernel shown in Figure 4.22, page 129, with $n_i = n_j = n_k = 2048$. The execution times presented here are kernel execution times. The GTX 8800 results are given aside because they are one order of magnitude slower. This architecture does not perform double precision computations: doubles are rounded to float before being processed. Hence, no OpenCL results are available in double precision for this GPU. 192

7.12 Execution times in μs (best over twenty runs) and speedups for scalarization using CUDA and OpenCL for different AMD and Nvidia GPUs. The code is the Scilab script after conversion to C and loop fusion shown in Figure 4.21b, page 129. The execution times shown are the kernel execution times. The GTX 8800 results are given aside because they are one order of magnitude slower. This architecture does not perform double precision computations: doubles are rounded to float before being processed. Hence, no OpenCL results are available in double precision for this GPU. 193

7.13 Execution times in μs (best over twenty runs) for the code shown in Figure 4.23c and Figure 4.23d. The value of `M` is 8192 and the values of `N` are on the x axis. The Nvidia board shows clearly a shift on the red curve for $N = 32$ corresponding to the the warp size. 194

7.14 Execution time in µs (best over twenty runs) and speedup for different unrolling factor using CUDA and OpenCL for different AMD and Nvidia GPUs. The execution times are the kernel execution times. Single precision floating point is used. The example is the code presented in Figure 4.22, page 129, used in the previous section. Loop unrolling is applied after scalarization to obtain the code shown in Figure 4.24, page 133. The GTX 8800 results are given aside because they are one order of magnitude slower. . . 196

7.15 Register counts for different Nvidia architectures and different unrolling factors. . . . 197

7.16 Kernel execution times in ms (best over twenty runs) and speedups for the array linearization transformation and different Nvidia GPUs, and with and without the scalarization illustrated in Figure 7.11. The example is the `matmul` kernel shown in Figure 4.22, page 129, with $n_i = n_j = n_k = 2048$. . 198

7.17 Execution times and speedups for versions of hotspot on GPU, with different iteration counts. . . . 199

7.18 Illustration for the code used to measure performance of the static approach on a Jacobi 2D scheme. . . . 201

7.19 Illustration for the code used to measure performance for the StarPU version of the Jacobi 2D scheme. . . . 202

7.20 `durbin` example from Polybench that shows the interest of sequential promotion. . . . 203

7.21 Execution times in ms and speedups for CUDA execution with communication optimization, using the classic scheme and the sequential promotion. The result are based on the average over five runs for `durbin` and `gramschmidt` examples (see Figures 7.20 and 3.17). . . . 204

7.22 Speedup relative to naive sequential version for an OpenMP version on the CPU, a version with basic PGI Accelerator and HMPP directives, a naive CUDA version, and an optimized CUDA version, all automatically generated from the naive sequential code. . . . 205

7.23 `bicg` example from Polybench that shows the impact of the different StarPU schedulers on a sequential example. There is a direct dependence between each of the kernels. Here $nx = ny = 8000$, thus the `kernel3()` is executed 8000 times. . . . 207

7.24 Average execution time in ms over ten runs for `3mm` and `bicg` examples. Note the impact of different StarPU schedulers, the default greedy one and the data-aware, *dmda*. . . . 207

7.25 The vector scaling example presented in Figure 6.4, modified to increase the computation time. . . . 208

7.26 Output from Nvidia Visual Profiler showing the communications in brown and the kernel executions in blue. The mapping is done using one C2070 only. The copy-out do not overlap properly with copy-in, which is unexpected and limits the acceleration that can be achieved. 209
7.27 Output from Nvidia Visual Profiler showing the communications in brown and the kernel executions in blue. The mapping is done using two C2070s. The copy-out do not overlap properly with copy-in, which is unexpected and limits the acceleration that can be achieved. 210

8.1 Overview of the global compilation scheme. 214

List of Tables

4.1 A comparison of some loop parallel algorithms (from data published in [Boulet *et al.* 1998], nonexhaustive). 101

7.1 The different hardware platforms available for the experiments. 179

7.2 Number of memory transfers after parallelization using Par4All naive allocation, using my automatic optimizing scheme, and as a developer would have put it. 200

Chapter 1

Introduction

Contents

1.1 The Prophecy 2
1.2 Motivation 3
1.3 Outline 7

1.1 The Prophecy

Once upon a time, software programmers were merrily writing their code with the simple von Neumann architecture in mind (see in Figure 1.1). Performance was important of course, but they were also protected by a godsend that let them launch a project requiring computing power that was not yet available. Indeed, the time-to-market period was for sure the scene of huge improvement in hardware performance. The prophecy that every programmer was relying on is known as Moore's law. It is commonly quoted as (see [Srinivasan 2012, Manegold 2002, Yang & Chang 2003])

> the CPU clock speed will double every eighteen months.

This short and simple sentence has been immersed in the mind of generations of programmers for decades. Everything was going along fine until a bird of ill omen came and stated

> it cannot continue forever. The nature of exponentials is that you push them out and eventually disaster happens.

He was not the first one to challenge the prophecy, but this time it was Gordon Moore himself [Dubash 2005], the author of the prophecy. It was terrible for the programmers, and most of them locked themselves into denial. Little by little, the idea that the clock speed does not continue to grow as before made its way. As matter of fact, the original prophecy could probably ranked close second on the Top 10 list for misquoted statements, right behind "Luke, I am your father." Actually Moore originally stated [Moore 1965] that

> the complexity for minimum component costs has increased at a rate of roughly a factor of two per year. . . . Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

The forty years of easy life ended, as shown in Figure 1.2, and programmers were about to face a new challenge. In fact, hardware designers, facing the frequency wall, jumped right into the parallel world. [1] The processor frequency was limited and they bypassed this issue

1. Parallelism has been already present in single-core processors since 1989 in the i860 Very Long Instruction Word (VLIW) processor, and later with the Matrix Math eXtension (MMX) instruction set in Pentium. Since then, programmers were offered the possibilities to express fine grained parallelism in

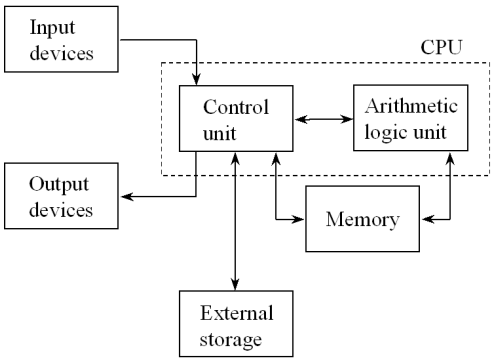


Figure 1.1: von Neumann architecture

by aggregating multiple processors per chip, thus increasing the peak performance of their chips. The multicore era had started.

Programmers discovered a new universe: the execution times of their programs were no longer reduced when a new processor was released. In this new world they had to rethink their algorithms to make use of multiple processors. As if it were not complicated enough, some hardware designers, who probably embraced the dark side of the force, started to introduce some more exotic pieces of hardware. These hardware platforms were highly parallel but very tricky to target. The white knight programmer taking up the challenge had not only to rethink algorithms, but also to manage some complex memory hierarchies for which hardware designers left the management on behalf of the programmer.

Welcome to the heterogeneous computing universe!

1.2 Motivation

"Your free lunch will soon be over." Herb Sutter started his 2005 article [Sutter 2005] with this declaration to software developers. The limits of frequency scaling now forbid automatic performance increase for sequential programs. The future is heterogeneous, from the embedded world of smartphones to the largest supercomputers. Sutter wrote a sequel to this article [Sutter 2011] in which he states quite accurately: "Now welcome to the hardware jungle." Figure 1.3 illustrates this evolution.

the instruction set, with the AMD K6-2 with 3DNow! vector instructions [Bush & Newman 1999] and Streaming SIMD Extension (SSE) since Intel Pentium III [Intel 2008].

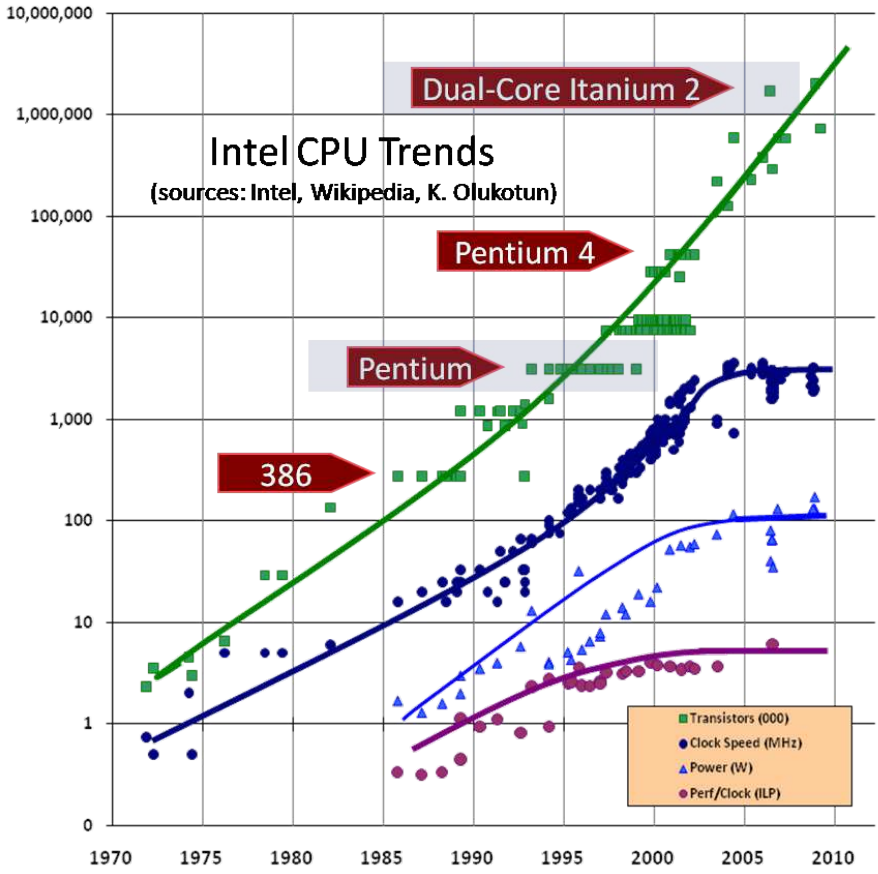


Figure 1.2: More than three decades of prosperity, the misquoted Moore's law (source [Sutter 2005], updated 2009, reprinted here with the kind permission of the author).

In the embedded system world, current high-end smartphones are based on a multi-core processor, and they include vector processing units and also a Graphics Processing Unit (GPU). For instance the A5 processor, used in the Apple iPhone 4S, is a dual-core ARM Cortex-A9 MPCore Central Processing Unit (CPU) together with a dual-core GPU [AnandTech 2011]. The same processor is used in the Apple iPad 2. The latest Tegra 3 processor from Nvidia is a quad-core ARM Cortex-A9 MPCore and a twelve-core GPU [Nvidia 2012b]. In both case, each core includes a 128-bit wide NEON vector unit [Wikipedia 2012a]. The next Tegra generation will support General-Purpose Processing on Graphics Processing Units (GPGPU) computing using Open Computing Language (OpenCL).

In the supercomputing world, parallelism has been present for decades now. Vector ma-

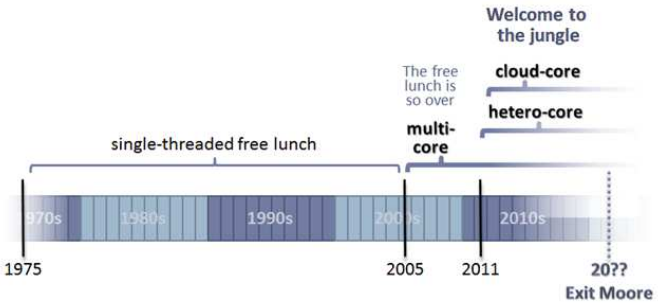


Figure 1.3: The free lunch is over. Now welcome to the hardware jungle (source [Sutter 2011], reprinted here with the kind permission of the author).

chines have been replaced by clusters of multicore multiprocessor systems in the Top500 list [TOP500 Supercomputing Sites 2012]. The new trend is now adding hardware accelerators to these systems, mostly using GPUs, adding a new layer of complexity. The June 2011 Top500 list includes three GPU-based systems in the top five, but there are also five GPU-based systems in the Green500 list [Feng & Cameron 2007] among the ten first entries [The Green500 2011]. The Nvidia Tesla K20 based *Titan* supercomputer trusts currently the last November 2012 list [TOP500 Supercomputing Sites 2012], and it is interesting to note that Intel with its Xeon Phi coprocessor enters at the 7th rank.

One cannot find a single-core Personal Computer (PC) nowadays. Dual-core is the standard at the entry level, quad-core in the mid-end, and it currently goes up to six-core in the high end. Required by the gaming industry, GPUs shipped with PCs are more and more powerful and are used in a growing set of applications beyond their primary usage: 3D rendering and graphic display.

The concern that arises now, as these heterogeneous platforms are widely available, can be summarized as the three P properties [Adve 1993, Benkner *et al.* 2011, Adve 2011] shown in Figure 1.4:

- Performance: the program makes use of the peak capability of the hardware.
- Portability: the code written by the programmer runs on a large range of platforms.
- Programmability: the programmer write his algorithms quickly.

A fourth P can now be added: Power. Not only because our cell phones have small batteries, but also because in 2007 each of the ten biggest supercomputers consumed as much energy as a city of forty thousand people [Feng & Cameron 2007]. People are looking for software that is power aware [Hsu & Feng 2005], using trade-offs between performance

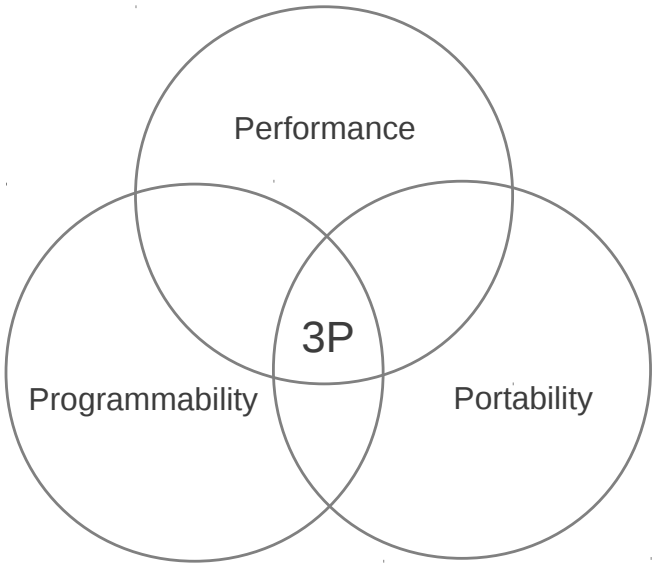


Figure 1.4: The three P properties.

and power consumption.

Solutions to address these properties have been sought for a long time, since clusters of computers entered the market. The programming complexity is increased when very specialized hardware accelerators are added in these machines. Many types of resources must be connected, and it becomes too much of a burden for the developer.

Performance has improved with compilers, allowing new languages to be competitive with the *king* C language, which is still the reference when close control of the hardware is necessary.

On the portability side, it is hardly possible to maintain a huge C code making use of a wide range of heterogeneous devices. A usual practice is then to restrict to a common subset of hardware features, limiting the practical performance one can expect with respect to the theoretical peak performance depending on the application.

Finally, the programmability has been largely addressed by Application Programming Interface (API) providers and language designers. For instance, UPC [UPC Consortium 2005], Co-Array Fortran [ISO 2010], or Titanium [Yelick *et al.* 1998] exploit the Partitioned Global Address Space (PGAS) model. The global memory address space is logically partitioned and physically distributed among processors [Coarfa *et al.* 2005]. The locality of references is then exploited by the runtime system with strategies like the owner

computes rule. The purpose of these languages is to let the programmer ignore the remote memory accesses, which leads to simpler code. This simple flat model has then evolved to Asynchronous Partitioned Global Address Space (APGAS) with the X10 [Ebcioğlu *et al.* 2004] or the Chapel [Chamberlain *et al.* 2007] languages. Concurrency has been made explicit and the programmers express asynchronous constructions on multiple levels. While the programmers have to change their usual approach to express their algorithms, these languages provide high-level abstractions of architecture in a layered manner. However, these languages are new and not widely adopted by developers. Criticisms about performance has been expressed: the code has to be optimized only with a good knowledge of the target architecture [Zhang *et al.* 2011a].

The recent OpenCL standard [Khronos OpenCL Working Group 2008, Khronos OpenCL Working Group 2011] has been developed to program accelerators. It provides an abstraction of the hardware, based on an API to manage the device, and a language derived from a subset of C to write *kernels*, i.e., functions to be executed on an accelerator. This standard provides some portability across vendors and programmability at the C level. However, performance portability is difficult to achieve [Komatsu *et al.* 2010]. Another approach is directive-based languages, following the well-known OpenMP standard [OpenMP Architecture Review Board 1997, OpenMP Architecture Review Board 2011] for shared memory systems. For example, some sets of directives like Hybrid Multicore Parallel Programming (HMPP) [Wikipedia 2012c], PGI Accelerator [Wolfe 2010], or more recently OpenACC [OpenACC Consortium 2012] provide an easier way to program accelerators, while preserving code portability.

1.3 Outline

The goal of this dissertation is to explore the potential of compilers to provide a solution to the three *P*s: Performance, Portability, and Programmability. The solution considered is the automatic code transformation of plain C or Fortran sequential code to accelerator-enabled equivalent code. The main target machines are accelerators like GPUs: massively parallel, with embedded memories in the GB range. A source-to-source approach takes advantage of the Compute Unified Device Architecture (CUDA) and the standard OpenCL APIs. Programmability and portability are enforced by the fully automatic approach. Numerous measurements are provided to show that performance is not sacrificed.

The approach is pragmatic and the ideas and schemes presented are implemented in a

new automatic source-to-source compiler, Par4All [SILKAN 2010 (perso)], and validated using benchmarks. The main goal is to provide a full end-to-end compilation chain, from the sequential code to the GPU-enabled binary, good enough as a prototype for an industrial solution. Therefore, instead of being deeply focused on a limited part of the problem, this work contributes to different aspects of the problem and attempts to explore and solve all the issues raised when building such a full compilation chain.

This compiler approach is useful for legacy applications and new developments as well. A compiler lowers the entry cost but also the exit cost when a new platform has to be targeted. Debugging and maintenance are easier since the code is written with a sequential semantics that is suitable for most programmers. When the compiled code is not executing fast enough, some specific costly parts of the code, the *hot spots*, can be manually optimized for a particular architecture: a source-to-source compiler makes manual optimizations possible on the code after processing by the heterogeneous compiler.

The choice of the C and Fortran languages is driven by their broad use in the high-performance community. C is also a common choice for other tools that generate code from a high-level representation or a scripting language. In order to illustrate the interest of this approach, examples of Scilab [Scilab Consortium 2003] code are included. They are automatically converted to sequential C with a Scilab compiler, and then transformed to exploit accelerators using the different methods presented in this dissertation.

I present the history of GPUs and the emergence of GPGPU in Chapter 2. The hardware evolution is mirrored by the associated programming languages that all failed to match the three *P*s criteria. I introduce he architectures of GPUs and their evolutions to show the constraints that should be met by a compiler to achieve performance: distributed memory, memory access patterns on GPUs, fine grained parallelism, and support for atomic operations.

In Chapter 3, I explore solutions to the automatic distribution of the data onto the CPU and accelerator memories. The convex array region abstract representation is first presented. A simple process to generate communications based on array regions is then explained. I propose a new interprocedural optimizing scheme, and I validate it using experiments. The algorithm relies on a new static analysis, *Kernel Data Mapping*, and minimizes the communications by preserving data on the GPU memory and avoiding redundant communications.

I identify a collection of program and loop transformations to isolate and optimize GPU codes in Chapter 4. I propose a flexible mapping of parallel loop nests on the different layers

of GPUs. I designed and implemented a transformation to substitute induction variables and enable further parallelization. I present two different loop parallelization algorithms and the consequences on code generation and performance. I modified them to handle reduction schemes and introduced *Coarse Grained with Reductions* (CGR). I designed and implemented a new transformation to benefit from hardware atomic operations when parallelizing loop nests with reductions. I designed and implemented a new loop fusion scheme, and I proposed heuristics to drive loop fusion to fit the GPUs' constraints. I present Three different scalarization schemes. I modified the existing transformation to provide better performance on GPUs. I also present the impact of loop unrolling and array linearization. I validated all these transformations with measurements.

I present the whole compilation process in Chapter 5, from the sequential source code to the final binary and the runtime associated at execution. The flexibility of a programmable pass manager is used to produce the compilation chain. Interprocedural analyses are used, and they require processing the source code of all functions in the call graph. It is an issue for external libraries. I defined a dynamic solution to feed the compiler on demand during the process.

I explore perspectives about extensions for multiple GPUs in Chapter 6. I study two different schemes to extract parallelism. I implemented a simple task parallelism extraction, and modified the existing symbolic tiling transformation. The StarPU runtime library is used to exploit task parallelism and schedule tasks on multiple GPUs.

I present all experimental results in Chapter 7 to validate the solutions defined in the previous chapters. I extracted twenty test cases from Polybench and Rodinia test suites. I also used a real numerical n-body simulation to show that speedups can be obtained automatically on application larger than the kernel benchmarks. Several target GPU boards from Nvidia and Advanced Micro Devices (AMD) are used to show how the impact of program transformations on performance depends on architectures.

Due to the variety of subjects tackled in this work, the presentation of the related works is included in each chapter.

Finally, to pay a tribute to the environment in which this work takes place, a summary in French is provided for each chapter at the end of the thesis.

Chapter 2

General-Purpose Processing on GPU : History and Context

Contents

2.1 History . . . 12
2.2 Languages, Frameworks, and Programming Models . . . 14
2.2.1 Open Graphics Library (OpenGL) . . . 15
2.2.2 Shaders . . . 15
2.2.3 Brook and BrookGPU . . . 17
2.2.4 Nvidia Compute Unified Device Architecture (CUDA) . . . 18
2.2.5 AMD Accelerated Parallel Processing, *FireStream* . . . 20
2.2.6 Open Computing Language (OpenCL) . . . 20
2.2.7 Microsoft DirectCompute . . . 21
2.2.8 C++ Accelerated Massive Parallelism (AMP) . . . 21
2.2.9 ΣC and the MPPA Accelerator . . . 23
2.2.10 Directive-Based Language and Frameworks . . . 23
2.2.11 Automatic Parallelization for GPGPU . . . 30
2.3 Focus on OpenCL . . . 30
2.3.1 Introduction . . . 31
2.3.2 OpenCL Architecture . . . 31
2.3.3 OpenCL Language . . . 38
2.4 Target Architectures . . . 39
2.4.1 From Specialized Hardware to a Massively Parallel Device . . . 40
2.4.2 Building a GPU . . . 40
2.4.3 Hardware Atomic Operations . . . 42
2.4.4 AMD, from R300 to Graphics Core Next . . . 43
2.4.5 Nvidia Computing Unified Device Architecture, from G80 to Kepler 48
2.4.6 Impact on Code Generation . . . 52
2.4.7 Summary . . . 54
2.5 Conclusion . . . 55

The reign of the classical Central Processing Unit (CPU) is no longer hegemonic and the computing world is now heterogeneous. The Graphics Processing Units (GPUs) have been candidate as CPUs co-processors for more than a decade now. Other architectures were also developed like the Intel Larabee [Seiler *et al.* 2008], which never really reached the market as GPU and was released recently as a co-processor under the name Xeon Phi [1] by the end of 2012, and the IBM and Sony Cell [Hofstee 2005], which was used in the Sony PlayStation 3. However, although many researchers have tried to map efficient algorithms on its complex architecture, it was discontinued. This failure resulted from its difficult programming and memory models, especially facing the emergence of alternatives in the industry: the GPU manufacturers entered the general computing market.

Dedicated graphic hardware units offer, generally via their drivers, access to a standard Application Programming Interface (API) such as OpenGL [Khronos OpenGL Working Group 1994, Khronos OpenGL Working Group 2012] and DirectX [Microsoft 1995, Microsoft 2012]. These APIs are specific to graphic processing, the main application domain for this kind of hardware. Graphic processing makes use of many vector operations, and GPUs can multiply a vector by a scalar in one operation. This capability has been hijacked from graphic processing toward general-purpose computations.

This chapter first presents in Section 2.1 the history of the general-purpose computing using GPUs, then Section 2.2 gives insights about the evolution of the programming model and the different initiatives taken to pave the way to General-Purpose Processing on Graphics Processing Units (GPGPU). The OpenCL standard is introduced with more details in Section 2.3. The contemporary GPU architectures are presented in Section 2.4. Finally I list the many programming challenges these architectures offer to programmers and compiler designers.

2.1 History

The use of graphic hardware for general-purpose computing has been a research domain for more than twenty years. Harris et al. proposed [Harris *et al.* 2002] a history starting with a machine like the Ikonas [England 1978], the Pixel Machine [Potmesil & Hoffert 1989], and Pixel-Planes 5 [Rhoades *et al.* 1992]. In 2000, Trendall and Stewart [Trendall & Stewart 2000] gave an overview of the past experiments with graphics hardware. Lengyel

1. It was also previously known under the codename *Many Integrated Core* (MIC), *Knights Ferry*, or *Knight Corner*.

et al. [Lengyel *et al.* 1990] performed real-time robot motion planning using rasterizing capabilities of graphics hardware. Bohn [Bohn 1998] interprets a rectangle of pixels as a four-dimensional vector function, to do computation on a Kohonen feature map. Hoff et al. [Hoff *et al.* 1999] describe how to compute Voronoi diagrams using z-buffers. Kedem et al. [Kedem & Ishihara 1999] use the PixelFlow SIMD graphics computer [Eyles *et al.* 1997] to decrypt Unix passwords. Finally some raytracing was performed on GPU in [Carr *et al.* 2002] and [Purcell *et al.* 2002]. A survey of GPGPU computation can be found in [Owens *et al.* 2007].

Until 2007, the GPUs exposed a graphic pipeline through the OpenGL API. All the *élégance* of this research rested in the mapping of general mathematical computations on this pipeline [Trendall & Stewart 2000]. A key limitation was that, at that time, GPU hardware offered only single-precision floating point units, although double precision floating point is often required for engineering and most scientific simulations.

GPUs have spread during the last decades, with an excellent cost/performance ratio that led to a trend in experimental research to use these specialized pieces of hardware. This trend was mirrored first with the evolution of the programming interface. Both OpenGL and DirectX introduced shaders (see Section 2.2.2) in 2001, and thus added programmability and flexibility to the graphic pipeline. However, using one of the graphic APIs was still mandatory and therefore General-Purpose Processing on Graphics Processing Units (GPGPU) was even more challenging than it is currently.

In 2003 Buck et al. [Buck *et al.* 2004] implemented a subset of the Brook streaming language to program GPUs. This new language, called BrookGPU, does not expose at all the graphic pipeline. The code is compiled toward DirectX and OpenGL. BrookGPU is used for instance in the Folding@home project [Pande lab Stanford University 2012]. More insight about Brook and BrookGPU is given in Section 2.2.3.

Ian Buck, who designed Brook and BrookGPU, has joined Nvidia to design the Compute Unified Device Architecture (CUDA) language, which shares similarities with BrookGPU. However, while BrookGPU is generic, CUDA API is specific to Nvidia and its then new scalar GPU architecture introduced with CUDA is presented in Section 2.4.5. CUDA is an API and a language to program GPUs more easily. The graphic pipeline does not exist anymore as such and the architecture is unified and exposed as multi-Single Instruction stream, Multiple Data streams (SIMD)-like processors. CUDA is introduced with more details in Section 2.2.4.

From 2004 to 2012, the evolution of GPUs' floating point performance increased much

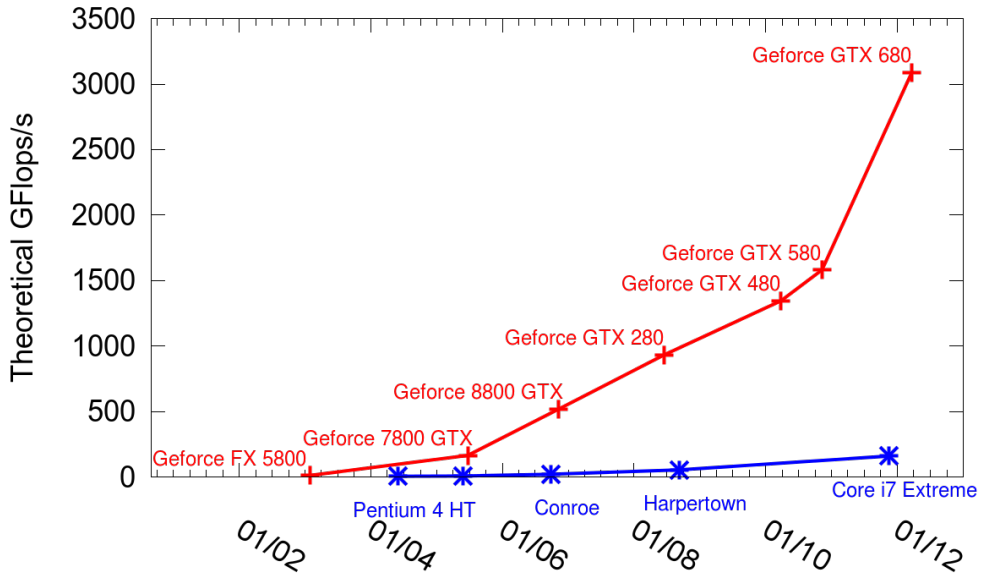


Figure 2.1: Performance evolution for single-precision floating point computation, for both Nvidia GPUs and Intel CPUs between 2003 and 2012, computed from vendors' datasheets.

faster than the CPUs' performance, as shown in Figure 2.1. The programmability offered by CUDA, combined with the GPU performance advantage, has made the GPGPU more and more popular for scientific computing during the past five years.

The increased interest in GPGPU attracted more attention and led to the standardization of a dedicated API and language to program accelerators: the Open Computing Language known as OpenCL (see Section 2.3).

Others programming models are emerging, such as directive-based languages. These let the programmers write portable, maintainable, and hopefully efficient code. Pragma-like directives are added to a sequential code to tell the compiler which pieces of code should be executed on accelerator. This method is less intrusive but may provide limited performance currently. Several sets of directives are presented in Section 2.2.10.

2.2 Languages, Frameworks, and Programming Models

The programming language history includes many languages, frameworks, and programming models that have been designed to program accelerators. Some were designed for the initial purpose of the accelerator, i.e., graphic computing, and were later diverted to-

ward general-purpose computation. Others were designed entirely from scratch to address GPGPU needs.

This section surveys the major contributions, approaches, and paradigms involved during the last decade to program hardware accelerators in general-purpose computations.

2.2.1 Open Graphics Library (OpenGL)

Open Graphics Library (OpenGL) is a specification for a multiplatform API that was developed in 1992 by Silicon Graphics Inc. It is used to program software that make use of 3D or 2D graphic processing and provides an abstraction of the different graphic units, hiding the complexities of interfacing with different 3D accelerators. OpenGL manipulates objects such as points, lines and polygons, and converts them into pixels via a graphics pipeline, parametrized with the OpenGL state machine.

OpenGL is a procedural API containing low-level primitives that must be used by the programmer to render a scene. OpenGL was designed upon a state machine that mimics the graphic hardwares available at that time. The programmer must have a good knowledge of the graphics pipeline.

OpenGL commands mostly issue objects (points, lines and polygons) to the graphics pipeline, or configure the pipeline stages that process these objects. Basically, each stage of the pipeline performs a fixed function and is configurable only within tight limits. But since OpenGL 2.0 [Khronos OpenGL Working Group 2004] and the introduction of shaders and the OpenGL Shading Language (GLSL) language, several stages are now fully programmable.

In august 2012, the version 4.3 is announced with a new feature: the possibility of executing *compute shaders* such as the `saxpy` example shown in Figure 2.2 without using the full OpenGL state machine. The shader program is executed by every single threads in parallel. Then conducting the same operation over a vector, which usually exhibits a loop, involves here an implicit iteration space. Figure 2.2 illustrates this execution model with one thread per iteration. An classic CPU version of `saxpy` is shown in Figure 2.4a.

2.2.2 Shaders

Shaders are small programs used in graphics processing to operate at a specific stage of the pipeline. They are used to describe light absorption and diffusion, the textures to apply, reflections and refractions, shadowing, moving primitives, or some other post-processing

```
#version 430
// Thread are grouped by "workgroups" of 256
layout(local_size_x=256) in;

// Operate on two buffers and using a global variable
buffer xBuffer { float x[]; };
buffer yBuffer { float y[]; };
uniform float alpha;

// The "main()" is executed by every single thread
void main() {
  // "i" gets the unique thread id
  int i = int(gl_GlobalInvocationID.x);
  // derive size from buffer bound
  if (i < x.length())
    y[i] = alpha*x[i] + y[i];
}
```

Figure 2.2: Example of a `saxpy` OpenGL 4.4 compute shader (adapted from [Kilgard 2012]).

effects. The rendering process makes the shaders perfect candidates for parallel execution on vector graphic processors, relieving the CPU and producing the result faster. Three types of shaders exist:

- Vertex shaders are executed on each vertex given to the GPU. The purpose is to transform each 3D position in the virtual space into the 2D coordinates on the target display, and a depth value for the Z-buffer. The vertex shaders can manipulate properties like position, color, and texture coordinates, but cannot spawn new vertices. The vertex shader output is transferred to the next graphic pipeline stage, a geometry shader if any, or directly to the rasterizer.

- Geometry shaders are able to add or remove vertices of a lattice and their output is sent to the rasterizer for the rendering of the final graphic picture.

- Pixel shaders, also known as fragment shaders, compute the color of each pixel individually. The input comes from the rasterizer, which fills the polygons sent in the pipeline. Pixel shaders are typically used for lighting and associated effects like bump mapping and color tone adjustment. Pixel shaders are often called many times per pixel on the display, one for each object, even if it is hidden. The Z-buffer is later used to sort objects and display only visible parts.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main() {
  for(int i = 0; i < gl_VerticesIn; ++i) {
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_PositionIn[i];
    EmitVertex();
  }
}
```

Figure 2.3: Example of a trivial *pass-through* GLSL geometry shader, which emits a vertex directly for each input vertex (source wikipedia [Wikipedia 2012b]).

Shaders are flexible and efficient. Complicated surfaces can be rendered from a simple geometry. For instance a shader can be used to generate a tiled floor from a plane description.

Initially languages close to assembly, shaders became more popular in 2001 with the definition of higher level languages and their adoption as extensions in OpenGL and DirectX. Shaders made it easier to use GPUs for a wider kind of algorithms. They are close to C and implicitly run in a parallel way on the GPU, but if they add flexibility and programmability to the graphic pipeline for general-purpose computation, they do not provide the programmer with a way to abstract the graphic APIs. Figure 2.3 contains an example of a simple *pass-through* GLSL geometry shader.

2.2.3 Brook and BrookGPU

Brook is a direct successor of the Stanford Merrimac project [Dally *et al.* 2003]. The goal of this project was to take advantage of a new compute model called *streaming*. This model offers two main advantages over classical languages:

- Data parallelism: Brook lets the programmer specify how to apply the same operation to different pieces of array elements.
- Arithmetic intensity: the programmer is encouraged to execute operations on data that minimize communications and maximize local computation.

The Merrimac project aimed at offering better performance than distributed memory [Project 2003], but using the same technology. A language is designed to take parallel processing concepts into a familiar and efficient language, using the streaming model.

```
void main (void) {
 float a;
 float4 X[100],
        Y[100],
        Result[100];

 // ... initialize a, b, and c.

 for (i=0; i<100; i++) {
   Result[i] = a*X[i]+Y[i];
 }
}
```

(a) Classical C code.

```
kernel void
saxpy (float a,
       float4 x<>,
       float4 y<>,
       out float4 result<>) {
 result = a*x + y;
}

void main (void) {
 float a;
 float4 X[100],Y[100],Result[100];
 float4 x<100>,y<100>,result<100>;
 ... initialize a, X, Y ...
 streamRead(x, X);
 //copy data from mem to stream
 streamRead(y, Y);
 //execute kernel on all elements
 saxpy(a, x, y, result);
 //copy data from stream to mem
 streamWrite(result, Result);
}
```

(b) Using Brook streaming kernel.

Figure 2.4: Example of a simple `saxpy` using BrookGPU (taken from [Buck *et al.* 2004]).

Brook is designed as a simple extension of ANSI C.

Until 2003, the only way to benefit from graphics hardware resources was the general APIs OpenGL and DirectX, and the shader programming. BrookGPU [Buck *et al.* 2004] implements a subset from the Brook specification [Buck 2003] to target GPUs. It allows compiling the same code in different target languages, OpenGL and DirectX of course, but also Nvidia Cg shaders and later the generalist Advanced Micro Devices (AMD) Close To Metal (CTM) API. BrookGPU was used for instance in the *Folding@home* project. Figure 2.4 illustrates a simple SAXPY operation using BrookGPU.

2.2.4 Nvidia Compute Unified Device Architecture (CUDA)

Nvidia hired Ian Buck, the main author of Brook and BrookGPU, to design CUDA. Thus there are similarities between CUDA and BrookGPU. However, BrookGPU is generic and has different back ends while CUDA exhibits features specific to Nvidia GPUs. CUDA offers features and low-level tuning unavailable in a portable and generic language such as

BrookGPU. CUDA removes also many limitations found in Brook, such as the memory model, which is quite rigid in Brook. Indeed it requires the programmers to map their algorithm around a fairly limited memory access pattern [Buck 2009].

CUDA technology was published by Nvidia in February 2007. It is a set of components shipped by Nvidia to program their GPUs: a driver, a runtime, libraries (BLAS, FFT, ...), a language based on a extension to a C++ subset, and an API that exhibits an abstraction model for the architecture.

The code that runs on the GPU is written in a C-like form and allows direct random accesses to the GPU memory. The CUDA API is high level and abstracts the hardware. However, to obtain a good percentage of the peak performance, the code must be tuned with a good knowledge of the underlying architecture. CUDA allows the programmers to bypass the compiler and to write directly code in Parallel Thread eXecution (PTX), a pseudo-assembly SIMD language that exhibits an infinite number of registers. The PTX is Just In Time (JIT) compiled by the CUDA driver for a given GPU using its own Instruction Set Architecture (ISA). This allows Nvidia to evolve their architecture while being backward compatible, thanks to the JIT compilation capability of the driver.

CUDA has many advantages over classic GPGPU schemes using the OpenGL API for instance:

- Use of the C language (with extensions) instead of the classical graphic API: a kernel is close to a function call.
- Possibility for sparse memory writes: the code can access a single address in memory.
- Threads can share up to 48 kB of local memory, that is nearly as fast as registers.
- Memory transfers between host and GPU are faster using page-locked memory.
- The instruction set is more extensive, for instance integer and bitwise operations and double precision computation are supported.

However, CUDA exhibits also some limits when compared to classic CPU programming:

- Texture rendering is supported in a limited way.
- Only the most recent architectures support function calls.
- The IEEE 754 floating point standard is not fully implemented.
- Threads execute by groups of thirty-two in a SIMD fashion, such a group is denoted *warp* by Nvidia. Branches do not impact performance significantly as long as all thirty-two *threads* in a group take the same path.

- GPUs compatible with CUDA are exclusively produced by Nvidia.

Nvidia has shipped dedicated boards for GPGPU: the Tesla series. These GPUs boards do not always have any display port and therefore can be used only for intensive compute processing. Usually Tesla boards provide dedicated features such as Error-correcting code (ECC) memory, larger memory sizes, and higher double precision peak performances.

2.2.5 AMD Accelerated Parallel Processing, *FireStream*

FireStream is the AMD GPGPU solution. The name refers to both the hardware and the software shipped by AMD. The hardware was released in 2006 under the name AMD Stream Processor. AMD claims that it was the industry's first commercially available hardware stream processing solution [Advanced Micro Devices 2006]. AMD introduced at the same time their own GPGPU API: Close To Metal (CTM). This API is very close to the hardware as it gives developers direct access to the native instruction set and memory, but the trade-off that arises when choosing a very low level API and language is the usual one: it raises the effort required from the programmer. AMD soon after proposed a new solution called Stream Computing Software Development Kit (SDK). It is a complete SDK and a compiler for Brook+, a high-level language based on Brook (see Section 2.2.3). At the same time they renamed CTM as Compute Abstraction Layer (CAL) [2], which is the target API for Brook+. CAL provides the API to control the device (open, close, managing context, transfer data from or to the device, . . .). It comes with the language CAL Intermediate Language (IL), an intermediate assembly-like language for AMD GPUs. IL is then compiled for the target ISA using the CAL API.

The latest version of AMD's technology is now called Accelerated Parallel Processing (APP) and is based upon Open Computing Language (OpenCL). The support for Brook+ and CTM has been discontinued, and CAL API is now deprecated in favor of OpenCL. The IL language is still the target language for the OpenCL compiler.

The *FireStream* GPU series, just as the Nvidia Tesla series, does not always provide any graphic output, and is intended to be a pure GPGPU solution.

2.2.6 Open Computing Language (OpenCL)

OpenCL is a software stack designed to write programs portable over a wide range of platforms like CPUs, GPUs, Field Programmable Gate Array (FPGA) or other em-

2. AMD CAL is unrelated to the eponymous language from Berkeley

bedded hardwares. It includes a language, based on the C99 standard, to write code for heterogeneous hardwares. It defines an API to manage the dedicated hardware from the host. OpenCL was proposed by Apple to the Khronos Group in 2008 to unify the various frameworks in one standard, which was defined later in the same year [Khronos OpenCL Working Group 2008]. I study OpenCL in detail in Section 2.3

2.2.7 Microsoft DirectCompute

Microsoft proposes its own dedicated GPGPU solution with DirectCompute [Microsoft 2010]. It was released in fall 2009 as part of DirectX 11. The DirectCompute API leverage the High Level Shader Language (HLSL) (same as Nvidia Cg) and provides a solution that bypasses the classical graphic pipeline in favor of a direct access like CUDA or OpenCL. Programmers familiar with HLSL/Cg are then able to transfer buffers directly to or from the GPU, and set shader-like kernels for processing these buffers. Figure 2.5 shows an example of such a shader. The input matrices `d_A` and `d_B` are multiplied into `d_C`, using a straightforward block matrix multiplication algorithm. The three matrices are $size * size$. The `mm` function is executed by $size * size$ number of threads. The scheduler is instructed to group the threads by workgroups of $16 * 16$ number of threads. This virtual organization is mapped on the hardware by ensuring that all threads in a virtual workgroup share some resources, at least till the point where they can be synchronized. The `groupshared` declaration of `local_a` and `local_b` is linked to this thread organization, these arrays are shared by all the threads in a virtual workgroup. The `local_a` and `local_b` array holds the current block of the input matrices during the computation. They are loaded by the threads among a group, and a synchronization enforce that they are fully loaded before each thread perform the multiplication on the blocks using these shared arrays. The shared arrays can be seen as a cache memory that is explicitly managed by the programmer.

2.2.8 C++ Accelerated Massive Parallelism (AMP)

Microsoft C++ Accelerated Massive Parallelism (AMP) is an open specification [Microsoft Corporation 2012a] for enabling data parallelism directly in C++. It was first released in January 2012. It is composed of a C++ language extension, a compiler, a runtime, and a programming model.

The C++ AMP programming model supports multidimensional arrays, indexing, memory transfer, and tiling. Some language extensions control the ways data are moved from

```
cbuffer CB : register(b0)
{
 int size;
};

StructuredBuffer<float> d_A : register(t0);
StructuredBuffer<float> d_B : register(t1);
RWStructuredBuffer<float> d_C : register(u0);

groupshared float local_a[16][16];
groupshared float local_b[16][16];

[numthreads(16, 16, 1)]
void mm(uint3 DTid : SV_DispatchThreadID, uint3 GTid : SV_GroupThreadID)
{
 int row = GTid.y;
 int col = GTid.x;
 float sum = 0.0f;
 for (int i = 0; i < size; i += 16) {
  local_a[row][col] = d_A[DTid.y * size + i + col];
  local_b[row][col] = d_B[(i + row) * size + DTid.x];
  AllMemoryBarrierWithGroupSync();
  for (int k = 0; k < 16; k++) {
   sum += local_a[row][k] * local_b[k][col];
  }
  AllMemoryBarrierWithGroupSync();
 }
 d_C[DTid.y * size + DTid.x] = sum;
}
```

Figure 2.5: Example of a Cg/HLSL shader for DirectCompute (source Microsoft [Deitz 2012]).

the CPU to the GPU and back.

Unlike Direct Compute presented in Section 2.2.7, there is no separation between the code running on the accelerator and the host code. Offloading a computation involves writing a kernel using a lambda function and a dedicated construction to express the iteration set like `parallel_for_each`. Figure 2.6 contains an example of C++ code before and after its conversion to C++ AMP. This example is a simple sum of two arrays. The `concurrency` namespace allows the use of AMP specific constructions and functions, such as `array_view` for example. The code exhibits a call to `discard_data()` on the `array_view` object sum.

This call is intended to hint the runtime so that an initial copy to the accelerator memory is avoided since `sum` does not contain any data.

C++ AMP does not seem to provide a new paradigm, but leverages C++ power and flexibility to provide a more relaxed programming model than Direct Compute or OpenCL. It seems to compete more against a directive-based language such as OpenACC, presented in Section 2.2.10.

2.2.9 ΣC and the MPPA Accelerator

While far from being a new paradigm, process network language may benefit from more consideration in the future. For instance Kalray leverages the ΣC language [Goubier *et al.* 2011] for its (yet unreleased) Multi-Purpose Processor Array (MPPA) accelerator [Kalray 2012]. It integrates a network of 256 Very Long Instruction Word (VLIW) processors, organized in sixteen clusters of sixteen processors, interconnected using a high-bandwidth network-on-chip, but embeds only a few tens of MB of memory. This accelerator leverages low consumption (estimated at around 5 W) when compared to power-hungry GPUs. For example, the Nvidia Tesla C2070 eats up to 238 W.

ΣC is based on the Kahn process network theory [Kahn 1974]. It has been designed to enforce properties like being deadlock-free and provides memory-bounded execution. Formal analysis is leveraged to achieve this goal. The ΣC programming model involves agents as the most basic units. An agent is a stateless independent thread with its own memory space. Agents communicate via First In, First Out (FIFO) queues. Then an application is designed by a set of communicating agents forming a graph. In a ΣC application, the graph is static during all the life of the application, no agent creation or destruction can occur neither any change to the graph topology.

2.2.10 Directive-Based Language and Frameworks

Addressing the programmers' difficulties to write efficient, portable, and maintainable code, as well as the ability to convert progressively existing sequential version toward GPGPU, several initiatives were launched, based on directives inserted in C or Fortran sequential code.

On the basis of the popular Open Multi Processing (OpenMP) standard, Lee et al. propose OpenMP for GPGPU [Lee *et al.* 2009]. They justify the advantages of OpenMP as a programming paradigm for GPGPU as follows:

```cpp
#include <iostream>

const int size = 5;

void StandardMethod() {
 int aCPP[]={1,2,3,4,5};
 int bCPP[]={6,7,8,9,10};
 int sumCPP[size];

 for(int idx=0;idx<5;idx++)
 {
  sumCPP[idx]=
    aCPP[idx]+bCPP[idx];
 }

 for(int idx=0;idx<5;idx++)
 {
  std::cout<<sumCPP[idx]
           <<"\n";
 }
}
```

(a) Pure C++.

```cpp
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
 int aCPP[]={1, 2, 3, 4, 5};
 int bCPP[]={6, 7, 8, 9, 10};
 int sumCPP[size];

 // Create C++ AMP objects.
 array_view<const int,1> a(size,aCPP);
 array_view<const int,1> b(size,bCPP);
 array_view<int, 1> sum(size, sumCPP);
 sum.discard_data();

 parallel_for_each(
  // Define the compute domain, which
  // is the set of threads that are
  // created.
  sum.extent,
  // Define the code to run on each
  // thread on the accelerator.
  [=](index<1> idx) restrict(amp)
 {
  sum[idx] = a[idx] + b[idx];
 }
 );

 // Print the results. The expected
 // output is "7, 9, 11, 13, 15".
 for (int i = 0; i < size; i++) {
  std::cout << sum[i] << "\n";
 }
}
```

(b) Using C++ AMP.

Figure 2.6: Rewriting a C++ computation using C++ AMP. The example shows the use of a lambda function and a `parallel_for_each` construct to express the parallelism (source Microsoft [Microsoft Corporation 2012b]).

- OpenMP is efficient at expressing loop-level parallelism in applications, which is an ideal target for utilizing the highly parallel GPU computing units to accelerate data parallel computations.
- The concept of a master thread and a pool of worker threads in OpenMP's fork-join model represents well the relationship between the master thread running in a host CPU and a pool of threads in a GPU device.
- Incremental parallelization of applications, which is one of OpenMP's features, can add the same benefit to GPGPU programming.

Following the same idea, the OMPCUDA project [Ohshima *et al.* 2010] extended the OMNI OpenMP Compiler to target CUDA.

As OpenMP is designed for shared memory systems, it can be difficult to convert automatically an OpenMP code optimized for CPU into a heterogeneous architecture. Thus other projects bypassed this issue and introduced new directives. Han and Abdelrahman propose with hiCUDA [Han & Abdelrahman 2009] a set of directives to manage data allocation and transfers, and kernel mapping on GPU. The main drawback is that even if it is simpler to write, hiCUDA still requires the programmer to have good knowledge of the target architecture and the way the algorithm maps onto the GPU. It is unclear how the code written this way is portable across architectures. Figure 2.7 shows a sample matrix multiplication using hiCUDA. The directives are tied to a particular architecture: the workgroup size is statically defined, so is the strip-mining width.

Bodin and Bihan propose Hybrid Multicore Parallel Programming (HMPP) [Bodin & Bihan 2009], another set of directives to perform heterogeneous computing. HMPP was then promoted as a standard, Open Hybrid Multicore Parallel Programming (OpenHMPP), in a consortium joining CAPS Entreprise and PathScale. HMPP requires that the code follows some restrictions. The code to be run on an accelerator must be wrapped in a separate function called a *codelet*. Here are the codelet properties [Consortium 2011]:

- It is a pure function.
  - It does not contain static or volatile variable declarations or refer to any global variables unless these have been declared by a HMPP directive "resident."
  - It does not contain any function calls with an invisible body (that cannot be inlined). This includes the use of libraries and system functions such as malloc, printf. . . .

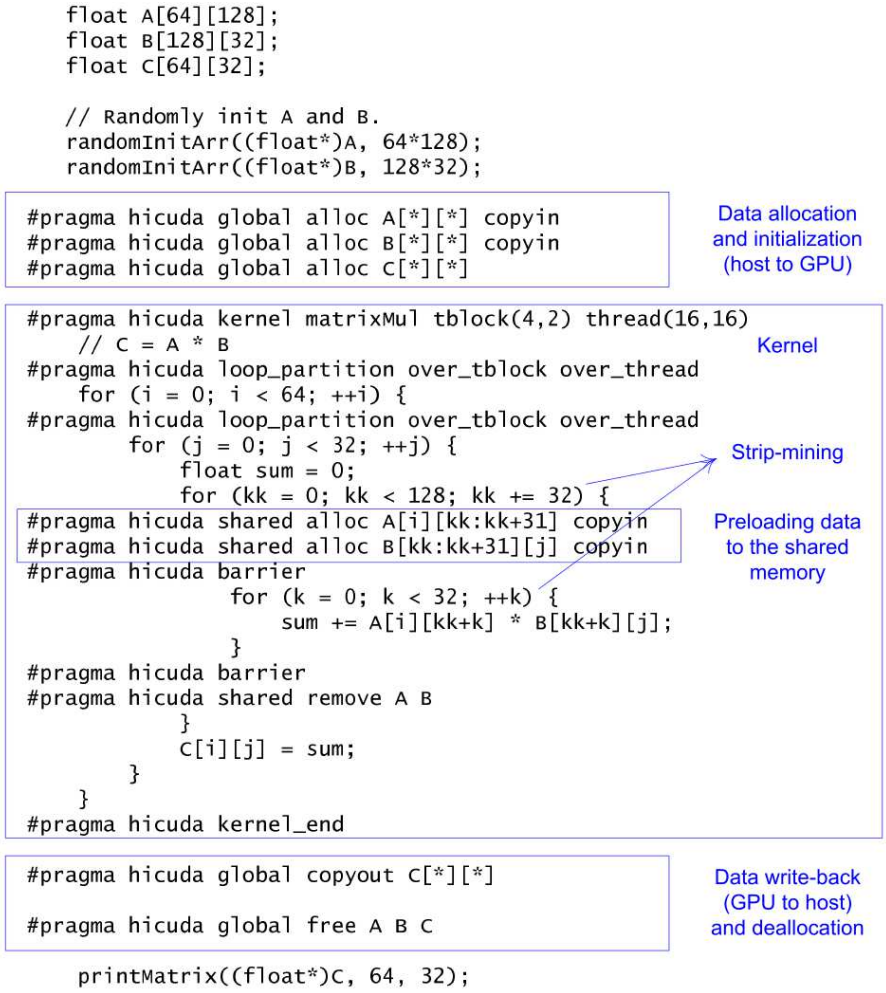


Figure 2.7: A sample matrix multiplication code with hiCUDA directives (source [Han & Abdelrahman 2009]).

◦ Every function call must refer to a static pure function (no function pointers).

- It does not return any value (void function in C or a subroutine in Fortran).
- The number of arguments should be set (i.e., it can not be a variadic function as in stdarg.h in C).
- It is not recursive.
- Its parameters are assumed to be non-aliased.

```
/* declaration of the codelet */
#pragma hmpp simple1 codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm, float inv[sm],
                   float inm[sn][sm], float *outv){
    int i, j;
    for (i = 0 ; i < sm ; i++) {
      float temp = outv[i];
      for (j = 0 ; j < sn ; j++) {
        temp += inv[j] * inm[i][ j];
    }
   outv[i] = temp;
}

int main(int argc, char **argv) {
  int n;
  ........

/* codelet use */
#pragma hmpp simple1 callsite, args[outv].size={n}
matvec(n, m, myinc, inm, myoutv);
  ........
}
```

Figure 2.8: Simple example for HMPP directive-based code writing (source wikipedia [Wikipedia 2012c]).

- It does not contain call site directives (i.e., RPC to another codelet) or other HMPP directives.

HMPP requires less effort from the programmer, and the HMPP compiler can manage automatically to map a given codelet on the GPU, as well as handling the data movement. The compiler can automatically detect the parallelism in a loop nest and take any decision involved in the process of generating the accelerator code. However HMPP offers advanced directive that allows the programmer to tune the compilation process to get better performance. But with the same drawbacks as in hiCUDA: the code is then likely to come tied to a specific target. Figure 2.8 contains a sample code written using HMPP without any specific directive.

PGI introduced the PGI Accelerator [Wolfe 2010], which uses the same idea as HMPP. The proposed directives are written *à la* OpenMP. The code is not outlined in a codelet by the programmer.

The initial PGI Accelerator provided a limited set of directives. The PGI compiler was

```
module globdata
  real, dimension(:), allocatable, device :: x
end module
module globsub
contains
 subroutine sub( y )
  use globdata
  real, dimension(:) :: y
  !$acc reflected(y)
  !$acc region
    do i = 1, ubound(y,1)
      y(i) = y(i) + x(i)
    enddo
  !$acc end region
 end subroutine
end module
subroutine roo( z )
  use globsub
  real :: z(:)
  !$acc data region copy(z)
   call sub( z )
  !$acc end data region
end subroutine
```

Figure 2.9: Example of a PGI Accelerator code using data movement optimization (source PGI Insider [Wolfe 2011]).

supposed to automatically do the conversion work. It was later updated with more possibilities available to the programmer to help the compiler to manage the data movements. HMPP includes also similar directives. Figure 2.9 shows a simple code written using these directives.

In November 2011 at the SuperComputing Conference, Nvidia, Cray, PGI, and CAPS announced that they agreed on a standard for directives: OpenACC. The OpenMP Architecture Review Board CEO Michael Wong declared at this occasion that he looked forward to work within the OpenMP organization to merge OpenACC with other ideas to create a common specification that extends OpenMP to support accelerators. The OpenACC standard [NVIDIA, Cray, PGI, CAPS 2011] seems to be based upon the PGI Accelerator solution: the directives show close similarities.

JCUDA [Yan *et al.* 2009] is a programming interface for Java that allows invoking CUDA kernels. JCUDA defines an extension of Java that needs to be preprocessed to

```
double[][] I_a= new double[NUM1][NUM2];
double[][][] I_aout =  new double[NUM1][NUM2][NUM3];
double[][] I_aex= new double[NUM1][NUM2];

initArray(I_A); initArray(I_aex); // initialize value in array

int[] ThreadsPerBlock = {16, 16, 1};
int[] BlocksPerGrid = new int[3]; BlocksPerGrid[3] = 1;
BlocksPerGrid[0] = (NUM1+ThreadsPerBlock[0]-1)/ThreadsPerBlock[0];
BlocksPerGrid[1] = (NUM2+ThreadsPerBlock[1]-1)/ThreadsPerBlock[1];

/* invoke device on this block/thread grid */
cudafoo.foo1<<<<BlocksPerGrid, ThreadsPerBlock>>>>(I_a,
                                                   I_aout,
                                                   I_aex);
printArray(I_a); printArray(I_aout); printArray(I_aex);

    ......

static lib cudafoo("cfoo","/opt/cudafoo/lib") {
 acc void foo1(IN double[][] a,
               OUT int[][] aout,
               INOUT float[][] aex);
 acc void foo2(IN short[][] a,
               INOUT double[][][] aex,
               IN int total);
}
```

Figure 2.10: A simple JCUDA example. Note the `IN`, `OUT`, and `INOUT` attributes in the kernel declaration that drive automatic memory transfers (source [Yan *et al.* 2009]).

generate the pure Java code and the Java Native Interface (JNI) glue to link against CUDA kernels. The programmers make use of annotation (IN, OUT, INOUT) in front of kernel arguments and the data transfers are managed automatically by JCUDA based only on the annotation, it implies that a mistake from the programmer in an annotation leads to a wrong code. A simple example of Java code invoking a kernel with JCUDA is showed in Figure 2.10. However, useless transfers cannot be avoided in this model: the programmer has no control to preserve data on the accelerator between two kernel calls, while the directive approach offer the possibility to the programmer to manage data movement across the whole program.

2.2.11 Automatic Parallelization for GPGPU

Not much work has been done about the automatic parallelization of a sequential program toward GPUs. Leung et al. [Leung *et al.* 2009] propose an extension to a Java JIT compiler that executes a parallel loop nest on the GPU. The major part of their contributions seems to be the handling of Java exception semantics and Java aliasing at runtime.

Nugteren et al. [Nugteren *et al.* 2011] present a technique to automatically map code on a GPU based on *skeletonization*. This technique is based on a predefined set of skeletons for image processing algorithms. A Skeletonization step recognizes the algorithm's functionalities in the sequential code using techniques like pattern matching, and replaces them with another implementations for the GPU selected from the available predefined implementations.

Reservoir Labs claims that its R-Stream parallelizing C compiler offers automatic parallelization from C code to CUDA since 2010 [Reservoir Labs 2012]. However, R-Stream is proprietary software not freely available and without academic or evaluation licensing, the few academic publications about this work are vague and there is no way to reproduce their claims and results.

CUDA-Chill [Rudy *et al.* 2011] provides automatic program transformation for GPU using the Chill framework for composing high-level loop transformations. However, the recipes have to be adapted to each input program, limiting the applicability and portability obtained.

Baskaran et al. [Baskaran *et al.* 2010] introduce a polyhedral approach to the automatic parallelization, using Pluto [Bondhugula *et al.* 2008c], of affine loop nest from C to CUDA. More recently, the on-going PPCG [Verdoolaege *et al.* 2013] project follows the same path and produces optimized kernels for GPU using the polyhedral model.

2.3 Focus on OpenCL

> Open Computing Language (OpenCL) is an open royalty-free standard for general-purpose parallel programming across CPUs, GPUs and other processors, giving software developers a portable and efficient access to the power of these heterogeneous processing platforms [Khronos OpenCL Working Group 2011].

2.3.1 Introduction

Promoted first by Apple in early 2008, OpenCL was quickly supported by many other vendors such as IBM, Nvidia, AMD, and Intel. It provides a software stack that addresses the challenges of programming heterogeneous parallel processing platforms. The first revision of the standard exhibits a logical model close to the Nvidia CUDA programming model. OpenCL does not limit itself to the dual CPU vs GPU issue, but also takes into account mobile devices up to high-performance computers, as well as desktop computer systems. It can target different kind of accelerators, like multicore CPUs and GPUs, but also more specific devices like Digital Signal Processing (DSP) processors and the Cell processor.

OpenCL API abstracts the hardware at a rather low level. The purpose is to provide high performance by being close-to-metal, and keeping it simple enough for compilers so that the implementation can be easy for a wider range of vendors. OpenCL targets expert programmers who want to write portable and efficient code. Thus it can be seen as the lower level upon which portable libraries, middleware, or software can be built. It also represents a first choice as a backend target for code-generating tools from higher level languages or constructions.

The OpenCL model is split between a host and computing devices in a master-and-slaves fashion. The host manages the devices and acts as a choreographer driving the process using the OpenCL API. On the device side, the code that is to be executed is contained in *kernels*. These kernels are written in a language that is based on a subset of ISO C99 with extensions for parallelism. The OpenCL API lets the host indifferently schedule data parallel kernels or task-based kernels or a combination of both.

2.3.2 OpenCL Architecture

The OpenCL standard is organized into four parts: the platform model (see Section 2.3.2.1), the memory model (see Section 2.3.2.3), the execution model (see Section 2.3.2.2), and the programming model (see Section 2.3.2.4).

The whole OpenCL abstract model is shown in Figures 2.11 and 2.12. The host entry point is the platform. It represents the vendor implementation. The host program sees as many platforms as there are vendor runtimes in the machine. After selecting one or several platforms, the host program can query a list of devices available for this platform. A device is defined in the OpenCL standard as *a collection of compute units. [...] OpenCL devices*

typically correspond to a GPU, a multi-core CPU, and other processors such as DSPs and the Cell/B.E. processor. To manage the devices, the host program has to create one or more contexts. A context is defined as the *environment within which the kernels execute and the domain in which synchronization and memory management is defined. The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects*.

2.3.2.1 Platform Model

OpenCL is strongly based on the concept of one host directly connected to a set of dedicated computing devices. This is the *platform* in OpenCL terminology. The host plays the role of an orchestrator and manages the devices. These can include many compute units, each made up of many processing elements.

For instance, current OpenCL implementations map a multicore CPU as a single device with as many compute units as the number of cores. The number of processing elements per compute units (per core) depends on the vectorizing capabilities of the OpenCL runtime. The Intel OpenCL runtime for instance, reports sixteen processing elements so that the code can self-align on multiples of sixteen and allows faster loads in vector registers. A GPU is shown as a single device, with the number of compute units corresponding to the available hardware.

2.3.2.2 Execution Model

The user program drives the host part of the OpenCL model. It acts as an orchestrator for the kernel part of the program. The host part is responsible for managing contexts of execution for the kernels, initializing the devices, controlling the data movements, and scheduling the execution of the kernels on the devices. To achieve this, it creates at least one context.

Contexts are created and managed using an API defined by the standard. A device can be associated with many contexts, and a single context can manage multiple devices. For a given context, each device has its own command queue. A command queue is the only way for the host to request any data transfer by device, or to launch a kernel.

On the kernel side, the execution model is very close to the CUDA programming model: a huge number of virtual threads are mapped onto real hardware threads using what Nvidia calls in CUDA the Single Instruction stream, Multiple Thread streams (SIMT) paradigm.

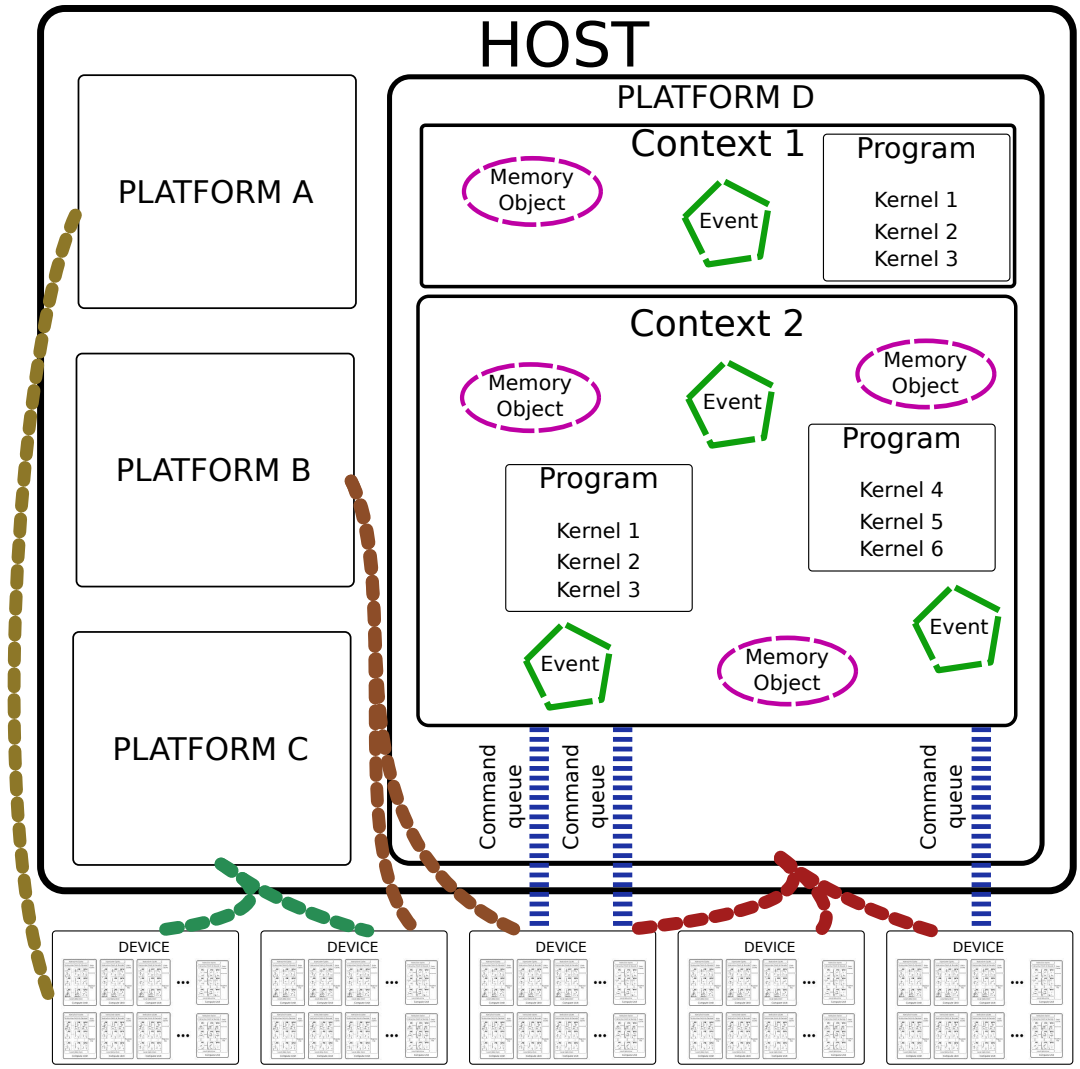


Figure 2.11: Simplified view of the OpenCL abstraction model. A host is connected to multiple devices (GPUs, FPGAs, DPSs, . . .). OpenCL platforms are vendors' implementations that target some types of devices. A context is created for a given platform and a set of devices. Memory objects and events are created context-wise. Devices are then controlled in a given context using command queues. There can be multiple command queues per device, and a device can be associated with queues from multiple contexts and platforms.

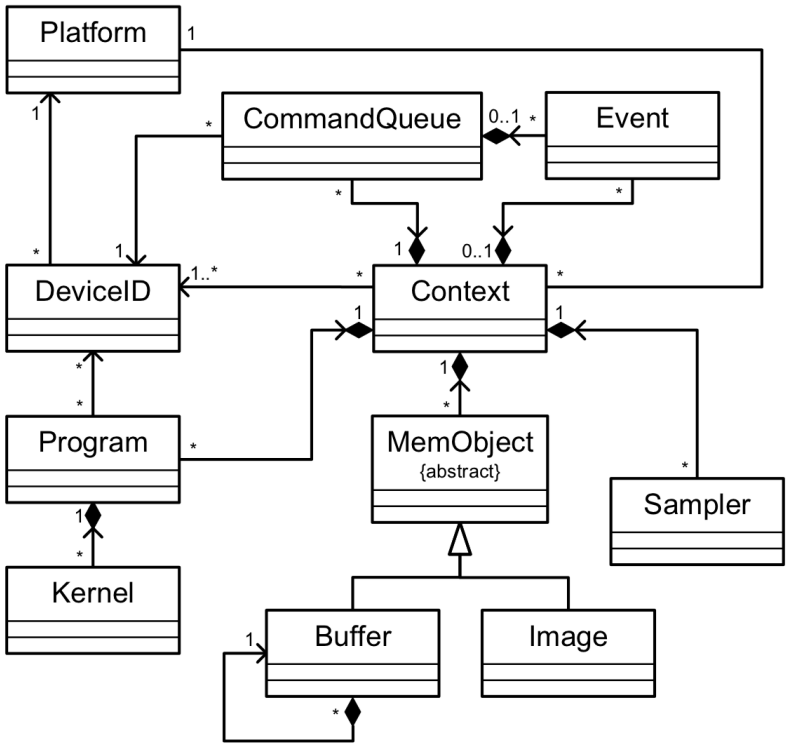


Figure 2.12: UML representation of the OpenCL abstraction model (see Figure 2.11) taken from the Standard [Khronos OpenCL Working Group 2011].

In the OpenCL terminology, the kernel is executed by a number of *work-items*. Each of these work-items has a unique identifier in a global index set named NDRange in OpenCL terminology. This set can have one, two, or three dimensions, and its bounds depend on the OpenCL runtime implementation and the device capability. The unique identifier is then a three-dimensional tuple. It is up to the programmer to exhibit enough data parallelism using a large index set and mapping different work-items to different sets of data.

Work-items are grouped in work-groups. Work-items inside a work-group execute on the same compute unit, using multiple processing elements to achieve parallelism. Synchronization can be performed in a work-group but not across different work-groups. A work-group shares also a dedicated memory space (see Section 2.3.2.3). Work-groups are assigned a unique id in the global NDRange the same way as work-items do.

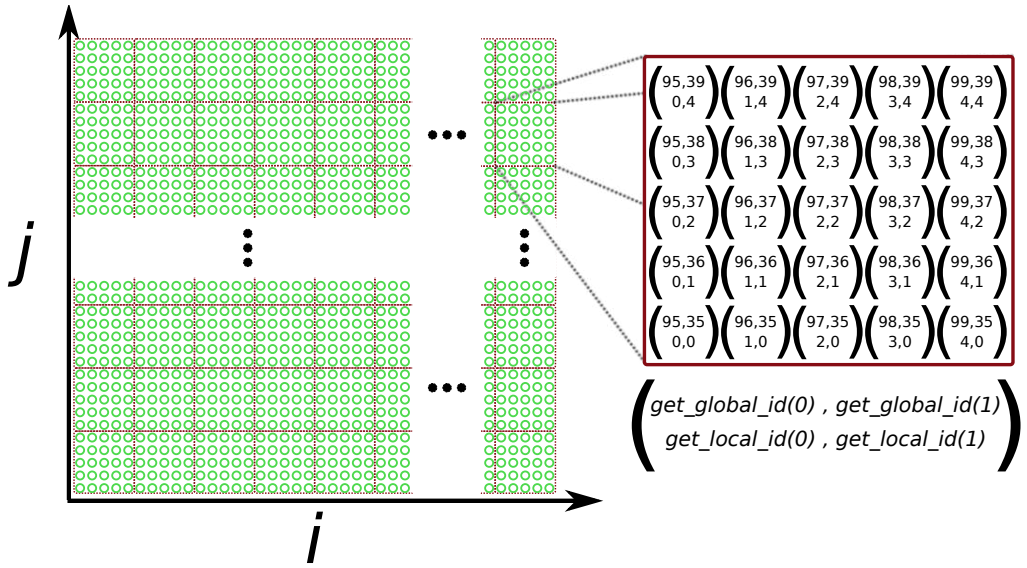
Figure 2.13 shows how a simple two-dimensional parallel loop nest can be mapped onto an OpenCL index set.

```
for(int i=0; i<100; i++) {
 for(int j=0; j<45; j++) {
  // Some parallel
  // computation here
  // ....
 }//
```

(a) Parallel loops

```
__kernel void
my_kernel(/* args list */ ...) {
int i = get_global_id(1);
int j = get_global_id(0);
// Some parallel
// computation here
// ....
//
```

(b) Equivalent OpenCL kernel.



(c) Mapping the iteration set to a NDRange.

Figure 2.13: A mapping example of a two-dimensional loop nest iteration set into an OpenCL index range. The mapping is the simplest possible; one work-item executes one iteration of the original loop nest. The work-group size used as an illustration on figure *c* is a two-dimensional square with an edge of five. Values for `get_global_id()` and `get_local_id()` OpenCL primitives are exhibited for a particular work-group.

2.3.2.3 Memory Model

OpenCL exhibits a hierarchical memory model with four distinct spaces:

- The Global Memory is local to a given device, but shared across all work-items for the device. It is accessible for reading or/and writing, depending on how it is allocated. It can be cached or not depending on the underlying hardware.
- The Constant Memory is a part of global memory accessible read-only from the

kernel. It has to be initialized from the host. It is usually advised to make use of the constant memory for performance reasons. On some devices it is more likely to be cached and optimized for read access.

- The Local Memory is a small dedicated memory for sharing temporary data between work-items in a work-group, to avoid redundant accesses to the global memory. Depending on the underlying hardware, it can be implemented with a very fast on-chip memory or emulated with a portion of the global memory. For instance on Nvidia Fermi architecture, it is nearly as fast as register accesses.
- The Private Memory is a portion of the memory that is private to a work-item and therefore not visible from any other work-item or from the host. It typically maps to registers on modern GPUs, but can also be mapped to global memory by the compiler.

This hierarchy is represented in Figure 2.14. There is no guarantee that all these memory areas are really separated on the hardware. The right part of the figure illustrates this situation. For example, classical multicore CPUs do not exhibit any separated memory space or software managed cache, embedded into each core or not. Then a kernel optimized for the more complex memory architecture on the left may lead to spurious costly memory duplication when using local or private memory on the simpler architecture.

At a more global level, the OpenCL runtime manipulates buffers, i.e., linear areas of memory that the host registers with the runtime before any use as a kernel argument. The host can then write to or read from these memory areas using the OpenCL API, or even directly map the memory area into the host memory space. The physical location of the buffer is undefined by the standard and is implementation specific. From OpenCL version 1.2 on, the programmer can explicitly request a buffer to be moved to a particular device. In any case, before a kernel is launched on a particular device, the OpenCL runtime ensures that the buffers used by the kernel are physically allocated and copied to the device. Therefore it has to keep track of the locations of the buffers and invalidate other copies when a buffer is written by a kernel. The programmer can optimize this management by giving hints at buffer creation times using flags like read-only or write-only. However, these are holding for the whole lifetime of the buffer and thus are not helpful when a buffer is read or written only by some kernels. The `const` qualifier in the kernel declaration arguments can be used as a hint to the runtime to avoid invalidating other copies of a buffer after a kernel execution.

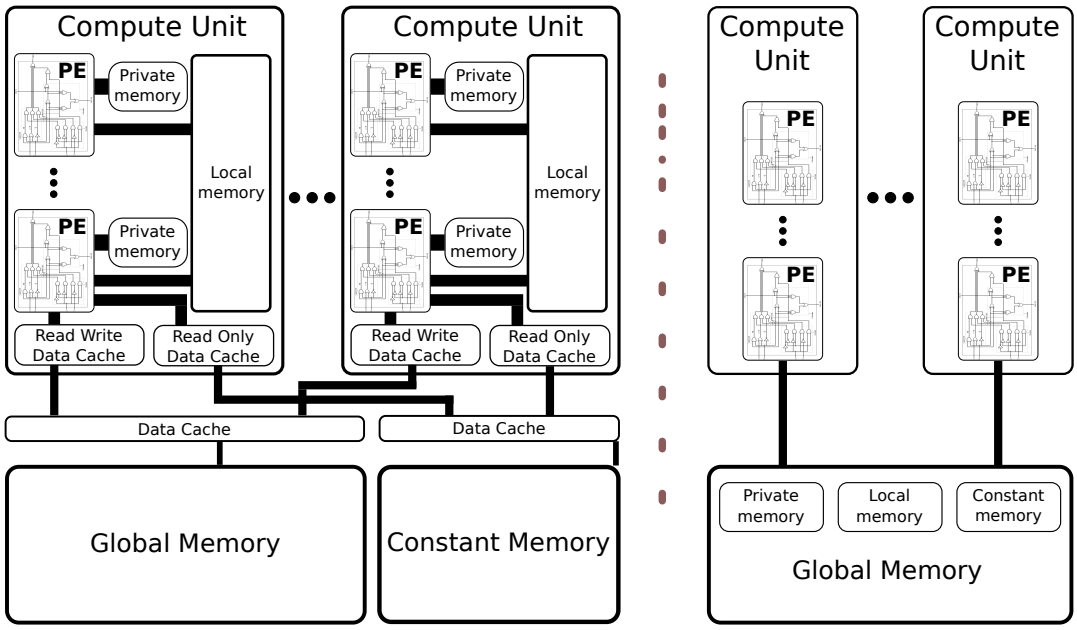


Figure 2.14: Visual example of the OpenCL memory model. Two possible mappings are illustrated: data caches are optional, and private, local, and constant memories are not necessarily dedicated. On the right the simplest mapping, for instance a CPU, merges all memory spaces onto the same piece of hardware.

2.3.2.4 Programming Model

The OpenCL programming model is a mix of the data parallel and task parallel paradigms. The data parallel one is the preferred way to program OpenCL devices like GPUs.

As explained in Section 2.3.2.2, the data parallel model involves a number of work-items that spread over an index set, computing different data in a SIMD/SIMT fashion. The model is relaxed and does not require that each work-item produces one element, and therefore a single work-item can produce as much output as required, or on the other hand only some work-items can produce output. This latter situation occurs when work-items in a work-group work together to produce a single reduced result. Then only one work-item in the work-group is in charge of recording it in the global memory. OpenCL provides full flexibility on this aspect.

The task parallel model is exposed by considering each kernel execution as a task. The parallelism between tasks can then be exploited in two different ways. First, the programmers can issue different tasks to different command queues and thus rely on the

OpenCL runtime to schedule them in parallel. But command queues can also be defined as out-of-order, and then again the OpenCL runtime is free to schedule at the same time as many tasks as submitted to such a queue.

The programmer can issue barriers in the queue to ensure synchronization points, but he can also make use of OpenCL events to enforce dependencies between tasks in a common context, either kernel launches or memory transfers. When a task is submitted to a command queue, a handler on this request is recorded as an OpenCL event. A collection of events can then be used when a new task is submitted in the same context, possibly in a different queue. All events in this collection have to complete before the new task starts.

2.3.3 OpenCL Language

The OpenCL language is a subset of the International Organization for Standardization (ISO) C99 standard. It is used only to create kernels in the OpenCL model.

When compared to plain C, the main differences are the following:

- vector types are natively present, for sizes 2, 3, 4, 8, and 16, and for the native types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and `double`;
- the alignment in memory is always guaranteed to be a multiple of the type size. For instance an `int16` vector is aligned to a 16*`sizeof(int)` boundary;
- shuffle can be written directly in a flexible way, for instance a `double4 a` can be initialized from `double4 b` and `double4 c`: `a = (b.w, c.zyx)`, equivalent to the sequence: `a.x=b.w; a.y=c.z; a.z=c.y; a.w=c.x;`
- keywords are defined for the different memory spaces: `__global`, `__local`, `__constant` and `__private`. Any pointer must make use of one of them so that the memory space to dereference is always known by the compiler;
- a special image object can be declared as `__read_only` or `__write_only` in kernel argument lists:

```
__kernel void foo (__read_only image2d_t imageA,
                   __write_only image2d_t imageB);
```

- the qualifier `__kernel` is used in front of a kernel declaration. Such a function always returns void. It identifies functions that can be used in an NDRange object issued in a command queue;

- variable length arrays and structures with flexible (or unsized) arrays are not supported;
- variadic macros and functions are not supported;
- The library functions defined in the C99 standard headers `assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h` and `wctype.h` are not available and cannot be included;
- recursion is not supported;
- built-in functions are provided to manage work-items, perform asynchronous or atomic memory operations.

2.3.3.1 Conclusion

OpenCL is a standard, which by itself is already a good thing for programmers concerned with portability. However, there are some caveats with OpenCL. The performance portability is not enforced and programmers have to write kernels for a given target. Another issue is programmability: OpenCL API is verbose and is rather designed as a target for libraries, frameworks, or code generators. In this case, OpenCL provides all the control that can be wished. Therefore it is suitable as a target for a source-to-source compiler such as the one proposed in this work.

2.4 Target Architectures

This thesis focuses on hardware accelerators like GPUs. The common characteristics of such accelerators are as follows:

- large embedded memory: over 1 GB;
- high level of parallelism: from a few tens of processing elements, to many thousands, possibly highly threaded;
- compliance with the OpenCL programming model introduced in Section 2.3.

The most widespread matching hardware platforms are manufactured by AMD and Nvidia, and are indeed ubiquitous in modern desktops. This section introduces some GPU architectures starting from a high-level view to a deeper comparison between the two current leading architectures. It also explains how two kinds of parallelism are exploited: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP).

2.4.1 From Specialized Hardware to a Massively Parallel Device

Dedicated graphic circuits were introduced in the 1980s to offload 2D primitives processing from the main CPU. At that time the purpose was to draw simple objects like a line, a rectangle, or to write some text in the video memory (framebuffer) displayed on the screen.

GPUs then evolved in the 1990s with the introduction of more 3D graphic processing. Starting with the OpenGL API (see Section 2.2.1) and later with Direct3D, a common set of features began to be used by game developers, leading to more and more vendors implementing these features in hardware in the mid-1990s. At that time, GPUs were not programmable at all and provided hardware for a limited set of operations, but there was already some parallel processing involved under the hood. However, it is only during the 2000s that GPUs became programmable, with the introduction of shaders (see Section 2.2.2). GPU designers then continued to fuse pipeline stages into unified programmable units emulating the plain old OpenGL graphic pipeline.

This architecture survey starts with the AMD architecture. Then the Nvidia G80 that came along with CUDA, and the evolution of the architecture to the current generation, are introduced. Finally the impact of architectural choices on high-level code writing is presented. This section focuses exclusively on main breakthroughs that are relevant for GPGPU. Thus it simply ignores changes that introduce only improvements very specific to graphic workloads.

2.4.2 Building a GPU

A GPU is a huge and complicated piece of hardware. It was traditionally built upon units very specialized for fixed graphic processing functions. With the introduction of shaders, it became more and more programmable. As GPGPU are the main focus of this work, only the computation power of shader parts and the memory hierarchy capabilities and specificities are surveyed.

At the lowest level we find the Processing Element (PE), capable of basic operations like addition or multiplication, or at best a Fused Multiply-Add (FMA). Usually they are limited to single-precision floating point and integer operations. There can also be *on-steroid* PEs able to compute transcendental functions such as trigonometric, exponential, or square roots. Such a unit is called a Special Function Unit (SFU).

Multiple PEs are then grouped together in a Compute Unit (CU). A CU includes all

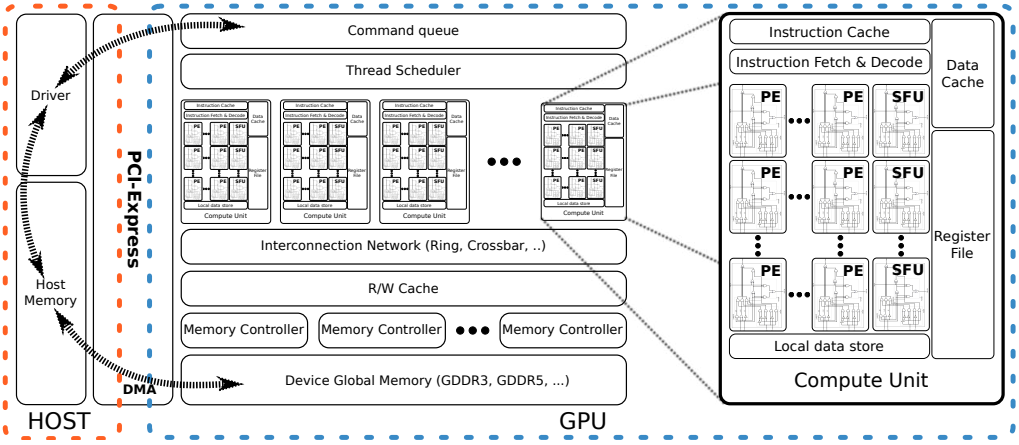


Figure 2.15: High-level simplified GPGPU-oriented view of generic GPU architecture.

the shared logic for PEs, such as instruction fetch and decode, registers, caches, scheduler, and so on.

A GPU chip can then be built by assembling many CUs with an interconnecting network, adding a global hardware scheduler to distribute the work among the CUs and some memory controllers. Sometimes CUs are grouped before being added to the network, and this group shares some resources like cache, on-ship memory network interconnect or also usually graphic centric units. Figure 2.15 illustrates this view of a GPU architecture.

Such a view is not so far from what can be seen in a multicore CPU, but the Devil is in the details. And the choices that are made at each level on the number of elements and the way they are grouped together have a significant impact on the resulting programmability. In general, unlike a CPU, most of the die space in a GPU is used for computing logics. This is why it has a lot of PEs with complex grouping, little to no cache, an important memory bandwidth, but also a long latency.

Most of the time, designers keep CUs as simple as possible and do not include any out-of-order execution capabilities, thus the main source of parallelism is Thread Level Parallelism (TLP). However, Instruction Level Parallelism (ILP) can be exploited by the compiler using a VLIW instruction set, or by the hardware scheduler to keep the pipeline full and to mask memory latency if there is not enough TLP. Figure 2.16 illustrates the difference between ILP and TLP.

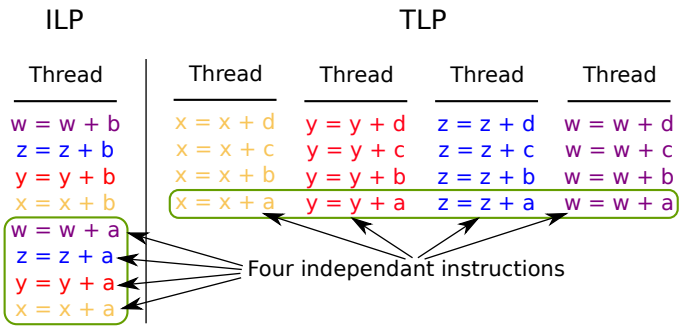


Figure 2.16: Instruction Level Parallelism (ILP) versus Thread Level Parallelism (TPL), two different ways of extracting parallelism in GPUs.

2.4.3 Hardware Atomic Operations

Hardware atomic operations are important for many parallel algorithms and widen the set of applications benefiting from a hardware accelerator. For instance, atomic operations on the GPU have been used to implement barrier synchronizations within a kernel [Xiao & chun Feng 2010], to build high-level programming frameworks such as MapReduce [Ji & Ma 2011], a memory allocator for MapReduce [Hong *et al.* 2010], an histogram [Aubert *et al.* 2009 (perso)].

Nvidia does not disclose any detail about the hardware implementation of atomic operations. It is only known that these units are located in each of the memory controllers [Collange 2010b] on GT200 and directly in the L2 cache since Fermi [Halfhill 2009, Patterson 2009, Collange 2010a].

AMD hardware implementation is slower, so much so that some proposals using software emulation were presented as faster [Elteir *et al.* 2011].

OpenCL supports as of version 1.2 the following integer atomic operations in 32-bit mode:

- `add`: adds an integer to a value at a memory location;
- `sub`: subtracts an integer to a value at a memory location;
- `xchg`: swaps an integer with the value at a memory location;
- `inc`: increments a value at a memory location;
- `dec`: decrements a value at a memory location;
- `cmpxchg`: compares an integer to the value at a memory location and `xchg` if they are equal;

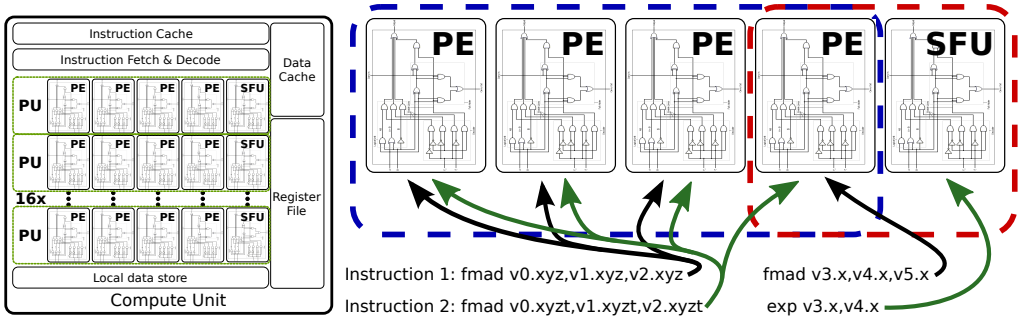


Figure 2.17: AMD R600 Compute Unit (CU) is built on top of 5-way VLIW instructions set. Four Processing Elements (PE) and a Special Function Unit (SFU) are grouped together in a Processing Unit (PU) to process instructions. These PUs are organized in a 16-wide SIMD array.

- `min`: compares an integer to a value at a memory location and stores the smallest value;
- `max`: compares an integer to a value at a memory location and stores the largest value;
- `and`: compares an integer to a value at a memory location and stores the result of a bitwise `and` operation;
- `or`: compares an integer to a value at a memory location and stores the result of a bitwise `or` operation;
- `xor`, compares an integer to a value at a memory location and stores the result of a bitwise `xor` operation.

All these functions operate either in global or local memory, and return the old value. The standard specifies 64-bit versions for all these operations, but the implementation is optional and programmers have to check the availability using OpenCL *extensions*. Both 32-bit and 64-bit versions are supported by Nvidia GPUs since Fermi.

2.4.4 AMD, from R300 to Graphics Core Next

Historically, AMD has used a vector instruction set, and then in 2002 introduced a 2-way VLIW ISA at the beginning of computing shaders with the R300. This architecture was proven to be efficient for handling graphics workload until DirectX 10 and its novelties in shading were introduced.

At that time, shading was quite new and pixel and vertex shaders were separate entities. Vertex shader designers decided that VLIW was the ideal architecture for a vertex shader. It allows processing at the same time one SIMD operation on a four-component vector (e.g., w, x, y, z) and one other operation on a separate scalar component (e.g., lighting).

This organization relies on the compiler to pack the VLIW bundles from the Instruction Level Parallelism (ILP) that can be found in a shader program. By contrast, Thread Level Parallelism (TLP) is handled by replicating these processing units. The static scheduling done by the compiler simplifies the hardware and allows using more of the die space for compute units instead of a complex hardware scheduler.

DirectX 10 introduces the new geometry shaders (see Section 2.2.2) and unifies the programming language for vertex and pixel shaders. These changes pushed GPU designers to unify the architecture. The same units are in charge of all kind of shaders. For AMD GPUs, this change happened with the R600 chip. To achieve such a change, the hardware had to evolve and include more control logic to schedule the different threads that compete for the computing resources. The introduction of hardware schedulers is an important point for GPGPU. It has been critical to enable further hardware evolutions on later architectures.

The novelties introduced by the new DirectX 10 version of the HLSL language drove the designers at AMD to choose a more flexible architecture. While previously based on a 2-way vector/scalar VLIW, the R600 introduced a 5-way pure scalar VLIW instruction set. This way, as before, five individual elements can be processed in each cycle. But the vector has been split. So instead of the same operation on four packed elements, it is possible now to execute five different operations.

ILP is still managed by the compiler that has to pack VLIW bundles. It is even more critical now that five different operations can be packed together. AMD introduced another SIMD level that is exploited implicitly by TLP. The new VLIW units are grouped in a SIMD array of sixteen units. The SIMD benefits only from TLP. At each cycle, one shader 5-way VLIW instruction is scheduled for sixteen different threads. From a graphic workload point of view, it means that a SIMD processing unit handles pixels or vertices by blocks of sixteen, as shown in Figure 2.17.

To increase the computing power of the architecture without increasing the complexity, the control units are limited as much as possible in favor of processing units. A common technique is to use a logical SIMD width wider than the hardware. AMD chose to rely on a virtual sixty-four wide SIMD so that if each cycle a block of sixteen threads is processed,

the instruction scheduler can feed the processing units with a VLIW instruction every four cycles on average. This allows the scheduler to run at a lower frequency than the compute units.

SIMD in such GPUs is managed differently than CPU extensions like Streaming SIMD Extension (SSE) or Advanced Vector eXtensions (AVX). GPU registers are not vectors but dynamically reconfigurable arrays of scalar values. The SIMD execution is implicit and managed by hardware. Another effect is that the mapping from the registers to the lightweight threads that run on the GPU is trivially reconfigurable, offering flexibility on the resource sharing.

While no divergence[3] occurs between the sixty-four threads, all units execute the instruction. If a branch occurs, then threads diverge and PEs are predicated[4]. Since there is only one program counter for a SIMD unit, the different branches are executed sequentially. This behavior offers flexibility to the programmer, who is able to code in a scalar fashion even if he has to keep in mind the architecture characteristics to avoid divergence as much as possible to maximize performance.

The two next generations R700 and Evergreen (R800) did not introduce major new breakthroughs. R700 scales up the R600: it increases frequency, supports Graphic Double Data Rate (GDDR) Dynamic Random Access Memory (DRAM) in version five, and improves the internal bus. Evergreen again extends R700 with more PEs and CUs. Fused Multiply-Add (FMA) and new DirectX 11 instructions are supported, and also improves PEs precision to be IEEE 754-2008 compliant.

The Radeon HD 6900 series, codename Cayman (R900), was released in 2010. This new generation switched to a narrower 4-way VLIW. This reduces the complexity of the units and it is more efficient on the average according to AMD internal tests. Indeed the VLIW average occupation was established to be 3.4 on common workloads. While shaders that were able to fill the VLIW with four-scalar operation and a transcendental operation at the same time suffer from a performance drop, these are not so common. All other shaders benefit from the increased number of SIMD units and the higher TLP.

The main limitation of VLIW comes from the inherent ILP that the compiler is statically able to find in the source program. Moreover, memory accesses are distinct instructions and have to be separated from Arithmetic and Logical Unit (ALU) instructions in

3. There is divergence when the code includes conditional branching and not all threads take the same execution path.

4. When a branch occurs and threads diverge, both paths are executed sequentially and the PEs corresponding to the threads that took the other path are disabled ; they are *predicated*.

Workload	ALUBusy (%)	Packing ratio (%)
BinomialOption	62.51	31.1
Blackscholes	58.58	95.75
Eigenvalue	18.32	54.44
Fastwalsh	56.94	30.83
FloydWarshall	20.35	32.3
Histogram	21.03	33.5
Matmul_2_smem	54.4	81.04
Matmul_no_smem	15.4	73.5
MonteCarloDP	49.29	71.9
Radixsort	3.12	30.9

Figure 2.18: Table summarizing the ALU occupation and the VLIW packing ratio for some computing kernels, taken from [Zhang *et al.* 2011b] (©2011 IEEE).

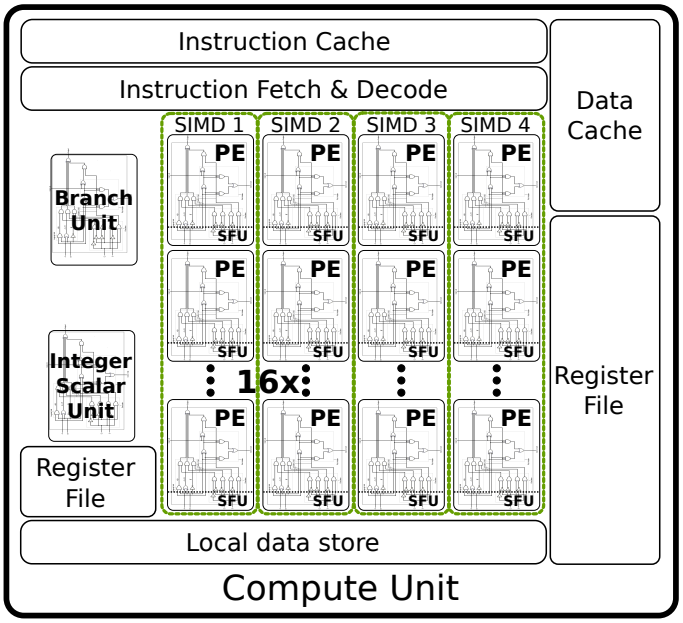


Figure 2.19: The 2012 AMD architecture Graphics Core Next. No longer VLIW, the four separate SIMD pipelines are independent. A new integer scalar unit is introduced. The scheduler feeds each SIMD every four cycles (one per cycle) with a 64-wide virtual SIMD instruction.

what AMD calls a *clause*. If simple graphic workloads are well suited to this constraint, it can be a difficult limitation for complex shaders and more specially for GPGPU. Figure 2.18 shows some statistics about the ALU occupation and the VLIW packing ratio[5] for some computing kernels.

5. The packing ratio indicates on the average how many instructions are packed in the VLIW by the compiler with respect to the VLIW width.

The last generation, codename Graphics Core Next (GCN) (R1000), introduces a major breakthrough. Driven by the success of GPGPU, AMD chose to quit their VLIW ISA in favor of a scalar architecture. Basically they have split their 4-way VLIW into four separate SIMD pipelines. It means that ILP will no longer be exhibited by the compiler and that these units exploits TLP instead. As long as the workload exhibits enough threads, it is easier to reach the architectural peak performance.

Another novelty from Graphics Core Next (GCN) is that that these four SIMDs are packed along with an ALU scalar unit. Among other uses, this unit can perform pure scalar computation and avoid wasting resources underusing a SIMD for branch or mask prediction computations, a function call, or a jump. An overview of a GCN's compute unit is given in Figure 2.19.

These changes put more pressure on the hardware scheduler. Early announcements about GCN mention that it is able to schedule ten groups of sixty-four threads per SIMD, that is 2560 threads per compute unit. Such a high number of threads helps to hide memory latency. The presence of four different pipelines to feed increases also the requirements on the scheduler. While previously one VLIW instruction was processed in four cycles by the SIMD pipeline, the scheduler has now to feed a separate instruction every cycle. Indeed it considers each cycle thread for one of the four SIMD and issues up to five instructions among these: one for the vector unit, one for the scalar ALU, one for a vector memory access, one for the branching unit, the local data store, for the global data share, or an internal one. [6]

GCN also includes for the first a time a fully hierarchical hardware-managed cache, while the previous architecture only had an L2 cache and a software-managed Local Data Store (LDS) located within each CU.

As of early 2012, GCN is not released and we have thus no way to experiment with this new architecture.

Figure 2.20 summarizes the evolution of PE grouping across AMD architectures.

AMD has later released this information about GCN in a white paper [AMD 2012].

6. Internal instructions are NOPs, barriers, etc.

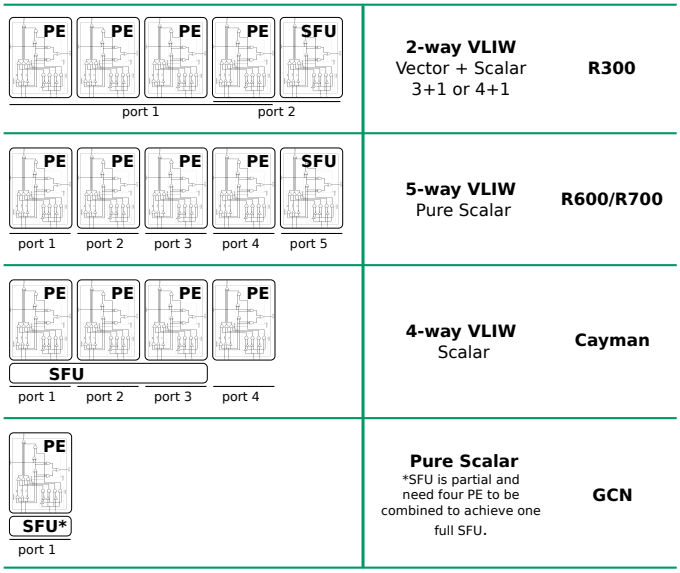


Figure 2.20: Evolution of Processing Element (PE) grouping across AMD architectures.

2.4.5 Nvidia Computing Unified Device Architecture, from G80 to Kepler

In the fall of 2006, Nvidia released the G80. It was the first DirectX 10 compliant GPU. It is the result of a four-year effort, starting nearly from scratch with a full redesign. While in previous Nvidia architecture the compute units were specialized, the G80 is a complete unification of many stages. As for the AMD R600, one of the most visible and effective change from a GPGPU viewpoint is that the compute units are now indifferently able to process any kind of shader (see Section 2.2.2).

Another important novelty is that the compute units offer a scalar interface, similar to the one AMD announced with AMD GCN, but six years earlier.

The G80 has many groups (typically sixteen) of eight scalar processors. Each group is considered by Nvidia as a multiprocessor. There is one shared instruction issue unit for a group, responsible of feeding the eight processors. This feeding unit runs at half the frequency of the scalar processors and needs two cycles to feed an instruction. The GPU efficiency is maximized when thirty-two threads execute the same instruction in a SIMD fashion (see Figure 2.21). Again TLP is exploited and ILP is not directly exhibited by the architecture, but can be exploited to hide memory latency as shown by Volkov [Volkov 2010]. The scheduler can benefit from independent instructions to issue

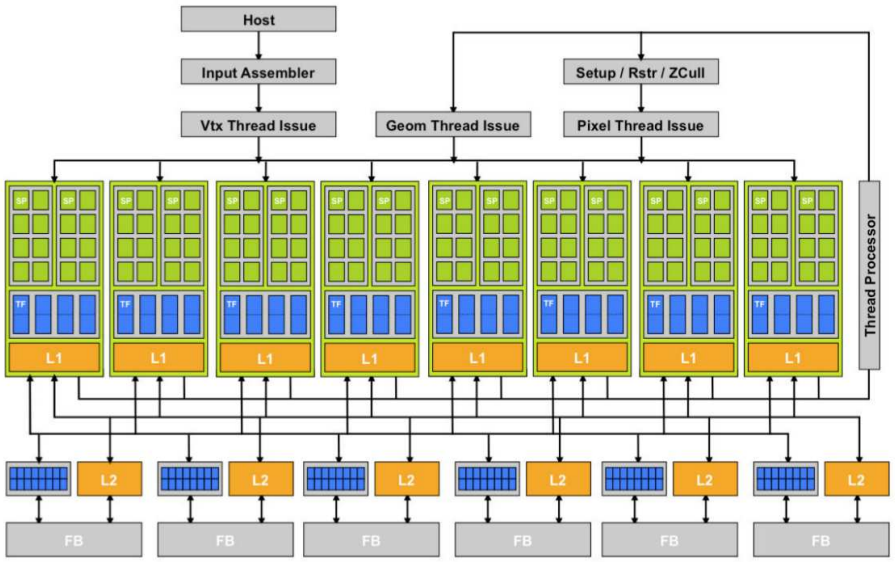


Figure 2.21: The GeForce 8800 architecture (G80) introduced unified shaders where shader programmable processors can be used to replace multiple stages of the classic graphic pipeline. There are still specialized units for some graphics operations. (Source: Nvidia)

multiple instructions for the same thread in the pipeline.

GT200 was released in 2008 as an evolution of the G80. The main visible change from a GPGPU point of view is the introduction of a double precision floating point unit in CUs along with the PEs, providing 1/8th the single-precision floating point computation power. Another novelty is the support of atomic operations in global memory.

In 2010, Nvidia released a major revision of the GT200: Fermi. It comes with a large number of improvements in a GPGPU perspective:

- Indirect control flow is now supported and opens the gate to C++ and virtual functions.
- Fine grained exception handling has been added to support C++ try-and-catch clause.
- Unified address space allows a simpler memory model where the hardware automatically resolves the location of an address (thread private, shared, global, system).
- Hardware-managed hierarchical caches are introduced for the first time. While the previous generation had read-only caches for texture, Fermi comes with a L1 cache located in each CUs, and a global L2 cache.

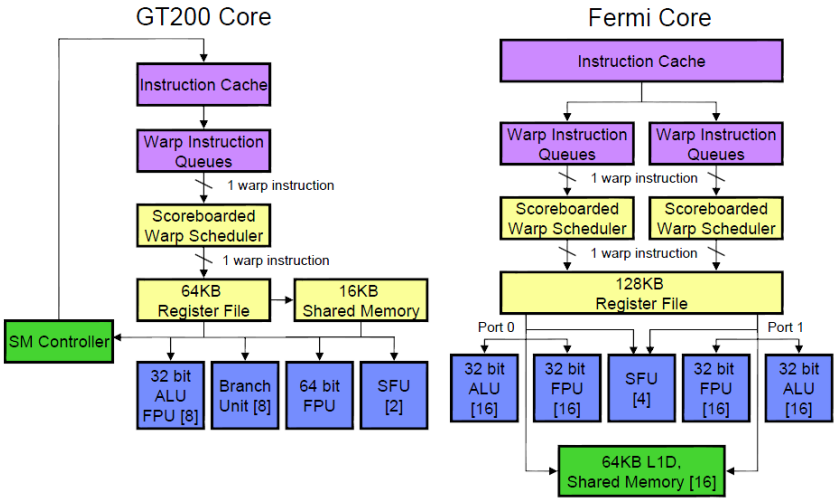


Figure 2.22: GT200 compute unit (CU) on the left, Fermi's on the right. Processing Elements (PE) upgrade from eight to sixteen per pipeline, but the logical SIMD width is unchanged, threads are scheduled by groups of thirty-two (source Nvidia).

- The PCI Express (PCIe) interface is now able to execute Direct Memory Access (DMA) in both direction at the same time.
- The global scheduler is now able to process multiple kernels at the same time. A CU still only has in-flight threads from one kernel at a time.
- FMA is supported in the PEs, and the IEEE-754 rounding modes are all supported.
- Atomic operations execute directly in the L2 cache without having to write back the result in global memory.
- ECC is supported in the global memory, the L1 and L2 caches, and the register file.

Figure 2.22 illustrates the evolution of CU between GT200 and Fermi side by side. The number of PEs is increased from eight to thirty-two, split in two separate pipelines and two schedulers. An instruction is processed each cycle for sixteen threads. Two instructions can be issued every two cycles, ending up with a logical thirty-two wide SIMD view as in G80 and GT200.

The Special Function Units (SFUs) have their own pipeline shared between the two schedulers. Since there are only four SFUs, the throughput is four times longer than for ALU operations. ILP allows a scheduler to feed the ALU pipeline while some SFU computations are running. Therefore there can be forty-eight threads processed per CU at the same time.

March 2012 has seen the release of a new major architecture by Nvidia, codename Kepler. The most important point that Nvidia emphasizes with Kepler is the performance-per-watt ratio, that they achieve mostly by reducing the frequency of the PEs by a half to be the same as the instruction scheduling unit. The four big new architectural improvements are the following:

- Dynamic Parallelism adds the capability for the GPU to generate new work for itself. It make available the CUDA host API directly in the device code. A kernel can then initiate memory transfers, or launch other kernels. This provides a disruptive change in the CUDA programming model as known for years.
- Hyper-Q is the mechanism that allows up to thirty-two host threads to initiate command to the GPU in parallel, allowing more concurrent kernel parallelism to be exploited.
- Grid Management Unit is the basis piece of hardware that enables Dynamic Parallelism. It replaces the previous scheduler providing flexibility in the dispatch, queuing, and dependency of up to 2000 different kernel launches waiting for execution.
- GPU Direct allows transfer of data between different GPUs or between a GPU and any other PCIe piece of hardware directly over the PCIe bus without involving the host.

Kepler is currently available only for gaming and graphical usage with the GTX 680. It does not currently include all the novelty of the architecture that will be available with the Tesla K20 by the end of 2012 along with CUDA 5. Other than these four key features, the most visible change at that time is the organization of the PEs in the CUs. While they were previously grouped by eight on the G80 and thirty-two or forty-eight on Fermi, Kepler is shipped with 192 PEs per CU while keeping the classical logical 32-wide SIMD view. The number of schedulers is doubled to four, but operating now at the same frequency as the PE, it provides the same ratio as the forty-eight PE Fermi CU. Also the ratio between double precision and single precision goes down to one third while it was one half on Fermi.

The Gefore GTX 680 is currently shipped with eight CUs, but Nvidia announced fifteen CUs, i.e., 2880 PE in the Tesla K20, resulting in over one TFlop of double precision throughput and over four TFlops using single precision.

On the memory side, the GDDR5 has been improved and should provide performance closer to the theoretical peak. The announced bandwidth for the Tesla K20 is raised to 320 GB/s, which is nearly twice Fermi's capability. The L2 cache is also doubled both in bandwidth and size, as the memory bandwidth.

It is interesting to note that the balance of resources per CU when compared to Fermi shows that the capacity of each CU scheduling has been doubled in term of number of workgroups but multiplied by only 1.3 in terms of number of threads[7]. It seems to be in favor of smaller workgroups when compared to Fermi. The L1 cache keeps the same size while the number of PE increases, leading to more potential concurrency and contention.

Other less important improvements reside in the more efficient atomic operations, the ECC overhead reduced by 66% on average, the GPU Boost technology that increases or decrease the frequency dynamically to keep the power consumption in a given limit, and a new `shuffle` instruction to exchange data between threads of a same warp.

2.4.6 Impact on Code Generation

In this dissertation, no particular architecture is targeted and we want to be able to generate code that runs efficiently on all the architectures introduced previously. The main question is this: to what extent is performance portable from one architecture to another?

Since scalar and VLIW targets are exposed, it is difficult to expect a unique universal solution. Extracting more ILP may require exposing less TLP and thus might lead to starving on a scalar architecture.

Chapter 7 presents various experiments, and the comparison of the performance obtained on different architectures after various transformations confirms that improving the performance for a given architecture reduces it on another architecture.

Another concern is about predicting statically that one version of a kernel will run faster than another. Even given a particular architecture it is a complex issue. For instance, for a very simple kernel, the number of work-items that we allocate in a work-group has an important impact on the resulting performance. Figure 2.23 shows the influence of runtime parameters on performance of Nvidia Fermi and AMD Evergreen. The left graphic shows different launch configurations for a set of kernels. While Evergreen is not very sensitive to it, Fermi shows up to a speedup of two by adjusting the workgroup size. The comparison of BinomialOption and Matmul_no_smem indicates that there is no universal work-group size. Zhang et al. [Zhang *et al.* 2011b] demonstrate that the loss in performance when increasing the work-group size from 128 to 256 for BinomialOption is correlated to a larger number of global memory accesses. In this case improving parallelism degrades the overall performance. On the right a matrix multiplication kernel, without any local

7. The maximum number of resident workgroups per multiprocessor is eight on Fermi and sixteen on Kepler, the maximum number of resident threads per multiprocessor is 1536 on Fermi and 2048 on Kepler.

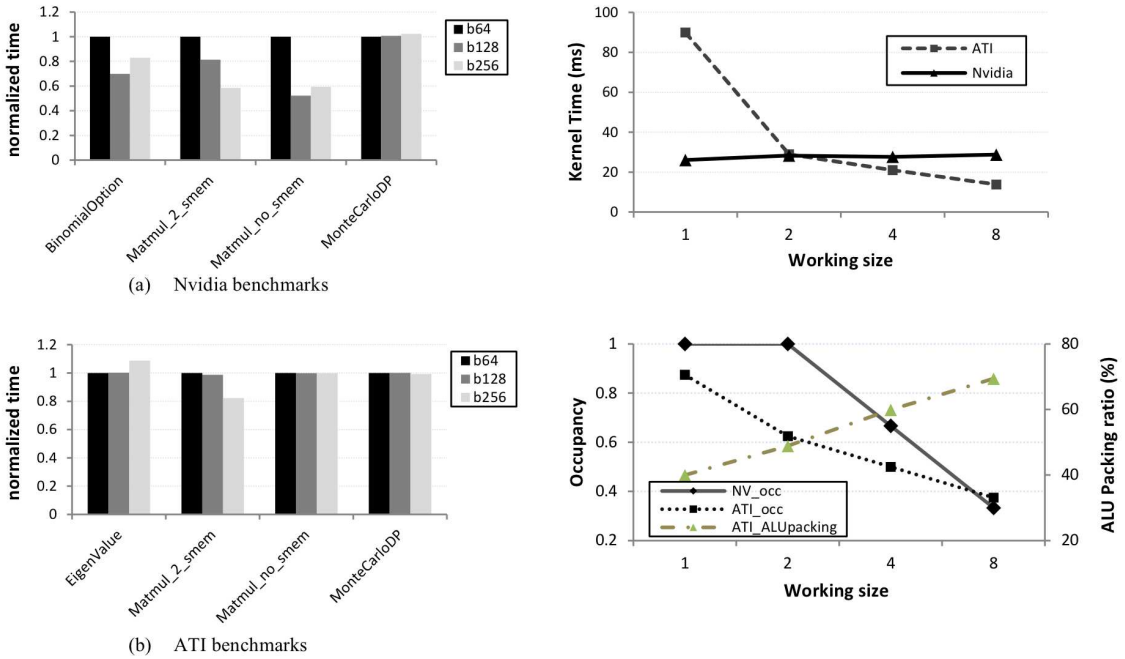


Figure 2.23: Influence of runtime parameters on performance for different kernels and different architectures, Nvidia Fermi and AMD Evergreen. On the left the launch configuration for different kernels shows that there is no universal work-group size. On the right a matrix multiply kernel without local data store optimization is used with one to four elements processed in each thread. The upper part shows the impact on performance for both architectures while the lower part shows the occupancy of the AMD GPU and the VLIW packing ratio. Taken from [Zhang *et al.* 2011b] (©2011 IEEE).

data store optimization, is tested with one to four elements produced in each thread. The upper part shows the impact on performance for both architectures while the lower part shows the occupancy [8] of the GPU and, for AMD, the VLIW packing ratio. The Evergreen VLIW architecture proves to be very sensitive as more elements to process means more opportunities for the compiler to extract ILP. The performance (upper graphic) and the packing ratio (lower graphic) are correlated and confirm this analysis. Fermi and its scalar architecture are less impacted by this change and exhibit nearly constant performance. But the lower graphic shows that the occupancy drops significantly, leading to fewer opportunities for TLP, and thus potentially fewer opportunities to mask memory latency with computations in some kernels.

The impact of the launch configuration is explored with more detail in Section 5.8.

8. Occupancy is ratio of the number of eligible threads over the maximum number of resident threads.

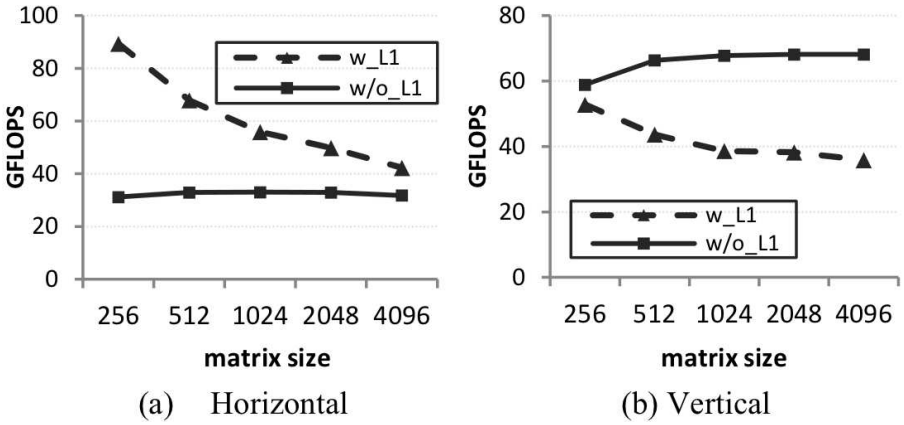


Figure 2.24: Performance of two different versions of matrix multiply kernel, a *horizontal* scheme and a vertical scheme, without local memory usage for a given architecture (Fermi), depending on the input size and the activation or not of the L1 cache. Taken from [Zhang *et al.* 2011b] (©2011 IEEE).

Finally, the input size, which generally defines the maximum number of work-items that are exploited, is known only at runtime, which limits the possibilities of one universal version for a particular kernel. Figure 2.24 shows the performance evolution depending on matrix size for two different versions of matrix multiplication on Fermi with and without L1 cache enabled. The local memory is not used in any of these two versions. The fact that L1 can be activated or not on a per kernel basis is another parameter that might influence the choice of a particular kernel version to get the best performance. Section 5.8 covers in detail the implication of the launch configuration over performance, and in Section 7.4 provides experimental results.

2.4.7 Summary

GPUs exhibit massively parallel architecture. They rely mostly on Thread Level Parallelism (TLP) to expose parallelism as thousands of threads, but, depending on the underlying architecture, Instruction Level Parallelism (ILP) may also be a must to get decent performance.

For a deeper insight in the architectural mysteries, such as latency for each operation, deep understanding of caches and so on, the reader is referred to the work of Taylor and Li on benchmarking the AMD architecture [Taylor & Li 2010], the work of Wong et al. [Wong *et al.* 2010] and Collange [Collange 2010b] for the GT200, and Lindholm et al. [Lindholm

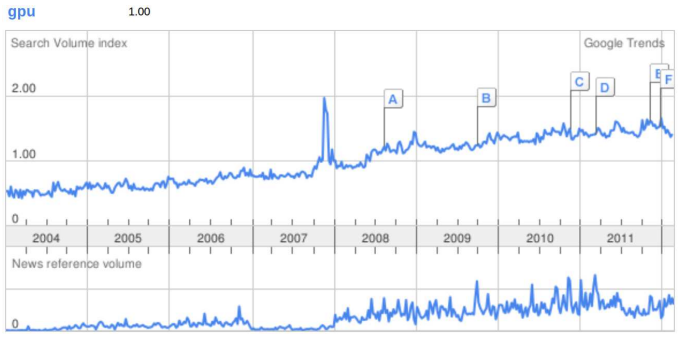


Figure 2.25: Google trends for the word GPU during last decade.

et al. 2008] and Collange's PhD thesis [Collange 2010a] for Fermi.

2.5 Conclusion

A decade ago, the General-Purpose Processing on Graphics Processing Units (GPGPU) computing was in its early days. Since then, it has been an intense field of research and it still very active as shown in Figure 2.25. As shown in this chapter, many languages, frameworks, and other solutions have been introduced to help programmers write programs that exploit accelerators. All these approaches provide varied trade-offs of the *three P*s: Performance, Portability, Programmability. The performance portability is a challenge in the context of GPGPU. The programmability has been addressed by several specialized programming languages.

The landscape in 2012 is very different from what it looked like more than ten years ago, when researchers were trying to leverage the pure graphic-oriented OpenGL pipeline to abstract mathematical operators [Trendall & Stewart 2000] or to use it as a target for compiling streaming language.

The programmability was very challenging. However, when a program was successfully mapped to the OpenGL API, performance and portability were obtained. The introduction of shaders in 2002 brought flexibility and exposed more features, resulting in improved programmability.

More recent approaches tried dedicated languages such as Brooks to trade performance for programmability. The underlying streaming programming model is a convenient interface for programmers, but is not flexible enough to be mainstream.

The evolution of DirectX drove GPU manufacturers toward more programmability. But

the tipping point is the introduction of CUDA, then followed by alternative programming languages and frameworks.

The CUDA and OpenCL programming models leverage the experience with shaders to provide an equivalent level of programmability but without all the rigid mechanisms implied by the graphic API. However, programmers have to know well the architecture to write efficient kernels: the portability is traded for performance.

Directive-based languages such as hiCUDA, JCUDA, HMPP, PGI Accelerator, or OpenACC (see Section 2.2.10) are less invasive and provide good portability at the expense of performance. The directives can be specialized for a given target to increase the performance, but at the price of portability.

My PhD work started just after the first release of OpenCL 1.0 for GPU by Nvidia in spring 2009 [Ramey 2009]. The goal of my work was to provide an end-to-end solution that relieves programmers of adapting their codes to hardware accelerators.

The programmability is as good as possible, since programmers write their codes using standard sequential programming languages, ignoring their heterogeneous targets. The compiler extracts the parallelism and the code to be executed on the accelerator. The performance may not match what an expert would get with effort. However, the trade-off on performance is acceptable if it is limited, such as for example ten, twenty, or thirty percent depending on the application domain.

Very few people tried to provide full automatic parallelization and transformation (see Section 2.2.11 page 30) from sequential code to GPU. Most are limited in applicability or focus only on part of the problem. My work tries to process a whole application, generate kernels, optimize them all, and generate the required communication, without any user input.

Guelton proposes in his PhD thesis [Guelton 2011a] a general high-level scheme for an heterogeneous compiler targeting GPUs. The compiler transforms the code, separating the host code and the kernel code, with the required *glue*. Each part is then compiled by dedicated binary compilers for the target. This is shown in Figure 2.26.

My work instantiates this high-level compilation scheme. An overview of my compiler structure and organization is presented in Figure 2.27. It addresses all compilation issues raised by heterogeneous computing with CPUs and GPUs. While not exploring deeply each concern, this dissertation provides solutions to many issues related to automatic parallelization for GPUs, ranging from parallelism detection to code generation, passing through loop nests optimizations and management of data mapping and consistency.

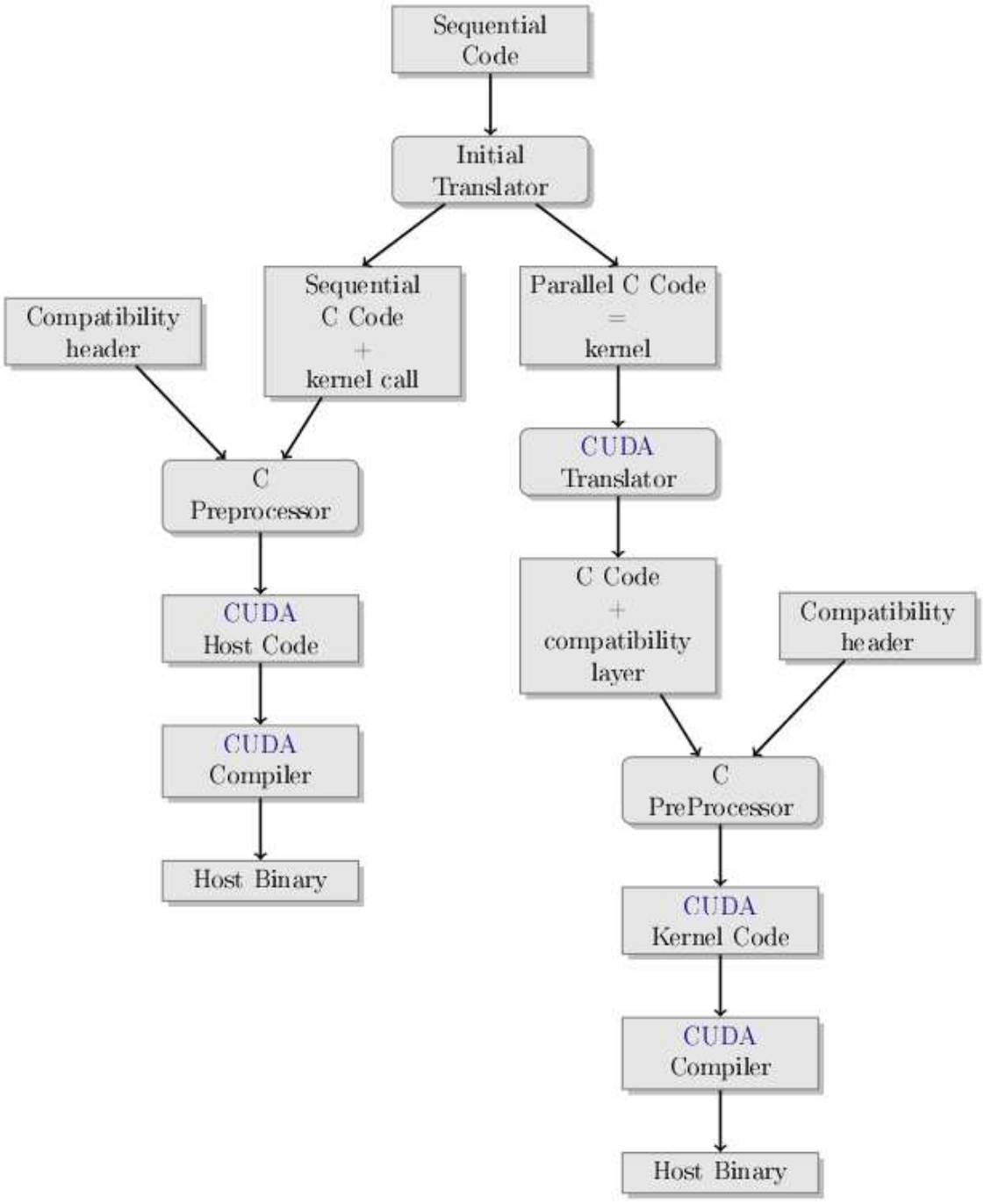


Figure 2.26: Source-to-source compilation scheme for GPU (source [Guelton 2011a]).

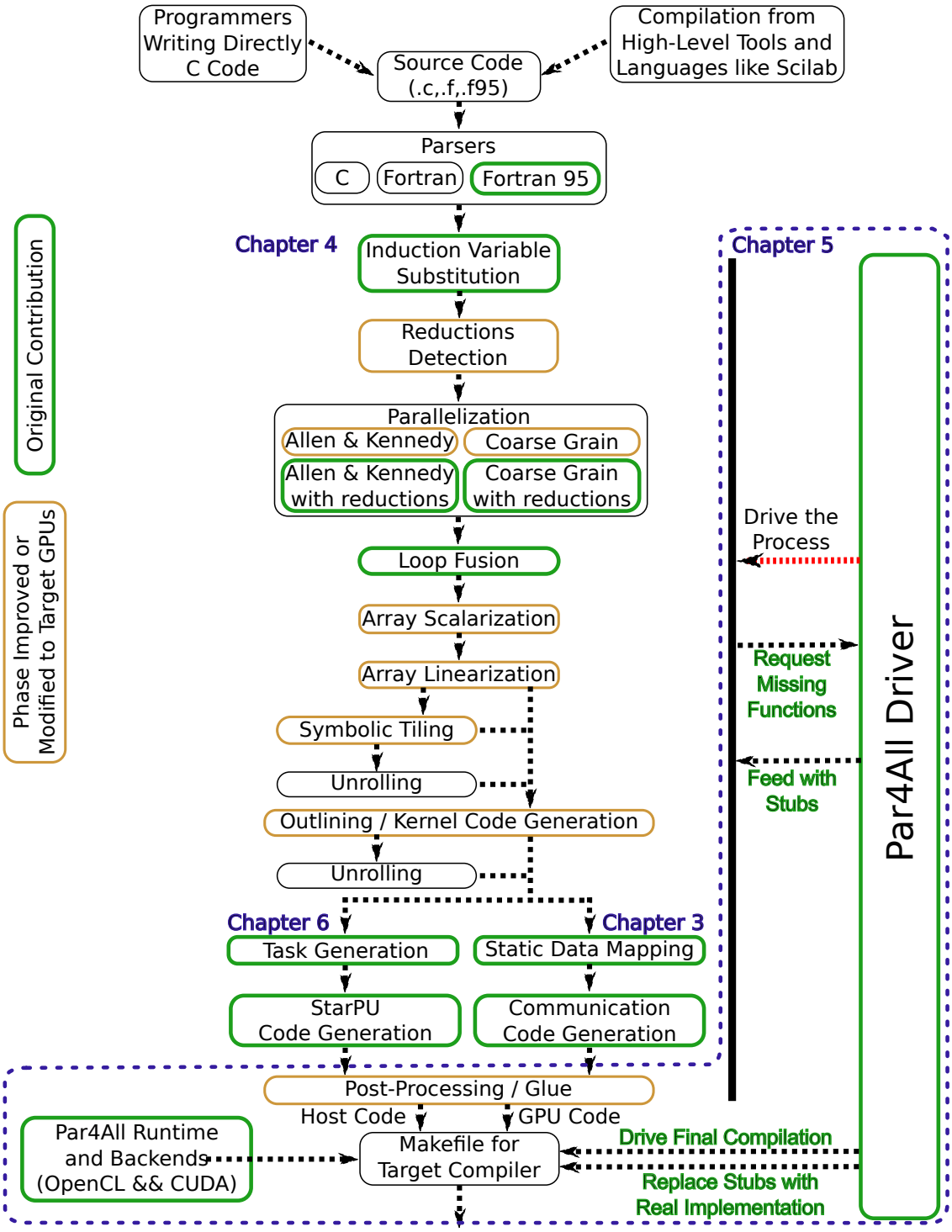


Figure 2.27: Overview of the global compilation scheme.

Figure 2.27 illustrates the compilation chain. The source code is first analyzed to find parallelism, and transformed before extracting the code to be executed on the GPU in new functions: the *kernels*. Some optimization phases can be applied such as loop fusion, array scalarization, array linearization, symbolic tiling, or unrolling. This part of the process is presented in Chapter 4. After kernel generation, analyses and transformations to generate communications are required. Array regions are used in Chapter 3 to achieve accurate communication generation. An interprocedural static analysis is proposed to optimize the communication by leaving data as much as possible on the GPU. Another path is the generation of tasks that are scheduled at runtime on multiple GPUs using StarPU. The task extraction and code generation for StarPU are presented in Chapter 6, along with another mapping on multiple GPUs based on symbolic tiling. The whole process is driven by the new Par4All driver, from the input source code to the final binary. It is based on a flexible pass manager. The challenge of automating the whole process is presented in Chapter 5. The experimental results are presented and discussed in Chapter 7.

Chapter 3

Data Mapping, Communications and Consistency

Contents

3.1 Case Study . . . 63
3.2 Array Region Analysis . . . 64
3.3 Basic Transformation Process . . . 68
3.4 Region Refinement Scheme . . . 70
3.4.1 Converting Convex Array Regions into Data Transfers . . . 73
3.4.2 Managing Variable Substitutions . . . 74
3.5 Limits . . . 76
3.6 Communication Optimization Algorithm . . . 77
3.6.1 A New Analysis: Kernel Data Mapping . . . 78
3.6.2 Definitions . . . 79
3.6.3 Intraprocedural Phase . . . 80
3.6.4 Interprocedural Extension . . . 81
3.6.5 Runtime Library . . . 82
3.7 Sequential Promotion . . . 84
3.7.1 Experimental Results . . . 86
3.8 Related Work . . . 87
3.8.1 Redundant Load-Store Elimination . . . 88
3.9 Optimizing a Tiled Loop Nest . . . 90
3.10 Conclusion . . . 93

GPU-like accelerators process data located in their own memory. Indeed, an accelerator board embeds a few gigabytes of memory with high bandwidth to feed their many CUs as discussed in Section 2.4. The difficulty is that this embedded memory is not visible from the host CPU and reciprocally host memory is not visible from the GPU. [1] The programmers then have to explicitly transfer input data from the host memory to the accelerator's before launching a kernel and then execute some opposite transfers from the accelerator memory to the host's after kernel execution for the data produced by the kernel.

These explicit communications use slow I/O buses. For example, PCIe 2.0 bus offers a peak 8 GB/s, to be compared with a few hundreds of GB/s available using the on-board GDDR memory. This is generally assumed to be *the* most important bottleneck for hybrid systems [Chen *et al.* 2010].

Work has been done to address this issue either using simplified input from programmers [Yan *et al.* 2009, CAPS Entreprise 2010, Wolfe 2011, NVIDIA, Cray, PGI, CAPS 2011], or automatically [Amini *et al.* 2011c (perso), Ventroux *et al.* 2012, Guelton 2011a, Alias *et al.* 2011, Wolfe 2010] using compilers. A lazy scheme has also been proposed by Enmyren and Kessler [Enmyren & Kessler 2010] in the SkePU C++ template library, a skeleton programming framework for multicore CPUs and multi-GPU systems.

This chapter studies the issues associated with the generation of communication in the context of automatically or semi-automatically offloading work to an accelerator and presents several contributions to address this issue: array regions are exploited to optimize the amount of data to transfer per kernel and a static interprocedural communication optimization scheme is designed and implemented in Paralléliseur Interprocédural de Programmes Scientifiques (PIPS).

PIPS is a twenty-year-old compiler framework [Irigoin *et al.* 1991, Amini *et al.* 2011a (perso)] that offers semantic analysis and transformation passes. Initially targeting Fortan 77 as an input, it has been then extended to handle C code. It aims at exploring different program optimizations using interprocedural analyses. Unlike heroes from other projects that target binary level parallelization [Pradelle *et al.* 2012, Kotha *et al.* 2010], PIPS operates at source level trying to regenerate a code as close as possible to the input.

First, the targeted program class is introduced with a case study: a cosmological simulation. Then the convex array region abstraction, which is the basis of most of the transformations this work relies on, is introduced in Section 3.2. The most basic mapping

1. Some recent solutions like Nvidia Zero-Copy allow mapping directly the host memory in the GPU virtual space and thus avoid the explicit copy. However, they do not provide good performance in the general case.

Figure 3.1: Stars-PM is a N-body cosmological simulation. Here a satellite triggers a bar and spiral arms in a galactic disc.

is then described in Section 3.3 to give insight on the principle involved. Array region analyses are used to refine the process of Section 3.3 in Section 3.4. The limits of this approach are given in Section 3.5. An interprocedural optimization is then proposed in Section 3.6 to efficiently map the data on the accelerator and limit the number of transfers.

The parallel promotion presented in Section 3.7 may help as a complement to loop fusion to reduce synchronization and sometimes memory transfers.

Finally, the related work about optimizing the communication for GPGPU is presented in Section 3.8.

3.1 Case Study

Small benchmarks like those used in the Polybench suite [Pouchet 2011] for example, are limited to a few kernels in sequence, sometimes surrounded by a time step loop. Therefore, if they are well suited for studying the pure performance of the GPUs, they cannot be considered representative of a whole application when it comes to evaluating a global problem like mapping of data between the host and the accelerator.

To address this issue, my study is based on a program more representative of numerical simulations. It is a real numerical simulation called Stars-PM, a particle mesh cosmological N-body code whose output is shown in Figure 3.1. The sequential version was written in

```
int main(int argc, char *argv[]) {

  // Read initial conditions from a file
  init_data(argv[1]);

  // Time loop
  for(t = 0; t < T; t += DT) {
    // Do computations for each iteration
  }

  // Output results to a file
  write_data(argv[2]);
}
```

Figure 3.2: Simplified global scheme commonly used in numerical simulations.

C at *Observatoire Astronomique de Strasbourg* and was later rewritten and optimized by hand using CUDA to target GPUs [Aubert *et al.* 2009 (perso)].

This simulation models the gravitational interactions between particles in space. It discretizes the three-dimensional space with a discrete grid on which particles are mapped. Initial conditions are read from a file. A sequential loop iterates over successive time steps, where the core of the simulation is computed. Results are finally extracted from the final grid state and stored in an output file. This general organization is shown in the simplified code in Figure 3.2. It is a common scheme in numerical simulations, while the core of each iteration can vary widely from one domain to the other. The sub-steps performed for processing a single time step in Stars-PM are illustrated Figure 3.3.

3.2 Array Region Analysis

Several transformations in the compilation flow used to target hardware accelerators are based on array regions. This section provides a basic introduction to this representation. Three examples are used throughout this section to illustrate this approach: the code in Figure 3.4 requires interprocedural array accesses analysis, the code in Figure 3.5 contains a `while` loop, for which the memory access pattern requires an approximated analysis, and the code in Figure 3.6 features a nontrivial switch-case.

Convex array regions were first introduced by Triolet [Triolet 1984, Triolet *et al.* 1986] with the initial purpose of summarizing the memory accesses performed on array element sets by function calls. The concept was later generalized and formally defined for any

```
void iteration(coord pos[NP][NP][NP],
               coord vel[NP][NP][NP],
               float dens[NP][NP][NP],
               int data[NP][NP][NP],
               int histo[NP][NP][NP]) {

  // Step 1 : Cut the 3D space in a regular mesh
  discretisation(pos, data);

  // Step 2 : Compute density on the grid
  histogram(data, histo);

  // Step 3 : Compute potential on the mesh
  // in the Fourier space
  potential(histo, dens);

  // Step 4 : For each dimension, compute the
  // force and then update the speed
  forcex(dens, force);
  updatevel(vel, force, data, X_DIM, dt);
  forcey(dens, force);
  updatevel(vel, force, data, Y_DIM, dt);
  forcez(dens, force);
  updatevel(vel, force, data, Z_DIM, dt);

  // Step 5 : Move particles
  updatepos(pos, vel);
}
```

Figure 3.3: Outline of one time step in the Stars-PM cosmological simulation code.

program statements by Creusillet [Creusillet & Irigoin 1996b, Creusillet 1996] and implemented in the PIPS compiler framework.

Informally, the READ (resp. WRITE) regions for a statement s are the set of all scalar variables and array elements that are read (resp. written) during the execution of s. This set generally depends on the values of some program variables at the entry point of statement s: the READ regions are said to be a function of the memory state σ preceding the statement execution, and they are collectively denoted $\mathcal{R}(s, \sigma)$ (resp. $\mathcal{W}(s, \sigma)$ for the WRITE regions).

For instance the READ regions associated to the `for` statement in function `kernel` in Figure 3.4 are these:

$$\mathcal{R}(s, \sigma) = \{\{\mathtt{v}\}, \{\mathtt{i}\}, \{\mathtt{j}\}, \{\mathtt{src}(\phi_1) \mid \phi_1 = \sigma(\mathtt{i}) + \sigma(\mathtt{j})\}, \{\mathtt{m}(\phi_1) \mid \phi_1 = \sigma(\mathtt{j})\}\}$$

```
// R(src) = {src[φ1] | i ≤ φ1 ≤ i + k − 1}
// W(dst) = {dst[φ1] | φ1 = i}
// R(m) = {m[φ1] | 0 ≤ φ1 ≤ k − 1}
int kernel(int i,int n,int k,int src[n],int dst[n-k],int m[k]) {
  int v=0;
  for( int j = 0; j < k; ++j )
    v += src[ i + j ] * m[ j ];
  dst[i]=v;
}
void fir( int n, int k, int src[n], int dst[n-k], int m[k]) {
  for( int i = 0; i < n - k+ 1; ++i )
    // R(src) = {src[φ1] | i ≤ φ1 ≤ i + k − 1, 0 ≤ i ≤ n − k}
    // R(m) = {m[φ1] | 0 ≤ φ1 ≤ k − 1}
    // W(dst) = {dst[φ1] | φ1 = i}
    kernel(i, n, k, src, dst, m);
}
```

Figure 3.4: Array regions on a code with a function call.

```
// R̄(randv) = {randv[φ1] | N − 3 ≤ 4 × φ1; 3 × φ1 ≤ N}
// W̄(a) = {a[φ1] | N − 3 ≤ 4 × φ1; 12 × φ1 ≤ 5 × N + 9}
void foo(int N, int a[N], int randv[N]) {
  int x=N/4,y=0;
  while(x<=N/3) {
    a[x+y] = x+y;
    if (randv[x-y]) x = x+2; else x++,y++;
  }
}
```

Figure 3.5: Array regions on a code with a `while` loop.

where ϕ_x is used to describe the constraints on the xth dimension of an array, and where $\sigma(\mathtt{i})$ denotes the value of the program variable `i` in the memory state σ. From this point, i is used instead of $\sigma(\mathtt{i})$ when there is no ambiguity.

The regions given above correspond to a very simple statement; however, they can be computed for every level of compound statements. For instance, the READ regions of the `for` loop on line 6 in the code in Figure 3.4 are these:

$$\mathcal{R}(s,\sigma) = \{\{\mathtt{v}\},\{\mathtt{i}\},\{\mathtt{src}(\phi_1) \mid i \leq \phi_1 \leq i+k-1\},\{\mathtt{m}(\phi_1) \mid 0 \leq \phi_1 \leq k-1\}\}$$

However, computing exact sets is not always possible, either because the compiler lacks information about the values of variables or the program control flow, or because the regions

```
// R̄(in) = {in[φ1] | i ≤ φ1 ≤ i + 2}
// W̄(out) = {out[φ1] | φ1 = i}
void foo(int n, int i, char c, int out[n], int in[n]) {
  switch(c){
    case 'a':
    case 'e':
      out[i]=in[i];
      break;
    default:
      out[i]=in[3*(i/3)+2];
  }
}
```

Figure 3.6: Array regions on a code with a `switch` case.

cannot be exactly represented by a convex polyhedron. In these cases, over-approximated convex sets (denoted $\overline{\mathcal{R}}$ and $\overline{\mathcal{W}}$) are computed. In the following example, the approximation is due to the fact that the exact set contains holes, and cannot be represented by a convex polyhedron:

$$\overline{\mathcal{W}}([\![\texttt{for(int i=0; i<n; i++) if (i != 3) a[i]=0;}]\!], \sigma) = \{\{\texttt{n}\}, \{\texttt{a}[\phi_0] \mid 0 \leq \phi_0 < n\}\}$$

whereas in the next example, the approximation is due to the fact that the condition and its negation are nonlinear expressions that cannot be represented exactly in PIPS framework:

$$\overline{\mathcal{R}}([\![\texttt{if(a[i]>3) b[i]=1; else c[i]=1}]\!], \sigma) = \{\{\texttt{i}\}, \{\texttt{a}[\phi_0] \mid \phi_0 = i\}, \{\texttt{b}[\phi_0] \mid \phi_0 = i\}, \{\texttt{c}[\phi_0] \mid \phi_0 = i\}\}$$

Under-approximations (denoted $\underline{\mathcal{R}}$ and $\underline{\mathcal{W}}$) are required when computing region differences (see [Creusillet & Irigoin 1996a] for more details on approximations when using the convex polyhedron lattice).

READ and WRITE regions summarize the effects of statements and functions upon array elements, but they do not take into account the flow of array element values. For that purpose, IN and OUT regions have been introduced in [Creusillet & Irigoin 1996b] to take array kills into account, that is to say, redefinitions of individual array elements:

- IN regions contain the array elements whose values are *imported* by the considered statement, which means the elements that are read before being possibly redefined by another instruction of the statement.

- OUT regions contain the array elements defined by the considered statement, which are used afterwards in the program continuation. They are the *live* or *exported* array elements.

As for READ and WRITE regions, IN and OUT regions may be over- or under-approximated.

There is a strong analogy between the array regions of a statement and the memory used in this statement, at least from an external point of view, which means excluding its privately declared variables. Intuitively, the memory footprint of a statement can be obtained by counting the points in its associated array regions. In the same way, the READ (or IN) and WRITE (or OUT) regions can be used to compute the memory transfers required to execute this statement in a new memory space built from the original space. This analogy is analyzed and leveraged in the following sections.

3.3 Basic Transformation Process

The most basic process for mapping data to the accelerator consists in sending to the accelerator all arrays that are referenced in a kernel prior executing it. The same set of arrays has to be transferred back from the accelerator memory at the end of kernel execution. This basic process is the most basic that can be used by automatic tools. It is represented in Figure 3.7.

The main issue arises when it is needed to count the number of array elements to transfer. Depending on the target language or framework, the information can be hard to figure out. Leung et al. [Leung *et al.* 2009] and JCUDA [Yan *et al.* 2009] target Java and benefit from runtime information about array sizes. Others such as Verdoolaege and Grosser [Verdoolaege *et al.* 2013] handle C code but are limited to arrays with size known at compile time. The algorithms used by proprietary software like R-Stream, HMPP, or PGI Accelerator are unknown, but they are most likely based on the same kind of scheme.

The proposed tool that comes along with this thesis, Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] (see detailed presentation in Section 5.1), relies on the same scheme in its most basic version, relaxing this constraint by handling C99 Variable Length Array (VLA). The effective size is then known only at runtime but the information is available symbolically at compile time.

Some polyhedral automatic tools do not consider this problem at all. While converting and automatically optimizing loop nests written in C code into CUDA or OpenCL kernels, they rely on the programmer to generate the host code. This is the case at least for

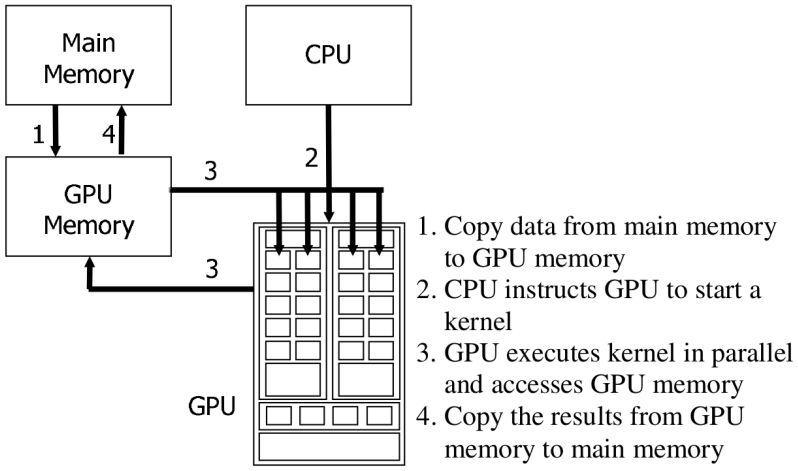


Figure 3.7: Basic process for mapping data to the accelerator (source [Yan *et al.* 2009], ©2011 Springer-Verlag)g.

Baskaran et al. [Baskaran *et al.* 2010].

The most common optimization at this level is local to each kernel. It consists in sending to the accelerator only the data that are used and to get back only the data that are defined. This can be done automatically as in PPCG [Verdoolaege *et al.* 2013] or directive hints given by the programmer as in JCUDA, HMPP, PGI Accelerator, or hiCUDA [Han & Abdelrahman 2009].

This basic process is illustrated below using as an example the first step of Stars-PM main iteration, the function `discretization()`. Figure 3.8 shows the sequential code of this function in its initial version.

The loop nest is detected as parallel and selected to be promoted as a kernel. The mapping on the accelerator is performed according to the technique presented forward in Section 4.2. The loop body is outlined to a new function that will be executed by the accelerator, and the loop nest is replaced by a call to a kernel launch function. Memory transfers are generated according to the basic technique introduced in this section. The resulting code is shown in Figure 3.9.

Looking at the original loop nest, it is clear that the `pos` array is used in the kernel, whereas the `data` array is written. Therefore two transfers have to be generated as can be seen in Figure 3.9. The first one ensures that `pos` are moved to the accelerator before kernel execution while the second one gets the data back into the host memory after the kernel execution.

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]){
  int i, j, k;
  float x, y, z;
  for (i = 0; i < NP; i++)
    for (j = 0; j < NP; j++)
      for (k = 0; k < NP; k++) {
        x = pos[i][j][k].x;
        y = pos[i][j][k].y;
        z = pos[i][j][k].z;
        data[i][j][k] = (int)(x/DX)*NP*NP
                      + (int)(y/DX)*NP
                      + (int)(z/DX);
      }
}
```

Figure 3.8: Sequential source code for function `discretization`, the first step of each Stars-PM simulation main iteration.

3.4 Region Refinement Scheme

This section introduces refinement of the basic scheme based on array declarations from the previous section using the convex array regions in Section 3.2. It also illustrates informally the process of *statement isolation* described formally in Guelton's PhD thesis [Guelton 2011b]. It turns a statement s into a new statement Isol(s) that shares no memory area with the remainder of the code, and is surrounded by the required memory transfers between the two memory spaces. In other words, if s is evaluated in a memory state function, σ, Isol(s) does not reference any element of Domain(σ). The generated memory transfers to and from the new memory space ensure the consistency and validity of the values used in the extended memory space during the execution of Isol(s) and once again, back to the original execution path.

To illustrate how the convex array regions are leveraged, the `while` loop in Figure 3.5 is used as an example. The exact and over-approximated array regions for this statement are as follows:

$$\mathcal{R} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{R}}(\mathrm{randv}) = \{\mathrm{randv}[\phi_1] \mid N - 3 \leq 4 \times \phi_1; 3 \times \phi_1 \leq N\}$$
$$\mathcal{W} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{W}}(\mathrm{a}) = \{\mathrm{a}[\phi_1] \mid N - 3 \leq 4 \times \phi_1; 12 \times \phi_1 \leq 5 \times N + 9\}$$

```
void discretization(coord pos[NP][NP][NP],int data[NP][NP][NP]) {
 // Declare pointers to buffers on accelerator
 coord (*pos0)[NP][NP][NP] = NULL;
 int (*data0)[NP][NP][NP] = NULL;

 // Allocate buffers on the GPU
 P4A_accel_malloc((void **) &data0, sizeof(int)*NP*NP*NP);
 P4A_accel_malloc((void **) &pos0, sizeof(coord)*NP*NP*NP);

 // Copy the input data to the GPU
 P4A_copy_to_accel(sizeof(coord)*NP*NP*NP, pos, *pos0);

 // Launch the kernel
 P4A_call_accel_kernel_2d(discretization_kernel,NP,NP,*pos0,*data0);

 // Copy the result back from the GPU
 P4A_copy_from_accel(sizeof(int)*NP*NP*NP, data, *data0);

 // Free GPU buffers
 P4A_accel_free(data0);
 P4A_accel_free(pos0);
}

// The kernel corresponding to loop-nest body
P4A_accel_kernel discretization_kernel( coord *pos, int *data ) {
   int k; float x, y, z;
   int i = P4A_vp_1; //  P4A_vp_* are mapped from CUDA BlockIdx.*
   int j = P4A_vp_0; //  and ThreadIdx.* to loop indices

   // Iteration clamping to avoid GPU iteration overrun
   if (i<=NP&&j<=NP)
      for(k = 0; k < NP; k += 1) {
         x = (*(pos+k+NP*NP*i+NP*j)).x;
         y = (*(pos+k+NP*NP*i+NP*j)).y;
         z = (*(pos+k+NP*NP*i+NP*j)).z;
         *(data+k+NP*NP*i+NP*j) = (int)(x/DX)*NP*NP
                                + (int)(y/DX)*NP
                                + (int)(z/DX);
      }
}
```

Figure 3.9: Code for function `discretization` after automatic GPU code generation.

```
void foo(int N, int a[N], int randv[N]) {
  int x=0,y=0;
  int A[N/6], RANDV[(N-9)/12], X, Y;
  memcpy(A, a+(N-3)/4, N/6*sizeof(int));                    //(1)
  memcpy(RANDV, randv+(N-3)/4, (N-9)/12*sizeof(int));       //(2)
  memcpy(&X, &x, sizeof(x)); memcpy(&Y, &y, sizeof(y));     //(3)
  while(X<=N/3) {
    A[X+Y-(N-3)/4] = X+Y;
    if (RANDV[X-Y-(N-3)/4]) X = X+2; else X++,Y++;
  }
  memcpy(a+(N-3)/4, A, N/6*sizeof(int));                    //(4)
  memcpy(&x, &X, sizeof(x)); memcpy(&y, &Y, sizeof(y));     //(5)
}
```

Figure 3.10: Isolation of the irregular `while` loop from Figure 3.5 using array region analysis.

The basic idea is to turn each region into a newly allocated variable, large enough to hold the region, then to generate data transfers from the original variables to the new ones, and finally to perform the required copy from the new variables to the original ones. This results in the code shown in Figure 3.10, where isolated variables have been put in uppercase. Statements `(3)` and `(5)` correspond to the exact regions on scalar variables. Statements `(2)` and `(4)` correspond to the over-approximated regions on array variables. Statement `(1)` is used to ensure data consistency, as explained later.

Notice how `memcpy` system calls are used here to simulate data transfers, and, in particular, how the sizes of the transfers are constrained with respect to the array regions.

The benefits of using new variables to simulate the extended memory space and of relying on a regular function to simulate the DMA are twofold:

1. The generated code can be executed on a general-purpose processor. It makes it possible to verify and validate the result without the need of an accelerator or a simulator.
2. The generated code is independent of the hardware target: specializing its implementation for a given accelerator requires only a specific implementation of the memory transfer instructions (here `memcpy`).

3.4.1 Converting Convex Array Regions into Data Transfers

From this point on, the availability of data transfer operators that can transfer rectangular subparts of n-dimensional arrays to or from the accelerator is assumed. For instance,

```
size_t memcpy2d(void* dest, void* src,
                size_t dim1, size_t offset1, size_t count1,
                size_t dim2, size_t offset2, size_t count2);
```

copies from `src` to `dest` the rectangular zone between (`offset1`, `offset2`) and (`offset1`+`count1`, `offset2`+`count2`). `dim1` and `dim2` are the sizes of the memory areas pointed to by `src` and `dest` on the host memory, and are used to compute the addresses of the memory elements to transfer.

We show how convex array regions are used to generate calls to these operators. Let src be a n-dimensional variable, and $\{src[\phi_1]\dots[\phi_n] \mid \psi(\phi_1,\dots,\phi_n)\}$ be a convex region of this variable.

As native DMA instructions are very seldom capable of transferring anything other than a rectangular memory area, the rectangular hull, denoted $\lceil\cdot\rceil$, is first computed so that the region is expressed in the form

$$\{src[\phi_1]\dots[\phi_n] \mid \alpha_1 \leq \phi_1 < \beta_1,\dots,\alpha_n \leq \phi_n < \beta_n\}$$

This transformation can lead to a loss of accuracy and the region approximation can thus shift from *exact* to *may*. This shift is performed when the original region is not equal to its rectangular envelope.

The call to the transfer function can then be generated with `offset`k $= \alpha_k$ and `count`k $= \beta_n - \alpha_k$ for each k in $[1\dots n]$.

For a statement s, the memory transfers from σ are generated using its read regions ($\mathcal{R}(s,\sigma)$): any array element read by s must have an up-to-date value in the extended memory space with respect to σ. Symmetrically, the memory transfers back to σ must include all updated values, represented by the written regions ($\mathcal{W}(s,\sigma')$), where σ' is the memory state once s is executed from σ.[2]

However, if the written region is over-approximated, part of the values it contains may not have been updated by the execution of Isol(s). Therefore, to guarantee the consistency of the values transferred *back* to σ, they must first be correctly initialized during the transfer *from* σ. These observations lead to the following equations for the convex array

2. Most of the time, variables used in the region description are not modified by the isolated statement and we can safely use $\mathcal{W}(s,\sigma)$. Otherwise, e.g. `a[i++]=1`, methods detailed in [Creusillet & Irigoin 1996b] must be applied to express the region in the right memory state.

regions transferred from and to σ, respectively denoted $Load(s,\sigma)$ and $Store(s,\sigma)$:

$$\begin{aligned} Store(s,\sigma) =& \lceil \overline{\mathcal{W}}(s,\sigma) \rceil \\ Load(s,\sigma) =& \lceil \overline{\mathcal{R}}(s,\sigma) \cup (Store(s,\sigma) - \underline{\mathcal{W}}(s,\sigma)) \rceil \end{aligned}$$

$Load(s,\sigma)$ and $Store(s,\sigma)$ are rectangular regions by definition and can be converted into memory transfers, as detailed previously. The new variables with *ad-hoc* dimensions are declared and a substitution taking into account the shifts is performed on s to generate Isol(s).

3.4.2 Managing Variable Substitutions

For each variable `v` to be transferred according to $Load(s,\sigma)$, a new variable `V` is declared, which must contain enough space to hold the loaded region. For instance if `v` holds short integers and

$$Load(s,\sigma) = \{\mathtt{v}[\phi_1][\phi_2] \mid \alpha_1 \leq \phi_1 < \beta_1, \alpha_2 \leq \phi_2 < \beta_2\}$$

then V will be declared as `short int V`$[\beta_1-\alpha_1][\beta_2-\alpha_2]$. The translation of an intraprocedural reference to `v` into a reference to `V` is straightforward as $\forall i,j,\ \mathtt{V}[i][j] = \mathtt{v}[i+\alpha_1][j+\alpha_2]$.

The combination of this variable substitution with convex array regions is what makes the isolate statement a powerful tool: all the complexity is hidden by the region abstraction. For instance, once the regions of the switch case in Figure 3.6 are computed as

$$\begin{aligned} \mathcal{R}(\mathrm{c}) &= \{\mathrm{c}\} & \mathcal{R}(\mathrm{i}) &= \{\mathrm{i}\} \\ \mathcal{W}(\mathrm{out}) &= \{\mathrm{out}[\phi_1] \mid \phi_1 = i\} & \overline{\mathcal{R}}(\mathrm{in}) &= \{\mathrm{in}[\phi_1] \mid i \leq \phi_1 \leq i+2\} \end{aligned}$$

the data transfer generation and variable substitutions lead to the isolated code given in Figure 3.11. The complexity of the isolated statement does not matter as long as it has been modeled by the convex array region analysis.

For interprocedural translation, a new version of the called function is created using the following scheme: for each transferred variable passed as an actual parameter, and for each of its dimensions, an extra parameter is added to the call and to the new function, holding the value of the corresponding offset. These extra parameters are then used to

```
void foo(int n, int i, char c, int out[n], int in[n]) {
  char C; int OUT[1], IN[3], I;
  memcpy(&I,&i,sizeof(int));
  memcpy(&C,&c,sizeof(char));
  memcpy(IN, in+i, sizeof(int)*3);
  switch(C) {
    case 'a':
    case 'e':
      OUT[I-I]=IN[I-I];
      break;
    default:
      OUT[I]=IN[3*(I/3)+2-I];
  }
  memcpy(out+i, OUT, sizeof(int));
}
```

Figure 3.11: Code with a `switch` case from Figure 3.6 after isolation.

```
void fir( int n, int k, int src[n], int dst[n-k], int m[k]) {
  int N=n - k+ 1;
  for( int i = 0; i < N; ++i ) {
    int DST[1],SRC[k],M[k];
    memcpy(SRC, src+i, k*sizeof(int));
    memcpy(M, m+0, k*sizeof(int));
    KERNEL(i, n, k, SRC, DST, M, i/*SRC*/, i/*DST*/, 0/*M*/);
    memcpy(dst, DST+0, 1*sizeof(int));
  }
}
```

Figure 3.12: Interprocedural isolation of the outermost loop of a Finite Impulse Response.

perform the translation in the called function.

The output of the whole process applied to the outermost loop of the Finite Impulse Response (FIR) is illustrated in Figure 3.12, where a new `KERNEL` function with two extra parameters is now called instead of the original `kernel` function. These parameters hold the offsets between the original array variables `src` and `m` and the isolated ones `SRC` and `M`.

The body of the new `KERNEL` function is given in Figure 3.13. The extra offset parameters are used to perform the translation on the array parameters. The same scheme applies for multidimensional arrays, with one offset per dimension.

```
void KERNEL(int i, int n, int k, int SRC[k], int DST[1], int M[k],
            int SRC_offset, int DST_offset, int M_offset) {
 int v=0;
 for( int j = 0; j < k; ++j )
  v += SRC[i+j-SRC_offset]*M[j-M_offset];
 DST[i-SRC_offset]=v;
}
```

Figure 3.13: Isolated version of the `KERNEL` function of the Finite Impulse Response (see Figure 3.4).

3.5 Limits

Data exchanges between host and accelerator are performed as DMA transfers between Random Access Memory (RAM) memories across the PCI Express bus, which currently offers a theoretical bandwidth of 8 GB/s. This is really small compared to the GPU inner memory bandwidth, which exceeds often 150 GB/s. This low bandwidth can annihilate all gains obtained when offloading computations in kernels, unless they are really compute-intensive.

With the available hardware (see Section 7.1), up to 5.6 GB/s was measured from the host to the GPU, and 6.2 GB/s back. This throughput is obtained for blocks of a few tens of MB, but decreases dramatically for smaller blocks. Moreover, this bandwidth is reduced by more than half when the transferred memory areas are not *pinned* ; i.e. subject to paging by the virtual memory manager of the operating system. Figure 3.14 illustrates this behavior.

Using as reference a cube with 128 cells per edge and as many particles as cells, for a function like `discretization`, one copy to the GPU for particle positions is a block of 25 MB. After execution, one copy back from the GPU for the particle-to-cell association is an 8 MB block.

The communication time for these two copies is about 5 ms. Recent GPUs offer ECC hardware memory error checking that more than doubles time needed for the same copies to 12 ms. Moreover, each buffer allocation and deallocation require 10 ms. In comparison, kernel execution for `discretization` and this problem size requires only 0.37 ms on the GPU, but 37 ms on the CPU.

Note that memory transfers and buffer allocations represent the largest part of the total execution time for the discretization step, and therefore the highest potential for obtaining accelerations. This is why the next section exposes a static interprocedural optimization

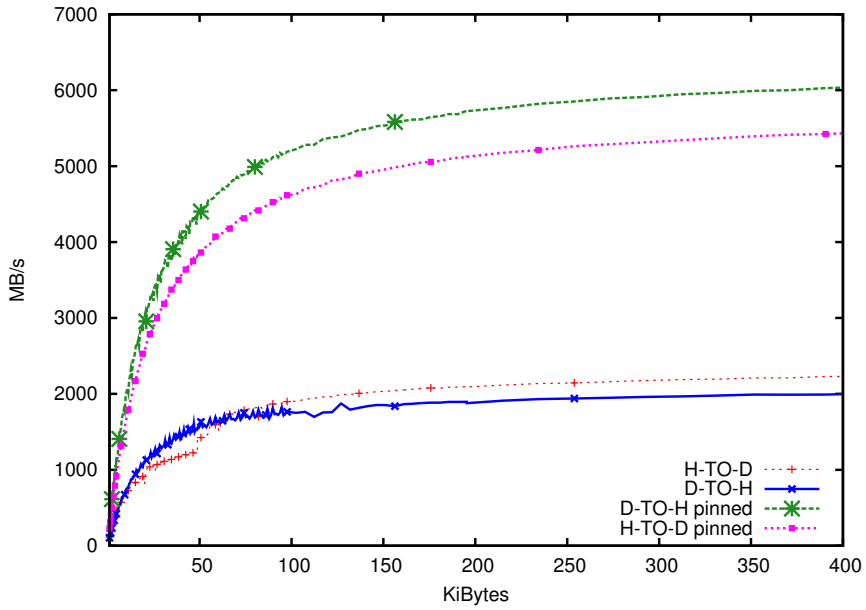


Figure 3.14: Bandwidth for memory transfers over the PCI-Express 2.0 bus as a function of block size. Results are shown for transfers from the host to the GPU (H-TO-D) and in the opposite direction (D-TO-H), each for pinned or standard allocated memory.

to map data transfers more efficiently.

3.6 Communication Optimization Algorithm

Much work has been done regarding communication optimization for distributed computers. Examples include message fusion in the context of Single Program Distributed Data (SPDD) [Gerndt & Zima 1990], data flow analysis based on array regions to eliminate redundant communications and to overlap the remaining communications with computations operations [Gong *et al.* 1993], and distribution in the context of High Performance Fortran (HPF) compilation [Coelho 1996, Coelho & Ancourt 1996].

Similar methods are applied in this section to offload computation in the context of a host–accelerator relationship and to integrate in a parallelizing compiler a transformation that limit the amount of CPU–GPU communications at compile time.

This section introduces a new data flow analysis designed to drive the static generation of memory transfers between host and accelerator. The main scheme is first presented and the intraprocedural algorithm is detailed. Then the interprocedural extension of the algo-

rithm is presented. The metric used to evaluate the scheme is introduced and experiments are performed using a 12-core Xeon multiprocessor machine with a Nvidia Tesla GPU C2050. The proposed solution is evaluated on well-known benchmarks [Pouchet 2011, Che *et al.* 2009] before showing how it scales with the real numerical cosmological simulation Stars-PM.

It is assumed for this section that the memory of the GPU is large enough to handle the arrays to process. While this assumption can represent an unacceptable constraint for some workloads, like those encountered when dealing with out-of-core computing, the many gigabytes of memory embedded in modern GPUs are large enough for a wide range of simulations.

3.6.1 A New Analysis: Kernel Data Mapping

At each time step, the function `iteration` (see in Figure 3.3) uses data defined by the previous step. The parallelized code performs many transfers to the GPU followed immediately by the opposite transfer.

Our simulation (see in Figure 3.2) presents the common pattern of data dependencies between loop iterations, where the current iteration uses data defined during previous ones. Such data should remain on the GPU, with copies back to the host only as needed for checkpoints and final results.

A new analysis is introduced for the compiler middle-end to generate efficient host–GPU data copying. The host and the accelerator have separated memory spaces; my analysis annotates internally the source program with information about the locations of up-to-date copies: in host memory and/or GPU memory. This allows a further additional transformation to statically determine interesting places to insert asynchronous transfers with a simple strategy: Launch transfers from host to GPU as early as possible and launch those from GPU back to host as late as possible, while still guaranteeing data integrity.

Additionally, launching transfers inside loops is avoided whenever possible. A heuristic is used to place transfers as high as possible in the call graph and in the Abstract Syntax Tree (AST). PIPS uses a hierarchical control flow graph [Irigoin *et al.* 1991, Amini *et al.* 2011a (perso)] to preserve as much as possible of the AST. However, to simplify the presentation of the analyses, equations are written in a classical way assuming that a traditional Control Flow Graph (CFG) is available.

The sets used in the data flow analysis are first introduced. Then the equations used for intraprocedural construction are presented, before extending them to interprocedural

construction. Finally the generation of transfers and the lightweight runtime involved to support the copy process are illustrated.

3.6.2 Definitions

The analysis computes the following sets for each statement:

- $\mathcal{U}$ is the set of arrays known to be *u*sed next ($>$) by the accelerator;
- $\mathcal{D}$ is the set of arrays known to be last ($<$) *d*efined by the accelerator, and not used on the host in the meantime;
- $\mathcal{T}_{H\to A}$ is the set of arrays to transfer to the accelerator memory space immediately after the statement;
- $\mathcal{T}_{A\to H}$ is the set of arrays to transfer from the accelerator to the host immediately before the statement.

These sets are initially empty for every statement. Note that even if array regions are used in the following equations to compute these sets, the granularity is the array. Data-flow equations presented in the next sections describe the computation of these sets on the control-flow graph of the global program. Let S denotes a statement of the program. It can be complex but in order to simplify in the following it is considered that statements are assignments or function calls. A call to a kernel on the GPU is handled through different equations. Such a statement is denoted S_k. The control-flow graph is represented with pred(S) for the set of statements that can precede immediately S at execution. Symmetrically, succ(S) stands for the set of statements that can be executed immediately after S.

As explained in Section 3.2, PIPS computes array regions. These analyses produce fine grained resources; these local fine grained pieces of information are used to build a coarse grained analysis in which arrays are represented atomically. Therefore the equations presented in the following do not require a deep understanding of array regions. The interested reader is referred to Béatrice Creusillet's PhD thesis [Creusillet & Irigoin 1996b] for more information.

In the equations below, the arrays totally or partially written by a statement S are denoted $\mathcal{W}(S)$. Similarly, the arrays read by S are denoted $\mathcal{R}(S)$.

When S is a function call, the set represents the summary of the function, i.e., the set of effects that can be seen on function parameters and on global variables.

Moreover, $\mathcal{OUT}(S_k)$ represents the set of arrays modified by a kernel for which PIPS established that they are alive, i.e., their value is potentially used by a later statement in the program. By contrast, $\mathcal{IN}(S_k)$ stands for the set of arrays consumed by the kernel, i.e., those for which a value is read without being previously produced in the kernel.

3.6.3 Intraprocedural Phase

The analysis begins with the set $\mathcal{D}$ in a forward pass through the control flow graph. An array is defined on the GPU for a statement S iff it is also the case for all its immediate predecessors in the control flow graph and if the array is not used or defined by the host, i.e., is not in the set $\mathcal{R}(S)$ or $\mathcal{W}(S)$ computed by PIPS:

$$\mathcal{D}(S) = \left(\bigcap_{S' \in \text{pred}(S)} \mathcal{D}(S') \right) - \mathcal{R}(S) - \mathcal{W}(S) \tag{3.1}$$

The initialization is performed by the first kernel call S_k with the arrays defined by the kernel k and used later, $\mathcal{OUT}(S_k)$. The following equation is involved at each kernel call site:

$$\mathcal{D}(S_k) = \mathcal{OUT}(S_k) \bigcup \left(\bigcap_{S' \in \text{pred}(S_k)} \mathcal{D}(S') \right) \tag{3.2}$$

A backward pass is then performed in order to compute $\mathcal{U}$. For a statement S, an array has its next use on the accelerator iff it is also the case for all statements immediately following in the control flow graph, and if it is not defined by S.

$$\mathcal{U}(S) = \left(\bigcup_{S' \in \text{succ}(S)} \mathcal{U}(S') \right) - \mathcal{W}(S) \tag{3.3}$$

As above with $\mathcal{D}$, $\mathcal{U}$ is initially empty and is first initialized at kernel call sites with the arrays necessary to run the kernel, $\mathcal{IN}(S_k)$, and the arrays defined by the kernel, $\mathcal{W}(S_k)$. These defined arrays have to be transferred to the GPU if it cannot be proved that they are written entirely by the kernel. Otherwise, if not all the values have been updated in

the GPU memory, the transfer may overwrite still-valid data on the CPU when copying back the array from the GPU after kernel execution:

$$\mathcal{U}(S_k) = \mathcal{IN}(S_k) \bigcup \mathcal{W}(S_k) \bigcup \left(\bigcup_{S' \in \text{succ}(S_k)} \mathcal{U}(S') \right) \tag{3.4}$$

An array must be transferred from the accelerator to the host after a statement S iff its last definition is in a kernel and if it is not the case for at least one of the immediately following statements:

$$\mathcal{T}_{A \to H}(S) = \mathcal{D}(S) - \bigcap_{S' \in \text{succ}(S)} \mathcal{D}(S') \tag{3.5}$$

This set is used to generate a copy operation at the latest possible location.

An array must be transferred from the host to the accelerator if it has a next use on the accelerator. In order to perform the communication at the earliest, its launch is placed immediately after the statement that defines it, i.e., the statement whose $\mathcal{W}(S)$ set contains it. The following equation applies for any S which is not a kernel call.

$$\mathcal{T}_{H \to A}(S) = \mathcal{W}(S) \bigcap \left(\bigcup_{S' \in \text{succ}(S)} \mathcal{U}(S') \right) \tag{3.6}$$

3.6.4 Interprocedural Extension

Kernel calls are potentially localized deep in the call graph. Consequently, a reuse between kernels requires interprocedural analysis. The function `iteration` (see in Figure 3.3) illustrates this situation: each step corresponds to one or more kernel executions.

My approach is to perform a backward analysis on the call graph. For each function f, *summary* sets $\overline{\mathcal{D}}(f)$ and $\overline{\mathcal{U}}(f)$ are computed. They summarize information about the formal parameters of the function and the global variables. These sets can be viewed as contracts. They specify a data mapping that the call site must conform to. All arrays present in $\overline{\mathcal{U}}(f)$ must be transferred to the GPU before the call, and all arrays defined in $\overline{\mathcal{D}}(f)$ must be transferred back from the GPU before any use on the host.

These sets are required in the computation of $\mathcal{D}$ and $\mathcal{U}$ when a call site is encountered. Indeed, at a call site c for a function f, each argument of the call that corresponds to a formal parameter present in $\overline{\mathcal{U}}$ must be transferred to the GPU before the call, because we know that the first use in the called function occurs in a kernel. Similarly, an argument that is present in $\overline{\mathcal{D}}$ has been defined in a kernel during the call and not already transferred back when the call ends. This transfer can be scheduled later, but before any use on the host.

Equations 3.1 and 3.3 are modified for a call site by adding a translation operator, $\text{trans}_{\text{f}\to\text{c}}$, between arguments and formal parameters:

$$\mathcal{D}(c) = \left[\text{trans}_{\text{f}\to\text{c}}(\overline{\mathcal{D}}(f)) \bigcup \left(\bigcap_{S' \in \text{pred}(c)} \mathcal{D}(S') \right)\right] - \mathcal{R}(c) - \mathcal{W}(c) \tag{3.7}$$

$$\mathcal{U}(c) = \left[\text{trans}_{\text{f}\to\text{c}}(\overline{\mathcal{U}}(f)) \bigcup \left(\bigcup_{S' \in \text{succ}(c)} \mathcal{U}(S') \right)\right] - \mathcal{W}(c) \tag{3.8}$$

The whole process implied by these equations is shown in Figure 3.15.

In the code in Figure 3.16, comparing the result of the interprocedural optimized code with the very local approach of Figure 3.9 shows that all communications and memory management instructions (allocation/deallocation) have been eliminated from the main loop.

3.6.5 Runtime Library

Our compiler Par4All includes a lightweight runtime library that lets the generated code be independent from the target (currently OpenCL and CUDA). Par4All also supports common functions such as memory allocation at kernel call sites and memory transfer sites. The runtime relies on a hash table that maps host addresses to GPU addresses. This hash table allows flexibility in the handling of the memory allocations. Using it, the user call sites and function signatures can be preserved, avoiding more advanced and heavy transformations, i.e., duplicating the function arguments for the arrays in the whole call graph and at all call sites to carry the CPU and GPU addresses.

The memory management in the runtime does not free the GPU buffers immediately after they have been used, but preserves them as long as there is enough memory on the

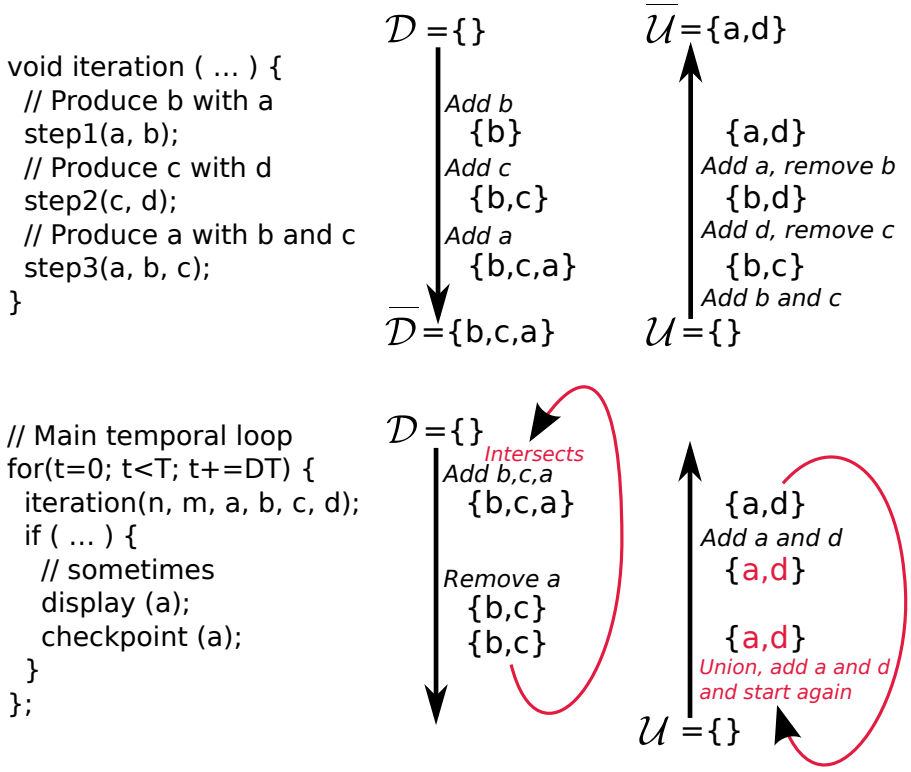


Figure 3.15: Illustration of set construction using the intraprocedural analysis on the function `iteration`. The different calls to `step` functions use and produce data on the GPU via kernel calls. Sometimes in the main loop, array `a` is read to display or to checkpoint. The interprocedural translation exploits at call site the summary computed on function `iteration`. A fix point is sought on the loop.

GPU. When a kernel execution requires more memory than is available, the runtime frees some buffers. The policy used for selecting a buffer to free can be the same as for cache and virtual memory management, for instance Least Recently Used (LRU) or Least Frequently Used (LFU).

This behavior requires updating hardware caches in Symmetric MultiProcessing (SMP) with protocols such as MESI. The scheme involved keeps a copy of the data up to date in the CPU and the accelerator memory at the same time. When the host or the accelerator writes data, the copy in the other one is invalidated and a transfer may be scheduled if necessary.

The calls to the runtime that retrieves addresses in the accelerator memory space for arrays `pos` and `data` can be noticed in Figure 3.16. If the arrays are not already allocated

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]) {
  //generated variable
  coord *pos0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(coord));
  int *data0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(int));
  // Call kernel
  P4A_call_accel_kernel_2d(discretization_kernel,
                           NP, NP, pos0, data0);
}
int main(int argc, char *argv[]) {
  // Read data from input files
  init_data(argv[1], ....);
  P4A_runtime_copy_to_accel(pos, ...*sizeof(...));
  // Main temporal loop
  for(t = 0; t < T; t+=DT)
    iteration(...);
  // Output results to a file
  P4A_runtime_copy_from_accel(pos, ...*sizeof(...));
  write_data(argv[2],....);
}
```

Figure 3.16: Simplified code for functions `discretization` and `main` after interprocedural communication optimization.

in the accelerator, a lazy allocation is done the first time. The code is lighter than the previous version shown in Figure 3.9, and easier to generate from the compiler point of view.

3.7 Sequential Promotion

Two parallel loop nests can be separated by some sequential code. When this sequential code uses or produces the data involved in the parallel computations, transfers may occur between the host memory and the accelerator memory.

A solution to this issue is to promote the sequential code as *parallel*, with only one thread that executes it. Using one thread on the GPU is totally inefficient. However, the slowdown can be dramatically lower than the cost of communication if the code is small enough. This issue is similar to the decision about the profitability whether or not to offload a kernel to the accelerator that is discussed more generally in Section 5.6 page 156.

The `gramschmidt` example mentioned in the previous section is shown in Figure 3.17.

```
for (k = 0; k < n; k++) {
  // The following is sequential
  nrm = 0;
  for (i = 0; i < m; i++)
    nrm += A[i][k] * A[i][k];
  R[k][k] = sqrt(nrm);

  // The following is parallel
  for (i = 0; i < m; i++)
    Q[i][k] = A[i][k] / R[k][k];
  for (j = k + 1; j < n; j++) {
    R[k][j] = 0;
    for (i = 0; i < m; i++)
     R[k][j] += Q[i][k] * A[i][j];
    for (i = 0; i < m; i++)
     A[i][j] = A[i][j] - Q[i][k] * R[k][j];
  }
}
```

```
kernel_0(R, n);
copy_from_accel(R);
for(k = 0; k <= n-1; k += 1) {
  // Sequential
  nrm = 0;
  for(i = 0; i <= m-1; i += 1)
    nrm += A[i][k]*A[i][k];
  R[k][k] = sqrt(nrm);

  // Parallel region
  copy_to_accel(R);
  kernel_1(A, Q, R, k, m);
  kernel_2(A, Q, R, k, m, n);
  kernel_3(A, Q, R, k, m, n);
  copy_from_accel(A);
  copy_from_accel(R);
}
```

(a) Usual Host code.

```
copy_to_accel(A);
kernel_0(R, n);
for(k = 0; k <= n-1; k += 1) {
  // Sequential code promoted
  // on the GPU
  sequential_kernel(A,R,m,k);

  // No more
  // transfers
  // here
  kernel_1(A, Q, R, k, m);
  kernel_2(A, Q, R, k, m, n);
  kernel_3(A, Q, R, k, m, n);
}
// transfers is outside of the loop
copy_from_accel(A);
```

(b) Host code after sequential promotion.

Figure 3.17: `gramschmidt` example taken from Polybench suite. The first part of the loop body is sequential while the following are parallel loop nests. The sequential promotion on the GPU avoids costly memory transfers.

The codes generated with and without sequential promotion illustrate how this trade-off can reduce the communication. The difficulty is to evaluate the trade-off. This depends on both the GPU PEs' speed and the PCIe bandwidth.

Section 7.8.4, page 201, contains measurements showing up to eight times speedup over the optimized scheme for the `gramschmidt` example, but also up to thirty-seven times speedup for the `durbin` example from the Polybench suite.

In case of inaccurate evaluation, the performance can be dramatically degraded. This transformation requires a careful evaluation of the execution time of both versions. One possibility to overcome this issue is to perform an off-line profiling with just one iteration of the sequential loop on the GPU and then decide at runtime if the overhead is worth the transfers that must be performed. Such an approach is explored in Section 5.7, page 158, however I did not study how it can be mixed with the communication optimization scheme introduced in this chapter.

3.7.1 Experimental Results

Section 7.8, page 197, presents detailed experimental results for the optimizing scheme introduced in this chapter.

The first question is: what should we measure? While speedup are a very effective metric commercially speaking, in the context of this optimization it is biased because it is largely impacted by input parameters (see Section 7.8.1, page 197). The very same benchmark exhibits speedups ranging from 1.4 to fourteen just by changing the input parameters.

A more objective measurement for evaluating the proposed approach is the number of communications removed and the comparison with a scheme written by an expert programmer. Focusing on the speedup would also emphasize the parallelizer capabilities.

Using benchmarks from Polybench 2.0 suite and Rodinia, along with the Stars-PM numerical simulation introduced in Section 3.1, Section 7.8.2, page 199 illustrates the performance of the optimizing scheme using this metric, and shows that the optimized code performs almost as well as a hand-written code.

One noticeable exception is `gramschmidt`. Communications cannot be moved out of any loop due to data dependencies introduced by some sequential code. The parallel promotion scheme shown in Section 3.7 helps by accepting a more slowly generated code and allowing data to stay on the accelerator. This is still valuable while the slowdown is significantly smaller than the communication overhead. The difficulty for the compiler is to evaluate the

slowdown and to attempt parallelization only if optimized communications lead to a net performance increase. The result of this scheme, shown in Section 7.8.4, page 201, exhibits promising results with a speedup of up to thirty-seven, depending on the test case.

Finally Section 7.8.3, page 199, explores the performance impact of deferring the decision at runtime using the StarPU library; speedup of up to five times is obtained with the proposed static approach. Although StarPU is a library that has capabilities ranging far beyond the issue of optimizing communications, my static scheme is relevant.

3.8 Related Work

Among the compilers that I evaluated, none implement such an automatic static interprocedural optimization. While Lee et al. address this issue [Lee *et al.* 2009, § 4.2.3], their work is limited to liveness of data and thus quite similar to the unoptimized scheme proposed in Section 3.3. Leung addresses the case of a sequential loop surrounding a kernel launch and moves the communications out of the loop [Leung 2008].

The optimizing scheme proposed in this chapter is independent of the parallelizing scheme involved, and is applicable to systems that transform OpenMP in CUDA or OpenCL like OMPCUDA [Ohshima *et al.* 2010] or OpenMP to GPU [Lee *et al.* 2009]. It is also relevant for a directive-based compiler, such as JCUDA and hiCUDA [Han & Abdelrahman 2009]. It would also complete the work done on OpenMPC [Lee & Eigenmann 2010] by not only removing useless communications but moving them up in the call graph. Finally it would free the programmer of the task of adding directives to manage data movements in HMPP [Bodin & Bihan 2009] and PGI Accelerator [Wolfe 2010].

My approach can be compared to the algorithm proposed by Alias et al. [Alias *et al.* 2011, Alias *et al.* 2012b, Alias *et al.* 2012a]. This work studies, at a very fine grained level, the loading and unloading of data from memory for a tiled code running on a FPGA. My scheme optimizes at a coarse grained level and keeps the data on the accelerator as late as possible.

In a recent paper [Jablin *et al.* 2011], Jablin et al. introduce CGCM, a system targeting exactly the same issue. CGCM, just like my scheme, is focused on transferring full allocation units. While my granularity is the array, CGCM is coarser and considers a structure of arrays as a single allocation unit. While my decision process is fully static, CGCM makes decisions dynamically. It relies on a complete runtime to handle general pointers to the middle of any heap-allocated structure, which we do not support at this time.

I obtain similar overall speedup results, and I used the same input sizes. However, CGCM is not publicly available and the author does not provide us with a version. Therefore it has not been possible to reproduce their results and compare my solution in the same experimental conditions.

Jablin et al. measured a less-than-eight geometric mean speedup vs. mines of more than fourteen. However, a direct comparison of my measurement is hazardous. I used GNU C Compiler (GCC) while Jablin et al. used *Clang*, which produces a sequential reference code slower than GCC. I measured a slowdown of up to 20% on this benchmark set. Moreover, I made my measurements on a Xeon Westmere while they use an older Core2Quad Kentsfield. They generate their GPU version using a PTX generator for Low Level Virtual Machine (LLVM) while I used NVCC, the Nvidia compiler toolchain.

Finally, a key point is the scope on which the optimization is applied. Jablin et al. perform the optimization across the whole program and measured wall clock time, while I exclude the initialization functions from the scope of my compiler and exclude them from my measurements. Indeed, if I do not do so, the initializations of small benchmarks like the one in the Polybench suite would be parallelized and offloaded on the GPU, then no copy to the GPU would be required. Therefore I limit myself from optimization possibilities because I consider that this way is closer to what can be seen in real-world programs where initializations cannot usually be parallelized.

The overhead introduced by the runtime system in CGCM is thus impossible to evaluate by a direct comparison of the speedups obtained by my implementation.

3.8.1 Redundant Load-Store Elimination

Note that PIPS also includes another approach to the communication optimization issue that has been described formally in Guelton's thesis [Guelton 2011b]. This section informally describes how this approach uses step-by-step propagation of the memory transfers across the CFG of the host program. PIPS represents the program using a Hierarchical Control Flow Graph (HCFG): for example the statements that are part of a loop body are stored at lower level that the loop header. The representation is close to an AST. The main idea is the same as the one expressed earlier in Section 3.6, i.e., to move *load* operations upward in the HCFG so that they are executed as soon as possible, while *store* operations are symmetrically moved so that they are executed as late as possible. Redundant load-store elimination is performed in the meantime. For instance, *load*s and *store*s inside a loop may be moved outwards, which is similar to invariant code motion. But this

propagation is also performed interprocedurally, as data transfers are also moved outward function boundaries whenever possible.

3.8.1.1 Interprocedural Propagation

When a *load* is performed at the entry point of a function, it may be interesting to move it at the call sites. However, this is valid only if the memory state before the call site is the same as the memory state at the function entry point, that is, if there is no write effect during the effective parameter evaluation. In that case, the *load* statement can be moved before the call sites, after backward translation from formal parameters to effective parameters.

Similarly, if the same *store* statement is found at each exit point of a function, it may be possible to move it past its call sites. Validity criteria include that the *store* statement depends only on formal parameters and that these parameters are not written by the function. If this the case, the *store* statement can be removed from the function call and added after each call site after backward translation of the formal parameters.

3.8.1.2 Combining Load and Store Elimination

In the meanwhile, the intraprocedural and interprocedural propagation of DMA may trigger other optimization opportunities. *Load*s and *store*s may for instance interact across loop iterations, when the loop body is surrounded by a load and a store; or when a kernel is called in a function to produce data immediately consumed by a kernel hosted in another function, and the DMA have been moved in the calling function.

The optimization then consists in removing *load* and *store* operations when they are in direct sequence. This relies on the following property: considering that the statement denoted by "`memcpy(a,b,10*sizeof(in))`" is a DMA and its reciprocal is denoted by "`memcpy(b,a,10*sizeof(in))`", then in the sequence `memcpy(a,b,10*sizeof(in));memcpy(b,a,10*sizeof(in))`, the second call can be removed since it would not change the values already stored in `a`.

Figure 3.18, page 90, illustrates the result of the algorithm on an example taken from the PIPS validation suite. It demonstrates the interprocedural elimination of data communications represented by the `memload` and `memstore` functions. These function calls are first moved outside of the loop, then outside of the `bar` function; finally, redundant *load*s are eliminated.

```
void bar(int i, int j[2], int k[2]) {
  while (i-->=0) {
    memload(k, j, sizeof(int)*2);
    k[0]++;
    memstore(j, k, sizeof(int)*2);
  }
}
void foo(int j[2], int k[2]) {
  bar(0, j, k);
  bar(1, j, k);
}
```

⇓

```
void bar(int i, int j[2], int k[2]) {
  memload(k, j, sizeof(int)*2);  // moved outside of the loop
  while (i-->=0) k[0]++;
  memstore(j, k, sizeof(int));
}
```

⇓

```
void bar(int i, int j[2], int k[2]) {
   while (i-->=0) k[0]++;
}
void foo(int j[2], int k[2]) {
  memload(k, j, sizeof(int)*2); // load moved before call
  bar(0, j, k);
  memstore(j, k, sizeof(int)*2);// redundant load eliminated
  bar(1, j, k);
  memstore(j, k, sizeof(int)*2);// store moved after call
}
```

Figure 3.18: Illustration of the redundant load-store elimination algorithm.

3.9 Optimizing a Tiled Loop Nest

Alias et al. have published an interesting study about fine grained optimization of communications in the context of FPGA [Alias *et al.* 2011, Alias *et al.* 2012b, Alias *et al.* 2012a]. The fact that they target FPGAs changes some considerations on the memory size: FPGAs usually embed a very small memory compared to the many gigabytes available in a GPU board. The proposal from Alias et al. focuses on optimizing loads from Double Data Rate

```
for( int i = 0; i < N; ++i ) {
 memcpy(M,m,k*sizeof(int));
 memcpy(&SRC[i],&src[i],k*sizeof(int));
 kernel(i, n, k, SRC, DST, M);
 memcpy(&dst[i],&DST[i],1*sizeof(int));
}
```

(a) With naive communication scheme.

```
for( int i = 0; i < N; ++i ) {
 if(i==0) {
   memcpy(SRC,src,k*sizeof(int));
   memcpy(M,m,k*sizeof(int));
 } else {
   memcpy(&SRC[i+k-1],&src[i+k-1],1*sizeof(int));
 }
 kernel(i, n, k, SRC, DST, m);
 if(i<N-1) {
   memcpy(&dst[i],&DST[i],1*sizeof(int));
 } else {
   memcpy(&dst[i],&DST[i],1*sizeof(int));
 }
}
```

(b) After the inter-iterations redundant elimination.

Figure 3.19: Code with communication for FIR function presented in Figure 3.4.

(DDR) in the context of a tiled loop nest, where the tiling is done such that tiles execute sequentially on the accelerator while the computation inside each tile can be parallelized.

While their work is based on the Quasi-Affine Selection Tree (QUAST) abstraction, this section recalls how their algorithm can be used with the less expensive convex array region abstraction.

The classical scheme proposed to isolate kernels would exhibit full communications as shown in Figure 3.19a. An inter-iteration analysis allows avoiding redundant communications and produces the code shown in Figure 3.19b. The inter-iteration analysis is performed on a do loop, but with the array regions. The code part to isolate is not bound by static control constraints.

The theorem proposed for exact sets in [Alias *et al.* 2011, Alias *et al.* 2012b, Alias *et al.* 2012a] is the following: [3]

3. Regions are supposed exact here; the equation can be adapted to under- and over-approximations.

Theorem 3.1

$$Load(T) = \mathcal{R}(T) - \left(\mathcal{R}(t < T) \bigcup \mathcal{W}(t < T)\right) \tag{3.9}$$

$$Store(T) = \mathcal{W}(T) - \mathcal{W}(t > T) \tag{3.10}$$

where T represents a tile, $t < T$ represents the tiles scheduled for execution before the tile T, and $t > T$ represents the tiles scheduled for execution after T. The denotation $\mathcal{W}(t > T)$ corresponds to $\bigcup_{t>T} \mathcal{W}(t)$.

In Theorem 3.1, a difference exists for each loop between the first iteration, the last one, and the rest of the iteration set. Indeed, the first iteration cannot benefit from reuse from previously transferred data and has to transfer all needed data. In other words, $\mathcal{R}(t < T)$ and $\mathcal{W}(t < T)$ are empty for the first iteration while $\mathcal{W}(t > T)$ is empty for the last iteration.

For instance, in the code presented in Figure 3.19a, three cases are considered: $i = 0$, $0 < i < N - 1$ and $i = N - 1$.

Using the array region abstraction available in PIPS, a refinement with respect to the naive case can be carried out to compute each case, starting with the full region, adding the necessary constraints and performing a difference.

For example, the region computed by PIPS to represent the set of elements read for array `src`, is, for each tile (here corresponding to a single iteration `i`)

$$\mathcal{R}(i) = \{\mathrm{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1,\ 0 \leq i < N\}$$

For each iteration `i` of the loop except the first one (here $i > 0$), the region of `src` that is read minus the elements read in all previous iterations $i' < i$ has to be processed; that is, $\bigcup_{i'} \mathcal{R}(i' < i)$.

$\mathcal{R}(i' < i)$ is built from $\mathcal{R}(i)$ by renaming i as i' and adding the constraint $0 \leq i' < i$ to the polyhedron:

$$\mathcal{R}(i' < i) = \{\mathrm{src}[\phi_1] \mid i' \leq \phi_1 \leq i' + k - 1,\ 0 \leq i' < i,\ 1 \leq i < N\}$$

i' is then eliminated to obtain $\bigcup_{i'} \mathcal{R}(i' < i)$:

$$\bigcup_{i'} \mathcal{R}(i' < i) = \{\mathrm{src}[\phi_1] \mid 0 \leq \phi_1 \leq i + k - 2,\ 1 \leq i < N\}$$

The result of the subtraction $\mathcal{R}(i > 0) - \bigcup_{i'} \mathcal{R}(i' < i)$ is then the following region: [4]

$$Load(i > 0) = \{\text{src}[\phi_1] \mid \phi_1 = i + k - 1,\ 1 \leq i < N\}$$

This region is then exploited for generating the *load*s for all iterations but the first one. The resulting code after optimization is presented in Figure 3.19b. While the naive version loads $i \times k \times 2$ elements, the optimized version exhibits loads only for $i + 2 \times k$ elements.

3.10 Conclusion

With the increasing use of hardware accelerators, automatic or semi-automatic transformations assisted by directives take on an ever-greater importance.

The communication impact is critical when targeting hardware accelerators for massively parallel code like numerical simulations. Optimizing data movements is thus a key to high performance.

An optimizing scheme that addresses this issue has been designed and implemented in PIPS and Par4All.

The proposed approach has been validated against twenty benchmarks of the Polybench 2.0 suite, three from Rodinia, and on one real numerical simulation code. They are presented in Sections 3.1 and 7.2. It was found that the proposed scheme performs very close to a hand-written mapping in terms of number of communications.

As for future work, the cache management in the runtime can be improved further than a classic cache management algorithm because, unlike a hardware cache, the runtime that comes along the proposed optimizing scheme is software managed and can be dynamically controlled by the compiler inserting hints in the code. Indeed data flow analyses provide knowledge on the potential future course of execution of the program. This can be used in metrics to choose the next buffer to free from the cache. Buffers unlikely to be used again should be discarded first, while those that are certain to be used again should be freed last.

The execution times measured with multicore processors show that attention should be paid to work sharing between hosts and accelerators rather than keeping the host idle during the completion of a kernel. Multicore and multi-GPU configurations are another path to explore, with new requirements to determine accurate array region based transfers and computation localization.

4. As the write regions are empty for `src`, this corresponds to the loads.

Most of the work described in this chapter was published in [Amini *et al.* 2011b (perso), Amini *et al.* 2011c (perso), Guelton *et al.* 2012 (perso), Amini *et al.* 2012a (perso)].

The next chapter presents the different steps performed on the sequential input code to achieve parallelization and GPU code generation.

Chapter 4

Transformations for GPGPU

Contents

4.1 Introduction . . . 96
4.2 Loop Nest Mapping on GPU . . . 98
4.3 Parallelism Detection . . . 101
4.3.1 Allen and Kennedy . . . 102
4.3.2 Coarse Grained Parallelization . . . 103
4.3.3 Impact on Code Generation . . . 104
4.4 Reduction Parallelization . . . 105
4.4.1 Detection . . . 105
4.4.2 Reduction Parallelization for GPU . . . 109
4.4.3 Parallel Prefix Operations on GPUs . . . 111
4.5 Induction Variable Substitution . . . 111
4.6 Loop Fusion . . . 112
4.6.1 Legality . . . 113
4.6.2 Different Goals . . . 115
4.6.3 Loop Fusion for GPGPU . . . 116
4.6.4 Loop Fusion in PIPS . . . 118
4.6.5 Loop Fusion Using Array Regions . . . 124
4.6.6 Further Special Considerations . . . 126
4.7 Scalarization . . . 127
4.7.1 Scalarization inside Kernel . . . 128
4.7.2 Scalarization after Loop Fusion . . . 128
4.7.3 Perfect Nesting of Loops . . . 130
4.7.4 Conclusion . . . 131
4.8 Loop Unrolling . . . 132
4.9 Array Linearization . . . 133
4.10 Toward a Compilation Scheme . . . 134

The contributions of this chapter leverage some of the previously existing transformations in PIPS, extending some of them to handle C code, improving others for specific requirements of GPU code generation, and finally introducing new ones.

4.1 Introduction

The path leading from a sequential code to efficient parallel code for GPU includes many analyses and transformations. Moreover, some specificities of the input programs have to be taken into account. For instance, hand-written programs do not exhibit the same patterns as automatically generated code from high-level tools or languages. The code in Figure 4.1 shows how a three-line-long Scilab script ends up with temporary arrays and five loop nests.

The whole compilation scheme involved going from the sequential code down to the GPU code is presented in Figure 2.27 page 58 and identifies the contributions presented in this chapter.

Section 4.2 explains my scheme to map a parallel loop nest to a GPU, and how it leverages an implicit blocking and loop interchange at runtime for the NDRange and work-group decomposition (see Section 2.3.2.2, page 32).

I studied the combination of two different parallelizing algorithms, with analysis of the impact on code generation of both of them in Section 4.3.

I improved the existing reduction detection analysis to handle C code more accurately, and leveraged this analysis to enable parallelization of loops with reduction by improving the existing parallelization algorithms. I implemented a mapping scheme for some loops with reductions onto the GPU using atomic operations supported by OpenCL and CUDA (see Section 4.4, page 105). Actually, I proposed a new generic scheme for parallelizing loops with reduction, and implemented it in PIPS. It provides improvements for other targets like multicore using OpenMP code generation.

I designed and implemented a new induction variable substitution transformation based on linear precondition analysis (see Section 4.5). This transformation can enable the parallelization of loops that contain induction variables.

I implemented two loop fusion transformation phases: one based on the dependence graph and the other on array regions (see in Section 4.6). I designed heuristics to drive the fusion in order to target GPUs. This is particularly critical when processing code generated from high-level tools and languages, such as Scilab, which include many loops.

```
a = ones(1000,1000);
b = a+a+a
c = b*2.+3.
```

(a) Scilab source.

```
double a[1000][1000];
for(i=0; i<1000; i++)
 for(j=0; j<1000; j++)
  a[i][j] = (double) (1.0);

double t0[1000][1000];
for(i=0; i<1000; i++)
 for(j=0; j<1000; j++)
  t0[i][j] = a[i][j]+a[i][j];

double b[1000][1000];
for(i=0; i<1000; i++)
 for(j=0; j<1000; j++)
  b[i][j] = t0[i][j]+a[i][j];
disp_s0d2("b",1000,1000,b);

double t1[1000][1000];
for(i=0; i<1000; i++)
 for(j=0; j<1000; j++)
  t1[i][j] = b[i][j]*2.;

double c[1000][1000];
for(i=0; i<1000; i++)
 for(j=0; j<1000; j++)
  c[i][j] = t1[i][j]+3.;
disp_s0d2("c",1000,1000,c);
```

(b) Generated C code.

Figure 4.1: Example of a short Scilab program with the generated C file.

This transformation enables removing some temporary arrays generated by such tools.

I studied different array scalarization schemes in the context of GPGPU in Section 4.7, page 127, and I modified the PIPS implementation to match requirements for GPU code generation, especially to enforce the perfect nesting of loops.

Section 4.8 and 4.9 explore the impact of unrolling and array linearization.

Finally, Section 4.10 summarizes the contributions of this chapter and how they are connected together in the next chapter to form a complete compilation chain.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    computation statements
```

(a) Input code.

```
for (T=...)                    // Sequential, on the CPU
 parfor (P=pl(T) to pu(T))     // NDRange decomposition
   for (t=...)                 // Sequential, on the GPU
     parfor (p=...)            // Thread parallelism inside work-group
       computation statements
```

(b) After loop transformation for GPU mapping.

Figure 4.2: Example from Baghdadi et al. [Baghdadi *et al.* 2010] that illustrates how to tile a loop nest to map the GPU execution.

4.2 Loop Nest Mapping on GPU

Scheduling a parallel loop nest on a GPU using CUDA or OpenCL requires an elaborate mapping from the iteration set of the loop nest to the abstraction of the threaded GPU execution exhibited by NDRange (see Section 2.3.2.2 page 32).

Previous works [Baghdadi *et al.* 2010, Lee *et al.* 2009, Baskaran *et al.* 2010] made the compiler aware of the whole execution model hierarchy and tried to express it using nested loops. The transformations performed are principally multilevel tilings with some restructuring including loop interchanges or index set splittings. Figure 4.2 illustrates an example of how a loop nest is tiled to map the two-level GPU execution model.

The approach implemented in our Par4All compiler [Amini *et al.* 2012b (perso)] is quite different and does not expose any explicit multilevel tiling operation. Instead the source code generated by PIPS keeps a sequential semantics and is specialized at post-processing. Let us assume that loops are first normalized, i.e., that they start at zero and have an increment of one. This is to express the iteration set using the OpenCL concept of NDRange introduced in Section 2.3.2.2. Figure 4.3 gives the four steps included in this transformation. First, the body of the initial parallel loop nest in Figure 4.3 is outlined to a new function, the kernel executed by each thread on the GPU. The loop indices are rebuilt in the kernel using two macros `P4A_vp_`x for each virtual processor dimension. The sequential execution is performed with an expansion of `#define P4A_vp_1 ti` and `#define P4A_vp_0 tj`. The parallel loop nest is then annotated with the iteration set as shown in Figure 4.3c. In fact, the rectangular hull of the iteration set is represented, as it is not possible to be more

precise using either OpenCL or CUDA. [1]

Finally, a post-processing phase matches the annotation and contracts the loop nest to a pseudo-call to a `Call_kernel_`x`d()` macro with the NDRange dimension x ranging from one to three. The result of the contraction is shown in Figure 4.3d. This macro abstracts the parallel execution of the kernel on an $l \times m \times n$ grid of threads. The work-group size is not expressed in this abstraction and can then be chosen at runtime according to different parameters.

The proposed abstraction is used to expand the macro at compile time according to the effective target. CUDA, OpenCL, and OpenMP back ends have been implemented. The latter is particularly useful for debugging purposes, since the parallel execution of the kernel is emulated using CPU threads with dedicated buffers in the host memory to simulate the separate memory spaces.

As the iteration set is approximated by a rectangular hull, there may be more threads than necessary to execute the kernel. This situation can occurs in two cases, (1) because of the over-approximation of some triangular iteration sets for example, and (2) because the CUDA API requires the iteration set as a multiple of the work-group size. While the former can be detected at compile time, the latter is known only at runtime when the work-group size can be known. The iteration set is then systematically clamped using a guard, as shown in Figure 4.4.

A key point when targeting GPU is memory coalescing. To benefit from the memory bandwidth without suffering from the latency, consecutive threads in a block should access a contiguous memory area. This constraint is naturally respected when writing a code for the CPU. Programmers are taught to write loop nests in such a way that two consecutive iterations access contiguous array elements to exploit spatial locality in the caches. CUDA and OpenCL schedule consecutive threads along the first dimension of the work-group, then along the second dimension, and finally along the last one. Therefore the loop nest must be mapped with the innermost loop along the first work-group dimension. In the proposed representation, the mapping of threads to the work-group dimension is performed in the kernel with the index recovery shown in Figure 4.3b. The macros `P4A_vp_`x mask the dimension of the work-group along which the index implicitly iterates.

The tiling is implicit since each loop iteration set is potentially split according to the work-group size chosen. Again, the macros `P4A_vp_`x are involved to perform implicitly this transformation.

1. CUDA 5 and Kepler GPU, which should both be released by the end of 2012, bring a potential solution introducing what Nvidia calls *Dynamic Parallelism*.

```
// parallel
for (i=0; i<=n; i++)
 // parallel
 for (j=0; j<=m; j++) {
  // computation statements
  ...
 }
```

(a) Input code.

```
void kernel(int ti,int tj,...) {
    int i = P4A_vp_1;
    int j = P4A_vp_0;
    // computation statements
    ...
}

// parallel
for (ti=0; ti<=n; ti++)
 // parallel
 for (tj=0; tj<=m; tj++)
  kernel(ti,tj,...);
```

(b) Body outlined in a new function.

```
// Loop nest P4A begin,2D(n, n)
// parallel
for (ti=0; ti<=n; ti++)
 // parallel
 for (tj=0; tj<=m; tj++) {
  // Loop nest P4A end
  kernel(ti,tj,...);
 }
```

(c) Annotated loop nest iteration set.

```
Call_kernel_2d(n, m, kernel,...);
```

(d) The loop nest replaced by an abstract macro call.

Figure 4.3: Illustration of the successive steps performed to map a loop nest on the GPU.

```
void kernel(int ti,int tj,...) {
    int i = P4A_vp_1;
    int j = P4A_vp_0;
    if(i<n&&j<m) { // Guard
      // computation statements
      ...
    }
}
```

Figure 4.4: The iteration set is over-approximated with a rectangular hull; a guard is added to clamp it.

Algorithms	Dependence abstraction	Loop transformations
Allen–Kennedy [Allen & Kennedy 1987]	Dependence level Multiple Statements Nonperfect	Distribution
Wolf–Lam [Wolf & Lam 1991a]	Direction vectors One statement Perfect	Unimodular
Darte–Vivien [Darte & Vivien 1996a]	Polyhedra Multiple Statements Perfect	Shifted Linear
Feautrier [Feautrier 1992]	Affine (exact) Multiple Statements Nonperfect	Affine
Lim–Lam [Lim & Lam 1997]	Affine (exact) Multiple Statements Nonperfect	Affine

Table 4.1: A comparison of some loop parallel algorithms (from data published in [Boulet *et al.* 1998], nonexhaustive).

The expressiveness of these macros is limited, but still allows application of transformations that would otherwise require some work in the compiler. For instance, loop interchange, strip mining, or tiling can be achieve by using different macros.

This representation allows postponing some decisions about the transformations, therefore providing a code that is more target independent while simplifying the compiler internal representation.

4.3 Parallelism Detection

Parallelism detection is the foundation of our process. It consists in proving that a loop can be scheduled for a parallel execution. Such techniques are well known in the compiler community, at least since the hyperplane method by Lamport in 1974 [Lamport 1974].

Among all parallelizing algorithms, the most famous is certainly the one from Allen and Kennedy [Allen & Kennedy 1982, Allen & Kennedy 1987]. Darte et Vivien, and Boulet et al. survey existing parallelization algorithms [Darte & Vivien 1997, Boulet *et al.* 1998] and classify them according to the underlying dependence abstraction involved. Table 4.1 reproduces the summary that they established.

PIPS relies on two parallelization algorithms, the first one is Allen and Kennedy's and

```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    a[i][j]=i+j;
    b[i][j]=b[i][j-1]+a[i][j]*c[i-1][j];
    c[i][j]=2*b[i][j]+a[i][j];
  }
}
```

(a) Input code.

```
#pragma omp parallel for
for(i=0; i<N; i++) {
  #pragma omp parallel for
  for(j=0; j<N; j++) {
    a[i][j]=i+j;
  }
}
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    b[i][j]=b[i][j-1]+a[i][j]*c[i-1][j];
  }
  #pragma omp parallel for
  for(j=0; j<N; j++) {
    c[i][j]=2*b[i][j]+a[i][j];
  }
}
```

(b) After parallelization.

Figure 4.5: Example of Allen and Kennedy algorithm as implemented in PIPS: loops are distributed and parallelism is expressed using OpenMP pragmas.

the other one is based on Creusillet's array region analysis [Creusillet & Irigoin 1996b] I detail further now.

4.3.1 Allen and Kennedy

The Allen and Kennedy algorithm is based on the dependence graph with levels. This algorithm has been proven optimal by Darte and Vivien [Darte & Vivien 1996b] for such dependence abstraction. This algorithm was designed for vector machines, and thus in its basic version distributes the loops as much as possible and maximizes parallelism.

The implementation in PIPS uses a dependence graph built using a dependence test [Irigoin *et al.* 1991] based on a variation of the Fourier–Motzkin pairwise elimina-

tion technique [Triolet *et al.* 1986]. Allen and Kennedy's algorithm is implemented by structuring the dependence graph into strongly connected components, each of which is recursively analyzed with an incremented dependence level.

An example illustrating the result of the processing of the PIPS implementation of the Allen and Kennedy algorithm is presented in Figure 4.5. The loop distribution involved exhibits the maximum parallelism but adds implicit synchronization barriers. Moreover, it can break cache temporal reuse and prevent array scalarization. In the code in Figure 4.5a the same reference to array `a` appears in the three statements, thus the corresponding array element may stay in the cache. Moreover, if `a` is not used later in the computation, the reference can be scalarized. The resulting code after transformation (shown in Figure 4.5b) shows that `a` cannot any longer be scalarized since it is referenced in more than one loop now. Moreover, it is less likely to remain in the caches and the ratio of the number of arithmetic operations to the number of memory accesses decreases. The drawback of loop distribution can be circumvented using a loop fusion algorithm presented in Section 4.6.

Another issue is that this algorithm in this basic form (the one implemented in PIPS) has restrictions on the control flow; for instance, no test is allowed in the loop body. The algorithm introduced in the next section addressed these issues by providing a coarse grained parallelization algorithm based on convex summarized array regions [Creusillet & Irigoin 1996b].

4.3.2 Coarse Grained Parallelization

The second parallelization algorithm is a coarse grained scheme that relies on array region analyses [Creusillet & Irigoin 1996b]. No specific loop transformation is involved. The details about this parallelization method have been published in [Irigoin *et al.* 2011 (perso)]. The process is summarized below.

Berstein's conditions [Bernstein 1966] are used between two iterations and extends the original definition to array regions. A loop is scheduled as parallel if no iteration reads or write an array element written by another iteration. It is expressed for any variable v:

$$\{\phi \mid \exists \sigma, \sigma' \in P_B \quad \phi \in (R_{B,v}(\sigma) \vee W_{B,v}(\sigma)) \wedge \phi \in W_{B,v}(\sigma') \wedge (\sigma(i) \neq \sigma'(i))\} = \varnothing$$

with σ the store, P_B the preconditions over the loop body, $R_{B,v}(\sigma)$ the read array region for the whole loop body for variable v, and finally $W_{B,v}(\sigma')$ the written array region for the whole loop body for variable v.

It can be rewritten more precisely:

$$\{\phi \mid \exists \sigma, \sigma' \in P_B \quad \phi \in (R_{B,v}(\sigma) \vee W_{B,v}(\sigma)) \wedge \phi \in W_{B,v}(\sigma') \wedge T_{B,B}(\sigma, \sigma')\} = \varnothing$$

where $T_{B,B}$ stands for the transformer expressing the transition of at least one iteration on the store, i.e., the transitive closure for one iteration $T_{B,B} = T_B^+$ considering that B includes going to the next iteration.

This algorithm is used extensively in PIPS because it is complementary with Allen and Kennedy. When parallelization is sought instead of vectorization, the Allen and Kennedy distribution adds undesired implicit synchronization barriers. Moreover, no dependence graph is involved, computation of which can be costly. The array regions can be costly as well, but while the dependence graph complexity depends on the number of statements involved, the complexity depends on the size of the linear algebra system resulting from the array accesses. There are no restrictions on the loop body such as on the control flow or function calls as introduced in Section 3.2 page 64; it avoids loop distribution and thus improves the locality and size of the loops. The main limitation is that the current implementation does not integrate an array privatization phase [Creusillet 1996] and a reduction detection. This latter point is addressed separately in PIPS as presented in the following section.

4.3.3 Impact on Code Generation

As shown above, there are two existing different parallelization algorithms implemented in PIPS. Figure 4.6 illustrates the impact of using one algorithm over the other. While Allen and Kennedy distribute the original loop nest in three different perfectly nested loop nests expressing two-dimensional parallelism, the coarse grained parallelization keeps the original nesting and detects one dimension as parallel. Moreover, the parallel dimension is inside a sequential one, which means that it leads to `m` kernel launches.

Section 7.3 provides experiments about the parallelizing algorithms and shows that overall the Allen and Kennedy scheme leads to a more efficient code on all tested architectures with respect to coarse grained parallelization. While the acceleration is very limited on old architectures such as the G80, dramatic improvement is observable on more recent GPUs with an execution time up to eight times faster on Fermi and four times on Kepler using the example figure 4.6.

```
/* Calculate the m * m correlation matrix. */
for(i=1;i<m;i++) {
 for(j=i;j<=m;j++) {
  symmat[i][j] = 0.0;
  for(k=1;k<=n;k++)
   symmat[i][j] += data[k][i] * data[k][j];
  symmat[j][i] = symmat[i][j];
 }
}
```

(a) Input code.

```
for(i=1;i<m;i++)//Parallel
 for(j=i;j<=m;j++)//Parallel
  symmat[i][j] = 0.0;
for(i=1;i<m;i++)//Parallel
 for(j=i;j<=m;j++)//Parallel
  for(k=1;k<=n;k++)
   symmat[i][j] +=
         data[k][i]*data[k][j];
for(i=1;i<m;i++)//Parallel
 for(j=i;j<=m;j++)//Parallel
   symmat[j][i] = symmat[i][j];
```

(b) After parallelization using Allen and Kennedy.

```
for(i=1;i<m;i++) {
 for(j=i;j<=m;j++) {//Parallel
  symmat[i][j] = 0.0;
  for(k=1;k<=n;k++)
   symmat[i][j] +=
        data[k][i]*data[k][j];
  symmat[j][i] = symmat[i][j];
 }
}
```

(c) After parallelization using Coarse Grained method.

Figure 4.6: The impact of the two parallelization schemes on a example of code performing a correlation. Allen and Kennedy algorithm results to three different parallel loop nests expressing the maximum parallelism, while the coarse grained algorithm detects only one parallel loop leading to less synchronization but also less exposed parallelism.

4.4 Reduction Parallelization

PIPS provides an algorithm for reduction detection based on the unified framework introduced by Jouvelot and Dehbonei [Jouvelot & Dehbonei 1989]. The implementation is rather straightforward yet powerful. Once detected, the reductions can be parallelized depending on the target capabilities.

4.4.1 Detection

The algorithm is interprocedural and requires that a summary is produced for all the callees in a function. This implies that the algorithm has to be applied first on the leaf of the call graph before handling callers. Intraprocedurally, the following algorithm detects

reductions in statements like

```
//   call sum[s[a]],sum[b],
   s[a] = s[a]+b++;
```

where you can see the comment added by PIPS that indicates that two reductions have been detected, one on $s[a]$ and the other on b. Statements are first processed individually and reductions are extracted according to these properties:

1. If the statement is a call, then get the translated interprocedural summary for this call site.
2. The statement is not a call, then perform a recursion on the inner expression to find an operation that would correspond to either an assignment, an update, or an unary operator. The recognized operators are the following: `+=`, `-=`, `*=`, `/=`, `|=`, `&=`, `^=`, `++` (pre and post), and `--` (pre and post).
3. For other than unary operators, assert that the left-hand side is a reference, either a scalar or an array.
4. Both the left-hand side and the right-hand side expressions (if any) must be side effect free. i.e., if any call occurs it has to be a call to a pure function.
5. In the case of an assignment, the right-hand side has to use a compatible operator, i.e., one of the following: `+`, `-`, `*`, `/`, `min`, `max`, `&&`, `||`, `&`, `|`, and `^`.
6. In the case of an assignment, the right-hand side also has to include a reference to the same reference as the one on the left-hand side. Perform this search with a recursion through the right-hand side expression accepting only compatible operators.
7. Assert that there is no effect that may conflict with the reduced variable other than the ones touching reference in the left-hand side of the considered reduction and the reference found in the right-hand side. This prevents the wrong catching of the following two examples

   ```
   //   call sum[b],sum[b],
      b = b+b;
   //   call sum[s[a]],sum[b],
      s[a] = s[a] + (b=b+1,b);
   ```

8. Finally, conduct a sanity pass to avoid the declaration of two reductions on the same variable. If compatible, they are merged.

The last step prevents situations such as the two following function calls with side effects:

```
C  summary sum[X],prod[Y],
      REAL FUNCTION FSUMPROD(X, Y, Z)
C  call sum[X],
      X = X+Z
C  call prod[Y],
      Y = Y*Z
      FSUMPROD = Z
      END

C  summary sum[X],prod[Y],
      SUBROUTINE SUMPROD(X, Y, Z)
C  call sum[X],
      X = X+Z
C  call prod[Y],
      Y = Y*Z
      END

   PROGRAM REDUCTION
   ...
C  call prod[P],sum[S],prod[P],sum[S],
         CALL SUMPROD(S, P, FSUMPROD(S, P, 3.))
C  call prod[S],sum[P],prod[P],sum[S],
         CALL SUMPROD(P, S, FSUMPROD(S, P, 3.))
```

The first function call shows that the reduction is duplicated for `S` and `P`, because they are present for both `SUMPROD` and `FSUMPROD`. Since they are compatible (only sum or only product on a given array) they can be merged and kept, this is not the case for the second call where the two reductions are mutually exclusive (sum and product) and are eliminated of the detected reductions by the algorithm.

The reduction information is summarized at each level of the PIPS' HCFG. For instance, it is summarized at loop level so that this information can be used for parallelization. The summarization ensures that there is no other write of the reduced reference that would be incompatible with the reduction. Figure 4.7 shows an example of an interprocedural analysis of the reduction in a Fortran program.

```
C  summary sum[X],prod[Y],
      REAL FUNCTION FSUMPROD(X, Y, Z)
C  call sum[X],
      X = X+Z
C  call prod[Y],
      Y = Y*Z
      FSUMPROD = Z
      END

C  summary sum[X],prod[Y],
      SUBROUTINE SUMPROD(X, Y, Z)
C  call sum[X],
      X = X+Z
C  call prod[Y],
      Y = Y*Z
      END

      PROGRAM INTERACT
      S = 0.
      P = 1.
C  call prod[P],sum[S],
      CALL SUMPROD(S, P, 2.1)
C  call prod[P],sum[S],
      CALL SUMPROD(S, P, 2.+I)
C  loop prod[P],sum[S],
      DO I = 1, N
C  call prod[P],sum[S],
         CALL SUMPROD(S, P, 2.+I)
C  call prod[P],sum[S],
         CALL SUMPROD(S, P, FSUMPROD(S, P, 3.))
      ENDDO
      DO I = 1, N
         CALL SUMPROD(P, S, FSUMPROD(S, P, 3.))
      ENDDO
      DO I = 1, N
C  call prod[P],sum[S],
         CALL SUMPROD(S, P, 2.+I)
C  call prod[S],sum[P],
         CALL SUMPROD(P, S, 1.-I)
      ENDDO
      END
```

Figure 4.7: Example of reduction detection and interprocedural propagation in PIPS.

4.4.2 Reduction Parallelization for GPU

The parallelization of loops with reductions can be handled in different ways. PIPS used to provide only a simple parallelization for OpenMP. This implementation was a simple transformation that was adding an OpenMP pragma with a reduce clause to loops whose statements were all scalar-only reductions.

I have designed and implemented a new method, called Coarse Grained Reductions (CGR) that fits within the existing parallelization algorithms. The implementation is made in the coarse grained parallelization algorithm presented in Section 4.3.2.

The coarse grained parallelization uses array region analysis to find conflicts between two iterations of a loop. Such conflicts prevent parallel scheduling of the loop. The algorithm has been adapted to handle reductions by ignoring conflicts related to references involved in the reduction. If ignoring a conflict eliminates all cycles, the loop is marked as *potentially* parallel. Potentially because another transformation replacing the reduction with a parallel compliant equivalent operation is necessary to execute the loop in parallel.

The fact that the parallelization phase does not directly modify the schedule but provides only the information that a potential parallelization may occur provides a decoupling of the reduction detection and the transformation. While this indicates the maximum potential parallelism that may be found in a code, not all reductions can be parallelized depending on the capabilities of the target. The OpenMP output benefits directly from this approach. It is generated by a new transformation that parallelizes a wider range of loops, since it is no longer limited to loops with a body that contains only reduction statements as was the legacy transformation available in PIPS.

Targeting GPU, one way of parallelizing loops with reductions is to make use of hardware atomic operations introduced in Section 2.4.3, Page 42. Since different GPUs do not share the same capabilities, and since CUDA and OpenCL do not exhibit the exact same set of functions, a rough description of the target capabilities is provided to the compiler. A new implemented transformation exploits this description to select compatible previously detected reductions. If the target supports the corresponding atomic operation, then the substitution is made and the loop is declared as parallel in order to be transformed as a kernel in a further phase. Figure 4.8 contains an example of a sequential histogram code and the resulting code after reduction detection and replacement with an atomic operation.

```
static void _histogram(int data[NP][NP][NP],
                int histo[NP][NP][NP]) {
  int i,j,k;
  for (i = 0; i < NP; i++) {
    for (j = 0; j < NP; j++) {
      for (k = 0; k < NP; k++) {
        int x = floor(((float)data[i][j][k]) / (float)(NP * NP));
        int y = floor(((float)(data[i][j][k] - x * NP * NP))
            / (float)(NP));
        int z = data[i][j][k] - x * NP * NP - y * NP;
        ++histo[x][y][z];
      }
    }
  }
}
```

(a) Input code.

```
static void _histogram(int data[NP][NP][NP],
                int histo[NP][NP][NP]) {
  int i,j,k;
  for (i = 0; i < NP; i++) { // Scheduled as parallel
    for (j = 0; j < NP; j++) { // Scheduled as parallel
      for (k = 0; k < NP; k++) { // Scheduled as parallel
        int x = floor(((float)data[i][j][k]) / (float)(NP * NP));
        int y = floor(((float)(data[i][j][k] - x * NP * NP))
            / (float)(NP));
        int z = data[i][j][k] - x * NP * NP - y * NP;
        atomicAddInt(&histo[x][y][z],1);
      }
    }
  }
}
```

(b) After replacement with atomic operation.

Figure 4.8: An example of reduction parallelization of an histogram using hardware atomic operations.

4.4.3 Parallel Prefix Operations on GPUs

To parallelize some reductions, parallel prefix operations can be used. In 1980, Ladner and Fischer introduced parallel prefix reductions [Ladner & Fischer 1980]: this has been a widely studied field since then. In 2004, Buck and Purcell [Buck & Purcell 2004] explained how map, reduce, scan, and sort can be implemented on a GPU using graphic primitives. Sengupta et al. presented later the implementation of parallel prefix scan using CUDA [Sengupta *et al.* 2007].

In 2010, Ravi et al. introduced a runtime system and framework to generate code from a high-level description of the reductions [Ravi *et al.* 2010]. The runtime scheduling is flexible enough to share the workload between GPUs and multicore CPUs. This system can be seen as a potential back end for an automatic parallelizer like ours.

The recognition and the parallelization of reductions that do not match the classic patterns like the ones detected in PIPS have been widely studied, and is still an active field [Redon & Feautrier 1993, Fisher & Ghuloum 1994, Matsuzaki *et al.* 2006, Zou & Rajopadhye 2012].

There has been work to provide efficient implementation for parallel prefix operations on GPUs [Sengupta *et al.* 2007, Harris *et al.* 2007, Capannini 2011]. An automatic scheme that detects such operations could be associated with a code generator that targets such libraries.

4.5 Induction Variable Substitution

Induction variable substitution is a classical transformation to enable parallelization. It is the opposite of strength reduction. Induction variables are usually detected in loops using pattern matching on the initialization and on the updates in the loop body. This section shows how the PIPS precondition analysis [Irigoin *et al.* 2011] is used to define a new algorithm to detect and replace induction variables. Given a loop L, the algorithm processes every statement S in its body, and performs the following steps:

1. Fetch the precondition P associated to S.
2. Evaluate individually each linear relation in P equations and inequations:
   (a) find in the relation a variable `k` modified in the loop body,
   (b) verify that all other variables are either loop indices or loop invariant,
   (c) construct the linear expression to replace `k`.

```
k = -1;
for(i=0;i<SIZE;i++) {
 k = i;
 for(j=0;j<SIZE;j++) {
  sum = B[j-k][k] + A[k];
  A[k++] = sum;
 }
}
k = SIZE;
for(i=0;i<SIZE;i++) {
 if(k--)
  A[k] +=
    B[j-k][k];
 if(--k)
  A[k] +=
    B[j-k][k];
}
```

(a) Original code.

```
k = -1;
for(i=0;i<SIZE;i++) {
 k = i;
 for(j=0;j<SIZE;j++) {
  sum = B[j-(i+j)][i+j]+A[i+j];
  A[k = i+j+1, k-1] = sum;
 }
}
k = SIZE;
for(i=0;i<SIZE;i++) {
 if(k = -2*i+SIZE-1, k-1)
  A[-2*i+SIZE-1] +=
    B[j-((-2)*i+SIZE-1)][-2*i+SIZE-1];
 if(k = -2*i+SIZE-2)
  A[-2*i+SIZE-2] +=
    B[j-((-2)*i+SIZE-2)][-2*i+SIZE-2];
}
```

(b) After induction substitution.

Figure 4.9: Example of induction variable substitution to enable loop nest parallelization.

3. Replace in the statement all `k` induction variables found with a linear expression.

This transformation is challenging in a source-to-source context when targeting C code. Figure 4.9a gives an example of such challenge. The C language allows side effects in references, for instance `A[k++] = ...`. The solution that I designed and implemented handles these references with respect to the C standard. For instance `A[k++] = ...` is replaced by `A[k = i+j+1, k-1] = ...` (see in Figure 4.9b), thanks to the comma operator that evaluates its first operand and discards the result, and then evaluates the second operand and returns this value. The transformed source code is as close as possible to the initial code and the number of statements is left unchanged.

4.6 Loop Fusion

Loop fusion is a transformation that consists in collapsing two or more loops together into one loop. It has been widely studied for a long time [Allen & Cocke 1972, Burstall & Darlington 1977, Kuck *et al.* 1981, Allen 1983, Goldberg & Paige 1984, Wolfe 1990, Bondhugula *et al.* 2010]. Finding an optimal solution to the global fusion problem is all but trivial [Darte 2000] and there are many ways to address the problem, as well as different

```
for(i=0; i<n; i++)
 for(j=0; j<m; j++)
  A[i][j] = (double) 1.0;
for(i=0; i<n; i++)
 for(j=0; j<m; j++)
  B[i][j] = (double) 1.0;
for(i=0; i<n; i++)
 for(j=0; j<m; j++)
  C[i][j] = A[i][j]+B[i][j];
```

(a) The code before fusion.

```
for(i=0; i<n; i++)
 for(j=0; j<m; j++) {
  A[i][j] = (double) 1.0;
  B[i][j] = (double) 1.0;
  C[i][j] = A[i][j]+B[i][j];
 }
```

(b) After fusion.

Figure 4.10: Example of loop fusion.

definitions of the problem itself.

This transformation helps to reduce the overhead of the branching and incrementation by eliminating loop headers and increase the size of the body. It exhibits several benefits such as

- more opportunities for data reuse, mostly temporal locality,
- more instructions can be scheduled, better pipeline usage or ILP,
- further array contraction [Gao *et al.* 1993, Sarkar & Gao 1991] (see also Figure 4.21 and Section 4.7).

However, loop fusion has some disadvantages. For instance, the pressure on the instruction cache and on the registers usage within the loop increases.

4.6.1 Legality

Loop fusion is not always legal as it may modify the semantics of the program. An invalid loop fusion can lead to a reverse order of dependent computations. Data dependence analysis is used to determine when the fusion is legal or not.

The validity of loop fusion has been widely studied [Allen & Cocke 1972, Warren 1984, Aho *et al.* 1986], but can be expressed in different ways. Allen and Cocke propose simple conditions for the validity of loop fusion in [Allen & Cocke 1972]:

1. the control conditions are unique among the loops,
2. the loop headers control the same number of iterations,
3. the loops are not linked by a data dependence.

However, it restricts the number of loops that can fused and prevents any array contraction since no data dependence is allowed. Warren [Warren 1984] proposes a slightly relaxed but still simple set of conditions:

1. the candidate loop nests are consecutive in the source code,
2. induction variables of both loops iterate toward the same upper bound after loop normalization,
3. the fused bodies preserve all the dependences from the first loop to the second loop.

To summarize, the first condition in both proposals aims at avoiding control dependences, i.e., it ensures that both loops always share the same execution path. The second condition intends to enforce that loops are compatible without any sophisticated transformations such as loop shifting or index set splitting for example, i.e., they have the same number of iterations. Finally, the last conditions guarantee the semantic equivalence of the two fused loops with the initial code. The first proposal is more restrictive because it prevents any data dependence between the two loops, while the second proposal is more general.

The last condition is key in establishing the validity of a loop fusion. It has been shown in [Warren 1984] that dependence with a distance vector allows stating whether the fusion is possible or not. If the distance is positive or null then the fusion is valid. Another definition without distance was given in [Kennedy & McKinley 1994]. The fusion is valid if no dependence arc from the first loop body to the second is inverted after fusion. Such dependence arcs are called fusion-preventing in the next section.

Irigoin et al. conjectured another solution [Irigoin *et al.* 2011 (perso)] based on array region analysis [Creusillet & Irigoin 1996b]. The proposal allows identifying these dependences without any dependence graph (see Section 4.6.5).

When fusing parallel loops, a legal fusion may end up with a sequential loop. This happens when the dependence distance is positive, or, with the alternative definition, when the dependence after fusion becomes carried by the loop. Such dependences might also be considered as fusion-preventing depending on the goals of the algorithm, as shown in Figure 4.11.

Some advanced transformations can remove fusion-preventing dependences. For example, Xue et al. eliminate anti-dependences using array copying [Xue 2005]. Shifting and peeling[2] techniques described in [Manjikian & Abdelrahman 1997] allow the fusion and

2. These transformations enable loop fusion for loops with different iteration set.

```
for(i=1;i<N;i++) // Parallel
  a[i]=0;
for(i=1;i<N;i++) // Parallel
  b[i]=a[i-1];
```

(a) Original code: two parallel fusable loops.

```
for(i=1;i<N;i++) { // Sequential
  a[i]=0;
  b[i]=a[i-1];
}
```

(b) After fusion, the loop is sequential.

Figure 4.11: Example of two parallel loops that can be legally fused, but the resulting loop nest would be sequential.

parallelization of multiple loops in the presence of loop-carried dependences. Loop shifting and peeling were also addressed by Darte et al. [Darte *et al.* 1996, Darte & Huard 2000].

Figure 4.10 illustrates a simple loop fusion.

4.6.2 Different Goals

While the earlier algorithms intended to maximize the number of fusions or minimize the total number of loops [Allen & Cocke 1972, Kuck *et al.* 1981, Warren 1984] to reduce the control overhead, later contributions extended the goals of loop fusion.

When any bad memory access pattern resulted in swapping a memory page, Kuck et al. studied the applicability of loop fusion for improving performance in environment with virtual memory management [Kuck *et al.* 1981]. It was also used to maximize the usage of vector registers [Kuck *et al.* 1981, Allen 1983] or to enable more effective scalar optimizations such as common subexpression elimination [Wolfe 1990]. Later, fusion was used to increase locality [Manjikian & Abdelrahman 1997, Bondhugula *et al.* 2010], to generate better access patterns for hardware prefetchers [Bondhugula *et al.* 2010], or even to reduce power consumption [Zhu *et al.* 2004, Wang *et al.* 2010].

Other algorithms using loop fusion were designed to maximize task parallelism with minimum barrier synchronization [Allen *et al.* 1987, Callahan 1987].

Kennedy and McKinley introduced an algorithm that focuses on maximizing the parallelism, the *ordered typed fusion* [Kennedy & McKinley 1994], by avoiding fusing sequential and parallel loops. The type carries the information about the schedule of the loop: parallel or sequential. This solution is minimal in term of number of parallel loops.

They extended this algorithm to handle an arbitrary number of types [Kennedy & Mckinley 1993] in order to handle noncompatible loop headers. They obtained a solution they claim to be minimal in the number of parallel loops and the total number of loops.

They then introduced the general weighted fusion algorithms [Kennedy & Mckin-

ley 1993] to maximize reuse. The weight represents any metric that would indicate that it is preferable to fuse a couple of loops instead of another one.

Gao et al. also proposed a weighted loop fusion to maximize array contraction [Gao *et al.* 1993] based on the maximum-flow/minimum-cut algorithm [Dantzig *et al.* 1954] but they did not address loops with different headers. Their algorithm relies on a Loop Dependence Graph (LDG) (see Section 4.6.4) where edges can be of three types: nonfusible, fusible and contractable, and fusible but noncontractable.

The weighted fusion problem was shown in 1994 to be NP-hard first in [Kennedy & McKinley 1994], then Darte proved it for a broader class of unweighted fusion [Darte 2000], including the typed fusion for two types or more.

Kennedy and McKinley proposed two polynomial-time heuristics [Kennedy & McKinley 1994] as a solution for the weighted loop fusion [Gao *et al.* 1993]. Finally Kennedy proposed a fast greedy weighted loop fusion heuristic [Kennedy 2001].

Megiddo et al. present a linear-sized integer programming formulation for weighted loop fusion [Megiddo & Sarkar 1997]. They claim that despite the NP-hardness of the problem, an optimal solution can be found within time constraints corresponding to a product-quality optimizing compiler.

Bondhugula et al. used the polyhedral model to provide first maximal fusion [Bondhugula *et al.* 2008c, Bondhugula *et al.* 2008a], and later a metric-based [Bondhugula *et al.* 2010] algorithm that optimizes at the same time for hardware prefetch, locality, and parallelism.

Pouchet et al. [Pouchet *et al.* 2010b] build a convex set that models the set of all legal possibilities on which an iterative empirical search is performed.

Loop fusion can also be used to extend the iteration set of a loop, using some index set splitting as shown in Figure 4.12. This technique is explored by Wang et al. [Wang *et al.* 2010].

The loop-fusion transformation has been widely studied in different contexts. The next section present a GPGPU perspective and the associated specific constraint.

4.6.3 Loop Fusion for GPGPU

In the context of GPGPU, loop fusion is directly related to the number of kernels obtained and their size, as presented in Section 4.2. Four major benefits are expected from loop fusion:

1. data reuse,

2. array contraction (see Section 4.7),
3. increasing the size of the kernel,
4. reducing the overhead associated to the kernel launch.

Items one and two are common benefits when dealing with loop fusion. Since recent GPU architectures include multilevel hardware caches (see Section 2.4), they may benefit from such reuse. However, caches are small when compared to the number of threads, and are intended mostly to provide spatial locality. We expect loop fusion to allow to keep data in registers, avoiding external memory accesses. Finally a fused-kernel exhibits more opportunities for data reuse in local memory, as shown in Section. 2.3.

The third benefit exposes potentially more ILP to the hardware scheduler and the compiler. This is helpful for small kernels, but also increases the register pressure and the code size dramatically for larger kernels. Section 2.4.2 demonstrates how ILP is exploited in modern GPUs.

Finally, the last benefit is directly linked to the reduction of the number of kernels, and thus the number of kernel calls. Launching a kernel requires the driver to send the binary code for the kernel, the parameters, and the launch configuration (number of threads, work-group size) to the GPU over the PCIe bus. Then the hardware scheduler begins scheduling and filling the multiprocessors with many thousands of threads (see Section 2.4). Finally, at the end of the computation, the scheduler has to wait for all the threads to finish, leaving potentially some multiprocessors stalled. These operations are not negligible for small kernels. Stock et al. measured the overhead of starting a kernel of the GPU as 20 µs for an Nvidia GTS 8800 512 and as 40 µs for an Nvidia GTX 280 [Stock & Koch 2010]. They manually performed loop fusions to reduce the number of kernel launches, and thus the overhead.

Overall, these fusions can improve significantly the performance. Membarth et al. obtained a 2.3 speedup by manually applying loop fusion on a multiresolution filtering application [Membarth *et al.* 2009]. Wang et al. published measurements with a speedup of five after loop fusion [Wang *et al.* 2010]. Fousek et al. evaluated the fusion of CUDA kernels in the context of predefined parallel map operations. The algorithm they proposed performs an oriented search over the set of valid fusions. They predict the resulting execution times, based on off-line benchmarking of predefined functions. On a simple example that chains six operations (matrix–matrix multiply, matrix–vector multiply, vector normalization, matrix–matrix multiply, matrix–matrix add, matrix–scalar multiply) they obtained a 2.49 speedup.

```
for(i=0;i<N;i++) {
  F(i,N,A,B,C);
}
for(i=0;i<M;i++) {
  G(i,M,X,Y,Z);
}
```

(a) Input code.

```
for(i=0;i<N+M;i++) {
 if(i<N)
  F(i,N,A,B,C);
 else
  G(i-N,M,X,Y,Z);
}
```

(b) Loop fusion to schedule more threads on the GPU.

Figure 4.12: Example of a loop fusion scheme to extend the iteration set of a loop nest.

Wang et al. [Wang *et al.* 2010] study three different types of loop fusion, extending the iteration set to concatenate the two original loop iteration set (see Figure 4.12). Their goal is reduction of power consumption and they do not improve performance with respect to the classical loop fusion scheme implemented in PIPS. Modern GPUs are able to schedule more than one kernel at a time on different multiprocessors. Therefore the performance improvement of this approach, even on small iteration set, is rather theoretical and only the launch overhead of the fused kernel may be avoided.

Loop fusion is also used in the GPU implementation of MapReduce [Catanzaro *et al.* 2008]. Map kernels are fused to reduce synchronizations, communications, and enabling data exchange in on-chip memories.

Finally the Thrust library manual [Bell & Hoberock 2011] recommends programmers to fuse explicitly several computation functions into a single kernel, as shown Figure 4.13. This is presented as a good practice and is a key point in order to get good performance.

4.6.4 Loop Fusion in PIPS

In 2010, PIPS did not include any algorithm for loop fusion. I implemented a heuristic-based algorithm that performs unweighted typed loop fusion. It can take into account two types: parallel and sequential loops.

Most algorithms from the previous sections are based on the Loop Dependence Graph (LDG) [Gao *et al.* 1993, Megiddo & Sarkar 1997]. It represents a sequence of loop nests and can be seen as a specialization of the Program Dependence Graph (PDG) [Ferrante *et al.* 1987]. Each loop nest is represented as a node of the LDG. Edges correspond to dependences between statements that belong to the two loop nest bodies. An edge represents a dependence and the information whether the corresponding dependence prevents

```
void saxpy_slow(float A,
                thrust::device_vector<float>& X,
                thrust::device_vector<float>& Y) {
    thrust::device_vector<float> temp(X.size());
    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);
    // temp <- A * X
    thrust::transform(X.begin(), X.end(),
                      temp.begin(), temp.begin(),
                      thrust::multiplies<float>());
    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(),
                      Y.begin(), Y.begin(),
                      thrust::plus<float>());
}
```

(a) Using native Thrust operator, performing 4N reads and 3N writes.

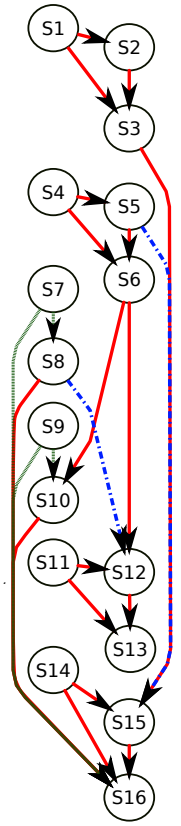
```
struct saxpy_functor {
 const float a;
 saxpy_functor(float _a) : a(_a) {}
 __host__ __device__ float operator()(const float& x,
                                      const float& y) const {
  return a * x + y;
 }
};
// Y <- A * X + Y
void saxpy_fast(float A,
                thrust::device_vector<float>& X,
                thrust::device_vector<float>& Y) {
 thrust::transform(X.begin(), X.end(),
                   Y.begin(), Y.begin(),
                   saxpy_functor(A));
}
```

(b) Using a user-defined kernel, performing 2N reads and N writes.

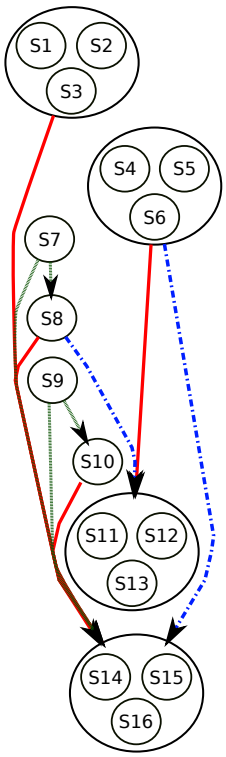
Figure 4.13: Example of manual kernel fusion using Thrust library and a SAXPY example. The first version is expressed using native Thrust operators and requires temporary arrays, the second version fuses the three steps in one user-defined kernel (source [Hoberock & Bell 2012]).

```
for(i=0; i<N; i++){//S1
  a[i]=b[i];        //S2
  a[i]+=2*c[i];     //S3
}

for(i=1; i<M; i++){//S4
  e[i]=c[i];        //S5
  e[i]+=2*b[i];     //S6
}

int k;              //S7
k = d[1];           //S8
int l;              //S9
l = e[1];           //S10

for(i=1; i<M; i++){//S11
  d[i]=2*e[i];      //S12
  d[i]+=b[i];       //S13
}

for(i=0; i<N; i++){//S14
  c[i]+=a[i];       //S15
  c[i]+=k+l;        //S16
}
```

(a) Input code.



(b) Dependence Graph (DG).



(c) Reduced Dependence Graph (RDG).

Figure 4.14: On the left, a sequence of statements, in the middle the associated Dependence Graph (DG), and on the right the corresponding Reduced Dependence Graph (RDG) obtained after clustering the vertices that belong to the same loop.
In solid red the flow dependences, in dashed blue the anti-dependence, and in dotted green the special dependences that model the declaration. The DG view showed here is simplified for the sake of clarity, for instance output dependences and the loop carried dependences are omitted.

the fusion of its two vertices. This information is a key point for many algorithms from the previous section.

The direction or distance vector dependence graphs, introduced in Section 4.6.1, are used to build the LDG with the fusion-compliant status attached to all the edges. But since the PIPS dependence graph exhibits an abstraction based on the dependence level, it does not provide the distance vector in the graph. It would require re-implementing

the PIPS dependence test [Irigoin *et al.* 1991] and the dependence graph representation to provide this information.

Without this information, the alternative method for identifying the fusion-preventing nature of an edge (see Section 4.6.1) requires first effectively fusing the loop nests and recomputing a dependence graph on the new body to establish if a dependence is fusion-preventing or not by a comparison with the original graph. However, it would require a lot of spurious computations to do that preventively for all possible combinations of two loop nests.

Instead I designed a heuristic that provides features similar to the typed fusion introduced by Kennedy and McKinley [Kennedy & McKinley 1994], but operating on an LDG that does not include any information on the fusion-preventing nature of the edges. The vertices are selected by the algorithm and the fusion is tried. It is only at that time that the validity is checked. The algorithm has to proceed using a trial-and-error strategy.

This algorithm traverses the PIPS Hierarchical Control Flow Graph (HCFG) and considers statements that share the same control in the same way as [Ferrante *et al.* 1987]. In PIPS terminology these statements are in the same *sequence* of statements, such as a compound block `{ ... }` in the C language, with no `goto` from or to the statements directly in the sequence.

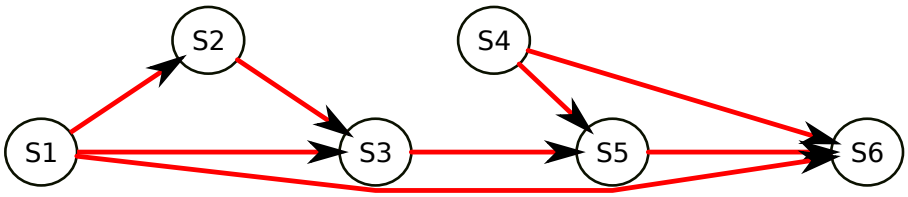
In a sequence, we have to take into account not only loops but also any kind of constructs such as tests, assignments, function calls, etc. The nodes of our graph are not necessarily loop nests. Therefore I use the generic term Reduced Dependence Graph (RDG) instead of LDG, with an example shown on Figure 4.14.

PIPS HCFG represents all these constructs as statements, and they are stored in a linked list to represent the sequence. The RDG is then built by

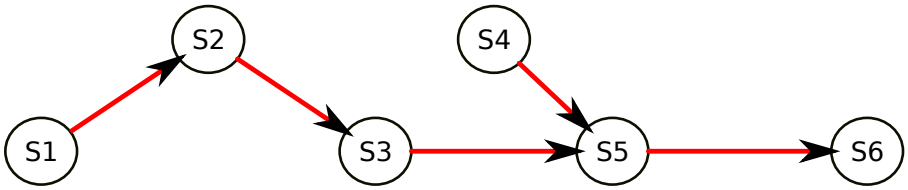
1. creating a vertex for each statement in the sequence,
2. mapping all the inner statements to the vertex statement, and
3. adding an edge for each edge in the dependence graph to the RDG using the previously computed mappings.

It considers only the dependence arcs related to the statements within the sequence, and then the dependence graph is acyclic, so is the RDG obtained.

Figure 4.14a, Page 113, contains a sequence of statements including some loops. The resulting (simplified) Dependence Graph (DG) and the RDG computed by PIPS are presented in Figures 4.14b and 4.14c. At all times, there is a one-to-one mapping from the vertices in the RDG and the statements in the sequence.



(a) The original RDG.



(b) The RDG after pruning.

Figure 4.15: The algorithm begins with a pruning phase. For each direct edge between two vertices it ensures that there is no other path between them.

The originality of the algorithm is to prune the RDG so that any direct arc between two statements guarantees that no other statement needs to be scheduled between the two statements. Therefore this pruning allows traversing the graph in any topological order and trying to fuse two adjacent statements without having first to check if a fusion would introduce a cycle.

Since PIPS is a source-to-source compiler, the readability of the transformed source is an important point. Each transformation must be designed to keep the code structure as close as possible to the original input code. For example, declarations are represented in the Internal Representation (IR) like any other statements. The DG includes edges from these declarations to every use of the variable. Therefore a transformation algorithm relying on the DG naturally keeps the declarations consistent with the uses of variables. My fusion algorithm allows declarations everywhere in the code without preventing fusion.

My algorithm traverses the graph in three independent ways. Each traversal has a dedicated goal:

1. The first traversal favors array contraction, and thus follows the edges in the RDG that correspond to flow dependences. These represent the opportunities to fuse to get reduced liveness, from the definition to the use of a value.

2. The second traversal improves temporal locality. It fuses vertices of the RDG that are linked by edge corresponding to a read-read dependence. In the context of GPGPU, the purposes are multiple. First, accesses to the same memory location are likely to

stay in a register. Then the accesses among a block of threads are more likely to benefit from the cache levels. Finally at a coarser grain level, some transfers between the CPU and the GPU may be avoided.

3. The last traversal is a greedy one that intends to minimize the total number of loops and therefore fuse all vertices in the graph that are not connected. The motivation is to minimize the number of kernel launches and to exhibit more instructions in kernels, leading to more ILP. The drawback is a potentially increased register pressure, and therefore spilling may occur in large kernels.

Note that the second traversal relies on edges that do not exist in the classic DG in PIPS. Moreover, it is not desirable to use them the same way as the other edges since they do not carry the same constraints: they do not imply any order between the connected vertices. One may think about these edges as being undirected arcs. However, since the RDG is a Directed Acyclic Graph (DAG), it cannot include any undirected edge. A separated undirected graph structure over the RDG vertices is computed to keep track of data reuse.

The first traversal starts with each vertex that does not have any predecessor in the graph. It then tries to fuse the vertex with each of its successors, and recurses on each, following edges in the RDG as paths. The algorithm presented in Figure 4.16 illustrates this process.

The second traversal is based on edges representing data reuse. The algorithm is similar to the one used for the first traversal and illustrated in Figure 4.16. Edges are pruned like arcs. The pruning involves the information about the arcs. In fact, an edge between two vertices is kept only if there are no paths between the two vertices in the directed graph. This guarantees that fusing two vertices linked by an undirected edge is always valid from the graph point of view, i.e., it does not create any cycle in the RDG.

Finally the last traversal is a greedy one; every possible pair of vertices that are not linked by any path in the RDG are tried for fusion.

Figure 4.17 presents the algorithm involved in pruning the graph when merging two vertices. The resulting process of this algorithm on the code shown in Figure 4.14 is presented in Figure 4.18. Note how the declaration and initialization of `k` are moved to ensure consistency.

The order these three traversals are performed matters. Since for example in a sequence of three loops, it can arise that the first one can be fused with the second or the third but not both. Therefore the heuristic to chose which fusion to perform instead of the other has

```
// v is modified as side-effect
function FUSE_RDG(V, E)
    for v ∈ V do
        if v has no predecessor then
            FUSE_ALONG_PATH(v)
        end if
    end for
end function

// Fuse all directly connected vertices starting from v
// v is modified as side-effect
function FUSE_ALONG_PATH(v)
    toFuse ← succ(v)
    alreadyTried ← ∅
    while toFuse ≠ ∅ do
        v' ←POP(toFuse)
        if v is a loop and v' is a loop then
            if TRY_TO_FUSE(v, v') then
                // Fusion succeeded, register successors of v' to be tried
                toFuse ← toFuse ⋃ ( succ(v') \ alreadyTried )
                // Fuse in the graph
                FUSE_VERTICES(v, v')
            else
                // Record the failure about fusion with v'
                alreadyTried ← alreadyTried ⋃{v'}
            end if
        end if
        // Recurse on v'
        FUSE_ALONG_PATH(v')
    end while
end function
```

Figure 4.16: Heuristic algorithm `FUSE_RDG` to traverse the RDG and apply fusion. The graph is modified as side effect.

to consider the criteria that is likely to provide the best performance.

4.6.5 Loop Fusion Using Array Regions

Section 4.6.1 presents the classic way of determining the legality of a fusion based on the dependence graph. Irigoin conjectured another solution [Irigoin *et al.* 2011 (perso)] exploiting the array region analysis [Creusillet & Irigoin 1996b]. I designed and implemented in

function FUSE_VERTICES(v, v')
  // Fuse v' into v
  $\text{pred}(v) = \text{pred}(v) \bigcup \big(\text{pred}(v') \setminus \{v\}\big)$
  $\text{succ}(v) = \text{succ}(v) \bigcup \text{succ}(v')$
  for $s \in \text{succ}(v')$ **do**
    // For each successor of v', replace v' with v as a predecessor
    $\text{pred}(s) = \big(\text{pred}(s) \setminus \{v'\}\big) \bigcup \{v\}$
  end for
  for $p \in \text{pred}(v')$ **do**
    if $p \neq v$ **then**
      // For each predecessor of v', replace v' with v as a successor
      $\text{succ}(p) = \big(\text{succ}(p) \setminus \{v'\}\big) \bigcup \{v\}$
    end if
  end for
  // Prune the graph, traversing all paths in both direction from v
  PRUNE(v)
end function

Figure 4.17: Merging two vertices in the graph while enforcing pruning as introduced in Figure 4.15.

```
int k;                  //S7
k = d[1];               //S8

for(i=1; i<M; i++){//S4
  e[i] = c[i];          //S5
  e[i] += 2*b[i];       //S6
  d[i] = 2*e[i];        //S12
  d[i] += b[i];         //S13
}

int l;                  //S9
l = e[0];               //S10

for(i=0; i<N; i++){//S1
  a[i] = b[i];          //S2
  a[i] += 2*c[i];       //S3
  c[i] += a[i];         //S15
  c[i] += k+l;          //S16
}
```

Figure 4.18: The resulting code after applying the loop-fusion algorithm on the code presented in Figure 4.14a.

PIPS the corresponding algorithm, *FusionWithRegions*, that is based on a summarization mechanism using array regions (see Section 3.2).

PIPS computes array region accesses for each structure of its HCFG, including loops and their body. This information, summarized at body-level, enables establishing if the loops can be fused with an extended dependence test [Triolet *et al.* 1986]. Array regions are convex polyhedra. The linear system is used directly in the PIPS dependence test [Irigoin *et al.* 1991] to establish the dependence between the array regions associated to the loop bodies.

In the case of conflict, the dependence test states whether the dependence is loop-carried or not, and whether it is a backward or a forward dependence. A forward loop-independent dependence is totally harmless and therefore can be safely ignored for the fusion.

A backward loop-carried dependence breaks the semantics of the program and always has to be considered as fusion-preventing [Warren 1984, Kennedy & McKinley 1994].

Finally a forward loop-carried dependence does not break the semantics of the program but serializes the execution of the loop. If the loop-fusion algorithm has to maximize the parallelism, then such a dependence has to be considered as fusion-preventing, if at least one of the original loops is parallel.

The main interest of the *FusionWithRegions* algorithm is the simplicity of its implementation in PIPS. It relies on a well-tried polyhedral framework used for array regions. This solution allows avoiding recomputing a full dependence graph each time a fusion is attempted by the algorithm `FUSE_RDG`.

4.6.6 Further Special Considerations

As mentioned in Section 4.2, the mapping of a loop nest on GPU involves only perfectly nested parallel loops. The fusion algorithm can be parametrized to enforce this property. When a fusion succeeds, if the original loops both contained a single parallel loop as body then a fusion is tried on these inner loops. In case of failure, the fusion of the outer loops is reverted.

The algorithm presented at the previous section has to be applied in sequences. The order sequences are picked for processing during HCFG traversal matters. Figure 4.19 presents an example where the sequence corresponding to the body of the first loop has to be processed first. If the inner loops are not fused first, then the outer loops are not fused to avoid breaking the perfect nesting.

Finally, when parallelizing for GPUs, since only the perfectly nested loops are scheduled

```
for(i=0; i<n; i++) {          // Parallel
  for(j=0; j<m; j++) {        // Parallel
    a[i][j]=b[j][i];
  }
  for(j=0; j<m; j++) {        // Parallel
    c[i][j]=a[i][j]+k*j;
  }
}
for(i=0; i<n; i++) {          // Parallel
  for(j=0; j<m; j++) {        // Parallel
    d[i][j]=sqrt(c[i][j]);
  }
}
```

Figure 4.19: Sample code showing that inner loops have to be fused first in order to be able to fuse the outer loops without breaking the perfect nesting.

```
for(i=0; i<n; i++) {          // Parallel
  int tmp[10];
  tmp[0]=0;
  for(j=1; j<10; j++) {       // Sequential
    tmp[j]=tmp[j-1]+a[i][j]+b[i][j];
  }
  for(j=1; j<10; j++) {       // Parallel
    c[i][j]+=tmp[j];
  }
}
```

Figure 4.20: Only perfectly nested loops are labeled parallel to avoid GPU unfriendly loop fusion.

on the GPUs, parallel loops that are located in the body of an outer parallel loop must be declared as sequential. If they are not, the fusion of the inner parallel loop with another inner sequential loop is prevented. This situation is illustrated in Figure 4.20. The second inner loop should not be declared parallel, so that it can be fused with the previous loop.

4.7 Scalarization

Scalarization is a transformation that replaces constant array references to arrays with scalars. This transformation can occur in the usual backend compiler, when it comes to keeping in a register a value fetched from memory as long as possible. Intuitively it means

that performing this transformation at source level might increase the pressure on registers and lead to spilling.

This transformation can also eliminate temporary arrays, most of the time after loop fusion and especially in the context of automatically generated code from high-level languages and tools. The generated C code from a three-line Scilab program contains three temporary arrays that can be totally replaced with scalars after loop fusion (see in Figure 4.21).

In the context of targeting accelerators like GPUs, this transformation is even more critical than on a shared memory system. Indeed, the generated kernel will be faster by performing fewer memory accesses, but it is probably from the reduced memory transfers over the PCIe bus that most of the gains are to be expected.

Array scalarization has been widely studied in the past [Gao *et al.* 1993, Sarkar & Gao 1991, Darte & Huard 2002, Carribault & Cohen 2004]. This section explores different schemes to apply this transformation in the context of offloading kernels to the GPU. The performance impact is evaluated for different GPU architectures.

4.7.1 Scalarization inside Kernel

A simple matrix multiplication naively mapped onto the GPU is shown in Figure 4.22. This kernel includes a sequential loop with a constant array reference. This reference can be kept in a scalar variable during the whole loop. These transformations could be done by the target backend compiler. However, the measurement presented in Figure 7.11, Page 192, indicates that performing it at source level is valuable on all architectures tested, with speedup up to 2.39.

4.7.2 Scalarization after Loop Fusion

Loop fusion generates code where definitions and uses of temporary arrays are in the same loop body. The arrays can be totally removed, saving both memory bandwidth and memory footprint. In the context of automatically generated code from high-level languages and tools, this situation is a common pattern. Figure 4.21 shows an example of such generated code from a three-line Scilab program. After loop fusion, the generated C code contains three temporary arrays that can be replaced by scalars as shown in Figure 4.21b.

To eliminate a temporary array, its elements must not be used later in the program execution. This is checked in PIPS with $\mathcal{OUT}$ regions (see Section 3.2, Page 64).

```
double a[1000][1000];
double t0[1000][1000];
double b[1000][1000];
double t1[1000][1000];
double c[1000][1000];
for(i=0; i<1000; i++) {
 for(j=0; j<1000; j++) {
  a[i][j] = (double) (1.0);
  t0[i][j] = a[i][j]+a[i][j];
  b[i][j] = t0[i][j]+a[i][j];
  t1[i][j] = b[i][j]*2.;
  c[i][j] = t1[i][j]+3.;
 }
}
disp_s0d2("b",1000,1000,b);
disp_s0d2("c",1000,1000,c);
```

(a) After loop fusion.

```
double b[1000][1000];
double c[1000][1000];
for(i=0; i<1000; i++) {
 for(j=0; j<1000; j++) {
  double a, t1, t0;
  a = (double) (1.0);
  t0 = a+a;
  b[i][j] = t0+a;
  t1 = b[i][j]*2.;
  c[i][j] = t1+3.;
 }
}
disp_s0d2("b",1000,1000,b);
disp_s0d2("c",1000,1000,c);
```

(b) After array scalarization.

Figure 4.21: Processing of example in Figure 4.1. A Scilab script compiled to C code offers good opportunities for loop fusion and array scalarization.

```
for (i = 0; i < ni; i++) {
 for (j = 0; j < nj; j++) {
  C[i][j] = 0;
  for (k = 0; k < nk; ++k)
   C[i][j]+=A[i][k]*B[k][j];
 }
}
```

```
C[i][j] = 0;
for (k = 0; k < nk; ++k)
 C[i][j]+=A[i][k]*B[k][j];
```

(a) Naive Kernel body.

```
int scal_C = 0;
for (k = 0; k < nk; ++k)
 scal_C+=A[i][k]*B[k][j];
C[i][j]=scal_C;
```

(b) After scalarization.

Figure 4.22: Simple matrix multiplication example to illustrate the impact of scalarization.

Section 7.5.2, Page 191, shows how this simple case exhibits speedup ranging from 1.96 up to 5.75. In this case the backend compiler cannot do anything and thus it is critical to apply this transformation at source level.

4.7.3 Perfect Nesting of Loops

In some cases, scalarization can break the perfect nesting of loops. Figure 4.23 illustrates such a situation. Constant references in the innermost loop are assigned to scalars and thus the two loops are no longer perfectly nested. Since only perfectly nested loops are mapped, then here only one of the two loops can be executed in parallel after transformation. The useful parallelism is reduced: fewer threads can execute on the GPU.

The kernel generated without scalarization is shown in Figure 4.23c, while the kernel generated for the outer loop is shown in Figure 4.23e: it contains a sequential loop in the kernel. Finally, Figure 4.23d illustrates the mapping of the inner loop resulting in a sequential loop on the host. These three versions differ on the host side; as shown on the host code in Figure 4.23d, no DMA operation is required for `u1` and `u2`. The drawback is that, while the other two versions include only one kernel launch, this one requires as many launches as iterations of the outer loop.

Evaluating which of these three versions leads to the best performance is highly dependent on data size. Here the two loops have different iteration numbers, `N` and `M`. First of all, the transfer times of arrays `u1` and `u2` for the versions in Figure 4.23e and in Figure 4.23c increase with `N`. The number of threads mapped on the GPU scales with `N` and `M` for version in Figure 4.23c, with `N` for version in Figure 4.23e, and `M` for version 4.23d. By increasing the number of threads, more computation has to be done on the accelerator but also more parallelism is exposed and thus it is potentially more likely to keep the GPU busy. The time for computing one kernel with one thread is constant for the versions in Figures 4.23c and 4.23d but scales with `M` in version in Figure 4.23e. Finally, version in Figure 4.23d may suffer from a high number of kernel launches when `N` grows.

This issue is multidimensional, and does not even take into account the difference in performance from one version to the other linked to architectural details like the memory accesses patterns or the potential ILP exposed.

Unless `N` is large and `M` is small, version 4.23e suffers from less parallelism exposed with respect to version 4.23c. When we compare this latter with version 4.23d, a small `N` and a high `M` may provide a slight advantage to version 4.23d because no data transfer is performed. Although the same amount of data has to be transferred, it will be performed using the arguments of the kernel call instead of a separate DMA. Since kernels are executed asynchronously, while the first kernel executes the argument for the second kernel are transfered, providing a kind of overlapping of transfers and computation. However, the overhead of `N` kernel launches can be a lot higher than a single DMA.

```
for(i=0; i<N; i++) {
 // u1[i] and u2[i] are constant
 // references in the inner loop
 for(j=0; j<M; j++) {
  A[i][j]=A[i][j]+u1[i]+u2[i];
 }
}
```

(a) Nest candidate to scalarization.

```
for(i=0; i<N; i++) {
 s_u1 = u1[i]
 s_u2 = u2[i]
 for(j=0; j<M; j++) {
  A[i][j]=A[i][j]+s_u1+s_u2;
 }
}
```

(b) After array scalarization.

```
void k_2d(
  data_type A[N][M],
  data_type u1[N],
  data_type u2[N])
{
 int i = P4A_vp_1;
 int j = P4A_vp_0;
 if(i<N && j<M)
  A[i][j]=A[i][j]+
        u1[i]+u2[i];
}



copy_to_accel(u1);
copy_to_accel(u2);
k_2d(A,u1,u2);
```

(c) Bi-dimensional kernel without scalarization.

```
void k_in(
  data_type A[N][M],
  data_type u1,
  data_type u2,
  int i)
{
 int j = P4A_vp_0;

 if(j<M)
  A[i][j]=A[i][j]+
            u1+u2;
}

for(i=0;i<N;i++) {
 u1_s = u1[i];
 u2_s = u2[i];
 k_in(A,u1_s,u2_s,i);
}
```

(d) Kernel when mapping the inner loop.

```
void k_out(
  data_type A[N][M],
  data_type u1[N],
  data_type u2[N])
{
 int j,i = P4A_vp_0;
 if(i<N) {
  u1_s = u1[i];
  u2_s = u2[i];
  for(j=0;j<M;j++)
   A[i][j]=A[i][j]
         +u1_s+u2_s;
 }
}

copy_to_accel(u1);
copy_to_accel(u2);
k_out(A,u1,u2);
```

(e) Kernel when mapping the outer loop.

Figure 4.23: Array scalarization can break the perfect nesting of loop nests, thus limiting potential parallelism when mapping on the GPU.

This analysis is confirmed by the experiments in Section 7.5.3, Page 191, and illustrated in Figure 7.13.

4.7.4 Conclusion

This section surveys a well-known transformation, scalarization, and shows how the implementation in PIPS, leveraging array region analysis, can help reducing the memory footprint and improving overall performance when targeting GPU.

The GPGPU puts some unusual constraints in such state-of-the-art transformation:

preserve the perfect nesting of loops. I modified the existing implementation to enforce this property. Detailed experiments are presented in Section 7.5, Page 181, and speedups ranging from 1.12 to 2.39 are obtained.

4.8 Loop Unrolling

Loop unrolling (or unwinding) is a well-known loop transformation [Aho & Ullman 1977] to improve the performance of loop execution time. The basic principle is to replicate the loop body many times to perform many iterations. The trip count is then reduced. Figure 4.24 shows an example of such a transformation. The original loop contains only one statement. After unrolling by a factor of four, the new loop body corresponds to four iterations of the original loop. A second loop is added to compute the remaining iterations. Indeed, the unrolled loop can compute only multiples of four iterations and thus, depending on the total number of iterations, the remainder must be processed separately.

This transformation is used to reduce the execution time. The original loop in Figure 4.24 contains just a few computations per iteration, thus the overhead of the loop header and the hazard associated to the branching may be significant. Moreover, the Instruction Level Parallelism available to the hardware scheduler is poor. The unrolled loop addresses these issues and exhibits also a larger potential for further optimization. This is obtained by means of increased register pressure [Bachir *et al.* 2008] and a larger code that might break the instruction cache. These shortcomings can annihilate any of the aforementioned benefits.

In spite of its drawbacks, unrolling is a common optimization technique implemented in all mainstream compilers. In the context of GPU programming, this transformation is interesting for two reasons. The first arises when sequential loops are encountered inside kernels, while the second consists in unrolling parallel loops that are mapped on threads as shown in Section 4.2 page 98. In this latter case, it is a trade-off since it reduces the TLP exposed in favor of potentially more ILP.

Section 2.4 presents the GPU architectures, and more especially how ILP can be exploited by current GPU architectures. For instance, some AMD architectures are based on a VLIW instruction set. Sohi & Vajapeyam show that loop unrolling is a must to get speedup for VLIW architectures [Sohi & Vajapeyam 1989]. Lee et al. show interesting speedups obtained with loop unrolling for superscalar architectures [Lee *et al.* 1991]. Nvidia architectures and the latest AMD one include a hardware scheduler that can ben-

```
for(k = 0; k < nk; k += 1) {
   D_scalar += alpha*A[i][k]*B[k][j];
}
```

(a) Original code.

```
for(k = 0; k < 4*((nk)/4); k += 4) {
  D_scalar += alpha*A[i][k]*B[k][j];
  D_scalar += alpha*A[i][k+1]*B[k+1][j];
  D_scalar += alpha*A[i][k+2]*B[k+2][j];
  D_scalar += alpha*A[i][k+3]*B[k+3][j];
}
// Epilogue
for(; k < nk; k += 1) {
  D_scalar += alpha*A[i][k]*B[k][j];
}
```

(b) After loop unrolling.

Figure 4.24: Example of loop unrolling with a factor four.

efit from unrolling in the same way. Stone et al. found that unrolling the parallel loops mapped on threads eight times can provide a speedup of two [Stone *et al.* 2007], Volkov confirmed later this results with other experiments [Volkov 2011].

Section 7.6 presents the performance gains that I obtained by simply unrolling the inner loop in the kernel from the code in Figure 4.22. An acceleration of up to 1.4 can be observed depending on the architecture. The register pressure impact is also studied, and it is shown that loop unrolling impacts the register consumption in kernels.

4.9 Array Linearization

Fortran and C programs make use of multidimensional arrays. However, this is an issue when using OpenCL: the standard does not allow the use of multidimensional arrays. These have to be converted to pointers or 1D arrays and the accesses have to be linearized. This transformation is also mandatory when using CUDA and C99 VLA arrays that are not supported.

The result of this transformation is illustrated in Figure 4.25.

The impact on performance for CUDA code is checked in Section 7.7. This transformation can lead to a slowdown up to twenty percent, but can also, in one configuration, leads to a small speedup of about two percent.

```
void k(int ni,int nj,int nk,
       double A[ni][nk],
       double B[nk][nj],
       double C[ni][nj],
       int i, int j) {
 int k;
 C[i][j] = 0;
 for(k=0; k<nk; ++k)
  C[i][j]+=A[i][k]*
           B[k][j];
}
```

(a) Original kernel.

```
void k(int ni,int nj,int nk,
       double *A,
       double *B,
       double *C,
       int i, int j) {
 int k;
 *(C+i*nk+j) = 0;
 for (k=0; k<nk; ++k)
  *(C+i*nj+j)+=*(A+i*nk+k) *
               *B(k*nj+j);
}
```

(b) After array linearization.

Figure 4.25: Simple matrix multiplication example to illustrate array linearization interest.

There is no reason in my opinion why a standard like OpenCL forbids the use of multidimensional arrays in the formal parameters of the functions, and, considering the performance impact, we hope that a future release will remove this constraint.

4.10 Toward a Compilation Scheme

This chapter presents many individual transformations that are applicable at different times in the whole process. The chaining of all these transformations can be tough and different choices in the process will lead to different performance results.

I proposed and implemented a simple but flexible mapping of parallel loop nests on the GPU. This mapping allows keeping the internal representation unaware of the full hierarchy implied by the OpenCL NDRange. I designed and implemented an induction variable substitution to enable parallelization of some loops, using an original scheme based on linear preconditions. I improved the parallelism detection in PIPS, especially the coupling with the reduction detection, and studied the impact of the two different algorithms on the code generation and on the performance on GPUs. I designed and implemented dedicated parallelization of reductions using the atomic operations available on GPUs. I designed and implemented a loop fusion phase in PIPS, including several different heuristics to favor a performing mapping onto GPUs. I improved the existing scalarization transformation in PIPS to keep the perfect nesting of loops. I identified three different schemes and analyzed the impact of scalarization in these cases. I conducted experiments to validate the individual impact of each of the transformations presented in this chapter. While most

concepts are well known, it is shown that for many transformations, the state-of-the-art scheme has to be adapted to the specific requirements of the massively parallel pieces of hardware that are GPUs.

PIPS offers a flexible framework, but the compilation flow among these individual transformations has to be driven to provide an automatic end-to-end solution, as shown in Figure 2.27 on page 58. The next chapter motivates and introduces the concepts of the programmable pass manager implemented in PIPS, which Par4All leverages to provide automated process driving all the transformation steps.

Chapter 5

Heterogeneous Compiler Design and Automation

Contents

5.1 Par4All Project . . . 138
5.2 Source-to-Source Transformation System . . . 140
5.3 Programmable Pass Managers . . . 141
5.3.1 PyPS . . . 142
5.3.2 Related Work . . . 149
5.3.3 Conclusion . . . 150
5.4 Library Handling . . . 151
5.4.1 Stubs Broker . . . 152
5.4.2 Handling Multiple Implementations of an API: Dealing with External Libraries . . . 153
5.5 Tool Combinations . . . 155
5.6 Profitability Criteria . . . 156
5.6.1 Static Approach . . . 157
5.6.2 Runtime Approach . . . 157
5.6.3 Conclusion . . . 158
5.7 Version Selection at Runtime . . . 158
5.8 Launch Configuration Heuristic . . . 159
5.8.1 Tuning the Work-Group Size . . . 159
5.8.2 Tuning the Block Dimensions . . . 162
5.9 Conclusion . . . 163

While previous chapters are focused on individual transformations implemented in the PIPS framework, this chapter addresses the issue of automating the whole compilation process, from the original source code to the final binary. To this end, we introduce Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] in Section 5.1. Par4All is an Open Source initiative that we propose to incubate efforts made around compilers to allow automatic parallelization of applications to hybrid architectures.

As hardware platforms grow in complexity, compiler infrastructures need more flexibility: due to the heterogeneity of these platforms, compiler phases must be combined in unusual and dynamic ways, and several tools may have to be combined to handle specific parts of the compilation process efficiently. The need for flexibility also appears in iterative compilation, when different phases orderings are explored.

In this context, we need to assemble pieces of software like compiler phases without having to dive into the tool internals. The entity in charge of this phase management in a standard monolithic compiler is called a *pass manager*. While pass managers usually rely on a statically defined schedule, the introduction of plug-ins in GCC and the current trends in compiler design showcased by LLVM pave the way for dynamic pass schedulers. Moreover, the heterogeneity of targets requires the combination of different tools in the compilation chain. In this context, automating the collaboration of such different tools requires defining a higher level *meta* pass manager.

The source-to-source aspect is key in this process, as explained in Section 5.2. A programmable pass manager is then introduced in Section 5.3.

Numerical simulations often make use of external libraries such as Basic Linear Algebra Subprograms (BLAS) or Fast Fourier transform (FFT) for example. Section 5.4 presents the handling of such specific libraries for mapping these computations on a GPU.

Section 5.5 gives insights on how different tools can collaborate. The profitability decision of offload computation is studied in Section 5.6. Solutions for selecting among different versions of a kernel at runtime are presented Section 5.7. Finally, Section 5.8 explores the impact of launch configuration on kernel performance.

5.1 Par4All Project

Recent compilers propose an incremental way for converting software toward accelerators. For instance, the PGI Accelerator [Wolfe 2010] or HMPP [Bodin & Bihan 2009] require the use of directives. The programmer must select the pieces of source that are

to be executed on the accelerator. He provides optional directives that act as hints for data allocations and transfers. The compiler then automatically generates a transformed code that targets a specific platform. JCUDA [Yan *et al.* 2009] offers a simpler interface to target CUDA from Java. There have been several attempts to automate transformations for OpenMP annotated source code to CUDA [Lee *et al.* 2009, Ohshima *et al.* 2010]. The GPU programming model and the host accelerator paradigm greatly restrict the potential of this approach, since OpenMP is designed for a shared memory computer. Recent work [Han & Abdelrahman 2009, Lee & Eigenmann 2010] adds extensions to OpenMP to account for CUDA specificity. These make programs easier to write, but the developer is still responsible for designing and writing communications code, and usually the programmer has to specialize his source code for a particular architecture. These previous works are presented with more detail in Section 2.2.

Unlike these approaches, Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] is an automatic parallelizing and optimizing compiler for C and Fortran sequential programs funded by the SILKAN company. The purpose of this source-to-source compiler is to integrate several compilation tools into an easy-to-use yet powerful compiler that automatically transforms existing programs to target various hardware platforms. Heterogeneity is everywhere nowadays, from the supercomputers to the mobile world, and the future seems to be more and more heterogeneous. Thus automatically adapting programs to targets such as multicore systems, embedded systems, high- performance computers and GPUs is a critical challenge.

Par4All is currently mainly based on the PIPS [Irigoin *et al.* 1991, Amini *et al.* 2011a (perso)] source-to-source compiler infrastructure and benefits from its interprocedural capabilities like memory effects, reduction detection, parallelism detection, but also polyhedral-based analyses such as convex array regions [Creusillet & Irigoin 1996b] and preconditions.

The source-to-source nature of Par4All makes it easy to integrate third-party tools into the compilation flow. For instance, PIPS is used to identify parts that are of interest in a whole program, and then Par4All relies on the PoCC [Pouchet *et al.* 2010a] or PPCG [Verdoolaege *et al.* 2013] polyhedral loop optimizers to perform memory accesses optimizations on these parts, in order to benefit from local memory for instance, as shown in Section 5.5.

The combination of PIPS' analyses together and the insertion of other optimizers in the middle of the compilation flow is automated by Par4All using a programmable pass manager (see Section 5.3) to perform whole-program analysis, spot parallel loops and generate mostly OpenMP, CUDA or OpenCL code.

To that end, we mainly face two challenges: parallelism detection and data transfer generation. The OpenMP directive generation relies on coarse grained parallelization and semantic-based reduction detection, as presented in Section 4.4. The CUDA and OpenCL targets add the difficulty of data transfer management. PIPS helps tackling this using convex array regions that are translated into optimized, interprocedural data transfers between host and accelerator as described in Chapter 3.

5.2 Source-to-Source Transformation System

Many previous successful compilers are source-to-source compilers [Bozkus *et al.* 1994, Frigo *et al.* 1998, Ayguadé *et al.* 1999, Munk *et al.* 2010] or based on source-to-source compiler infrastructures [Irigoin *et al.* 1991, Wilson *et al.* 1994, Quinlan 2000, ik Lee *et al.* 2003, Derrien *et al.* 2012]. They provide interesting transformations in the context of heterogeneous computing, such as parallelism detection algorithms (see Section 4.3), variable privatization, and many others including those presented in Chapter 4.

In the heterogeneous world, it is common to rely on specific hardware compilers to generate binary code for the part of the application intended to be run on a particular hardware. Such compilers usually take a C dialect as input language to generate assembly code. Thus, it is mandatory for the whole toolbox to be able to generate C code as the result of its processing.

In addition to the intuitive collaboration with hardware compilers, source-to-source compilers can also collaborate with each other to achieve their goal, using source files as a common medium, at the expense of extra conversions between the Textual Representation (TR) and the IR. Figure 5.1 illustrates this generic behavior and, in Section 5.5, the use of external polyhedral tools for some loop nest optimizations is presented. Moreover, two source-to-source compilers written in the same infrastructure can be combined in that way. For instance, an SIMD instruction generator has been used for both the generation of SSE instructions on Intel processors and enhancing the code generated by the CUDA/OpenCL generator presented in Chapter 4.

More traditional advantages of source-to-source compilers include their ease of debugging: the IR can be dumped as a TR at anytime and executed. For the same reason, they are very pedagogical tools and make it easier to illustrate the behavior of a transformation.

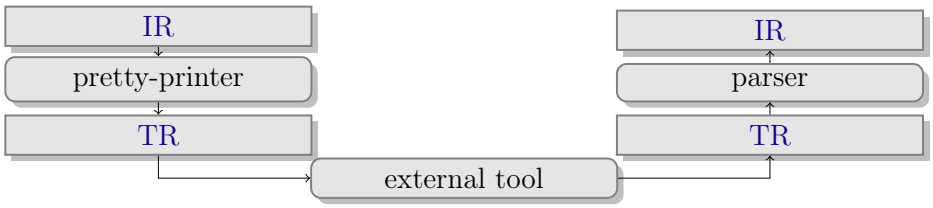


Figure 5.1: Source-to-source cooperation with external tools.

5.3 Programmable Pass Managers

The continuous search for performance leads to numerous different hardware architectures, as showcased by current trends in heterogeneous computing. To use these architectures efficiently, new languages and paradigms are often introduced, but they typically target only specific architectures. For instance AVX intrinsics target vector registers of recent $x86$ processors, OpenMP directives target multicores, and most noticeably CUDA targets Nvidia's GPU. It is difficult to master all these language-hardware bindings without losing control of the original code. Thus compilers play a key role for "filling the gap" between generic sequential languages and specific parallel languages [Asanović *et al.* 2006]. Because of the diversity of targeted architecture, flexibility and retargetability are critical properties for compiler frameworks that must keep up with the ongoing work of hardware designers and founders. Also, applications targeting heterogeneous architectures, e.g. GPGPU with an $x86$ host, raise new problems such as the generation of different codes in different assembly languages, remote memory management, data transfer generations, etc., thus requiring new functionalities that are not available in current mainline compilers.

A recurrent point when compiling for heterogeneous platforms is the need to dynamically create new functions that will be mapped onto specific pieces of hardware, using a transformation called *outlining*. This transformation dynamically creates new functions and new compilation units depending on the processed code, so it does not fit well into static pass managers.

Additionally, iterative compilation [Goldberg 1989, Kulkarni *et al.* 2003]—the process of iteratively transforming, compiling and evaluating a program to maximize a fitness function—is more and more considered as an alternative to standard program optimizations to solve complex problems, but it requires a dynamic reconfiguration of the compilation process. In a compiler infrastructure, the latter is managed by the *pass manager*. Because of the much more complicated compilation schemes, this pass manager must be flexible and provide ways of overtaking the traditional hard-coded pass sequence to al-

low compiler developers to manipulate the interactions between passes and the compiled program dynamically.

This section is organized as follows: Section 5.3.1 overviews Pythonic PIPS (PyPS), the pass manager implemented in the PIPS source-to-source compiler framework. It involves an API with a high-level abstraction of compilation entities such as analyses, passes, functions, loops, etc. Building upon the fact that an API is relevant only when used extensively, some cases of use are mentioned in Section 5.3.1.5, with the summarized compilation scheme of distinct compiler prototypes using this model. Finally, related works is presented in Section 5.3.2.

5.3.1 PyPS

A formal model description is available in [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso), Guelton 2011a]. Multiple operators are proposed to describe transformations, error handling, and different pass-combination schemes. Guelton also improves it and provides a significantly extended version in his PhD thesis [Guelton 2011a]. The assumption made in PyPS is that the resources required to execute a given pass are transparently provided by an underlying component. In PIPS, the consistency manager PIPS-Make is present. It takes care of executing the analysis required by a compiling pass and keeps track of any change that invalidates the results of an analysis.

Instead of introducing yet another new domain-specific language to express these operators, we benefit from existing tools and languages, taking advantage of the similarity with existing control flow operators. Indeed the transformation composition is similar to a function definition; the failsafe operator can be implemented using exception handling and the conditional composition performs a branching operation. This leads to the idea of using a general-purpose language coupled with an existing compiler infrastructure, while clearly separating the concerns.

5.3.1.1 Benefiting from Python: *on the shoulders of giants*

Using a programming language to manage pass interactions offers all the flexibility needed to drive complex compilation processes, without the need of much insight on the actual IR. Conceptually, a scripting language is not required. However, it speeds up the development process without being a burden in terms of performance as all the time is spent in the transformations themselves.

Some approaches introduced dedicated language [Yi 2011] for the pass management, but we rather follow the well-known Bernard de Chartres' motto: "on the shoulders of giants" and thus use Python as our base language. This choice proved to be better than expected by not only providing high-level constructions in the language but also by opening access to a rich ecosystem that widens the set of possibilities, at the expense of a dependency on the Python interpreter.

5.3.1.2 Program Abstractions

In the model presented in [Guelton *et al.* 2011b (perso), Guelton *et al.* 2011a (perso)], transformations process the program as a whole. However, transformations can proceed at lower granularity: *compilation unit level,*[1] *function level*[2] or *loop level*:

- at compilation unit level, decisions based upon the target can be made following the rule of thumb "one compilation unit per target.' This helps drive the compilation process by applying different transformations to different compilation units;
- most functions that consider stack-allocated variables work at the function level: *common subexpression elimination*, *forward substitution* or *partial evaluation* are good examples;
- a lot of optimizations are dedicated to loop nests, without taking care of the surrounding statements. This is the case for polyhedral transformations.

Interprocedural transformations, like building the *callgraph*, require knowledge of the whole program, or can be improved by such knowledge (e.g. *constant propagation*), thus the program granularity is still relevant.

The class diagram in Figure 5.2 shows the relations between all these abstractions. These—and only these—are exposed to the pass manager. The *builder*, in charge of the compilation process, is introduced in Section 5.3.1.4.

5.3.1.3 Control Structures

The main control structures involved are introduced here. The complete formal description of the operators is found in [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso), Guelton 2011a].

1. A source file in C.
2. Also referred as "module level."

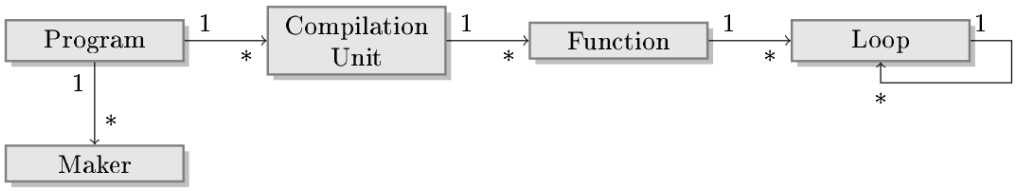


Figure 5.2: PyPS class hierarchy (source [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso)]).

```
# first argument is the argument name, second is the default value
if conditions.get('if_conversion',False):
  module.if_conversion()
```

Figure 5.3: Conditionals in PyPS.

```
for kernel in terapix_kernels:
  kernel.microcode_normalize()
```

(a) Iteration over selected modules

```
for l in module.inner_loops(): l.unroll(4)
```

(b) Iteration over inner loops

```
for pattern in ["min","max","adds"]:
  module.pattern_recognition(pattern)
```

(c) Multiple instruction selection.

Figure 5.4: For loop is a control structure commonly involved in PyPS

Conditionals Conditionals are used when transformation scheduling depends on user input or on the current compilation state. Figure 5.3, extracted from the SIMD Architecture Compiler (SAC) compiler [Guelton 2011a], illustrates the use of conditionals to implement the `-fno-pass-name/-fpass-name` switch as in GCC.

For Loops For loops are used to perform repetitive tasks (see in Figure 5.4):

1. applying a transformation to each function or loop of a set;
2. applying a transformation iteratively with varying parameters.

Figure 5.4a illustrates a basic iteration over selected modules in the compiler for Terapix [Guelton 2011a]. Figure 5.4b from the SAC compiler [Guelton 2011a] shows how to unroll all inner loops by a factor of four. Finally, in the SAC compiler, early pattern recognition is also performed. Figure 5.4c demonstrates the use of loops to apply this pass for

```
# Try most efficient parallelization pass first:
try: module.coarse_grain_parallelization()
except:
# or downgrade to loop_distribution
  try: module.loop_distribution()
  except: pass # or do nothing
```

Figure 5.5: Using exceptions to adapt the compilation process.

```
try:
  while True:
    module.redundant_load_store_elimination()
    module.dead_code_elimination()
except:
  # an exception is raised by the previous
  # passes when nothing is cleaned up
  pass
```

Figure 5.6: Searching fix point.

various patterns.

Exceptions A PyPS exception can be raised in three situations:

- a user input is invalid;
- a pass fails;
- an internal error happens.

Each of them has a different meaning and can be caught or propagated depending on the pass manager developer goals. A sample usage is given in Figure 5.5. Figure 5.6 illustrates the error propagation operator [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso)] on a sequence of two passes in a loop: `dead_code_elimination` is applied if `redundant_load_store_elimination` succeeds.

While Loops While loops are useful to achieve a goal within an unknown number of iterations. In the SAC compiler, while loops are combined with exception handling to reach a fix point. Figure 5.6 shows this pattern on a data transfer optimizer.

5.3.1.4 Builder

As code for heterogeneous targets is generated in the form of several source code files, a final compilation step is needed to get the final binary (or binaries). *Stricto sensu*, this is not the role of the pass manager. However, as it is in charge of the source generation, it has information about the mapping of the different files onto the targets and hence about which specific compilers to use. Moreover, complex transformations for hybrid or distributed architectures might require the addition of runtime calls. They are frequently shipped as shared library and thus the final link step is dependent on the transformations performed. For instance, if some FFT operations using *FFTW3* are automatically transformed into the equivalent GPU operations with *CuFFT*, and the link process has to be adjusted. The pass manager can keep track of this. As a consequence, it is useful to provide primitives to drive the final compilation process.

The point here is to associate an automatically generated Makefile to the generated sources. This makefile is a template that holds generic rules to compile CUDA/OpenCL/AVX/OpenMP code, and it is filled with the proper dependencies to run the compilation with appropriate compiler/linker flags. This approach offers the advantage of relieving the user by delegating most of the difficulties to third-party tools, in a source-to-source manner.

5.3.1.5 Heterogeneous Compiler Developements

Several real-world architectures have benefited from the flexibility of the proposed pass manager:

Terapix The Terapix architecture [Bonnot *et al.* 2008] is a FPGA-based accelerator for image processing developed by Thales. It achieves high throughput and low energy consumption thanks to a highly specialized architecture and a limited ISA. In particular it uses VLIW instructions written in a dedicated assembly. The compilation scheme summarized in Figure 5.7 shows how the name of generated functions takes part to the process. An external VLIW code compactor is used to compact the generated code, enforcing tool reuse. The generated makefile calls a C compiler for the host code and a specific assembler for the accelerator. This was implemented by Guelton in 2011 and described in detail in his PhD thesis [Guelton 2011a].

The whole compilation scheme is written with 411 Python Source Lines Of Code (SLOC).

```
def terapix(module):
  counter=0
  kernels=list()
  for loop in module.loops():
    try:
      # Generate a new name from parent and a numeric suffix
      kernel=module.name+str(counter)
      loop.parallelize()
      module.outline(label=loop.label,name=kernel)
      # Add the kernel object from its name to our kernel list:
      kernels.append(workspace[kernel])
    except:pass
  for kernel in kernels:
    kernel.terapix_assembly()
    kernel.pipe("vliw_code_compactor")
```

Figure 5.7: Terapix compilation scheme.

```
def autotiler(loop):
  if loop.loops():
    try: loop.tile(...)
    except: map(autotiler,loop.loops())
  else: loop.unroll(...)
```

Figure 5.8: SAC compilation scheme extract.

SAC SAC is an automatic generator of multimedia instructions that can output either AVX, SSE or NEON intrinsics. It is presented in [Guelton 2011a]. It combines polyhedral transformations at the loop level and pattern matching at the function level to generate vector instructions. Figure 5.8 shows an extract of its compilation scheme in charge of tiling as many loops as possible.

The whole compilation scheme is 220 Python SLOC.

An Iterative Compiler PyPS makes it easier to build an iterative compiler. Cloning the workspace (saving the internal state of the compiler), iterating over a parameter domain for transformations such as *loop unrolling,* is straightforward. A genetic algorithm was implemented to take advantage of PyPS abstraction to test different transformation schedules, benchmark the resulting applications and so forth. Additionally, each iteration is embedded in a remote process, *à la* Erlang, which gives a free performance boost on distributed multicore machines. The whole compiler is 600 Python SLOC.

```
def openmp(module,distribute=True):
  #find reductions
  module.reduction_detection()
  #find private variables
  module.privatize()
  #find parallelism or raise exception
  try:
    module.parallelize()
  except:
    if distribute:
      #loop distribution
      module.distribute()
  finally:
    #directive generation
    module.generate_omp_pragma()
```

Figure 5.9: OpenMP compilation scheme.

Par4All Par4All (presented in Section 5.1) is an open-source initiative to incubate efforts made around compilers to allow automatic use of hybrid architectures. It relies heavily on PyPS capabilities. The compilation process is driven by user switches that select different back ends:

- the OpenMP back end uses a classic compilation scheme summarized in Figure 5.9, but still uses the basic pass manager. Two parallelization algorithms are tried successively regardless of the method applied, directives are generated.
- CUDA and OpenCL back ends rely on the parallelization algorithms used for OpenMP and add complex handling in the pass management process to address cases like C99 idioms unsupported by the Nvidia compiler or Fortran to CUDA interface. The subtle differences between the two back ends are mostly handled at runtime. Some other few differences are addressed in the process of pass selection using branches wherever required.
- SIMD targets SSE, AVX, NEON and is eligible as a stand-alone target or as a post-optimization step for OpenMP and OpenCL back ends. A high-level entry point is presented and hooked in the compilation process flow depending on command line options.
- the SCMP back end implemented by Creusillet [Creusillet 2011] generates task-based applications for an heterogeneous MultiProcessor System-on-Chip (MP-SoC) archi-

tecture designed by CEA for the execution of dynamic or streaming applications [Ventroux & David 2010]. To generate intertask communications, the compiler relies on a PIPS phase also used by the CUDA back end. The resulting code is then post-processed by a phase directly coded in Python for further adaptation to a runtime specific to the SCMP architecture.

The Par4All project was started in 2009, with a first release in July 2010. It is already a large project. It ensures the portability of the original source code to several targets with different options. It is a perfect case study for validating the concepts that were proposed within PyPS, and its implementation is actually made easier by the reuse and the easy combination of the different constructs available in PyPS.

Par4All represents around ten thousand lines of Python source code, including many other features than the pass management. For the pass management part, the core system takes only seventy-one SLOC, the OpenMP part sixteen, the SCMP part thirty-eight, and the CUDA/OpenCL part 117. This one is larger than expected because the transformation of Fortran or C99 to CUDA requires much special handling.

5.3.2 Related Work

In traditional compilers such as GCC, a compilation scheme is simply a sequence of transformations chained one after the other and applied iteratively to each function of the original source code [Wikibooks 2009]. This rigid behavior led to the development of a plug-in mechanism motivated by the difficulty of providing additional features to the existing infrastructure. Samples of successful plug-ins include "dragonegg" and "graphite." GCC's shortcoming led to the development of LLVM [Lattner & Adve 2004], which addresses the lack of flexibility by providing an advanced pass manager that can change the transformation chaining at runtime.

The source-to-source ROSE Compiler [Quinlan 2000, Schordan & Quinlan 2003] does not include any pass manager but provides full access to all analyses, passes and IR through a clean C++ API. A Haskell wrapper provides scripting facilities. Although this approach offers good flexibility (any kind of pass composition is possible), it offers neither clean concept separation nor any abstraction. This leads to complicated interactions with the compiler. The addition of the scripting wrapper does not solve the problem because it is a raw one-to-one binding of the full API. The Poet [Yi 2011] language, designed to build compilers, suffers from the same drawbacks, as there is no clean separation between compiler internals and pass management: everything is bundled all together.

Compiler directives offer a non-intrusive way to drive the compilation process, much as a pass manager does. They are extensively used in several projects [Kusano & Sato 1999, Donadio *et al.* 2006, NVIDIA, Cray, PGI, CAPS 2011] to generate efficient code targeting multicores, GPGPU or multimedia instruction set. They can specify a sequential ordering of passes, but they do not provide extra control flow, and may have complex composition semantics.

Transformation recipes, i.e., specialized pass manager implementations, are presented in [Hall *et al.* 2010]. This section emphasizes the need for a common API to manipulate compiler passes. It leads to a Lua-based[3] approach proposed in [Rudy *et al.* 2011] to optimize the generation of CUDA codes. It provides bindings in this scripting language for a limited number of parametric polyhedral loop transformations. The addition of scripting facilities makes it possible to benefit from complex control flow, but the interface is limited to polyhedral transformations. The approach is procedural and the API allows access to analysis results such as control flow graphs, dependence graphs, etc.

The need for iterative compilation led to the development of an extension of the GCC pass manager [Fursin *et al.* 2008]. It provides a GCC plug-in that can monitor or replace the GCC pass manager. It is used to apply passes to specific functions but the ordering is still sequential. The underlying middleware, ICI, is also used in [Fursin & Cohen 2007] where an interactive compiler interface is presented. The authors enforce the need for clear encapsulation and concept separation, and propose an Extensible Markup Language (XML) based interface to manage the transformation sequencing.

This approach basically turns a compiler into a program transformation system, which is an active research area. FermaT [Ward 1999] focuses on program refinement and composition, but is limited to transformation pipelining. Stratego [Olmos *et al.* 2005] software transformation framework does not allow much more than basic chaining of transformations using pipelines. CIL [Necula *et al.* 2002] provides only a flag-based composition system that is activation or deactivation of passes in a predefined sequence without taking into account any feedback from the processing.

5.3.3 Conclusion

This section introduced PyPS, a pass manager API for the PIPS source-to-source compilation infrastructure. This compilation framework is reusable, flexible and maintainable, which are the three properties required to develop new compilers at low cost while meet-

3. Lua is a scripting language.

ing time-to-market constraints. This is due to a clear separation of concepts between the internal representation, the passes, the consistency manager and the pass manager. This clear separation is not implemented in GCC and is not fully exploited either in research compilers, although pass management is becoming a research topic. Five specific compilers and other auxiliary tools are implemented using this API: OpenMP, SSE, SCMP, and CUDA/OpenCL generators, and an optimizing iterative compiler. We show how it is possible to address retargetability by combining different transformation tools in an end-to-end parallelizing compiler for heterogeneous architectures. Moreover, it is done in an elegant and efficient way relying on the clear separation of concepts that we identified and on facilities offered by Python ecosystem.

This work was previously published in [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso)].

5.4 Library Handling

Developers often make use of external libraries, highly optimized for dedicated computing such as linear algebra, image processing, or signal processing. The issue, in the context of automatic parallelization of such programs, is that most of the time the source code of the library is not available either because it is a proprietary piece of code or simply because there is only the binary present on the computer. Even if the code were available, it may represent a huge amount of code to process. Moreover, it would be probably so tied to a given CPU architecture that any transformation for the GPU is far beyond what may be achieved by an automatic parallelizer.

PIPS is an interprocedural compiler. It analyses the whole call tree and propagates results of analyses to call sites. A current limitation of the implementation is that the whole call tree is required, i.e., the source code of all leave functions must be available for processing. As a major consequence, PIPS cannot process a source code that makes use of external libraries. The source is not always available or optimized for the CPU and too complicated to be properly analyzed and automatically transformed to a GPU version.

On the other hand, many libraries are available for GPUs and provide similar if not equal functionalities. This section covers how the handling of libraries is performed in the compilation chain. First, an automatic way of providing stubs on demand to the compiler is presented in Section 5.4.1. Then different techniques are used to replace the CPU library code with a GPU equivalent implementation in the backend (see Section 5.4.2).

5.4.1 Stubs Broker

As an interprocedural compiler, PIPS stops as soon as it has to analyze a function whose source code is not available. To avoid the problem, the usual process requires providing stubs to the compiler. Stubs are fake functions, with the exact same signature, that mimic the effects of a library call on the memory, since PIPS uses this information for parallelization purpose, for example. Memory effects can be of *read* or *written* types, and are handling part or the whole of the call arguments, but they can also involve manipulating library-internal variables generating side effects.

In the context of the Par4All tool, the goal is to provide a seamless experience to the end user. The user writes his own code and provides it to the compiler, with a flag that indicates which library is in use. Par4All includes a generic mechanism to handle libraries and to provide retargetability. We have historically called this mechanism the stubs broker.

This piece of code is invoked by many steps of the compilation scheme. Firstly, when PIPS requires a source code for a function, the stubs broker is in charge of retrieving the required code and feeding PIPS with it. Then, when it is time to call the backend compiler, it has the responsibility to retrieve the actual implementation by providing either a link flag to a binary library or a source code to compile.

The process is the following:

1. PIPS needs to analyze a function with a call site to a missing function.
2. An exception is raised with the name of the missing function.
3. The exception is passed to the stubs broker handler.
4. The stubs broker handler goes through the list of available brokers and requires a stub for the missing function.
5. When a broker has a stub for this function, the broker handler registers the stub and the broker that has given it. There are also some meta-data associated with the stub, such as the different implementations available (for instance CUDA or OpenCL).
6. The code for the stub is given to PIPS, and the analyses continue. The stub has been flagged in order not to be modified in any way by PIPS, since it is intended to be replaced by another implementation in the backend compiler. For example, inlining of this stub is not desirable. The meta-data are also used to indicate that communications have to be generated in case a given stub corresponds to a GPU implementation.

7. At the end of the process, the resulting code is passed to the Par4All back end. At that time the back end is fed with the parameters given by the broker handler for the retrieved stubs. A new source code can be added, or some compilation flag to link against some libraries could be added.

This flow is represented in Figure 5.10. This scheme has been validated with the implementation of a parallelizing Scilab-to-C compiler (the Wild Cruncher from SILKAN). A specific compiler is in charge of producing C code from Scilab scripts. The resulting code makes use of a large number of runtime functions to interact with Scilab internal (e.g. for the display) or to perform some specific optimized computations (e.g. particular numeric solvers).

In this context, feeding PIPS with thousands of unused functions was not an option as the processing would take hours! Moreover, the source code is not available; the runtime is distributed as a binary shared library.

Instead, stubs are provided; most of them are generated automatically while some are hand-written. The stubs broker interface is simple enough to ease the implementation of any new custom broker. Moreover, Par4All exposes entry points to add dynamically a new stubs broker in addition to the default ones. For the Scilab runtime, a third-party broker has been written to handle the huge number of stubs included. Less than fifty lines of Python were sufficient to provide a complete stubs broker.

5.4.2 Handling Multiple Implementations of an API: Dealing with External Libraries

A broker is not in charge of a given library but rather of a *feature*. For instance, a broker can be in charge of the FFT, whatever libraries are included. The reason is that while a client code may use one particular CPU FFT library, we ought to replace it with another library on the GPU. This organization is emphasized in Figure 5.10; the broker handler is in charge of orchestrating the whole process.

Let us use an example based on the FFTW library [Frigo & Johnson 2005] with the cosmological simulation named Stars-PM [Aubert *et al.* 2009 (perso)] presented in detail in Section 3.1. The FFT is used during the third step exposed in Figure 3.3, page 65. The involved calls to FFTW are exhibited in Figure 5.11. These two calls include only one parameter: the *plan*. It is an opaque FFTW structure. This structure has to be initialized with all the information required to perform the operation. It includes the pointers to the

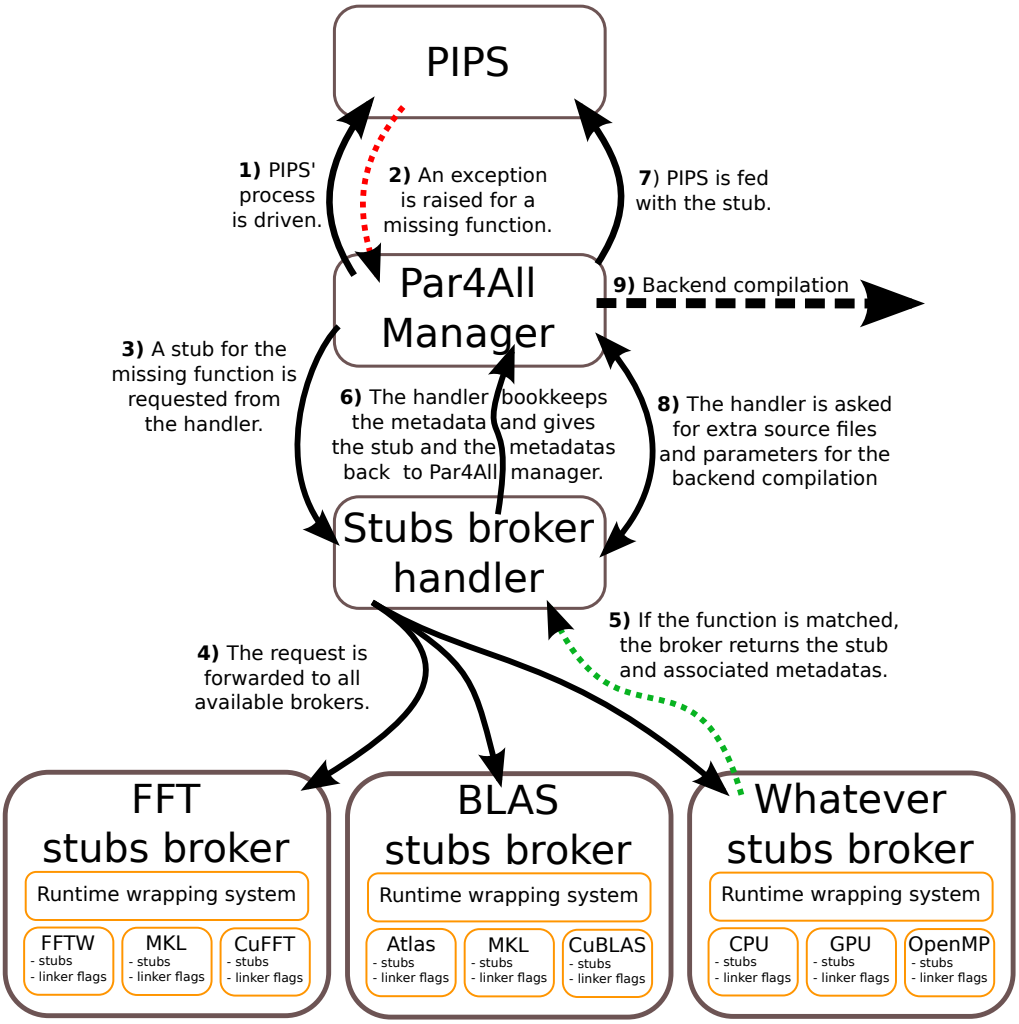


Figure 5.10: The brokers infrastructure and compilation flow in Par4All.

source and destination arrays, the size of the arrays, the direction for the FFT, and other parameters. The initialization of the two involved plans is shown in Figure 5.12.

Par4All is launched with a flag `-fftw` that indicates the use of the FFTW library, but it could also be automatically detected. The FFT broker then automatically feeds PIPS when it complains about the FFTW missing functions, as explained in the previous section. At the end of the source-to-source processing, the Par4All stubs-broker manager adds sources and/or parameters to the backend process. For instance, it may add some C files that provide a wrapping feature around FFTW to retarget to the CUDA equivalent, i.e., CuFFT. It may also add a flag for the linker. For FFTW it would be `-lcufft`.

```
void potential(int NP,
               int histo[NP][NP][NP],
               float dens[NP][NP][NP],
               float cdens[NP][NP][NP][2]) {

  real2Complex(cdens,dens); // Convert to complex
  fftwf_execute(fft_forward); // go in Fourier's space
  fft_laplacian7(cdens); // Solve in Fourier's space
  fftwf_execute(fft_backward); // back from Fourier's space

}
```

Figure 5.11: The original source code for the `potential` step that involves two calls to FFTW library.

```
static fftwf_plan fft_forward;
static fftwf_plan fft_backward;

void potential_init_plan(int NP, float cdens[NP][NP][NP][2]) {

  fft_forward = fftwf_plan_dft_3d(NP,NP,NP,
                                  (fftwf_complex*)cdens,
                                  (fftwf_complex*)cdens,
                                  FFTW_FORWARD,
                                  FFTW_ESTIMATE);
  fft_backward = fftwf_plan_dft_3d(NP,NP,NP,
                                   (fftwf_complex*)cdens,
                                   (fftwf_complex*)cdens,
                                   FFTW_BACKWARD,
                                   FFTW_ESTIMATE);
}
```

Figure 5.12: FFTW library requires that a *plan* is initialized. Here in the original source code, two plans are initialized for the `potential` code presented in Figure 5.11.

5.5 Tool Combinations

As heterogeneity grows, it is less likely to have one compiler framework that fulfills all needs. Some frameworks can have an internal representation that better fits some specific optimizations but not all. For example, PIPS benefits from interprocedural analyses, but inherently lacks capabilities that others have, such as PoCC [Pouchet *et al.* 2010a], Pluto [Bondhugula *et al.* 2008b], or PPCG [Verdoolaege *et al.* 2013]. These are some optimizers dedicated to the polyhedral model, but restricted to a particular class of programs.

At some point. depending on the input and the target, it is interesting to combine the right set of tools to perform the best chain of analyses and transformations.

Some experiments were conducted in Par4All to make use of PoCC [Pouchet *et al.* 2010a] at some point of the process. PIPS takes care of the first phase, in which a detection of static control parts is performed. They are the code sections in which all loops are `DO` loops whose bounds are functions of parameters, numerical constants and enclosing loops iterators as defined in [Feautrier 1991]. PIPS benefits from interprocedural capabilities that allow it to assert some properties on function calls, like detecting that the called function is pure. Indeed, PoCC assumes that any function call in a code portion flagged as static control is pure.

The code sections detected by PIPS as static control code can be processed by PoCC, applying various optimizations. Currently no optimizations in PoCC are dedicated to the GPU. The extension of Pluto, the polyhedral engine in PoCC, to optimize a loop nest for the GPGPU computing is left as future work.

Another polyhedral optimizer that includes transformations that target GPUs is PPCG [Verdoolaege *et al.* 2013]. This is a very recent project but it seems very promising and complementary to an interprocedural compiler like PIPS. Other compilers take CUDA kernels as an input like Yang et al. [Yang *et al.* 2010].

A future piece of work currently identified is to pass kernels to a kernel optimizer to optimize them for different architectures, and get back different versions of a kernel to be incorporated in the final binary with some mechanisms to select the right one at runtime, using one of the methods presented in Section 5.7.

To achieve all these combinations of different tools, the source level, eventually augmented with semantic information by pragmas, is perfectly suitable as explained in Section 5.2, page 140. The C language is a widely used language and it has been extended many times using proprietary extensions, e.g. CUDA, or using pragmas to include meta-informations in the code, e.g. OpenMP [OpenMP Architecture Review Board 2011].

5.6 Profitability Criteria

Deciding whether to offload a computation on the GPU or not is not a trivial problem. Two approaches can be compared. The first one makes a decision based on static information. The second postpones the decision until runtime, benefiting from more information, but at the expense of cost and complexity at runtime.

5.6.1 Static Approach

The static approach requires an estimation of the execution time on both the GPU and the CPU. Moreover, the time to transfer the data has to be estimated too. This transfer can be accounted for both computations; it depends on the current data location. If the data are located in the GPU memory, even if the computation is a little bit faster on the CPU, the time required to transfer the data may be in favor of offloading the computation to the GPU. This situation is mentioned in Section 3.7, page 84. A portion of sequential code is offloaded to the GPU using only one thread to avoid a memory transfer.

Assuming the data location could be known at compile time, the decision still requires evaluating the time required to process the kernel both on CPU and GPU. To this end, there is no single solution that would perfectly address this issue; the execution time estimation is a problem.

Even if the loop bounds are known, there is no single formula out there that would fit all the different architectures. Leung et al. proposed a parametrized formula [Leung 2008] to match a class of GPUs. The model is quite simple and uses off-line a set of "representative" micro-benchmarks to compute the instruction throughput of both the CPU and the GPU. This throughput is then multiplied by the number of instructions in the kernel and the number of threads. The GPU-estimated time also takes into account a constant startup time (a few microseconds) and the memory transfer time. Another assumption is that branches are taken half of the time. They show that this model is accurate enough for simple matrix multiplication but it is doubtful that it can predict as accurately a wide range of kernels. More precise models have been introduced [Kothapalli *et al.* 2009, Liu *et al.* 2007, Hong & Kim 2009, Baghsorkhi *et al.* 2010], but they are limited in the level of detail taken into account like diverging control flow, memory bank conflicts, and SIMD pipeline delay. Moreover, they are quickly outdated by the evolution of architectures, for instance the introduction of L1/L2 caches in Fermi (see in Section 2.4.5).

Par4All does not include any performance model specific for GPU architectures. It can use a generic complexity analysis [Zhou 1992] from PIPS that can be used to avoid parallelizing loops with small amount of computation.

5.6.2 Runtime Approach

Decisions taken based on runtime information are more accurate. For instance the StarPU runtime system includes many scheduler algorithms, some able to track past

executions of a kernel and extrapolate the performance based on a nonlinear regression $a \times size^b + c$. The scheduler can then take decisions about whether to schedule a kernel on an accelerator or not, based on a performance model, the actual locality of the data, but also on the current availability of the resources. Section 6.2 gives more information about this scheme.

5.6.3 Conclusion

The profitability criteria is still an unsolved issue. Several approaches may be used. The next section presents a solution that involves runtime selection between multiple versions, based on off-line profiling.

5.7 Version Selection at Runtime

Another approach was explored by Dollinger and Loechner [Dollinger & Loechner 2011]. At compile time they execute multiple versions of a kernel using a large set of possible iteration sets, and record the results to postpone until runtime the scheduling decision based on the actual size of the loop bound. They show that one kernel version can be more adapted for small dataset while another version provides better performance for a large dataset. The drawback is that the target accelerator must be available at compile time for execution.

This approach can be related to and shares the same basic idea as the ATLAS library [Whaley & Dongarra 1998] that auto-tunes for the target platform by running multiple versions once at installation.

Another path is to simulate the execution of a kernel to predict the performance on various architectures. Recently, various simulators for Nvidia G80 [Bakhoda *et al.* 2009a, Collange *et al.* 2010], for AMD Evergreen [Ubal *et al.* 2007], or multiple architectures [Bakhoda *et al.* 2009b, Diamos *et al.* 2010] have been introduced. However, these simulators can take a very long time.

The main limitation of this approach is probably that some kernels need data initialized in a particular way, for instance because of data-dependent branches; then this method requires a representative set of data available at compile time, which is not always possible, or not in all possible sizes. This method is complementary with StarPU middleware [Augonnet *et al.* 2010b]. The latter accepts multiple kernel versions, with a custom callback

function, to decide which one should be executed considering the actual arguments and the available architecture.

5.8 Launch Configuration Heuristic

CUDA and OpenCL require expressing the iteration set in the form of a three-dimensional grid with two levels of refinement: the main grid is itself split into three-dimensional blocks of threads. This is the NDRange in OpenCL terminology introduced in Section 2.3.2.2, page 32.

This grid has to be configured manually by the programmers. To get the best performance, they usually have to use a trial-and-error approach. Moreover, finding the best configuration is architecture dependent and data size dependent, and thus no generic formula is available.

5.8.1 Tuning the Work-Group Size

The block (or work-group) size has a potentially large impact on performance. An obvious practice is to have a block size multiple of the SIMD width; for instance it is always thirty-two on currently released Nvidia architectures. The CUDA best practice guide [Nvidia 2012a] mentions that the number of threads should be a multiple of sixty-four in order for the compiler and the hardware thread scheduler to schedule instructions as optimally as possible to avoid register memory bank conflicts. It also advises defining a configuration that allows using several smaller thread blocks rather than one large thread block per multiprocessor. This is particularly beneficial to kernels that often call `__syncthreads()`. Finally it states that a good choice is usually between 128 and 256 threads per block in general.

A key point in the CUDA programming guide is presented as maximizing occupancy. Even if Volkov showed that it is not always the metric to maximize to get the best performance [Volkov 2010], it is at least a good starting point for the kernels that can be generated automatically when using Par4All. Moreover, this metric has the merit to be easily maximized, adjusting the launch configuration for a given kernel and a given architecture. Since all these parameters are known at runtime, the launch configuration can be dynamically adjusted. The occupancy is the ratio of the number of threads ready to execute considering the resource used by the kernel and the block size to the maximum number of threads the hardware scheduler can handle. It is intended to ensure that the

hardware scheduler has enough threads available to hide the memory latency with other thread operations.

The occupancy is computed with the following parameters:

- The current GPU architecture capabilities: shared memory size *SharedSize*, number of registers per CU *NRegisterPerCU*, maximum number of threads per CU *MaxThreadsPerCU*, maximum number of threads per block *MaxThreadsPerBlock*, maximum number of block per CU *MaxBlocksPerCU*.
- The resource usage of the kernel: number of registers per thread *KernelRegistersPerThread* and requested shared memory size per block *KernelSharedPerBlock*. These are given by the OpenCL and CUDA APIs.

For a candidate block size *BlockSize*, the occupancy can be computed in the following way. First, considering the shared memory usage, the limit in term of number of blocks per CU is

$$SharedMemBound = \frac{SharedSize}{KernelSharedPerBlock}$$

Then considering separately the register usage per thread, the limit of number of blocks per CU is

$$RegistersBound = \frac{NRegisterPerCU}{KernelRegistersPerThread \times BlockSize}$$

For the candidate block size, the maximum number of blocks per CU is

$$BlockThreadsBound = \frac{MaxThreadsPerCU}{BlockSize}$$

The effective number of blocks schedulable is then the lowest of the previous computed values:

$$SchedulableBlocks = \min(SharedMemBound, RegistersBound, BlockThreadsBound, MaxBlocksPerCU)$$

Finally, the occupancy is then the expected ratio of the number of threads schedulable per CU over the capacity of the scheduler:

$$Occupancy = \frac{SchedulableBlocks \times BlockSize}{MaxThreadsPerCU}$$

To ensure that the resource limits are not overwhelmed, both OpenCL and CUDA API

function BLOCKSIZEFOROCCUPANCY(*MaxSize*,*NRegisterPerCU*,
KernelRegistersPerThread,*MaxThreadsPerCU*,*WarpSize*,
SharedSize,*KernelSharedPerBlock*)
// Minimum block size, all Nvidia GPUs currently have *WarpSize*=32
$BlockSize = WarpSize$
$BestBlockSize = BlockSize$
$BestOccupancy = 0$
$SharedMemBound = SharedSize/KernelSharedPerBlock$
while $BlockSize \leq MaxSize$ **do**
$RegistersBound = \frac{NRegisterPerCU}{KernelRegistersPerThread \times BlockSize}$
$BlockThreadsBound = \frac{MaxThreadsPerCU}{BlockSize}$
$SchedulableBlocks = \min(SharedMemBound,$
$RegistersBound, BlockThreadsBound)$
$Occupancy = SchedulableBlocks \times BlockSize$
if $Occupancy > BestOccupancy$ **then**
$BestOccupancy = Occupancy$
$BestBlockSize = BlockSize$
end if
$BlockSize = BlockSize + WarpSize$
end while
return *BestBlockSize*
end function

Figure 5.13: Computing the block size that maximizes the occupancy for a given kernel and a given GPU.

provide the maximum block size possible for a given kernel and a given device. The algorithm to find the block size that would maximize the occupancy, implemented in the Par4All runtime, is shown in Figure 5.13.

However, maximizing the occupancy is nonsense if the iteration set is too small and leads to too few blocks globally. For instance, using a Tesla C2050 shipped with fourteen CU, at least twenty-eight and preferably forty-two blocks have to be launched to expect good usage of the GPU.

$$MaxSize = \min\left(BlockSizeForOccupancy, \left\lceil \frac{GlobalWorkSize/NCU}{WarpSize} \right\rceil\right)$$

This formula also ensures that the block size is rounded to the next multiple of the warp size. While the algorithm *BlockSizeForOccupancy* presented in Figure 5.13 requires being processed only the first time a kernel is executed on a given device, this last formula depends on the *GlobalWorkSize* (i.e., the iteration set), and thus has to be checked each

time a kernel is launched.

This is particularly visible when a triangular loop nest has an external sequential loop sequential. Then for each iteration of the outer loop, a kernel that maps the inner loop is launched. The original code could look like this:

```
for(i=0;i<N-1;i++) // Sequential
 for(j=i;j<N;j++)  // Parallel
   // ...
```

The kernel corresponding to the inner loop has a constant block size that maximizes the occupancy for a given architecture. However, as `i` grows, the global work size for this kernel decreases until not enough blocks to fill the GPU are available. Then the limitation for the block size in the runtime decreases the occupancy of each individual CU to favor a better occupancy at the scale of the whole GPU.

Figure 7.2 shows over a range of benchmarks and on various architectures how a carefully chosen block size leads to an acceleration up to 2.4 when compared to a naive constant allocation.

5.8.2 Tuning the Block Dimensions

The CUDA best practice guide [Nvidia 2012a] states that the multidimensional aspect of the block size allows easier mapping of multidimensional problems to CUDA but does not impact performance. However, while it does not impact the hardware scheduler, it changes the memory access pattern and may lead to a very different performance.

The performance obtained for a large range of possible configurations for multiple block sizes is shown in Figures 7.3, 7.4, 7.5, and 7.6 page 183 for the code illustrated in Figure 5.14, and in Figures 7.7, 7.8, 7.9, and 7.10 page 187 for the `matmul` example presented in Figure 4.22. For a given block size, choosing rectangular mapping for the threads leads to a speedup of two to fifteen and more. These two examples show a loop nest with two-dimensional array accesses and an inner sequential loop in the kernel.

The measures conducted in Section 7.4, page 180, lead to another runtime heuristic. For a given block size, the shape of the block is as much as possible a square, but with a number of threads as close as possible to a multiple of thirty-two (or sixteen on older GPUs) threads on the first dimension, i.e., a warp size (or half warp size on older GPUs). This heuristic is implemented in the Par4All runtime.

```
for(i=0;i<n;i++) //Parallel
 for(j=0;j<n;j++) //Parallel
  for(k=0;k<m;k++) //Sequential
   C[i][j]+=alpha*
            A[i][k]*A[j][k];
```

```
int i, j, k;
i = P4A_vp_1;
j = P4A_vp_0;
if (i<N && j<N) {
 double C_0 = 0;
 for(k=0;k<m;k++) {
  C_0+=alpha*
    A[i][k]*A[j][k];
 }
 C[i][j] = C_0;
}
```

Figure 5.14: The main loop nest in `syrk` benchmark from the Polybench suite and the resulting kernel.

5.9 Conclusion

This chapter addresses the issue of automating the whole compilation process, from the original source code to the final binary, but also the runtime.

Due to the heterogeneity of the current platforms, compiler phases need to be combined in unusual and dynamic ways. Moreover, several tools may need to be combined to handle specific parts of the compilation process.

The analyses and transformations presented Chapters 3 and 4 are chained to form a coherent automatic process. I implemented a fully automated solution, within the Par4All project [Amini *et al.* 2012b (perso)], that exploits this process and includes a dedicated runtime for CUDA and OpenCL.

This chapter provides several concepts that I tested and validated within Par4All: the advantage of the source-to-source aspect, the role of a programmable pass manager, handling libraries, making different tools to collaborate, profitability-based scheduling, and choosing a launch configuration at runtime.

Chapter 6

Management of Multi-GPUs

Contents

6.1 Introduction . . . 166
6.2 Task Parallelism . . . 166
6.2.1 The StarPU Runtime . . . 166
6.2.2 Task Extraction in PIPS . . . 167
6.2.3 Code Generation for StarPU . . . 168
6.3 Data Parallelism Using Loop Nest Tiling . . . 170
6.3.1 Performance . . . 171
6.4 Related Work . . . 173
6.5 Conclusion . . . 175

Most of the work presented in this dissertation deals with a host and only one accelerator. This chapter presents some preliminary experiments carried out with multiple accelerators attached to a unique host.

6.1 Introduction

While the CPU manufacturers have faced the frequency wall for a few years now, a lot of research has been conducted on the benefit of adding an accelerator to a classical processor (see Chapter 2). The next step is seen naturally as adding more accelerators per host. While it is cheap from a hardware point of view, it represents a tough challenge for software programmers who are still having trouble programming one accelerator.

This is even more challenging when the ambition is to provide an automatic solution, as in this dissertation. Some related work is described in Section 6.4.

Two major approaches are considered in this chapter. The first one relies on task parallelism where each task is a kernel that can be executed either on an accelerator, or on a CPU, or on both. It is presented in Section 6.2. The second approach presented in Section 6.3 consists in splitting the iteration set and the data among different accelerators.

Some experiments are presented in Section 7.10, even though this chapter presents only some early work on this subject, promising more developments in the next decade.

6.2 Task Parallelism

Improving the state-of-the-art automatic task extraction is beyond the scope of this dissertation. Assuming that a satisfying scheme in the compiler can extract task parallelism automatically or that the programmer expresses it, how can it be exploited to target multiple accelerators?

This section studies how a runtime system like StarPU [Augonnet *et al.* 2011] can be suitable as a target in the back end of a compiler targeting multiple accelerators.

6.2.1 The StarPU Runtime

StarPU [Augonnet 2011, Augonnet *et al.* 2010b] is a runtime system that offers task abstraction to programmers. The computation performed by a program has to be split in separated functions, the *codelets*, with some restrictions. For instance, no global or static

variables can be used in task functions. The tasks take as parameters scalars or buffers. These latter have to be registered with StarPU first, and once registered cannot be used outside of a task without acquiring the buffer from StarPU. This model is very similar to the OpenCL runtime specification (see Section 2.3, page 30).

For a task to run on several architectures, the programmer has to give one implementation per target architecture, or even more than one. A custom helper function can select the best implementation using runtime information provided by StarPU, such as the exact GPU model selected for execution.

StarPU then takes care of the scheduling and executes those codelets as efficiently as possible over all the resources available at runtime. Memory coherency is enforced over registered buffers. Moreover, the scheduler can decide where to run a computation based on the actual location of the input data.

StarPU provides interesting performance when distributing work among multiple GPUs and CPUs concurrently [Augonnet *et al.* 2010a].

6.2.2 Task Extraction in PIPS

A task extraction phase was added in PIPS to support the experimentations conducted in this chapter. This implementation was naive at that time, far from the interprocedural and hierarchical schemes mentioned in Section 6.4. Moreover, it is based on a simple heuristic based on the presence of loops instead of a connection with the complexity analysis [Zhou 1992] provided by PIPS to make profitability-based decisions. However, in the context of this chapter, it is sufficient for simple benchmarks like the Polybench or on some Scilab script automatically compiled to C.

The principle is to group all computations in identified *tasks*, i.e., functions that will be called by the StarPU runtime system. This process extensively uses the *outlining* phase (illustrated in Figure 4.3b page 100), which extracts a sequence of statements from an existing function and replaces them with a call to a new function built with that sequence.

The heuristic detects loop nests in a sequence and checks that they cannot be grouped in a common task. Thus any optimization that would be impossible by this separation in functions like the loop fusion transformation (see Section 4.6 page 112) is expected to be applied first.

This transformation also ignores loop nests that have already been converted into kernel calls and that can be considered as tasks.

Figure 6.1 illustrates this process on the simple `3mm` example from the Polybench suite.

```
/* Initialize array. */
init_array();

/* E := A*B */
for(i=0;i<=ni-1;i++)
 for(j=0;j<=nj-1;j++) {
  E[i][j]=0;
  for(k=0;k <= nk-1;k++)
   E[i][j]+=A[i][k]*B[k][j];
 }
/* F := C*D */
for(i=0;i<=nj-1;i++)
 for(j=0;j<=nl-1;j++) {
  F[i][j]=0;
  for(k=0;k <= nm-1;k++)
   F[i][j]+=C[i][k]*D[k][j];
 }
/* G := E*F */
for(i=0;i<=ni-1;i++)
 for(j=0;j<=nl-1;j++) {
  G[i][j]=0;
  for(k=0;k <= nj-1;k++)
   G[i][j]+=E[i][k]*F[k][j];
 }

print_array(argc, argv);
```

(a) Input code.

```
init_array();

/* E := A*B */
P4A_task_main(A,B,E,
              i,j,k,
              ni,nj,nk);
/* F := C*D */
P4A_task_main_1(C,D,F,
                i,j,k,
                nj,nl,nm);
/* G := E*F */
P4A_task_main_2(E,F,G,
                i,j,k,
                ni,nj,nl);

print_array(argc,argv);
```

(b) After task transformation.

Figure 6.1: Task transformation process. `3mm` example from the Polybench suite automatically transformed with tasks.

6.2.3 Code Generation for StarPU

After the task generation has been performed, the code has to be decorated with the pragmas that will trigger the StarPU runtime to catch calls to task functions.

All arrays in the code are registered with StarPU as buffers. StarPU then takes care of the dependences between tasks, based on the arguments, at runtime. It also automatically transfers the data if the task is expected to be run on a GPU. The *host* code calls StarPU tasks asynchronously. Some global synchronizations can be inserted, or alternatively synchronizations based on the availability of a buffer, i.e., waiting for a buffer to be filled.

```
void task_main(const double A[512][512], const double B[512][512],
               __attribute__((output)) double E[512][512],
               int i, int j, int k, int ni, int nj, int nk)
  __attribute__((task));
void task_main_cpu(const double A[512][512],
                   const double B[512][512],
                   __attribute__((output)) double E[512][512],
                   int i, int j, int k, int ni, int nj, int nk)
  __attribute__((task_implementation("cpu",task_main)))
{// CPU implementation
 // ...
}
int main() {
 #pragma starpu initialize
 #pragma starpu register A
 #pragma starpu register B
 #pragma starpu register C
 #pragma starpu register D
 #pragma starpu register E
 #pragma starpu register F
 #pragma starpu register G

 init_array();

 /* E := A*B */
 P4A_task_main(A,B,E,i,j,k,ni,nj,nk);
 /* F := C*D */
 P4A_task_main_1(C,D,F,i,j,k,nj,nl,nm);
 /* G := E*F */
 P4A_task_main_2(E,F,G,i,j,k,ni,nj,nl);

 print_array(argc,argv);

 #pragma starpu unregister A
 #pragma starpu unregister B
 #pragma starpu unregister C
 #pragma starpu unregister D
 #pragma starpu unregister E
 #pragma starpu unregister F
 #pragma starpu unregister G
 #pragma starpu shutdown
}
```

Figure 6.2: Code generated automatically with pragmas to drive the StarPU runtime. The tasks have been declared with attributes in order to drive StarPU, specifying the suitable target platform for each implementation. Task parameters are declared with a `const` qualifier when used as read-only, and with an `__attribute__((output))` when used as write-only.

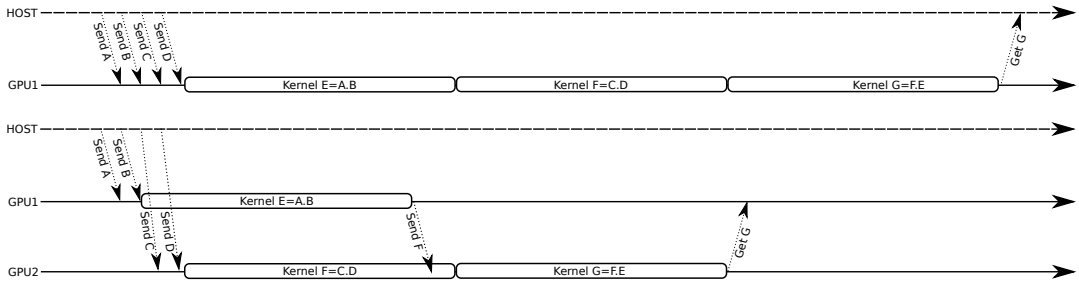


Figure 6.3: Execution timeline when using parallel tasks on one GPU (upper part) and on two GPUs (lower part) for the 3mm code presented in Figure 6.2.

Hints can be given to StarPU about buffers. A buffer used only for reading by a task can be flagged as constant and StarPU will not generate a copy-out for it or invalidate other copies. Moreover, another task using the same buffer can be scheduled in parallel. In the same way, buffers can be flagged as write-only so that no copy-in is generated. The array region analysis introduced in Section 3.2 page 64 provides this information at compile time.

Figure 6.2 illustrates the result of this transformation on the 3mm example. Tasks have been generated for each matrix multiplication, pragmas have been generated so that StarPU can track the buffers, and task parameters have been declared with a `const` qualifier or an `__attribute__((output))`.

6.3 Data Parallelism Using Loop Nest Tiling

Section 3.9 page 90 explored the minimization of communications when mapping a sequential tiled loop nest on an accelerator. This section introduces the mapping of a loop nest composed of fully parallel tiles onto multiple GPUs.

Loop tiling is a well known transformation [Wolfe 1989, Irigoin & Triolet 1988, Dongarra & Schreiber 1990, Wolf & Lam 1991b, Xue 2000, Andonov & Rajopadhye 1994], but it is still an active field [Hartono *et al.* 2009, Kim & Rajopadhye 2010, Yuki *et al.* 2010, Bao & Xiang 2012, Zhou *et al.* 2012, Lin *et al.* 2011, Renganarayanan *et al.* 2012]. Guelton introduced in PIPS a Symbolic Tiling scheme to adjust the memory footprint [Guelton 2011a] of kernels. The tile size can be instantiated at runtime according to the accelerator used. His goal was to generate code for an embedded accelerator with only a few kilobytes of memory. I have implemented another slightly different symbolic tiling scheme in PIPS in order to instantiate at runtime the number of tiles instead of the tile size. To feed multiple

GPUs, the loop nest is split into tiles that are distributed over the accelerators.

Figure 6.4b illustrates this transformation, and shows the array regions computed by PIPS on this example. The final code with the DMA instructions is shown in Figure 6.4c. The calls to functions prefixed with `P4A_` are asynchronous calls.

At execution time, a number of OpenCL queues are created and associated with some devices and the loop nest is dynamically split into as many tiles as queues, thanks to the symbolic tiling scheme. A dedicated runtime system was developed to support this scheme. It registers the data to transfer for each tile, and asynchronously schedules the communications and the kernel launches. It automatically provides the correct OpenCL events to express the dependences between commands in the different queues.

As a side effect, and because the OpenCL-based runtime is able to associate multiple command queues with each device, this scheme can be used to overlap communication and computation on one device. When the computation and communication times are perfectly balanced, this could lead to a speedup up of two. However, in practice this is seldom the case. Moreover, the bandwidth between the GPU memory and the PEs suffers from contention due to the concurrent transfers between the host and the GPU.

6.3.1 Performance

Section 7.10, page 205, provides experimental results for the two schemes presented in the previous sections. The task parallelism scheme shows that it can be relevant for some kinds of workloads. For instance the `3mm` example from Polybench shown in Figure 6.1a is sped up by thirty percent when using two Nvidia C1060 instead of one. However, this approach is limited by the number of tasks (or kernels) that can be run in parallel. The `3mm` example has only two kernels that can be run in parallel, followed by a synchronization point before the last kernel. Adding more than two GPUs is useless in this case.

Another limitation is the StarPU runtime scheduling cost. The default greedy scheduler provides a rather low overhead, However, since it does not take data locality into account, it ends up with a mapping resulting in too many data transfers to expect any improvements. The problem can be observed when the `3mm` example is executed with three GPUs. The last kernel is then mapped on the third GPU resulting in two additive data transfers, one from each of the other two GPUs. It is even clearer on the `bicg` Polybench test case. A large number of dependent kernels are scheduled on different GPUs resulting in a flip-flop situation with respect to the memory location of the data. The performance impact on this last example is dramatic with a slowdown of 8000 when compared to the same code executed

```
for(i = 0; i <= N-1; i += 1)
  A[i] = B[i]*c;
```

(a) Input code.

```
for(I = 0; I <= p4a_nkernels; I++) {
  int I_up = MIN(N-1, (I+1)*(1+(N-1)/p4a_nkernels)-1);
  int I_low = I*(1+(N-1)/p4a_nkernels);
  //  W(A) = {A[φ1] | I_low ≤ φ1 ≤ I_up}
  //  R(B) = {B[φ1] | I_low ≤ φ1 ≤ I_up}
  for(i = I_low; i <= I_up; i++)
    A[i] = B[i]*c;
}
```

(b) After tiling with the convex array regions computed by PIPS.

```
void p4a_launcher_scal_1(int N, double *A, double *B,
                         int I_up, int I_low, double c) {
 int i; double *A0 = (double *) 0, *B0 = (double *) 0;
 if (I_up-I_low>0&&I_up-I_low>0) {
  P4A_accel_malloc((void **) &B0, sizeof(double)*(I_up-I_low+1));
  P4A_accel_malloc((void **) &A0, sizeof(double)*(I_up-I_low+1));
  // Copy-in B tile
  P4A_copy_to_accel_1d(sizeof(double), N,
                       I_up-I_low+1, I_low,
                       B+0, B0);
  // Call kernel
  P4A_call_accel_kernel_1d(p4a_wrapper_scal_1,
                           I_up-I_low+1, I_up,
                           I_low, A0, B0, i, c);
  // Copy-out A tile
  P4A_copy_from_accel_1d(sizeof(double), N,
                         I_up-I_low+1,
                         I_low, A+0, A0);
  P4A_accel_free(B0);
  P4A_accel_free(A0);
 }
}

// Loop over tiles
for(I=0;I<=p4a_nkernels;I++) {
 int I_up = MIN(N-1, (I+1)*(1+(N-1)/p4a_nkernels)-1),
 int I_low = I*(1+(N-1)/p4a_nkernels);
 p4a_launcher_scal_1(N, A, B, I_up, I_low, c);
}
```

(c) After kernel and transfer code generation.

Figure 6.4: Tiling transformation process illustrated on a simple vector scaling multiplication example.

with only one GPU worker. Fortunately, StarPU provides other schedulers including some that take into account the task affinity and manage to provide better mapping. The measurement Section 7.10 page 205 confirms that using a data-aware scheduler avoids this issue.

The results show that this approach may provide good performance, even if the Nvidia OpenCL implementation limits currently to 1.63 the maximum speedup I managed to obtain, either using two GPUs or overlapping communications and computations on one GPU.

Some future work would replace our custom runtime by a more flexible library like StarPU, with a more dynamic adaptation to the workload, and capabilities to distribute tasks on both GPUs and multicore CPUs.

6.4 Related Work

A lot of previous work has been done to extract tasks from sequential code, starting with the work of Sarkar and Henessy [Sarkar & Hennessy 1986] about compile-time partitioning and scheduling techniques and Cytron et al. [Cytron *et al.* 1989] on the parallelism extraction from a DAG. From a sequential program with the data and control dependences, their implementation in the PTRAN system [Allen *et al.* 1988] produces an equivalent fork-join parallel program. A hierarchical task graph is used to extract coarse grained task level parallelism by Girkar et al. [Girkar & Polychronopoulos 1994, Girkar & Polychronopoulos 1995]. Hall et al. use interprocedural analyses in the SUIF compiler to extract coarse grained parallelism [Hall *et al.* 1995]. Sarkar [Sarkar 1991] and Ottoni [Ottoni *et al.* 2005] use some derived program dependence graphs to parallelize application. Verdoolaege et al. [Verdoolaege *et al.* 2007] use process networks to extract parallelism from a sequential application, however, limited to affine loops. Ceng et al. proposed a semi-automatic scheme [Ceng *et al.* 2008]. They derive a weighted control data flow graph from the sequential code and iterate, using a heuristic and user inputs, to cluster and generate tasks. Cordes et al. [Cordes *et al.* 2010] makes use of Integer Linear Programming to steer the task granularity for a given hardware platform, whose characteristics have to be known at compile time.

Stuart et al. [Stuart *et al.* 2010] developed a library to target multiple GPUs using the MapReduce programming model [Dean & Ghemawat 2004]. Other work has been conducted for ordering tasks at runtime [Augonnet *et al.* 2010a] or with preemption and

priority handling for critical application [Membarth *et al.* 2012].

When sharing work between CPUs and GPUs, Hermann [Hermann *et al.* 2010] found that placing small tasks on the CPUs makes mapping GPUs more efficient and leads to a speedup greater than the sum of the speedups separately obtained on the GPUs and the CPUs. Following this idea, complexity analysis like the one existing in PIPS [Zhou 1992] might be used to generate hints about the relative *size* of the different tasks and to lead to better decision by the runtime system.

Huynh et al. introduced a framework [Huynh *et al.* 2012] as a back end of the StreamIt [Gordon *et al.* 2002] compiler. StreamIt is a language using the streaming programming model. The technique employed to identify suitable partitions relies on a modified well-known k-way graph partitioning algorithm [Karypis & Kumar 1998]. It computes load-balanced partitions with minimum communications.

Chen et al. [Chen *et al.* 2011] manually explore both the task and data parallelism for unstructured grid applications, using multiple GPUs and overlapping computation and communication.

Enmyren et al. [Enmyren & Kessler 2010, Dastgeer *et al.* 2011] propose SkePU, a C++ template library that provides a unified interface for specifying data parallel computations with the help of skeletons. They make use of lazy memory copying to avoid useless transfer between the host and the accelerator.

The automatic extraction of data parallelism for distributed memory systems has been studied too. For instance, Ancourt et al. proposed a linear algebra framework for static HPF code distribution [Ancourt *et al.* 1993, Ancourt *et al.* 1997]. STEP [Millot *et al.* 2008] is a framework to convert OpenMP code to MPI using PIPS. Such works share common problems with the multiple-GPU target, like the distributed memory space and data coherency issues.

Not much work has been done extracting data parallelism automatically for multiple GPUs. Many experiments were carried out manually by scientists in different fields (see for example [Aubert 2010]). Leung et al. [Leung *et al.* 2010] gave some ideas about using the polyhedral model to map a loop nest on a GPU, but despite the attractive title the specific issues of multi-GPUs are not addressed, e.g. the memory transfers between the host and the accelerators.

6.5 Conclusion

The free lunch is over: parallel programming is now mandatory. Heterogeneity is not the future but the present. The last years have seen a lot of research about software development for accelerators. The compilers and tools field has been mostly limited to one GPU and one host, but the next step seems clearly to be inching toward more heterogeneity using multiple accelerators.

This chapter presents two different ways of extracting parallelism in order to map kernels onto multiple accelerators. The first one relies on task parallelism where each task is a kernel that can execute on an accelerator, on a CPU, or on both. I implemented a very simple automatic task parallelism extraction in PIPS, even if the approach is independent of the extraction. I also implemented a code generation phase for the StarPU runtime system, and validated the full chain with experiences. The second approach consists in splitting the iteration set and the data among different accelerators. I modified the existing parallel symbolic tiling transformation in PIPS to provide the expected resulting code. I implemented and validated with experiences a dedicated runtime to map the tiles' execution over multiple GPUs.

Some limitations in the OpenCL implementation provided by Nvidia still limit the performance that can be obtained, but future releases and the forthcoming Kepler-based Tesla with improved GPU–GPU communications should improve the situation.

The experimental results obtained with these two basic schemes are encouraging, and many more deeper researches are expected in this challenging field for the next decades.

The next chapter presents the experiments conducted to validate all the proposal developed in the previous chapters.

Chapter 7

Experiments

Contents

7.1 Hardware Platforms Used . . . 178
7.2 Benchmarks, Kernels, and Applications Used for Experiments . 179
7.3 Parallelization Algorithm . . . 180
7.4 Launch Configuration . . . 180
7.5 Scalarization . . . 181
7.5.1 Scalarization inside Kernel . . . 191
7.5.2 Full Array Contraction . . . 191
7.5.3 Perfect Nesting of Loops . . . 191
7.6 Loop Unrolling . . . 195
7.7 Array Linearization . . . 195
7.8 Communication Optimization . . . 197
7.8.1 Metric . . . 197
7.8.2 Results . . . 199
7.8.3 Comparison with Respect to a Fully Dynamic Approach . . . 199
7.8.4 Sequential Promotion . . . 201
7.9 Overall Results . . . 202
7.10 Multiple GPUs . . . 205
7.10.1 Task Parallelism . . . 205
7.10.2 Data Parallelism . . . 206
7.11 Conclusions . . . 208

This chapter presents the experiments that validate the different program transformations introduced in the previous chapters. Various benchmarks (see Section 7.2) and several hardware platforms (see Section 7.1) have been used.

7.1 Hardware Platforms Used

To explore the validity of transformations or other schemes among different architecture generations, many hardware platforms have been used involved in my tests, using Nvidia and AMD GPUs.

The different platforms used are listed in Table 7.1.

- The 8800GTX is the first CUDA-enabled GPU. It was introduced in 2007. It provides low performance when compared to more recent hardware, However, its architecture with very naive memory controllers exhibits interesting patterns. The board is plugged into a host with a 2-core Xeon 5130 running at 2 GHz.
- The C1060 is the first Tesla, i.e., the first GPU from Nvidia to be especially floating-point computation oriented. Two Intel X5570 CPUs (4-core each) are running the host at 2.93 GHz.
- The C2070 can have ECC enabled or not, impacting the performance and the total available memory. It is still today the most high-end compute-oriented solution from Nvidia while awaiting the Kepler-based Tesla at the end of 2012. The board is plugged into a host with two six-core Intel Xeon X5670 processors running at 2.93 GHz.
- The AMD 6970 is now outdated by the more recent 7970 based on the GCN architecture (see Section 2.4.4, page 43). The host runs a 6-core AMD FX-6100 at 3.3 GHz. It is the same generation as the Fermi-based Nvidia boards.
- Finally, the GTX 680 is based on the new Nvidia Kepler architecture, but is not compute oriented and lacks many of the Kepler novelties, like the Dynamic Parallelism feature. Also, it does not include the new GDDR5 memory interface and provides a theoretical bandwidth lower than the 320 GB/s announced for the yet unreleased Kepler-based Tesla. This chip should provide 2888 cores with 1/3 float to double precision ratio, while the GTX 680 that we use ships *only* 1536 cores and only 1/24 float to double precision ratio. The CPU is an 4-core Intel Core i7 3820 running at 3.6 GHz.

GPU	8800GTX	3xC1060	2xC2070	6970	GTX680
Chipset	G80	GT200	Fermi	Cayman	Kepler
GPU Memory (GB)	0.768	$3x4$	$2x6$	2	2
GPU Cores	128	$3x240$	$2x448$	1536	1536
GPU Freq (MHz)	1350	1300	1150	880	1006–1058
GPU Theoretical Bw (GB/s)	86.4	102	144	176	192.2
Host	X5130	$2 \times$ X5570	$2 \times$ E5620	FX-6100	i7-3820
CPU Physical Cores	2	2×4	2×4	6	4
Host Memory (GB)	4	24	24	8	8

Table 7.1: The different hardware platforms available for the experiments.

7.2 Benchmarks, Kernels, and Applications Used for Experiments

Benchmarks involved in the experiments conducted in this chapter are taken from the Polybench 2.0 suite and from the Rodinia suite. Some kernels were extracted to exhibit interesting behaviors and validate some transformations.

A more representative *real-world* example introduced in Section 3.1 page 63 is also used to evaluate the whole tool chain with optimized data mapping (see Chapter 3). This numerical simulation, called Stars-PM, is a particle mesh cosmological N-body code.

Some Scilab examples, first compiled into an intermediate C version, are also used.

All these codes were used as is, except those from Rodinia. Those were rewritten partially to conform to some coding rules [Amini *et al.* 2012c (perso)]. The Rodina rules specify mostly that array accesses should not be linearized and C99 should be used as much as possible to type precisely the arrays. Therefore it is only a syntactic rewrite: C99 arrays are used and references adjusted without algorithmic modifications. This rewriting is due to the parallelization schemes implemented in PIPS and not to my communication optimization scheme (see Section 3.6, page 77).

For this reason, and because some tests are intrinsically not adapted to GPU computing, only the suitable part of the benchmark suites are presented here.

Some Polybench test cases use default sizes that are too small to amortize even the initial data transfer to the accelerator. Following Jablin et al. [Jablin *et al.* 2011], the input sizes were adapted to match the capabilities of the GPUs.

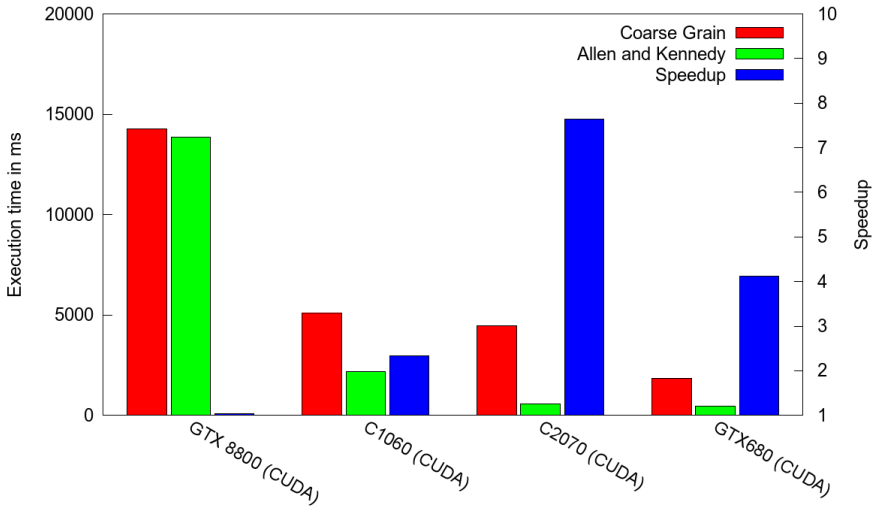


Figure 7.1: Kernel execution times in ms (best over twenty runs) and speedups between the Coarse Grained and the Allen and Kennedy parallelization algorithms using CUDA for different Nvidia GPUs. The example used here is the correlation loop nest shown in Figure 4.6 with $m = 3000$.

7.3 Parallelization Algorithm

As explained in Section 4.3.3 page 104, the choice of the parallelization algorithm has an impact on the parallelism expressed and on the resulting number of kernels. Because the Allen and Kennedy algorithm introduced in Section 4.3.1 distributes the loops and leads to more kernels, it expresses finer parallelism than the coarse grained algorithm presented in Section 4.3.2 page 103.

Figure 7.1 shows how the Allen and Kennedy scheme leads to a more efficient code on all architectures. While the difference is very limited on old architectures such as the G80, it is dramatic on recent GPUs, which benefit from the larger amount of parallelism exposed. The execution time is up to eight times faster on Fermi and four times on Kepler in favor of Allen and Kennedy's parallelization.

7.4 Launch Configuration

OpenCL and CUDA split the iteration set into a grid of work-items (or threads) and group them into work-groups (or blocks), as explained in Section 2.3.2.2 page 32. It is up to the programmer to chose the size and the shape of the work-groups.

Figure 7.2 shows the impact of the work-group size on a large range of benchmarks. All work-items are allocated to the first dimension of the work-group. The performance impact of the work-group size is important, with an acceleration factor up to 2.4 for a given test case. More interestingly there is no general optimal size; that depends on the kernel and also on the architecture. The evolution of the performance as a function of the work-group size varies across tests, but also across different architectures for the very same test. This means that the block sizes have to be adjusted at runtime, kernel per kernel, depending on the global work-grid dimensions, and on the GPU architecture.

For a given work-group size, another parameter is the balance between the two dimension, which we call *dimensionality*. The impact of the different dimensionalities for various work-group sizes is illustrated for several architectures in Figures 7.3, 7.4, 7.5, and 7.6 for the `syrk` example Figure 5.14 page 163, and in Figures 7.7, 7.8, 7.9, and 7.10 for the `matmul` example of Figure 4.22 page 129.

The balance of work-items between the first two dimensions in a work-group largely impacts the performance. A common pattern is visible, with better performance when exploiting the two dimensions as much as possible. While the improvement over mapping all work-items on the first dimension is quite low on `matmul`, on `syrk` it can lead to a speedup of more than fifteen for some work-group sizes. Choosing the best work-group size on the latest GTX 680 Nvidia GPU achieves a four times speedup by adjusting the dimensionality.

7.5 Scalarization

The transformation presented in Section 4.7, page 127, exhibits three interesting patterns that are measured here:

- scalarization of array accesses in a sequential loop inside the kernel (see Section 4.7.1, page 128);
- full array contraction, which eliminates temporary arrays, usually after loop fusion (see Section 4.7.2, page 128);
- scalarization can break the perfect nesting of loops (see Section 4.7.3 page 130).

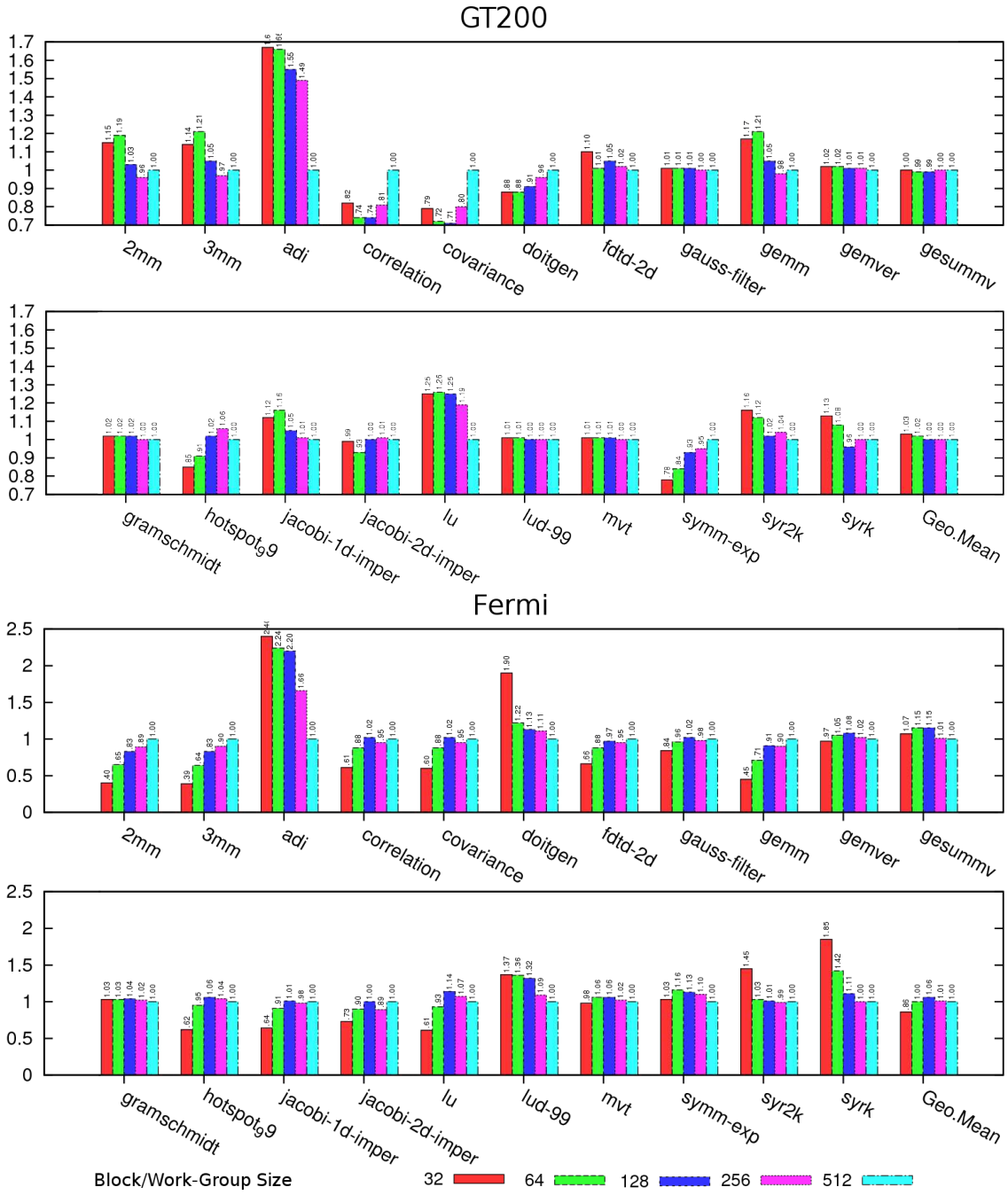


Figure 7.2: Impact of work-group size on GT200 and Fermi architectures, speedup normalized with respect to a size of 512 work-items per work-group. The CUDA API is used in this experiment.

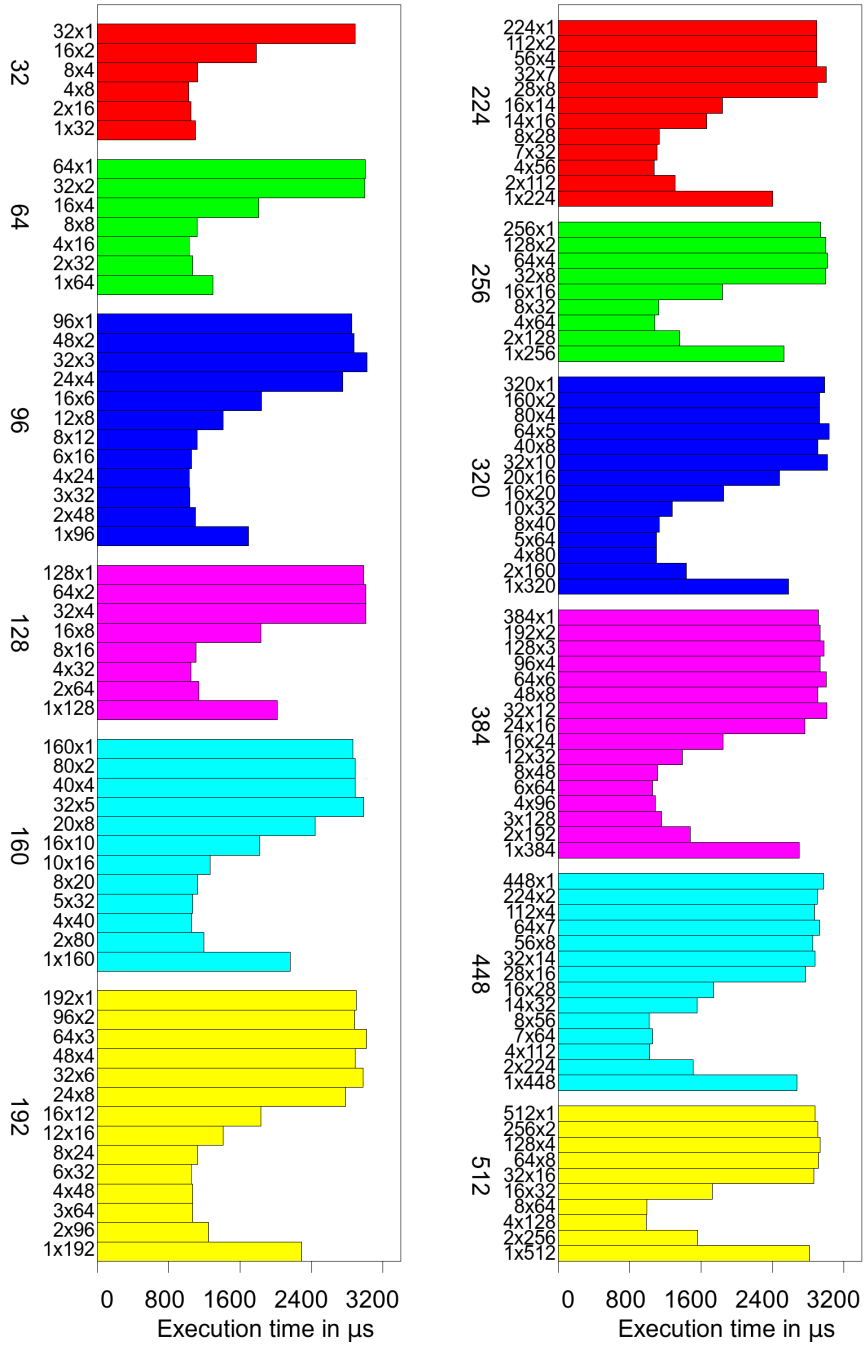


Figure 7.3: Impact of block dimensionality on performance for different block sizess expressed in µs for G80 architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163.

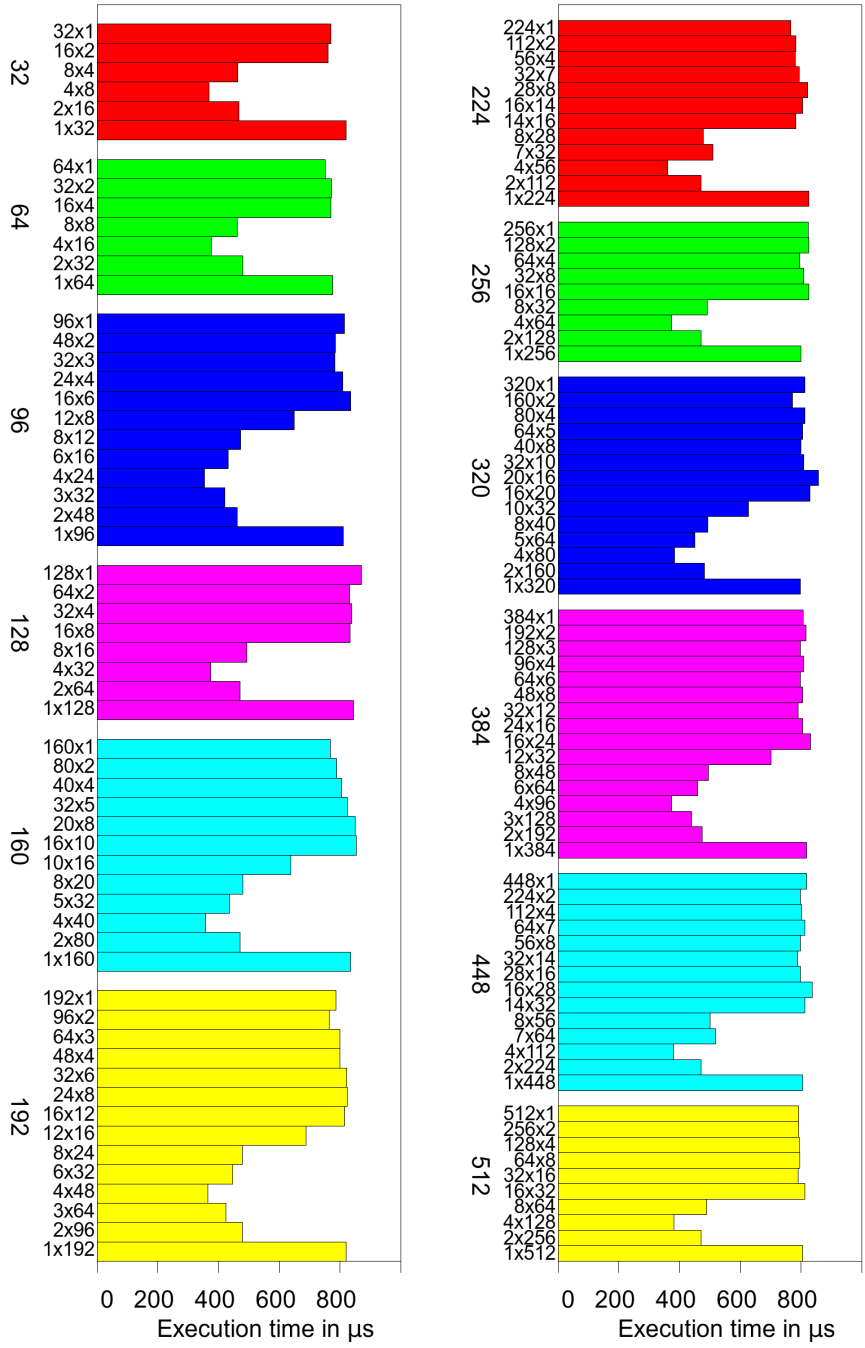


Figure 7.4: Impact of block dimensionality on performance for different block sizes expressed in µs for GT200 architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163.

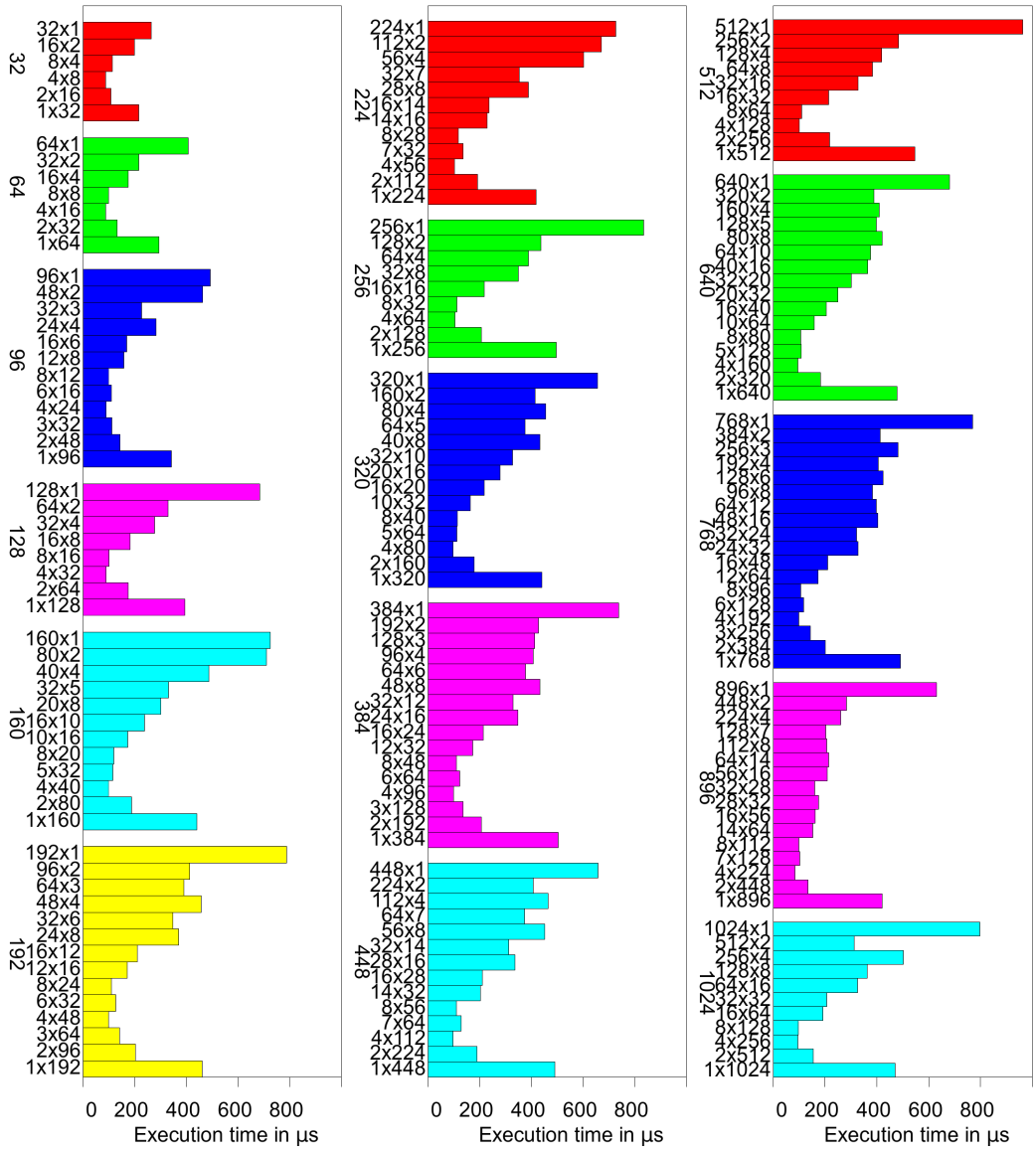


Figure 7.5: Impact of block dimensionality on performance for different block sizes expressed in µs for Fermi architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163.

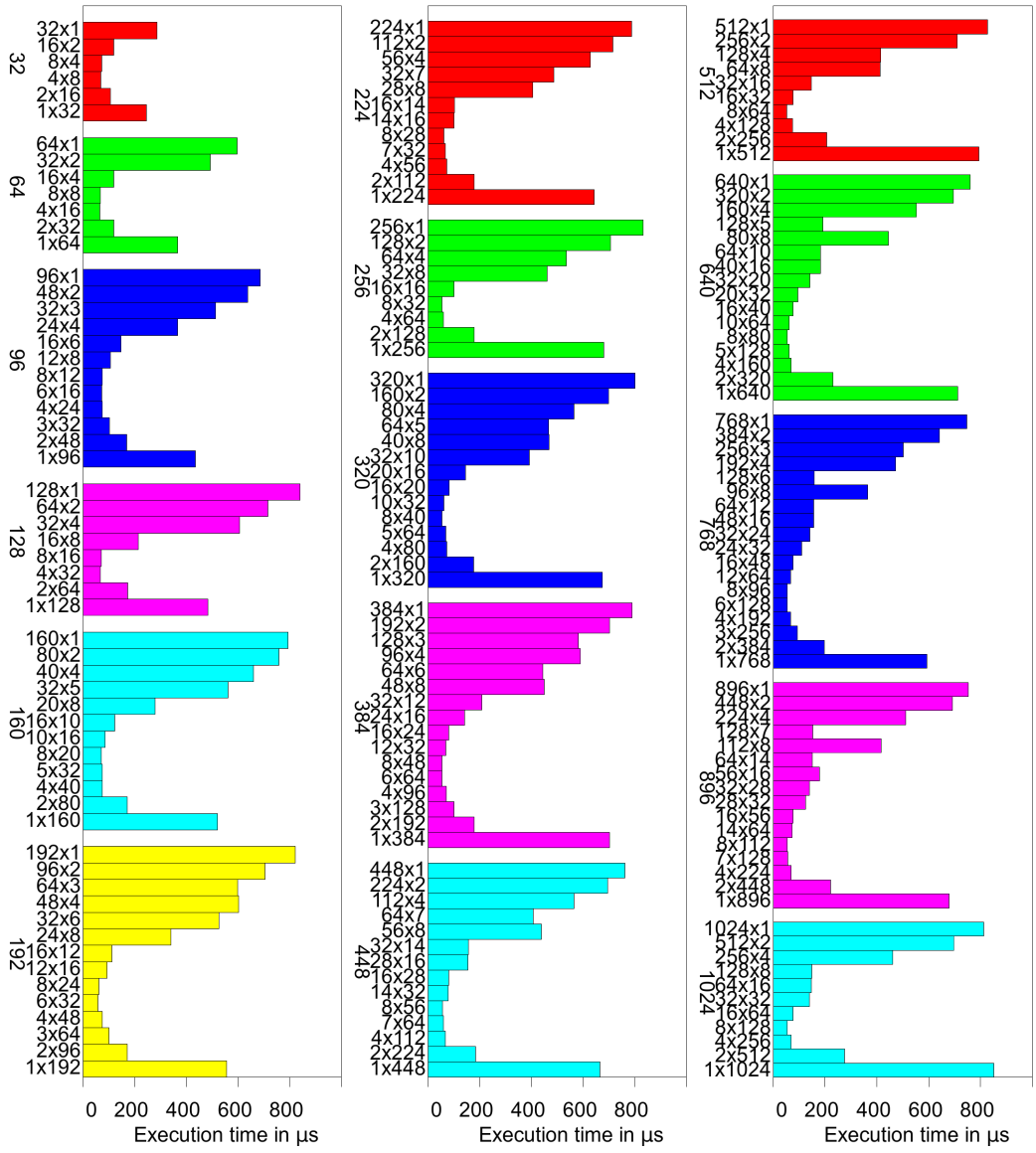


Figure 7.6: Impact of block dimensionality on performance for different block sizes expressed in µs for Kepler architecture. The reference kernel here is the main loop nest from `syrk` (Polybench suite) shown in Figure 5.14 page 163.

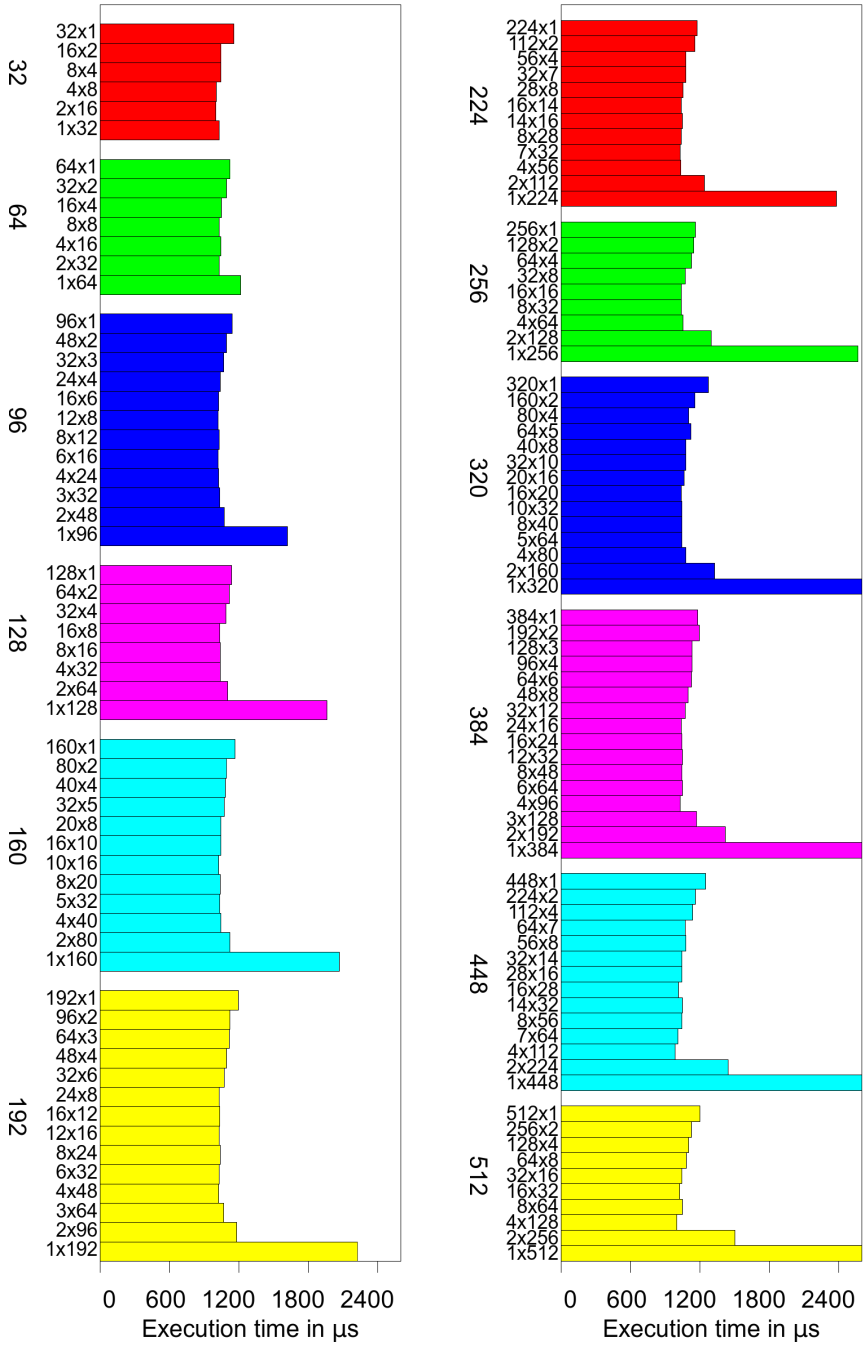


Figure 7.7: Impact of block dimensionality on performance for different block sizes expressed in µs for G80 architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129.

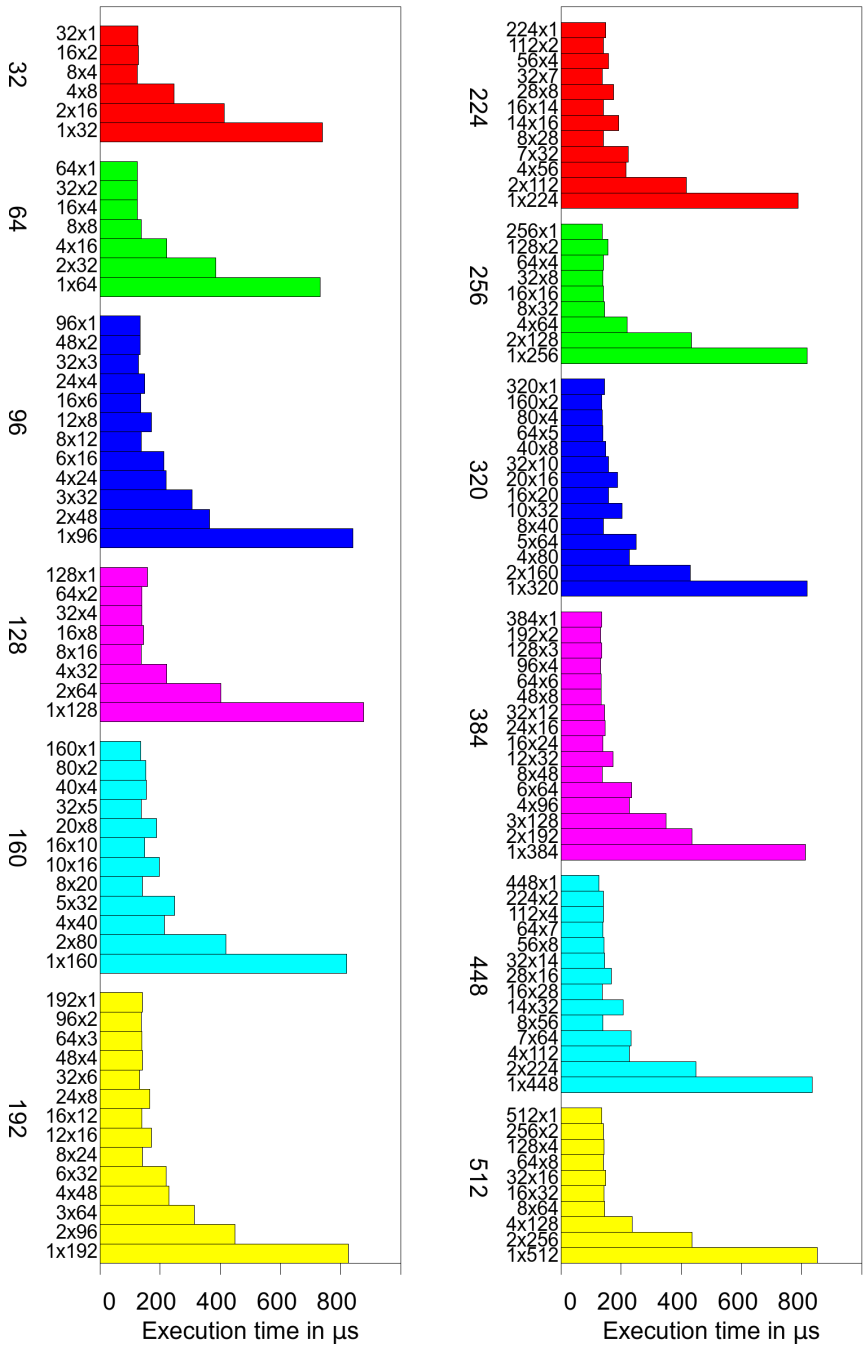


Figure 7.8: Impact of block dimensionality on performance for different block sizes expressed in µs for GT200 architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129.

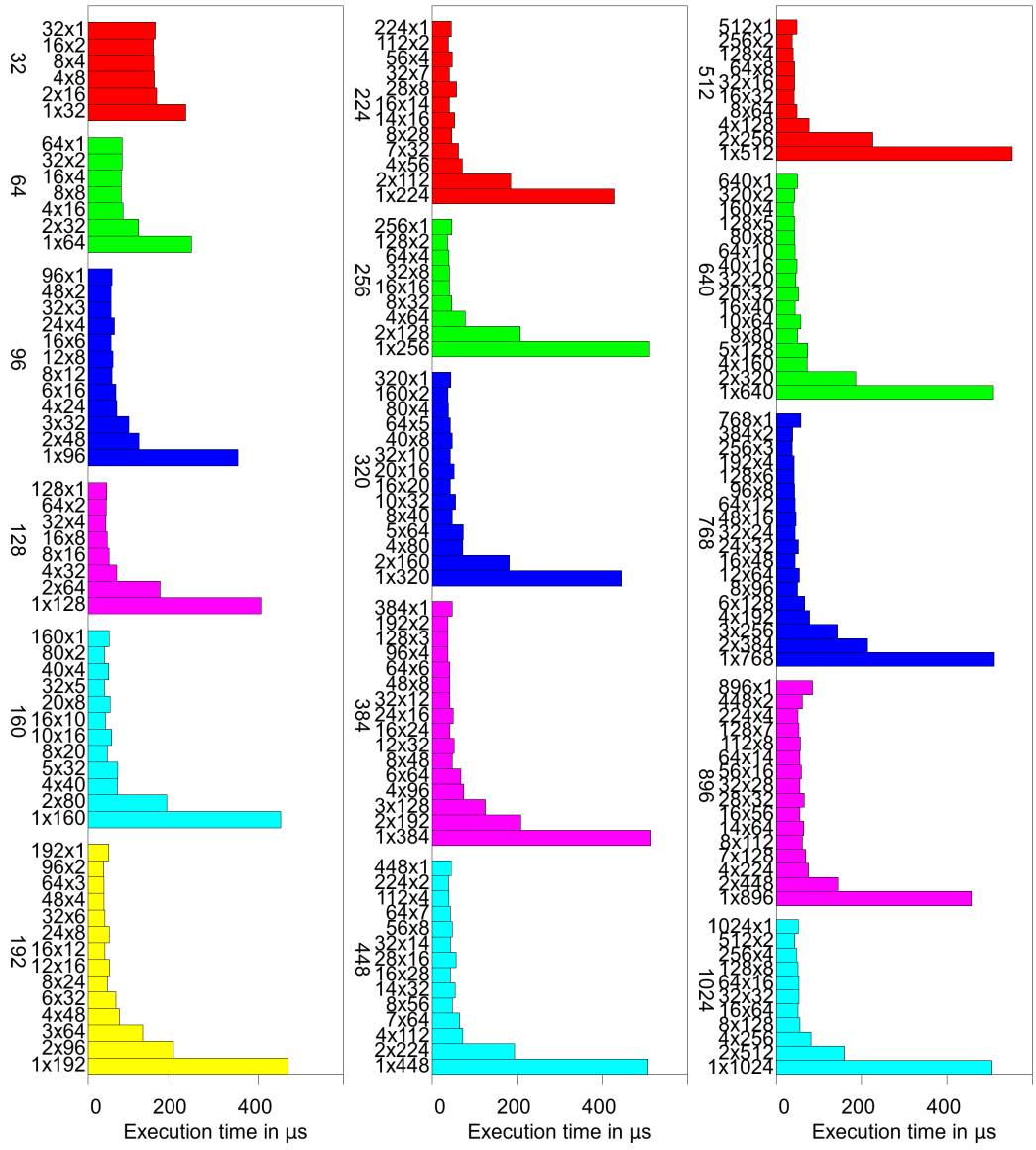


Figure 7.9: Impact of block dimensionality on performance for different block sizes expressed in µs for Fermi architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129.

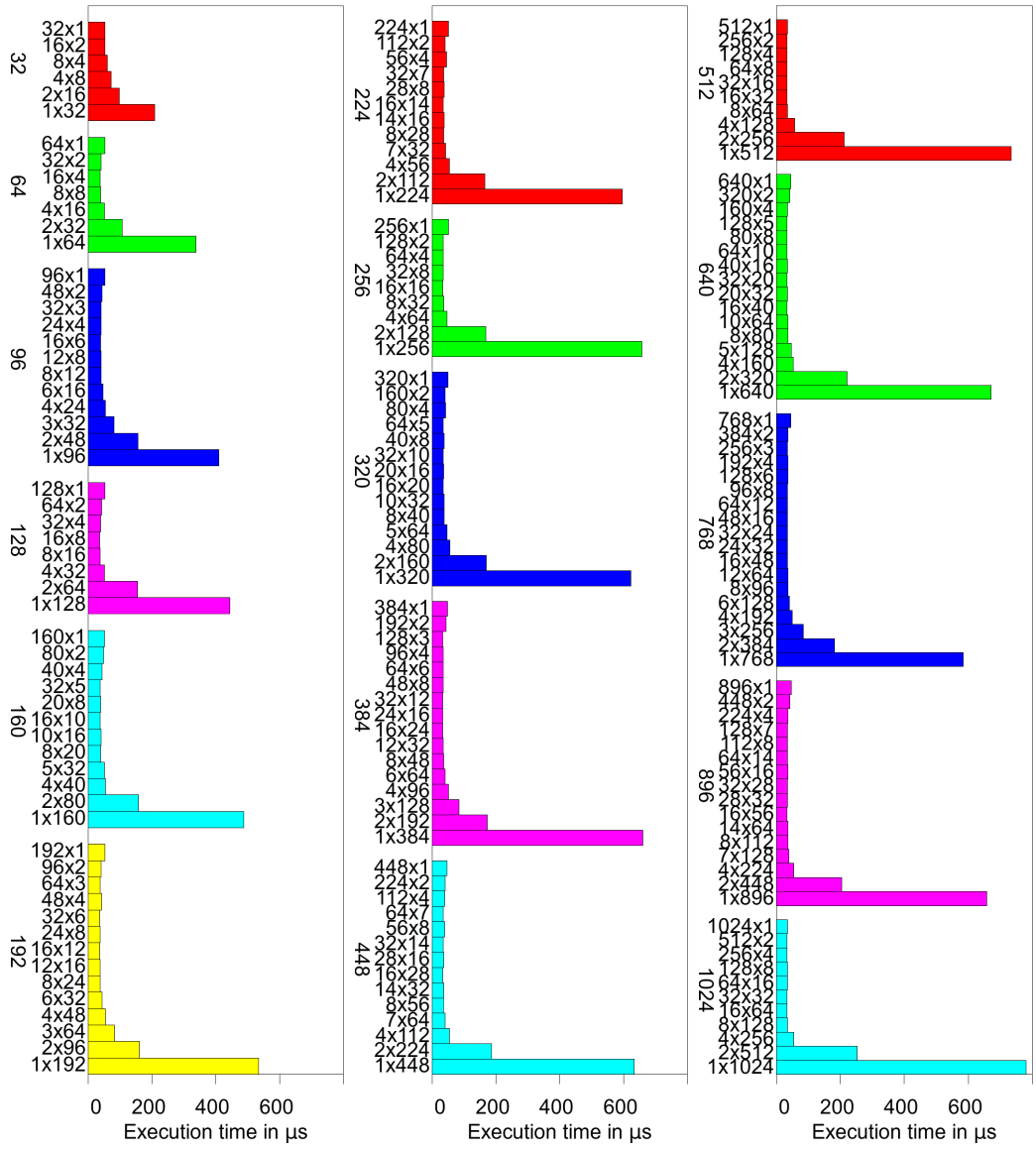


Figure 7.10: Impact of block dimensionality on performance for different block sizes expressed in µs for Kepler architecture. The reference kernel is the `matmul` example shown in Figure 4.22 page 129.

7.5.1 Scalarization inside Kernel

The first transformation to measure is presented in Figure 4.22 page 129 with a simple matrix multiplication example. The interest is to keep in a register the reference to the array and therefore reduce memory accesses. This transformation could be done in the target backend compiler (`nvcc` for Nvidia): it is exposed at source level here. The speedup is between 1.12 and 1.59 in double precision floating point, but more impressive in single precision with a speedup ranging from 1.37 up to 2.39, and no slowdown is observed (see Figure 7.11).

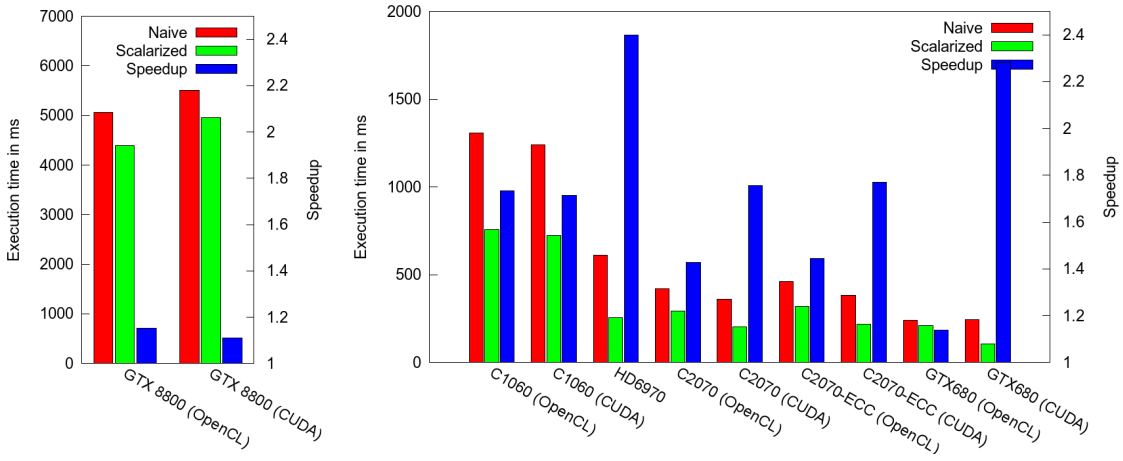
7.5.2 Full Array Contraction

The second transformation allows a total array contraction, usually after loop fusion as shown with the Scilab script of Section 4.7 (see Figure 4.21, page 129). The speedups obtained vary from 1.96 up to 5.75 (see Figure 7.12). Moreover, three arrays no longer need to be allocated on the accelerator, reducing the kernel memory footprint, and the compiler does not generate any transfers, which would each take as long as the kernel execution time. Unlike the previous pattern, this one cannot be performed by the target binary compiler, and therefore it is necessary to detect this opportunity and to perform the transformation at source level.

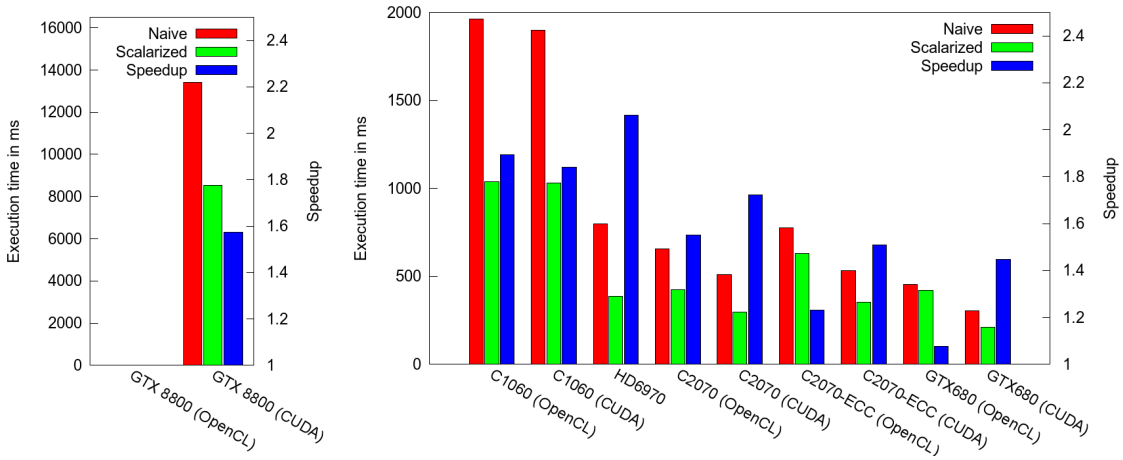
7.5.3 Perfect Nesting of Loops

My last transformation explores the three possibilities exposed by the code in Figure 4.23, page 131, and discussed in Section 4.7.3, page 130. The scalarization breaks the perfect nesting of the loops, and thus there is only one iteration space mapped onto the GPU. Instead of one kernel launch, there are now multiple launches with fewer threads each. Figure 7.13 shows the performance comparison of the original version and the version after scalarization with the inner loop mapping. The value for `M` has been fixed at 8192 and the values for `N` follow the x axis. The third version is not considered because it is by far slower than the others.

As expected, the execution time scales nearly linearly and quickly with `N` for the scalarized version with inner mapping, while the time for the non-scalarized version remains low. Small jumps are observed on the latter because of the SIMD width of thirty-two on this architecture, and because of the various overheads not overlapped with computation because only N threads are available (see Sections 4.7.3). The inner mapping scheme is

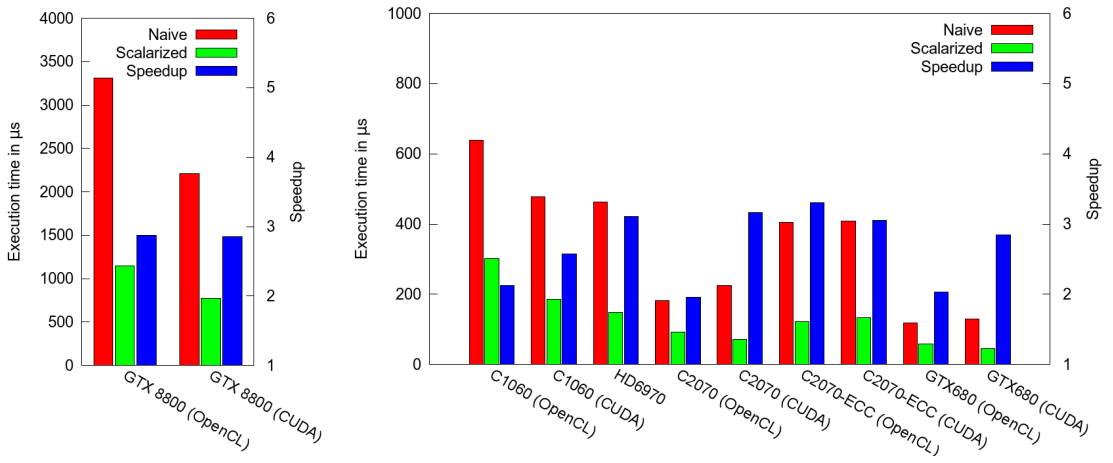


(a) single precision.

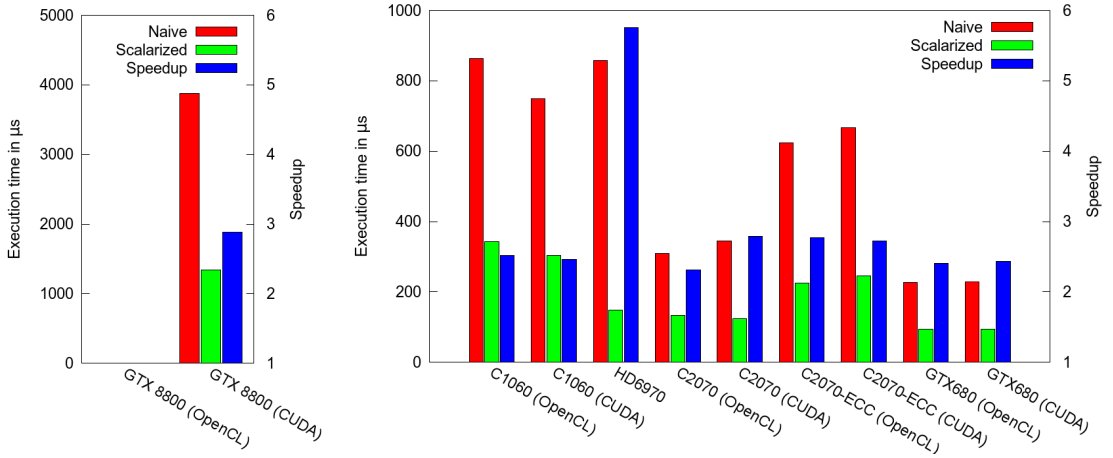


(b) Double precision.

Figure 7.11: Execution times in ms (best over twenty runs) and speedups for scalarization using CUDA and OpenCL for different AMD and Nvidia GPUs. The example is the `matmul` kernel shown in Figure 4.22, page 129, with $n_i = n_j = n_k = 2048$. The execution times presented here are kernel execution times. The GTX 8800 results are given aside because they are one order of magnitude slower. This architecture does not perform double precision computations: doubles are rounded to float before being processed. Hence, no OpenCL results are available in double precision for this GPU.

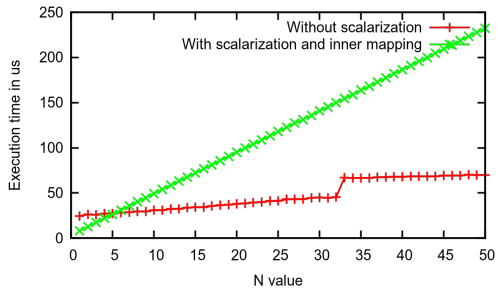


(a) single precision.

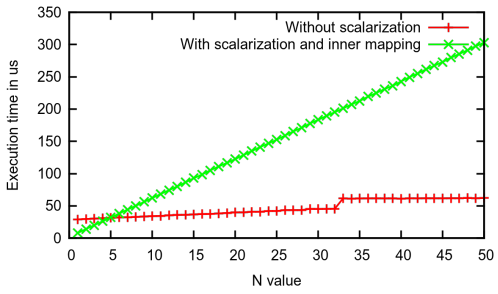


(b) Double precision.

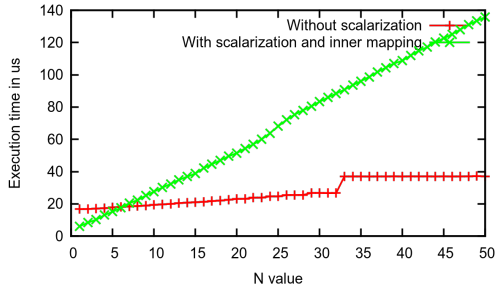
Figure 7.12: Execution times in μs (best over twenty runs) and speedups for scalarization using CUDA and OpenCL for different AMD and Nvidia GPUs. The code is the Scilab script after conversion to C and loop fusion shown in Figure 4.21b, page 129. The execution times shown are the kernel execution times. The GTX 8800 results are given aside because they are one order of magnitude slower. This architecture does not perform double precision computations: doubles are rounded to float before being processed. Hence, no OpenCL results are available in double precision for this GPU.



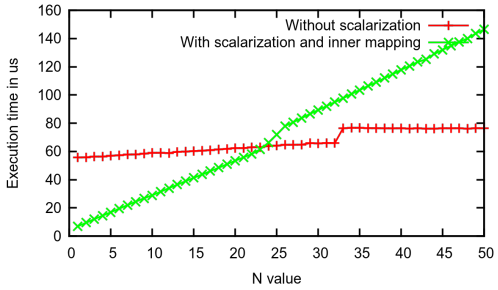
(a) GTX 8800



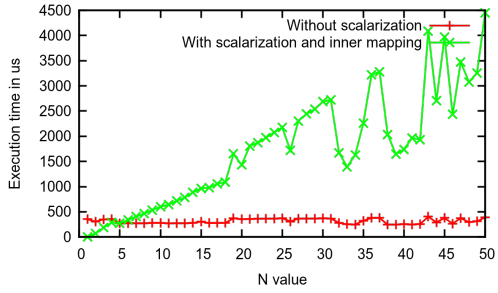
(b) C1060



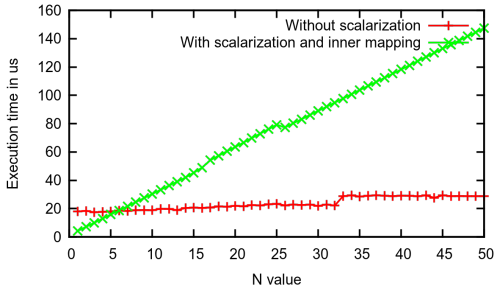
(c) C2070 without ECC



(d) C2070 with ECC



(e) HD6970



(f) GTX 680

Figure 7.13: Execution times in μs (best over twenty runs) for the code shown in Figure 4.23c and Figure 4.23d. The value of `M` is 8192 and the values of `N` are on the x axis. The Nvidia board shows clearly a shift on the red curve for $N = 32$ corresponding to the the warp size.

interesting only for very small values of `N`. An off-line profiling of the two versions would provide the test to select the best versions according the `M` and `N` values at runtime (see Section 5.7, page 158).

7.6 Loop Unrolling

Some kernels include a sequential loop inside the kernel and an unrolling loop at source level as shown in Section 4.8, which can have some impact on the performance. Figure 7.14 explores the performance impact of the unrolling of the inner loop in the matrix multiplication kernel presented in Figure 4.22 page 129. Many unroll factors are tested, showing different results depending on the target architecture.

It is possible to get better performance after unrolling on all architectures. However, the factors differ and so do the speedups. While the Fermi architecture benefits from a 1.35 speedup using an unroll factor of eight, the Kepler architecture exhibits only a 1.025 acceleration using an unroll factor of two. The G80 does not benefit from loop unrolling as much as the last architectures, with a 1.04 speedup using an unroll factor of thirty-two.

Finally, the impact of unrolling on the register pressure is shown Figure 7.15 for Nvidia GPUs. The code generated for Fermi shows that the register count is increased by a factor up to ten and then stays relatively stable. On the other hand, older architectures have a generated code using a number of registers that scales linearly with the unroll factor. The Nvidia compiler targets a pseudo-assembly language: PTX. The PTX is using an arbitrary number of registers, and the output of the compiler is Simple Static Assignment (SSA) form. The Nvidia binary compiler, *ptxas*, reports the final number of register when compiling the PTX to ISA. The ISA is not released by Nvidia, and therefore the origin of register usage is hard to track in the PTX.

Excessive register pressure implies more spilling. Since the number of registers is shared among threads in flight on a CU, increasing the requirement makes the occupancy dramatically decrease; i.e., the hardware scheduler does not have enough thread in flight to hide memory latency. This issue is also explored in Section 5.8.1, page 159, with the selection of the launch configuration.

7.7 Array Linearization

While the array linearization presented in Section 4.9 is a transformation mandatory to produce a valid OpenCL kernel, the performance impact is not known. Since CUDA makes it possible to apply it or not, it is possible to generate and compare both versions.

The benchmark used the simple matrix multiplication example presented in Figure 4.22. The result presented in Figure 7.16 shows that the impact varies with the target architecture. It can lead to a slowdown of up to twenty percent, but also in one configuration—the

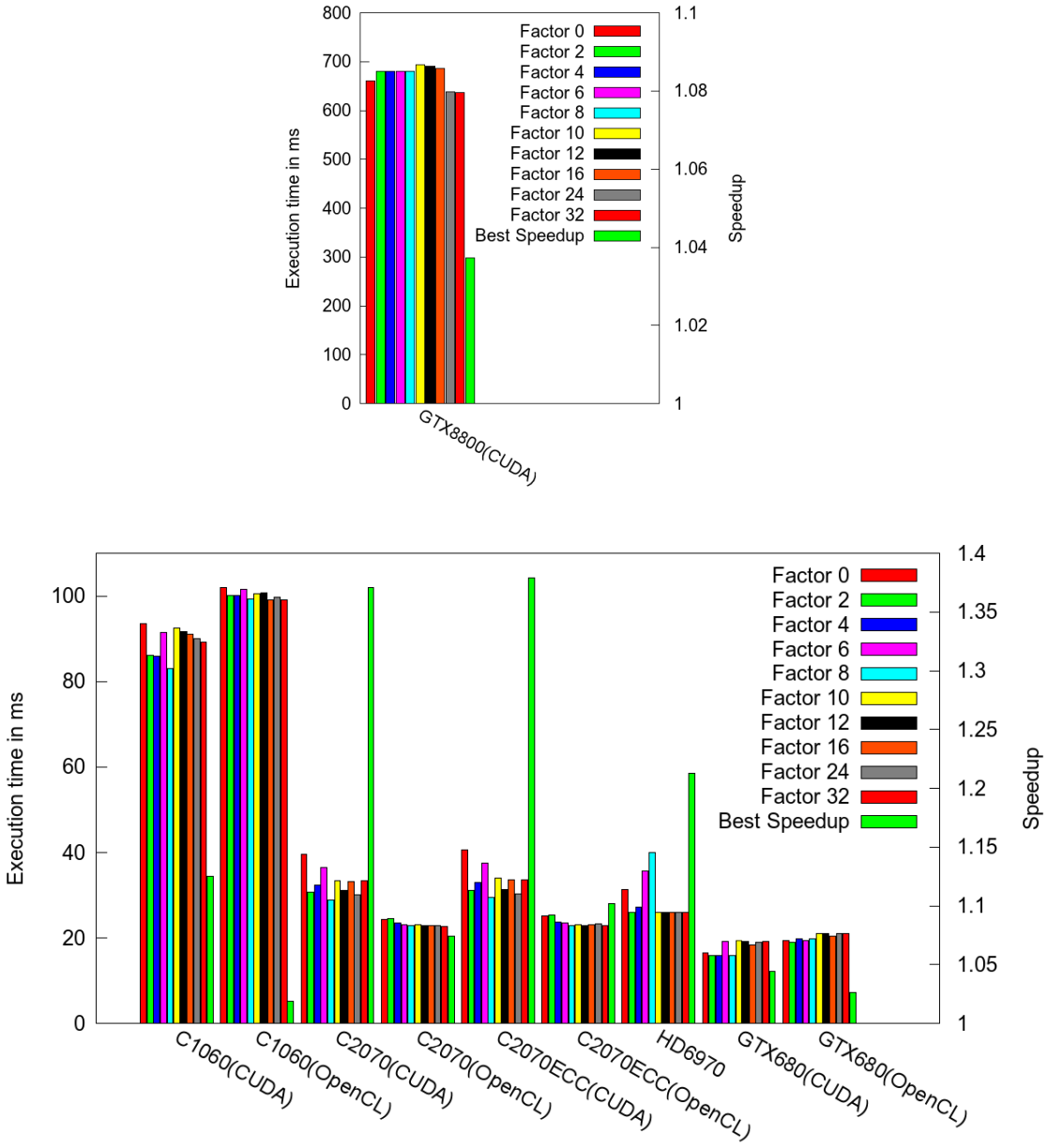


Figure 7.14: Execution time in µs (best over twenty runs) and speedup for different unrolling factor using CUDA and OpenCL for different AMD and Nvidia GPUs. The execution times are the kernel execution times. Single precision floating point is used. The example is the code presented in Figure 4.22, page 129, used in the previous section. Loop unrolling is applied after scalarization to obtain the code shown in Figure 4.24, page 133. The GTX 8800 results are given aside because they are one order of magnitude slower.

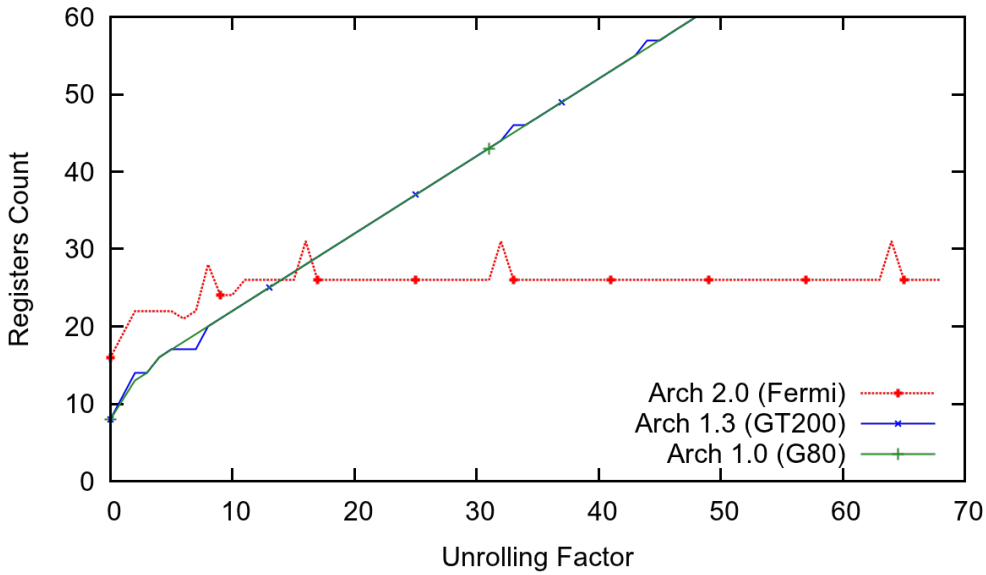


Figure 7.15: Register counts for different Nvidia architectures and different unrolling factors.

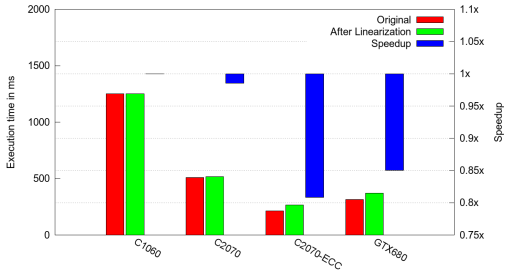
GTX 680 with single precision—can lead to a small speedup of about two percent.

7.8 Communication Optimization

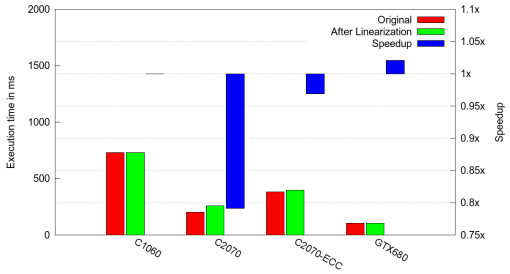
This section details the experiments carried out to validate the communication optimizing scheme presented in Chapter 3.

7.8.1 Metric

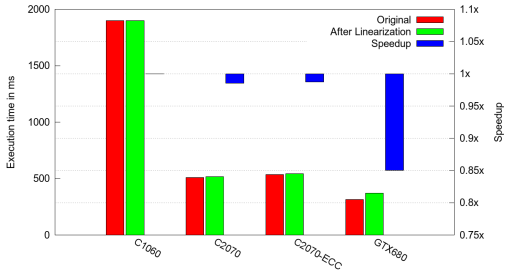
The first question is: what should we measure? While speedups in terms of CPU and wall clock time are most important to users, if not to administrators, many parameters impact the results obtained. Let us consider for example the popular *hotspot* benchmark [Huang *et al.* 2006]. Its execution time depends on two parameters: the matrix size and the number of time steps. In the general context of GPU, the matrix size should be large enough to fully load the GPU. For my communication optimization scheme, the time step parameter is at least as important since data transfers are hoisted out of the time step loop. Figure 7.17 shows how *hotspot* is affected by the number of time step iterations and approaches an asymptote; acceleration ranges from 1.4 to 14. The single speedup metric



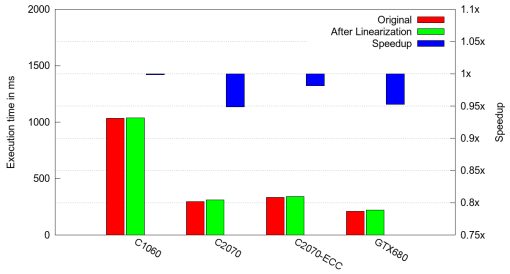
(a) single precision, no scalarization.



(b) single precision, with scalarization.



(c) Double precision, no scalarization.



(d) Double precision, with scalarization.

Figure 7.16: Kernel execution times in ms (best over twenty runs) and speedups for the array linearization transformation and different Nvidia GPUs, and with and without the scalarization illustrated in Figure 7.11. The example is the `matmul` kernel shown in Figure 4.22, page 129, with $n_i = n_j = n_k = 2048$.

is not adapted to properly evaluate my scheme.

A more objective measurement for evaluating my approach is the number of communications removed and the comparison with a scheme written by an expert programmer. Not only being inappropriate, focusing on the speedup would also emphasize the parallelizer capabilities.

The number of memory transfers generated for each version of the code is counted. When the communication occurs in a loop, this metric is parametrized by the surrounding loop iteration space. For instance, many benchmarks are parametrized with a number of time steps t, thus if three memory transfers are present in the time step loop, and two are outside of the loop, the number of communications will be expressed as $(3 \times t) + 2$.

Sometimes a loop that iterates over a matrix dimension cannot be parallelized, either intrinsically or because of the limited capability of the compiler. Memory transfers in such loop have a huge performance impact. In this case n is used to emphasize the difference

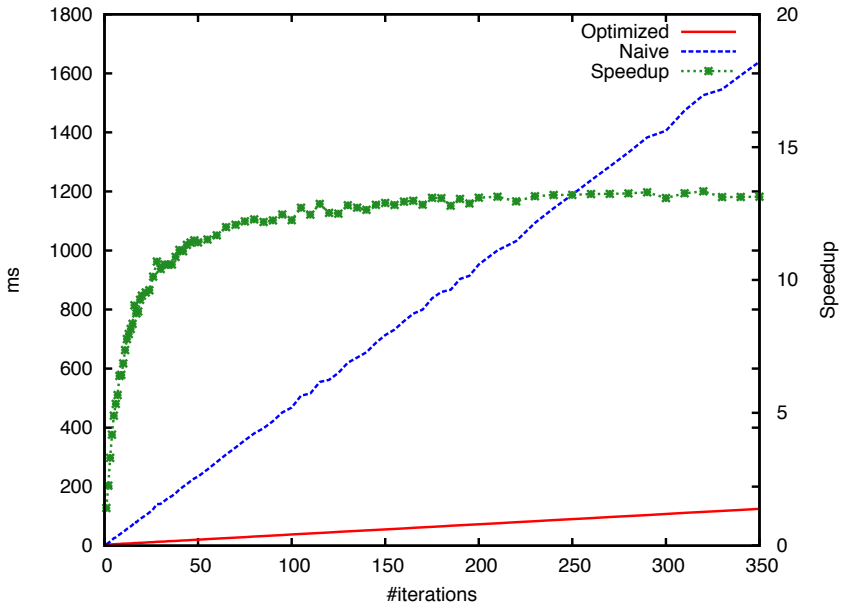


Figure 7.17: Execution times and speedups for versions of hotspot on GPU, with different iteration counts.

with the time step iteration set.

7.8.2 Results

This optimization is evaluated using the metric introduced in the previous section. It is more relevant to evaluate my optimization with the number of communications removed. The result in Table 7.2 shows that, when counting the number of memory transfers, the optimized code performed is very close to a hand-written mapping. One noticeable exception is `gramschmidt`. As explained in Section 3.7.1, page 86, communications cannot be moved out of any loop due to data dependencies introduced by some sequential code. The parallel promotion scheme shown in Section 3.7, page 84, and evaluated in Section 7.8.4 helps by accepting a more slowly generated code and allowing data to stay on the accelerator.

7.8.3 Comparison with Respect to a Fully Dynamic Approach

This section compares the static scheme to a runtime-based system like the StarPU [Augonnet *et al.* 2011] library. StarPU is used here in its 0.4 version, configured and compiled with its default options.

	2mm	3mm	adi	atax	bicg	correlation	covariance	doitgen	fdtd-2d
Naive	8	12	15t +9nt	6 +4n	6 +4n	18	15	5	13t
Opt.	5	6	4	3	5	2	2	3	6
Hand	4	5	4	3	5	2	2	3	6

	gauss-filter	gemm	gemver	gesummv	gramschmidt	jacobi-1d	jacobi-2d
Naive	7	4	17	7	9n	6t	6t
Opt.	4	4	10	3	2n+1	3	3
Hand	4	4	10	3	3	3	3

	lu	mvt	symm-exp	syrk	syr2k	hotspot99	lud99	srad99	Stars-PM
Naive	4n	8	6n	5	6	5t	4n	24t	25t
Opt.	2	7	4	3	4	3	2	t+1	3
Hand	2	7	4	3	4	3	2	t+1	3

Table 7.2: Number of memory transfers after parallelization using Par4All naive allocation, using my automatic optimizing scheme, and as a developer would have put it.

A test case included with the Par4All distribution is used and rewritten using StarPU in order to evaluate the overhead of the dynamic management of communication compared to my static scheme. This example performs 400 iterations of a simple *Jacobi* scheme on a 512×512 pixel picture loaded from a file and stores the result in another file. Figure 7.18 presents the core computation of this program and the processing using Par4All, while Figure 7.19 shows the encapsulation of the previously shown kernel in a StarPU task, and the use of StarPU to manage the task.

StarPU's traces show that all spurious communications are removed, just as my static scheme does. The manual rewrite using StarPU and a GPU with CUDA offers a 162 ms execution time for the StarPU version while the statically optimized scheme performed in 32 ms. The speedup is about five.

Although StarPU is a library that has capabilities ranging far beyond the issue of optimizing communications, the measured overhead confirmed that my static approach can be relevant.

```
static __global__ void flipflop(real_t src[SIZE][SIZE],
                                real_t dest[SIZE][SIZE]) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i<SIZE-2&&j<SIZE-2)
    dest[i+1][j+1] = 0.25*(src[i+1-1][j+1]+src[i+1+1][j+1]+
                           src[i+1][j+1-1]+src[i+1][j+1+1]);
}
```

(a) CUDA kernel for one Jacobi 2D iteration.

```
gettimeofday(&p4a_time_begin, NULL);
// Send data
P4A_runtime_copy_to_accel(space, sizeof(real_t)*SIZE*SIZE);
P4A_runtime_copy_to_accel(save, sizeof(real_t)*SIZE*SIZE);
for(t = 0; t < 400; t++) {
  dim3 dimGrid(SIZE/32+SIZE%32,SIZE/8+SIZE%8,1);
  dim3 dimBlock(32,8,1);
  real_t (*d_space)[SIZE] = P4A_get_accel_ptr(space);
  real_t (*d_save)[SIZE]  = P4A_get_accel_ptr(save);
  flipflop<<<dimGrid,dimBlock>>>(d_space, d_save);
  flipflop<<<dimGrid,dimBlock>>>(d_save, d_space);
  cudaThreadSynchronize();
}
// Get back data from the GPU
P4A_runtime_copy_from_accel(space, sizeof(real_t)*SIZE*SIZE);

gettimeofday(&p4a_time_end, NULL);
```

(b) Host code using Par4All runtime.

Figure 7.18: Illustration for the code used to measure performance of the static approach on a Jacobi 2D scheme.

7.8.4 Sequential Promotion

The transformation introduced in Section 3.7 consists of mapping some intrinsically sequential code onto one GPU thread. This transformation is performed to avoid some costly transfer over the PCIe bus. Two examples from the Polybench suite provide interesting results when using this scheme, `gramschmidt` shown in Figure 3.17 page 85 and `durbin` shown in Figure 7.20. The communication count goes from $2n + 1$ down to 3 for `gramschmidt` and from $3n + 4$ down to 2 for `durbin`.

In terms of execution times, Figure 7.21 presents the times measured on two architec-

```
void compute_cuda_func(void *buffers[], void *_args) {
 real_t (*space)[SIZE] = STARPU_MATRIX_GET_PTR(buffers[0]);
 real_t (*save)[SIZE]  = STARPU_MATRIX_GET_PTR(buffers[1]);

 dim3 dimGrid(SIZE/32+SIZE%32,SIZE/8+SIZE%8,1);
 dim3 dimBlock(32,8,1);
 flipflop<<<dimGrid,dimBlock>>>(space, save);
 flipflop<<<dimGrid,dimBlock>>>(save, space);
 cudaThreadSynchronize();
}
```

(a) StarPU task to encapsulate the kernel launch.

```
gettimeofday(&time_begin, NULL);
for(t = 0; t < 400; t++) {
  starpu_task_submit(task);
}
starpu_task_wait_for_all();
// Require a copy-out from the runtime
starpu_data_acquire(space_handle, STARPU_R);
gettimeofday(&time_end, NULL);
```

(b) Host code using the StarPU runtime.

Figure 7.19: Illustration for the code used to measure performance for the StarPU version of the Jacobi 2D scheme.

tures, a Tesla C1060 and a Fermi C2070. The speedups are very high for the `durbin` example, respectively 37.2 and 35.6, and less impressive but still very interesting for `gramschmidt` with respectively 8.3 and 6.5.

7.9 Overall Results

Figure 7.22 shows results for twenty benchmarks of the Polybench suite, three from Rodinia, and the Stars-PM application (see Section 3.1). Measurements were performed on a machine with two Xeon Westmere X5670 (twelves cores at 2.93 GHz) and a Nvidia GPU Tesla C2050. The OpenMP versions used for the experiments are generated automatically by Par4All and are not manually optimized or modified. The OpenMP version runs on the twelves X5760 cores only and does not use the GPU.

Kernels are exactly the same for the two automatically generated CUDA versions using Par4All, the naive and the one with communication optimization. The Nvidia CUDA SDK

```
for (k = 1; k < n; k++) {
  beta[k] = beta[k-1] -
            alpha[k-1]*alpha[k-1]*beta[k-1];
  sum[0][k] = r[k];
  for (i = 0; i <= k-1; i++)
    sum[i + 1][k] = sum[i][k] +
                    r[k-i-1] * y[i][k-1];
  alpha[k] = -sum[k][k] * beta[k];
  for (i = 0; i <= k-1; i++)
    y[i][k] = y[i][k-1] +
              alpha[k] * y[k-i-1][k-1];
  y[k][k] = alpha[k];
}
for (i = 0; i < n; i++)
  out[i] = y[i][N-1];
```

```
copy_to_accel(r);
kernel_0(r, sum, n);
copy_from_accel(sum);
for(k = 1; k <= n-1; k += 1) {
  beta[k] = beta[k-1]-alpha[k-1]*
            alpha[k-1]*beta[k-1];
  for(i = 0; i <= k-1; i += 1)
    sum[i+1][k] = sum[i][k]+
        r[k-i-1]*y[i][k-1];
  alpha[k] = -sum[k][k]*beta[k];
  copy_to_accel(alpha);
  kernel_1(alpha, y, k);
  copy_from_accel(y);
  y[k][k] = alpha[k];
  copy_to_accel(y);
}
kernel_2(out, y, n);
copy_from_accel(out);
```

(a) Usual host code.

```
copy_to_accel(r);
kernel_0(r, sum, n);

for(k = 1; k <= n-1; k += 1) {
  // Sequential code promoted
  // on the GPU
  sequential_kernel(alpha,
                    beta,
                    r, sum,
                    y, k);

  kernel_1(alpha, y, k);

  sequential_kernel(alpha,y);

}
kernel_2(out, y, n);
copy_from_accel(out);
```

(b) Host code after sequential promotion.

Figure 7.20: `durbin` example from Polybench that shows the interest of sequential promotion.

	C1060			C2070		
	Classic	Promoted	**Speedup**	Classic	Promoted	**Speedup**
Durbin	200149.84	5374.74	**37.2**	127865.4	3592.7	**35.6**
Gramschmidt	64450.5	7732.6	**8.3**	38999.2	6010.08	**6.5**

Figure 7.21: Execution times in ms and speedups for CUDA execution with communication optimization, using the classic scheme and the sequential promotion. The result are based on the average over five runs for `durbin` and `gramschmidt` examples (see Figures 7.20 and 3.17).

4.0, and GCC 4.4.5 with `-O3` were used.

For Polybench, Par4All was forced to ignore array initializations because they are both easily parallelized and occur before any of the timed algorithms. The normal optimizer configuration would thus have "optimized away" the initial data transfer to the GPU within the timed code. It seems more relevant to prevent this optimization even though it is not advantaging Par4All results.

The measurements in Figure 7.22 include all communications.

For the cosmological simulation (see Section 3.1), the communication optimization speeds up execution by a factor of 14 compared to the version without my optimization, and 52 compared to the sequential code.

Results are provided for HMPP and PGI Accelerator. Only basic directives were added by hand. We did not use more advanced options of the directives, thus the compiler does not have any hints on how to optimize communications.

The geometric mean over all test cases shows that this optimization improves by a factor of 4 to 5 over Par4All, PGI Accelerator and HMPP naive versions.

One noticeable exception is `gramschmidt`. Communications cannot be moved out of any loop due to data dependencies introduced by some sequential code. The results for the parallel promotion scheme (see Section 7.8.4) show that accepting a slower generated code but allowing data to stay on the accelerator leads to performance improvement on such code. My measure provides a 6.5 times speedup over the optimized scheme on this architecture, which means a speedup of 17.5 over the sequential code, closer to HMPP results.

The geometric mean of the speedup obtained with the communication optimization scheme is over 14, while a naive parallelization using CUDA achieves a speedup of 2.22, and on the CPU only the automatic OpenMP loop parallelization one of 3.9.

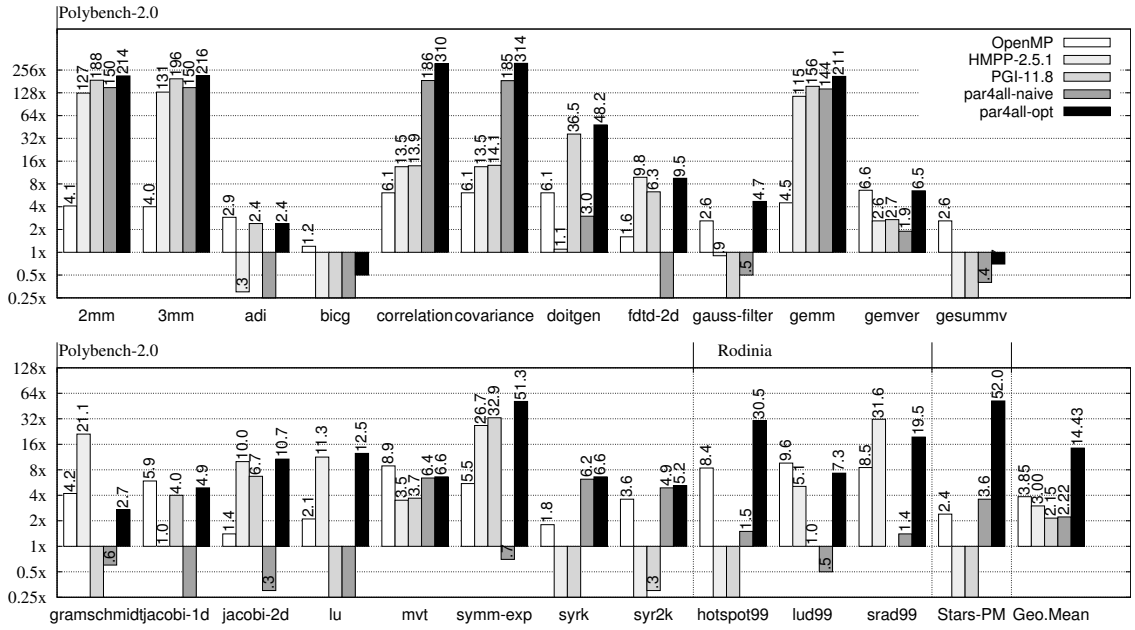


Figure 7.22: Speedup relative to naive sequential version for an OpenMP version on the CPU, a version with basic PGI Accelerator and HMPP directives, a naive CUDA version, and an optimized CUDA version, all automatically generated from the naive sequential code.

7.10 Multiple GPUs

7.10.1 Task Parallelism

This section measures the benefit obtained by attaching multiple GPUs to one host, using current technologies and a runtime like StarPU introduced in Section 6.2.1. Two different examples were selected from the Polybench benchmark suite. The first one is `3mm` (see Figure 6.1 page 168) and the second one is `bicg` (see Figure 7.23).

The `3mm` example uses three kernels. The first two can run independently while the third one consumes the results of the two previous ones. The expected execution timeline is shown in Figure 6.3, page 170. The `bicg` example does not exhibit any task parallelism but tests the scheduling capacity and overhead by launching more than 8000 dependent kernels. A greedy scheduler that allocates a new task to a new GPU for each launch would result in a very inefficient mapping. Indeed a large number of memory transfers would occur while ensuring synchronization since every kernel is dependent on the result of the

previous one.

The `3mm` example contains large kernels whose execution times dominate the communication times. On the contrary, the `bicg` example contains a large number of very small kernels whose execution times are lower than the communication times individually, but a good mapping avoids most of the communications.

Figure 7.23 shows the average execution times over ten runs for both examples on two different multi-GPU configurations, one with two C2070 and the other with three C1060.

Much can be observed from these measurements. Firstly, using one GPU, the greedy scheduler provides better results than the static scheduling. The static scheduling is synchronous, while StarPU uses asynchronous scheduling and thus some communication and computation overlapping occurs. Also, the time to take scheduling decisions on the host is hidden by the computation time on the GPU.

Using two GPUs, the `3mm` example shows a speedup up to 1.47, which is consistent with the expected speedup from the timeline in Figure 6.3. The `bicg` example with two GPUs shows the impact of the scheduler selection. A data-aware scheduler, while not perfect, ends up about *only* two times slower than the same code using one GPU. On the other hand, the default greedy scheduler ends up with totally inefficient mapping and an execution time up to 8127 slower than the time measured using only one GPU.

Finally, the three GPU configuration confirms the slowdown exhibited by the default StarPU scheduler on the `bicg` example with two GPUs. The `3mm` example, thanks to its favorable task graph, does not benefit from the scheduler, but at least it confirms that the overhead due to the more complex scheduler is hidden by the computations that run asynchronously.

7.10.2 Data Parallelism

Section 6.3 illustrates how parallel tiles in a loop nest can be generated with symbolic tiling, delaying the choice of the tile size and the mapping on the hardware at runtime.

This section evaluates this scheme using a variant of the vector scaling example shown in Figure 7.25. The kernel is modified to include a loop that artificially increases the computation time. This is necessary in order to experiment with the current mapping of Nvidia OpenCL runtime on high-end hardware.

Figure 7.26 shows the mapping at runtime using one GPU and from one to ten queues associated with it. Figure 7.27 involves two GPUs and shows the mapping for one to five queues associated with each of them. The main observation is that the Nvidia OpenCL

```
for(i=0;i<ny;i++)
 s[i] = 0;

for(i=0;i<nx;i++){
 q[i] = 0;
 for(j=0;j<ny;j++){
  s[j] = s[j] + r[i] * A[i][j];
  q[i] = q[i] + A[i][j] * p[j];
 }
}
```

(a) Input code.

```
for(i=0;i<ny;i++)// Parallel
 s[i] = 0;

for(i=0;i<nx;i++){// Parallel
 q[i] = 0;
 for(j=0;j<ny;j++)// Sequential
  q[i] = q[i] + A[i][j] * p[j];
}
for(i=0;i<nx;i++)// Sequential
 for(j=0;j<ny;j++)// Parallel
  s[j] = s[j] + r[i] * A[i][j];
```

(b) After parallelization, ready to be transformed into kernels.

```
kernel1(s, ny);
kernel2(A, p, q, nx, ny);
for(i=0;i<nx;i++)
  kernel3(A, s, i, ny, r_0);
```

(c) After task transformation.

Figure 7.23: `bicg` example from Polybench that shows the impact of the different StarPU schedulers on a sequential example. There is a direct dependence between each of the kernels. Here $nx = ny = 8000$, thus the `kernel3()` is executed 8000 times.

	3mm	bicg
One C2070 static synchronous scheduling	962.3	175.3
One C2070 StarPU default scheduler	943.8	163.9
One C2070 StarPU data-aware scheduler	942.6	208.1
Two C2070 StarPU default schedulers	809.5	859517.2
Two C2070 StarPU data-aware schedulers	724.8	404.7
One C1060 static synchronous scheduling	3231.8	301.2
One C1060 StarPU default scheduler	3167.6	280.2
One C1060 StarPU data-aware scheduler	3177.0	279.8
Two C1060 StarPU default schedulers	2151.5	2446426.6
Two C1060 StarPU data-aware schedulers	2151.1	629.6
Three C1060 StarPU default schedulers	2162.5	2199509.7
Three C1060 StarPU data-aware schedulers	2155.8	599.7

Figure 7.24: Average execution time in ms over ten runs for `3mm` and `bicg` examples. Note the impact of different StarPU schedulers, the default greedy one and the data-aware, *dmda*.

```
P4A_accel_kernel p4a_kernel_scal_1(int I_up,  int I_low,
                                   double  *A0, double  *B0,
                                   double c) {
 int i = get_global_id(0);
 if (i<=I_4-I_5) {
  for(int j=0; j<N; j++)
   *(A0+i) += *(B0+i)*c;
 }
}
```

Figure 7.25: The vector scaling example presented in Figure 6.4, modified to increase the computation time.

runtime implementation is far from being perfect, especially when overloaded with many parallel queues.

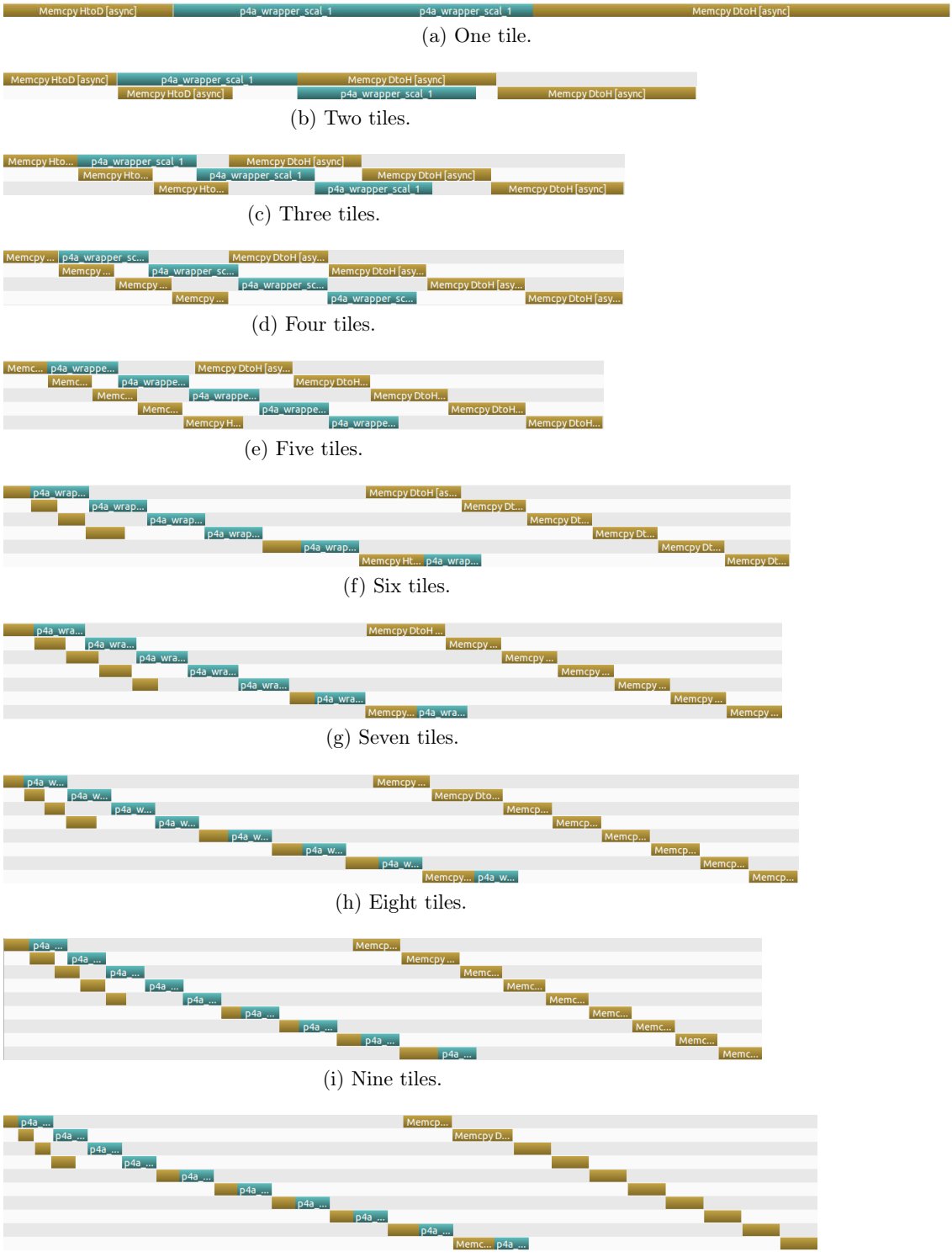
However, using only one GPU, the overlapping of computations and communications shows interesting speedups of up to 1.63. Using two GPUs does not provide any further improvements. We suppose that the main limitation is in the OpenCL runtime implementation. It serializes of the transfers to and from the two GPUs. The timelines presented in Figures 7.26 and 7.27 are promising, as the kernels are mapped on the different GPUs available and the communications are overlapped. One can expect future implementations of the runtime and future pieces of hardware to be able to provide higher speedups for this mapping scheme.

7.11 Conclusions

Many experiments are presented in this chapter to validate the transformations and the different schemes presented in Chapters 3, 4, 5, and 6 in this dissertation. Multiple architectures are tested to ensure that the approach is portable enough to provide a good retargetability.

The results show that the approach is rather robust. Measures on a n-body numerical simulation code show speedups of twelve compared to a naive parallelization and eight compared to the automatically generated OpenMP version on two six-core processors.

Some early work about multi-GPUs mapping is presented, and the measurements show that while the Nvidia OpenCL implementation is not perfect yet, opportunities for speedup already exist. Future work in this area is promising.



(a) One tile.

(b) Two tiles.

(c) Three tiles.

(d) Four tiles.

(e) Five tiles.

(f) Six tiles.

(g) Seven tiles.

(h) Eight tiles.

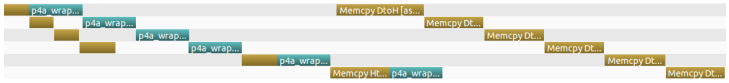
(i) Nine tiles.

(j) Ten tiles.

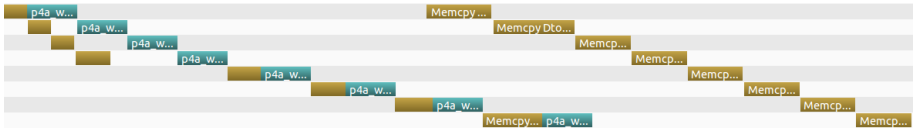
Figure 7.26: Output from Nvidia Visual Profiler showing the communications in brown and the kernel executions in blue. The mapping is done using one C2070 only. The copy-out do not overlap properly with copy-in, which is unexpected and limits the acceleration that can be achieved.

(a) Two tiles.

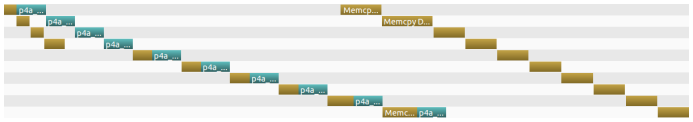
(b) Four tiles.



(c) Six tiles.



(d) Eight tiles.



(e) Ten tiles.

Figure 7.27: Output from Nvidia Visual Profiler showing the communications in brown and the kernel executions in blue. The mapping is done using two C2070s. The copy-out do not overlap properly with copy-in, which is unexpected and limits the acceleration that can be achieved.

Chapter 8

Conclusion

Welcome to the hardware jungle.

Herb Sutter's statement [Sutter 2011] is more than ever a burning issue. From the cell phones in every pocket to the supercomputers, the hardware is more and more heterogeneous and challenging to program.

No satisfying solution has been proposed for the three P properties: Performance, Portability, and Programmability. At least no solution is close to having unanimous support from programmers.

The jungle is now also visible in the software world, with more and more programming models, new languages, different APIs, etc. Although no *one-fits-all* solution has emerged, I have designed and developed a compiler-based automatic solution to partially answer the problem. The programmability and portability are enforced by definition, the performance may not be as good as what can be obtained by an expert programmer, but is still excellent for a wide range of kernels and applications. By using a GPU instead of the CPU, I obtained a speed increase by a factor of fifty-two for an n-body numerical simulation when compared to a naive sequential code compiled with GCC.

This work explores the many issues associated with a fully automatic solution, addressing the three P issues by changing neither the programming model nor the language, but by implementing compiler transformations and runtime support backed by semantics analyses in a compiler. The portability and programmability are obviously matched; the programmer does not have to know well the underlying platform.

The trade on performance is limited, as shown by many experiments presented in Chapter 7. They also show that portability of performance can be achieved over multiple architectures.

Contributions

I designed and implemented several new transformations and analyses in Par4All and in the PIPS source-to-source framework compiler. I also modified state-of-the-art trans-

formations to obtained a GPU-specific scheme, as shown in Figure 8.1.

I follow the chronological process of the figure rather than the chapter order to present my contributions.

I designed and implemented new induction variable substitution transformations based on linear precondition analysis (see Section 4.5, page 111). This transformation enables the parallelization of loops in spite of induction variables.

I studied the combinations of two different parallelizing algorithms, with an analysis of the impact on code generation of both of them in Section 4.3, page 101.

I improved the existing reduction detection analysis to handle C code more accurately, and leveraged this analysis to enable parallelization of loops with reduction by improving the existing parallelization algorithms. I implemented a mapping scheme for some loops with reductions onto the GPU using atomic operations supported by OpenCL and CUDA (see Section 4.4, page 105). Actually, I proposed a new generic scheme for parallelizing loops with reduction, and implemented it in PIPS. It provides improvements for other targets like multicore using OpenMP code generation.

I implemented two loop fusion transformation phases: one based on the dependence graph and the other on array regions. I designed heuristics to drive the fusion in order to target GPUs. This is particularly critical when processing code generated from high-level tools and languages, such as Scilab, which include many loops. This transformation allows a later scalarization phase to remove many temporary arrays generated by such tools.

I studied different array scalarization schemes in the context of GPGPU in Section 4.7, page 127, and I modified the PIPS implementation to match requirements for GPU code generation, especially to enforce the perfect nesting of loops.

Loop unrolling was presented in Section 4.8, page 132, and its impact was shown and analyzed with several experiments.

A runtime system to map loops on the GPU in a flexible manner, including transformations like tiling or loop interchange, is proposed in Section 4.2, page 98.

The issue of generating code for mapping data communications between the CPU and the GPU is studied in Chapter 3. Convex array regions are used to generate accurate communication. I designed and implemented an analysis, *Kernel Data Mapping*, a transformation, and a runtime system to perform interprocedural optimization to preserve the data on the GPU as long as possible to avoid redundant communications.

Code generation for multiple GPUs is addressed in Chapter 6. Two different sources of parallelism are presented and compared. I implemented a simple task extraction trans-

formation, and a code generator targeting the StarPU runtime system. I also modified the existing symbolic tiling transformation to distribute loop nests on multiple GPUs. I implemented a dedicated runtime system to perform this distribution using OpenCL.

The whole process is automated thanks to a programmable pass manager (see Chapter 5). I implemented a generic *stubs broker* in charge of handling external library used in the processed code. It integrates the compilation chain and provides information to the final binary linking process.

I also implemented a runtime system to manage kernel launches; especially I developed a heuristic to choose at runtime a launch configuration based on the data size and on the target architecture.

Finally, I conducted a large range of experiments and analyses in Chapter 7 to validate all proposed transformations, schemes, and the implemented runtime systems. Twenty benchmarks of the Polybench suite, three from Rodinia, and the Stars-PM n-body application were used. The geometric mean speedup over all test cases that I obtain is over fourteen when compared to the sequential input code compiler with GCC. Stars-PM is accelerated by a factor fifty-two. Using multiple GPUs provides a speedup up to 1.63 when compared to the automatic compilation using only one GPU.

Following Arch Robison's statement [Robison 2001]

> Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers.

The major contribution of this work, beyond the concepts, is included in an industrial integrated prototype released as open-source: the Par4All compiler. Based on the PIPS framework, it provides developers an automatic solution to achieve retargetability. The interface is minimal and Par4All can be invoked as simply as: `p4a --opencl my_code.c -o my_binary`.

I also implemented a Fortran 95 frontend in PIPS. It is present in the form of a bridge that converts the abstract syntax tree from the Fortran 95 parser in `gfortran` to the PIPS internal representation.

Moreover, the SILKAN company has applied the techniques presented in this dissertation in a user-friendly environment and delivered to Scilab programmers a *one-click* access to the power of hardware accelerators like GPUs.

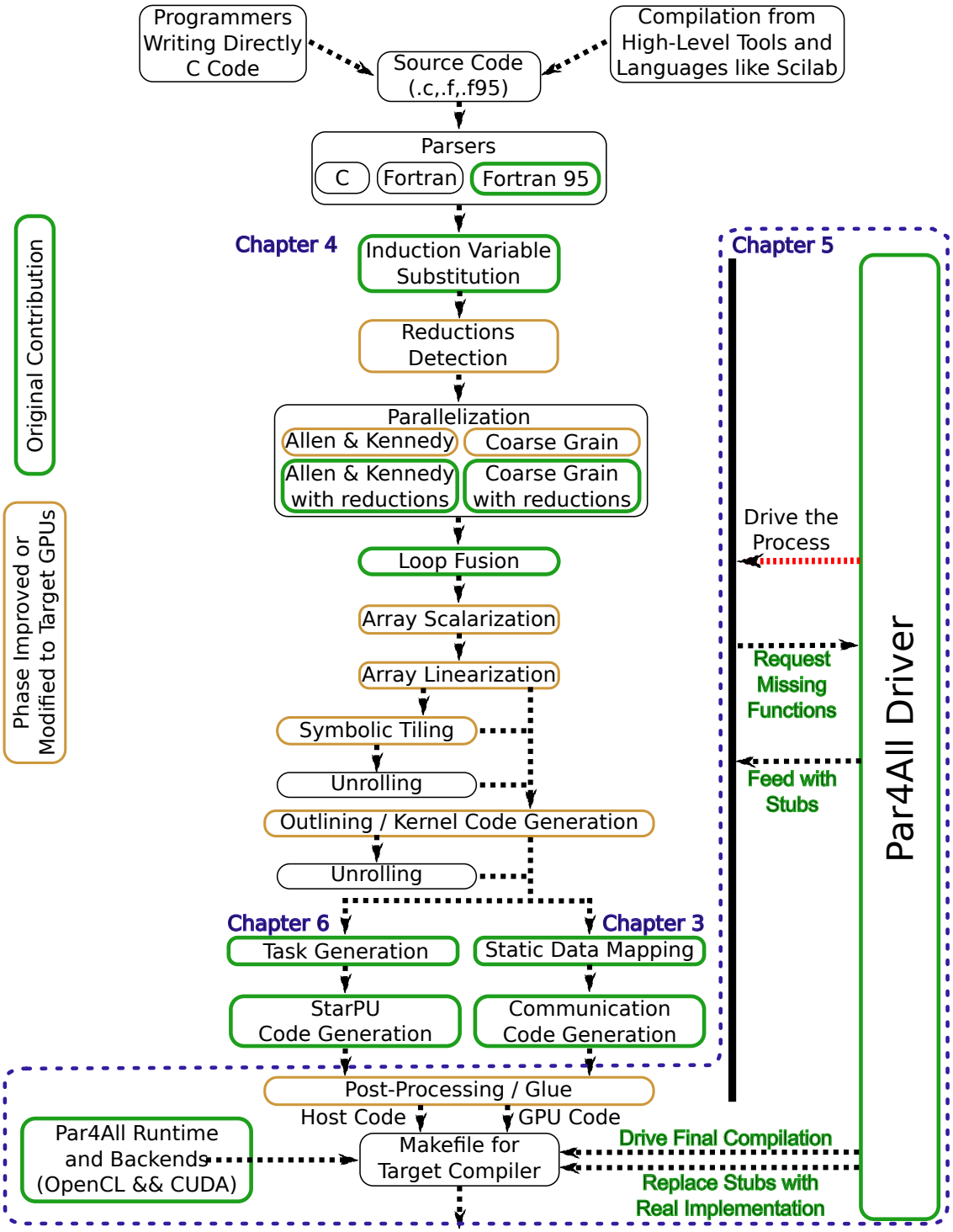


Figure 8.1: Overview of the global compilation scheme.

Future Work

Chapter 6 contains some preliminary work about multiple GPUs, with encouraging experimental results. More work has to be done on kernel-specific optimization for multiple GPUs. For instance, loop nests have to be restructured to ensure better locality and to optimize the communications when distributing the data onto multiple GPUs.

In general, the trade-offs between the static decisions made by the compiler off-line and those made by the runtime systems have to be explored in more detail. The compiler could generate hints for the runtime system, such as estimation of the execution time of a kernel, or the usage of memory buffers as shown with StarPU in Chapter 6.

The data layout transformations are not studied in this dissertation. Some polyhedral tools may be well suited to perform this kind of transformation. Coupling such tools within Par4All should be explored further.

The use of OpenCL local memory is not addressed. It would be interesting to implement a scheme based on array regions and to compare it with polyhedral based solution.

The three *P*s Performance, Portability, and Programmability are nowadays completed with a fourth *P*: Power. Some libraries or Domain Specific Language (DSL) are designed to handle power-aware execution. Trading the existing three *P*s in a power-aware compiler offers room for further research.

Other paradigms emerge, for instance the ΣC dataflow language that comes along with the MPPA accelerator, both introduced in Section 2.2.9. While the authors of ΣC expect that it can be targeted using tools like Par4All [Goubier *et al.* 2011], more research is needed to achieve that goal.

Finally, the hardware improves year after year. This work started before Nvidia *Fermi* was available and the next generation *Kepler* is now about to be released with new capabilities. The programming model of the GPUs changes. The *dynamic parallelism* feature introduced by Nvidia paves the way toward more and more flexibility exposed by the hardware through APIs like CUDA. Automatically mapping programs without hints is more difficult in this new context.

The collaboration between code generators from high-level tools or programming languages and a retargetable compiler like Par4All is a crucial way to explore. The high-level semantic informations are known by the code generator, and be capable to propagate them in the optimizing compiler seems to be the path to follow.

The GPU computing era and more generally the heterogeneous computing era is still at its beginning and many interesting challenges will be thrown to compiler developers.

Personal Bibliography

[Amini *et al.* 2011a (perso)] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell and Pierre Villalon. *PIPS Is not (just) Polyhedral Software*. In 1st International Workshop on Polyhedral Compilation Techniques, Impact, (in conjunction with CGO 2011), April 2011. (Cited on pages 62, 78, 139, 281 and 300.)

[Amini *et al.* 2011b (perso)] Mehdi Amini, Fabien Coelho, François Irigoin and Ronan Keryell. *Compilation et Optimisation Statique des Communications Hôte-Accélérateur*. In Rencontres Francophones du Parallélisme (RenPar'20), May 2011. (Cited on page 94.)

[Amini *et al.* 2011c (perso)] Mehdi Amini, Fabien Coelho, François Irigoin and Ronan Keryell. *Static Compilation Analysis for Host-Accelerator Communication Optimization*. In The 24th International Workshop on Languages and Compilers for Parallel Computing, LCPC, 2011. (Cited on pages 62, 94 and 280.)

[Amini *et al.* 2012a (perso)] Mehdi Amini, Fabien Coelho, François Irigoin and Ronan Keryell. *Compilation et Optimisation Statique des Communications Hôte-Accélérateur (version étendue)*. Technique et Science Informatiques, Numéro spécial RenPar'20, 2012. To appear. (Cited on page 94.)

[Amini *et al.* 2012b (perso)] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François Xavier Pasquier, Grégoire Péan and Pierre Villalon. *Par4All: From Convex Array Regions to Heterogeneous Computing*. In 2nd International Workshop on Polyhedral Compilation Techniques, Impact, (in conjunction with HiPEAC 2012), January 2012. (Cited on pages 68, 98, 138, 139, 163, 283, 289, 299, 300 and 310.)

[Amini *et al.* 2012c (perso)] Mehdi Amini, Ronan Keryell, Beatrice Creusillet, Corinne Ancourt and François Irigoin. *Few Simple (Sequential) Programming Practices to Maximize Parallelism Detection*. Rapport technique, SILKAN, MINES-ParisTech CRI, 2012. (Cited on page 179.)

[Aubert *et al.* 2009 (perso)] Dominique Aubert, Mehdi Amini and Romaric David. *A Particle-Mesh Integrator for Galactic Dynamics Powered by GPGPUs*. In International Conference on Computational Science: Part I, ICCS '09, pages 874–883. Springer-Verlag, 2009. (Cited on pages 42, 64, 153, 282 and 306.)

[Guelton *et al.* 2011a (perso)] Serge Guelton, Mehdi Amini, Ronan Keryell and Béatrice Creusillet. *PyPS, a Programmable Pass Manager*. Poster at the 24th International Workshop on Languages and Compilers for Parallel Computing, September 2011. (Cited on pages xxii, 142, 143, 144, 145, 151, 302 and 304.)

[Guelton *et al.* 2011b (perso)] Serge Guelton, Mehdi Amini, Ronan Keryell and Béatrice Creusillet. *PyPS, a Programmable Pass Manager*. Rapport technique, CRI, MINES ParisTech, June 2011. (Cited on pages xxii, 142, 143, 144, 145, 151, 302 and 304.)

[Guelton *et al.* 2012 (perso)] Serge Guelton, Mehdi Amini and Béatrice Creusillet. *Beyond Do Loops: Data Transfer Generation with Convex Array Regions*. In The 25th International Workshop on Languages and Compilers for Parallel Computing, LCPC, 2012. To appear. (Cited on page 94.)

[Irigoin *et al.* 2011 (perso)] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet and Ronan Keryell. *Polyèdres et Compilation*. In Rencontres Francophones du Parallélisme (RenPar'20), May 2011. (Cited on pages 103, 114, 124, 291 and 293.)

[SILKAN 2010 (perso)] SILKAN. *Par4All Initiative for automatic parallelization*. `http://www.par4all.org`, 2010. (Cited on pages 8, 68, 138, 139, 263, 283, 299 and 300.)

Bibliography

[Advanced Micro Devices 2006] Advanced Micro Devices. *AMD Introduces World's First Dedicated Enterprise Stream Processor*, November 2006. Online; accessed 29-July-2012; available at http://www.amd.com/us/press-releases/Pages/Press_Release_114146.aspx. (Cited on page 20.)

[Adve 1993] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessor*. Rapport technique, University of Wisconsin-Madison, 1993. (Cited on pages 5 and 261.)

[Adve 2011] Sarita V. Adve. *Rethinking shared-memory languages and hardware*. In Proceedings of the International Conference on Supercomputing, ICS '11, pages 1–1, New York, NY, USA, 2011. ACM. (Cited on pages 5 and 261.)

[Aho & Ullman 1977] Alfred V. Aho and Jeffrey D. Ullman. Principles of compiler design. Addison-Wesley, Reading, Massachusetts, 1977. (Cited on pages 132 and 297.)

[Aho *et al.* 1986] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cited on pages 113 and 293.)

[Alias *et al.* 2011] Christophe Alias, Alain Darte and Alexandru Plesco. *Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis*. Rapport de recherche RR-7648, INRIA, June 2011. (Cited on pages 62, 87, 90, 91, 280 and 287.)

[Alias *et al.* 2012a] Christophe Alias, Alain Darte and Alexandru Plesco. *Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA*. In 2nd International Workshop on Polyhedral Compilation Techniques, Impact, (in conjunction with HiPEAC 2012), January 2012. (Cited on pages 87, 90, 91 and 287.)

[Alias *et al.* 2012b] Christophe Alias, Alain Darte and Alexandru Plesco. *Optimizing Remote Accesses for Offloaded Kernels: Application to High-level Synthesis for FPGA*. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 1–10, New York, NY, USA, 2012. ACM. (Cited on pages 87, 90, 91 and 287.)

[Allen & Cocke 1972] Frances Allen and John Cocke. A catalogue of optimizing transformations. Prentice-Hall series in automatic computation. Prentice-Hall, 1972. (Cited on pages 112, 113, 115 and 293.)

[Allen & Kennedy 1982] Randy Allen and Ken Kennedy. *PFC: A program to convert Fortran to parallel form.* MaSc Technical Report 826, 1982. (Cited on page 101.)

[Allen & Kennedy 1987] Randy Allen and Ken Kennedy. *Automatic translation of FORTRAN programs to vector form.* ACM Trans. Program. Lang. Syst., vol. 9, pages 491–542, October 1987. (Cited on page 101.)

[Allen *et al.* 1987] R. Allen, D. Callahan and K. Kennedy. *Automatic decomposition of scientific programs for parallel execution.* In Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '87, pages 63–76, New York, NY, USA, 1987. ACM. (Cited on page 115.)

[Allen *et al.* 1988] F. Allen, M. Burke, P. Charles, R. Cytron and J. Ferrante. *An overview of the PTRAN analysis system for multiprocessing.* In Proceedings of the 1st International Conference on Supercomputing, pages 194–211, New York, NY, USA, 1988. Springer-Verlag New York, Inc. (Cited on page 173.)

[Allen 1983] John Randal Allen. *Dependence analysis for subscripted variables and its application to program transformations.* PhD thesis, Rice University, Houston, TX, USA, 1983. AAI8314916. (Cited on pages 112, 115 and 293.)

[AMD 2012] AMD. *GRAPHICS CORES NEXT (GCN) ARCHITECTURE*, June 2012. (Cited on page 47.)

[AnandTech 2011] AnandTech. *Apple iPad 2 Preview*, 2011. Online; accessed 24-February-2012; available at http://www.anandtech.com/show/4215/apple-ipad-2-benchmarked-dualcore-cortex-a9-powervr-sgx-543mp2/2. (Cited on pages 4 and 260.)

[Ancourt *et al.* 1993] Corinne Ancourt, Fabien Coelho, François Irigoin and Ronan Keryell. *A linear algebra framework for static hpf code distribution.* In Fourth International Workshop on Compilers for Parallel Computers, 1993. (Cited on page 174.)

[Ancourt *et al.* 1997] Corinne Ancourt, Fabien Coelho, François Irigoin and Ronan Keryell. *A Linear Algebra Framework for Static High Performance Fortran Code Distribution.* vol. 6, no. 1, pages 3–27, 1997. (Cited on page 174.)

[Andonov & Rajopadhye 1994] Rumen Andonov and Sanjay V. Rajopadhye. *Optimal Tile Sizing.* In Proceedings of the Third Joint International Conference on Vector and

Parallel Processing: Parallel Processing, CONPAR 94 - VAPP VI, pages 701–712, London, UK, UK, 1994. Springer-Verlag. (Cited on page 170.)

[Asanović *et al.* 2006] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Rapport technique UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on page 141.)

[Aubert 2010] Dominique Aubert. *Numerical Cosmology powered by GPUs*. Proceedings of the International Astronomical Union, vol. 6, no. Symposium S270, pages 397–400, 2010. (Cited on page 174.)

[Augonnet *et al.* 2010a] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault and Raymond Namyst. *Data-Aware Task Scheduling on Multi-accelerator Based Platforms*. In Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems, ICPADS '10, pages 291–298, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 167, 173 and 311.)

[Augonnet *et al.* 2010b] Cédric Augonnet, Samuel Thibault and Raymond Namyst. *StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*. Rapport de recherche RR-7240, INRIA, March 2010. (Cited on pages 158, 166, 308 and 311.)

[Augonnet *et al.* 2011] Cédric Augonnet, Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier. *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, vol. 23, pages 187–198, February 2011. (Cited on pages 166, 199 and 311.)

[Augonnet 2011] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2011. (Cited on pages 166 and 311.)

[Ayguadé *et al.* 1999] Eduard Ayguadé, Marc González, Jesús Labarta, Xavier Martorell, Nacho Navarro and José Oliver. *NanosCompiler: A Research Platform for OpenMP Extensions*. In In First European Workshop on OpenMP, pages 27–31, 1999. (Cited on pages 140 and 301.)

[Bachir *et al.* 2008] Mounira Bachir, Sid-Ahmed-Ali Touati and Albert Cohen. *Post-pass periodic register allocation to minimise loop unrolling degree*. In Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '08, pages 141–150, New York, NY, USA, 2008. ACM. (Cited on pages 132 and 297.)

[Baghdadi *et al.* 2010] Soufiane Baghdadi, Armin Größlinger and Albert Cohen. *Putting Automatic Polyhedral Compilation for GPGPU to Work*. In Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10), Vienna, Autriche, July 2010. (Cited on pages xx, 98 and 289.)

[Baghsorkhi *et al.* 2010] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp and Wen-mei W. Hwu. *An adaptive performance modeling tool for GPU architectures*. SIGPLAN Not., vol. 45, no. 5, pages 105–114, January 2010. (Cited on pages 157 and 307.)

[Bakhoda *et al.* 2009a] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt. *Analyzing CUDA workloads using a detailed GPU simulator*. In ISPASS, pages 163–174. IEEE, 2009. (Cited on pages 158 and 308.)

[Bakhoda *et al.* 2009b] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt. *Analyzing CUDA Workloads Using a Detailed GPU Simulator*. In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), pages 163–174, April 2009. (Cited on pages 158 and 308.)

[Bao & Xiang 2012] Bin Bao and Xiaoya Xiang. *Defensive loop tiling for multi-core processor*. In Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC '12, pages 76–77, New York, NY, USA, 2012. ACM. (Cited on page 170.)

[Baskaran *et al.* 2010] Muthu Baskaran, J. Ramanujam and P. Sadayappan. *Automatic C-to-CUDA Code Generation for Affine Programs*. In Rajiv Gupta, editor, Compiler Construction, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010. (Cited on pages 30, 69, 98, 271 and 289.)

[Bell & Hoberock 2011] Nathan Bell and Jared Hoberock. *Thrust: A Productivity-Oriented Library for CUDA*. In Wen mei W. Hwu, editor, GPU Computing Gems, October 2011. (Cited on page 118.)

[Benkner *et al.* 2011] Siegfried Benkner, Sabri Pllana, Jesper Larsson Traff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph Kessler, David

Moloney and Vitaly Osipov. *PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems*. IEEE Micro, vol. 31, pages 28–41, September 2011. (Cited on pages 5 and 261.)

[Bernstein 1966] Arthur Bernstein. *Analysis of programs for parallel processing*. IEEE Transactions on Electronic Computers, vol. 15, no. 5, pages 757–763, 1966. (Cited on page 103.)

[Bodin & Bihan 2009] François Bodin and Stéphane Bihan. *Heterogeneous multicore parallel programming for graphics processing units*. Sci. Program., vol. 17, pages 325–336, December 2009. (Cited on pages 25, 87, 138 and 300.)

[Bohn 1998] Christian-A. Bohn. *Kohonen Feature Mapping through Graphics Hardware*. In In Proceedings of Int. Conf. on Compu. Intelligence and Neurosciences, pages 64–67, 1998. (Cited on page 13.)

[Bondhugula *et al.* 2008a] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev and P. Sadayappan. *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*. In Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 116.)

[Bondhugula *et al.* 2008b] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev and P. Sadayappan. *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*. In Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on pages 155 and 306.)

[Bondhugula *et al.* 2008c] Uday Bondhugula, Albert Hartono, J. Ramanujam and P. Sadayappan. *A practical automatic polyhedral parallelizer and locality optimizer*. SIGPLAN Not., vol. 43, pages 101–113, June 2008. (Cited on pages 30, 116 and 271.)

[Bondhugula *et al.* 2010] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash and Lakshminarayanan Renganarayanan. *A model for fusion and code motion in an automatic parallelizing compiler*. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM. (Cited on pages 112, 115, 116 and 293.)

[Bonnot *et al.* 2008] Philippe Bonnot, Fabrice Lemonnier, Gilbert Édelin, Gérard Gaillat, Olivier Ruch and Pascal Gauget. *Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA*. In Design Automation and Test in Europe, DATE, pages 610–615. IEEE Computer Society Press, 2008. (Cited on page 146.)

[Boulet *et al.* 1998] Pierre Boulet, Alain Darte, Georges-André Silber and Frédéric Vivien. *Loop parallelization algorithms: from parallelism extraction to code generation*. Parallel Comput., vol. 24, pages 421–444, May 1998. (Cited on pages xxvii and 101.)

[Bozkus *et al.* 1994] Zeki Bozkus, Alok Choudhary, Geoffrey Fox and Tomasz Haupt. *Compiling FORTRAN 90D/HPF for distributed memory MIMD computers*. Journal of Parallel and Distributed Computing, vol. 21, no. 1, pages 15–26, April 1994. (Cited on pages 140 and 301.)

[Buck & Purcell 2004] I. Buck and T. Purcell. *A toolkit for computation on GPUs*. In GPU Gems, page 621–636, 2004. (Cited on page 111.)

[Buck *et al.* 2004] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston and Pat Hanrahan. *Brook for GPUs: stream computing on graphics hardware*. ACM Trans. Graph., vol. 23, no. 3, pages 777–786, 2004. (Cited on pages xvii, 13, 18, 266 and 268.)

[Buck 2003] Ian Buck. *Brook Spec v0.2*, October 2003. `http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf`. (Cited on pages 18 and 268.)

[Buck 2009] Ian Buck. *Nvidia's Ian Buck Talks GPGPU*, September 2009. `http://www.tomshardware.com/reviews/ian-buck-nvidia,2393.html`. (Cited on pages 19 and 269.)

[Burstall & Darlington 1977] R. M. Burstall and John Darlington. *A Transformation System for Developing Recursive Programs*. J. ACM, vol. 24, pages 44–67, January 1977. (Cited on pages 112 and 293.)

[Bush & Newman 1999] Jonathan Bush and Timothy S. Newman. *Effectively Utilizing 3DNow! in Linux*. Linux Journal, 1999. Online, available at `http://www.linuxjournal.com/article/3685`. (Cited on page 3.)

[Callahan 1987] C. D. Callahan II. *A global approach to detection of parallelism*. PhD thesis, Rice University, Houston, TX, USA, 1987. UMI Order No. GAX87-18697. (Cited on page 115.)

[Capannini 2011] Gabriele Capannini. *Designing Efficient Parallel Prefix Sum Algorithms for GPUs*. In Proceedings of the 2011 IEEE 11th International Conference on Computer and Information Technology, CIT '11, pages 189–196, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 111.)

[CAPS Entreprise 2010] CAPS Entreprise. *HMPP Workbench*. http://www.caps-entreprise.com/, 2010. (Cited on pages 62 and 280.)

[Carr *et al.* 2002] N. Carr, J. Hall and J. Hart. *The ray engine*, 2002. (Cited on page 13.)

[Carribault & Cohen 2004] Patrick Carribault and Albert Cohen. *Applications of storage mapping optimization to register promotion*. In Proceedings of the 18th annual international conference on Supercomputing, ICS '04, pages 247–256, New York, NY, USA, 2004. ACM. (Cited on pages 128 and 294.)

[Catanzaro *et al.* 2008] Bryan Catanzaro, Narayanan Sundaram and Kurt Keutzer. *A map reduce framework for programming graphics processors*. In In Workshop on Software Tools for MultiCore Systems, 2008. (Cited on page 118.)

[Ceng *et al.* 2008] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki and H. Kunieda. *MAPS: an integrated framework for MPSoC application parallelization*. In Proceedings of the 45th annual Design Automation Conference, DAC '08, pages 754–759, New York, NY, USA, 2008. ACM. (Cited on page 173.)

[Chamberlain *et al.* 2007] B.L. Chamberlain, D. Callahan and H.P. Zima. *Parallel Programmability and the Chapel Language*. International Journal of High Performance Computing Applications, vol. 21, no. 3, pages 291–312, 2007. (Cited on pages 7 and 262.)

[Che *et al.* 2009] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee and Kevin Skadron. *Rodinia: A benchmark suite for heterogeneous computing*. In IEEE International Symposium on Workload Characterization, 2009. (Cited on page 78.)

[Chen *et al.* 2010] Yifeng Chen, Xiang Cui and Hong Mei. *Large-scale FFT on GPU clusters*. In 24th ACM International Conference on Supercomputing, ICS '10, pages 315–324, New York, NY, USA, 2010. ACM. (Cited on pages 62 and 280.)

[Chen *et al.* 2011] Long Chen, Oreste Villa and Guang R. Gao. *Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems*. In Proceedings of the 2011 IEEE

International Conference on Cluster Computing, CLUSTER '11, pages 386–394, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 174.)

[Coarfa *et al.* 2005] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao and Daniel Chavarría-Miranda. *An evaluation of global address space languages: co-array fortran and unified parallel C*. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, 2005. (Cited on pages 6 and 261.)

[Coelho & Ancourt 1996] Fabien Coelho and Corinne Ancourt. *Optimal Compilation of HPF Remappings*. Journal of Parallel and Distributed Computing, vol. 38, no. 2, pages 229–236, November 1996. Also TR EMP CRI A-277 (October 1995); Extented abstract, TR A/276. (Cited on pages 77 and 284.)

[Coelho 1996] Fabien Coelho. *Contributions to High Performance Fortran Compilation*. PhD thesis, École des mines de Paris, 1996. (Cited on pages 77 and 284.)

[Collange *et al.* 2010] Sylvain Collange, Marc Daumas, David Defour and David Parello. *Barra: A Parallel Functional Simulator for GPGPU*. In Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '10, pages 351–360, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 158 and 308.)

[Collange 2010a] Silvain Collange. *Enjeux de conception des architectures GPGPU : unités arithmétiques spécialisées et exploitation de la régularité*. PhD thesis, Université de Perpignan, 2010. (Cited on pages 42, 55 and 278.)

[Collange 2010b] Sylvain Collange. *Analyse de l'architecture GPU Tesla*. Rapport technique, Université de Perpignan, January 2010. (Cited on pages 42, 54 and 278.)

[Consortium 2011] OpenHMPP Consortium. *OpenHMPP Concepts and Directives*, June 2011. (Cited on page 25.)

[Cordes *et al.* 2010] Daniel Cordes, Peter Marwedel and Arindam Mallik. *Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming*. In Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10, pages 267–276, New York, NY, USA, 2010. ACM. (Cited on page 173.)

[Creusillet & Irigoin 1996a] Béatrice Creusillet and François Irigoin. *Exact vs. Approximate Array Region Analyses*. In Languages and Compilers for Parallel Computing,

numéro 1239 de Lecture Notes in Computer Science, pages 86–100. Springer-Verlag, August 1996. (Cited on page 67.)

[Creusillet & Irigoin 1996b] Béatrice Creusillet and François Irigoin. *Interprocedural array region analyses*. Int. J. Parallel Program., vol. 24, no. 6, pages 513–546, 1996. (Cited on pages 65, 67, 73, 79, 102, 103, 114, 124, 139, 291, 293 and 300.)

[Creusillet 1996] Béatrice Creusillet. *Array Region Analyses and Applications*. PhD thesis, MINES ParisTech, 1996. (Cited on pages 65 and 104.)

[Creusillet 2011] Béatrice Creusillet. *Automatic Task Generation on the SCMP architecture for data flow applications*. Rapport technique, HPC Project, 2011. (Cited on page 148.)

[Cytron *et al.* 1989] R. Cytron, M. Hind and W. Hsieh. *Automatic generation of DAG parallelism*. In Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, PLDI '89, pages 54–68, New York, NY, USA, 1989. ACM. (Cited on page 173.)

[Dally *et al.* 2003] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju and I. Buck. *Merrimac: Supercomputing with streams*. In SC'03, Phoenix, Arizona, November 2003. (Cited on page 17.)

[Dantzig *et al.* 1954] G Dantzig, R Fulkerson and S Johnson. *Solution of a large-scale traveling-salesman problem*. Operations Research, vol. 2, pages 393–410, 1954. (Cited on page 116.)

[Darte & Huard 2000] Alain Darte and Guillaume Huard. *Loop Shifting for Loop Compaction*. Int. J. Parallel Program., vol. 28, pages 499–534, October 2000. (Cited on page 115.)

[Darte & Huard 2002] Alain Darte and Guillaume Huard. *New Results on Array Contraction*. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, ASAP '02, pages 359–, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 128 and 294.)

[Darte & Vivien 1996a] A. Darte and F. Vivien. *Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs*. In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96, pages 281–, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on page 101.)

[Darte & Vivien 1996b] Alain Darte and Frédéric Vivien. *On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops*. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte and Yves Robert, editors, Euro-Par'96 Parallel Processing, volume 1123 of *Lecture Notes in Computer Science*, pages 379–388. Springer Berlin / Heidelberg, 1996. (Cited on pages 102 and 290.)

[Darte & Vivien 1997] Alain Darte and Frédéric Vivien. *Parallelizing Nested Loops with Approximations of Distance Vectors: A Survey*. Parallel Processing Letters, vol. 7, no. 2, pages 133–144, 1997. (Cited on page 101.)

[Darte *et al.* 1996] Alain Darte, Georges-André Silber and Frédéric Vivien. *Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling*. Parallel Processing Letters, pages 379–392, 1996. (Cited on page 115.)

[Darte 2000] Alain Darte. *On the complexity of loop fusion*. Parallel Computing, vol. 26, no. 9, pages 1175 – 1193, 2000. (Cited on pages 112, 116 and 293.)

[Dastgeer *et al.* 2011] Usman Dastgeer, Johan Enmyren and Christoph W. Kessler. *Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems*. In Proceedings of the 4th International Workshop on Multicore Software Engineering, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM. (Cited on page 174.)

[Dean & Ghemawat 2004] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 173.)

[Deitz 2012] Steve Deitz. *C++ AMP for the DirectCompute Programmer*, April 2012. Online; accessed 29-July-2012; available at http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/09/c-amp-for-the-directcompute-programmer.aspx. (Cited on pages xvii and 22.)

[Derrien *et al.* 2012] Steven Derrien, Daniel Ménard, Kevin Martin, Antoine Floch, Antoine Morvan, Adeel Pasha, Patrice Quinton, Amit Kumar and Loïc Cloatre. *GeCoS: Generic Compiler Suite*. http://gecos.gforge.inria.fr, 2012. (Cited on pages 140 and 301.)

[Diamos *et al.* 2010] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili and Nathan Clark. *Ocelot: a dynamic optimization framework for bulk-*

synchronous applications in heterogeneous systems. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM. (Cited on pages 158 and 308.)

[Dollinger & Loechner 2011] Jean-François Dollinger and Vincent Loechner. *Sélection adaptative de codes polyédriques pour GPU/CPU*. In Quatrièmes rencontres de la communauté française de compilation, November 2011. (Cited on pages 158 and 308.)

[Donadio *et al.* 2006] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Garzarán, David Padua and Keshav Pingali. *A Language for the Compact Representation of Multiple Program Versions*. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam and P. Sadayappan, editors, Languages and Compilers for Parallel Computing, volume 4339 of *Lecture Notes in Computer Science*, pages 136–151. Springer Berlin / Heidelberg, 2006. (Cited on pages 150 and 304.)

[Dongarra & Schreiber 1990] Jack Dongarra and Robert Schreiber. *Automatic Blocking of Nested Loops*. Rapport technique, Knoxville, TN, USA, 1990. (Cited on page 170.)

[Dubash 2005] Manek Dubash. *Moore's Law is dead, says Gordon Moore*, April 2005. Online; accessed 29-July-2012; available at `http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/`. (Cited on pages 2 and 258.)

[Ebcioğlu *et al.* 2004] Kemal Ebcioğlu, Vijay Saraswat and Vivek Sarkar. *X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access*. In Proceedings of the International Workshop on Language Runtimes, OOPSLA, 2004. (Cited on pages 7 and 262.)

[Elteir *et al.* 2011] Marwa Elteir, Heshan Lin and Wu-Chun Feng. *Performance Characterization and Optimization of Atomic Operations on AMD GPUs*. In Proceedings of the 2011 IEEE International Conference on Cluster Computing, CLUSTER '11, pages 234–243, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 42.)

[England 1978] J. N. England. *A system for interactive modeling of physical curved surface objects*. SIGGRAPH Comput. Graph., vol. 12, no. 3, pages 336–340, 1978. (Cited on pages 12 and 266.)

[Enmyren & Kessler 2010] Johan Enmyren and Christoph W. Kessler. *SkePU: a multi-backend skeleton programming library for multi-GPU systems*. In Proceedings of the fourth international workshop on High-level parallel programming and applications, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM. (Cited on pages 62, 174 and 281.)

[Eyles *et al.* 1997] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England and Lee Westover. *PixelFlow: the realization*. In HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pages 57–68, New York, NY, USA, 1997. ACM. (Cited on page 13.)

[Feautrier 1991] Paul Feautrier. *Dataflow analysis of array and scalar references*. International Journal of Parallel Programming, vol. 20, pages 23–53, 1991. 10.1007/BF01407931. (Cited on pages 156 and 306.)

[Feautrier 1992] Paul Feautrier. *Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time*. International Journal of Parallel Programming, vol. 21, pages 389–420, 1992. (Cited on page 101.)

[Feng & Cameron 2007] Wu-chun Feng and Kirk Cameron. *The Green500 List: Encouraging Sustainable Supercomputing*. Computer, vol. 40, pages 50–55, December 2007. (Cited on pages 5, 260 and 261.)

[Ferrante *et al.* 1987] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. *The program dependence graph and its use in optimization*. ACM Trans. Program. Lang. Syst., vol. 9, pages 319–349, July 1987. (Cited on pages 118 and 121.)

[Fisher & Ghuloum 1994] Allan L. Fisher and Anwar M. Ghuloum. *Parallelizing complex scans and reductions*. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, PLDI '94, pages 135–146, New York, NY, USA, 1994. ACM. (Cited on page 111.)

[Frigo & Johnson 2005] Matteo Frigo and Steven G. Johnson. *The Design and Implementation of FFTW3*. Proceedings of the IEEE, vol. 93, no. 2, pages 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". (Cited on pages 153 and 306.)

[Frigo *et al.* 1998] Matteo Frigo, Charles E. Leiserson and Keith H. Randall. *The Implementation of the Cilk-5 Multithreaded Language*. In Proceedings of the SIGPLAN Conference on Program Language Design and Implementation, PLDI, pages 212–223, 1998. (Cited on pages 140 and 301.)

[Fursin & Cohen 2007] Grigori Fursin and Albert Cohen. *Building a practical iterative interactive compiler*. In In 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), 2007. (Cited on page 150.)

[Fursin *et al.* 2008] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Éric Courtois and François Bodin. *MILEPOST GCC: machine learning based research compiler*. In GCC Summit, Ottawa, Canada, 2008. MILEPOST project (http://www.milepost.eu). (Cited on page 150.)

[Gao *et al.* 1993] Guang R. Gao, R. Olsen, Vivek Sarkar and Radhika Thekkath. *Collective Loop Fusion for Array Contraction*. In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, pages 281–295, London, UK, 1993. Springer-Verlag. (Cited on pages 113, 116, 118, 128 and 294.)

[Gerndt & Zima 1990] Hans Michael Gerndt and Hans Peter Zima. *Optimizing communication in SUPERB*. In Proceedings of the joint international conference on Vector and parallel processing, CONPAR 90-VAPP IV, 1990. (Cited on pages 77 and 284.)

[Girkar & Polychronopoulos 1994] Milind Girkar and Constantine D. Polychronopoulos. *The hierarchical task graph as a universal intermediate representation*. Int. J. Parallel Program., vol. 22, no. 5, pages 519–551, October 1994. (Cited on page 173.)

[Girkar & Polychronopoulos 1995] Milind Girkar and Constantine D. Polychronopoulos. *Extracting task-level parallelism*. ACM Trans. Program. Lang. Syst., vol. 17, no. 4, pages 600–634, July 1995. (Cited on page 173.)

[Goldberg & Paige 1984] Allen Goldberg and Robert Paige. *Stream processing*. In Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP '84, pages 53–62, New York, NY, USA, 1984. ACM. (Cited on pages 112 and 293.)

[Goldberg 1989] David E. Goldberg. Genetic algorithms in search, optimization and machine learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. (Cited on pages 141 and 302.)

[Gong *et al.* 1993] Chun Gong, Rajiv Gupta and Rami Melhem. *Compilation Techniques for Optimizing Communication on Distributed-Memory Systems*. In ICPP '93, 1993. (Cited on pages 77 and 284.)

[Gordon *et al.* 2002] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze and Saman Amarasinghe. *A stream compiler for communication-exposed architectures*. SIGOPS Oper. Syst. Rev., vol. 36, no. 5, pages 291–303, October 2002. (Cited on page 174.)

[Goubier *et al.* 2011] Thierry Goubier, Renaud Sirdey, Stéphane Louise and Vincent David. *ΣC: a programming model and language for embedded manycores*. In Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I, ICA3PP'11, pages 385–394, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 23, 215 and 317.)

[Guelton 2011a] Serge Guelton. *Building Source-to-Source Compilers for Heterogeneous Targets*. PhD thesis, Université de Bretagne Ouest, Brest, France, 2011. (Cited on pages xix, 56, 57, 62, 142, 143, 144, 146, 147, 170, 232, 279, 280, 302, 303 and 312.)

[Guelton 2011b] Serge Guelton. *Transformations for Memory Size and Distribution*. In Université de Bretagne Ouest [Guelton 2011a], chapitre 6. (Cited on pages 70, 88 and 283.)

[Halfhill 2009] Analyst Tom R Halfhill. *White Paper, Looking Beyond Graphics*, 2009. (Cited on page 42.)

[Hall *et al.* 1995] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao and Monica S. Lam. *Detecting coarse-grain parallelism using an interprocedural parallelizing compiler*. In Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '95, New York, NY, USA, 1995. ACM. (Cited on page 173.)

[Hall *et al.* 2010] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy and Malik Khan. *Loop Transformation Recipes for Code Generation and Auto-Tuning*. In Guang Gao, Lori Pollock, John Cavazos and Xiaoming Li, editors, Languages and Compilers for Parallel Computing, volume 5898 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin / Heidelberg, 2010. (Cited on page 150.)

[Han & Abdelrahman 2009] Tianyi David Han and Tarek S. Abdelrahman. *hiCUDA: a high-level directive-based language for GPU programming*. In Proceedings of GPGPU-2. ACM, 2009. (Cited on pages xvii, 25, 26, 69, 87, 139 and 287.)

[Harris *et al.* 2002] Mark J. Harris, Greg Coombe, Thorsten Scheuermann and Anselmo Lastra. *Physically-based visual simulation on graphics hardware*. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '02, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. (Cited on pages 12 and 266.)

[Harris *et al.* 2007] Mark Harris, Shubhabrata Sengupta and John D. Owens. *Parallel Prefix Sum (Scan) with CUDA*. In Hubert Nguyen, editor, GPU Gems 3. Addison Wesley, August 2007. (Cited on page 111.)

[Hartono *et al.* 2009] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam and P. Sadayappan. *Parametric multi-level tiling of imperfectly nested loops*. In Proceedings of the 23rd international conference on Supercomputing, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM. (Cited on page 170.)

[Hermann *et al.* 2010] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier and Jérémie Allard. *Multi-GPU and multi-CPU parallelization for interactive physics simulations*. In Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II, Euro-Par'10, pages 235–246, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 174.)

[Hoberock & Bell 2012] Jared Hoberock and Nathan Bell. *Thrust Quick Start Guide*, 2012. Online; accessed 24-February-2012; available at `https://github.com/thrust/thrust/wiki/Quick-Start-Guide`. (Cited on pages xxi and 119.)

[Hoff *et al.* 1999] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha and Tim Culver. *Fast computation of generalized Voronoi diagrams using graphics hardware*. In Proceedings of the 26th annual conference on Computer graphics and interactive techniques, SIGGRAPH '99, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. (Cited on page 13.)

[Hofstee 2005] H. Peter Hofstee. *Power Efficient Processor Architecture and The Cell Processor*. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA '05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on pages 12 and 265.)

[Hong & Kim 2009] Sunpyo Hong and Hyesoon Kim. *An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness*. In Proceedings of the 36th annual international symposium on Computer architecture, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM. (Cited on pages 157 and 307.)

[Hong *et al.* 2010] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng and Haibo Lin. *MapCG: writing parallel program portable between CPU and GPU*. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM. (Cited on page 42.)

[Hsu & Feng 2005] Chung-hsing Hsu and Wu-chun Feng. *A Power-Aware Run-Time System for High-Performance Computing*. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on pages 5 and 261.)

[Huang *et al.* 2006] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron and Mircea R. Stan. *Hotspot: acompact thermal modeling methodology for early-stage VLSI design*. IEEE Trans. Very Large Scale Integr. Syst., May 2006. (Cited on page 197.)

[Huynh *et al.* 2012] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong and Rick Siow Mong Goh. *Scalable framework for mapping streaming applications onto multi-GPU systems*. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 1–10, New York, NY, USA, 2012. ACM. (Cited on page 174.)

[ik Lee *et al.* 2003] Sang ik Lee, Troy A. Johnson and Rudolf Eigenmann. *Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation*. In 16th International Workshop on Languages and Compilers for Parallel Computing, volume 2958 of *LCPC*, pages 539–553, College Station, TX, USA, 2003. (Cited on pages 140 and 301.)

[Intel 2008] Intel. *Intel Museum*, 2008. Online, archived at `http://web.archive.org/web/20080406154333/http://www.intel.com/museum/online/hist_micro/hof/`. (Cited on page 3.)

[Irigoin & Triolet 1988] François Irigoin and Rémi Triolet. *Supernode partitioning*. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM. (Cited on page 170.)

[Irigoin *et al.* 1991] François Irigoin, Pierre Jouvelot and Rémi Triolet. *Semantical interprocedural parallelization: an overview of the PIPS project*. In Proceedings of the 5th international conference on Supercomputing, ICS '91, pages 244–251, New

York, NY, USA, 1991. ACM. (Cited on pages 62, 78, 102, 121, 126, 139, 140, 281, 300 and 301.)

[Irigoin *et al.* 2011] François Irigoin, Fabien Coelho and Béatrice Creusillet. *Dependencies between Analyses and Transformations in the Middle-End of a Compiler*. In Analyse to Compile, Compile to Analyse Workshop (ACCA), in conjunction with CGO 2011, April 2011. (Cited on pages 111 and 293.)

[ISO 2010] ISO. Programming languages – Fortran – Part 1: Base language. International Organization for Standardization, June 2010. (Cited on pages 6 and 261.)

[Jablin *et al.* 2011] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard and David I. August. *Automatic CPU-GPU communication management and optimization*. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM. (Cited on pages 87, 179 and 287.)

[Ji & Ma 2011] Feng Ji and Xiaosong Ma. *Using Shared Memory to Accelerate MapReduce on Graphics Processing Units*. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 805–816, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 42.)

[Jouvelot & Dehbonei 1989] Pierre Jouvelot and Babak Dehbonei. *A unified semantic approach for the vectorization and parallelization of generalized reductions*. In Proceedings of the 3rd international conference on Supercomputing, ICS '89, pages 186–194, New York, NY, USA, 1989. ACM. (Cited on pages 105 and 291.)

[Kahn 1974] Gilles Kahn. *The Semantics of Simple Language for Parallel Programming*. In IFIP Congress, pages 471–475, 1974. (Cited on page 23.)

[Kalray 2012] Kalray. *MPPA : Multi-Purpose Processor Array*, 2012. Online; accessed 16-June-2012; available at `http://www.kalray.eu/en/products/mppa.html`. (Cited on page 23.)

[Karypis & Kumar 1998] George Karypis and Vipin Kumar. *Multilevel k-way partitioning scheme for irregular graphs*. J. Parallel Distrib. Comput., vol. 48, no. 1, pages 96–129, January 1998. (Cited on page 174.)

[Kedem & Ishihara 1999] Gershon Kedem and Yuriko Ishihara. *Brute force attack on UNIX passwords with SIMD computer*. In SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association. (Cited on page 13.)

[Kennedy & Mckinley 1993] Ken Kennedy and Kathryn S. Mckinley. *Typed fusion with applications to parallel and sequential code generation.* Rapport technique, Center for Research on Parallel Computation, Rice University, 1993. (Cited on pages 115 and 116.)

[Kennedy & McKinley 1994] Ken Kennedy and Kathryn S. McKinley. *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution.* In Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing, pages 301–320, London, UK, 1994. Springer-Verlag. (Cited on pages 114, 115, 116, 121, 126 and 293.)

[Kennedy 2001] Ken Kennedy. *Fast Greedy Weighted Fusion.* Int. J. Parallel Program., vol. 29, pages 463–491, October 2001. (Cited on page 116.)

[Khronos OpenCL Working Group 2008] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0*, 8 December 2008. (Cited on pages 7, 21, 262 and 270.)

[Khronos OpenCL Working Group 2011] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, 15 November 2011. (Cited on pages xviii, 7, 30, 34 and 262.)

[Khronos OpenGL Working Group 1994] Khronos OpenGL Working Group. *The OpenGL Graphics System: A Specification (Version 4.2)*, July 1994. (Cited on pages 12 and 265.)

[Khronos OpenGL Working Group 2004] Khronos OpenGL Working Group. *The OpenGL Graphics System: A Specification (Version 2.0)*, October 2004. (Cited on pages 15 and 268.)

[Khronos OpenGL Working Group 2012] Khronos OpenGL Working Group. *The OpenGL Graphics System: A Specification (Version 4.2)*, April 2012. (Cited on pages 12 and 265.)

[Kilgard 2012] Mark Kilgard. *NVIDIA OpenGL in 2012: Version 4.3 is here!* In GPU Technology Conference, 2012. (Cited on pages xvii and 16.)

[Kim & Rajopadhye 2010] DaeGon Kim and Sanjay Rajopadhye. *Efficient tiled loop generation: D-tiling.* In Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing, LCPC'09, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 170.)

[Komatsu *et al.* 2010] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa and Hiroaki Kobayashi. *Evaluating Performance and Porta-*

bility of OpenCL Programs. In The Fifth International Workshop on Automatic Performance Tuning, June 2010. (Cited on pages 7 and 262.)

[Kotha *et al.* 2010] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy and Rajeev Barua. *Automatic Parallelization in a Binary Rewriter*. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 62.)

[Kothapalli *et al.* 2009] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan and Kannan Srinathan. *A performance prediction model for the CUDA GPGPU platform*. In Proceedings of 2009 International Conference on High Performance Computing (HiPC), pages 463–472, December 2009. (Cited on pages 157 and 307.)

[Kuck *et al.* 1981] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe. *Dependence graphs and compiler optimizations*. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM. (Cited on pages 112, 115 and 293.)

[Kulkarni *et al.* 2003] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek and Kyle Gallivan. *Finding effective optimization phase sequences*. In Conference on Language, Compiler, and Tool for Embedded Systems, pages 12–23. ACM Press, 2003. (Cited on pages 141 and 302.)

[Kusano & Sato 1999] Kazuhiro Kusano and Mitsuhisa Sato. *A Comparison of Automatic Parallelizing Compiler and Improvements by Compiler Directives*. In International Symposium on High Performance Computing, ISHPC '99, 1999. (Cited on pages 150 and 304.)

[Ladner & Fischer 1980] Richard E. Ladner and Michael J. Fischer. *Parallel Prefix Computation*. J. ACM, vol. 27, no. 4, pages 831–838, October 1980. (Cited on page 111.)

[Lamport 1974] Leslie Lamport. *The parallel execution of DO loops*. Communications of the ACM, vol. 17, pages 83–93, February 1974. (Cited on pages 101 and 290.)

[Lattner & Adve 2004] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In International Symposium

on Code Generation and Optimization, CGO, Palo Alto, California, 2004. (Cited on pages 149 and 304.)

[Lee & Eigenmann 2010] Seyong Lee and Rudolf Eigenmann. *OpenMPC: Extended OpenMP Programming and Tuning for GPUs*. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11. IEEE Computer Society, 2010. (Cited on pages 87, 139 and 287.)

[Lee *et al.* 1991] Roland L. Lee, Alex Y. Kwok and Fayé A. Briggs. *The floating point performance of a superscalar SPARC processor*. SIGARCH Comput. Archit. News, vol. 19, no. 2, pages 28–37, April 1991. (Cited on page 132.)

[Lee *et al.* 2009] Seyong Lee, Seung-Jai Min and Rudolf Eigenmann. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization*. In Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 101–110, New York, NY, USA, 2009. ACM. (Cited on pages 23, 87, 98, 139, 286 and 289.)

[Lengyel *et al.* 1990] Jed Lengyel, Mark Reichert, Bruce R. Donald and Donald P. Greenberg. *Real-time robot motion planning using rasterizing computer graphics hardware*. In Proceedings of the 17th annual conference on Computer graphics and interactive techniques, SIGGRAPH '90, pages 327–335, New York, NY, USA, 1990. ACM. (Cited on page 13.)

[Leung *et al.* 2009] Alan Leung, Ondřej Lhoták and Ghulam Lashari. *Automatic parallelization for graphics processing units*. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, pages 91–100, New York, NY, USA, 2009. ACM. (Cited on pages 30, 68, 271 and 283.)

[Leung *et al.* 2010] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul and Richard Lethin. *A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction*. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM. (Cited on page 174.)

[Leung 2008] Alan Chun-Wai Leung. Automatic Parallelization for Graphics Processing Units in JikesRVM. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2008. (Cited on pages 87, 157, 286 and 307.)

[Lim & Lam 1997] Amy W. Lim and Monica S. Lam. *Maximizing parallelism and minimizing synchronization with affine transforms.* In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM. (Cited on page 101.)

[Lin *et al.* 2011] Haibo Lin, Tao Liu, Lakshminarayanan Renganarayana, Huoding Li, Tong Chen, Kevin O'Brien and Ling Shao. *Automatic Loop Tiling for Direct Memory Access.* In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 479–489, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 170.)

[Lindholm *et al.* 2008] Erik Lindholm, John Nickolls, Stuart Oberman and John Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture.* IEEE Micro, vol. 28, no. 2, pages 39–55, March 2008. (Cited on pages 55 and 278.)

[Liu *et al.* 2007] Weiguo Liu, Wolfgang Muller-Wittig and Bertil Schmidt. *Performance Predictions for General-Purpose Computation on GPUs.* In Proceedings of the 2007 International Conference on Parallel Processing, ICPP '07, pages 50–, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on pages 157 and 307.)

[Manegold 2002] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance.* PhD thesis, Universiteit van Amsterdam, December 2002. (Cited on pages 2 and 258.)

[Manjikian & Abdelrahman 1997] Naraig Manjikian and Tarek S. Abdelrahman. *Fusion of Loops for Parallelism and Locality.* IEEE Trans. Parallel Distrib. Syst., vol. 8, pages 193–209, February 1997. (Cited on pages 114 and 115.)

[Matsuzaki *et al.* 2006] Kiminori Matsuzaki, Zhenjiang Hu and Masato Takeichi. *Towards automatic parallelization of tree reductions in dynamic programming.* In Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, SPAA '06, pages 39–48, New York, NY, USA, 2006. ACM. (Cited on page 111.)

[Megiddo & Sarkar 1997] Nimrod Megiddo and Vivek Sarkar. *Optimal weighted loop fusion for parallel programs.* In Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, SPAA '97, pages 282–291, New York, NY, USA, 1997. ACM. (Cited on pages 116 and 118.)

[Membarth *et al.* 2009] Richard Membarth, Philipp Kutzer, Hritam Dutta, Frank Hannig and Jürgen Teich. *Acceleration of Multiresolution Imaging Algorithms: A Compar-*

ative Study. In Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP '09, pages 211–214, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 117.)

[Membarth *et al.* 2012] Richard Membarth, Jan-Hugo Lupp, Frank Hannig, Jürgen Teich, Mario Körner and Wieland Eckert. *Dynamic task-scheduling and resource management for GPU accelerators in medical imaging*. In Proceedings of the 25th international conference on Architecture of Computing Systems, ARCS'12, pages 147–159, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 174.)

[Microsoft Corporation 2012a] Microsoft Corporation. *C++ AMP : Language and Programming Model*, May 2012. (Cited on pages 21 and 270.)

[Microsoft Corporation 2012b] Microsoft Corporation. *C++ AMP Overview*, 2012. Online; accessed 21-June-2012; available at `http://msdn.microsoft.com/en-us/library/hh265136%28v=vs.110%29.aspx`. (Cited on pages xvii and 24.)

[Microsoft 1995] Microsoft. *DirectX Developer Center*, September 1995. (Cited on pages 12 and 265.)

[Microsoft 2010] Microsoft. *Compute Shader Overview*, July 2010. Online; accessed 29-July-2012; available at `http://msdn.microsoft.com/en-us/library/ff476331%28v=VS.85%29.aspx`. (Cited on pages 21 and 270.)

[Microsoft 2012] Microsoft. *DirectX Developer Center*, 2012. (Cited on pages 12 and 265.)

[Millot *et al.* 2008] Daniel Millot, Alain Muller, Christian Parrot and Frédérique Silber-Chaussumier. *STEP: a distributed OpenMP for coarse-grain parallelism tool*. In Proceedings of the 4th international conference on OpenMP in a new era of parallelism, IWOMP'08, pages 83–99, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 174.)

[Moore 1965] Gordon E. Moore. *Cramming more components onto integrated circuits*. Electronics, vol. 38, no. 8, April 1965. (Cited on page 2.)

[Munk *et al.* 2010] Harm Munk, Eduard Ayguadé, Cédric Bastoul, Paul Carpenter, Zbigniew Chamski, Albert Cohen, Marco Cornero, Philippe Dumont, Marc Duranton, Mohammed Fellahi, Roger Ferrer, Razya Ladelsky, Menno Lindwer, Xavier Martorell, Cupertino Miranda, Dorit Nuzman, Andrea Ornstein, Antoniu Pop, Sebastian Pop, Louis-Noël Pouchet, Alex Ramírez, David Ródenas, Erven Rohou, Ira Rosen, Uzi Shvadron, Konrad Trifunović and Ayal Zaks. *Acotes project: Advanced*

compiler technologies for embedded streaming. International Journal of Parallel Programming, 2010. Special issue on European HiPEAC network of excellence members projects. To appear. (Cited on pages 140 and 301.)

[Necula *et al.* 2002] George C. Necula, Scott McPeak, Shree Prakash Rahul and Westley Weimer. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. In CC, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, April 2002. (Cited on page 150.)

[Nugteren *et al.* 2011] Cédric Nugteren, Henk Corporaal and Bart Mesman. *Skeleton-based automatic parallelization of image processing algorithms for GPUs*. In Luigi Carro and Andy D. Pimentel, editors, ICSAMOS, pages 25–32. IEEE, 2011. (Cited on pages 30 and 271.)

[NVIDIA, Cray, PGI, CAPS 2011] NVIDIA, Cray, PGI, CAPS. *The OpenACC Specification, version 1.0*, 3 November 2011. (Cited on pages 28, 62, 150, 280 and 304.)

[Nvidia 2012a] Nvidia. *CUDA C Best Practices Guide*, January 2012. (Cited on pages 159 and 162.)

[Nvidia 2012b] Nvidia. *Tegra 2 & Tegra 3 Super Chip Processors*, 2012. Online; accessed 24-February-2012; available at `http://www.nvidia.com/object/tegra-superchip.html`. (Cited on pages 4 and 260.)

[Ohshima *et al.* 2010] Satoshi Ohshima, Shoichi Hirasawa and Hiroki Honda. *OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler*. In Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, volume 6132 of *Lecture Notes in Computer Science*, pages 161–173. Springer Verlag, 2010. (Cited on pages 25, 87, 139 and 286.)

[Olmos *et al.* 2005] Karina Olmos, Karina Olmos, Eelco Visser and Eelco Visser. *Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules*. In International Conference on Compiler Construction, volume 3443 of *LNCS*, pages 204–220. Springer-Verlag, 2005. (Cited on page 150.)

[OpenACC Consortium 2012] OpenACC Consortium. *OpenACC Application Programming Interface*, 2012. (Cited on pages 7 and 262.)

[OpenMP Architecture Review Board 1997] OpenMP Architecture Review Board. *OpenMP Standard, 1.0*, October 1997. (Cited on page 7.)

[OpenMP Architecture Review Board 2011] OpenMP Architecture Review Board. *OpenMP Standard, 3.1*, July 2011. (Cited on pages 7 and 156.)

[Ottoni *et al.* 2005] Guilherme Ottoni, Ram Rangan, Adam Stoler and David I. August. *Automatic Thread Extraction with Decoupled Software Pipelining*. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 173.)

[Owens *et al.* 2007] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn and Timothy J. Purcell. *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, vol. 26, no. 1, pages 80–113, 2007. (Cited on page 13.)

[Pande lab Stanford University 2012] Pande lab Stanford University. *Folding@home distributed computing*, 2012. Online; accessed 24-February-2012; available at `http://folding.stanford.edu/`. (Cited on pages 13 and 267.)

[Patterson 2009] David Patterson. *The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges*, 2009. (Cited on page 42.)

[Potmesil & Hoffert 1989] Michael Potmesil and Eric M. Hoffert. *The pixel machine: a parallel image computer*. SIGGRAPH Comput. Graph., vol. 23, no. 3, pages 69–78, 1989. (Cited on pages 12 and 266.)

[Pouchet *et al.* 2010a] Louis-Noel Pouchet, Cédric Bastoul and Uday Bondhugula. *PoCC: the Polyhedral Compiler Collection*, 2010. `http://pocc.sf.net`. (Cited on pages 139, 155, 156, 300 and 306.)

[Pouchet *et al.* 2010b] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam and P. Sadayappan. *Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework*. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 116.)

[Pouchet 2011] Louis-Noël Pouchet. *The Polyhedral Benchmark suite 2.0*, March 2011. (Cited on pages 63, 78 and 281.)

[Pradelle *et al.* 2012] Benoît Pradelle, Alain Ketterlin and Philippe Clauss. *Polyhedral parallelization of binary code*. ACM Trans. Archit. Code Optim., vol. 8, no. 4, pages 39:1–39:21, January 2012. (Cited on page 62.)

[Project 2003] Stanford Streaming Supercomputer Project. *Merrimac*, 2003. Online; accessed 29-July-2012; available at http://merrimac.stanford.edu/. (Cited on page 17.)

[Purcell *et al.* 2002] Timothy J. Purcell, Ian Buck, William R. Mark and Pat Hanrahan. *Ray tracing on programmable graphics hardware*. ACM Trans. Graph., vol. 21, no. 3, pages 703–712, 2002. (Cited on page 13.)

[Quinlan 2000] Daniel J. Quinlan. *ROSE: Compiler Support for Object-Oriented Frameworks*. Parallel Processing Letters, vol. 10, no. 2/3, pages 215–226, 2000. (Cited on pages 140, 149, 301 and 304.)

[Ramey 2009] Will Ramey. *Languages, APIs and Development Tools for GPU Computing 2009*, September 2009. Online; accessed 31-July-2012; available at http://www.nvidia.com/content/GTC/documents/SC09_Languages_DevTools_Ramey.pdf. (Cited on pages 56 and 279.)

[Ravi *et al.* 2010] Vignesh T. Ravi, Wenjing Ma, David Chiu and Gagan Agrawal. *Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations*. In Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM. (Cited on page 111.)

[Redon & Feautrier 1993] Xavier Redon and Paul Feautrier. *Detection of Recurrences in Sequential Programs with Loops*. In Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe, PARLE '93, pages 132–145, London, UK, UK, 1993. Springer-Verlag. (Cited on page 111.)

[Renganarayanan *et al.* 2012] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout and Sanjay Rajopadhye. *Parameterized loop tiling*. ACM Trans. Program. Lang. Syst., vol. 34, no. 1, pages 3:1–3:41, May 2012. (Cited on page 170.)

[Reservoir Labs 2012] Reservoir Labs. *R-Stream - High Level Compiler*, 2012. Online; accessed 24-February-2012; available at https://www.reservoir.com/rstream. (Cited on page 30.)

[Rhoades *et al.* 1992] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann and Amitabh Varshney. *Real-time procedural textures*. In SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics, pages 95–100, New York, NY, USA, 1992. ACM. (Cited on pages 12 and 266.)

[Robison 2001] Arch D. Robison. *Impact of economics on compiler optimization*. In Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, JGI '01, pages 1–10, New York, NY, USA, 2001. ACM. (Cited on pages 213 and 316.)

[Rudy *et al.* 2011] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen and Jacqueline Cham. *A programming language interface to describe transformations and code generation*. In Proceedings of the 23rd international conference on Languages and compilers for parallel computing, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 30, 150 and 271.)

[Sarkar & Gao 1991] Vivek Sarkar and Guang R. Gao. *Optimization of array accesses by collective loop transformations*. In Proceedings of the 5th international conference on Supercomputing, ICS '91, pages 194–205, New York, NY, USA, 1991. ACM. (Cited on pages 113, 128 and 294.)

[Sarkar & Hennessy 1986] Vivek Sarkar and John Hennessy. *Compile-time partitioning and scheduling of parallel programs*. In Proceedings of the 1986 SIGPLAN symposium on Compiler construction, SIGPLAN '86, pages 17–26, New York, NY, USA, 1986. ACM. (Cited on page 173.)

[Sarkar 1991] V. Sarkar. *Automatic partitioning of a program dependence graph into parallel tasks*. IBM J. Res. Dev., vol. 35, no. 5-6, pages 779–804, September 1991. (Cited on page 173.)

[Schordan & Quinlan 2003] Markus Schordan and Dan Quinlan. *A Source-To-Source Architecture for User-Defined Optimizations*. In Modular Programming Languages, volume 2789 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. (Cited on pages 149 and 304.)

[Scilab Consortium 2003] Scilab Consortium. *Scilab*, 2003. `http://www.scilab.org/`. (Cited on pages 8 and 263.)

[Seiler *et al.* 2008] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan and Pat Hanrahan. *Larrabee: a many-core x86 architecture for visual computing*. ACM Trans. Graph., vol. 27, no. 3, pages 18:1–18:15, August 2008. (Cited on pages 12 and 265.)

[Sengupta *et al.* 2007] Shubhabrata Sengupta, Mark Harris, Yao Zhang and John D. Owens. *Scan primitives for GPU computing*. In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '07, pages

97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. (Cited on page 111.)

[Sohi & Vajapeyam 1989] Gurindar S. Sohi and Sriram Vajapeyam. *Tradeoffs in instruction format design for horizontal architectures*. SIGARCH Comput. Archit. News, vol. 17, no. 2, pages 15–25, April 1989. (Cited on page 132.)

[Srinivasan 2012] Krishnan Srinivasan. *How CPU Speed Increases Over the Years*, April 2012. Online; accessed 29-July-2012; available at http://www.ehow.com/facts_7677990_cpu-speed-increases-over-years.html. (Cited on pages 2 and 258.)

[Stock & Koch 2010] Florian Stock and Andreas Koch. *A fast GPU implementation for solving sparse ill-posed linear equation systems*. In Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I, PPAM'09, pages 457–466, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 117.)

[Stone *et al.* 2007] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco and Klaus Schulten. *Accelerating molecular modeling applications with graphics processors.* Journal of Computational Chemistry, vol. 28, no. 16, pages 2618–2640, 2007. (Cited on page 133.)

[Stuart *et al.* 2010] Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma and John D. Owens. *Multi-GPU volume rendering using MapReduce*. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 841–848, New York, NY, USA, 2010. ACM. (Cited on page 173.)

[Sutter 2005] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Dr. Dobb's Journal, vol. 30, no. 3, 2005. (Cited on pages xvii, 3, 4 and 259.)

[Sutter 2011] Herb Sutter. *Welcome to the Jungle*. Sutter's Mill, 12 2011. http://herbsutter.com/2011/12/29/welcome-to-the-jungle/. (Cited on pages xvii, 3, 5, 211, 260 and 314.)

[Taylor & Li 2010] Ryan Taylor and Xiaoming Li. *A Micro-benchmark Suite for AMD GPUs*. In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW '10, pages 387–396, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 54 and 277.)

[The Green500 2011] The Green500. *November 2011*, 2011. Online; available at http://www.green500.org/lists/2011/11/top/list.php. (Cited on pages 5 and 260.)

[TOP500 Supercomputing Sites 2012] TOP500 Supercomputing Sites. *November 2012*, 2012. Online; available at `http://top500.org/lists/2012/11`. (Cited on pages 5 and 260.)

[Trendall & Stewart 2000] C. Trendall and A. Stewart. *General calculations using graphics hardware with applications to interactive caustics*, 2000. (Cited on pages 12, 13, 55, 266 and 278.)

[Triolet *et al.* 1986] Rémi Triolet, François Irigoin and Paul Feautrier. *Direct parallelization of call statements*. In Proceedings of the 1986 SIGPLAN symposium on Compiler construction, SIGPLAN '86, pages 176–185, New York, NY, USA, 1986. ACM. (Cited on pages 64, 103 and 126.)

[Triolet 1984] Rémi Triolet. *Contribution à la Parallélisation Automatique de Programmes Fortran Comportant des Appels de Procédures*. PhD thesis, Université Paris VI, 1984. (Cited on page 64.)

[Ubal *et al.* 2007] R. Ubal, J. Sahuquillo, S. Petit and P. López. *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors*. In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, Oct. 2007. (Cited on pages 158 and 308.)

[UPC Consortium 2005] UPC Consortium. *UPC Language Specifications, v1.2*, 2005. (Cited on pages 6 and 261.)

[Ventroux & David 2010] Nicolas Ventroux and Raphaël David. *SCMP architecture: an asymmetric multiprocessor system-on-chip for dynamic applications*. In International Forum on Next-Generation Multicore/Manycore Technologies, IFMT, pages 6:1–6:12, New York, NY, USA, June 2010. ACM. (Cited on page 149.)

[Ventroux *et al.* 2012] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet and R. Keryell. *SESAM/ Par4All: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation*. In Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '12, pages 9–16, New York, NY, USA, 2012. ACM. (Cited on pages 62 and 280.)

[Verdoolaege *et al.* 2007] Sven Verdoolaege, Hristo Nikolov and Todor Stefanov. *pn: a tool for improved derivation of process networks*. EURASIP J. Embedded Syst., vol. 2007, no. 1, pages 19–19, January 2007. (Cited on page 173.)

[Verdoolaege *et al.* 2013] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado and Francky Catthoor. *Polyhedral Parallel Code*

Generation for CUDA. In ACM Transactions on Architecture and Code Optimization (TACO), 2013. To appear. (Cited on pages 30, 68, 69, 139, 155, 156, 271, 283, 300 and 306.)

[Volkov 2010] Vasily Volkov. *Better Performance at Lower Occupancy*. In Proceedings of GPU Technology Conference (GTC), September 2010. (Cited on pages 48, 159 and 275.)

[Volkov 2011] Vasily Volkov. *Unrolling parallel loops*. In Tutorial at the 2011 ACM/IEEE conference on Supercomputing, Supercomputing '11, 2011. (Cited on page 133.)

[Wang *et al.* 2010] Guibin Wang, YiSong Lin and Wei Yi. *Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU*. In Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing, GREENCOM-CPSCOM '10, pages 344–350, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 115, 116, 117 and 118.)

[Ward 1999] Martin P. Ward. *Assembler to C Migration Using the FermaT Transformation System*. In ICSM, pages 67–76, 1999. (Cited on page 150.)

[Warren 1984] Joe Warren. *A hierarchical basis for reordering transformations*. In Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '84, pages 272–282, New York, NY, USA, 1984. ACM. (Cited on pages 113, 114, 115, 126 and 293.)

[Whaley & Dongarra 1998] R. Clint Whaley and Jack J. Dongarra. *Automatically tuned linear algebra software*. In Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on pages 158 and 308.)

[Wikibooks 2009] Wikibooks, editor. GNU C compiler internals. `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`, 2006-2009. (Cited on pages 149 and 303.)

[Wikipedia 2012a] Wikipedia. *ARM Cortex-A9 MPCore — Wikipedia, the free encyclopedia*, 2012. Online; accessed 24-February-2012; available at `http://en.wikipedia.org/wiki/ARM_Cortex-A9_MPCore`. (Cited on pages 4 and 260.)

[Wikipedia 2012b] Wikipedia. *GLSL — Wikipedia, The Free Encyclopedia*, 2012. Online; accessed 24-February-2012; available at `http://en.wikipedia.org/wiki/GLSL#A_sample_trivial_GLSL_geometry_shader`. (Cited on pages xvii and 17.)

[Wikipedia 2012c] Wikipedia. *OpenHMPP — Wikipedia, The Free Encyclopedia*, 2012. Online; accessed 24-February-2012; available at http://en.wikipedia.org/wiki/OpenHMPP. (Cited on pages xvii, 7, 27 and 262.)

[Wilson *et al.* 1994] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih wei Liao, Chau wen Tseng, Mary W. Hall, Monica S. Lam and John L. Hennessy. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*. SIGPLAN Notices, vol. 29, pages 31–37, 1994. (Cited on pages 140 and 301.)

[Wolf & Lam 1991a] M. E. Wolf and M. S. Lam. *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*. IEEE Trans. Parallel Distrib. Syst., vol. 2, pages 452–471, October 1991. (Cited on page 101.)

[Wolf & Lam 1991b] Michael E. Wolf and Monica S. Lam. *A data locality optimizing algorithm*. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM. (Cited on page 170.)

[Wolfe 1989] Michael Wolfe. *Iteration Space Tiling for Memory Hierarchies*. In Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. (Cited on page 170.)

[Wolfe 1990] Michael Joseph Wolfe. Optimizing supercompilers for supercomputers. MIT Press, Cambridge, MA, USA, 1990. (Cited on pages 112, 115 and 293.)

[Wolfe 2010] Michael Wolfe. *Implementing the PGI Accelerator model*. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU, pages 43–50, New York, NY, USA, 2010. ACM. (Cited on pages 7, 27, 62, 87, 138, 262, 280 and 300.)

[Wolfe 2011] Michael Wolfe. *Optimizing Data Movement in the PGI Accelerator Programming Model*, February 2011. Online; accessed 24-February-2012; available at http://www.pgroup.com/lit/articles/insider/v3n1a1.htm. (Cited on pages xvii, 28, 62 and 280.)

[Wong *et al.* 2010] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos. *Demystifying GPU microarchitecture through microbenchmarking*. In Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on, pages 235 –246, march 2010. (Cited on pages 54 and 278.)

[Xiao & chun Feng 2010] Shucai Xiao and Wu chun Feng. *Inter-Block GPU Communication via Fast Barrier Synchronization*. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010. (Cited on page 42.)

[Xue 2000] Jingling Xue. Loop tiling for parallelism. Kluwer Academic Publishers, Norwell, MA, USA, 2000. (Cited on page 170.)

[Xue 2005] Jingling Xue. *Enabling Loop Fusion and Tiling for Cache Performance by Fixing Fusion-Preventing Data Dependences*. In Proceedings of the 2005 International Conference on Parallel Processing, pages 107–115, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 114.)

[Yan *et al.* 2009] Yonghong Yan, Max Grossman and Vivek Sarkar. *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA*. In Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on pages xvii, xix, 28, 29, 62, 68, 69, 139, 280 and 283.)

[Yang & Chang 2003] Chao-tung Yang and Shun-chyi Chang. *A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters*. In Proc. of Intl Conf. on Computational Science, pages 1079–1088. Springer-Verlag, 2003. (Cited on pages 2 and 258.)

[Yang *et al.* 2010] Yi Yang, Ping Xiang, Jingfei Kong and Huiyang Zhou. *A GPGPU compiler for memory optimization and parallelism management*. In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. (Cited on page 156.)

[Yelick *et al.* 1998] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella and Alex Aiken. *Titanium: A High-Performance Java Dialect*. In Concurrency: Practice and Experience, pages 10–11, 1998. (Cited on pages 6 and 261.)

[Yi 2011] Qing Yi. *Automated Programmable Control and Parameterization of Compiler Optimizations*. In Proceedings of the International Symposium on Code Generation and Optimization, New York, NY, USA, April 2011. ACM. (Cited on pages 143, 149, 302 and 304.)

[Yuki *et al.* 2010] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger and Kevin O'Brien. *Automatic creation of tile size selection models.* In Proceedings of the 8th annual IEEE/ACM international symposium on Code Generation and Optimization, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM. (Cited on page 170.)

[Zhang *et al.* 2011a] Junchao Zhang, Babak Behzad and Marc Snir. *Optimizing the Barnes-Hut algorithm in UPC.* In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 75:1–75:11, New York, NY, USA, 2011. ACM. (Cited on pages 7 and 262.)

[Zhang *et al.* 2011b] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir and Jianmin Chen. *Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications.* In Workload Characterization (IISWC), 2011 IEEE International Symposium on, pages 205 –215, nov. 2011. (Cited on pages xviii, xix, 46, 52, 53 and 54.)

[Zhou *et al.* 2012] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni and David Padua. *Hierarchical overlapped tiling.* In Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM. (Cited on page 170.)

[Zhou 1992] Lei Zhou. *Complexity estimation in the PIPS parallel programming environment.* In Luc Bougé, Michel Cosnard, Yves Robert and Denis Trystram, editors, Parallel Processing: CONPAR 92—VAPP V, volume 634 of *Lecture Notes in Computer Science*, pages 845–846. Springer Berlin / Heidelberg, 1992. (Cited on pages 157, 167, 174, 307 and 311.)

[Zhu *et al.* 2004] YongKang Zhu, Grigorios Magklis, Michael L. Scott, Chen Ding and David H. Albonesi. *The Energy Impact of Aggressive Loop Fusion.* In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04, pages 153–164, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 115.)

[Zou & Rajopadhye 2012] Yun Zou and Sanjay Rajopadhye. *Scan detection and parallelization in "inherently sequential" nested loop programs.* In Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, pages 74–83, New York, NY, USA, 2012. ACM. (Cited on page 111.)

Acronyms

ALU Arithmetic and Logical Unit. 45–47, 50

AMD Advanced Micro Devices. 9, 18, 20, 31, 39, 40, 42–48, 53, 132, 158, 178, 261, 264, 265, 268–271, 304

AMP Accelerated Massive Parallelism. 21–23, 266

APGAS Asynchronous Partitioned Global Address Space. 7, 258

API Application Programming Interface. 6, 7, 12–15, 17–21, 31, 32, 36, 39, 40, 51, 55, 56, 99, 142, 149–151, 160, 211, 215, 256

APP Accelerated Parallel Processing. 20, 265

AST Abstract Syntax Tree. 78, 88

AVX Advanced Vector eXtensions. 45, 141, 146–148, 270

BLAS Basic Linear Algebra Subprograms. 138, 295

CAL Compute Abstraction Layer. 20, 265

CEA Commissariat à l'énergie atomique et aux énergies alternatives. 149

CFG Control Flow Graph. 78, 88

CPU Central Processing Unit. 4, 8, 12, 14–16, 19, 20, 22, 25, 30–32, 36, 40, 41, 45, 56, 62, 76, 77, 81–83, 99, 111, 123, 151, 153, 157, 166, 167, 173–175, 178, 197, 204, 211, 212, 259, 263–265, 268, 270, 281, 301, 303, 306, 307, 309

CTM Close To Metal. 18, 20, 264, 265

CUDA Compute Unified Device Architecture. 7, 13, 14, 18–21, 25, 28–32, 40, 51, 56, 64, 68, 82, 87, 96, 98, 99, 109, 111, 117, 133, 139–141, 146, 148–152, 154, 156, 159, 160, 162, 163, 178, 180, 195, 200, 202, 204, 212, 215, 258, 263, 265–267, 274, 278, 282, 285, 293, 294, 296, 297, 299, 300, 304–306, 311, 314

CU Compute Unit. 40, 41, 45, 47, 49–52, 62, 160–162, 195, 268, 269, 271, 272

DAG Directed Acyclic Graph. 123, 173

DDR Double Data Rate. 90, 268, 283

DG Dependence Graph. 121–123

DMA Direct Memory Access. 50, 72, 73, 76, 89, 130, 171, 280, 292

DRAM Dynamic Random Access Memory. 45

DSL Domain Specific Language. 215

DSP Digital Signal Processing. 31

ECC Error-correcting code. 20, 50, 52, 76, 178, 272

FFT Fast Fourier transform. 138, 146, 153, 154, 295, 299, 302

FIFO First In, First Out. 23

FIR Finite Impulse Response. 75

FMA Fused Multiply-Add. 40, 45, 50, 268

FPGA Field Programmable Gate Array. 20, 90, 146, 265, 283, 299

GCC GNU C Compiler. 88, 138, 144, 149–151, 204, 211, 213, 295, 299, 300, 310, 312

GCN Graphics Core Next. 47, 48, 271

GDDR Graphic Double Data Rate. 45, 51, 62, 178

GLSL OpenGL Shading Language. 15, 17, 264

GPGPU General-Purpose Processing on Graphics Processing Units. 4, 8, 12–15, 19–21, 23, 25, 40, 44, 46–49, 55, 63, 97, 116, 122, 131, 141, 150, 156, 212, 256, 259, 262–266, 268, 270, 271, 274, 277, 290, 293, 311

GPU Graphics Processing Unit. xix, 4, 5, 7–9, 12–14, 16–23, 25, 27, 30–32, 36, 37, 39–45, 48, 51–57, 59, 62–64, 76–84, 86–88, 90, 93, 94, 96–99, 104, 109, 111, 117, 118, 123, 126–128, 130–132, 134, 135, 138, 139, 141, 146, 151–153, 156, 157, 160–162, 167, 168, 170, 171, 173–175, 178–181, 191, 195, 197, 200–202, 204–206, 208, 211–213, 215, 256, 258–265, 267, 268, 270–278, 280–288, 290–295, 299–303, 307–314

HCFG Hierarchical Control Flow Graph. 88, 107, 121, 126

HLSL High Level Shader Language. 21, 44, 266, 270

HMPP Hybrid Multicore Parallel Programming. 7, 25, 27, 28, 56, 68, 87, 138, 204, 258, 267, 274, 279, 296

HPF High Performance Fortran. 77, 174, 280

IEEE Institute of Electrical and Electronics Engineers. 45, 50

ILP Instruction Level Parallelism. 39, 41, 44, 45, 47, 48, 50, 52–54, 113, 117, 123, 130, 132, 268–271, 273, 293

IL CAL Intermediate Language. 20, 265

IR Internal Representation. 122, 140–142, 149, 297

ISA Instruction Set Architecture. 19, 20, 43, 47, 146, 195

ISO International Organization for Standardization. 31, 38

JIT Just In Time. 19, 30, 267

JNI Java Native Interface. 29

LDG Loop Dependence Graph. 116, 118, 120, 121

LDS Local Data Store. 47

LFU Least Frequently Used. 83

LLVM Low Level Virtual Machine. 88, 138, 149, 299

LRU Least Recently Used. 83

MESI Modified Exclusive Shared Invalid. 83

MMX Matrix Math eXtension. 2, 255

MP-SoC MultiProcessor System-on-Chip. 148

MPPA Multi-Purpose Processor Array. 23, 215, 313

OpenACC . 7, 23, 28, 56, 258, 274

OpenCL Open Computing Language. 4, 7, 14, 20, 21, 23, 30–32, 34–39, 42, 43, 56, 68, 82, 87, 96, 98, 99, 109, 133, 134, 139, 140, 146, 148, 149, 151, 152, 159, 160, 163, 167, 171, 173, 175, 180, 195, 206, 208, 212, 213, 256, 258, 263, 265–267, 274, 275, 282, 285, 286, 293, 294, 296, 297, 299, 300, 304–308, 311, 312

OpenGL Open Graphics Library. 15, 17, 18, 264

OpenHMPP Open Hybrid Multicore Parallel Programming. 25

OpenMP Open Multi Processing. 23, 25, 27, 28, 87, 96, 139–141, 146, 148, 149, 202, 204, 208, 212, 266, 282, 296, 299, 309

PCIe PCI Express. 50, 51, 62, 76, 86, 117, 128, 201, 272, 276, 280, 290

PC Personal Computer. 5

PE Processing Element. 40, 41, 45, 47, 49–52, 86, 171, 268–272

PGAS Partitioned Global Address Space. 6, 257

PIPS Paralléliseur Interprocédural de Programmes Scientifiques. 62, 65, 78–80, 88, 89, 92, 93, 96–98, 101–107, 109, 111, 118, 120–122, 126, 128, 131, 134, 135, 138–140, 142, 149–157, 167, 170, 171, 174, 175, 211–213, 277, 283–292, 294–303, 307–309, 311, 312

PTX Parallel Thread eXecution. 19, 88, 195

PyPS Pythonic PIPS. 142, 145, 147–150, 298–300

QUAST Quasi-Affine Selection Tree. 91, 283

RAM Random Access Memory. 76

RDG Reduced Dependence Graph. 121–123

SAC SIMD Architecture Compiler. 144, 145

SDK Software Development Kit. 20, 202

SFU Special Function Unit. 40, 50, 268

SIMD Single Instruction stream, Multiple Data streams. 13, 19, 37, 44, 45, 47, 50, 51, 140, 157, 159, 191, 269–272, 297, 303, 305

SIMT Single Instruction stream, Multiple Thread streams. 32, 37

SLOC Source Lines Of Code. 146, 147, 149

SMP Symmetric MultiProcessing. 83

SPDD Single Program Distributed Data. 77, 280

SSA Simple Static Assignment. 195

SSE Streaming SIMD Extension. 3, 45, 140, 147, 148, 151, 270, 297, 300

TLP Thread Level Parallelism. 39, 41, 44, 45, 47, 48, 52–54, 132, 268–271, 273, 293

TR Textual Representation. 140, 141, 297

VLA Variable Length Array. 68, 133, 279, 293

VLIW Very Long Instruction Word. 2, 23, 41, 43–47, 52, 53, 132, 146, 255, 269–271, 299

XML Extensible Markup Language. 150

Résumé en français

Cette section contient pour chaque chapitre une traduction de son introduction et un résumé de chacune de ses sections. Le lecteur intéressé est invité à se reporter au chapitre correspondant pour obtenir des précisions sur un sujet particulier.

1 Introduction

1.1 La Prophétie

Il était une fois, un monde dans lequel les développeurs écrivaient joyeusement leur code avec en tête la simple architecture de von Neumann (voir Figure 1.1). La performance était un facteur important bien sûr, mais ils étaient alors protégés par une bénédiction qui leur permettait de démarrer un projet nécessitant une puissance de calcul alors indisponible. En effet, le temps que le projet aboutisse était l'occasion d'évolutions significatives des performances des processeurs. La prophétie sur laquelle comptait chaque développeur était connue sous le nom de *Loi de Moore*. Elle est régulièrement citée sous la forme suivante (see [Srinivasan 2012, Manegold 2002, Yang & Chang 2003]) :

> la fréquence des CPUs doublera tous les dix-huit mois. [1]

Cette courte et simple affirmation a été insérée dans l'esprit de générations de développeurs pendant des décennies. Tout allait pour le mieux jusqu'à ce qu'un oiseau de mauvais augure annonça :

> ça ne peut pas continuer éternellement. La nature des exponentielles est que vous les poussez trop loin et finalement une catastrophe se produit. [2]

Il n'était pas le premier à remettre en cause la Prophétie, mais cette fois ce n'était autre que Gordon Moore lui-même [Dubash 2005], l'auteur de la Prophétie. Ce fût terrible pour les développeurs, et la plupart d'entre-eux s'enfermèrent dans le déni. Petit à petit, l'idée que la fréquence des processeurs n'allait pas continuer à augmenter comme auparavant a fait son chemin. En réalité, la Prophétie originelle peut figurer dans le top 10 des affirmations déformées, juste après "Luke, je suis ton père." En fait Moore avait affirmé à l'origine que :

> La complexité du coût minimum des composants a augmenté environ d'un facteur deux chaque année . . . . Certainement qu'à court terme on peut s'at-

1. "the CPU clock speed will double every eighteen months."
2. "it cannot continue forever. The nature of exponentials is that you push them out and eventually disaster happens."

> tendre à ce que ce taux continue, s'il n'augmente pas. Sur le long terme, le taux d'augmentation est plus incertain, bien qu'il n'y ait pas de raison de croire qu'il ne va pas rester environ constant pour au moins dix années. [3]

Les quarante années de vie facile étaient terminées, comme l'illustre la Figure 1.2, et les développeurs étaient sur le point de faire face à un nouveau défi. En effet les concepteurs de circuits, face à l'impossibilité d'augmenter la fréquence, ont sauté à pieds joints dans un monde parallèle [4]. Ils ont alors contourné la limite de fréquence des processeurs en agrégeant plusieurs processeurs par circuit, multipliant la performance brute maximale théorique atteignable par un circuit. L'ère du multicœurs avait débuté.

Les développeurs découvraient un nouvel univers : le temps d'exécution de leurs programmes n'était plus réduit significativement quand un nouveau processeur apparaissait sur le marché. Dans ce nouvel univers ils devaient repenser leurs algorithmes pour exploiter de multiples processeurs. Comme si ce n'était pas assez compliqué, certains concepteurs de circuits, qui avaient probablement rejoint le côté obscur de la force, commençaient à introduire des composants exotiques. Ces plate-formes matérielles étaient hautement parallèles, mais également très difficiles à programmer. Ainsi le chevalier blanc des développeurs qui souhaitait relever le défi devait non seulement repenser ses algorithmes mais également gérer manuellement des hiérarchies mémoires complexes.

Bienvenue dans le monde du calcul hétérogène !

1.2 Motivation

"Votre déjeuner gratuit est bientôt terminé" [5] Herb Sutter commença son article en 2005 [Sutter 2005] avec cette déclaration. La limite de l'augmentation de fréquence interdit maintenant d'espérer une augmentation significative de la performance des programmes séquentiels. Le futur est hétérogène, du monde de l'embarqué des *téléphones intelligents* [6] et des tablettes jusqu'aux plus grands super-calculateurs. Sutter a écrit une suite à son

3. "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year. . . . Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years."

4. Le parallélisme a été présent dans les processeurs à cœur unique depuis 1989 avec le processeur Very Long Instruction Word (VLIW) i860, et ensuite avec le jeux d'instruction Matrix Math eXtension (MMX) dans le Pentium. Depuis les développeurs avaient la possibilité d'exprimer du parallélisme fin dans le jeu d'instruction.

5. "Your free lunch will soon be over."

6. smartphones

article [Sutter 2011] dans laquelle il déclare : “Maintenant bienvenue dans la jungle des circuits matériels.”[7]. La Figure 1.3 illustre cette évolution.

Dans le monde des systèmes embarqués, les *téléphones intelligents* sont basés sur des processeurs multicœurs et incluent des unités de calcul vectorielles ainsi qu'un processeur graphique[8]. Par exemple le processeur A5, utilisé dans l'iPhone 4S, comporte deux cœurs généralistes couplés à deux cœurs spécialisés pour les calculs graphiques [AnandTech 2011]. Le même processeur est également utilisé dans la tablette Apple iPad 2. Le dernier processeur Tegra 3 de Nvidia comporte quatre cœurs généralistes et douze cœurs graphiques [Nvidia 2012b]. Dans les deux cas, chaque cœur inclut une unité vectorielle NEON d'une largeur de 128 bits [Wikipedia 2012a]. La prochaine génération de Tegra offrira l'accès aux cœurs graphiques pour du calcul généraliste[9] à l'aide de l'Application Programming Interface (API) standard Open Computing Language (OpenCL).

Dans le monde des supercalculateurs, le parallélisme est présent depuis des décennies maintenant. Les machines vectorielles ont été remplacées par des grappes de systèmes multiprocesseurs multicœurs dans la liste du Top500 [TOP500 Supercomputing Sites 2012]. La nouvelle tendance est maintenant d'ajouter des accélérateurs matériels dans ces systèmes, pour la plupart des processeurs graphiques (GPUs), ajoutant un nouveau niveau de complexité. La liste Top500 de juin 2011 comportait trois systèmes comportant des GPUs parmi les cinq premières places. Il y avait aussi cinq systèmes basés sur des GPUs dans la liste Green500 [Feng & Cameron 2007] parmi les dix premières places [The Green500 2011]. Le supercalculateur *Titan*, à base de carte graphique Nvidia Tesla K20, décroche la première place du dernier classement du Top500 de novembre 2012 [TOP500 Supercomputing Sites 2012], et il est aussi intéressant de noter l'entrée à la septième place d'un système qui inclut des coprocesseurs Intel Xeon Phi.

Il est difficile voir impossible de trouver un ordinateur personnel ne possédant qu'un seul cœur de nos jours. Le double-cœurs est le standard en entrée de gamme, le quad-cœurs occupe les gammes intermédiaires, et le haut de gamme offre six cœurs et plus. Rendu nécessaire par l'industrie du jeu vidéo, les cartes graphiques livrées avec les ordinateurs personnels sont de plus en plus puissantes et sont utilisées dans un nombre croissant d'applications en dehors de leur utilisation première : le rendu et l'affichage de scènes 3D.

Le problème qui se pose maintenant, alors que les plate-formes hétérogènes sont largement disponibles, peut être résumé sous la forme des propriétés *triple-P* [Adve 1993, Benk-

7. “Now welcome to the hardware jungle.”
8. Graphics Processing Unit (GPU)
9. General-Purpose Processing on Graphics Processing Units (GPGPU)

ner *et al.* 2011, Adve 2011] illustrées par la Figure 1.4 :

- Performance : le programme exploite les capacités maximales du matériel.
- Portabilité : le code écrit par le développeur s'exécute sur un grand nombre de plate-formes.
- Programmabilité : le développeur implémente ses algorithmes rapidement.

Une quatrième propriété peut être ajoutée : l'économie d'énergie[10]. Non seulement parce que la batterie de nos *téléphones intelligents* est limitée, mais également car en 2007 chacun des dix plus grands supercalculateurs consommait autant d'énergie qu'une ville de quarante mille habitants [Feng & Cameron 2007]. Le concept de programmes qui adaptent leur exécution pour économiser de l'énergie est étudié [Hsu & Feng 2005], en faisant appel à un compromis entre performance et consommation d'énergie.

Des solutions qui tentent de répondre à toutes ces propriétés ont été explorées depuis des décennies. La complexité de programmation augmente avec la spécialisation des circuits tels que les accélérateurs matériels, au point de rendre la programmation impraticable pour la majorité des développeurs.

La performance s'est améliorée avec les compilateurs, permettant à de nouveaux langages de devenir concurrentiels face au langage C, *la* référence. Ce dernier reste incontesté quand il est nécessaire d'avoir un contrôle précis de l'exécution d'un programme pour du matériel donné.

Du point de vue de la portabilité, il est difficile de maintenir un programme de grande taille écrit en C qui exploite plusieurs plate-formes hétérogènes. Une pratique courante est alors de se restreindre à un ensemble de fonctionnalités communes à ces plate-formes, limitant alors les performances atteignables en pratique par rapport à la capacité maximale du matériel.

Finalement, la programmabilité a été traitée sous la forme de bibliothèques et de nouveaux langages. Par exemple, les langages UPC [UPC Consortium 2005], Co-Array Fortran [ISO 2010], ou Titanium [Yelick *et al.* 1998] utilise le modèle d'un espace d'adressage global partitionné (Partitioned Global Address Space (PGAS)). Le partitionnement est effectué de telle sorte que la mémoire est répartie entre les différents processeurs [Coarfa *et al.* 2005]. La localité des références mémoires est exploitée lors de l'exécution à l'aide de stratégies telles que *celui qui possède la donnée effectue le calcul*. L'objectif de ces langages est de soulager le développeur de l'application du poids de la gestion des accès mémoires distants. Ce modèle mémoire plat a ensuite évolué avec l'ajout de l'asynchronisme dans

10. "**Power**" en anglais

les languages dits "Asynchronous Partitioned Global Address Space (APGAS)" tels que X10 [Ebcioğlu *et al.* 2004] ou Chapel [Chamberlain *et al.* 2007]. L'exécution concurrente est explicite et les développeurs expriment des constructions asynchrones à plusieurs niveaux. Alors que ces langages exigent des développeurs qu'ils modifient leur conception des algorithmes utilisés, des interfaces de haut-niveau sont fournies pour abstraire l'architecture matérielle sous forme de couches. Toutefois ces langages sont récents et pas encore très répandus. Des critiques ont été exprimées au sujet de la performance atteignable : dans la pratique le code semble devoir être optimisé pour une plate-forme cible pour obtenir de bonne performance [Zhang *et al.* 2011a].

Le standard OpenCL [Khronos OpenCL Working Group 2008, Khronos OpenCL Working Group 2011] a été développé pour cibler les accélérateurs matériels. Il fournit une abstraction du matériel, basée sur une interface pour gérer des accélérateurs attachés à un hôte, et un langage dérivé d'un sous-ensemble de C pour écrire des *noyaux*, c'est à dire des fonctions destinées à s'exécuter sur un accélérateur. Ce standard fournit une portabilité entre différents fournisseurs de matériel et place la programmabilité au niveau du langage C. Toutefois, la portabilité de la performance est difficile voir impossible à atteindre [Komatsu *et al.* 2010]. Une autre approche est basée sur des langages de directives, dans la lignée du célèbre standard OpenMP pour les systèmes à mémoire partagée. Par exemple, des ensembles de directives telles que proposées par Hybrid Multicore Parallel Programming (HMPP) [Wikipedia 2012c], PGI Accelerator [Wolfe 2010], ou plus récemment OpenACC [OpenACC Consortium 2012] permettent aux développeurs de programmer facilement des accélérateurs, tout en préservant la portabilité du code.

1.3 Plan

Le but de cette thèse est d'explorer le potentiel des compilateurs pour fournir une solution aux trois emphP : Performance, Portabilité, et Programmabilité. La solution considérée est la transformation automatique de code C ou Fortran vers un code équivalent exploitant un accélérateur matériel. La cible principale est le cas des accélérateurs de type GPUs : massivement parallèles et embarquant des giga-octets de mémoire. L'approche source-à-source bénéficie des interfaces de programmation Compute Unified Device Architecture (CUDA) et OpenCL. La programmabilité et la portabilité sont assurées par l'approche entièrement automatique. Les résultats de nombreuses mesures sont fournies pour montrer que la performance n'est pas sacrifiée.

L'approche retenue est pragmatique et les idées et stratégies présentées sont implémentées dans un nouveau compilateur source-à-source, Par4All [SILKAN 2010 (perso)], et validée avec des jeux d'essai. Le but principal est de fournir un prototype industriel d'une chaîne de compilation complète de bout en bout : du code séquentiel jusqu'au binaire prêt à s'exécuter en tirant partie d'un accélérateur matériel. Par conséquent, plutôt que de se concentrer sur une partie très limitée du problème, cette thèse contribue à différent aspects de la problématique et tente d'explorer et de résoudre les problèmes qui se posent quand on construit une telle chaîne complète de compilation.

Cette approche d'un compilateur est utile pour les applications existantes comme pour les nouveaux développements. Un compilateur efficace diminue le coût d'entrée mais également le coût de sortie quand une nouvelle plate-forme doit être ciblée. La maintenance est facilitée dans la mesure où le code est exprimé avec une sémantique séquentielle cohérence avec les habitudes des développeurs. Quand le code compilé n'est pas assez rapide, des parties spécifiques et particulièrement coûteuses peuvent être optimisées manuellement pour une architecture donnée : l'approche source-à-source rend possible des optimisations sur le code obtenu après traitement par le compilateur hétérogène.

Le choix des langages C et Fortran est motivé par leur utilisation courante dans le domaine des simulations numériques et du calcul à haute performance. Le langage C est également un choix courant pour les outils générant du code depuis une représentation de plus haut niveau ou un langage de script. Pour illustrer l'intérêt de ce choix, des exemples de code Scilab [Scilab Consortium 2003] sont inclus. Ils sont automatiquement convertis en code C séquentiel à l'aide d'un compilateur Scilab, et ensuite transformés pour exploiter les accélérateurs matériels en utilisant les différentes techniques présentées dans ce manuscrit.

Le Chapitre 2 présente l'histoire des GPUs et l'émergence de l'utilisation des cœurs graphiques pour du calcul généraliste (GPGPU). L'évolution des circuits matériels est suivie par celle des langages de programmation qui échouent à répondre aux trois *P*s à la fois. Les architecture des GPUs et leur évolution sont introduites pour illustrer les contraintes à respecter par un compilateur pour exploiter la capacité maximale des circuits : mémoires distribuées, motifs d'accès la mémoire sur GPU, parallélisme de grain fin, et support d'opérations atomiques.

Le Chapitre 3 explore des solutions à la distribution automatique des données sur le Central Processing Unit (CPU) et les mémoires de l'accélérateur. La représentation abstraite sous forme régions de tableaux convexes est d'abord présentée. Une génération des communications hôte-accélérateur basée sur les régions de tableau est ensuite exposée.

Une nouvelle méthode interprocédurale d'optimisation est ensuite proposée et validée à l'aide d'expériences. L'algorithme se base sur une analyse statique et minimise les communications en préservant les données dans la mémoire de l'accélérateur pour éviter les communications redondantes.

Un ensemble de transformations de programmes et de boucles pour isoler et optimiser le code destiné à être exécuté par l'accélérateur sont développés dans le Chapitre 4. Une méthode souple est proposée pour faire correspondre l'espace d'itération d'un nid de boucles parallèles aux différentes couches d'exécution des GPUs. J'ai conçu et implémenté une transformation pour substituer les variables d'induction dans des nids de boucles et permettre leur parallélisation. Deux différents algorithmes de parallélisation de nids de boucles sont présentés. Leur impact respectif, ainsi que l'impact de leur combinaison est étudié. J'ai proposé et implémenté une modification de ces algorithmes pour paralléliser en présence de réductions. J'ai conçu et implémenté une nouvelle transformation pour exploiter les capacités des circuits matériel à exécuter des opérations atomiques pour paralléliser les nids de boucles comportant des réductions. J'ai conçu et implémenté une méthode de fusion de boucles, et j'ai proposé des heuristiques pour conduire le choix des fusions à effectuer pour répondre aux contraintes particulières posées par les GPUs. Trois différents motifs de remplacement de tableaux par des scalaires sont présentés. J'ai modifié l'implémentation existante de cette transformation pour améliorer les performances dans le cas particulier des GPUs. L'impact du déroulage de boucles et de la linéarisation des accès aux tableaux est également présenté. Toutes ces transformations sont validées par des mesures de performance.

La chaîne complète de compilation est présentée dans le Chapitre 5, du code source séquentiel jusqu'au binaire final ainsi que le support exécutif proposé. Un gestionnaire de passes de compilation programmable est présenté, et la flexibilité qu'il offre est exploitée pour produire la chaîne de compilation. Des analyses interprocédurales sont mises en œuvre et requièrent le code source de toutes les fonctions présentent dans le graphe d'appel. Cette contrainte peut poser un problème, particulièrement pour les librairies externes. J'ai proposé et implémenté une solution souple qui répond dynamiquement aux besoins du compilateur lors des différentes phases du processus de compilation.

Les perspectives d'extensions pour le cas de multiples accélérateurs sont explorées dans le Chapitre 6. Deux approches pour extraire le parallélisme sont étudiées. J'ai implémenté une extraction simple du parallélisme de tâche, et modifié l'implémentation existante du tuilage symbolique pour l'adapter à cette problématique. Le cadre exécutif StarPU est

utilisé pour exploiter le parallélisme de tâches et gérer l'ordonnancement sur plusieurs GPUs.

Finalement, les résultats expérimentaux sont présentés dans le Chapitre 7 pour valider les solutions définies dans les chapitres précédents. Vingt jeux d'essai sont extraits des suites Polybench et Rodinia. Une simulation numérique réelle de type n-corps est utilisée pour montrer qu'une accélération peut être obtenue automatiquement sur une application qui dépasse le cadre du simple jeu d'essai. Plusieurs cibles matérielles de Nvidia et d'Advanced Micro Devices (AMD) sont utilisées pour montrer de quelle manière les transformations de programmes mises en œuvre impactent les performances de différentes manières en fonction des architectures.

En raison de la variété des sujets adressés dans cette thèse, la présentation des travaux liés est répartie dans les différents chapitres.

2 Calcul généraliste sur processeurs graphiques : histoire et contexte

Le règne du processeur généraliste sous sa forme classique n'est plus hégémonique et le monde des processeurs est maintenant hétérogène. Les GPUs sont candidats au rôle de coprocesseurs depuis plus d'une décennie maintenant. D'autres architectures ont aussi été développées, tel que le Larabee d'Intel [Seiler *et al.* 2008]. Ce dernier n'a jamais vraiment été mis sur le marché en tant que processeur graphique et a récemment trouvé une autre voie sous la forme de coprocesseur pour calculs intensifs en adoptant le nom commercial de Xeon Phi à la fin de l'année 2012. Citons également le processeur Cell [Hofstee 2005] développé par IBM et popularisé par son utilisation dans la console de jeu PlayStation 3 de Sony. Toutefois, bien que plusieurs travaux de recherches ont visé à adapter des algorithmes pour tirer parti de son architecture complexe, le Cell a été abandonné. Cet échec est lié à son modèle mémoire et son modèle de programmation trop complexe, particulièrement face aux alternatives industrielles : les fabricants de circuits graphiques sont entrés dans le marché du calcul généraliste.

Les unités dédiées au calcul graphique offrent, en général via leurs pilotes, un accès à une interface de programmation standard tel qu'OpenGL [Khronos OpenGL Working Group 1994, Khronos OpenGL Working Group 2012] et DirectX [Microsoft 1995, Microsoft 2012]. Ces interfaces sont spécifiques au traitement graphique, l'application principale de ce type de circuits. Le traitement graphique fait appel à de nombreuses opérations naturellement vectorielles, les GPUs ont donc été dotés de capacités visant à multiplier un

vecteur par un scalaire en une opération unique. Ces capacités vectorielles ont été détournées du traitement graphique vers des calculs généralistes.

Ce chapitre présente d'abord dans la Section 2.1 l'histoire du calcul généraliste basé sur circuits graphiques (GPGPU), puis la Section 2.2 relève les points clés de l'évolution des modèles de programmation, ainsi que les différentes initiatives qui ont été prises pour placer au premier plan la pratique du GPGPU. Le standard OpenCL est introduit en détail Section 2.3. Les architectures GPU contemporaines sont présentées Section 2.4. Finalement je liste les différents défis que ces architectures présentent aux développeurs et aux concepteurs de compilateurs.

2.1 Historique

L'utilisation de matériel graphique pour des calculs généralistes a été un sujet de recherche pendant plus de vingt ans. Harris et al. propose [Harris *et al.* 2002] un historique qui commence avec la machine *Ikonas* [England 1978], la *Pixel Machine* [Potmesil & Hoffert 1989], et *Pixel-Planes 5* [Rhoades *et al.* 1992]. En 2000, Trendall et Stewart [Trendall & Stewart 2000] établissaient un aperçu des expériences passées exploitant du matériel graphique. Jusqu'en 2007, les GPUs n'exposaient qu'un flux d'exécution propre au traitement graphique au travers d'interface de programmation tel qu'OpenGL. Toute l'élégance la recherche d'alors résidait dans la correspondance établie entre des opérations mathématiques généralistes et les opérations graphiques possibles dans ce flux d'exécution [Trendall & Stewart 2000]. Une limitation clé était qu'à cette époque les circuits graphiques n'offraien t que des unités de calculs flottants en simple précision, alors que les calculs en double précision sont souvent requis pour la plupart des simulations numériques.

Les GPUs se sont popularisés pendant les dernières décennies, avec un excellent ratio coût/performance. Une conséquence a été la naissance d'une forte mode dans la recherche expérimentale pour exploiter ces circuits spécialisés. Cette tendance s'est reflétée d'abord dans l'évolution des interfaces de programmation. A la fois OpenGL et DirectX ont introduit les *shaders* en 2001, et par là même ajouter programmabilité et flexibilité au flux d'exécution graphique. Toutefois, utiliser une de ces interfaces de programmation graphique était toujours un point de passage obligatoire. Par conséquent la pratique du GPGPU était un défi encore plus important qu'il ne l'est aujourd'hui.

En 2003 Buck et al. [Buck *et al.* 2004] ont implémenté un sous-ensemble du langage de *streaming* Brook pour cibler les GPUs. Ce nouveau langage, nommé BrookGPU, ne présente pas au développeur le flux d'exécution graphique. Le code est compilé vers les

interfaces DirectX ou OpenGL. BrookGPU a été utilisé par exemple dans le cadre du projet Folding@home [Pande lab Stanford University 2012]. Plus d'informations au sujet de Brook et BrookGPU sont données à la Section 2.2.3.

Ian Buck, qui a conçu Brook et BrookGPU, a rejoint Nvidia pour concevoir l'environnement CUDA. Le langage défini par CUDA partage des similitudes avec BrookGPU. Toutefois, alors que BrookGPU propose une abstraction générique, l'interface CUDA est spécifique à Nvidia et à la nouvelle architecture scalaire introduite simultanément. La Section 2.4.5 présente cette architecture de manière plus détaillée. CUDA est à la fois une interface de programmation et un langage pour programmer les GPUs plus facilement. Le flux d'exécution graphique n'existe plus en tant que tel et l'architecture matérielle est unifiée et exposée comme un ensemble de processeurs vectoriels. CUDA est introduit avec plus de détail à la Section 2.2.4.

De 2004 à 2012, l'évolution de la performance en calcul flottant des GPUs a augmenté bien plus rapidement que celle des CPUs, comme illustré par la Figure 2.1. La programmabilité offerte par CUDA, combinée avec l'avantage de performance des GPUs, a rendu la pratique du GPGPU de plus en plus populaire pour les calculs scientifiques au cours des cinq dernières années.

L'intérêt croissant dans la pratique du GPGPU a concentré beaucoup d'attentions, au point d'aboutir à la standardisation d'une interface de programmation dédiée et d'un langage pour programmer les accélérateurs : le *Open Computing Language* connu sous le sigle OpenCL (voir Section 2.3 pour plus de détail).

D'autres modèles de programmation émergent, tels que les langages à base de directives. Ces langages laissent les développeurs écrire du code portable, maintenable, et, avec un peu de chance, efficace. Les directives sous forme de *pragma*s sont ajoutées au code séquentiel pour indiquer au compilateur quelle partie du code doit être exécutée sur l'accélérateur. Cette méthode est moins intrusive mais aboutit actuellement à des performances plus limitées. Plusieurs propositions de directives sont présentées à la Section 2.2.10.

2.2 Langages, cadres d'applications, et modèles de programmation

L'historique des langages de programmation comporte de nombreux langages, cadres d'exécution, et modèles de programmation conçus pour programmer les accélérateurs. Certains ont été conçus dans le but spécifique de la cible d'origine des GPUs, c'est à dire le calcul graphique, et ont été détournés plus tard pour la pratique du GPGPU. D'autres ont

été conçus dès la base pour répondre au besoin de la pratique du GPGPU.

Cette section passe en revue les contributions, approches, et paradigmes majeurs impliqués au cours de la dernière décennie dans la programmation des accélérateurs matériels.

2.2.1 Open Graphics Library (OpenGL)

En 1992, Silicon Graphics Inc. introduit la spécification d'OpenGL. Cette interface est utilisée pour programmer les applications qui font appel à du calcul graphique 2D et 3D et fournit une abstraction du flux de traitement graphique. Les objets manipulés sont des points, lignes et polygones. Ils sont convertis en pixels via le flux d'exécution graphique, paramétré par la machine à état d'OpenGL. A la base, chaque étage du flux de traitement n'était capable que d'effectuer des opérations prédéfinies et n'était configurable que dans des limites restreintes. Cependant depuis l'introduction des *shaders* dans Open Graphics Library (OpenGL) 2.0 [Khronos OpenGL Working Group 2004], le langage OpenGL Shading Language (GLSL) rend plusieurs étages complètement programmables. La Figure 2.2 montre l'exemple d'un *shader* de calcul généraliste.

2.2.2 Shaders

Les *shaders* sont de petits programmes utilisés dans le cadre du traitement graphique à différents étages du flux de traitement. Ils sont utilisés pour décrire l'absorption et la diffusion de la lumiére ainsi que les textures à appliquer, les réflexions et réfractions, l'ombrage, et le déplacement de primitives graphiques. Le processus de rendu place les *shaders* comme de parfaits candidats pour une exécution parallèle sur des processeurs graphiques vectoriels, libérant le CPU de cette tâche coûteuse et produisant des résultats plus rapidement.

Le niveau de programmation est actuellement proche du langage C et le parallélisme est implicite. Cependant s'ils ajoutent de la flexibilité et de la programmabilité au flux d'exécution graphique pour les calculs généralistes, ils ne permettent pas aux développeurs de s'abstraire de l'interface graphique et des primitives associées. La Figure 2.3 illustre un exemple de *shader* géométrique utilisant le langage GLSL.

2.2.3 Brook and BrookGPU

Jusqu'à 2003, le seul moyen de bénéficier des ressources du matériel graphique était d'utiliser les *shaders* proposés par les interfaces graphiques OpenGL et DirectX. BrookGPU [Buck *et al.* 2004] implémente un sous-ensemble de la spécification du langage Brook [Buck 2003] pour cibler les GPUs. Il offre la possibilité de compiler le même code pour différentes cibles, OpenGL and DirectX bien sûr, mais aussi les *shaders* Nvidia Cg et plus tard l'interface généraliste d'AMD : Close To Metal (CTM). La Figure 2.4 reproduit

un exemple d'une simple opération SAXPY à l'aide de BrookGPU.

2.2.4 Nvidia Compute Unified Device Architecture (CUDA)

L'interface propriétaire Nvidia CUDA partage des similitudes avec BrookGPU. Toutefois là où BrookGPU est générique et possède des générateurs pour plusieurs cibles, CUDA offre un accès à des fonctionnalités spécifiques aux architectures de Nvidia. CUDA s'affranchit également de limitations majeures de Brook, tel que le modèle mémoire rigide de Brook [Buck 2009]. La technologie CUDA inclut : un pilote, un support d'exécution[11], des librairies mathématiques (BLAS, FFT, ...), un langage basé sur une extension d'un sous-ensemble de C++, et une interface fournissant un modèle d'abstraction pour l'architecture. La mémoire du GPU est accédée de manière linéaire. Toutefois, pour obtenir de bonnes performances, le code doit être optimisé spécifiquement pour l'architecture.

2.2.5 AMD Accelerated Parallel Processing, *FireStream*

FireStream est la solution d'AMD pour la pratique du GPGPU. Le nom fait référence à la fois au matériel et au logiciel livré par AMD. La première carte accélératrice a été livrée en 2006 conjointement à une interface de programmation pour la pratique du GPGPU : Close To Metal (CTM). Cette interface est proche du matériel et expose au développeur le jeu d'instruction natif. Ce compromis offre la possibilité pour le développeur d'optimiser son code de manière très fine, mais au prix d'un effort accru dans le cas général. C'est pourquoi AMD a rapidement proposé une nouvelle solution, *Stream Computing*, basée sur Brook+. Ce dernier étant un langage de haut niveau basé sur Brook (voir Section 2.2.3). Au même moment CTM a été renommé Compute Abstraction Layer (CAL), devenu l'interface cible pour le code généré par Brook+. CAL fournit l'interface requise pour contrôler la carte graphique, ainsi que CAL Intermediate Language (IL), un langage proche de l'assembleur pour les GPUs AMD. La dernière version de la technologie d'AMD's est maintenant nommée Accelerated Parallel Processing (APP) et est basée sur OpenCL. Le support pour Brook+ et CTM a été stoppé, et l'interface CAL est dépréciée en faveur d'OpenCL. Le langage assembleur IL reste la cible pour le compilateur OpenCL.

2.2.6 Open Computing Language (OpenCL)

Le standard OpenCL définit une pile logicielle conçue pour écrire des programmes ciblant un grand nombre de plate-forme tels que CPUs, GPUs, Field Programmable Gate Array (FPGA) ou autres architectures embarquées. Ce standard inclut un langage, basé sur le standard C99, pour écrire du code s'exécutant sur des plate-formes hétérogènes. Il

11. "runtime"

définit une interface de programmation pour gérer les accélérateurs depuis l'hôte. OpenCL a été proposé par Apple au groupe Khronos en 2008 pour unifier les différents environnements de programmation dans un standard, qui a été finalement adopté plus tard la même année [Khronos OpenCL Working Group 2008]. La Section 2.3 fournit plus de détails sur OpenCL.

2.2.7 Microsoft DirectCompute

Microsoft propose sa propre solution pour la pratique du GPGPU avec DirectCompute [Microsoft 2010] livré en tant que composant de DirectX 11 fin 2009. L'interface de DirectCompute se base sur le langage de *shader* High Level Shader Language (HLSL) (identique à Nvidia Cg) pour fournir une solution qui s'affranchie du flux d'exécution graphique en faveur d'un accès direct tel que CUDA ou OpenCL. Les développeurs familiers avec HLSL/Cg peuvent alors transférer des zones mémoires directement vers ou depuis la mémoire embarquée de l'accélérateur, puis exécuter des *shaders* (ou noyaux de calcul) pour traiter ces zones mémoires. La Figure 2.5 montre un exemple d'un tel noyau de calculs.

2.2.8 C++ Accelerated Massive Parallelism (AMP)

Microsoft C++ Accelerated Massive Parallelism (AMP) est une spécification ouverte [Microsoft Corporation 2012a] pour permettre du parallélisme de données en tant qu'extension C++. La suite AMP de Microsoft a été livrée pour la première fois en janvier 2012. Elle est composée d'une extension au langage C++, d'un compilateur, d'un support exécutif.

Contrairement à Direct Compute présenté Section 2, il n'y a pas de séparation entre le code qui tourne sur l'accélérateur et le code hôte. Les calculs à exécuter sur un accélérateur sont isolés sous forme de lambda fonctions encapsulées dans une des constructions introduite par la spécification pour exprimer l'espace d'itération, telle que `parallel_for_each` par exemple. La Figure 2.6 montre un exemple de code C++ avant et après conversion vers C++ AMP.

2.2.9 Langage à base de directives et cadre de programmation

Suivant l'exemple du populaire standard Open Multi Processing (OpenMP) pour les machines à mémoire partagée, plusieurs propositions ont été faites pour l'ajout de directives, principalement aux langages C et Fortran. L'objectif est de répondre aux difficultés rencontrées par les développeurs pour écrire du code efficace, portable, et maintenable, tout en convertissant progressivement des applications séquentielles existantes vers l'utilisation d'accélérateurs.

Le principal inconvénient, est que même si les directives rendent le code plus simple à écrire, le développeur doit cependant posséder une bonne connaissance de la cible et de la manière dont l'algorithme peut être appliqué sur un accélérateur. Le code résultant de l'ajout des directives est finalement optimisé de manière spécifique à une architecture.

Les Figures 2.7, 2.8, 2.9, et 2.10 illustrent des exemples de traitements avec les langages de directives hiCUDA, HMPP, PGI Accelerator, et JCUDA.

2.2.10 Parallélisation automatique pour la pratique du GPGPU

Les travaux de parallélisation automatique de code séquentiel pour cibler des GPUs sont récents. Leung et al. [Leung *et al.* 2009] proposent une extension à un compilateur Just In Time (JIT) Java qui exécute des boucles parallèles sur un GPU. Nugteren et al. [Nugteren *et al.* 2011] présentent une technique pour automatiquement faire correspondre du code séquentiel à une implémentation sur un accélérateur en utilisant une approche de *skeletonization*. Cette technique est basée sur un jeu de *squelettes* prédéfinis pour le traitement d'images. L'étape de *skeletonization* reconnaît les fonctionnalités de l'algorithme à l'aide de reconnaissance de motifs, puis les remplace avec une autre implémentation prédéfinie pour l'accélérateur cible.

CUDA-Chill [Rudy *et al.* 2011] fournit une transformation automatique de programmes pour GPU utilisant le cadre de compilation Chill pour assembler des transformations de boucles. Toutefois les *recettes* Chill doivent être adaptées à chaque programme à traiter, limitant la portabilité de l'approche.

Baskaran et al. [Baskaran *et al.* 2010] introduisent une approche polyédrique à la parallélisation automatique vers CUDA de nids de boucles affines exprimées en C dans l'outil Pluto [Bondhugula *et al.* 2008c]. Plus récemment, le projet PPCG [Verdoolaege *et al.* 2013] suit le même chemin et produit des noyaux optimisés pour GPU en utilisant le modèle polyédrique.

2.3 Architectures cibles

Cette thèse se concentre sur les accélérateurs matériels tels que les GPUs. Les caractéristiques communes de tels accélérateurs sont les suivantes :

- une large mémoire embarquée : de l'ordre de quelques giga-octets ;
- un parallélisme massif : de quelques dizaines d'unités de calcul, à plusieurs milliers ;
- compatibilité avec le modèle de programmation défini par le standard OpenCL introduit dans la section 2.3.

Les plateformes matérielles les plus répandues qui correspondent à ces critères sont

fabriquées par AMD et Nvidia, et sont présentes dans les ordinateurs grand public. Cette section introduit et compare les architectures actuelles des GPUs. Les deux types de parallélisme exploité sont présentés : le parallélisme au niveau du flux d'instruction (Instruction Level Parallelism (ILP)) et le parallélisme entre fils d'exécution (Thread Level Parallelism (TLP)).

2.3.1 Conception d'un circuit graphique (GPU)

Un GPU est un circuit énorme et complexe, traditionnellement construit sur des unités de calcul très spécialisées pour traiter efficacement des opérateurs graphiques prédéfinis. Avec l'introduction des *shaders*, ces circuits sont devenus de plus en plus flexibles et programmables. Puisque que la pratique du GPGPU est l'objectif principal de cette thèse, seule la puissance de calcul dédiée aux *shader* et les hiérarchies mémoires sont importantes. Les circuits dédiés au traitement graphique sont volontairement omis dans cette section.

Au niveau le plus fin, nous trouvons les unités de calcul (Processing Element (PE)), capables d'opérations basiques telles que des additions, des multiplications, ou au mieux des opérations combinées (Fused Multiply-Add (FMA)). Certains PEs *sous stéroïdes* sont capables de traiter des fonctions transcendantes telles que des opérations trigonométriques, des exponentielles, ou des racines carrées. De telles unités sont nommées *Special Function Unit (SFU)*.

Plusieurs PEs sont alors groupés ensemble dans un multiprocesseur (Compute Unit (CU)). Une CU inclue les circuits de logiques partagés par les PEs, tels que le décodage des instructions, les registres, le cache, un ordonnanceur, etc.

Un circuit graphique est ensuite conçu par assemblage de plusieurs CUs à l'aide d'un réseau d'interconnexion. Un ordonnanceur matériel global est ensuite ajouté pour distribué le travail aux différentesCUs, et enfin des contrôleurs mémoire incluant un éventuel cache global offrent un accès aux giga-octets de DDR embarqués. Les CUs peuvent également être groupées avant d'être reliées au réseau d'interconnexion et ce groupe partage certaines ressources tels que le cache ou des unités de traitement graphique. La Figure 2.15 illustre cette vue conceptuelle de l'architecture d'un GPU.

Une telle vue n'est pas si éloignée de ce qu'on retrouve dans un processeur multi-cœurs, mais le diable est dans les détails. Les choix qui sont réalisés à chaque niveau sur le nombre d'éléments et la manière dont ils sont groupés et partagent des ressources a un impact important sur la programmabilité. En général, par rapport à un CPU, l'essentiel

de la surface du circuit est utilisée pour de la logique de calcul. C'est pourquoi il y a beaucoup de PEs avec un groupement complexe partageant les ressources qui n'effectuent pas de calculs, peu de mémoire cache, et une bande passante mémoire importante, mais également une latence élevée.

En général, les concepteurs essayent d'obtenir des CUs aussi simple que possible et n'incluent pas de capacité de réordonnancement du flux d'instruction [12], et donc la source principale de parallélisme est le parallélisme de fils d'exécution (Thread Level Parallelism (TLP)). Toutefois, le parallélisme dans le flux d'instruction (Instruction Level Parallelism (ILP)) peut être exploité par le compilateur ou le développeur en utilisant un jeu d'instruction spécifique (Very Long Instruction Word (VLIW)), ou par l'ordonnanceur matériel en gérant les dépendances entre instruction pour conserver un fort taux de remplissage du *pipeline*. La Figure 2.16 illustre les différences entre l'ILP et le TLP.

2.3.2 AMD, de l'architecture R300 à *Graphics Core Next*

Historiquement, AMD a utilisé un jeu d'instruction vectoriel, et a introduit en 2002 avec l'architecture R300 un jeu d'instruction VLIW à deux voies pour accompagner les débuts des calculs dans les *shaders*

Cette architecture s'est révélée efficace pour traiter les problèmes posés par les applications graphiques, jusqu'à l'arrivée de DirectX 10 et des nouvelles fonctionnalités introduites dans les *shaders*.

Jusqu'à ce moment, les *shaders* étaient encore nouveaux et séparés dans des circuits dédiés pour les *shaders* de géométrie et les *shaders* de sommets. Les concepteurs de circuits graphiques avaient conclu qu'un jeu d'instruction VLIW était l'architecture idéale pour les *shaders* de sommets. Ce jeu d'instruction permet le traitement simultané d'une opération Single Instruction stream, Multiple Data streams (SIMD) sur un vecteur à quatre composants (w, x, y, z) et une autre opération sur un composant scalaire séparé, par exemple la luminosité.

Le compilateur joue alors un rôle crucial pour assembler les opérations dans une instruction VLIW en exploitant l'ILP qui peut être trouvée dans un *shaders*. Par opposition, le TLP est géré en répliquant les unités de calcul et en traitant plus de données simultanément. L'ordonnancement statique effecuté par le compilateur simplifie le matériel et permet de dédier une plus grande proportion des transistors au profit des unités de calcul plutôt que pour complexifier l'ordonnanceur matériel.

DirectX 10 unifie la programmation des différents type de *shaders*. Ces changements ont

12. "out-of-order execution"

poussé les concepteurs de circuits graphiques à refléter ces changements dans l'architecture. Les mêmes unités sont alors en charge de l'exécution de tous les types de *shaders*. Pour AMD ce changement est introduit dans le circuit R600. Des nouveaux circuits de logique sont introduits pour prendre en charge l'ordonnancement des différents fils d'exécution en provenance des différents types de *shaders* qui entrent en concurrence pour les ressources. L'introduction d'un ordonnanceur matériel est un point important pour l'évolution de la pratique du GPGPU comme cela s'est confirmé par la suite des évolutions matérielles qui ont proposé des fonctionnalités construites au dessus de cet ordonnanceur.

Les nouveautés introduites dans le langage HLSL par DirectX 10 a conduit les concepteurs des puces AMD à choisir une architecture plus flexible. Le R600 remplace l'architecture VLIW mixte vecteur/scalaire par un jeu d'instructions VLIW à cinq voies purement scalaires. De cette manière, comme précédemment, cinq éléments individuels peuvent être traités à chaque cycle. De cette manière, comme précédemment, cinq éléments peuvent être traités à chaque cycle. Cependant le vecteur a été séparé et au lieu de devoir effectuer la même opération sur quatre éléments il est alors possible maintenant d'exécuter cinq opérations différentes.

L'exécution SIMD dans de tels GPUs est gérée différemment que les extensions CPU telles que Streaming SIMD Extension (SSE) ou Advanced Vector eXtensions (AVX). Les registres des GPUs ne sont pas vectoriels mais de larges zones reconfigurables dynamiquement à souhait. L'exécution SIMD est implicite et gérée par l'ordonnanceur matériel. Les PEs partagent une unité de décodage d'instruction, et tant que les différents fils d'exécution ne divergent pas et continuent d'exécuter la même instruction, tous les PEs sont exploités. En cas de branchement dans le flot de contrôle qui ne serait pas suivi par tous les fils d'exécution, les branches sont exécutées en séquence, et des PEs sont désactivés, réduisant le taux utilisation du GPU.

La série Radeon HD 6900 series, de nom de code Cayman (R900), est livrée en 2010. Cette nouvelle génération évolue vers un jeu d'instruction VLIW de quatre voies, sensé être plus efficace d'après les tests effectués par AMD sur le taux de remplissage VLIW moyen obtenu en pratique par le compilateur. Cette simplification permet d'augmenter le nombre d'unité SIMD qui profite au TLP.

La limitation principale du jeu d'instruction VLIW réside dans la pression mise sur le compilateur chargé d'extraire l'ILP statiquement dans le programme source. Cette limitation est acceptable pour des domaines restreints, tels que les applications graphiques, mais peut handicaper les performances atteignables par la pratique du GPGPU. La Fi-

gure 2.18 montre des statistiques sur le taux d'utilisation des unités de calcul et du taux de remplissage des instructions VLIW.

La dernière génération de circuit AMD, nom de code Graphics Core Next (GCN) (R1000), introduit une évolution majeure. L'architecture VLIW est abandonnée pour une architecture complètement scalaire. Les instructions VLIW à quatre voies sont séparées dans quatre flux d'exécution SIMD. Cela signifie que l'ILP n'est plus exploitée par le compilateur et que les unités de calcul bénéficient implicitement du TLP. Les applications doivent alors fournir un nombre suffisant de fils d'exécution pour utiliser toutes les unités de calcul. Le prix à payer est la présence d'un ordonnanceur matériel plus complexe.

La Figure 2.20 résume les évolutions du groupement des unités de calcul au fur et à mesure de l'évolution des architectures AMD.

2.3.3 Nvidia Computing Unified Device Architecture, from G80 to Kepler

A la fin de l'année 2006, Nvidia livrait le circuit G80. C'était le premier circuit compatible avec DirectX 10. Alors que les architectures précédentes proposées par Nvidia comportaient pour l'essentiel des unités de calculs spécialisées pour un nombre fixe de traitements graphiques, le G80 unifie les étages du flux d'exécution graphique en proposant des unités de calcul flexibles. Comme pour le R600 d'AMD, le changement majeur pour la pratique du GPGPU est que les unités de calcul peuvent traiter indifféremment tous types de *shaders*.

Une autre nouveauté est que les unités de calcul exploitent un jeu d'instruction scalaire, similaire à la solution annoncée par AMD six ans plus tard avec Graphics Core Next (GCN).

Le G80 possède plusieurs CUs composées de huit PEs. Une instruction est décodée par CU tous les deux cycles. Les PEs fonctionnent à une fréquence double du décodage d'instruction et chaque instruction est donc exécutée trente-deux fois (sur des données différentes). L'utilisation du GPU est maximisée quand trente-deux fils d'exécution traitent la même instruction simultanément, le SIMD est implicite. A nouveau le TLP est exploité et l'ILP n'est pas directement exposée par l'architecture mais peut être bénéfique pour masquer la latence mémoire, comme démontré par Volkov [Volkov 2010]. L'ordonnanceur bénéficie d'instructions indépendantes en séquence pour les émettre sans délai dans le flux d'exécution.

Le GT200 succède au G80 en 2008. L'évolution principale du point de vue de la pratique du GPGPU est l'introduction d'unités de calcul flottant à double précision. Une autre nouveauté importante est la présence d'opération atomique en mémoire globale.

En 2010, Nvidia livre une révision majeure de l'architecture avec Fermi. De nombreuses améliorations sont présentes du point de vue de la pratique du GPGPU : le support des

contrôles de flot indirect ouvrant la voie au C++ et aux fonctions virtuelles, une hiérarchie de caches gérés par le matériel sont introduits, les transferts de données sur le bus PCI Express (PCIe) s'effectuent dans les deux sens simultanément, plusieurs noyaux peuvent être ordonnancer simultanément se partageant les ressources matérielles, et l'introduction de la mémoire à correction d'erreur Error-correcting code (ECC).

La Figure 2.22 illustre l'évolution des CUs introduite avec Fermi. Le nombre de PEs est augmenté de huit à trente-deux, et ceux-ci sont séparés en deux flux d'exécution qui possèdent chacun leur ordonnanceur. Une instruction est traitée à chaque cycle pour seize fils d'exécution simultanément, soit la même vue d'une largeur logique SIMD de trente-deux que dans les architectures précédentes.

La dernière révision majeure en date de l'architecture Nvidia a été livrée en mars 2012 et porte le nom de Kepler. Nvidia met l'accent sur le ratio de la capacité de calcul par watt, amélioré en réduisant la fréquence des PEs de moitié de sorte à être identique aux unités d'ordonnancement. Les quatre nouveautés architecturales principales sont les suivantes :

- Le parallélisme dynamique offre la capacité de générer de nouveaux fils d'exécution sur le GPU depuis l'exécution d'un noyaux. Un noyaux peut initier des transferts mémoires entre l'hôte et l'accélérateur, ou lancer l'exécution de nouveaux noyaux. C'est un changement important dans le modèle de programmation existant.
- Le mécanisme de files multiples *Hyper-Q* permet jusqu'à trente-deux fils d'exécution de l'hôte de soumettre des commandes à un accélérateur dans des files indépendantes, permettant un partage plus efficace d'un accélérateur.
- L'unité de gestion de grille est le circuit matériel de base pour activer le parallélisme dynamique. Il remplace l'ordonnanceur précédent en offrant plus de flexibilité dans la répartition, la mise en file, et les dépendances pour jusqu'à 2000 différents contextes d'exécution simultanés.
- GPU Direct est la technologie qui autorise à transférer des données directement entre deux accélérateurs ou entre la mémoire d'un GPU et un autre périphérique connecté au bus PCIe, tel qu'un adaptateur réseau Infiniband par exemple.

2.3.4 Impact sur la génération de code

Dans cette thèse, je ne cible pas une architecture en particulier et je souhaite offrir la possibilité de générer du code qui s'exécutera efficacement sur l'ensemble des architectures introduites précédemment. La question principale est la suivante : à quel point la performance obtenue en transformant un code est portable d'une architecture à une autre ?

Puisque les architectures exploitent différemment l'ILP du TLP, il est difficile de s'attendre à une solution universelle. Extraire plus d'ILP peut se faire au détriment du TLP et donc diminuer le taux d'utilisation d'une architecture scalaire.

Les expériences du Chapitre 7 sur les différentes architectures évaluées confirment que les transformations de code qui améliorent les performances sur une architecture donnée ont l'effet inverse sur une autre architecture.

Un autre problème concerne la capacité de prédiction au moment de la compilation de la performance obtenue par l'une ou l'autre version d'un noyau. Même en se restreignant à une architecture donnée, la problématique est complexe. Par exemple pour un noyau très simple, le seul choix de la manière dont les fils d'exécution seront groupés a un impact important sur les performances obtenues. La Figure 2.23 montre l'impact de ces paramètres d'exécution sur la performance obtenue pour les architectures Nvidia Fermi et AMD Evergreen. Le graphique de gauche indique différentes configurations d'exécution pour un jeu de noyaux. Alors qu'Evergreen n'y semble que peu sensible, Fermi montre une accélération jusqu'à un facteur deux. La comparaison de BinomialOption et Matmul_no_smem montre qu'il n'y pas de choix de taille de groupe universelle. L'impact de ces paramètres est évalué avec plus de détails dans la Section 5.8.

Enfin, la taille des données d'entrée, qui en définit en général le nombre maximum de fils d'exécution qui peuvent être exploités, n'est connue qu'à l'exécution. La Figure 2.24 montre l'évolution des performances en fonction de la taille d'une matrice pour deux noyaux différents réalisant la multiplication de matrices sur Fermi, avec et sans l'activation du cache de niveau 1. La mémoire partagée n'est utilisée dans aucune de ces deux versions. Le fait que le cache de niveau 1 puisse être activé ou désactivé noyau par noyau est un paramètre de plus qui influence le choix de la version du noyau à exécuter pour maximiser les performances atteignables. La Section 5.8 couvre en détail l'impact du choix des configurations d'exécution sur les performances, et la Section 7.4 fournit des résultats expérimentaux.

2.3.5 Résumé

Les GPUs sont des circuits hautement parallèles. Ils reposent principalement sur le TLP pour exploiter le parallélisme pour des milliers de fils d'exécution. Cependant, selon l'architecture, l'ILP peut également être nécessaire pour s'approcher des performances maximales théoriques.

Pour plus d'informations sur les mystères architecturaux des GPUs, tels que la latence par opération, la compréhension profonde du fonctionnement des caches, etc., le lecteur est renvoyé aux travaux de Taylor et Li sur le test de l'architecture AMD [Taylor & Li 2010],

les travaux de Wong et al. [Wong *et al.* 2010] et Collange [Collange 2010b] pour le GT200, et Lindholm et al. [Lindholm *et al.* 2008] et la thèse de Collange [Collange 2010a] pour Fermi.

2.4 Conclusion

Il y a une décennie, la pratique du GPGPU n'en était qu'à ses débuts. Depuis elle a été l'objet d'intenses recherches et est toujours un domaine très actif, comme illustré sur la Figure 2.25. Ce chapitre a montré que plusieurs langages, cadres de programmation, et autres solutions ont été proposés et expérimentés pour aider les développeurs à écrire des programmes qui exploitent les accélérateurs. Toutes ces approches se basent sur différents compromis des propriétés *trois P*s : Performance, Portabilité, Programmabilité. La performance portabilité est un challenge dans le contexte de la pratique du GPGPU. La programmabilité a été traitée par divers langages spécialisés.

Le paysage en 2012 est très différent de ce qu'il était il y a dix ans quand les chercheurs contournaient le flux d'exécution purement graphique d'OpenGL pour traiter des opérations mathématiques abstraites [Trendall & Stewart 2000].

La programmabilité était un défi. Toutefois, quand un programme était converti vers l'interface OpenGL, les performances et la portabilité étaient garantis. L'introduction des *shaders* en 2002 a apporté de la flexibilité et offert plus de fonctionnalités et amélioré la programmabilité.

Des approches plus récentes tel que le langage Brooks ont tenté de faire un compromis sur les performances au profit de la programmabilité. L'interface de programmation par flux répond à certains besoins des développeurs, mais n'est pas suffisamment flexible pour se généraliser.

L'évolution de DirectX a conduit les fabricants de GPUs vers plus de programmabilité. Cependant le point clé dans la démocratisation de la pratique du GPGPU a été l'introduction de CUDA, suivi ensuite par des langages alternatifs et des cadres de programmation.

Les modèles de programmation CUDA et OpenCL bénéficient de l'expérience des *shaders* pour fournir un niveau équivalent de programmabilité mais en s'affranchissant des mécanismes rigides et inadaptés imposés par l'utilisation des flux de traitement graphiques d'OpenGL et Direct X. Toutefois les développeurs doivent connaître l'architecture pour écrire des noyaux de calcul efficaces : la performance est favorisée au détriment de la portabilité .

Les langages à base de directives tels que hiCUDA, JCUDA, HMPP, PGI Accelerator, ou OpenACC (voir Section 2.2.10) sont moins intrusifs et offrent une bonne portabilité au

prix de la performance. Les directives peuvent être spécialisées pour une cible donnée pour favoriser la performance, mais en sacrifiant la portabilité.

Mon travail de doctorat a démarré juste après l'annonce de la première version du standard OpenCL au printemps 2009 [Ramey 2009]. Le but de ce travail était de fournir une solution de bout en bout qui évite aux développeurs l'adaptation manuelle de leurs codes pour les accélérateurs graphiques.

La programmabilité est aussi bonne que possible, puisque les développeurs écrivent leurs codes en utilisant des langages de programmation séquentiels et en ignorant la présence de l'accélérateur. Le compilateur extrait le parallélisme et le code à exécuter sur l'accélérateur. La performance peut ne pas atteindre ce qu'un expert pourrait obtenir avec des efforts. Toutefois le compromis sur la performance est acceptable si la différence avec le code produit par un expert pour une architecture donnée est limitée, par exemple dix, vingt, ou trente pour cent selon les domaines d'application.

Très peu de travaux ont porté sur des chaînes entièrement automatiques de parallélisation automatique et de transformation (voir Section 2.2.11 page 30) d'un code séquentiel vers du code GPU. La plupart des propositions sont d'applications limitées ou se concentre seulement sur une partie du problème. Mon travail tente de traiter une application complète, de générer des noyaux, de les optimiser, et de générer les communications requises, sans impliquer l'utilisateur dans le processus de décision. Il a été montré à la Section 2.4.6 de quelle manière l'évolution des architectures des GPUs impacte la génération de code.

Guelton propose dans sa thèse [Guelton 2011a] une vue de haut-niveau d'un compilateur hétérogène pour cibler des accélérateurs. Le compilateur transforme le code, sépare le code hôte et le code des noyaux, avec la *colle* nécessaire pour assembler les morceaux. Chaque partie est ensuite compilée par le compilateur binaire de la cible. La Figure 2.26 illustre cette vue de haut-niveau.

Mon travail propose d'instancier cette vue de haut-niveau dans un compilateur pour GPUss. Un aperçu de la structure du compilateur proposé est présenté dans la Figure 2.27, qui distingue mes contributions. Sans explorer tous les aspects de chaque élément de la chaîne, cette thèse fournit des solutions à plusieurs problématiques liées à la parallélisation automatique pour GPUs, allant de la détection du parallélisme à la génération de code, en passant par les optimisations de nids de boucles et la gestion du placement des données.

La Figure 2.27 illustre cette chaîne de compilation. Le code source est d'abord analysé pour trouver le parallélisme, puis transformé, avant d'extraire le code à exécuter sur le GPU dans de nouvelles fonctions : les noyaux. Certaines phases d'optimisation peuvent

être appliquées, telle que la fusion de boucles, le remplacement de tableaux par des scalaires, la linéarisation des accès au tableaux, le tuilage symbolique, ou encore le déroulage de boucles. Cette partie du processus est présentée au Chapitre 4. Après la génération des noyaux, des analyses et des transformations pour générer les communications sont nécessaires. Les régions convexes de tableaux sont utilisées dans le Chapitre 3 pour générer des communications efficaces. Une analyse interprocédurale statique est proposée pour optimiser les communications en laissant les données autant que possible sur le GPU. Une autre voie est la génération de tâches qui sont ordonnancées à l'exécution sur plusieurs accélérateurs à l'aide du support exécutif *StarPU*. L'extraction de tâches et la génération de code pour *StarPU* sont présentées dans le Chapitre 6, tout comme une autre méthode basée sur du tuilage symbolique. Le processus complet est piloté par Par4All, depuis le code source d'entrée jusqu'au binaire final. Par4All est basé sur un gestionnaire de phases de compilation flexibles. Le défi d'automatiser un processus complet est présenté dans le Chapitre 5. Les résultats expérimentaux sont présentés et discutés dans le Chapitre 7.

3 Placement des données, communications, et cohérence

Les accélérateurs tels que les GPUs traitent des données placées dans leur mémoire embarquée. En effet un circuit graphique est accompagné d'une mémoire dédiée à haut-débit de plusieurs giga-octets, comme exposé à la Section 2.4. La difficulté est que cette mémoire embarquée n'est pas directement accessible depuis l'hôte et réciproquement la mémoire de l'hôte n'est pas directement accessible depuis l'accélérateur. Les développeurs doivent explicitement transférer les données à traiter dans la mémoire de l'accélérateur avant d'exécuter un noyau, et transférer dans le sens opposé les données résultant du traitement sur l'accélérateur.

Ces communications explicites utilisent des bus à la bande passante limitée. Par exemple, le bus PCIe 2.0 offre une bande passante théorique maximale de 8 GB/s, ce qui reste peu comparé à la bande passante de la mémoire embarquée des GPUs de l'ordre de centaines de GB/s. Cette limitation est régulièrement considérée comme le goulet d'étranglement pour la programmation hybride [Chen *et al.* 2010].

Des travaux ont été menés pour répondre à cette problématique, soit en utilisant des informations fournies par le développeur [Yan *et al.* 2009, CAPS Entreprise 2010, Wolfe 2011, NVIDIA, Cray, PGI, CAPS 2011], soit automatiquement [Amini *et al.* 2011c (perso), Ventroux *et al.* 2012, Guelton 2011a, Alias *et al.* 2011, Wolfe 2010] par des analyses de compila-

tion. Une méthode *paresseuse* a également été proposée par Enmyren et Kessler [Enmyren & Kessler 2010] dans la bibliothèque de squelette C++ SkePU.

Ce chapitre étudie la problématique de la génération des communications dans le contexte de la parallélisation automatique ou semi-automatique et présente plusieurs contributions pour résoudre ce problème : les régions convexes de tableau sont exploitées pour optimiser la quantité de données à transférer par noyau, ainsi qu'une méthode de placement statique basée sur une analyse interprocédurale.

Paralléliseur Interprocédural de Programmes Scientifiques (PIPS) est un cadre de compilation développé depuis plus de vingt ans [Irigoin *et al.* 1991, Amini *et al.* 2011a (perso)] qui comporte un ensemble de phases d'analyse sémantique et de phases de transformation de code. Initialement traitant le langage Fortan 77, il a été étendu pour gérer le langage C. PIPS génère un code source modifié, aussi proche que possible du code d'entrée.

D'abord, le type de programmes ciblés est présenté avec une étude de cas : une simulation cosmologique. Ensuite l'abstraction des régions de tableaux convexes, qui est à la base de la plupart des transformations, est présentée à la Section 3.2. Le placement le plus simple est ensuite décrit à la Section 3.3 pour introduire les mécanismes mis en jeu. Les régions de tableaux convexes sont ensuite utilisées à la Section 3.4 pour améliorer le résultat de la section précédente. Les limites de l'approche sont exposées à la Section 3.5. Une optimisation interprocédurale est proposée Section 3.6 pour placer efficacement les données sur un accélérateur et limiter le nombre de transfers.

La promotion parallèle de code séquentiel présentée à la Section 3.7 peut aider en complément de la fusion de boucles pour réduire les synchronisations entre deux noyaux mais également en éliminant certains transferts mémoires.

Finalement, les travaux liés à l'optimisation de communication pour la pratique du GPGPU sont présentés Section 3.8.

3.1 Étude de cas

Les jeux d'essai tels que ceux utilisés dans la suite Polybench [Pouchet 2011] par exemple, sont limités à quelques noyaux en séquence, de temps en temps encapsulés dans une boucle de temps. Par conséquent, s'ils sont adaptés à l'étude la performance brute de l'exécution d'un noyau sur GPU, ils ne peuvent être considérés comme représentatifs d'une application complète lorsqu'il s'agit d'évaluer la problématique du placement des données entre hôte et accélérateur.

L'application présentée dans cette section est plus représentative des simulations numé-

riques. C'est un code réel de simulation cosmologique nommé Stars-PM et dont le résultat du traitement est illustré sur la Figure 3.1. La version séquentielle a été écrite en langage C à l'Observatoire Astronomique de Strasbourg, puis par la suite portée et optimisée à la main en utilisant CUDA pour exploiter des GPUs [Aubert *et al.* 2009 (perso)].

Cette simulation modélise les interactions gravitationnelles entre particules dans l'espace. Elle fait appel à une discrétisation de l'espace tri-dimensionnel sur une grille discrète sur laquelle les particules évoluent. Les conditions initiales sont lues depuis un fichier, puis une boucle séquentielle itère sur des pas de temps successifs. A chaque itération, le cœur de la simulation est traité. Les résultats sont extraits de l'état final de la grille et stockés dans un fichier de sortie. Cette organisation générale est illustrée sur la Figure 3.2. C'est un schéma général assez commun dans les simulations numériques, alors que le cœur de chaque itération peut varier fortement d'un domaine à l'autre. Les étapes effectuées à l'intérieur d'une itération dans le cas de Stars-PM sont illustrées sur la Figure 3.3.

3.2 Analyses de régions de tableau

Les régions de tableaux sont à la base de plusieurs transformations qui sont mises en œuvre dans la chaîne de compilation pour cibler des accélérateurs. Cette section fournit une introduction basique pour cette abstraction. Trois exemples sont utilisés pour illustrer l'approche : le code de la Figure 3.4 fait appel à une propagation interprocédurale, le code de la Figure 3.5 contient une boucle `while`, pour laquelle le motif d'accès à la mémoire requiert une analyse approximée, et le code de la Figure 3.6 met en œuvre une construction *switch-case* élaborée.

De manière informelle, les régions lues "READ" (resp. écrite "WRITE") pour une instruction i représentent l'ensemble des variables scalaires et des éléments de tableaux qui sont lus (resp. écrits) lors de l'exécution de i. Cet ensemble dépend généralement des valeurs de certaines variables du programme avant l'exécution de i : les régions READ sont définies comme une fonction de l'état mémoire σ qui précède l'exécution de l'instruction, et sont dénotées $\mathcal{R}(s, \sigma)$ (resp. $\mathcal{W}(s, \sigma)$ pour la région WRITE).

Par exemple pour les régions READ de l'instruction à la line 6 dans la Figure 3.4 :

$$\mathcal{R}(s, \sigma) = \{\{\mathtt{v}\}, \{\mathtt{i}\}, \{\mathtt{j}\}, \{\mathtt{src}(\phi_1) \mid \phi_1 = \sigma(\mathtt{i}) + \sigma(\mathtt{j})\}, \{\mathtt{m}(\phi_1) \mid \phi_1 = \sigma(\mathtt{j})\}\}$$

où ϕ_x est utilisé pour décrire les contraintes sur la $x$ième dimension d'un tableau, et où

$\sigma(\texttt{i})$ dénote la valeur de la variable `i` dans l'état mémoire σ.

3.3 Processus basique de transformation

Le processus le plus basique pour placer les données sur l'accélérateur consiste à envoyer vers l'accélérateur tous les tableaux référencés dans un noyau avant de l'exécuter. Le même ensemble de tableaux doit être transféré depuis l'accélérateur à la fin de l'exécution du noyau. Ce processus basique, le plus simple qui puisse être mis en œuvre par des outils automatiques, est représenté Figure 3.7.

La problématique principale survient quand il est nécessaire de compter le nombre d'éléments de tableau à transférer. Selon le langage, l'information peut être délicate à trouver. Leung et al. [Leung *et al.* 2009] et JCUDA [Yan *et al.* 2009] ciblent Java et bénéficient d'informations à l'exécution sur la taille des tableaux. D'autres comme Verdoolaege and Grosser [Verdoolaege *et al.* 2013] traitent le langage C mais sont limités aux tableaux dont la taille est connue à la compilation. Les algorithmes utilisés par les outils propriétaires tels que R-Stream, HMPP, ou PGI Accelerator sont inconnus, mais ils semblent suivre le même principe.

L'outil proposé avec cette thèse, Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] (présenté en détail Section 5.1), se base sur le même principe dans sa version la plus basique, en étant toutefois un peu plus souple en acceptant les tableaux C99 "Variable Length Array (VLA)". La taille concrète ne sera connue qu'à l'exécution mais l'information est utilisée de manière symbolique lors de la compilation.

3.4 Raffinement à l'aide de régions convexes

Cette section présente un raffinement du modèle basique basé sur les déclarations des tableaux introduit dans la section précédente. Les régions convexes de tableaux introduites à la section Section 3.2 sont mises en œuvre. Le principe sous-jacent, l'isolation d'instructions, a été décrit de manière formelle dans la thèse de Guelton's [Guelton 2011b].

Pour illustrer comment les régions de tableau sont exploitées, l'exemple de la boucle `while` sur la Figure 3.5 illustre le résultat du calcul des régions sur-approximées :

$$\mathcal{R} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{R}}(\mathrm{randv}) = \{\mathrm{randv}[\phi_1] \mid N - 3 \leq 4 \times \phi_1; 3 \times \phi_1 \leq N\}$$
$$\mathcal{W} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{W}}(\mathrm{a}) = \{\mathrm{a}[\phi_1] \mid N - 3 \leq 4 \times \phi_1; 12 \times \phi_1 \leq 5 \times N + 9\}$$

L'idée de base est de se servir des régions pour créer une nouvelle allocation suffisamment grande, puis de générer les transferts mémoires depuis la zone mémoire d'origine vers cette

nouvelle allocation, et enfin d'effectuer la copie de cette allocation vers la zone d'origine. La Figure 3.10 illustre le résultat de ce traitement, dans lequel les variables isolées sont en majuscules. Les instructions `(3)` et `(5)` correspondent à la région exacte sur les variables scalaires. Les instructions `(2)` et `(4)` correspondent aux régions sur-approximées sur les tableaux. L'instruction `(1)` est utilisée pour s'assurer de la cohérence des données, comme expliqué dans la suite.

Il est intéressant de noter comment les appels systèmes `memcpy` sont utilisés ici pour simuler les transferts de données dans un nouvel espace mémoire, et en particulier comment la taille des transferts est contrainte d'après les régions de tableau.

3.5 Limites

L'échange des données entre l'hôte et l'accélérateur sont effectués sous forme de transferts Direct Memory Access (DMA) entre mémoires à travers du bus PCI Express, qui offre actuellement une bande passante théorique de 8 GB/s. C'est très lent face à la bande passante de la mémoire embarquée du GPU, qui dépasse souvent 150 GB/s. Cette bande passante faible peut annihiler tous les gains obtenus lors du déport de noyaux.

Le matériel que nous avons utilisé (voir Section 7.1) nous permet de mesurer jusqu'à 5.6 GB/s lors des transferts de l'hôte vers le GPU, et 6.2 GB/s dans l'autre sens. Cette bande passante est obtenue pour des morceaux de plusieurs MB, mais diminue pour des morceaux plus petits. De plus cette bande passante maximale est réduite de moitié environ quand les zones mémoires sur l'hôte ne sont pas allouées de manière *épinglée* ; c'est à dire sujet à la gestion de la mémoire virtuelle paginée par le système d'exploitation, la Figure 3.14 illustre l'évolution de la bande passante en fonction de la taille de la zone mémoire à transférer et de l'allocation des données.

3.6 Algorithme d'optimisation des communications

Beaucoup de travaux ont été fait au sujet des optimisations de communications pour les ordinateurs distribués, tels que la fusion de message dans le contexte des programmes unique à mémoires distribuées (Single Program Distributed Data (SPDD)) [Gerndt & Zima 1990], des analyses de flots de données basées sur des régions de tableau pour éliminer les communications redondantes et masquer les communications par des calculs [Gong *et al.* 1993], et la distribution dans le contexte de la compilation High Performance Fortran (HPF) [Coelho 1996, Coelho & Ancourt 1996].

Des méthodes similaires sont appliquées dans cette section au déport des calculs dans le contexte d'une relation hôte-accélérateur et pour intégrer dans un compilateur parallélisant

une transformation qui limite la quantité de communication CPU–GPU à la compilation.

Cette section introduit une analyse de flot de données conçue pour améliorer la génération statique des transferts mémoires entre l'hôte et l'accélérateur. Une hypothèse de base dans cette section est que la mémoire du GPU est suffisamment grande pour contenir l'ensemble des tableaux à traiter. Alors que cette supposition peut sembler une contrainte inacceptable pour certaines applications, les mémoires des GPUs modernes de plusieurs giga-octets sont suffisamment importantes pour une large gamme de simulations.

3.7 Promotion de code séquentiel

Des nids de boucles parallèles peuvent être séparés par du code séquentiel. Quand ce code séquentiel utilise ou produit des données utilisées ou produites par les noyaux avant et après, des transferts entre la mémoire de l'hôte et celle de l'accélérateur sont nécessaires.

Une solution à ce problème est de promouvoir ce code séquentiel sur un unique fil d'exécution sur l'accélérateur. Alors que l'exécution d'un unique fil d'exécution sur le GPU est inefficace, la perte peut être bien moindre que le coût de la communication. La problématique de la profitabilité de cette transformation rejoint celle de la profitabilité de l'exécution d'un noyau sur un accélérateur, discutée plus spécifiquement à la Section 5.6 page 156.

L'exemple `gramschmidt` mentionné dans la section précédente est illustré par la Figure 3.17. Les codes générés avec et sans promotion séquentielle illustrent comment le compromis réalisé sur la performance du code séquentiel peut réduire les communications.

La Section 7.8.4, page 201, montre nos mesures qui indiquent une accélération jusqu'à huit fois pour l'exemple `gramschmidt`, mais aussi jusqu'à trente-sept fois pour l'exemple `durbin` de la suite Polybench.

En revanche, en cas de mauvais évaluation de la profitabilité, les performances peuvent chuter drastiquement. Cette transformation requiert une évaluation précise de l'exécution de chacune des versions. Une manière de résoudre ce problème est d'effectuer une mesure *à froid* avec un échantillon d'une unique itération de la boucle séquentielle sur le GPU et de retarder la décision au moment de l'exécution. Cette approche est explorée à la Section 5.7, page 158, serait toutefois difficile à combiner avec la méthode d'optimisation de communication introduite dans ce chapitre.

3.8 Résultats expérimentaux

La Section 7.8, page 197, présente des résultats expérimentaux pour la méthode de limitation des communications introduite dans ce chapitre.

La première question est : que devons-nous mesurer ? Alors que l'accélération est une métrique très efficace, commercialement parlant, c'est une mesure biaisée dans ce contexte car elle très impactée par les paramètres d'entrée (voir la Section 7.8.1, page 197). Un même jeu d'essai peut obtenir une accélération allant de 1.4 à quatorze juste en changeant les paramètres d'entrée.

Une mesure plus objective pour évaluer la méthode proposée est le nombre de communications supprimées et la comparaison avec le placement produit par un programmeur expert. Se concentrer sur l'accélération obtenue aurait aussi l'inconvénient de mettre en avant les capacités du générateur de code noyau.

La Section 7.8.2, page 199, illustre la performance de la méthode de réduction des communications en utilisant cette métrique sur les jeux d'essai des suites Polybench 2.0 et Rodinia, ainsi que sur la simulation numérique Stars-PM introduite à la Section 3.1. La méthode de réduction obtient un résultat proche de ce qui serait écrit manuellement.

Une exception remarquable est l'exemple de `gramschmidt`. Les communications ne peuvent être déplacées hors de la boucle en raison de dépendances introduites par du code séquentiel. La méthode de promotion parallèle présentée à la Section 3.7 apporte une solution en acceptant un code plus lent mais en permettant de laisser les données sur l'accélérateur.

La Section 7.8.3, page 199, explore l'impact sur la performance de retarder la décision à l'exécution en utilisant la bibliothèque StarPU ; le code généré statiquement est jusqu'à cinq fois plus rapide. Bien que StarPU présente des fonctionnalités bien plus larges que la seule problématique de l'optimisation des communications, le placement statique est intéressant.

3.9 Travaux liés

Parmi les compilateurs évalués, aucun n'implémente une analyse interprocédurale de placement automatique des communications telle que celle proposée ici. Alors que Lee et al. traitent cette problématique [Lee *et al.* 2009, § 4.2.3], leur travail est limité à l'analyse des données vivantes et donc assez similaire à la méthode proposée à la Section 3.3. Leung traite le cas des boucles séquentielle autour d'un lancement de noyau et sort les communications de la boucle [Leung 2008].

La méthode d'optimisation proposée dans ce chapitre est indépendant de la méthode de parallélisation, et est applicable à des systèmes qui transforment un code OpenMP en CUDA ou OpenCL comme OMPCUDA [Ohshima *et al.* 2010] ou "OpenMP to GPU" [Lee *et al.* 2009]. Elle est également applicable pour les approches à base de directives, telle que

JCUDA et hiCUDA [Han & Abdelrahman 2009]. Elle pourrait aussi compléter les travaux effectués sur OpenMPC [Lee & Eigenmann 2010] en déplaçant les communications dans le graphe d'appels.

Cette approche peut-être comparée à l'algorithme proposé par Alias et al. [Alias *et al.* 2011, Alias *et al.* 2012b, Alias *et al.* 2012a], toutefois à un niveau de granularité différent.

Dans un article récent [Jablin *et al.* 2011], Jablin et al. introduisent CGCM, un système qui vise exactement la même problématique. CGCM, tout comme la méthode proposée ici, est focalisée sur le transfert d'allocation mémoire entière. Alors que le processus ici est entièrement statique, CGCM prend des décisions à l'exécution. Le support exécutif gère les pointeurs en général, y compris ceux qui renvoient sur le milieu d'une structure allouée sur le tas, ce qui n'est pas géré par la méthode que je propose.

3.10 Optimisation d'un nid de boucles tuilées

Alias et al. ont publié des travaux intéressants sur l'optimisation fine des communications dans le contexte des FPGAs [Alias *et al.* 2011, Alias *et al.* 2012b, Alias *et al.* 2012a]. Le fait qu'ils ciblent des FPGAs change des considérations sur la taille de la mémoire : les FPGAs embarquent en général très peu de mémoire en comparaison des GPUs. La proposition d'Alias et al. se focalise sur l'optimisation des chargements depuis la mémoire Double Data Rate (DDR) dans le cas d'un nid de boucles tuilées, pour lequel le tuilage est effectué de sorte que les tuiles s'exécutent en séquence sur l'accélérateur alors que les itérations à l'intérieur des tuiles peuvent être parallélisées.

Alors que leur travaux sont basés sur l'abstraction sous forme de Quasi-Affine Selection Tree (QUAST), cette section montre comment leur algorithme peut être utilisé avec les régions de tableaux convexes.

3.11 Conclusion

Avec l'augmentation de l'utilisation des accélérateurs matériels, la parallélisation automatique ou semi-automatique assistée par des directives prend une importance croissante.

L'impact des communications est critique quand on cible des accélérateurs pour des programmes massivement parallèles telles que les simulations numériques. L'optimisation du placement des données est donc clé pour l'obtention de performance.

Un schéma d'optimisation qui traite cette problématique a été conçu et implémenté dans PIPS et dans Par4All.

L'approche proposée a été validée à l'aide de vingt jeux d'essai de la suite Polybench,

trois de la suite Rodinia, et une simulation numérique réelle. La méthode proposée donne des résultats proche d'un placement manuel en terme de nombre de communications effectuées.

Le prochain Chapitre présente les différentes transformations effectuées sur le code séquentiel pour obtenir un code GPU.

4 Transformations pour GPGPU

Les contributions de ce chapitre se basent sur les analyses et transformations existantes dans l'environnement de compilation PIPS, étendant certaines pour gérer le langage C, améliorant d'autres pour les besoins spécifiques de la génération de code pour GPU, et finalement en introduisant de nouvelles transformations quand PIPS ne proposait pas de solution adaptée.

4.1 Introduction

Le chemin qui mène du code séquentiel vers du code parallèle efficace pour GPU est pavé de nombreuses analyses et transformations. De plus, certaines spécificités des programmes d'entrée doivent être prises en compte. Par exemple, des programmes écrits par un développeur ne présentent pas les mêmes motifs inclus dans des programmes automatiquement générés depuis des outils ou langages de conception de haut-niveau. Le code de la Figure 4.1 illustre de quelle manière un script de trois lignes de Scilab aboutit à un code C avec des tableaux temporaires et cinq nids de boucles.

Le flot de compilation complet pour transformer un code séquentiel en code GPU est présenté à la Figure 2.27 page 58 et identifie les contributions présentées dans ce chapitre.

La Section 4.2 explique la manière dont un nid de boucles est placé sur GPU, ainsi que la manière dont le tuilage implicite et l'échange de boucles sont mis en œuvre à l'exécution (voir la Section 2.3.2.2, page 32).

J'ai conçu et implémenté une nouvelle transformation de substitution de variables d'induction basée sur l'analyse de préconditions linéaires, présentée à la Section 4.5. Cette transformation peut permettre la parallélisation des boucles qui contiennent des variables d'induction.

J'ai étudié la combinaison de deux algorithmes de parallélisation, en analysant l'impact sur la génération de code pour chacun d'entre eux à la Section 4.3.

J'ai amélioré la phase de détection de réductions existante dans PIPS pour traiter le langage C plus précisément, et j'ai exploité cette analyse pour permettre la parallélisation

des boucles avec réduction en améliorant les algorithmes de parallélisation pour s'appuyer sur la détection des réductions et les capacités matérielles des GPUs. La génération de code avec instructions atomiques est présentée à la Section 4.4, page 105.

J'ai implémenté deux transformations de fusion de boucles. La première est basée sur le calcul d'un graphe de dépendance, et la seconde sur les régions de tableaux. J'ai mis en œuvre des heuristiques pour piloter la fusion dans le contexte des GPUs. La fusion de boucles est particulièrement critique quand le code à traiter est généré depuis une représentation de plus haut-niveau, telle que Scilab ou Matlab. Cette transformation peut permettre d'éliminer des tableaux temporaires dans le code généré.

J'ai étudié différents schémas de transformation des tableaux en scalaire dans le contexte de la génération de code pour GPUs, à la Section 4.7, page 127, et j'ai modifié l'implémentation de PIPS pour répondre aux spécificités de la génération de code pour GPUs, particulièrement l'encapsulation parfaite des nids de boucles.

Finalement la Section 4.10 résume les contributions de ce chapitre et introduit la manière dont elles sont assemblées dans le chapitre suivant pour former une chaîne de compilation complète.

4.2 Mise en correspondance de l'espace d'itération d'un nid de boucles sur GPU

Exécuter un nid de boucles parallèles sur un GPU en utilisant CUDA ou OpenCL requiert une association précise de l'espace d'itération du nid de boucles vers l'abstraction de fils d'exécution sur GPU utilisant un *NDRange* (voir la Section 2.3.2.2 page 32).

Les travaux liés précédents [Baghdadi *et al.* 2010, Lee *et al.* 2009, Baskaran *et al.* 2010] représentent la hiérarchie d'exécution complète dans le compilateur et essayent d'exprimer cette hiérarchie en utilisant des boucles imbriquées. Les transformations effectuées sont principalement du tuilage à plusieurs niveaux avec des restructurations incluant l'échange de boucles et la séparation de l'espace d'itération (index set splitting). La Figure 4.2 montre la manièrre donc un nid de boucles est tuilé pour correspondre au modèle d'exécution à deux niveaux sur GPU.

L'approche implémentée dans le compilateur Par4All [Amini *et al.* 2012b (perso)] est assez différente et ne fait pas appel à un tuilage explicite, mais à la place le code généré par PIPS présente un modèle *à plat* et est ensuite spécialisé par un après-traitement. Considérons que les boucles ont été normalisées, c'est à dire qu'elles sont indexées en commençant à zéro et avec un incrément de un, ce qui est nécessaire pour exprimer l'espace d'itération

en utilisant le concept de NDRange d'OpenCL introduit à la Section 2.3.2.2. La Figure 4.3 indique les quatre étapes impliquées dans cette transformation. D'abord, le corps du nid de boucles initial dans la Figure 4.3 est détouré dans une nouvelle fonction, le noyau, qui sera exécuté dans chaque fil d'exécution sur le GPU. Les indices de boucles sont reconstruits dans le noyaux à l'aide de deux macros `P4A_vp_`x pour chaque dimension. L'exécution séquentielle est effectuée avec une simple expansion de ces macros `#define P4A_vp_1 ti` et `#define P4A_vp_0 tj`. Le nid de boucles parallèles est alors annoté avec l'espace d'itération, comme illustré par la Figure 4.3c.

Finalement, une étape d'après-traitement remplace l'annotation et contracte le nid de boucles dans un pseudo-appel à une macro `Call_kernel_`x`d()` avec la dimension x allant de un à trois, pour respecter les contraintes du NDRange OpenCL. Le résultat de la contraction est montré sur la Figure 4.3d. Cette macro masque l'exécution parallèle du noyau sur une grille de $l \times m \times n$ fils d'exécution. La taille de groupement des fils d'exécution sur la grille n'est pas exprimée explicitement est n'est jamais manipulée dans la représentation interne du compilateur, mais construite dynamique à l'exécution selon différents paramètres.

Cette représentation permet de retarder certaines décisions sur des transformations, et donc conserve un code plus indépendant de la cible tout en simplifiant la représentation interne du compilateur.

4.3 Détection du parallélisme

La détection du parallélisme est à la base du processus. Cela consiste à prouver qu'une boucle peut être ordonnancé de manière parallèle. De telles techniques sont bien connues dans la communauté de compilation, au moins depuis la méthode des hyperplans proposée par Lamport en 1974 [Lamport 1974].

4.3.1 Allen et Kennedy

L'algorithme d'Allen et Kennedy est basé sur un graphe de dépendance à niveaux. Cet algorithme a été prouvé optimal par Darte et Vivien [Darte & Vivien 1996b] pour une telle abstraction de dépendances. Cet algorithme a été conçu pour les machines vectorielles, et donc dans sa version basique distribue les boucles autant que possible et maximise le parallélisme.

Un exemple qui illustre le traitement de l'implémentation dans PIPS de l'algorithme d'Allen et Kennedy est présenté à la Figure 4.5. La distribution de boucles expose le maximum de parallélisme mais augmente le nombre de barrières implicites. De plus les op-

portunités de réutilisation temporelle du cache et de contraction de tableaux sont moindres, comme illustré sur les Figures 4.5a et 4.5b. Ces inconvénients de la distribution de boucles peuvent être limités par une transformation de fusion de boucles, telle que présentée à la Section 4.6.

Une autre limitation de cet algorithme réside dans les limitations imposées sur le flot de contrôle du code à traiter, par exemple le corps de boucles ne peut contenir de branchement. L'algorithme introduit à la prochaine section fournit une solution à ce problème en fournissant une parallélisation à gros grain basé sur un résumé de régions de tableaux [Creusillet & Irigoin 1996b].

4.3.2 Parallélisation à gros grain

Le deuxième algorithme de parallélisation est une méthode à gros grain qui repose sur les analyses de régions de tableaux [Creusillet & Irigoin 1996b]. Aucune transformation de boucles n'est mise en œuvre. Les détails au sujet de cette méthode de parallélisation ont été publiés dans [Irigoin *et al.* 2011 (perso)]. Le processus est résumé dans cette section.

Cet algorithme est utilisé de manière extensive dans PIPS en raison de sa complémentarité avec l'algorithme d'Allen et Kennedy. En effet les boucles ne sont pas distribuées, et cet algorithme ne comporte pas de restrictions sur le flot de contrôle.

4.3.3 Impact sur la génération de code

Comme montré précédemment, il y a deux algorithmes de parallélisation implémentés dans PIPS. La Figure 4.6 illustre l'impact de l'utilisation d'un algorithme par rapport à l'autre. Alors que Allen et Kennedy distribuent le nid de boucles d'origine dans trois nids de boucles différentes parfaitement imbriquées exprimant un parallélisme à deux dimensions. L'algorithme de parallélisation à gros grain conserve l'imbrication d'origine et détecte la dimension externe comme parallèle. De plus, la dimension parallèle est à l'intérieur d'une boucle séquentielle, ce qui implique `m` lancements de noyaux.

La Section 7.3 présente des expériences sur les algorithmes de parallélisation et montre que la méthode d'Allen et Kennedy aboutit à un code plus efficace sur toutes les architectures testées par rapport à la parallélisation à gros grain. Alors que l'accélération est limitée sur les architectures anciennes tel que le G80, elle est notable sur les GPUs récents et atteint jusqu'à huit sur Fermi et quatre sur Kepler.

4.4 Parallélisation des réductions

PIPS inclut un algorithme pour la détection des réductions basé sur un cadre introduit par Jouvelot et Dehbonei [Jouvelot & Dehbonei 1989]. Après détection les réductions

peuvent être parallélisées selon les capacités de l'architecture cible.

4.4.1 Détection

L'algorithme est interprocédural et requiert un résumé pour les fonctions appelées pour traiter une fonction. De manière intraprocédurale, l'algorithme détecte les réductions dans les suites d'instructions telle que :

```
//  call sum[s[a]],sum[b],
    s[a] = s[a]+b++;
```

dans laquelle on observe le commentaire ajouté par PIPS qui indique que deux réductions ont été détectées.

4.4.2 Parallélisation des réductions pour GPU

La parallélisation des boucles avec réductions peut être être réalisées de différentes façons. J'ai conçu et implémenté une nouvelle méthode qui s'insère dans les algorithmes de parallélisation. L'implémentation dans l'algorithme de parallélisation à gros grain présenté à la Section 4.3.2.

La parallélisation à gros grain utilise l'analyse de région de tableaux pour trouver les conflits entre deux itérations d'une boucle. De tels conflits empêchent la parallélisation de la boucle. L'algorithme a été adapté pour gérer les réductions en ignorant les conflits liés aux références impliquées dans la réduction. Si ignorer un conflit élimine tous les cycles, la boucle est alors marquée comme *potentiellement* parallèle. Une autre transformation peut potentiellement remplacer la réduction avec une opération équivalente qui permettra l'exécution parallèle de la boucle.

Pour les GPUs, un moyen de paralléliser les boucles qui comportent les réductions est de faire usage des opérations atomiques présentées à la Section 2.4.3, page 42. Si la cible est compatible avec les opérations atomiques nécessaire, alors la substitution est faite et la boucle est déclarée parallèle pour une future extraction en noyau. La Figure 4.8 contient un exemple de code séquentiel d'histogramme, ainsi que le code résultant après détection des réductions et remplacement par des opérations atomiques.

4.5 Substitution des variables d'induction

La substitution des variables d'induction est une transformation classique qui permet la parallélisation de certaines boucles. Les variables d'induction sont en général détectées par des boucles qui utilisent de la reconnaissance de motifs basée sur l'initialisation et les mises à jour d'une variable dans le corps de boucles. Cette section indique comment les

analyses de préconditions linéaires dans PIPS [Irigoin *et al.* 2011] sont utilisées pour définir un nouvel algorithme pour détecter et remplacer les variables d'induction. Considérant une boucle L, l'algorithme traite toutes les instructions dans le corps de la boucle et utilise les préconditions linéaires associées pour remplacer les variables d'induction.

Cette transformation est un défi dans le cadre du source-à-source et du langage C. La Figure 4.9a donne un exemple de code C avec des effets de bord dans les références, par exemple `A[k++] = ...`. La solution que j'ai conçue et implémentée traite ces références selon le standard C. Par exemple `A[k++] = ...` est remplacé par `A[k = i+j+1, k-1] = ...` comme montré sur la Figure 4.9b. Le code source transformé conserve autant que possible la structure du code initial.

4.6 Fusion de boucles

La fusion de boucles est une transformation qui consiste à fusionner deux boucles ou plus en une seule. Cette transformation a été très étudiée pendant des décennies [Allen & Cocke 1972, Burstall & Darlington 1977, Kuck *et al.* 1981, Allen 1983, Goldberg & Paige 1984, Wolfe 1990, Bondhugula *et al.* 2010]. Trouver une solution optimale à la problématique globale de la fusion est tout sauf trivial [Darte 2000] et il y a plusieurs manières de traiter ce problème, tout comme il y a différentes définitions du problème lui-même.

Cette transformation peut réduire la surcharge induite par le branchement et l'incrémentation en éliminant des en-têtes de boucles et en augmentant la taille du corps de boucle. Cependant la pression sur le cache d'instruction et sur l'utilisation des registres dans le corps de la boucle est plus forte.

4.7 Légalité

La fusion de boucles n'est pas toujours valide et peut modifier la sémantique du programme. Une fusion invalide peut aboutir à l'ordre inverse des calculs dépendants. L'analyse de dépendance de données est utilisée pour déterminer quand la fusion est valide ou pas.

La validité de la fusion de boucles a été largement étudiée [Allen & Cocke 1972, Warren 1984, Aho *et al.* 1986], mais peut être exprimée de différentes manières. Kennedy [Kennedy & McKinley 1994] la définit comme si aucun arc de dépendance du corps de la première boucle vers le corps de la seconde est inversé après fusion.

Irigoin et al. proposent une autre solution [Irigoin *et al.* 2011 (perso)] basée sur les régions de tableaux [Creusillet & Irigoin 1996b]. Cette proposition, qui permet d'identifier ces dépendances sans aucun graphe de dépendances, est présentée à la Section 4.6.5.

La fusion de boucles parallèles peut aboutir à une boucle séquentielle. Cette situation,

illustrée par la Figure 4.11, peut être considérée comme bloquante pour la fusion dans un objectif de maximisation du parallélisme.

La Figure 4.10 illustre une simple fusion de boucles.

Cette section présente les différents objectifs de la fusion de boucles dans les travaux précédents, de la réduction de consommation d'énergie à une meilleure utilisation de registres vectoriels, puis présente l'intérêt de cette transformation dans le contexte de la pratique du GPGPU. C'est à dire la réduction du nombre de noyaux exécutés et l'augmentation de leur taille, la réutilisation de données, et la contraction des tableaux. L'algorithme proposé et implémenté dans PIPS est ensuite détaillé, puis une alternative basée sur les régions de tableaux est introduite. Enfin des considérations particulières liées à la génération de code pour accélérateurs sont abordées.

4.8 Remplacement des tableaux par des scalaires

Le remplacement de tableaux par des scalaires remplace les références constantes à des tableaux à l'aide de scalaires. Cette transformation peut être rapprochée de qui est réalisé dans la phase d'allocation des registres du compilateur chargé de générer le binaire final, quand il s'agit de conserver dans un registre une valeur mémoire le plus longtemps possible. Intuitivement, cela signifie qu'effectuer cette transformation au niveau du code source peut augmenter la pression sur les registres et aboutir à la sauvegarde de registres en mémoire.

Cette transformation peut aussi éliminer les tableaux temporaires, la plupart du temps après la fusion de boucles et particulièrement dans le contexte de codes générés automatiquement à partir de représentations de plus haut-niveaux. La Figure 4.21 indique comment trois lignes d'un script Scilab aboutissent à un trois tableaux temporaires qui peuvent être totalement remplacés par des scalaires après fusion de boucles.

Dans le contexte des accélérateurs, cette transformation est encore plus critique que sur un système à mémoires partagées. En effet les noyaux générés seront plus rapides en effectuant moins d'accès mémoires, mais la source principale de gain sera probablement l'élimination de transferts sur le bus PCIe.

Le remplacement de tableaux par des scalaires a été largement étudié [Gao *et al.* 1993, Sarkar & Gao 1991, Darte & Huard 2002, Carribault & Cohen 2004]. Cette section explore les différentes méthodes pour appliquer cette transformation dans le contexte du déport de noyaux vers un GPU. L'impact sur la performance est évalué pour différentes architectures

de GPU.

4.8.1 A l'intérieur du noyau

Une simple multiplication de matrices naïvement exécutée sur GPU est montrée à la Figure 4.22. Ce noyau inclut une boucle séquentiel avec une référence constante à un tableau. Cette référence peut être conservée dans une variable scalaire pendant l'exécution complète de la boucle. Cette transformation peut être réalisée par le compilateur qui génère le binaire. Les mesures présentées à la Figure 7.11, page 192, indiquent que l'effectuer au niveau du source est intéressant sur toutes les architectures testées, avec une accélération jusqu'à 2,39.

4.8.2 Après la fusion de boucles

La fusion de boucles génère un code dans lequel les instructions qui produisent et utilisent un tableau temporaire sont autant que possible dans le même corps de boucles. Certain de ces tableaux peuvent alors être complètement supprimés, économisant la bande passante et l'empreinte mémoire. Dans le contexte de code automatiquement généré depuis des représentations de haut-niveau, cette situation est courante. La Figure 4.21 montre un exemple d'un code généré depuis un script de trois lignes de Scilab. Après la fusion de boucles, le code C généré contient trois tableaux temporaires qui peuvent être remplacés par des scalaire comme illustré sur la Figure 4.21b.

Pour éliminer un tableau temporaire, ses éléments ne doivent pas être utilisés plus tard dans l'exécution du programme. PIPS fournit l'information de régions vivantes ($\mathcal{OUT}$, voir Section 3.2, page 64).

La Section 7.5.2, page 191, montre comment ce simple exemple aboutit a une accélération de 1,96 à 5,75 selon l'architecture. Contrairement à l'exemple précédent, dans ce cas le compilateur qui génère le binaire du noyau est démuni et il est donc critique d'effectuer cette transformation au niveau source.

4.8.3 Perfect Nesting of Loops

Dans certains cas, cette transformation peut casser l'imbrication parfaite des boucles. La Figure 4.23 illustre une telle situation. Les références constantes dans la boucle la plus interne sont affectées à des scalaires entre les deux boucles qui ne sont donc plus parfaitement imbriquées. Puisque seules les boucles parfaitement imbriquées sont exécutées en parallèle, ici seule l'une des deux boucles pourra être exécutée en parallèle après transformation. Le parallélisme est réduit : peu de fils d'exécution peuvent s'exécuter sur le GPU.

Le noyau généré sans remplacement est montré sur la Figure 4.23c, le noyau généré pour la boucle externe est montré sur la Figure 4.23e, la boucle interne est alors exécutée séquentiellement dans le noyau. Finalement la Figure 4.23d illustre le choix de paralléliser la boucle interne et de conserver la boucle séquentielle sur l'hôte. Aucune opération de transfert mémoire n'est requise pour cette version pour `u1` and `u2`. L'inconvénient est que alors que les deux autres versions n'ont qu'un unique appel de noyau, celle-ci requiert autant de lancements que d'itérations de la boucle externe.

Évaluer laquelle de ces trois versions est la plus performante est très dépendant de la taille des données. Les deux boucles ont un nombre différent d'itérations `N` and `M`. D'abord, le temps de transfert des tableaux `u1` et `u2` pour les versions des Figures 4.23e et 4.23c augmente avec `N`. Le nombre de fils d'exécution lancés sur le GPU augmente avec `N` et `M` pour la version de la Figure 4.23c, avec `N` pour la version de la Figure 4.23e, et avec `M` pour la version 4.23d. En augmentant le nombre de fils d'exécution, plus de calculs peuvent être réalisés sur l'accélérateur, mais aussi plus de parallélisme est exprimé ce qui augmente les probabilités de garder un fort taux d'utilisation de l'accélérateur. Le temps de calcul d'une exécution du noyau pour un fil d'exécution est constant pour les versions des Figures 4.23c et 4.23d mais augmente avec `M` dans la version de la Figure 4.23e. Finalement, la version de la Figure 4.23d peut souffrir d'un grand nombre de lancement de noyaux quand `N` croît. Ce problème est multi-dimensionel.

A moins que `N` soit large et `M` petit, la version 4.23e souffre d'un manque de parallélisme exprimé par rapport à la version 4.23c. Quand on compare cette dernière avec la version 4.23d, le cas de `N` petit et `M` grand aboutit à un petit avantage à la version 4.23d parcequ'il n'y pas de transferts de données. Bien que la même quantitée de données soit transférées au bout du compte, ce sera en utilisant les arguments de l'appel du noyau au lieu d'appel à des transferts DMA. À la place, les données seront transférées par les arguments de l'appel au noyau, fournissant une sorte de recouvrement de calcul et communication. Toutefois, la surcharge de `N` lancement de noyaux peut être plus importante qu'un unique transfert mémoire selon les architectures.

Cette analyse est confirmée par les expériences de la Section 7.5.3, page 191, et illustrée sur la Figure 7.13.

4.8.4 Conclusion

Cette section passe en revue une transformation connue, et montre comment l'implémentation dans PIPS, bénéficiant des analyses de régions de tableaux, peut aider à réduire l'empreinte mémoire et améliorer les performances globales quand on cible un GPU.

La pratique du GPGPU pose des contraintes inhabituelles pour cette transformation : conserver l'imbrication parfaite des boucles. J'ai modifié l'implémentation existante pour vérifier cette propriété. Des expériences détaillées sont présentées dans la Section 7.5, page 181, et une accélération de 1,12 à 2,39 est obtenue.

4.9 Déroulage de boucles

Le déroulage de boucles est une transformation éprouvée [Aho & Ullman 1977], qui vise à améliorer la performance de l'exécution d'une boucle. Le principe de base est de répliquer le corps de la boucle plusieurs fois pour effectuer plusieurs itérations de la boucle d'origine. Le nombre d'itérations de la nouvelle boucle est alors réduit. La Figure 4.24 montre un exemple de cette transformation. La boucle d'origine ne contient que peu de calcul pour chaque itération, donc le coût de l'entête de la boucle ainsi que le coût du branchement peuvent être significatifs. De plus, le parallélisme d'instruction (ILP) disponible pour l'ordonnanceur matériel est faible. La boucle déroulée répond à ces problèmes, au prix d'une pression sur les registres accrue [Bachir *et al.* 2008] et un code plus large qui pourrait nuire au cache d'instruction. Ces inconvénients peuvent annihiler tout gain apporter par ailleurs par le déroulage.

Dans le contexte de la programmation de GPU, cette transformation est intéressante pour deux raisons. La première concerne les boucles séquentielles rencontrées à l'intérieur des noyaux, alors que la seconde consiste à dérouler des boucles parallèles avant d'associer les itérations à des fils d'exécution. Dans ce dernier cas c'est un compromis entre la quantité de fils d'exécution exposée (TLP) et du parallélisme d'instruction (ILP) potentiel.

La section 7.6 présente les gains en performance que j'ai obtenu en déroulant la boucle interne du noyau de multiplication de matrices de la Figure 4.22. Une accélération jusqu'à 1,4 peut être observée selon l'architecture. L'impact sur la pression des registres est également étudié, et il est montré que le déroulage de boucles impacte fortement la consommation de registres par les noyaux.

4.10 Linéarisation des accès aux tableaux

Les programmes Fortran et C font usage de tableaux multi-dimensionnels. Toutefois, le standard OpenCL n'accepte pas l'utilisation de tels tableaux, ils doivent être convertis en pointeurs ou tableaux à une dimension et les accès doivent être linéarisés. Cette transformation est également obligatoire quand on utilise des tableaux de taille variable (C99 VLA) avec CUDA.

Le résultat de cette transformation est illustré par la Figure 4.25.

L'impact sur la performance pour du code CUDA est évalué à la Section 7.7. Cette transformation peut aboutir à un ralentissement jusqu'à vingt pour cent mais également, dans une seule configuration, aboutir à une petite accélération de deux pour cent.

Il n'y a pas de raison selon mon opinion qu'un standard tel que OpenCL interdise l'utilisation de tableaux multi-dimensionnels dans les paramètres formels des noyaux, et, considérant l'impact sur les performances, nous espérons qu'une version future supprimera cette contrainte.

4.11 Vers une chaîne de compilation

Ce chapitre présente plusieurs transformations individuelles qui sont applicables à différent moment du processus complet. L'enchaînement de toutes ces transformations peut être délicat et différent choix dans le processus aboutiront à différents résultats en terme de performance.

J'ai proposé et implémenté une méthode de mise en correspondance simple mais flexible des nids de boucles sur le GPU. Cette méthode permet de conserver une représentation interne simple et ignore la hiérarchie complète induite par l'utilisation de *NDRange* OpenCL. J'ai conçu et implémenté une phase de substitution de variable d'induction basée sur une méthode originale utilisant des préconditions linéaires. J'ai amélioré la détection du parallélisme dans PIPS, particulièrement le couplage avec la détection des réductions, et j'ai étudié l'impact de la combinaison de deux algorithmes différents sur la génération du code et la performance obtenue lors de l'exécution sur GPU. J'ai conçu et implémenté une phase de parallélisation dédiée aux réductions en utilisant les opérations disponibles sur GPU. J'ai conçu et implémenté une phase de fusion de boucles dans PIPS, incluant plusieurs heuristiques qui favorisent la mise en œuvre sur GPU. J'ai amélioré la phase de contraction de tableaux existante dans PIPS pour garder l'imbrication parfaite de boucles. J'ai identifié trois motifs différents et analysé l'impact de la scalarisation sur ces motifs. J'ai conduit des expériences pour valider l'impact individuel de chaque transformation présentée dans ce chapitre. Alors que la plupart des concepts sont bien connus, il est montré que pour plusieurs transformations, la méthode de l'état de l'art doit être adaptée pour répondre parfaitement aux contraintes des circuits massivement parallèles des GPUs.

PIPS offre un cadre de compilation flexible, mais le flot de compilation doit être piloté, comme indiqué sur la Figure 2.27 de la page 58. Le prochain chapitre motive et introduit les concepts d'un gestionnaire de passes programmable, implémenté dans PIPS, et sur lequel

Par4All construit le processus qui automatise les étapes de transformation.

5 Conception de compilateurs hétérogènes et automatisation

Alors que les chapitres précédents sont focalisés sur des transformations individuelles implémentées principalement dans PIPS, ce chapitre traite le processus de compilation complet, du code source original jusqu'au binaire final. A cette fin, nous introduisons Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] à la Section 5.1. Par4All est une initiative Open Source visant à fédérer les efforts réalisés sur les compilateurs pour permettre la parallélisation automatique des applications pour architectures hybrides.

Alors que les plate-formes matérielles grandissent en complexité, les infrastructures de compilation ont besoin de plus de flexibilité : à cause de l'hétérogénéité des plate-formes, les phases de compilation doivent être combinées de manière inhabituelle et dynamique, et plusieurs outils doivent être combinés pour gérer efficacement des parties spécifiques du processus de compilation. Le besoin de flexibilité apparaît également lors de la mise en œuvre de processus itératif de compilation, quand différentes combinaisons de phases de transformation sont explorées.

Dans ce contexte, il est nécessaire d'assembler des composants logiciels telles que les phases de compilation, sans avoir à plonger dans les arcanes des différents outils. L'entité en charge de la combinaison des phases de compilation est le gestionnaire de phases dans un compilateur monolithique classique. Alors que les gestionnaires de phases se basent habituellement sur un ordonnancement statique, l'introduction de *plug-ins* dans GCC et les tendances actuelles dans la conception des compilateurs ouvrent la voie des gestionnaires de phases dynamiques. De plus, la combinaison de différents outils dans la chaîne de compilation est rendue obligatoire par l'hétérogénéité des cibles. Dans ce contexte, l'automatisation de la collaboration d'outils si différents implique de définir un méta-gestionnaire de phases de haut-niveau.

L'aspect source-à-source est clé dans ce processus, comme expliqué dans la Section 5.2. Un gestionnaire de phases programmable est alors introduit à la Section 5.3.

Les simulations numériques font souvent appel à des bibliothèques de fonctions externes telles que Basic Linear Algebra Subprograms (BLAS) ou Fast Fourier transform (FFT) par exemple. La Section 5.4 présente comment l'utilisation de telles bibliothèques est exploitée pour déporter les calculs sur un GPU.

Finalement, la Section 5.5 propose un aperçu sur la manière dont différents outils

peuvent collaborer.

5.1 Projet Par4All

Les compilateurs récents proposent un moyen incrémental pour convertir des programmes vers des accélérateurs. Par exemple PGI Accelerator [Wolfe 2010] ou HMPP [Bodin & Bihan 2009] requiert l'insertion de directives. Le développeur doit sélectionner les parties du code qui doivent être déportées sur l'accélérateur. Il ajoute également des directives optionnelles qui pilote la gestion de l'allocation des données et des transferts. Les programmes sont plus faciles à écrire, mais le développeur doit toujours spécialiser son code source pour une architecture ou un modèle d'exécution. Ces travaux liés sont présentés en détails à la Section 2.2.

Contrairement à ces approches, Par4All [SILKAN 2010 (perso), Amini *et al.* 2012b (perso)] est un compilateur qui extrait le parallélisme automatiquement de codes C et Fortran. Le but de ce compilateur source-à-source est d'intégrer plusieurs outils de compilation dans un compilateur facile d'utilisation mais performant qui permet de cibler de multiple plate-formes matérielles. L'hétérogénéité est partout de nos jours, des super-ordinateurs jusqu'au monde de l'embarqué et du mobile. Adapté automatiquement les programmes aux cibles est donc un défi critique.

Par4All est actuellement basé sur l'infrastructure de compilation source-à-source PIPS [Irigoin *et al.* 1991, Amini *et al.* 2011a (perso)] et bénéficie de ses capacités interprocédurales, tels que les effets mémoires, la détection des réductions, la détection du parallélisme, mais aussi les analyses polyédriques telles que les régions de tableaux [Creusillet & Irigoin 1996b] et les préconditions linéaires.

La nature source-à-source de Par4All rend facile l'intégration de différents outils externes dans le flot de compilation. Par exemple, PIPS peut être utilisé pour identifier les parties qui sont intéressantes dans un programme et ensuite Par4All peut déléguer le traitement de ces morceaux à un optimiseur polyédrique comme PoCC [Pouchet *et al.* 2010a] ou PPCG [Verdoolaege *et al.* 2013], comme montré à la Section 5.5.

La combinaison des analyses de PIPS et l'insertion d'autres optimiseurs dans le flot de compilation est automatisé par Par4All en utilisant un gestionnaire de phase programmable (voir Section 5.3) pour effectuer des analyses à l'échelle du programme, trouver les boucles parallèles, et générer principalement de l'OpenMP, du code CUDA, ou du code OpenCL.

A cette fin, nous faisons principalement face à deux défis : la détection du parallélisme et la génération des transferts. La génération de directives OpenMP repose sur la paral-

lélisation à gros grain et la détection de réductions, comme présenté à la Section 4.4. Les cibles CUDA et OpenCL ajoutent la difficulté de la gestion des transferts de données. PIPS aide à répondre à ce problème en utilisant entre autre les régions de tableaux, pour générer des transferts entre l'hôte et l'accélérateur, comme introduit dans le Chapitre 3.

5.2 Système de transformation source-à-Source

Plusieurs des compilateurs de recherche passés sont source-à-source [Bozkus *et al.* 1994, Frigo *et al.* 1998, Ayguadé *et al.* 1999, Munk *et al.* 2010] ou basés sur une infrastructure de compilation source-à-source [Irigoin *et al.* 1991, Wilson *et al.* 1994, Quinlan 2000, ik Lee *et al.* 2003, Derrien *et al.* 2012]. Ils fournissent des transformations intéressantes dans le contexte du calcul hétérogène, tels que les algorithmes de détection du parallélisme (voir Section 4.3), la privatisation des variables, et plusieurs autres incluant celles présentées au Chapitre 4.

Dans le monde hétérogène, il est courant de se baser sur des compilateurs spécifiques à la cible pour générer du code binaire. De tels compilateurs prennent en général en entrée un dialecte de langage C pour générer du code assembleur. Il est donc critique de pouvoir générer du code source C pour les outils comme résultat du traitement.

En addition à la collaboration intuitive avec les compilateurs matériels, les compilateurs source-à-source peuvent aussi collaborer entre-eux pour accomplir leurs objectifs, utilisant alors le code source comme un médium commun, au prix de conversions entre la représentation textuelle (Textual Representation (TR)) et leur propre représentation interne (Internal Representation (IR)). La Figure 5.1 illustre ce comportement générique et, à la Section 5.5, illustre l'utilisation d'outil polyédrique externe pour des optimisations de nid de boucles. De plus, deux compilateurs source-à-source écrits dans la même infrastructure peuvent être combinés de cette manière. Par exemple, un générateur d'instruction SIMD a été utilisé pour la génération d'instructions SSE sur processeurs Intel et pour améliorer le code généré par le generateur CUDA/OpenCL présenté au Chapitre 4.

Un autre avantage traditionnel des compilateurs source-à-source inclut la facilité de débogue : l'IR peut être générée sous forme textuelle à tout moment et être exécutée. Pour la même raison ce sont des outils très pédagogiques qui illustrent le comportement d'une transformation.

5.3 Gestionnaire de passes programmable

Un problème récurrent dans la compilation pour plate-formes hétérogènes est le besoin de créer dynamiquement de nouvelles fonctions qui seront exécutées par différents

circuits matériels, en utilisant une transformation nommée détourage (*outlining*). Cette transformation introduisant de nouvelles fonctions voir de nouvelles unités de compilation ne s'insère pas aisément dans les gestionnaires de phases statiques. De plus, la compilation itérative [Goldberg 1989, Kulkarni *et al.* 2003] est de plus en plus considérée comme une alternative aux optimisations standards mais requiert la reconfiguration dynamique du processus de compilation.

Cette section est organisées comme suit : la Section 5.3.1 propose une vue d'ensemble de Pythonic PIPS (PyPS), le gestionnaire de phases implémenté dans PIPS. Il met en œuvre une interface de programmation avec une abstraction de haut-niveau des entités de compilation telles que les analyses, les phases, les fonctions, les boucles, etc. la Section 5.3.1.5 introduit des cas d'utilisation. Et enfin les travaux liés sont présentés à la Section 5.3.2.

5.3.1 PyPS

Une description formelle est disponible dans [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso), Guelton 2011a]. Des opérateurs multiples sont proposés pour décrire des transformations, gérer les erreurs, et différentes combinaisons de phases. Guelton améliore cette description et fournit une version significativement étendue dans sa thèse [Guelton 2011a].

Certaines approches introduisent un langage dédié [Yi 2011] pour le gestionnaire de phases, mais plutôt que d'introduire encore un nouveau langage spécifique (DSL), et suivant la fameuse phrase de Bernard de Chartres : "comme des nains juchés sur des épaules de géants" le langage Python est utilisé comme langage de base. Ce choix se révèle encore plus intéressant qu'attendu en fournissant non seulement des constructions de haut-niveau dans le langage mais aussi en ouvrant l'accès à un écosystème riche qui élargi l'ensemble des possibilités, au prix d'une dépendance sur l'interpréteur Python.

Dans le modèle présenté dans [Guelton *et al.* 2011b (perso), Guelton *et al.* 2011a (perso)], les transformations traitent le programme dans son ensemble. Toutefois, elle peuvent s'appliquer avec une granularité plus fine : au niveau de l'unité de compilation, d'une fonction, ou d'une boucle par exemple.

Le diagramme de la Figure 5.2 indique les relations entre toutes ces abstractions. Celles-ci uniquement sont exploitées par le gestionnaires de phases.

Les principales structures de contrôle impliquées sont introduites ici. Il s'agit des conditions et branchements, des boucles *pour*, des boucles *tant que*, et des exceptions. La description formelle complète des opérateurs peut être trouvée dans [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso), Guelton 2011a].

Alors que le code qui exploite des cibles matérielles hétérogènes est généré depuis plusieurs sources, une étape finale de compilation est requise pour obtenir le ou les binaires finaux. Ce n'est pas à proprement parler le rôle du gestionnaire de phases, toutefois comme pièce centrale il possède l'information précise requise pour piloter ce processus et les différents outils de construction du binaire à mettre en œuvre.

De plus, des transformations complexes pour architectures hybrides ou distribuées requièrent l'ajout d'appel à un support exécutif, souvent livré sous forme de bibliothèque partagée, et donc l'étape d'édition des liens est dépendantes des transformations effectuées. Par exemple si dans le flot de compilation, une opération de FFT avec la bibliothèque *FFTW3* est transformée dans son équivalent GPU avec *CuFFT*, le gestionnaire de passes peut propager cette information pour l'étape d'édition des liens.

Plusieurs architectures existantes et passées ont bénéficié de la flexibilité du gestionnaire de phases programmable implémenté dans PIPS :

- Terapix est un accélérateur à base de FPGA pour le traitement d'image proposé par Thales qui met en œuvre un jeu d'instruction VLIW, pour lequel le flot de compilation est résumé à la Figure 5.7. Ce flot de compilation a été implémenté par Guelton et décrit en détail dans sa thèse [Guelton 2011a].
- SAC est un générateur d'instructions vectorielles introduit dans [Guelton 2011a]. La Figure 5.8 montre un extrait de son processus de compilation en charge du tuilage.
- Un compilateur itératif qui requiert le clonage de l'état interne du compilateur et itère sur un domaine paramétrique de transformations en utilisant un algorithme génétique a été implémenté aisément à l'aide des abstractions fournies par PyPS.
- Par4All, présenté à la Section 5.1, est un compilateur multi-cibles (OpenMP,CUDA, OpenCL, SCMP, etc.) qui traite du Fortran et du C. Il repose fortement sur les capacités de PyPS et bénéficie de l'environnement Python.

5.3.2 Travaux liés

Dans les compilateurs traditionnels tel que GNU C Compiler (GCC), un processus de compilation est simplement une séquence de transformations chaînées les unes après les autres et appliquées de manière itérative sur chaque fonction du code source d'origine [Wikibooks 2009]. Ce comportement rigide a mené au développement d'un mécanisme de motivé par la difficulté à fournir des fonctionnalités additionnelles à l'infrastructure existante. Des exemples de *plug-ins* à succès incluent "dragonegg" et "graphite." Certaines limitations de GCC ont conduit au développement de Low Level Virtual Machine (LLVM) [Lattner

& Adve 2004], qui traite le manque de flexibilité en fournissant un gestionnaire de phases plus élaboré qui peut modifier le chaînage des phases à l'exécution.

Le compilateur source-à-source ROSE [Quinlan 2000, Schordan & Quinlan 2003] n'inclut pas de gestionnaire de phases particulier, mais fournit une interface C++ et Haskell. Le langage Poet [Yi 2011] est conçu pour construire des compilateurs, mais souffre de certaines limitations, comme il n'y a pas de séparation claire des principes entre les interfaces internes du compilateur le gestionnaire de phases.

Les directives de compilation offrent un moyen non-intrusif d'intervenir sur le processus de compilation. Elles sont intensivement utilisées dans plusieurs projets [Kusano & Sato 1999, Donadio *et al.* 2006, NVIDIA, Cray, PGI, CAPS 2011] pour générer du code efficace qui cible des multicœurs, des GPUs, ou des jeux d'instruction vectoriels Ils peuvent spécifier un ordre séquentiel des phases, mais ils ne fournissent pas de contrôle supplémentaire, et peuvent avoir une sémantique de composition complexe.

5.3.3 Conclusion

Cette section introduit PyPS, une interface de programmation de gestionnaire de phases pour l'infrastructure de compilation source-à-source PIPS. Cette infrastructure de compilation est réutilisable, flexible, et maintenable. Ce sont les trois propriétés requises dans le développement de nouveaux compilateurs à bas coût tout en respectant des contraintes de délai de mise sur le marché. Ceci est possible grâce à une séparation claire des concepts entre les interfaces internes, les phases de compilation, le gestionnaire de cohérence, et le gestionnaire de phases. Cette séparation claire n'est pas présente dans GCC et n'est pas complètement mis en œuvre non plus dans les compilateurs de recherche, bien que la problématique de la gestion des phases devient un sujet de recherche actif. Cinq compilateurs spécifiques et d'autres outils annexes sont implémentés avec cette interface de programmation : de générateurs OpenMP, SSE, SCMP, et CUDA/OpenCL, ainsi qu'un compilateur itératif optimisant.

Les travaux présentés dans cette section ont été publiés précédemment dans [Guelton *et al.* 2011a (perso), Guelton *et al.* 2011b (perso)].

5.4 Traitement des bibliothèques externes

Les développeurs font souvent usage de bibliothèques externes, hautement optimisées pour des traitements particuliers, comme de l'algèbre linéaire, du traitement d'image, du traitement du signal, etc. Le problème, dans le contexte de la parallélisation automatique de tels programmes, est qu'en général le code source de la bibliothèque n'est pas disponible soit

parce que le code est propriétaire, ou simplement car le processus de compilation normal ne requiert que le binaire installé sur la machine. Même si le code était disponible, ça pourrait représenter une énorme quantité de code à traiter. De plus le code serait probablement tellement optimisé pour une architecture de processeur généraliste qu'une transformation automatique pour GPU ou autre accélérateur semble utopique.

PIPS est un compilateur interprocédural, il analyse le graphe d'appel complet et propage les résultats des analyses aux sites d'appel. Une limitation actuelle de l'implémentation est que les fonctions de tout le graphe d'appel doivent être disponibles pour le traitement. La conséquence principal est qu'un code qui contient des appels à des bibliothèques ne peut être traité par PIPS.

D'un autre côté, plusieurs bibliothèques sont disponibles pour les GPUs et fournissent des fonctionnalités similaires, voir équivalentes. Cette section couvre la manière dont la gestion des bibliothèques est réalisée dans la chaîne de compilation. D'abord, une méthode automatique pour fournir des fonctions mimes (*stubs*) à la demande au compilateur est présentée à la Section 5.4.1. Ces différentes techniques sont utilisées pour remplacer les appels au code de bibliothèque CPU par l'équivalent GPU, comme introduit à la Section 5.4.2.

5.4.1 Fournisseur de fonctions mimes

En tant que compilateur purement interprocédural, PIPS arrête son traitement dès qu'il a besoin du résultat d'analyse d'une fonction pour laquelle le code source n'est pas disponible. Des fonctions mimes sont utilisées pour éviter le problème. Ce sont des fonctions qui possèdent la même signature et miment les effets de l'appel à la bibliothèque sur la mémoire, puisque PIPS utilise cette information, à fin de parallélisation par exemple.

Dans le contexte de l'outil Par4All, le but est de fournir une expérience décousue à l'utilisateur. Le développeur écrit son propre code et le fournit au compilateur, avec un paramètre qui indique la bibliothèque utilisée. Par4All inclut un mécanisme générique pour gérer les bibliothèques, le fournisseur de fonctions mimes, qui est invoqué par plusieurs étapes du processus de compilation.

Le flot qui met en œuvre le fournisseur de fonctions mimes est présenté sur la Figure 5.10. Ce processus a été validée avec l'implémentation d'un compilation Scilab-to-C (*Wild Cruncher* de SILKAN). Les fonctions mimes pour le support exécutif fourni par ce compilateur sont produites automatiquement à partir d'une description de plus haut niveau. Le code résultant fait appel à des bibliothèques internes de Scilab, par exemple

pour l'affichage, ou pour des calculs optimisés.

5.4.2 Gérer plusieurs implémentations d'une API : gestion des bibliothèques externes

Le fournisseur de fonctions mimes n'est en fait pas en charge d'une bibliothèque donnée mais d'une fonctionnalité donnée. Par exemple, un fournisseur peut être en charge de la FFT. Cette organisation est présentée par la Figure 5.10.

L'exemple Stars-PM [Aubert *et al.* 2009 (perso)] présenté en détail à la Section 3.1 utilise la bibliothèque FFTW [Frigo & Johnson 2005]. La FFT est effectué lors de la troisième étape montrée sur la Figure 3.3, page 65. Les appels à la bibliothèque FFTW correspondants sont montrés sur la Figure 5.11.

Le fournisseur de fonctions mimes dédié à la FFT fournit alors automatiquement à PIPS quand il se plaint des fonctions manquantes de la bibiolthèque FFT, comme expliqué dans la section précédente. A la fin du processus de compilation source-à-source, Par4All ajoute les sources et/ou les paramètres indiquées par le fournisseur à l'étape finale de génération du code. Par exemple un paramètre peut être ajouté à l'édition des liens, comme `-lcufft` pour le remplacement la bibliothèque FFTW sur GPU, et des sources peuvent être ajoutés pour faire l'interface.

5.5 Combinaisons d'outils

Avec l'hétérogénéité croissante, il est moins fréquent d'avoir une infrastructure de compilation qui répond à tous les besoins. Certaines infrastructures peuvent avoir une représentation interne qui intègre mieux certaines optimisations particulières, mais pas toutes. Par exemple, PIPS bénéficie d'analyses interprocédurales, mais n'a pas certaines capacités que d'autres outils tels que PoCC [Pouchet *et al.* 2010a], Pluto [Bondhugula *et al.* 2008b], ou PPCG [Verdoolaege *et al.* 2013] ont. Cet optimiseurs sont dédiés au modèle polyédrique, et donc restreints à une catégorie de programme en particulier.

Des expériences ont été menées dans Par4All pour exploiter PoCC [Pouchet *et al.* 2010a] à certain niveau du processus. PIPS détecte les parties à contrôle statique [Feautrier 1991], et les extrait pour être traitée par un outil polyédrique.

Pour effectuer de telles combinaisons d'outils, le source semble un niveau intéressant de représentation intermédiaire. Et le langage C, avec des directives, a fait ses preuves comme présenté à la Section 5.2, page 140.

5.6 Critères de profitabilité

Décider de l'intérêt de déporter un calcul sur le GPU n'est pas un problème simple.Deux

approches sont en concurrence, la première prend une décision basée sur une information complètement statique, alors que la seconde retarde la décision jusqu'au moment de l'exécution et profite d'informations plus précises, mais au prix d'un coût qui pénalise le temps d'exécution.

Une approche statique requiert d'estimer le temps d'exécution pour un noyau à la fois sur le GPU et le CPU. De plus le temps requis pour transférer les données doit également être estimé. Ce temps de transfert peut être pris en compte pour les deux placements. En effet si les données sont placées dans la mémoire du GPU, même si un calcul est un petit peu plus rapide sur CPU, le temps de transférer les données peut être en faveur d'effectuer le calcul sur le GPU. Cette situation est mentionnée à la Section 3.7, page 84.

Même si les bornes de boucles sont connues, il n'existe pas de formule universelle qui convienne à l'ensemble des architectures GPU. Leung et al. proposent une formule paramétrique [Leung 2008] pour une certaine catégorie de GPUs. Le modèle est simple et se base sur une suite de micro-jeux d'essai "représentatifs". Ils montrent que leur modèle est suffisamment précis pour une multiplication de matrices mais il semble peu probable que ce soit suffisant pour prédire une large gamme de noyaux. Des modèles plus précis ont été proposés [Kothapalli *et al.* 2009, Liu *et al.* 2007, Hong & Kim 2009, Baghsorkhi *et al.* 2010], mais ils sont limités dans le niveau de détail pris en compte, tel que le flot de contrôle divergeant, les conflits de bancs mémoire, et le délai du flux d'exécution SIMD. De plus, il sont rapidement dépassés par l'évolution des architectures, par exemple l'introduction de caches L1/L2 avec Fermi (voir Section 2.4.5).

Par4All n'inclut aucun modèle de performance spécifique pour les architectures GPU. Il peut bénéficier de l'analyse de complexité générique présent dans PIPS [Zhou 1992] pour éviter de paralléliser des boucles avec peu de calculs.

Les décisions prises à l'aide d'informations à l'exécution sont plus précises. Par exemple le support exécutif StarPU inclut plusieurs algorithmes d'ordonnancement, dont certains capables de garder trace des exécutions précédentes d'un noyau pour extrapoler le temps d'exécution en se basant sur un modèle de régression non linéaire $a \times size^b + c$. L'ordonnanceur peut prendre des décisions sur le placement d'un noyau en se basant sur un modèle de performance, l'emplacement actuel des données, ainsi que la disponibilité des ressources. La Section 6.2 donne plus d'information à ce sujet.

Le critère de profitabilité reste un problème non résolu, et plusieurs approches peuvent être mises en œuvre. La prochaine section présente une solution qui implique une sélection

à l'exécution en se basant sur des mesures réalisées au moment de la compilation.

5.7 Sélection de version à l'exécution

Une autre approche explorée par Dollinger and Loechner [Dollinger & Loechner 2011] consiste à exécuter plusieurs versions d'un noyau au moment de la compilation en utilisant un grand nombre d'espaces d'itération. Le temps d'exécution est à chaque fois enregistré, et les tables produites sont exploitées à l'exécution pour choisir la meilleure version en fonction des bornes de boucles. Ils montrent qu'une version de noyau peut être mieux adaptée pour une petite taille de jeu de données alors qu'une autre conviendra mieux pour une grande taille de jeu de données. Cette approche peut être rapprochée de ce qui est réalisé par la bibliothèque ATLAS [Whaley & Dongarra 1998] qui se spécialise pour une plate-forme automatiquement à la compilation en exécutant de multiples versions de chaque fonction.

Un inconvénient de l'approche est que l'accélérateur cible doit être disponible au moment de la compilation. Un autre moyen est de simuler l'exécution d'un noyau pour prédire la performance attendue sur plusieurs architectures. Récemment, plusieurs simulateurs pour Nvidia G80 [Bakhoda *et al.* 2009a, Collange *et al.* 2010], AMD Evergreen [Ubal *et al.* 2007], ou multiples architectures [Bakhoda *et al.* 2009b, Diamos *et al.* 2010] ont été proposés. Toutefois, ces simulateurs peuvent prendre beaucoup de temps.

La principal limitation de ces approches est peut-être que des kernels sont dépendants du contenu des données d'entrée, par exemple car certains branchements se basent sur ces données. Un ensemble de jeux de données "représentatif" doit être disponible au moment de la compilation, ce qui n'est pas toujours possible, ou pas forcément dans toutes les tailles. Cette méthode est complémentaire avec StarPU [Augonnet *et al.* 2010b]. Ce dernier accepte plusieurs versions d'un noyau, avec une fonction utilisateur appelée à l'exécution pour faire le choix selon les arguments et l'architecture.

5.8 Heuristique de configuration d'exécution

CUDA et OpenCL requiert l'expression d'un espace d'itération de la forme d'une grille tri-dimensionnelle avec deux niveaux de raffinement : la grille principale est elle-même découpée en blocs tri-dimensionnels. Cette abstraction est nommée *NDRange* dans la terminologie OpenCL introduite à la Section 2.3.2.2, page 32.

Cette grille doit être configurée manuellement par les développeurs. Pour obtenir la meilleure performance, ils font habituellement usage d'une approche d'essais-erreurs. De plus trouver la meilleure configuration est dépendant de l'architecture et de la taille des

données, et donc il n'y a pas de formule générique disponible.

5.8.1 Choisir la taille de groupe de travail

La taille de groupe de travail (*work-group* OpenCL) a un fort impact sur le temps d'exécution d'un noyau. Une pratique évidente est d'avoir une taille de groupe de travail multiple de la largeur SIMD ; par exemple toujours trente-deux sur les architectures Nvidia à ce jour.

Un point clé indiqué dans le guide de programmation CUDA est présenté comme maximisant le taux d'utilisation. Cette métrique a le mérite d'être facilement maximisée en ajustant la configuration de lancement pour un noyau et une architecture donnée. Puisque ces paramètres sont connus à l'exécution, la configuration de lancement peut être ajustée dynamiquement.

Le cadre exécutif de Par4All exploite la définition du taux d'utilisation pour construire une heuristique de choix de la configuration de lancement. Cette heuristique est présentée à la Section 5.8.1.

La Figure 7.2 montre sur une suite de jeux d'essai et plusieurs architectures la manière dont le choix de la taille du groupe de travail mène à une accélération jusqu'à 2,4.

5.8.2 Ajuster la dimensionnalité

Dans le contexte de la parallélisation automatique mis en œuvre ici, la répartition sur les dimensions d'un groupe de travail pour une taille donnée change les motifs d'accès à la mémoire et peut aboutir à un temps d'exécution très différent.

Les mesures obtenues pour une large gamme de configurations et différentes architectures sont présentées sur les Figures 7.3, 7.4, 7.5, et 7.6 page 183, pour le code illustré à la Figure 5.14, et sur les Figures 7.7, 7.8, 7.9, et 7.10, page 187, pour l'exemple de `matmul` présenté à la Figure 4.22. Pour une taille de groupe de travail, le choix d'une dimensionnalité rectangulaire mène à une accélération de deux à plus de quinze. Ces deux exemples mettent en œuvre un nid de boucles imbriquées avec des accès aux tableaux et une boucle séquentielle dans le noyau.

Les mesures mises en œuvre à la Section 7.4, page 180, mènent à une autre heuristique. Pour une taille de groupe de travail donnée, la forme retenue est autant que possible un carré, tout en conservant un nombre de fils d'exécution aussi proche que possible d'un multiple de trente-deux fils d'exécution sur la première dimension. Cette heuristique est

implémentée dans le support exécutif de Par4All.

5.9 Conclusion

Ce chapitre traite de la problématique d'automatiser le processus complet de compilation, du code d'origine jusqu'au binaire final, en passant par le support exécutif.

En raison de l'hétérogénéité des plate-formes actuelles, les phases de compilation doivent être combinées de manière inhabituelles et dynamiques. De plus, plusieurs outils peuvent être combinés pour gérer des parties spécifiques du processus de compilation.

Les analyses et transformations présentées au Chapitres 3 et 4 sont chaînées pour former un processus automatique cohérent. J'ai implémenté une solution entièrement automatique, dans le projet Par4All [Amini *et al.* 2012b (perso)], qui exploite ce processus et inclut un cadre d'exécution pour CUDA et OpenCL.

Ce chapitre introduit plusieurs concepts que j'ai testé et validé dans Par4All : l'avantage de l'aspect source-à-source, le role d'un gestionnaire de phases programmable, la gestion des bibliothèques, la collaboration entre différents outils, et le choix d'une configuration de lancement à l'exécution.

6 Gestion de multiples accélérateurs

L'essentiel du travail présenté dans cette thèse considère le cas d'un hôte et un accélérateur. Ce chapitre présente des expériences préliminaires menées avec plusieurs accélérateurs attachés à un hôte unique.

6.1 Introduction

Alors que cette étape est peu onéreuse d'un point de vue du matériel, ça représente un défi difficile pour les développeurs qui essayent encore de programmer un accélérateur unique.

C'est même encore plus difficile quand l'ambition porte sur la fourniture d'une solution automatique, comme dans cette thèse. Certains travaux liés sont décrits dans la Section 6.4.

Deux approches majeures sont étudiées dans ce chapitre. La première, présentée à la Section 6.2, se base sur du parallélisme de tâche, où chaque tâche est un noyau qui peut être exécuté sur un accélérateur, un CPU, ou les deux. La seconde approche présentée à la Section 6.3 consiste à séparer l'espace d'itération et les données sur plusieurs accélérateurs.

Certaines expériences sont présentées à la Section 7.10, bien que ce chapitre ne présente que des travaux préliminaires sur ce sujet, qui promet plus de développements dans la

prochaine décennie.

6.2 Parallélisme de tâches

Améliorer l'état de l'art sur l'extraction automatique de tâches est au delà du sujet de cette thèse. Supposant qu'une méthode satisfaisante existe dans le compilateur pour extraire le parallélisme de tâches ou que celui-ci soit exprimé par le développeur, comment peut-il être exploité pour cibler plusieurs accélérateurs ?

Cette section étudie la manière dont un système exécutif comme StarPU [Augonnet *et al.* 2011] peut être adapté comme cible de compilation pour exploiter plusieurs accélérateurs.

StarPU [Augonnet 2011, Augonnet *et al.* 2010b] est un système de support exécutif qui offre une abstraction de tâche au développeur. Les calculs sont séparés dans des fonctions, les *codelets*, avec des restrictions. Par exemple, aucune variables globales ou statiques ne peut être dans les tâches. Les tâches prennent comme paramètres des scalaires ou des tampons pré-enregistrés auprès de StarPU. Ces derniers une fois enregistrés ne peuvent être utilisés en dehors d'un *codelets*. Ce modèle est très similaire à la spécification OpenCL (voir Section 2.3, page 30).

StarPU prend en charge l'ordonnancement et l'exécution de ces *codelets* aussi efficacement que possible en exploitant l'ensemble des ressources disponibles à l'exécution. La cohérence du placement mémoire est assurée pour les tampons mémoires enregistrés. De plus, l'ordonnanceur peut décider où exécuter un traitement en se basant sur l'emplacement actuel des données d'entrée.

StarPU obtient des performances intéressantes en distribuant le travail parmi multiples GPUs et CPUs simultanément [Augonnet *et al.* 2010a].

Une méthode d'extraction de tâches a été ajoutée à PIPS pour permettre les expériences menées dans ce chapitre. Cette implémentation est naïve, bien loin des méthodes interprocédurales et hiérarchiques mentionnées à la Section 6.4. De plus, elle repose sur une simple heuristique basée sur la présence de boucles à la place d'une connexion avec l'analyse de complexité [Zhou 1992] fournie par PIPS pour les décisions de profitabilité. Toutefois dans le contexte de ce chapitre, c'est suffisant pour de simples jeux d'essai tels que les Polybenchs ou des scripts Scilab automatiquement convertis en C. La Figure 6.1 illustre l'extraction en tâche sur le simple exemple `3mm` de la suite Polybench.

Une fois les tâches extraites, le code doit être décoré à l'aide de directives qui vont déclencher les appels au support exécutif StarPU.

Les régions de tableaux sont utilisées pour générer des informations supplémentaires exploitées par StarPU pour l'optimisation des communications.

La Figure 6.2 illustre le résultat de cette transformation sur l'exemple 3mm. Les tâches ont été générées pour chaque multiplication de matrices, les pragmas ont été générés de telle sorte que StarPU puisse suivre l'utilisation des tampons, et les paramètres des tâches ont été déclarés avec le qualificatif `const` ou `__attribute__((output))`.

6.3 Parallélisme de données via tuilage de nid de boucles

La Section 3.9 page 90 explore la minimisation des communications dans le cadre de l'exécution d'un nid de boucle tuilé sur un accélérateur. Cette section introduit la méthode d'exécution d'un nid de boucles composé de tuiles parallèles sur plusieurs GPUs.

Guelton a introduit dans PIPS une méthode de tuilage symbolique pour ajuster l'empreinte mémoire à la capacité de l'accélérateur [Guelton 2011a]. J'ai dérivé cette méthode de manière un petit peu différente dans PIPS pour instancier à l'exécution le nombre de tuiles au lieu d'exprimer la taille de tuile. Pour gérer plusieurs GPUs, le nid de boucles est découpé en tuiles qui sont distribuées sur plusieurs accélérateurs.

La Figure 6.4b illustre cette transformation, et indique les régions de tableau calculées par PIPS sur cet exemple. Le code final avec les instructions de transferts est montré sur la Figure 6.4c. Les appels aux fonctions préfixées par `P4A_` sont des appels asynchrones.

A l'exécution, des files OpenCL sont crées et associées avec des accélérateurs et l'espace d'itération est dynamiquement découpé en autant de tuiles que de files, grâce à la méthode de tuilage dynamique. Un système de support exécutif a été développé pour répondre à ce mode de fonctionnement. Il enregistre implicitement les données à transférer pour chaque tuile, ordonnance de manière asynchrone les communications et les lancements de noyaux. Les dépendances sont exprimées par des événements Aopencl dans les différentes files.

Par effet de bord, et parce que le support exécutif basé sur OpenCL-based peut associer de multiples files de commande par accélérateur, et ainsi être utilisé pour recouvrir calcul et communication sur un accélérateur.

La Section 7.10, page 205, présente des résultats expérimentaux pour les deux méthodes présentées. La méthode de parallélisme de tâche se montre intéressante pour certains types d'application. Par exemple l'exemple 3mm de la suite Polybench montré sur la Figure 6.1a est accéléré de trente pour cent avec deux Nvidia C1060 au lieu d'un. Toutefois, cette approche est limitée par le nombre de tâches (ou noyaux) qui peuvent être exécutés en parallèle. L'exemple 3mm possède deux tâches qui peuvent être exécutées en parallèle, suivie d'un point de synchronisation avant la dernière tâche. Ajouter plus que deux GPUs est inutile

dans ce cas.

Une autre limitation est le coût d'ordonnancement de StarPU. L'ordonnanceur naif est peu coûteux, mais aboutit à trop de transferts inutiles puisqu'il ne prend pas en compte l'emplacement des données dans ses choix. D'autres ordonnanceurs fournis prennent en compte l'affinité entre tâches et obtiennent de meilleurs résultats. Les mesures de la Section 7.10 page 205 confirment que l'utilisation d'un ordonnanceur confirme cette analyse.

6.4 Conclusion

L'hétérogène n'est plus le futur, mais notre présent. Les compilateurs et les outils de développement sont en général limités à un GPU et un hôte, mais l'étape suivante est clairement en direction de plus d'hétérogénéité à l'aide de plus d'accélérateurs.

Ce chapitre présente deux manières différentes d'extraire le parallélisme dans le but d'exécuter des noyaux sur plusieurs accélérateurs. La première se repose sur le parallélisme de tâche dans lequel chaque tâche est un noyau qui peut s'exécuter sur un accélérateur et/ou sur un CPU. J'ai conçu et implémenté une simple extraction de tâches dans PIPS, même si l'approche d'exécution est indépendante de la méthode d'extraction. J'ai également implémenté une phase de génération de code pour piloter le système exécutif StarPU et validé la chaîne complète avec des expériences. La deuxième approche consiste à séparer l'espace d'itération et les données entre différents accélérateurs. J'ai modifié la méthode de tuilage symbolique existante dans PIPS. J'ai implémenté et validé à l'aide d'expériences un support exécutif dédié pour associer l'exécution des tuiles à plusieurs GPUs.

Les résultats expérimentaux obtenus sont encourageants, et des travaux approfondis sont attendus dans ce domaine plein de défis pour les prochaines décennies.

Le prochain chapitre présente les expériences mises en œuvre pour valider toutes les propositions développées dans les chapitres précédents.

7 Expériences

Ce chapitre présente les résultats expérimentaux qui valident les différentes transformations de programmes présentées dans les Chapters 3, 4, 5, et 6. Des jeux d'essai présentés à la Section 7.2 et plusieurs plate-formes matérielles présentées à la Section 7.1 ont été utilisés.

Les résultats montrent que l'approche est plutôt robuste. Les mesures sur un code de simulation numérique n-corps montrent une accélération de douze par rapport à une parallélisation naïve et huit par rapport à une version OpenMP automatiquement générée

sur deux processeurs à six cœurs.

Des travaux préliminaires au sujet d'une exécution distribuée sur plusieurs GPUs sont présentés, et les mesures montrent que des opportunités d'accélération existent déjà. Les travaux futurs dans ce domaine sont prometteurs.

8 Conclusion

> Bienvenue dans la jungle matérielle [13].

Dix ans après l'écriture de cette affirmation par Herb Sutter [Sutter 2011], elle touche plus que jamais une problématique brûlante. Des téléphones mobiles dans chaque poche jusqu'aux super-calculateurs, l'exploitation du matériel représente un défi croissant pour les développeurs.

Il n'y a pas de solution satisfaisante qui a été proposée pour les trois "P" propriétés : Performance, Portabilité, et Programmabilité.

La jungle actuelle dans l'écosystème des circuits matériels est maintenant reflétée dans le monde du logiciel, avec de plus en plus de modèles de programmation, de nouveaux langages, différentes interfaces de programmation, etc. Cependant aucune solution universelle n'a encore émergée. J'ai conçu et implémenté une solution automatique basée sur un compilateur pour répondre en partie au problème. La programmabilité et la portabilité sont assurées par définition, et si la performance n'est pas toujours au niveau de ce qu'obtiendrait un développeur expert, elle reste excellente sur une large gamme de noyaux et d'applications. J'ai obtenu une accélération de cinquante-deux pour une simulation numérique n-corps par rapport au code séquentiel compilé avec GCC.

Ce travail explore les problèmes associés avec une solution entièrement automatique, traitant la problématique des trois Ps en ne changeant ni le modèle de programmation, ni le langage, mais en implémentant des transformations avancées et un support exécutif. La portabilité et la programmabilité sont évidement atteintes : le développeur n'a pas besoin d'avoir une bonne connaissance de la plate-forme sous-jacente.

Le compromis sur les performances est limité, comme illustré par les expériences du Chapitre 7. Elles montrent également que la portabilité des performances peut être atteinte sur de multiples architectures.

13. Welcome to the hardware jungle

Contributions

J'ai conçu et implémenté plusieurs nouvelles transformations et analyses dans Par4All et dans l'infrastructure de compilation source-à-source PIPS. J'ai également modifié des transformations de l'état de l'art pour obtenir une transformation dérivée mais adaptée aux contraintes particulières des GPUs, comme illustré dans la Figure 8.1.

Je suis le processus chronologique de la figure plutôt que l'ordre des chapitres pour présenter mes contributions ci-après.

J'ai conçu et implémenté une nouvelle transformation de substitution de variables d'induction basée sur une analyse de préconditions linéaires (voir Section 4.5, page 111). Cette transformation permet la parallélisation de boucles en présence de variables d'induction.

J'ai étudié les combinaisons de deux algorithmes de parallélisation, en analysant l'impact sur la génération de code de chacun d'entre eux à la Section 4.3, page 101.

J'ai amélioré l'analyse existante de détection des réductions pour couvrir des spécificités du langage C, et exploité cette analyse pour permettre la parallélisation de boucles qui comporte des réductions en améliorant les algorithmes de parallélisation. J'ai implémenté une méthode de mise en correspondance pour certaines boucles avec réductions sur GPU, utilisant les opérations atomiques supportées par OpenCL et CUDA (voir Section 4.4, page 105).

J'ai implémenté deux versions d'une phase de fusion de boucles : une basée sur un graphe de dépendances et l'autre sur les régions de tableaux. J'ai mis au point des heuristiques pour piloter la fusion pour cibler les GPUs. C'est une transformation particulièrement critique quand le code à traiter a été généré depuis des représentations de plus haut niveau, tel que Scilab par exemple. Cette transformation permet alors à une phase ultérieure de remplacer des tableaux par des scalaires, jusqu'à supprimer complètement plusieurs tableaux temporaires générés par de tels outils.

J'ai étudié plusieurs méthodes de remplacement de tableaux par des scalaires dans le contexte de la pratique du GPGPU à la Section 4.7, page 127, et j'ai modifié l'implémentation de PIPS pour satisfaire aux contraintes particulières de la génération de code pour GPUs, particulièrement l'imbrication parfaite des nids de boucles.

Le déroulage de boucle a été présenté à la Section 4.8, page 132, et son impact a été analysé et validé à l'aide de plusieurs expériences.

Un support exécutif pour associer un espace d'itération à l'exécution d'un noyau sur un GPU de manière flexible, incluant des transformations tel que le tuilage par exemple, est proposé à la Section 4.2, page 98.

La problématique de la génération de code de communication entre l'hôte et l'accélérateur est étudiée au Chapitre 3. Les régions de tableaux convexes sont utilisées pour générer des communications efficaces. J'ai conçu et implémenté une analyse, une transformation, et un support exécutif pour optimiser de manière interprocédurale le placement des communications en préservant les données sur le GPU aussi longtemps que possible pour éviter les communications redondantes.

La génération de code pour multiples GPUs est traitée dans le Chapitre 6. Deux différentes sources de parallélisme sont présentées et comparées. J'ai implémenté une méthode simple d'extraction de tâches, ainsi qu'un générateur de code pour le support exécutif StarPU. J'ai également modifié la transformation existante de tuilage symbolique pour distribuer les nids de boucles sur plusieurs GPUs. J'ai implémenté le support d'exécutif qui réalise cette distribution à l'aide d'OpenCL.

Le processus complet est automatisé à l'aide d'un gestionnaire de phases programmable présenté dans le Chapitre 5. J'ai implémenté un fournisseur de fonctions mimes génériques en charge de la gestion des bibliothèques externes utilisées dans le code à traiter.

J'ai également implémenté un support exécutif pour gérer les configurations de lancement des noyaux. En particulier j'ai mis au point des heuristiques de choix de configuration de lancement à l'exécution basées sur la taille des données et l'architecture cible.

Finalement, j'ai mis en œuvre une vaste gamme d'expériences et d'analyses dans le Chapitre 7 pour valider toutes les transformations proposées, et les supports exécutifs implémentés. Vingt jeux d'essai de la suite Polybench, trois de Rodinia, et la simulation cosmologique n-corps Stars-PM ont été utilisés. J'ai obtenu une moyenne géométrique de quatorze pour l'accélération mesurée en comparaison avec le code séquentiel d'origine compilé avec GCC. La simulation numérique Stars-PM est accélérée d'un facteur cinquante-deux. Utiliser plusieurs GPUs permet d'obtenir une accélération de 1,63 par rapport à l'utilisation d'un GPU unique.

Dans la foulée de l'affirmation d'Arch Robison [Robison 2001]

> Les optimisations de programme à la compilation sont similaires à de la poésie :
> il y a plus d'écrit que réellement publié dans des compilateurs commerciaux.

La majeur contribution de ce travail, au delà des concepts, est incluse dans un prototype industriel livré sous licence open-source : le compilateur Par4All. Basé sur l'infrastructure PIPS, il fournit aux développeurs une solution automatique pour offrir du reciblage d'applications. L'interface est minimale et Par4All peut être invoqué aussi simplement que : `p4a --opencl my_code.c -o my_binary`.

De plus, la compagnie SILKAN a intégré les techniques présentées dans cette dissertation dans un environnement utilisateur adapté aux développeurs Scilab, offrant en un "click" accès à la puissance des accélérateurs matériels tels que les GPUs.

Travaux Futurs

Le Chapitre 6 contient des travaux préliminaires au sujet de l'utilisation de plusieurs GPUs, avec des résultats expérimentaux encourageants. Plus de travail doit être réalisé sur des optimisations de noyaux pour cibler de multiples GPUs. Par exemple les nids de boucles doivent être restructurés pour s'assurer d'une meilleure localité des données et optimiser les communications en distribuant les données sur de multiples GPUs.

En général, les compromis entre les décisions statiques prises par le compilateur et celles prises lors de l'exécution par un support exécutif doivent être explorés avec plus de détails. Le compilateur pourrait générer des informations supplémentaires destinées à être exploitées par le support exécutif, tels que l'estimation du temps d'exécution d'un noyau ou l'usage de tampon mémoire avec StarPU, comme introduit dans le Chapitre 6.

Les transformations de l'organisation mémoire des données ne sont pas étudiées dans cette thèse. Des outils polyédriques peuvent être adaptés à ce type de transformation. Le couplage de tels outils à l'intérieur de Par4All devrait être poussé un cran au delà.

L'utilisation des mémoires locales OpenCL n'est pas traitée. Il serait intéressant d'implémenter une méthode à base de régions de tableau et de comparer le résultat avec une solution dans le modèle polyédrique tel que proposé par PPCG.

Les trois *P*s, Performance, Portabilité, et Programmabilité, sont de nos jours complétés par la contrainte énergétique. Certaines bibliothèques ou certains langages spécifiques sont conçus pour prendre en compte une exécution qui s'adapte automatiquement à la consommation. Faire des compromis sur les trois *P*s dans un compilateur en prenant en compte la consommation énergétique offre des opportunités intéressantes de recherches futures.

D'autres paradigmes ont émergés, par exemple le langage de flot de données ΣC exploité par l'accélérateur Multi-Purpose Processor Array (MPPA), introduit à la Section 2.2.9. Alors que les auteurs de ΣC s'attendent à ce qu'il puisse être ciblé par des outils tels que Par4All [Goubier *et al.* 2011], plus de recherches sont nécessaires pour atteindre cet objectif.

Finalement, le matériel progresse d'année en année. Cette thèse a débuté avant que l'architecture *Fermi* ne soit disponible, et la génération suivante, *Kepler*, est maintenant

sur le point d'être livrée avec de nouvelles capacités. Le modèle de programmation des GPUs évolue. Le parallélisme dynamique introduit par Nvidia ouvre la voie vers de plus en plus de flexibilité offerte par le matériel au travers d'interfaces telle que CUDA. La mise en correspondance automatique de programmes avec ce modèle sans information supplémentaire devient difficile.

La collaboration entre générateurs de code depuis une abstraction de haut niveau et un compilateur tel que Par4All est une piste cruciale. Les informations sémantiques de haut-niveau sont connues par le générateur de code, et être capables de les propager dans le compilateur optimisant semble être la voie à suivre.

L'ère du calcul à l'aide de GPUs et plus généralement du calcul hétérogène n'en est qu'à ses débuts, et plusieurs défis intéressants seront rencontrés par les développeurs de compilateurs s'aventurant dans cette direction.

Transformations de programme automatiques et source-à-source pour accélérateurs matériels de type GPU

Résumé : Depuis le début des années 2000, la performance brute des cœurs des processeurs a cessé son augmentation exponentielle. Les circuits graphiques (GPUs) modernes ont été conçus comme des circuits composés d'une véritable grille de plusieurs centaines voir milliers d'unités de calcul. Leur capacité de calcul les a amenés à être rapidement détournés de leur fonction première d'affichage pour être exploités comme accélérateurs de calculs généralistes. Toutefois programmer un GPU efficacement en dehors du rendu de scènes 3D reste un défi.

La jungle qui règne dans l'écosystème du matériel se reflète dans le monde du logiciel, avec de plus en plus de modèles de programmation, langages, ou API, sans laisser émerger de solution universelle.

Cette thèse propose une solution de compilation pour répondre partiellement aux trois "*P*" propriétés : Performance, Portabilité, et Programmabilité. Le but est de transformer automatiquement un programme séquentiel en un programme équivalent accéléré à l'aide d'un GPU. Un prototype, Par4All, est implémenté et validé par de nombreuses expériences. La programmabilité et la portabilité sont assurées par définition, et si la performance n'est pas toujours au niveau de ce qu'obtiendrait un développeur expert, elle reste excellente sur une large gamme de noyaux et d'applications.

Une étude des architectures des GPUs et les tendances dans la conception des langages et cadres de programmation est présentée. Le placement des données entre l'hôte et l'accélérateur est réalisé sans impliquer le développeur. Un algorithme d'optimisation des communications est proposé pour envoyer les données sur le GPU dès que possible et les y conserver aussi longtemps qu'elle ne sont pas requises sur l'hôte. Des techniques de transformations de boucles pour la génération de code noyau sont utilisées, et même certaines connues et éprouvées doivent être adaptées aux contraintes posées par les GPUs. Elles sont assemblées de manière cohérente, et ordonnancées dans le flot d'un compilateur interprocédural. Des travaux préliminaires sont présentés au sujet de l'extension de l'approche pour cibler de multiples GPUs.

Mots clés : GPU, CUDA, OpenCL, Parallélisation automatisée, Compilation

Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators

Abstract: Since the beginning of the 2000s, the raw performance of processors stopped its exponential increase. The modern graphic processing units (GPUs) have been designed as array of hundreds or thousands of compute units. The GPUs' compute capacity quickly leads them to be diverted from their original target to be used as accelerators for general purpose computation. However programming a GPU efficiently to perform other computations than 3D rendering remains challenging.

The current jungle in the hardware ecosystem is mirrored by the software world, with more and more programming models, new languages, different APIs, etc. But no *one-fits-all* solution has emerged.

This thesis proposes a compiler-based solution to partially answer the three "*P*" properties: Performance, Portability, and Programmability. The goal is to transform automatically a sequential program into an equivalent program accelerated with a GPU. A prototype, Par4All, is implemented and validated with numerous experiences. The programmability and portability are enforced by definition, and the performance may not be as good as what can be obtained by an expert programmer, but still has been measured excellent for a wide range of kernels and applications.

A survey of the GPU architectures and the trends in the languages and framework design is presented. The data movement between the host and the accelerator is managed without involving the developer. An algorithm is proposed to optimize the communication by sending data to the GPU as early as possible and keeping them on the GPU as long as they are not required by the host. Loop transformations techniques for kernel code generation are involved, and even well-known ones have to be adapted to match specific GPU constraints. They are combined in a coherent and flexible way and dynamically scheduled within the compilation process of an interprocedural compiler. Some preliminary work is presented about the extension of the approach toward multiple GPUs.

Keywords: GPU, CUDA, OpenCL, Automatic Parallelization, Compilation

MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY