NNT : 2017SACLX090

THÈSE DE DOCTORAT

DE

L'UNIVERSITÉ PARIS-SACLAY

PRÉPARÉ À

L'ÉCOLE POLYTECHNIQUE

ÉCOLE DOCTORALE N° 573

INTERFACES : APPROCHES INTERDISCIPLINAIRES / FONDEMENTS, APPLICATIONS ET INNOVATION

Spécialité de doctorat: Informatique

par

M. Yunsong Wang

Optimization of Monte Carlo Neutron Transport Simulations by Using Emerging Architectures

Thèse présentée et soutenue à Gif-sur-Yvette, le 14 décembre 2017 :

Composition de jury :

M. Marc Verderi	Directeur de Recherche, CNRS/IN2P3/LLR	Président du jury
M. Andrew Siegel	Expert Senior, Argonne National Laboratory	Rapporteur
M. Raymond Namyst	Professeur, Université de Bordeaux/LABRI	Rapporteur
M. David Chamont	Chargé de Recherche, CNRS/IN2P3/LAL	Examinateur
M. David Riz	Ingénieur, CEA/DAM	Examinateur
M. Christophe Calvin	Directeur de Recherche, CEA/DRF	Directeur de thèse
M. Fausto Malvagi	Expert Senior, CEA/DEN	Encadrant
M. Emeric Brun	Ingénieur, CEA/DEN	Encadrant

Résumé

Les méthodes Monte Carlo (MC) sont largement utilisées dans le domaine du nucléaire pour résoudre les équations du transport neutronique. Ce type de méthode stochastique offre l'avantage de minimiser les approximations mais présente un taux de convergence lent régit par la loi des grands nombres, ce qui rend les simulations très coûteuses en termes de temps de calcul. Auparavant, l'évolution régulière de la puissance des processeurs permettait d'améliorer les performances du code sans avoir à reconcevoir les algorithmes. De nos jours, avec l'avènement des architectures many-cœurs (Intel MIC) et les accélérateurs matériels, les travaux d'optimisation sont indispensables et sont une préoccupation importante de la communauté MC.

Dans ce type de simulation, les calculs les plus coûteux concernent le calcul des sections efficaces, qui modélisent l'interaction probabiliste d'un neutron avec les nucléides qu'il rencontre sur sa trajectoire, au niveau microscopique. L'approche conventionnelle consiste à pré-calculer avant la simulation les sections efficaces de chaque type de nucléide, à chaque température intervenant dans le système à partir des informations des bibliothèques de données nucléaires, et rangé dans des tables indexées par l'énergie incidente. Ces données sont ensuite chargées en mémoire. Pendant le calcul, elles sont ensuite récupérées dans des tables et le caractère stochastique du transport MC induit des accès à la mémoire aléatoires pour un coût de calcul minime correspondant à une interpolation linéaire. Dans ce cas, l'essentiel du temps de calcul se concentre sur des recherches dans ces tables de probabilités, qui sont de tailles très variables en fonction de chaque nucléide.

Afin de minimiser les conflits d'accès mémoire, nous avons étudié et optimisé une vaste collection d'algorithmes de recherche afin d'accélérer ce processus de récupération de données. Dans un premier temps, nous avons étudié et proposé plusieurs alternatives à la recherche binaire conventionnelle, telles que la recherche N-aire (variante vectorisée de la recherche binaire) et la recherche linéaire vectorisée. Les évaluations montrent qu'une accélération significative peut être obtenue par rapport à la recherche binaire conventionnelle à la fois sur CPU classique et architecture many-cœurs. Cependant, la recherche N-aire n'est pas plus performante que la recherche binaire, ce qui indique que les efforts de vectorisation sont pénalisés par des accès dispersés et une mauvaise utilisation du cache. La vectorisation de certains des algorithmes s'est montrée efficace sur l'architecture MIC grâce à ses unités vectorielles de 512 bits. Cette amélioration est moins nette sur CPU, où les registres vectoriels étant deux fois plus petits. Dans la seconde partie, nous avons étudié des multiples structures de données possibles permettant d'optimiser l'accès aux tableaux des sections efficaces. Ainsi, la grille d'énergie unifiée est la plus performante dans tous les cas. Ce type de méthode synthètise tous les

tableaux en un seul, donc la recherche dans la table n'a besoin d'être effectuée qu'une seule fois pour chaque calcul de section efficace macroscopique. Cascade fractionnaire, un compromis similaire utilisant moins d'espace mémoire, est également implémentée et évaluée. Les méthodes d'accès par tables de hachage, avec des stratégies de hachage linéaires et logarithmiques, présente le meilleur compromis entre performance et empreinte mémoire. A partir de ces algorithmes existent, nous avons proposé le N-ary map en utilisant des tampons supplémentaires pour assurer des accès mémoire efficaces lors du chargement des bordures dans les registres vectoriels. Toutes ces variantes sont évaluées sur les systèmes de multi-cœurs et de many-cœurs. Cependant, ils sont redoutablement inefficaces du fait de la saturation de la mémoire et du manque de vectorisation de ces algorithmes. En d'autres termes, la puissance de calcul des architectures modernes est en grande partie gaspillée. Une optimisation supplémentaire comme la réduction de la mémoire s'avère très importante car elle améliore en grande partie les performances informatiques. Un autre problème majeur de ces méthodes est l'empreinte mémoire très importante qu'elles induisent quand un grand nombre de températures sont à considérer comme par exemple dans le cadre d'un couplage à la thermo-hydraulique. En effet, les sections efficaces à une température pour les quatre cent noyaux impliquées dans une simulation représentent près d'un gigaoctet d'espace mémoire, ce qui rend impossible la simulation à plusieurs centaines de températures.

Afin de résoudre ce problème, nous avons étudié une approche radicalement opposée : la reconstruction au vol des sections efficaces. L'idée est de réaliser le calcul des sections efficaces à partir des données élémentaires se trouvant dans les bibliothèques de données nucléaires (description des résonances) ainsi que le traitement en température (élargissement Doppler) pendant la simulation, à chaque fois qu'une section efficace est nécessaire. Cet algorithme, très calculatoire, repose sur une formulation similaire à celle utilisée dans le calcul des bibliothèques standards de neutronique. Cette approche permet de passer d'un problème de type "memory-bound" à un problème de type "compute-bound" : seules quelques variables pour chaque résonance sont nécessaires au lieu de la table conventionnelle. L'espace mémoire est ainsi largement réduit, ainsi que les conflits d'accès. Cette méthode est cependant extrêmement coûteuse en temps de calcul. Après une série d'optimisations, les résultats montrent que le noyau de reconstruction bénéficie de la vectorisation et peut atteindre 1,806 GFLOPS (en simple précision) sur un Knights Landing 7250, ce qui représente 67% de la puissance crête. Cela permet d'envisager des simulations à plusieurs centaines de températures, ce qui est impossible avec une approche classique.

Même si ces efforts d'optimisation améliorent significativement la performance, ce calcul

à la volée reste encore beaucoup plus lent que les méthodes de recherche conventionnelles. Les travaux futurs devront donc se concentrer sur les optimisations algorithmiques comme des méthodes de fenêtrage de multipole. Si ce travail s'est concentré sur les architectures many-cœurs, les implémentations sur accélérateur graphique (GPGPU) méritent d'être étudiées pour espérer atteindre des performances plus élevées. D'autre part, la prise en compte de l'efficacité énergétique devient cruciale pour évaluer les algorithmes sur les architectures modernes. Une solution de compromis entre la performance maximale et la consommation d'énergie est désormais nécessaires. Ainsi les algorithmes proposés devront être réexaminés sous l'angle de la consommation énergétique et pas uniquement sur leur performance brute ou leur occupation mémoire.

Acknowledgements

I would firstly like to thank my thesis director, Christophe Calvin, for the opportunity he has given me as a Ph.D. student, and for his lightning-fast supports every time I was in trouble. I am grateful for his wise guidance and inclusiveness throughout my three year's study.

I acknowledge my advisor, Fausto Malvagi, for his patience with me, and his willingness to share his invaluable insights to the field. I really miss the discussions we had every Friday after lunch in his office.

I would also like to thank my advisor, Emeric Brun, for the aids he offered to deal with technical issues. I am grateful to him for remembering my girlfriend's birthday during countless ssh connections.

I thank François-Xavier Hugot for introducing OCaml as well as his skillful codes. I want to express my gratitude to Philippe Thierry of Intel, for always providing us with the latest hardware. I would like to thank David Chamont for his role as my academical advisor. Specifically, I thank Pierre Guérin of EDF R&D for guiding me into the field of HPC and Monte Carlo transport.

I thank the director of Maison de la Simulation, Edouard Audit, for accepting me to study at this laboratory. I would also thank, Michel Kern for inviting me to give the talk, Valérie Belle and all secretary members for helping me with numerous administrative work, Pierre Kestener, Julien Derouillat, and Martial Mancip for their hardware support, Pascal Tremblin for his follow-up to my study life. Besides, I have had useful talks with Arnaud Durocher, Yacine Ould-Rouis, Matthieu Haefele, Mathieu Lobet, Ksander Ejjaaouani, and Julien Bigot. I thank also all current and previous MDLS members whose names are not listed here.

My gratitude naturally goes to my classmates of the HUST Sino-French project all the way from 2008 in China until now in France. We met each other in the beginning of this adventure and now most of us share a Ph.D. title.

愿我们的故事一直波澜壮阔，祝我们的人生永远平淡如真。

To my family, thank you for your understanding and unwavering support.

Contents

Résumé i

Acknowledgements v

Contents vi

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Monte Carlo Neutronics . . . 1
- 1.2 HPC and Hardware Evolution . . . 2
- 1.3 Motivations and Goals . . . 3
- 1.4 Outline . . . 6

2 Modern Parallel Computing 9
- 2.1 Computing Architectures . . . 9
  - 2.1.1 CPU Architectures . . . 9
    - 2.1.1.1 Memory Access . . . 10
    - 2.1.1.2 Hardware Parallelism . . . 12
    - 2.1.1.3 Sandy Bridge and Broadwell . . . 12
  - 2.1.2 Many Integrated Cores . . . 13
    - 2.1.2.1 Knights Corner . . . 13
    - 2.1.2.2 Knights Landing . . . 16
  - 2.1.3 GPGPU . . . 20
    - 2.1.3.1 Tesla P100 . . . 21
- 2.2 Programming Models . . . 22
  - 2.2.1 OpenMP . . . 22
  - 2.2.2 Threading Building Blocks . . . 22
  - 2.2.3 OpenACC . . . 23
- 2.3 Vectorization . . . 23
  - 2.3.1 Methodology . . . 27
    - 2.3.1.1 Intrinsics . . . 28
    - 2.3.1.2 Directives . . . 29

2.3.1.3 Libraries . . . 30
2.3.1.4 Auto-Vectorization . . . 30
2.4 Useful Tools for Performance Analysis . . . 31
2.4.1 TAU . . . 31
2.4.2 Intel VTune . . . 32
2.4.3 Intel Optimization Report . . . 34
2.4.4 Intel Advisor . . . 36

3 Monte Carlo Neutron Simulations 41
3.1 Nuclear Reactors . . . 41
3.2 Nuclear Reactions . . . 42
3.2.1 Cross-Section . . . 43
3.3 Effects of Temperature on Cross-sections . . . 45
3.4 Neutron Transport . . . 46
3.4.1 Neutron Transport Equation . . . 46
3.4.2 Monte Carlo Simulation . . . 47
3.5 Simulation Codes . . . 49
3.5.1 MCNP . . . 49
3.5.2 TRIPOLI . . . 49
3.5.3 PATMOS . . . 50
3.6 HPC and Monte Carlo Transports . . . 51
3.6.1 History-Based and Event-Based . . . 54
3.6.2 Accelerators . . . 55
3.6.3 Cross-Section Computations . . . 56
3.6.4 Shared-Memory Model . . . 57

4 Energy Lookup Algorithms 59
4.1 Working Environments . . . 59
4.1.1 Machines . . . 60
4.1.2 `PointKernel` Benchmark . . . 60
4.2 Porting and Profiling . . . 60
4.2.1 Adaptations to KNC . . . 60
4.2.2 Code Profiling . . . 61
4.2.2.1 Profiling on Intel Sandy Bridge . . . 61
4.2.2.2 Profiling on KNC . . . 62
4.3 Binary Search and Alternative Search Methods . . . 62
4.3.1 Manual Binary Search . . . 63
4.3.2 Vectorized N-ary Search . . . 64
4.3.3 Vectorized Linear Search . . . 65
4.3.3.1 Data Alignment for C++ Member Variables . . . 66
4.3.4 Comparison of Different Search Algorithms . . . 67
4.4 Unionized Energy Grid . . . 69
4.4.1 Optimizations for the Unionized Method . . . 69
4.4.1.1 Initialization . . . 69
4.4.1.2 Data Structure . . . 71
4.5 Fractional Cascading . . . 71
4.5.1 Reordered Fractional Cascading . . . 72

4.6 Hash Map . . . 73
4.6.1 Isotope Hashing . . . 73
4.6.2 Material Hashing . . . 74
4.6.3 Efficient Hashing Strategies . . . 74
4.6.3.1 Hashing Size . . . 74
4.6.3.2 Logarithmic vs. Linear Hashing . . . 75
4.6.3.3 Search Efficiency within Hashing Bins . . . 75
4.7 N-ary Map . . . 76
4.8 Full Simulation Results . . . 78
4.8.1 Performance . . . 78
4.8.2 Memory Optimization . . . 78
4.8.3 Scalability . . . 79
4.8.4 Results on Latest Architectures . . . 81

5 Cross-section Reconstruction 83
5.1 Working Environments . . . 83
5.1.1 Machines . . . 83
5.1.2 `PointKernel` Benchmark . . . 84
5.2 Algorithm . . . 84
5.2.1 Resolved Resonance Region Formula . . . 84
5.2.1.1 Single-Level Breit-Wigner . . . 84
5.2.1.2 Multi-Level Breit-Wigner . . . 87
5.2.1.3 Doppler Broadening . . . 87
5.2.2 Faddeeva Function . . . 88
5.3 Implementations and Optimizations . . . 88
5.3.1 Faddeeva Implementations . . . 88
5.3.1.1 ACM680 W . . . 89
5.3.1.2 MIT W . . . 89
5.3.2 Scalar Tuning . . . 90
5.3.2.1 Algorithm Simplification . . . 91
5.3.2.2 Code Reorganization . . . 91
5.3.2.3 Strength Reduction . . . 92
5.3.2.4 STL Functions . . . 93
5.3.3 Vectorization . . . 93
5.3.3.1 Collapse . . . 93
5.3.3.2 No Algorithm Branch . . . 94
5.3.3.3 Loop Splitting . . . 95
5.3.3.4 Declare SIMD Directives . . . 95
5.3.3.5 Float . . . 96
5.3.3.6 SoA . . . 97
5.3.3.7 Data Alignment and Data Padding . . . 97
5.3.4 Threading . . . 97
5.4 Tests and Results . . . 98
5.4.1 Unit Test of Faddeeva Functions . . . 98
5.4.1.1 Preliminary Numerical Evaluation . . . 99
5.4.2 Reconstruction in PATMOS . . . 100
5.4.2.1 Unit Test of Cross Section Calculation . . . 100

5.4.2.2 Performance in `PointKernel` . . . 101
5.4.2.3 Memory Requirement . . . 102
5.4.2.4 Roofline Analysis . . . 103
5.4.3 Energy Lookups vs. On-the-fly Reconstruction . . . 104

6 Conclusion and Perspective **107**
6.1 Conclusion . . . 107
6.2 Future Work . . . 109

Bibliography **111**

List of Figures

2.1 Computer architectures. 10
2.2 Modern memory hierarchies composed of register, cache and memory. . . 11
2.3 Hardware-level parallelism where T is *thread*, C is core, MC is memory controller. 12
2.4 Abstraction of a Knights Corner coprocessor core. 14
2.5 The ring organization of a Knights Corner coprocessor. 15
2.6 The organization of a Knights Landing processor. 16
2.7 Memory modes in Knights Landing processor. 18
2.8 The architectural difference between CPUs and GPUs: CPUs are *narrow and deep*, while GPUs are *wide and shallow*. 20
2.9 Pascal P100 streaming multiprocessor. 21
2.10 Vectorization is the process of rewriting multiple independent instructions with one SIMD instruction. 24
2.11 *Summary* report of a *Basic Hotspots* analysis. 32
2.12 *Bottom-up* report show hotspots from the largest to the smallest. 33
2.13 Identify instructions in the source code corresponding to the hotspot. . . . 33
2.14 Advisor analysis with trip counts and FLOPS for the target program on a *Broadwell* architecture. 37
2.15 Direct Roofline analysis with Advisor. 38
2.16 An example of the naive Roofline model. 38

3.1 Energy-dependent cross-section tables $^{1}\mathtt{H}$ and $^{238}\mathtt{U}$. 43
3.2 Isotopic energy table lengths for 390 isotopes at $T = 300$K. The minimum is for $^{3}\mathtt{H}$, which has only 469 energy points, and the largest is for $^{238}\mathtt{U}$ with 156,976 points. The average length over all isotopes is around 12,000. . . 44
3.3 The random walk process of a neutron from its birth to disappearance. . . 48

4.1 *Caller-Callee* view for `PointKernel` on Intel Sandy Bridge. 62
4.2 TAU profiling for `PointKernel` on Intel Sandy Bridge. 63
4.3 Performance of several versions of search method as a function of array size. The standard binary search has been tested in two different implementations, one coming from the STL *std::lowerbound* and one rewritten for this work (Manual Binary). 68
4.4 The average hash bin size for an isotope varies significantly from each other. 75
4.5 Balanced bin size with N-ary map method. 76
4.6 Performance comparison between naive and memory-optimized prototype (the lower the better). 79
4.7 Speedup for energy lookup algorithms for the `PointKernel` test case. . . . 80
4.8 Execution performance of competing energy lookup algorithms. 81

4.9 KNL shows 1.5× performance improvement over KNC for energy lookup algorithms by using hyper-threading. 82
4.10 BDW shows > 2× speedup over KNL for energy lookup algorithms. . . . 82

5.1 On-the-fly Doppler broadening applies on the resonance region. 85
5.2 Performance tests of 100,000,000 calls of Faddeeva function on a single thread. 99
5.3 FLOP usage of the `PointKernel` benchmark on BDW and KNL. 102
5.4 Roofline Analysis of the naive implementation on KNL. 103
5.5 Roofline Analysis of the optimized implementation on KNL. 104
5.6 The performance effect of the memory mode on KNL. 105

List of Tables

2.1 SIMD extensions in the Intel processor family. 25

4.1 Time (s) to initialize an unionized grid with different methods. 70
4.2 Speedup of optimized unionized method for the *PointKernel* test case. . . 71
4.3 Effects of isotope ordering when constructing fractional cascading maps for the *PointKernel* test case. 72
4.4 Search performance in the hash bin (with `N`=500). 76
4.5 Performance and memory usage for energy lookup algorithms for the *PointKernel* test case. 78
4.6 Performance and memory usage after memory optimization. 79

5.1 Elapsed time (s) of 100,000,000 calls of Faddeeva functions on a single thread with accumulated optimization efforts. 98
5.2 Accuracy evaluation of Faddeeva implementations. 100
5.3 Evaluation of cross section calculation rate. 100
5.4 Timing performance (s) per batch with accumulated optimizations. 101
5.5 Timing performance (s) per batch of cross section computation methods. 104

Chapter 1

Introduction

Energy is not created out of thin air but can be only converted from one form to another. Compared to other energy sources like coal or natural gas, the nuclear source is "factor of a million" more powerful [1] due to the dense atomic nucleus representing nearly all the mass of an atom but six orders of magnitude smaller in diameter. The nucleus is very stable since protons and neutrons inside it are bound together by the strong nuclear force. This stability can be broken by certain reactions, for example, when a fissile nuclide absorbs a free neutron, it has a very high probability to become unstable and splits into smaller parts. During this splitting process or fission, a large amount of energy is released. Current nuclear power plants profit from such energy to generate heat, which then drives steam turbines to produce electricity. Nuclear power has been considered as an important solution for the energy crisis [2] since it has one of the lowest carbon dioxide emission out of all commercial base-load electricity technologies [3]. Nowadays, France derives about 75% of its electricity from nuclear energy, which is the highest percentage in the world [4].

1.1 Monte Carlo Neutronics

Considering that nuclear energy is so powerful and so widely used, we should give careful thought to the design and the safe operation of all devices related to nuclear interactions like nuclear reactor and the Large Hadron Collider (LHC) [5] at European Organization for Nuclear Research (CERN), where the Monte Carlo (MC) method is used to interpret a large amount of experimental data [6]. More precisely, neutron transport is dedicated to study the motions of neutrons with materials and go a step further to understand how reactors will behave in given conditions. As for modelization, this problem can be

described by the Boltzmann Transport Equation (BTE). There are two prevalent approaches to solve this equation: the stochastic and the deterministic. Within these two approaches, the stochastic MC method is largely applied in the nuclear community to perform reference calculations [7, 8] because it is numerically-accurate with few approximations. This method simulates the physics by following a neutron inside a system from birth to absorption or leakage. The entire trajectory during a neutron's life-cycle is called a *history*. All random movements in a history are governed by interaction probabilities described by microscopic cross-sections. Macroscopic quantities like neutron densities can be estimated by repeating a large number of histories, which makes the MC method much more computationally expensive than the deterministic. As can be imagined, the slow convergence of the MC method leads to the fact that simulations need to be run on large supercomputers to produce results in reasonable time for real-case problems.

1.2 HPC and Hardware Evolution

High-Performance Computing (HPC) refers to the use of aggregated computing power in order to solve large problems which can not be handled on a personal desktop due to various factors, for example, limited computing capability, unaffordable memory requirement and so on. By assembling a large number of computing units into one system, HPC accumulates computational capability via the arithmetic operation and data collection across individual units within the platform. It gathers technologies such as architecture, system, algorithms, and applications under a single canopy to solve problems effectively and quickly. An efficient HPC system requires a high-bandwidth, low-latency network to connect multiple power-friendly nodes or processors. This multi-discipline field grows rapidly due to considerable demands in science, engineering, and business.

Evolution of hardware architectures has great impacts on the HPC community. Prior to 2002, processor performance improves mainly by increasing the number of transistors per integrated circuit [9]. According to Moore's law [10], this number doubles every 18 months. Larger transistor count in an integrated circuit indicates smaller size for each, which allows achieving faster clock rate. Combined with more complicated architectural and organizational designs, programs at that time required few optimizations since its computing capability enhanced involuntarily with higher operational frequency. However, such free lunch is over: overheating brought by higher frequency is approaching the limit of air cooling, not to mention that there is a cubic dependency between clock speed and power consumption. Here comes the famous *power wall*: processors making a dramatic jump in frequency and power requirement show much less improvement in performance due to thermal losses. From then onwards, the concept of modern architectures

with multiple processors on-chip, simpler pipeline and lower operating frequency comes into focus. One representative of such emerging accelerators is the Intel Many Integrated Core Architecture (MIC) [11]. It exists other competitors like general-purpose computing on graphics processing units (GPGPU), the many-core SW26010 [12], the Matrix-2000 GPDSP (General-Purpose Digital Signal Processor) [13], the Sun SPARC64 [14], and so on. In the November 2016 Top 500 list [15], 86 systems out of 500 use accelerator or coprocessor technology. Moreover, these 86 systems take up a large portion of the top 117 supercomputers with performance greater than one PFLOPS (Peta Floating-Point Operations Per Second). Combined with or without the host CPU (Central Processing Unit), low-frequency accelerators have begun playing an important role in the community. Jointly with the increasing computing power, the growing disparity of development between processor and memory, referred as *memory wall*, is becoming more and more serious. In contrast to the annual 50% increase rate of processor speed, memory speed only improves 10% per year [16]. It has been predicted that memory speed would become an overwhelming bottleneck in computer performance [9]. This *Two Wall* problem is intractable since every time we focus on optimizing one wall, we aggravate another.

1.3 Motivations and Goals

Computational science represents a broad discipline that uses HPC to understand and solve complex problems, among which MC neutronics has been selected as one of the twenty-two key applications to conduct with the oncoming *exascale computing* [17]. Exascale represents a system with more than one exaFLOPS performance. Compared to today's petascale computing, exascale is supposed to achieve 1000× higher performance by preserving current consumption level [18, 19]. In order to carry out this, new processor architectures remain still unknown but will surely operate at low clock rate thus the number of processing units on a single chip will have to increase [20]. Memory bandwidth and capacity will probably not keep pace with the improvement in FLOPS [21], which makes certain current algorithms no longer practical. As a result, modern algorithms should be designed to have high FLOP usage and reduce data movement as much as possible for both energy and performance concerns. Memory-bound algorithms will suffer more and more from future architectures and will be outperformed by alternative compute-bound ones.

The fundamental of MC transport has been established since the appearance of scientific computing [22]; though some new physics came out from time to time to improve it, the principle remains the same. In order to prepare MC transport for exascale, several issues should be taken into serious consideration:

- `Demand`: evolved along with hardware development, the numerical expectation to Monte Carlo calculations progresses constantly. What the community needs is not to solve current problems more rapidly with more advanced hardware, but to perform efficiently much more detailed simulations with larger machines, for example, we used to simulate a physic-simplified problem with two or three temperatures but nowadays hundreds of temperatures are required instead.

- `Challenge`: given the demand of MC transport and the actuality of hardware evolution, how to carry out MC calculations with modern or future HPC facilities is not evident. As stated before, the principle of MC calculations has been set up since the 1940s. Though several algorithms perform well for a long time and would still be the most effective for the foreseeable future, they have been found using less computing resources compared to the past. In other words, the improvement on hardware performance brings less so on simulation performance. To some extent, this situation is caused by the *top500* list where all supercomputers are evaluated and recognized by solving the LINPACK benchmark [23] — one specific linear algebra problem scaling easily and therefore, applicable for any machine no matter its size or structure. As a result, manufacturers offer massively regular vectorization support at the hardware level to achieve a higher ranking, by ignoring the fact that this may make no sense to a lot of real case problems related to science and engineering. Common MC transports show little SIMD (Single Instruction on Multiple Data) opportunities thus can be a typical example of this issue. On the other hand, more detailed simulations will require more complicated numerical and programming model. Though the MC process is intrinsically parallel and requires a few communications, it will still be a challenge when the parallelism degree increases to the level of billion. Another major issue is memory space: as can be imagined, memory requirement will increase in order to perform more physically accurate calculations, thus today's voluminous nuclear data sheet will not comply with the future trend.

- `How`: one direct idea to answer the challenges mentioned above is co-design, which refers to a computer system design process where scientific problem requirements influence architecture design and technology and constraints inform formulation and design of algorithms and software [24]. It requires the cooperation of hardware architects, system engineers, domain scientists, HPC experts to work together and balance benefits and drawbacks in the design of the hardware, software and underlying algorithms. For instance, an ideal machine for the conventional (binary search on pretabulated cross-section data) MC calculation should consist of:

- simplified memory hierarchy by removing data cache system since data access is totally random and cache contributes little to overall performance;
- low latency and large space memory set as closely as possible to the processor while memory bandwidth is not crucial;
- no vector units but only scalar pipeline;
- shorter pipeline stages since no complex arithmetic operation to deal with;
- hyper-threading support to hide data load overhead and keep processor busy;
- a large number of cores within a processor to fit the embarrassingly parallel computation;

As can be observed from the above features, several of them strongly oppose the hardware evolution trend. In fact, actual architectures are not driven by this co-design process but by the market and hardware vendors. Removing cache means no data locality, which may be helpful for some specific problems but turns out to be extremely inefficient for common use. Without vector units, the processor can only have the same performance at the price of much more power consumption. In a word, this advanced co-design methodology is not realistic for the current computational science community due to various difficulties like demand, practicality, and funding. Consequently, the most practical way of modern computational science is that software developers are responsible to transform their codes to conform to hardware features. More precisely for MC transports, simulations should be performed with small memory requirement and high vectorization usage.

The cross-section calculation has been figured as the number one performance bottleneck of MC simulations since this process is high-frequently repeated and can take up to 80% of overall computing time [25]. This is the case in particular with criticality calculations where many isotopes and few tallies present. A conventional way to perform this lookup is using the binary search, but retrieving data in large tables with the bouncing search method results in high cache misses and therefore degrades efficiency. Previous studies focused on this issue have already proposed several solutions. However, all the proposals are latency-bound due to the lookup nature. Moreover, the pretabulated data requires a great deal of memory space: in order to simulate the more than 400 isotopes available in a neutron cross section library, more than one GB memory space is required for one single temperature. In simulations coupled with thermal-hydraulic feedback where hundreds or thousands of different temperatures are involved, cross section calculations by energy lookups become infeasible for contemporary memory volume. An alternative cross-section reconstruction can resolve the problems relevant to energy lookups. The basic idea behind the reconstruction is to do the Doppler broadening (performing a

convolution integral) computation of cross sections on-the-fly, each time a cross section is needed, with a formulation close to standard neutron cross section libraries, and based on the same amount of data. This approach largely reduces memory footprint since only several variables for each resonance are required instead of the conventional pointwise table covering the entire resolved resonance region. There is no more frequent data retrieving, so the problem is no longer memory-bound. The trade-off is that such on-the-fly calculations are even more time-consuming than received energy lookups.

1.4 Outline

This dissertation is dedicated to optimize the cross section computation of MC codes and investigate its performance potential by using modern computing architectures. Firstly in Chapter 2, the concept of computer architecture and three emerging accelerators, Intel Knights Corner, Intel Knights Landing, and Nvidia P100, will be introduced with details. Apart from hardware, programming models, useful tools and vectorization techniques related to the current work will be discussed as well.

Chapter 3 will cover background information around MC transport. It gives a brief introduction to nuclear analysis, Monte Carlo method, physics, and mathematics of kinetics involved in neutron transport. Cross section and Doppler broadening will then be explained since they stand for the primary performance issue. Besides, two reference codes and the PATMOS prototype that we are working with will be presented. At last, a discussion about HPC and Monte Carlo transport calculations will present the state-of-the-art progress in the community.

Chapter 4 focuses on competing energy lookup algorithms of cross section computation. It begins with algorithm presentation of each proposal. Then, it describes vectorizations and other optimizations applied in this work. All efforts are firstly tested in a stand-alone setting independent of PATMOS. After that, numerical results and algorithms will be evaluated in a full neutron simulation with PATMOS in terms of performance, scalability and memory footprint.

The following chapter introduces the on-the-fly cross section reconstruction. It will firstly present the initial multipole cross-section representation and the Faddeeva function. Then, it will detail our alternative cross-section reconstruction and govern equations behind it. Corresponding optimizations and implementations will be fully explained. Evaluation of these efforts will take place in terms of numerical validation, computing

performance and power consumption on modern multi-core and many-core architectures. FLOP usage and program limitation will be investigated with Roofline analysis model [26].

The final chapter draws conclusions from the previous chapters. In addition to respective conclusions for both energy lookup and reconstruction. Global comparison between these two paths will be analyzed as well. After these concluding remarks, a future roadmap for cross-section calculation is proposed. Furthermore, other threading and optimization techniques for PATMOS to perform efficient on-node Monte Carlo transport parallelism will be enumerated.

Chapter 2

Modern Parallel Computing

2.1 Computing Architectures

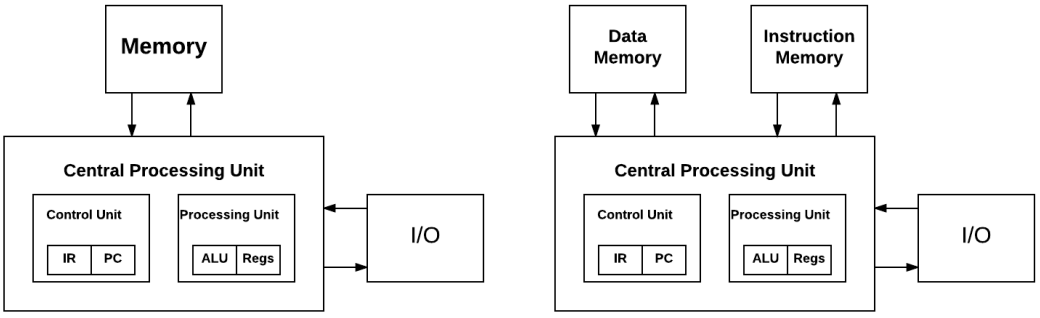
Each hardware platform performs calculations in its own way. In order to have a code running efficiently on target architectures, any optimization efforts should be based on deep understandings of the target hardware. In this section, state-of-art processors and accelerators related to the thesis work will be introduced.

2.1.1 CPU Architectures

The *von Neumann architecture* introduced by the Hungarian-American mathematician and physicist John von Neumann [27] is valued as the "referential model" of computer architectures. It is always said that current computing devices are all evolved from this concept. As shown in Figure 2.1(a), this stored-program system consists of four parts:

- `Control unit`: comprised of an instruction register (IR) and a program counter (PC), within which the IR is responsible to hold instructions currently decoded or executed. The PC is used to indicate the advancement of the executing program. It is a counter that accumulates itself after executing each instruction, and in the meanwhile points to the next instruction to be executed;
- `Processing unit`: comprised of an arithmetic logic unit (ALU) and registers. ALU is a fundamental component for modern computing architectures since it handles arithmetic operations on operands and carries out calculations. Registers are a small-size storage system responsible to provide fast accesses for both data and instructions;

- `Memory`: large storage mechanism to hold data and instructions with much higher latency;
- `I/O`: input and output devices.



(a) Von Neumann architecture (b) Harvard architecture

FIGURE 2.1: Computer architectures.

The main problem of the *von Neumann architecture* is the limited data transfer rate between CPU and memory. Since both data and instructions in memory share the same bus to communicate with CPU, computers can not handle data operations and instruction fetches simultaneously. Such low throughput makes processing unit running idle and always waiting for data. In order to deal with this *von Neumann bottleneck*, several mitigations are applied to solve the problem:

- set up an intermediate memory layer between memory and CPU (for example, cache and scratchpad memory);
- individual accesses to data and instructions instead of the single shared path.

This improved concept, called *Harvard architecture* (Figure 2.1(b)), is the fundamental of modern computing devices. In fact, data and instructions paths are usually separated at the cache-level for modern chips. Between the main memory and the CPU, however, instructions are still handled as if they were data for efficiency concern thus the processor works more like a *von Neumann architecture*. Such half-*von Neumann* half-*Harvard* concept is referred as the *modified Harvard architecture*, but always loosely documented as the *Harvard architecture*.

2.1.1.1 Memory Access

As shown in Figure 2.2, modern memory systems are usually composed of several stages to overcome the *von Neumann bottleneck* and provide high-efficiency data/instruction

access. Registers are usually at the top of the memory hierarchy and provide the fastest data transfer rate among the entire memory system. Target data to be dealt with arithmetic or logic operations (in the form of assembly languages) will be temporarily held in the register to carry out computations. Due to the limited number of registers, they are identified from each other by distinct names but not different memory addresses. Cache is an intermediate memory layer made of Static Random-Access Memory (SRAM). Unlike the data management in registers, developers do not have full control of data load by using explicit instructions. Data manipulation at the cache level are directly built on die and handled by the hardware *memory controller* (Figure 2.2). Generally, the cache follows the *Least Recently Used* (LRU) policy to allocate and release data by the unit of a cache line. Every time the *memory controller* indicates a new address, the cache will load not an arbitrary amount but one cache line (typically 64 bytes) of data. This notion is really important for code optimizations since it changes a lot the way to profit from cache and data reusability. For many code developers, the register has so small a size and the memory is so far away from the processor that making good use of cache is the most practical way to improve the data transfer efficiency. Locating at the bottom of the hierarchy, memory has the lowest bandwidth and the highest latency compared to register and cache. It is made of Dynamic Random-Access Memory (DRAM) in which binary bits are represented by the high-level or low-level of small capacitors. Compared to the SRAM used by cache, DRAM sacrifices transfer efficiency for the privilege of fabrication cost.

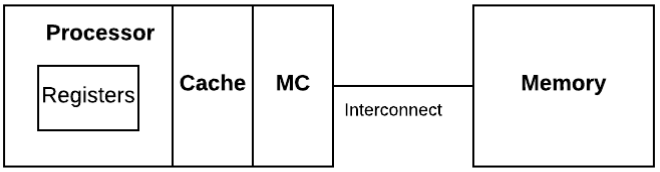
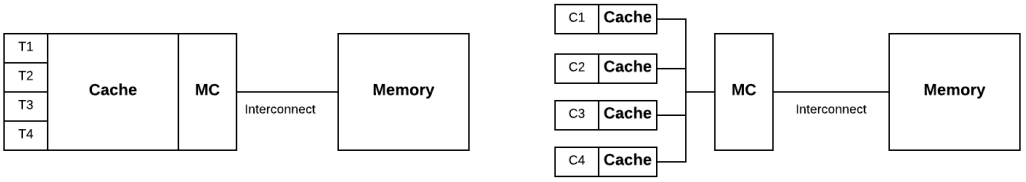


FIGURE 2.2: Modern memory hierarchies composed of register, cache and memory.

The trend in hardware has been towards many cores in one processor, which leads to a problem for the traditional Symmetric MultiProcessing (SMP) model where a large number of cores must compete for data access through the unique data bus. One direct solution is to use multiple system buses and each of them has its own I/O ports. Every bus serves only a small group of cores and memory. All groups combined together is determined as a Non-Uniform Memory Access (NUMA) architecture, each group is called a NUMA node. Data access to memory of remote nodes takes longer than that of the local node memory. Local and remote memories are usually used in reference to a currently running thread where local memory indicates the memory on the same

node as the running thread while remote memory is the memory that does not belong to the node on which the thread is running. Though both local and remote memories are transparent to each hardware thread like before, such non-uniform memory access pattern requires more work and carefulness for program developments. It is important to make the running thread using local memory as much as possible to avoid additional access overhead (thread affinity control).

2.1.1.2 Hardware Parallelism



(a) Hyper-threading (b) Multi-threading

FIGURE 2.3: Hardware-level parallelism where T is *thread*, C is core, MC is memory controller.

Based on the *Harvard architecture*, modern processors employ hyper-threading, multi-threading and SIMD techniques to reinforce their computing power. As shown in Figure 2.3(a), multiple hardware threads share the same core (or computing resources) and appear as if there were multiple cores for software developers that can execute different instructions in parallel. Another evolution for hardware parallelism is multi-threading: instead of replicating partially the computing resource, the entire core is cloned multiple times to achieve independently parallel executions for all cores within the processor. As for the vector processing, the SIMD technique (see Section 2.3) is used inside each core where multiple data can be handled with same operations so that this can further extend the hardware parallelism. The change from higher clock rates, which requires significant increase of power consumption to energy-efficient massive parallelism, transforms the code performance work depending more on software engineers than hardware engineers.

2.1.1.3 Sandy Bridge and Broadwell

Intel Sandy Bridge micro-architecture [28] with the 32 nm process was firstly released in early 2011. It implements many new hardware features, among which the AVX (Advanced Vector Extension) vector instructions introduced the renaissance of SIMD technique. Intel Broadwell [29] with 14 nm transistors and the more powerful AVX2 has a lot of improvements like larger out-of-order scheduler, reduced instruction latencies, larger L2 Translation Lookaside Buffer (TLB), and so on. It should be noted that in

this thesis, we always compare one MIC with a dual-socket CPU because they share nearly the same price and power consumption.

2.1.2 Many Integrated Cores

The performance improvement of processors comes from the increasing number of computing units within the small chip area. Thanks to advanced semiconductor processes, more transistors can be built in one shared-memory system to do multiple things at once: from the view of programmers, this can be realized in two ways: different data or tasks execute in multiple individual computing units (multi-thread) or in long uniform instruction decoders (vectorization). In order to answer the need of parallelism and vector capabilities, Intel initially introduced wide vector units (512 bits) to an x86-based GPGPU chip codenamed *Larrabee* [30]. This project was then terminated in 2010 due to its poor early performance but techniques around are largely inherited by the latter high-performance computing architecture – Many Integrated Cores.

The initial MIC board, codenamed *Knights Ferry* was announced just after the termination of the *Larrabee*. Inheriting the ring structure of the initial design, *Knights Ferry* has 32 in-order cores built on a 45 nm process with four hardware threads per core [11]. The card supports only single precision floating point instructions and can achieve 750 GFLOPS. Like GPGPUs, it works as an accelerator and connected to the host via PCI (Peripheral Component Interconnect) bus.

2.1.2.1 Knights Corner

The first commercial MIC card after the prototype *Knights Ferry*, named *Knights Corner* (KNC) [31], follows the same design but is built with the more advanced 22 nm process, which allows up to 61 cores working at 1.1 GHz and begins to support double-precision operations.

Each core is very similar to the older Intel Pentium processor [33]. Figure 2.4 shows the scalar and vector pipelines inside a KNC core and the instruction/data flow model. Each core supports four hyper-threads to keep pushing data into processing units and hide memory load latency. It has a two-level cache system in which the L1 cache consists of a 32-KB L1 instruction cache and a 32-KB L1 data cache. The 512-KB L2 cache is unified since it does not distinguish data and instructions. Every computing core has an L2 cache, and all L2 caches are coherent via the bi-directional ring network. Inside of each core, there are two individual pipelines (U-pipe and V-pipe) allowing it to execute two instructions per cycle: one on U-pipe and the other on V-pipe [34]. The MIC

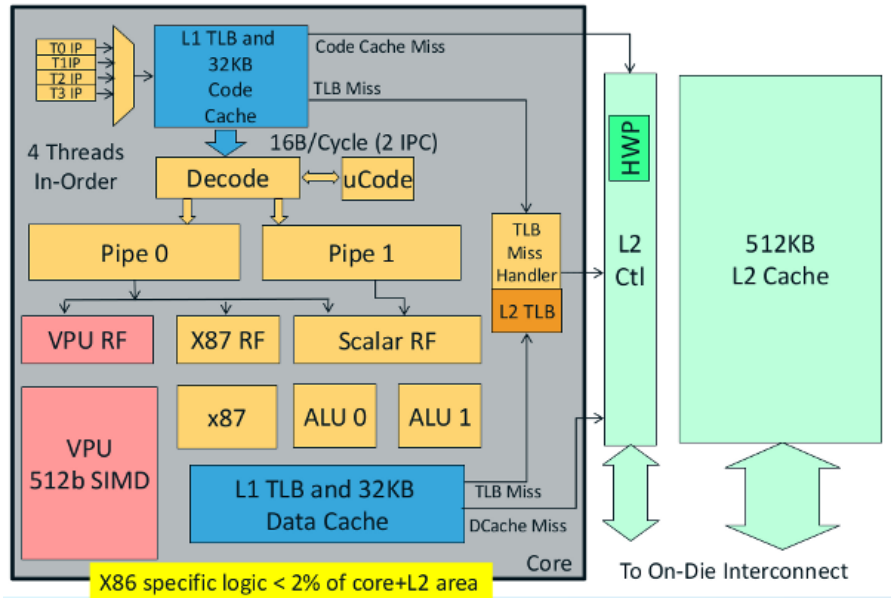


FIGURE 2.4: Abstraction of a Knights Corner coprocessor core [32].

architecture has relatively short pipelines compared to CPUs. Instruction prefetching follows a round-robin order so the currently running thread can not deal with instructions continuously. As a result, using hyper-threading turns out to be necessary for KNC to hide such overhead and achieve optimum performance. The theoretical performance of a KNC coprocessor with 60 usable cores can be calculated as follows:

- `Single-precision`: 16 (SP SIMD lane) × 2 (*fused multiply–add* or FMA) × 1.1 (GHz) × 60 (cores) = 2112 GFLOPS
- `Double-precision`: 8 (DP SIMD lane) × 2 (FMA) × 1.1 (GHz) × 60 (cores) = 1056 GFLOPS

The wide 512-bit Vector Processing Unit (VPU) is the key feature of MIC. It can simultaneously issue 16 single-precision floats or 32-bit integers; or 8 double-precision floating point variables. Since there are no bypasses between the double and the single, mixing these two elements in a code will cause performance penalties. The VPU state per thread is maintained in 32 512-bit general vector registers, 8 16-bit mask registers, and a status register VXCSR [35]. The VPU is fully pipelined and executes most instructions with 4-cycle latency. KNC implements a novel instruction set architecture (ISA) called Intel Initial Many-Core Instructions (IMCI), which supports 218 new instructions compared to those implemented in the traditional SIMD instruction sets (MMX, SSE, AVX, etc.). The IMCI supports scatter and gather instructions to enable vectorizations over a sparse data layout. It also implements mask operations to work on specific elements of a vector register. It should be noted that common operations (for example, unaligned data load)

are not fully supported by the preliminary IMCI, which make optimizations by directly using intrinsics relatively difficult compared to other SIMD instruction sets.

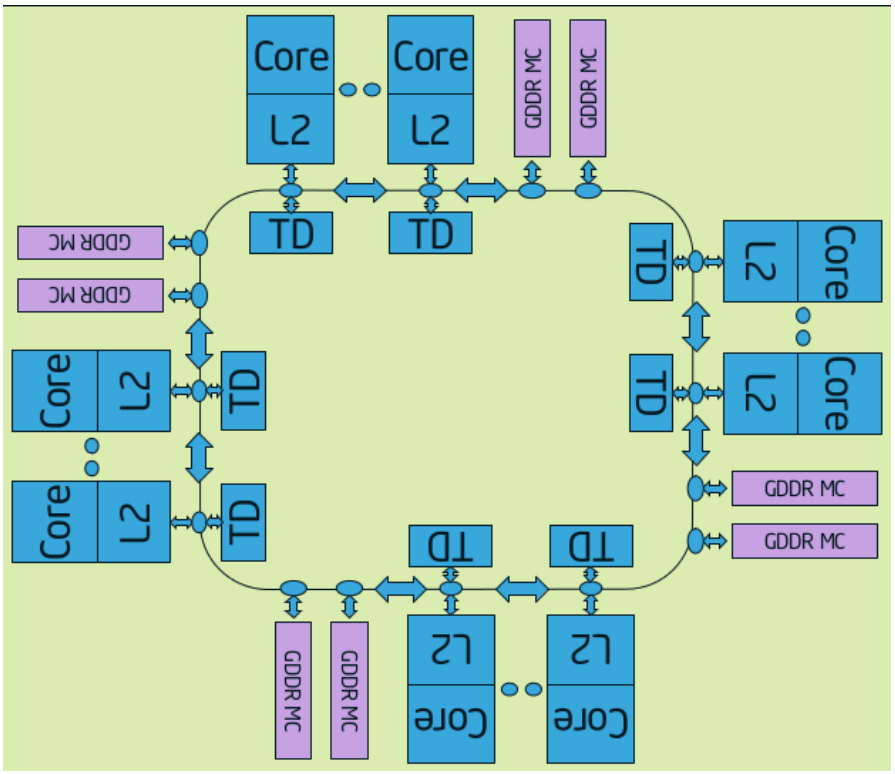


FIGURE 2.5: The ring organization of a Knights Corner coprocessor [32].

As mentioned before, cores are connected together via a ring structure (Figure 2.5). The bi-directional ring is composed of ten system buses, five in each direction, which allows sending data across the ring once per clock per controller. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 (Graphics Double Data Rate) memory (bandwidth ~350 GB/s) on the coprocessor and the PCIe bus, respectively. Core-private L2 caches are coherent by the global-distributed *tag directory* (TD). Though operands found on remote L2 caches will be literally determined as a *cache hit*, a longer access time via the global TD does actually cause a *cache miss*. KNC has a Linux-based micro-OS (*operating system*) running on it. The OS takes up a core to deal with hardware/software requests like interrupts, so it is a common practice to use 60 cores out of 61 to perform pure computation tasks.

Serving the host CPU as an accelerator card, KNC has three working models:

- `Offload mode`: the MIC card works like a GPGPU. This execution model is also known as the heterogeneous programming mode. The host CPU offloads partial or whole calculations to the accelerator via PCIe bus. The application starts and

ends execution on the host, but assigns certain sub-work to execute in the remote during its lifecycle;

- `Native mode`: since the coprocessor hosts a micro OS, it can appear as an independent computing node, and perform the whole calculation all the way from the beginning to the end. This execution environment allows the users to view the coprocessor as another compute node. Native execution requires cross-compilation for the MIC operating environment. The advantage is that such mode avoids the significant data transfer overhead between CPUs and MICs;
- `Symmetric mode`: the coprocessor works in a hybrid way by mixing *native mode* and *offload mode*. Host and accelerator communicate through message passing interface (MPI). This execution treats accelerator card as another node in a cluster in a heterogeneous cluster environment.

In this thesis, we focus only on the *native mode* to evaluate KNC's optimum performance.

2.1.2.2 Knights Landing

Intel officially first revealed the latest MIC codenamed *Knights Landing* (KNL) in 2013 [36]. Being available as a coprocessor like previous boards, KNL can also serve as a self-boot MIC processor that is binary compatible with standard CPUs and boot standard OS [37]. Another key feature is the on-card *high-bandwidth memory* (HBM) which provides high bandwidth and large capacity to run large HPC workloads.

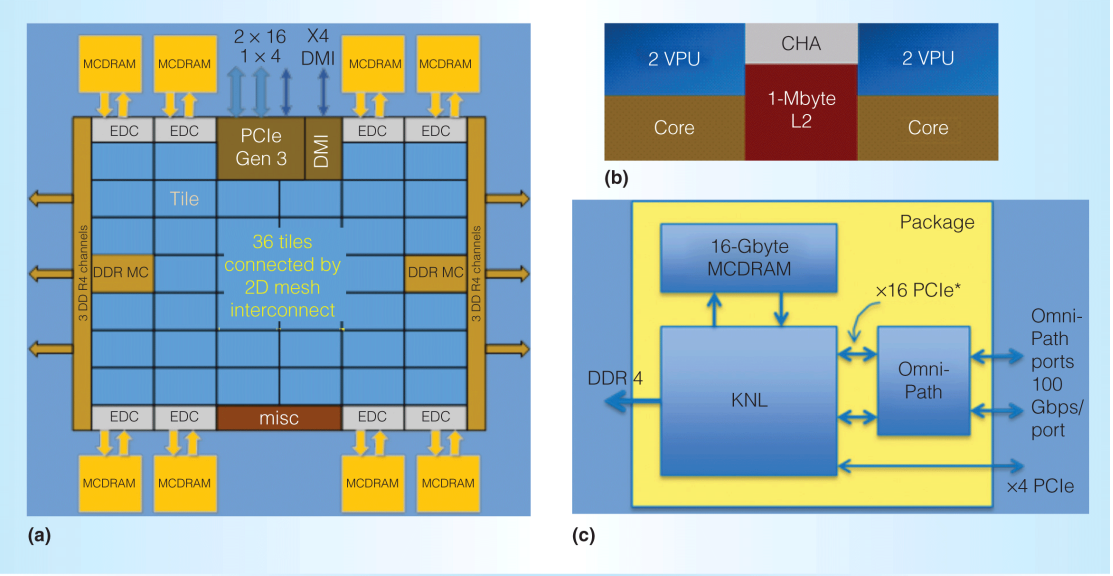


FIGURE 2.6: The organization of a Knights Landing processor [38].

Unlike the previous ring structure, KNL cores based on *Silvermont* microarchitecture [39] are organized as a matrix of up to 36 active *tiles* (Figure 2.6a). Each *tile* comprises two cores which share a one MB L2 cache (Figure 2.6b). The new out-of-order core has 2 VPUs, and includes many changes to better support HPC workloads: higher cache bandwidth, deeper out-of-order buffers, new instructions, etc.

The new 2D cache-coherent mesh interconnect is introduced to link all components on die, such as *tiles*, memory controllers, I/O controllers, etc. All L2 caches are kept coherent by the mesh with the MESIF (modified, exclusive, shared, invalid, forward) protocol [40], which provides more uniform data access with high bandwidth to different parts of the chip. To maintain cache coherency, KNL has a *distributed tag directories* (DTD) to identify the state and the location of any cache line with a hashing function. This change in cache organization requires more complicated hardware and increases the complexity of cache management. In order to prepare the cache for different applications, various cache clustering modes are offered by KNL:

- `All-to-All`: memory addresses are uniformly distributed to all L2 caches. Generally, all cache accesses are remote so the latency of *cache hits* and *cache misses* is long. It is designed not to perform daily computations but only to provide backup solution for hardware fault;
- `Quadrant`: the mesh structure is divided into four *quadrants* where each has its own *memory controller*. It is the default clustering mode in KNL, in which four quadrants are hidden from the OS and appear as one contiguous memory block from the user's perspective [41]. Memory addresses are guaranteed to be locally mapped within the quadrant, so there is no remote memory request at all;
- `Hemisphere`: similar to *quadrant* mode but the mesh is divided into two *hemispheres*.
- `Sub-NUMA clustering (SNC)`: *SNC-2* and *SNC-4* further extend *quadrant* and *hemisphere* components to individual NUMA nodes to the OS. This kind of clustering mode can have the best latency at the price of more NUMA optimizations to deal with.

Memory bandwidth is one of the common performance bottlenecks for computational applications due to the *memory wall*. KNL implements a two-level memory system to address this issue. The first 16 GB HBM based on Multi-Channel Dynamic Random Access Memory (MCDRAM) has a 400 GB/s bandwidth, and the second DDR4 memory can deliver about 90 GB/s with up to 384 GB in size. For any bandwidth-bound

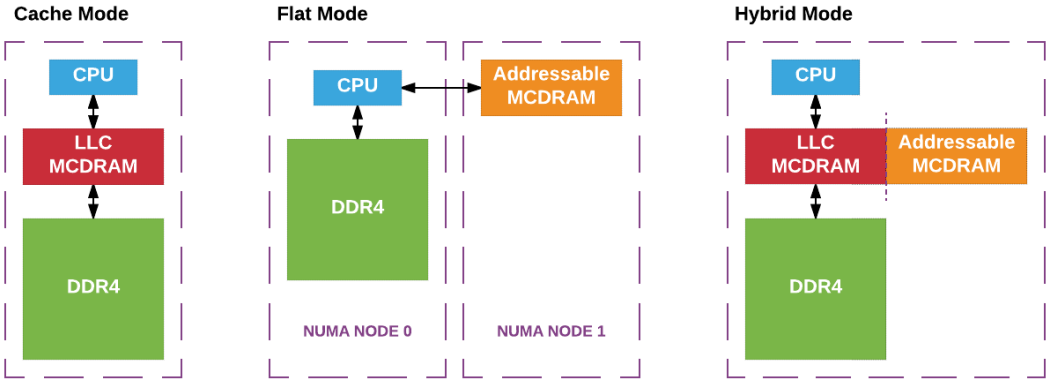


FIGURE 2.7: Memory modes in Knights Landing processor.

applications on KNL, the key to getting optimum performance is to fully profit from MCDRAM. HBM on a KNL can be used either as a *last-level cache* (LLC) or an addressable memory. Like the cache clustering mode in KNL, this configuration is also determined at boot time via BIOS (Basic Input Output System) with three models (Figure 2.7):

- `Cache mode`: MCDRAM is seen as an LLC caching the main DDR4 memory, and itself cached by L2 cache. In this mode, HBM cache is hidden from the platform so no user-side adjustment is required for computing applications. However, deeper cache hierarchy results in longer access to the addressable memory (HBM bandwidth ~300 GB/s);
- `Flat mode`: both MCDRAM and DDR4 are used as addressable memory and cached by L2 cache. Access to the main DDR4 memory is no longer necessary to query HBM (bandwidth ~400 GB/s). CPU and DDR4 are set in one NUMA node while HBM is separated in another node without cores. This mode allows finer manipulations over MCDRAM but also requires more development work since memory allocation will take place on DDR4 by default. Performance tuning around HBM and NUMA configurations need specific optimization knowledge as well as considerable working hours;
- `Hybrid mode`: as shown in the figure, part of the HBM is served as addressable and the rest is used as LLC. The ratio between the two can be chosen at boot time. Generally, this mode is designed for large clusters where different users have their own configuration requirements.

Since there are three memory modes available to perform computations on KNL, how to choose the optimal mode of HBM utilization for target application becomes an important question for developers. Typically, if the program can entirely fit into MCDRAM, which is the ideal case for KNL, the *flat mode* and the *numactl* [42] instructions should

be used. This mode requires no source code changes, and the program can naturally take advantage of the HBM. If the entire program is too voluminous for HBM but the bandwidth-bound kernel within it can fit in HBM. the *flat mode* and the *memkind* [43]) library are recommended. Developers only need to explicitly allocate the critical kernel on HBM with a few code modifications and leave the rest of the code unchanged like before. The worst case is that the program is too large, and it is difficult to extract a critical kernel fitting into HBM. In this case, the *cache mode* is recommended to at least make use of the HBM as much as possible.

By supporting all legacy instruction sets including 128-bit SSE (Streaming SIMD Extensions) and 256-bit AVX technologies, KNL also introduces a new AVX-512 instruction set which is composed of four sub-sets:

- `AVX-512F`: the fundamental instruction set. It contains vectorized arithmetic operations, comparisons, data movement, bitwise operations and so on. It is similar to the core category of the AVX2 instruction set, with the difference of wider registers, and more double-precision support;
- `AVX-512CD`: CD indicates "conflict detection". Previous ISA are not capable to perform basic analysis of data dependencies inside a loop so they have to leave the loop unvectorizable for safety concern. AVX-512CD can improve this situation and allow more loops to be auto-vectorized by the compiler;
- `AVX-512ER`: ER stands for "exponential and reciprocal". Exponential functions and reciprocals are programmed in the card for both single-precision and double-precision variables with rounding and masking options. Previously, such operations were realized via software implementations thus were computationally expensive;
- `AVX-512PF`: the hardware "prefetch" support dedicated to gather and scatter instructions.

There are nine sub-sets in total for the AVX-512 instruction set, of which some are supported by KNL, and some are supported by the upcoming Intel Xeon Scalable Processor Family (Skylake) [44]. If one wants to maintain binary compatibility between KNL and SKL, only AVX-512F and AVX-512CD can be used in the implementation. Otherwise, if optimum code performance on KNL is desirable, using AVX-512ER and AVX-512PF is necessary.

2.1.3 GPGPU

A *graphics processing unit* is an electronic device specialized to rapidly perform image-array operations for outputs to a display device. Graphical pixels presented as large uniform datasets are offloaded from the host CPU to the remote graphic card to carry on certain compute-intensive calculations which are typically linear algebra operations. Before, GPU calculations were hidden to developers since non-graphic operations like looping and floating point math were not supported, and only high-level APIs (Application Programming Interface) were provided to communicate CPU and GPU. Since the turn of the century, GPUs have been found a powerful backup to overcome the *power wall* thus the *general purpose graphics processing unit* has been rapidly developed. With the general purpose support for non-graphic operations, GPGPUs are tailored to deal with a large amount of science and engineering problems that can be abstracted as linear algebra calculations. Due to its specific functionality, the GPGPU will not be a replacement of the main CPU but serves as an accelerator to improve the overall performance of the computer.

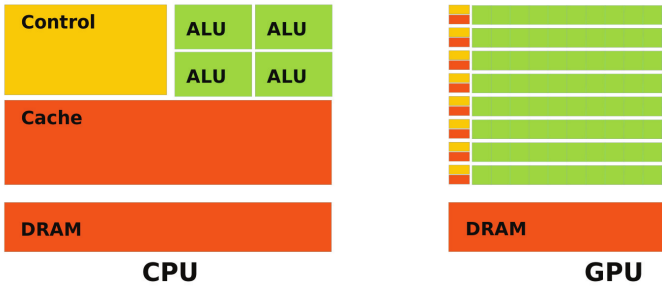


FIGURE 2.8: The architectural difference between CPUs and GPUs: CPUs are *narrow and deep*, while GPUs are *wide and shallow* [45].

Fundamentally, the CPU and the GPU architecture differ in their overall shape: the CPU has a small number of complex cores and massive caches while the GPU has thousands of simple cores and small caches. The host character of the CPU requires complicated cores and deep pipelines to deal with all kinds of operations. It usually runs at higher frequency and supports *branch prediction*. The GPU only focuses on data-parallel image renderings thus the pipeline is shallow. The same instructions are used on large datasets in parallel with thousands of hardware cores, so the *branch prediction* is not necessary, and memory access latency is hidden by important arithmetic operations instead of caching.

2.1.3.1 Tesla P100

Named after the French mathematician Blaise Pascal, Pascal is currently the latest micro-architecture developed by Nvidia. It was firstly introduced with the P100 card in mid-2016. According to Nvidia, one P100 can deliver 5.3 TFLOPS in double-precision floating point thus 10.6 TFLOPS in single-precision [46].



FIGURE 2.9: Tesla P100 streaming multiprocessor [46].

The basic computing unit of one Nvidia's GPGPU is a *streaming multiprocessor* or SM. As shown in Figure 2.9, each *Pascal* SM has been partitioned into two processing blocks. Though the total number of cores in a P100 SM is only half of the previous *Maxwell* SM, the entire card supports more registers within an SM and more SMs within a chip. *Pascal* SM units natively support reduced-precision (16-bit) operations thus could be competitive candidates for many *deep learning* algorithms. In total, one P100 has up to 3840 single-precision CUDA (Compute Unified Device Architecture) cores and eight 512-bit memory controllers. Each memory controller is attached to 512 KB of L2 cache, and each HBM2 DRAM stack is controlled by a pair of memory controllers. The full GPU includes a total of 4096 KB of L2 cache [46]. HBM2 memory is used to address the issue of memory access overhead where it can deliver three times higher memory bandwidth compared to the previous GDDR5. NVLink interconnect is employed to replace the previous PCIe bus for high-efficiency communications (5× memory bandwidth compared to PCIe) among multiple GPUs or between CPU and GPU [47]. Besides, *unified memory* and *atomic operations* are largely improved at both software and hardware level as well.

2.2 Programming Models

After the introduction of physic background and hardware, threading libraries involved in this thesis work will be presented in this section.

2.2.1 OpenMP

OpenMP [48] is a set of directives that extends C/C++ and Fortran languages for parallel executions in shared memory environment. When target machines do not support OpenMP, these directives can be ignored by the compiler and the program can execute in a sequential way. By switching on the *-qopenmp* option for Intel compilers, *pragmas* or directives are interpreted to guide the compiler for parallel execution. Using OpenMP to parallelize a serial program is simple but requires a lot of specific knowledge for profit its full efficiency. Since OpenMP 3.0 [49], it has begun to support *task* scheduling strategies. It includes (since version 4.0 [50]) also SIMD directives to aid high-level vectorization. Besides, it has been extended with *offload* directives to perform calculations in heterogeneous systems [51, 52].

The most common use of this API is to distribute a loop in parallel:

```
#pragma omp parallel for
for (...)
    do some calculations...
```

By using the "fork and join" execution model, the initial thread executes the program sequentially until the *parallel* construct is encountered. Then, this master thread creates a number of child threads (a *thread pool*) and distributes workload for each of them in order to have all threads work simultaneously. Finally, all threads *join* at the end of the *parallel* construct. In OpenMP, all threads have access to the same shared global memory. Each thread has access to its private local memory. Threads synchronize implicitly by reading and writing shared variables. No explicit communication is needed between threads.

2.2.2 Threading Building Blocks

Threading Building Blocks (TBB) [53] is a C++ template library that provides task-parallel solutions for modern multi-core and many-core systems. It is highly portable since workloads are organized as *tasks* instead of *threads* and thread management to

specific architectures are abstracted by the universal C++ templates. TBB also provides subsets of C++ native threading, thread-safe data containers, scalable memory allocators, mutual exclusions, and atomic operations.

TBB is designed to work without any compiler changes. In order to use TBB in the program, developers simply include the $< tbb/tbb.h >$ header and functions can be found in *tbb* and *tbb:flow* namespaces. For a loop:

```
for(i = first; i < last; ++i) f(i);
```

the easiest way to parallelize it with TBB is:

```
tbb:parallel_for(first, last, f);
```

Where *first* and *last* indicate the begin and the end of the loop, and *f* represents the calculations to perform in each iteration. Complex loops can be performed by other high-level templates like *parallel_do*, *parallel_reduce*, *parallel_pipeline* and so on. If the algorithm does not map onto one of the high-level templates, developers can also create their own templates with the optional *task scheduler*.

2.2.3 OpenACC

OpenACC (Open ACCelerators) [54] is a portable parallel programming model dedicated to heterogeneous CPU/GPU systems. Support of OpenACC is available in PGI compilers since the version 12.6 [55]. Experimental GCC support for OpenACC has been included since the version 5.1. Similar to OpenMP, it utilizes compiler directives to perform high-level parallelizations for C/C++ and Fortran source codes: *#pragma acc parallel* and *#pragma acc kernels* can be used to define the *parallel* construct for remote accelerators. *#pragma acc data* and its subsets are responsible to handle data transfer between the host and the remote. *#pragma acc loop* defines the type of parallelism in the *parallel* construct. Precise optimizations like data reduction, loop collapse, asynchronous operation, and multi-device programming can be also realized with corresponding directives. Compared to the native CUDA programming, using OpenACC requires few code modifications and much less parallel-computing skills for developers.

2.3 Vectorization

Processors usually include a set of 16 registers that perform arithmetic operations on integers. These registers are called scalar registers since each of them holds just one variable at any time. A 64-bit processor indicates the width of each scalar register is

64-bit. In order to add two integers together, the program transformed by compiler tells the processor to get the two variables from the scalar registers and add them. Due to the "scalar" nature, if thirty-two pairs of integers are required to perform such addition, the above process will be repeated for thirty-two times:

```
for(int i=0; i<32; ++i)
    C[i] = A[i] + B[i];
```

In recent years, modern processors have added an extra set of large registers (128-bit, 256-bit, or 512-bit) besides the scalar one. Rather than hold a single value, those large vector registers can contain a small set of variables at one time. Together with the help of vector instructions, same arithmetic operations can be carried out simultaneously on multiple data elements. Take the former case, for example, here one 256-bit vector register can hold eight 32-bit integer variables:

```
for (int i=0; i<32; i+=8)
    doEightIntegerAdditionsAndStoreResult(C[i], A[i], B[i]);
```

In the same time that it took to add one pair of integers with the scalar register, a single vector instruction abstracted as *doEightIntegerAdditionsAndStoreResult()* can add eight pairs of integers. Instead of repeating the scalar operation thirty-two times, using the vector instruction repeat the same process only four times. In computer science, such effort by using the vector register as well as its corresponding instructions to rewrite a loop is called *vectorization* (Figure 2.10).

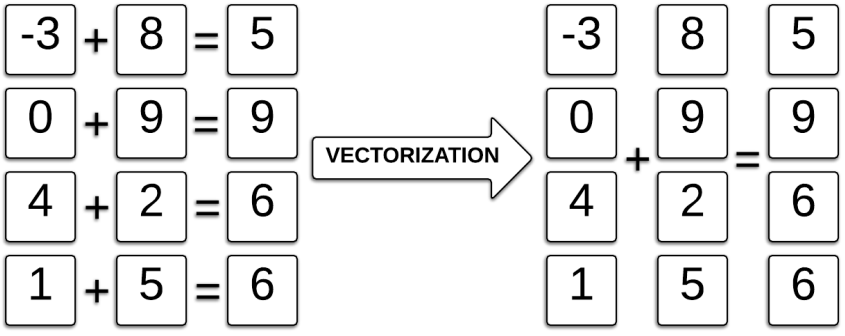


FIGURE 2.10: Vectorization is the process of rewriting multiple independent instructions with one SIMD instruction.

Along with the appearance of vector machines, the notion of vectorization can date back to the early 1970s. Vector machines dominated the supercomputer design for about thirty years until the late 1990s, and then they were gradually replaced by clusters of conventional microprocessors. Modern processors renounce the pure vector design but

ISA	Architecture	Register
MMX	P5-based Pentium	64-bit MM
SSE	Pentium III	128-bit XMM
AVX	Sandy-Bridge	256-bit YMM
IMCI	Knights Corner	512-bit ZMM
AVX-512	Knights Landing	512-bit ZMM

TABLE 2.1: SIMD extensions in the Intel processor family.

adapt vector unit as an additional computing engine besides the scalar pipeline. Table 2.1 shows SIMD extensions for the traditional x86 architectures. Different ISAs can be distinguished by their specific register type. This notion is really important because developers may need to verify the vectorization efficiency by checking assembly codes. Even if a loop compiled for KNL is reported as "vectorized" by the Intel vectorization report (see Subsection 2.4.3), it can happen that most instructions are performed on non-ZMM registers, which indicates that the 512-bit vector unit is underutilized with some 128-bit or 256-bit instructions. In this case, replacing these instructions with the corresponding AVX-512 ones will aid to achieve full vectorization efficiency.

The number one reason for vectorizing a loop is to achieve better execution performance. This is also the key for fully using accelerators like MIC and GPGPU (though SIMD potential fulfilled alternatively). Since vectorization is so important, is this true that all kinds of loops can be vectorized? The answer is no. Take the Intel compiler, for example, generally, vectorization candidates should meet the following criteria [56]:

1. **Constant loop count**: The loop trip count must be constant for the duration of the loop, and the exit of the loop should be independent of the trip count. Uncountable loops can not be vectorizable because the loop exit is not predictable;

2. **No algorithm branches**: The essential of SIMD instructions is to perform the same operations on different data elements. It does not allow different iterations to have different control flow [56] which indicates that there should not be "if" statements inside the loop. Though "mask" assignments can help vectorization sometimes depending on the hardware, they are neither efficient nor feasible all the time;

3. **Innermost loop**: Except for the case by using techniques like loop unrolling and loop collapsing, only the innermost loop of a nest can be vectorized;

4. **No function calls**: A loop with user-defined function calls inside is not vectorizable by intrinsics programming because it is seen as a "nonstandard loop" by

the compiler. Common math operations like *sin* and inline functions are two exceptions. The lastest *declare* and its sub-directives are proved to be a practical solution for this problem (see Subsection 5.3.3.4).

All implementations preventing auto-vectorization can be described with two features: discontiguous memory access and data dependency. Discontiguous memory access can be in different forms:

```
// memory access with a stride of 2
for(int i=0; i<N; i+=2)
    C[i] = A[i] + B[i];

// indirect addressing
for(int i=0; i<N; ++i)
    C[i] = A[B[i]] + B[i];
```

Both non-unit stride access and indirect addressing will high-probably hinder auto-vectorization because the compiler does not vectorize a loop without knowing whether it is efficient to do so or not. An ideal loop is in which data elements written in one iteration will not ever be read or written again in any other iteration of the loop. In most cases, the compiler can not analyze data dependencies by itself so it prevents auto-vectorization by default for safety concern.

After the introduction of requirements and obstacles to vectorization, there are a general guideline for developers to write vectorizable codes:

- Determine a loop with constant trip count. Use the conventional loop form *for(i=start;i<end;++i)* by explicitly defining the begin and the end of the loop, STL loop functions like *std:accumulate()* and *std:for_each()* have not been recognized by the compiler yet.
- Write straight line codes, which means that algorithm branches like *if*, *switch*, *goto* and *return* statements should be avoided in the implementation.
- Design or reorganize a loop to remove potential data dependencies (or at least *read-after-write* dependencies [56]).
- Prefer pointers to STL containers (for example, *std:vector* and *std:list*) to express array notation. High level language-specific containers are not recognizable by the compiler thus will prevent auto-vectorization.
- Use efficient memory accesses: create unit stride loops, avoid indirect addressing, align array data, use SoA (Structure of Array) instead of AoS (Array of Structure).

- Minimize function calls by inlining or using performance libraries like Intel math kernel library (MKL) [57].

2.3.1 Methodology

Efficient vectorization comes from a combination of efficient data movement and recognition of vectorization opportunities in the program. A general six-step approach introduced by Jeffers has been proved really useful to achieve vectorization [58]:

1. **Set up baseline build**: before all profiling and optimization work, using optimization levels 2 or 3 to avoid *debug build*, and enable compiler auto-vectorization to set up a baseline *release build*. This step will help developers to get a first idea of how efficient compilers can do for auto-vectorization. This baseline performance will be a reference to illustrate how following optimizations go on.

2. **Find hotspots**: identifying hotspots will help developers to find which areas of the code take the most execution time, and allow them to focus on the right kernels to carry out following work. According to the Amdahl's law [59], concentrating on unimportant performance bottlenecks or serialization portions of the code will bring few speedups for the overall performance. It is recommended to only focus on functions taking at least 10% of the program's overall execution time [58].

3. **Determine loop candidates**: starting an initial performance investigation with the help of *Intel Optimization Report* (Subsection 2.4.3). This Intel tool will perform a brief vectorization bulletin at compile time. Without running the code, programmers can know whether the hotspot loops identified in the last step are vectorized or not. Moreover, it provides useful information like the theoretical speedup brought by vectorization or why the loop is not vectorized. It should be noted this preliminary profiling is not accurate since it is based on the compile-time check, finer runtime profiling is necessary to achieve full optimization efficiency.

4. **Profile with Intel Advisor**: profiling with the vectorization analysis of *Intel Advisor* (Subsection 2.4.4), developers can have a global picture of the program's runtime behavior. For each loop in the program, all detail information around vectorization will be fully presented. Corresponding optimization suggestions will be given.

5. **Implement recommendations**: developers must verify that the suggestions provided at the previous step will not change the semantics of the code before implementing them in the program.

6. **Repeat**: iterate through the process until the desired performance is achieved.

After the general methodology, there are eventually four approaches to achieve vectorization: from low-level intrinsics to high-level libraries, directives, and auto-vectorization; each has its advantages and constraints which will be explained in the following subsections:

2.3.1.1 Intrinsics

Intrinsic functions are built-in in the compiler which allows to perform operations directly when possible, rather than linking to libraries for implementation. Due to the intimate connection between intrinsic and compiler, these functions are usually used to implement optimizations which are not addressed by high-level programming languages. Take the Intel compiler, for example, intrinsics map directly to x86 SIMD instructions (MMX, SSE, AVX/AVX2, IMCI, and AVX-512). Intrinsic functions are quite similar to assembly language: they are both low-level programming languages that have a strong correspondence to the target hardware. Generally, they are not portable across different architectures, and do not need interpreting or compiling.

```
__mmask8 _mm512_cmp_pd_mask (__m512d a, __m512d b, const int imm8)
```

Here is an example of the "compare" instruction supported by IMCI and AVX-512: *mm512* signifies that this is a 512-bit instruction; *cmp* indicates comparison; *pd* means that it is used for double-precision variables. *a* and *b* are two inputs with each consisting of eight double-precision variables. The third input *imm8* is responsible for specifying the comparison operand. According to the *Intel Intrinsics Guide* [60], thirty-two types of comparisons can be performed with this instruction, which includes "equal", "less than", "greater than" and so on. The final output is a 8-bit *__mmask8* (or *char*) variable where the comparison result for each pair of double-precision variables is represented by 1 (*true*) or 0 (*false*). The *Intel Intrinsics Guide* is a practical online manual describing all Intel intrinsics. Developers can search for specific instructions with their semantics (*load*, *mask*, etc.) or technologies (SSE, AVX, etc.). For each instruction, detail information like corresponding header-file and instruction description will be introduced as well.

The advantage of using intrinsics is that developers can fully profit from the hardware's computing capability by explicitly calling the latest (the most powerful) instructions supported by the target architecture. Using intrinsics can help to get the best execution performance among four vectorization approaches. However, as shown in the above example, programming with such low-level intrinsics is not productive at all because it requires developers to take care of hardware details, which indicates a large amount of

extra development work. Moreover, the program is not portable meaning that the implementation work should be repeated, and the program should be adjusted for every individual architecture. Such redundancy makes intrinsics impossible for the optimization of a whole project like TRIPOLI, where tens of thousands of lines of codes are involved. Therefore, intrinsics are usually applied in a small part (the most computationally expensive kernel) of large codes to balance the requirement of higher performance and the productivity problem.

2.3.1.2 Directives

The SIMD directives become standard since OpenMP 4.0. Extra directives added into the original program will not change the program's runtime behavior if no hardware or software supports are available. They enforce vectorization of loops with simple pragmas, which places all the burden on the developer to ensure code correctness.

Two kinds of directives are employed for the current work: the Intel directives and the OpenMP directives. These two directives are generally identical but the Intel one is only functional with the Intel compiler while the OpenMP standard is universal to all available compilers. The following example gives a simple demonstration for the use of these two directives:

```
// Intel directives
#pragma vector aligned
for (i = 0; i < N; ++i)
    a[i] = a[i] * b;

// OpenMP directives
#pragma omp simd aligned(a:32)
for (i = 0; i < N; ++i)
    a[i] = a[i] * b;
```

In Intel directives, *vector* indicates that the following loop is forced to vectorized; *aligned* mandates the compiler that any array variables involved in the loop have been already aligned to the right boundary to avoid any potential check. The expression with the OpenMP directives is slightly different: *omp simd* plays the same role as *vector*; data alignment is more manual since the aligned array a and the aligned boundary (32-bit in this case) should be specified explicitly. Clauses are used to modify or extend the meaning of the original directive, the additional *aligned* clause after the *omp simd* directive provides more vectorization control over the target loop. Take *omp simd* for example, the directive can be decorated with seven clauses: *aligned*, *collapse*, *lastprivate*, *linear*, *private*, *reduction*, and *safelen*.

Compared to the intrinsics, using directives to aid vectorization requires much less implementation work. Instead of rewriting the program with excessive intrinsic instructions, adding several directives to the target loop is simple and direct. The modified code is still readable, which also makes code maintenance much easier. Besides, optimizations are no longer restricted to individual architectures but could be portable from one system to another. The only issue of using directives is that the developer should carefully handle the code in order not to change its semantics. This problem sounds easy to deal with but occurs too often during implementation.

2.3.1.3 Libraries

Another practical way to achieve vectorization is to use libraries. Thanks to this approach, developers are no longer necessary to have strong hardware and optimization background. The work is much simplified with encapsulated interfaces.

There are a lot of SIMD libraries to help vectorization. For example, Intel MKL is one of the most commonly used libraries on multi-core and many-core systems which concludes popular functions like Basic Linear Algebra Subprograms (BLAS), Fastest Fourier Transform (FFT), etc. The functions are highly optimized and require no code change for utilization. It is compatible with different languages, operating systems, linking and threading models. Cilk/Clik+ is initially a threading library to extend C and C++ by the fork-join idiom. Since Cilk+ [61], vector extensions like new array notation ($A[0:n]$) and SIMD directives are introduced to help code vectorization. Intel SIMD Data Layout Templates (SDLT) [58] is a template library to provide SIMD-specific containers for C++. Instead of the naive standard containers provided by STL, these new containers encourage SIMD code generation and solve the productivity problem of low-level techniques. Concerns around data movement (alignment, prefetching, etc.) are automatically handled by the library so there is a higher chance for the program to achieve efficient vectorization. Other libraries like Vc [62], Boost.SIMD [63], and Eigen [64] not all enumerated here could also be a solution for high-level vectorization.

2.3.1.4 Auto-Vectorization

Auto-vectorization counts on the compiler to figure out all the work. Developers concentrate on code programming and let the compiler do its best for vectorization. Basic tips to help compiler auto-vectorization have been already listed at the beginning of this section. This will aid compiler to better understand the program's semantics thus may generate more efficient SIMD codes, but such method is not reliable at all because vectorization will always limit by language and compiler technology. Compiling work

do not consider hardware features of different architectures so it generates SIMD codes with no difference between an SSE Pentium and an AVX-512 Knights Landing. As a consequence, the vectorization is not portable since it may use the latest ISA by random choice on one system, and do the opposite on others. It exists compilers like *ispc* (Intel SPMD Program Compiler) [65] that can improve auto-vectorization, but they require supplementary work to babysit the compiler with specific options thus make them no different from common libraries. In a conclusion, there is no free lunch – counting on the compiler is not reliable for high-efficiency vectorization.

2.4 Useful Tools for Performance Analysis

Performance analysis tools are important for optimization. They help developers to concentrate on the right kernel, and can also provide useful optimization suggestions. In this section, four profiling tools involved in this thesis will be introduced.

2.4.1 TAU

TAU [66, 67] is a profiling and tracing toolkit for performance analysis of parallel programs. The project was originally proposed by the University of Oregon, and now becomes a cooperative work between the University of Oregon, the Los Alamos National Laboratory, and the Jülich Research Centre.

TAU is capable to deal with programming languages like Python, Java, Fortran, C/C++ or even mixed programming. This cross-platform toolkit is free and can run on different architectures like CPU, GPGPU, and MIC. It includes parallel profiling and tracing not only for shared-memory systems but also for distributed ones. Besides, code profiling can measure time as well as hardware performance counters like cache miss, memory bandwidth and FLOP usage with the help of PAPI [68]. Users can start the analysis by the auto-instrumentation: no code changes are required in this mode; users only need to recompile the program with the package. Basic information like performance bottlenecks, hotspot loops will be shown directly with the help of ParaProf visualizer. More detailed analysis can be realized by manual instrumentation. Probe instructions need to be used to indicate explicitly the target part of the code and the type of analysis to perform. The manual mode provides more precise evaluation with the price of extra implementation work.

2.4.2 Intel VTune

VTune [69] is a performance analysis toolkit dedicated to serial and multithreaded applications. This profiler has two user-modes: GUI (graphic user interface) and command line. Though in the latest release (version 2018) it begins to support MPI profiling and programming languages other than Fortran and C/C++, using these new features to perform code profiling is not included in the current thesis's work.

Compared to TAU, VTune has better supports for analyzing programming languages and a much more user-friendly GUI. One common use of VTune is to find program's hotspots – the most time-consuming program units. By running the *Basic Hotspots* analysis, users will have a first profiling summary for the target program (Figure 2.11): the ratio between serial execution time and parallel region time, the potential gain with optimal OpenMP codes, top five hotspots, etc. Without stepping into details, users can already have a global picture of major performance issues of the code.

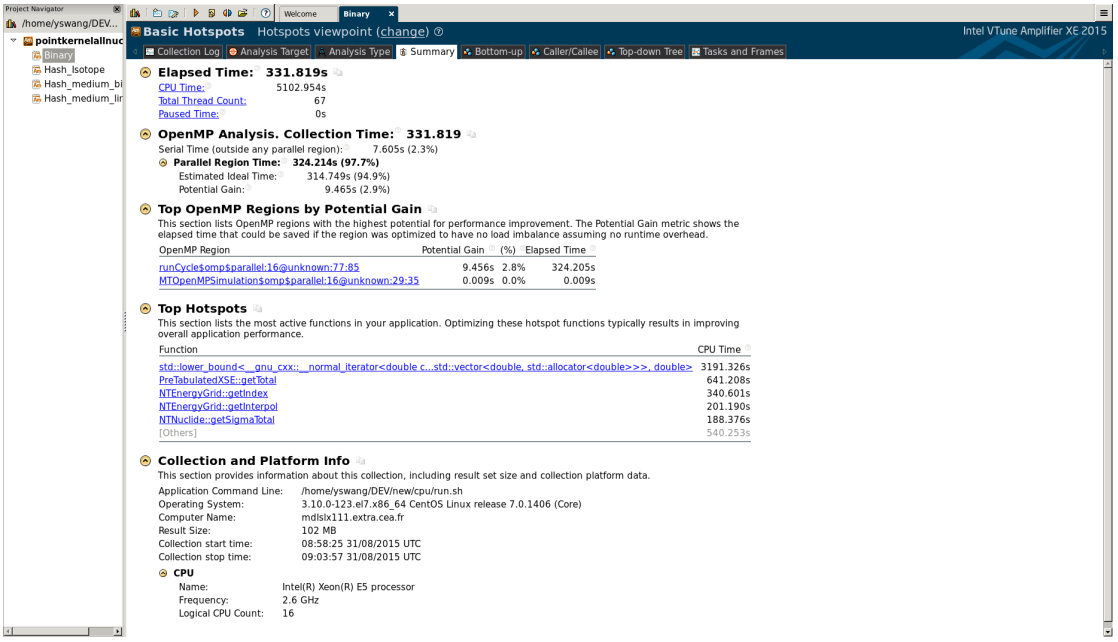


FIGURE 2.11: *Summary* report of a *Basic Hotspots* analysis.

In the *Bottom-up* view, one can explore finer information about the hotspots (Figure 2.12). Double-clicking on each hotspot allows users to identify the hotspot directly in the source code as well as the assembly (Figure 2.13), which is really useful for following optimization work.

Except for the hotspot detection (with *Basic Hotspots* and *Advanced Hotspots* modes), VTune also provides other performance analysis like:

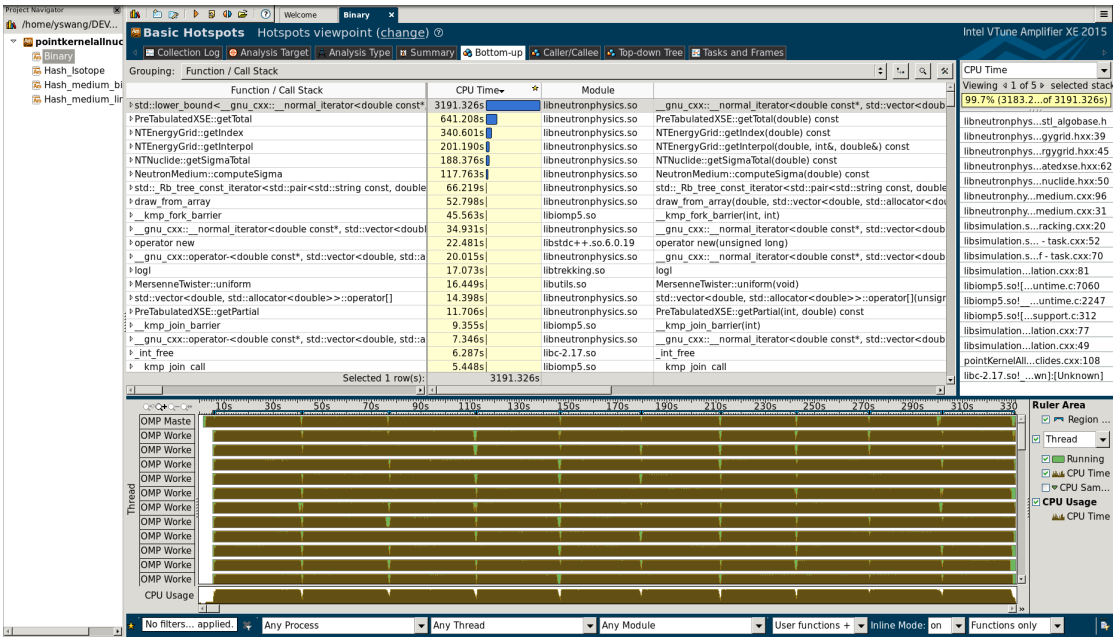


FIGURE 2.12: *Bottom-up* report show hotspots from the largest to the smallest.

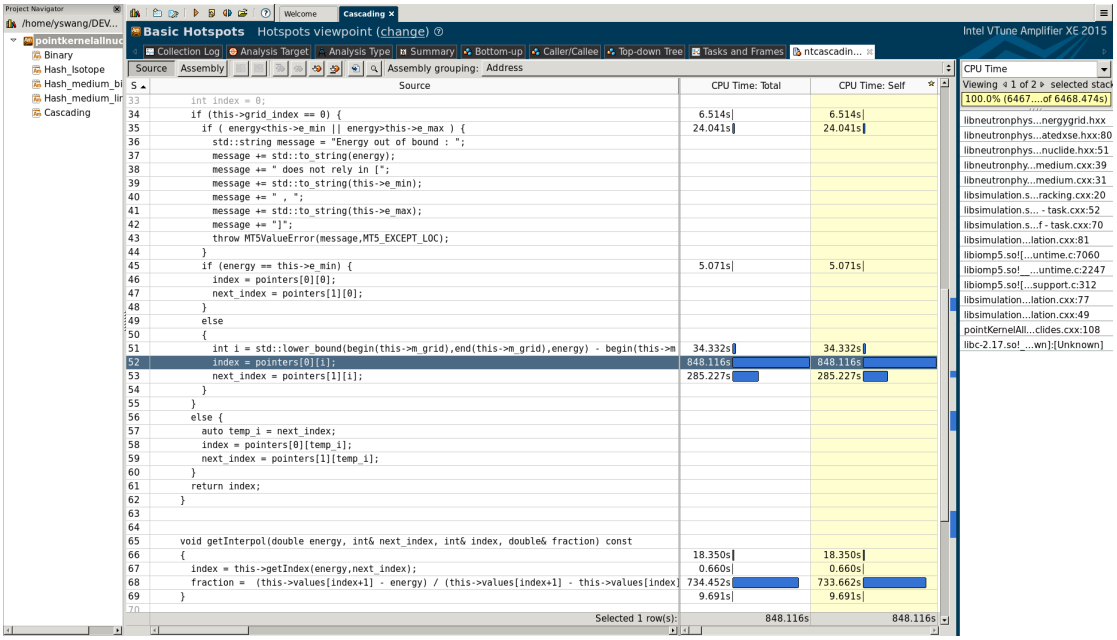


FIGURE 2.13: Identify instructions in the source code corresponding to the hotspot.

- **Locks and Waits**: Identify synchronization objects that cause contention and prevent parallelization.
- **General Exploration**: Identify hardware-issues like data sharing, cache misses, branch misprediction, etc.
- **Memory Access**: Analyze memory latency, memory bandwidth, false sharing, etc.

With this information, developers can identify the candidate kernels for optimizations and understand why the hotspots come out.

2.4.3 Intel Optimization Report

Since the version 15.0 of Intel compiler, the optimization report [70] has been introduced to provide developers with a readable compiling report for further application tuning. Firstly, the report informs developers which loops in the code are vectorized. Then, it reports also why the compiler did not vectorize a loop. The aim of this report is not only to help developers understand what the compiler does, but also to help them understand the obstacles that encountered so that corresponding actions can be taken to improve the situation.

By adding *-qopt-report[=N]* into compile options, developers can choose N from 0 to 5 (2 by default) levels of report details (0: no diagnostic information, 5: the greatest level of detail reporting non-vectorized loops and dependency information). After compilation, the report will be stored in a file ended by *.optrpt*. The following example shows the layout of a $N = 5$ level report on the major loop of the Faddeeva unit-test (see Subsection 5.4.1):

```
LOOP BEGIN at special.cc(407,3)
<Peeled loop for vectorization>
   remark #15331: loop was not optimized
LOOP END

LOOP BEGIN at special.cc(407,3)
   remark #15389: vectorization support: reference vz_r has
   unaligned access   [ faddeeva.cc(408,5) ]
   remark #15389: vectorization support: reference vz_i has
   unaligned access   [ faddeeva.cc(408,5) ]
   remark #15381: vectorization support: unaligned access
   used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set
   to 2
   remark #15309: vectorization support: normalized
   vectorization overhead 0.082
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15450: unmasked unaligned unit stride loads: 2
```

```
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 117
   remark #15477: vector loop cost: 16.000
   remark #15478: estimated potential speedup: 5.810
   remark #15484: vector function calls: 1
   remark #15487: type converts: 2
   remark #15488: --- end vector loop cost summary ---
   remark #15489: --- begin vector function matching report
   ---
   remark #15490: Function call: wofz(float, float, float &,
    float &, int *) with simdlen=8, actual parameter types:
   (vector,vector,linear:4,linear:4,uniform)   [ faddeeva.cc
   (408,5) ]
   remark #15492: A suitable vector variant was found (out
   of 2) with ymm2, simdlen=8, unmasked, formal parameter
   types: (vector,vector,vector,vector,vector)
   remark #15493: --- end vector function matching report
   ---
LOOP END

LOOP BEGIN at special.cc(407,3)
<Remainder loop for vectorization>
   remark #15389: vectorization support: reference vz_r has
   unaligned access   [ faddeeva.cc(408,5) ]
   remark #15389: vectorization support: reference vz_i has
   unaligned access   [ faddeeva.cc(408,5) ]
   remark #15381: vectorization support: unaligned access
   used inside loop body
   remark #15335: remainder loop was not vectorized:
   vectorization possible but seems inefficient. Use vector
   always directive or -vec-threshold0 to override
   remark #15305: vectorization support: vector length 4
   remark #15309: vectorization support: normalized
   vectorization overhead 0.038
LOOP END
```

The compiler generates a “LOOP BEGIN” and a corresponding “LOOP END” message to signify the scope of the loop. Filename, line number, and column number listed just after are practical to fast locate the target loop. As shown in the report, the compiler

generated supplementary peeled loop and remainder loop outside the main loop body, which indicates that data are not aligned in the main loop, and the loop trip count is not multiple of the *vector length*, respectively. The data alignment problem is then confirmed by the message "remark #15389: vectorization support: reference ... has unaligned access" and "remark #15381: vectorization support: unaligned access". For a single-precision floating point variable (32-bit), an AVX2 register can simultaneously hold eight such variables. This information is represented by the message "remark #15305: vectorization support : vector length 8" since $256\textit{(bit)}/32\textit{(bit)} = 8$. It reflects another fact that the compiler vectorized the loop with 256-bit instructions. It is a good sign in this case because the code was compiled to run on an AVX2 architecture where the 256-bit instructions are the latest ISAs supported by the system. According to the message "remark #15399: vectorization support: unroll factor set to 2", the compiler chose to unroll the single loop content to two. "OpenMP SIMD LOOP WAS VECTORIZED" indicates that the main loop body was vectorized with the help of OpenMP directives. According to the "vector loop cost summary", the scalar execution of the target loop should be 117 seconds, and after vectorization, the figure is around 16. Besides, even if an external function call presents inside the loop body ("remark #15484: vector function calls : 1"), the compiler succeeded to vectorize it with the help of *declare* directives (not shown here, see Subsection 5.3.3.4). The 5.81× speedup is a theoretical figure estimated by the compiler, the real speedup should be measured with runtime results (for example, using Intel Advisor). Unlike the main loop body, the peeled loop and the remainder loop were not vectorized because the compiler thought it is not efficient to do so. Moreover, the *vector length* in the remainder loop was recognized as four instead of eight, which indicates that the hardware feature was not fully used by the compiler.

For such a small example, the report can already provide a rich source of information to guide the following work. This compile-time tool allows developers to begin initial optimizations without time-consuming run time profilings thus is a good start-point for the entire optimization process.

2.4.4 Intel Advisor

Profiling tools like TAU and VTune can identify performance bottlenecks of the program, but none of them indicates explicitly how to deal with these problems. Intel Advisor is an optimization assistance toolkit that provides developers with optimization tips. Figure 2.14 shows an example of analysis results. In the `FLOPS` tab, `GFLOPS` indicates the FLOP usage of the loop; `AI` represents the loop's arithmetic intensity (ratio between arithmetic operations and memory accesses). For non-vectorized loops, the toolkit will

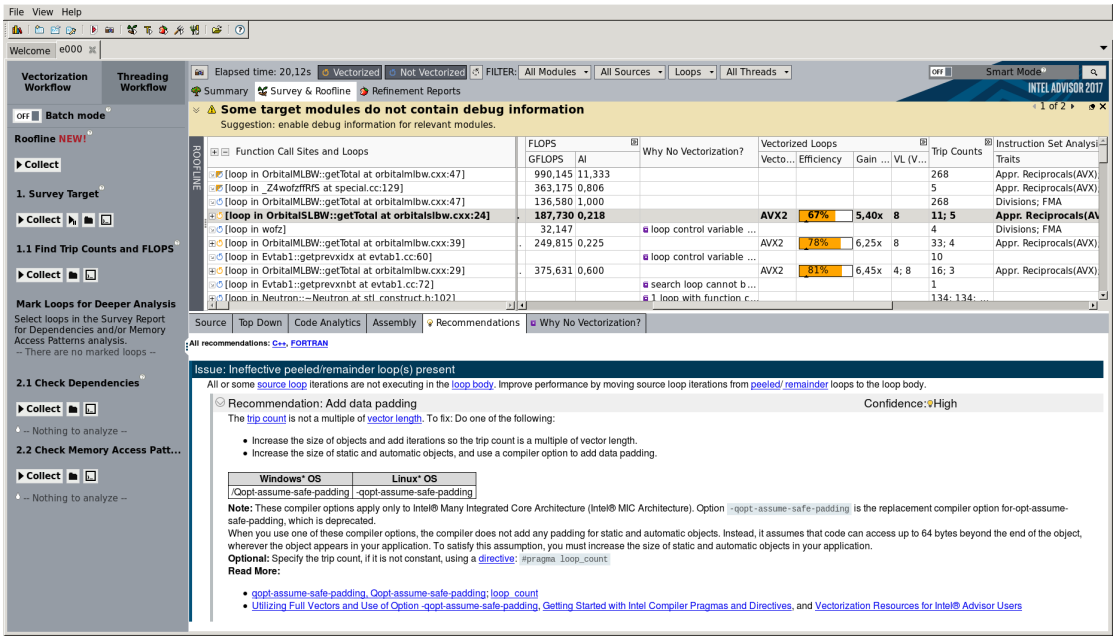


FIGURE 2.14: Advisor analysis with trip counts and FLOPS for the target program on a *Broadwell* architecture.

explain why vectorization did not happen. For vectorized loops, Advisor shows which ISA was used to carry out the vectorization (as the `AVX2` shown in the figure); vectorization efficiency and gain are highlighted to tell if vector units were fully used (for `VL=8`, the ideal gain brought by vectorization should be 8× instead of 5.40× or 6.25× shown in the result). `Instruction Set Analysis` shows time-consuming instructions present in the loop body, for example, divisions, FMA, reciprocal approximations, etc. Like VTune, Advisor also supports hotspot identifications in the source code (`Source` and `Assembly` tabs in Figure 2.14). In the `Recommendations` pane, optimization tips are listed and explained with external links as well as code examples. These tips are sorted by their confidence, which allows developers to distinguish optimization priority and focus on important suggestions. As shown in the figure, the remainder loop largely reduces vectorization efficiency due to the hint: `Confidence:High`, so data padding to make the trip count a multiple of vector length should bring significant speedup. It should be noted that these tips are extremely important to get optimal performance with Intel architectures since several techniques mentioned can be found nowhere in open-access manuals.

By specifying the target loop or function, users can obtain further optimization hints with more detailed `Check Dependencies` and `Check Memory Access Pattern` analysis. It should be noted that finer the analysis is, the more time it will take. For a `Survey` analysis which takes about several seconds, a duration around several minutes is expected for just analyzing one most important kernel with the corresponding `Check Dependencies`. Intel Advisor also provides the Roofline analysis to target applications

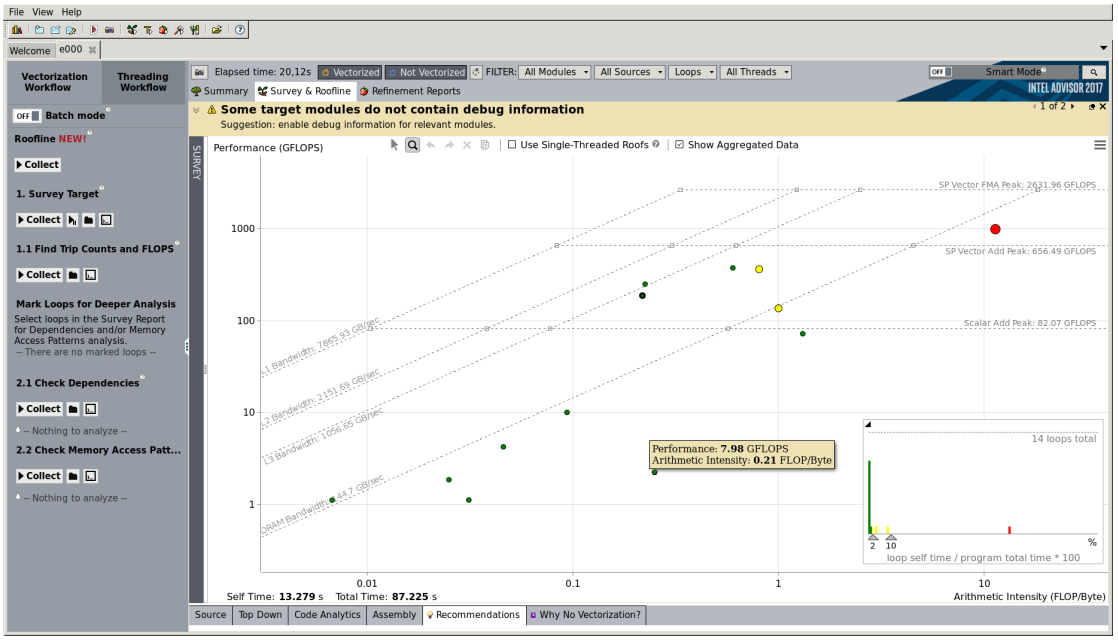


FIGURE 2.15: Direct Roofline analysis with Advisor.

(Figure 2.15). This model was firstly proposed by Williams [26] in 2009, and then was improved as a "cache-aware" representation by Ilic [71] in 2013. It can be used to visualize the relationship between application performances and effective hardware limitations and therefore, help developers figure out performance bottlenecks and their corresponding solutions.

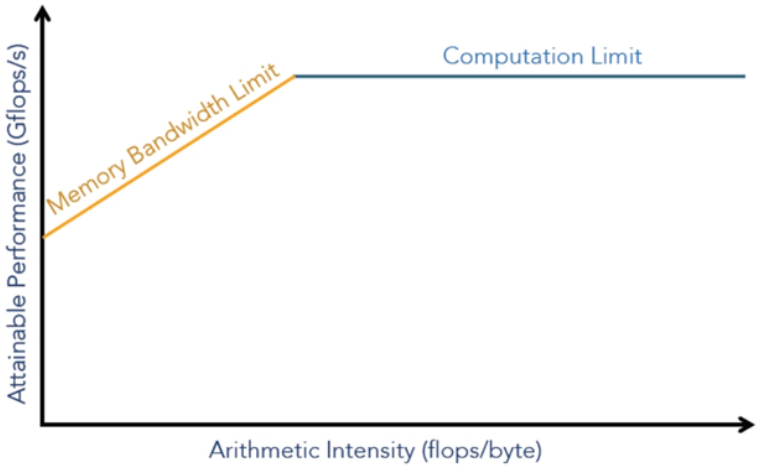


FIGURE 2.16: An example of the naive Roofline model [72].

The basic Roofline chart (Figure 2.16) consists of two hardware-specific limits: the processor's computing performance and the memory bandwidth. The X-axis is the arithmetic intensity (ratio of floating-point operations to memory accesses) and the Y-axis represents the FLOP usage. Both axes are in logarithmic scale. Generally, for a

given computing kernel, it is memory-bound if its arithmetic intensity is underneath the bandwidth roof, and it is compute-bound if underneath the computation roof. The Roofline model integrated into Advisor is improved with multiple ceilings for each limit. Effects of the cache/memory hierarchy, the vector unit, and FMA are also taken account into the performance modeling, which allows for more accurate bottleneck analysis.

Chapter 3

Monte Carlo Neutron Simulations

3.1 Nuclear Reactors

Generally speaking, the nuclear energy can be harnessed in two main ways: fusion and fission. Fusion is a kind of nuclear reaction in which two or more light nuclei collide at a very high energy and merge together as a new heavy nucleus. The total mass of the heavy nucleus is smaller than the total mass of the fusing nuclei and the mass difference is released as energy. Though the fusion power offers an inexhaustible source of energy and fusion reactions have energy density much higher than nuclear fission, how to trigger and sustain such reaction in a practical manner is still unknown. Nuclear fission can be seen as the opposite of fusion reaction: contrary to fuse together, a heavy nucleus absorbs a neutron and decays by splitting into smaller elements. This process produces energy and free neutrons, which are candidates to initiate new fissions. Such subsequent processes result in a self-propagating nuclear chain reaction, which often releases several million times more energy per reaction than any chemical one. At present, nuclear power plants are all based on this fission chain reaction. Like conventional thermal power stations generate electricity by harnessing the thermal energy released from burning fossil fuels, nuclear reactors convert the energy released from fissions to heat which then drives steam turbines to produce electricity.

One major difference between the conventional fossil power and the nuclear power is the way they deal with fuel. Instead of feeding the thermal power plant by continuously burning fossils, nuclear fuel in forms of assemblies or bundles is carefully kept inside the reactor core during its whole operating-cycle. There are many nuclei that can undergo fission, but only a few of them (for example, $^{233}\mathtt{U}$, $^{235}\mathtt{U}$ and $^{239}\mathtt{Pu}$) can sustain a fission chain reaction and therefore be used to produce nuclear fuel.

In a nuclear reactor, the neutron population is a function of the rate of neutron production and the rate of neutron loss. If the production rate is exactly the same as the loss rate, the chain reaction is self-sustaining and can be referred to as *critical* or with $k_{eff} = 1$, where k_{eff} stands for the *effective multiplication factor* of a finite multiplying system. A *critical* reactor indicates that the reactor can be operated in a stationary fashion and the neutron flux is stable at a given power level, or in other words, it is able to maintain a constant neutron population and reaction rate without any external neutron sources. The effective multiplication factor can also describe the time rate of change of the neutron population if the average neutron lifetime is known. It is an important quantity in characterizing the reactor behavior since reactor power is proportional to the reaction rate, and the multiplication factor, therefore, dictates the time rate of change of the core power [73].

3.2 Nuclear Reactions

Neutrons that travel in matter collide with the different nuclides present. The probability of a collision with a certain nuclide is given by its *microscopic cross-section*. Upon collision, a compound nucleus is generally formed, in an excited state. This compound nucleus then decays back to its ground state by emitting one or more particles and eventually some gamma rays. From the point of view of predicting the behavior of the neutron population, what is important is the number of neutrons emitted and their kinetics parameters after emission: energy and direction. Accordingly, the reactions can be classified as follows:

- `Elastic scattering`: one neutron is emitted and the collided nucleus goes back to its ground state. The kinetics parameters of the exiting nucleus can be deduced from two-body kinematics (preservation of total momentum and total kinetic energy).
- `Inelastic scattering`: one neutron is emitted, but the nucleus is left in an excited state. If the excitation energy is known, then the kinetic parameters of the exiting neutron can still be deduced from two-body kinematics. Gamma rays are also usually emitted.
- `Absorption`: no neutron is emitted, but only gamma rays and eventually other particles like protons, deutons, etc. which are not followed further.
- `Fission`: several (between 0 and 7) neutrons can be emitted, with an energy distribution specific to each nuclide. Two *fission fragments* are also emitted, which carry most of the fission energy (200 MeV per fission), as well as gamma rays.

Each reaction type is characterized by its own cross-section.

3.2.1 Cross-Section

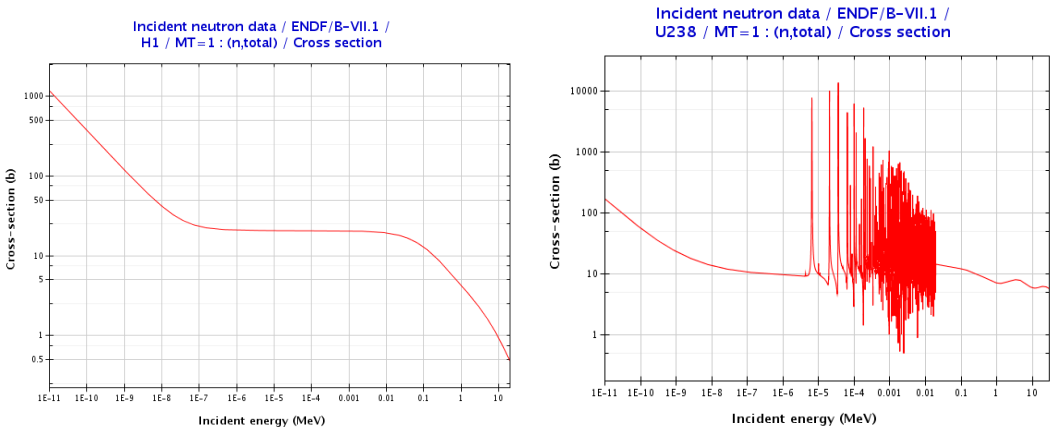


FIGURE 3.1: Energy-dependent cross-section tables $^{1}\mathtt{H}$ and $^{238}\mathtt{U}$ [74].

Figure 3.1 shows the total cross-sections for $^{1}\mathtt{H}$ and $^{238}\mathtt{U}$. We see that the cross-section value depends on the (kinetic) energy of the neutron undergoing collision. This dependency can vary wildly from light elements, for which the plot is rather smooth, to heavy elements which present many (up to several thousand) *resonances*. A resonance, which is related to the energy levels of the compound nucleus, is a region where the cross-section can change of several orders of magnitude (from one to 10^5 barns) in a very little energy range (about a fraction of eV). These resonances have a very pronounced effect on the neutron behavior and need to be carefully taken into account in the neutron transport simulations.

As a general behavior, at low energy ($< 1eV$), cross-sections are smooth and proportional to the time the neutron spends within the reach of the nuclear force ($1/\sqrt{E}$). At intermediate energy, resonances occur randomly spaced and with unpredictable characteristics (height and width). Above 1 MeV the cross-sections go back to a smooth behavior as a function of energy.

Cross-sections and resonance parameters are available from *Evaluated Nuclear Data Files* or Nuclear Data Libraries, of which ENDF/B-VII.1 [75] from the USA and JEFF-3.1 [76] from Europe are examples. Starting from that data *pointwise* cross-sections can be reconstructed according to a small number of resonance models like Single Level and Multilevel Breigt-Wigner, Reich-Moore, and R-Matrix (see Chapter 6).

Reconstructed cross-sections are stored as long tables of σ as a function of energy, meant for linear reconstruction. That is, whenever we need $\sigma(E)$ we can compute it as:

$$\sigma(E) = \sigma(E_k) + \frac{E - E_k}{E_{k+1} - E_k}[\sigma(E_{k+1}) - \sigma(E_k)] \tag{3.1}$$

where E_k and E_{k+1} are the nearest lower and upper energy points in the energy grid which bound the arbitrary energy E. Energy lookups are thus required to find these two points and then retrieve corresponding data from memory. The number of points necessary for the linear reconstruction with given accuracy (usually 0.1 percent), varies with the isotope behavior, spanning from 600 points for Hydrogen to 150000 points for ^{238}U. Figure 3.2 gives an idea of the variability of the number of energy points needed. It should be noted that energy points are not evenly distributed within the entire energy range covered by the cross-section tables. *Resonance* regions need much more data to express the abrupt change of cross-section values. A consequence is that each isotope has its own energy grid, optimized for its cross-section behavior.

When a neutron travels through a material made up of different nuclides, its *mean free path* is the inverse of the *macroscopic cross-section*, defined from the microscopic cross-sections of its constituents as:

$$\Sigma(E) = \sum_i N_i \sigma_i(E) \tag{3.2}$$

where i is the nuclide index, N_i its concentration in the material, and $\sigma_i(E)$ the corresponding microscopic cross-section at energy E.

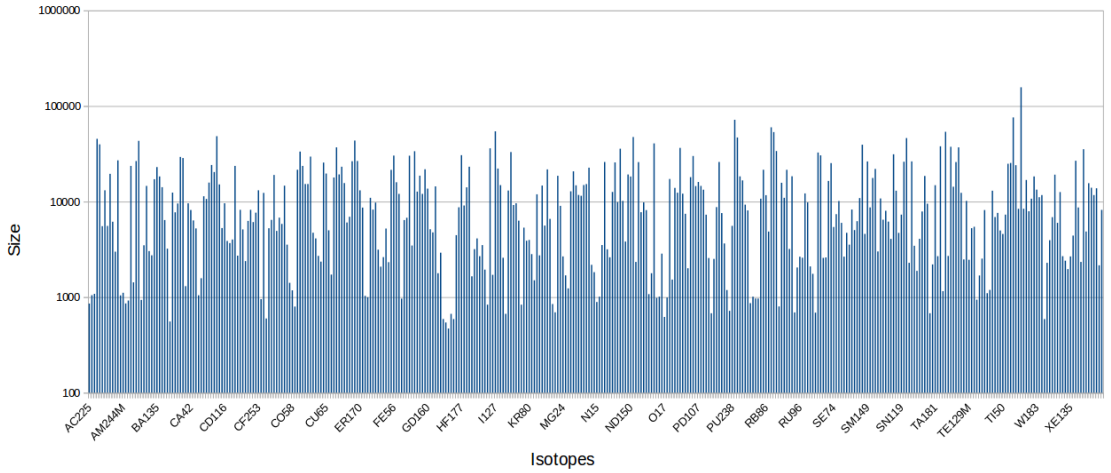


FIGURE 3.2: Isotopic energy table lengths for 390 isotopes at $T = 300$K. The minimum is for ^{3}H, which has only 469 energy points, and the largest is for ^{238}U with 156,976 points. The average length over all isotopes is around 12,000.

3.3 Effects of Temperature on Cross-sections

The cross-sections given in the Nuclear Data Libraries are valid for a neutron colliding with an isotope *at rest*; they are thus called cross-sections at 0 Kelvin. The thermal motion of atoms can influence the interaction between target nuclei and free neutrons. If the neutron is fast enough, with a kinetic energy above 1 MeV, we can neglect the thermal motion of the nuclides and use the 0K cross-sections.

In order to accurate cross-sections on considering temperature effects, one must be aware of the distribution of target velocities. The thermal velocity of the target system follows the *Maxwell-Boltzmann distribution* [77], and can be expressed as a function of absolute temperature (T) and target mass (M):

$$P(v_T)dv_T = \frac{4}{\pi}\beta^{\frac{3}{2}}v_T^2 exp^{-\beta v_T^2}dv_T \tag{3.3}$$

where $\beta = M/2k_BT$, k_B is the *Boltzmann constant*. High temperatures change the shape of cross-section resonances by broadening the resonance width and pulling down the resonance peak. This widening of resonances is referred to as *Doppler broadening* [78]. Such effect caused by the wider target motion distribution results in a significant increase of absorption probability in the resonance region as neutrons scatter down to thermal energies. It has an advanced meaning for reactor safety since it establishes a negative feedback mechanism for temperature increase, and therefore prevents a meltdown of the system.

In order to conserve the real reaction rate, the effective probability of collision should be obtained by considering the material temperature. This process is described by:

$$\sigma_{eff}(v) = \frac{1}{v}\int_{all:v'>0} v'\sigma(v')P(v_T)dv_T \tag{3.4}$$

where v is the velocity of the neutron, $v' = |v - v_T|$ is the relative velocity between the neutron and the target nucleus, and $P(v_T)$ is the target motion distribution. Expanding the above formula with Equation 3.3, the complex integral can be transformed into a much simpler form:

$$\sigma_{eff}(v) = \sqrt{\frac{\beta}{\pi}}\int_0^\infty (\frac{v'}{v})^2 \sigma(v')[exp^{-\beta(v'-v)^2} - exp^{-\beta(v'+v)^2}]dv' \tag{3.5}$$

A lot of studies have been done on how to efficiently implement this equation, among which the *SIGMA1* method [79] is an exact one without numerical simplification thus known as the reference. It is commonly used by preprocessing tools like NJOY [80].

3.4 Neutron Transport

The objective of neutron transport is to compute the distribution of the neutron population inside a nuclear reactor. From this knowledge, many quantities of interest can be computed, such as the power production distribution inside the reactor, the hot spots, vessel fluence, the energy deposited to structural material, the dose that operator will encounter if they where to approach the reactor. It helps nuclear scientists and engineers to understand the behavior of a nuclear reactor in both nominal and accidental conditions, inside and outside a reactor core, and thus operate it in safety conditions.

Since neutrons are neutral particles, they always travel in straight lines and only deviate from their path when they collide with a nucleus to be scattered into a new direction or absorbed. One common assumption in neutron transport is that neutrons do not interact with each other. This follows from the fact that the neutron concentration in a nuclear reactor is much less than that of materials, so neutrons are much more likely to interact with materials than between themselves.

3.4.1 Neutron Transport Equation

$$\begin{aligned}
&\frac{1}{v(E)}\frac{\partial}{\partial t}\psi(\overrightarrow{r},\hat{\Omega},E,t)+\\
&\overbrace{\hat{\Omega}\cdot\nabla\psi(\overrightarrow{r},\hat{\Omega},E,t)}^{Steaming}+\\
&\overbrace{\Sigma_t(\overrightarrow{r},E)\psi(\overrightarrow{r},\hat{\Omega},E,t)}^{Collisions}\\
&\qquad=\\
&\qquad\overbrace{\int_0^\infty dE'\int_{\hat{\Omega}}d\Omega'\Sigma_s(E'\to E,\hat{\Omega}'\to\hat{\Omega})\psi(\overrightarrow{r},\hat{\Omega}',E',t)}^{Scattering}\\
&\qquad+\overbrace{\frac{\chi(E)}{4\pi}\int_0^\infty dE'\int_{\hat{\Omega}}d\Omega'\nu(E')\Sigma_f(\overrightarrow{r},E')\psi(\overrightarrow{r},\hat{\Omega}',E',t)}^{Fissions}\\
&\qquad+S_{external}
\end{aligned}\tag{3.6}$$

The *neutron transport equation*, derived from the *Boltzmann transport equation*, is a linear equation which represents a balance statement where the number of neutrons gained equals to the number of neutrons lost. Equation 3.6 represents such a balance for the angular neutron density ψ in the element of the phase-space $(\overrightarrow{r},E,\hat{\Omega})$ at time t. $\overrightarrow{r}$ is the spatial position of the neutron, E its kinetic energy, and $\hat{\Omega}$ its direction of propagation.

The quantities introduced are:

- $\Sigma_t(\overrightarrow{r}, E)$: is the total cross-section describing the inverse of the mean free path of a neutron in the material in position $\overrightarrow{r}$.
- $\Sigma_s(E' \to E, \hat{\Omega}' \to \hat{\Omega})$: the differential scattering cross-section characterizing the probability that a scattered neutron of energy E in a direction $\hat{\Omega}$ exits the collision with energy E' and direction $\hat{\Omega}'$.
- $\Sigma_f(\overrightarrow{r}, E')$: the fission probability for a neutron of energy E in the material at position $\overrightarrow{r}$.
- $\chi(E)$: the fission spectrum, which describes the energy distribution of neutrons produced by fission.
- ν: the average number of neutrons produced per fission.
- $S_{external}$: the external source that accounts for any neutron sources not induced by the neutron flux. itself

The *deterministic methods* for neutron transport simulations start from this integro-differential equations and try to solve it by discretizing its variables $(\overrightarrow{r}, E, \hat{\Omega})$, and t and applying various approximations and numerical methods. These methods are widely used in nuclear applications, but they suffer from approximations and numerical errors which are hard to quantify, and thus require massive efforts for justifying their use in a particular situation.

3.4.2 Monte Carlo Simulation

There are two prevalent approaches to deal with the *neutron transport equation*: the deterministic and the stochastic. The deterministic method directly solves the equation in a numerically approximated manner by discretizing numerous variables like spatial, angular, energy and time. Although such method requires less memory footprint during simulation, the approximations along with discretization result in numerical errors. Moreover, in complex problems where spatial and energy variables are impractical or impossible to discretize, the alternative stochastic method turns out to be the only solution.

The Monte Carlo simulation, also called the stochastic method, has a completely different approach and does not directly deal with the transport equation. The idea is straightforward and brute-force: rather than solving a complex equation for the neutron flux distribution, the MC method attempts to reproduce exactly (simulate) what

happens in reality. The essence of the Monte Carlo method is to use a given data-generating mechanism for the process model one wishes to understand, produce new samples of simulated data, and examine the results of those samples.

In neutron transport, a neutron is followed from its birth to its demise, either by absorption or by leakage out of the system. The simulation is repeated a large number of times. Macroscopic quantities like fluxes and reaction rates are then computed from individual traces.

In order to follow a neutron, we need to compute:

- distance to next collision
- isotope the neutron is having the collision with
- which type of interaction the neutron is going through
- kinetic parameters of the exiting neutrons (if any)

All of the above are random variables whose distributions are described by cross-sections known from the Nuclear Data Libraries. Those distributions can be sampled via the use of random numbers; pseudo-random number generator is universally used in order to make simulations reproducible, which is of invaluable for numerical verification.

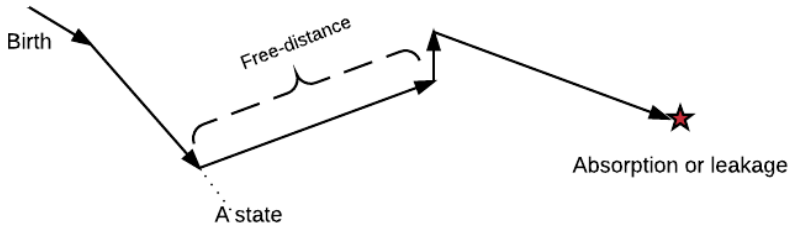


FIGURE 3.3: The random walk process of a neutron from its birth to disappearance.

As mentioned before, the main advantage of the Monte Carlo method is its minimal approximations: it has no geometry approximations as long as only planar, cylindrical, spherical, and toroidal surfaces are present; there are no phase space mesh problems related to discretization; from the view of physical part, cross-section data of ENDF (Evaluated Nuclear Data Files) format [75] preserves full numerical accuracy without simplifications. In addition, as long as neutron *histories* are independent, the embarrassingly parallel nature of MC transport makes it always an ideal candidate for parallel computing.

The crucial downside of the Monte Carlo method is its slow convergence rate governed by the *central limit theorem*. For a calculation involved with N neutron *histories*, this method displays a $1/\sqrt{N}$ convergence. One direct consequence is that Monte Carlo simulations are more computationally expensive than other methods. As a result, any efforts to accelerate Monte Carlo methods are of great interest to the neutron transport community and accelerators like MICs and GPGPUs are widely explored in this field. Apart from issues related to the numerical nature, more and more concentrations are focused on how to efficiently perform the MC calculation on modern architectures. As stated before, the pretabulated cross-section data requires large memory spaces — nearly one GB per temperature, which makes it impractical to perform complex simulations with modern computing materials (where only a dozen GB on-chip memory present), not to mention burn-up computations in which hundreds of temperatures are involved. Besides, there is little vectorization potential for the conventional MC transport calculations meaning the code can not profit the full computing capability of advanced processors, and therefore results in significant resource waste.

3.5 Simulation Codes

There are a certain number Monte Carlo simulation codes developed in the neutron transport community, among which MCNP and TRIPOLI are chosen as the two references for the numerical validation of PATMOS.

3.5.1 MCNP

MCNP, short for Monte Carlo N-Particle transport code, is a general-purpose transport code developed by Los Alamos National Laboratory since the late 1950s. This code tries to include all physical features related to nuclear processes and is valued as the "gold standard" of Monte Carlo neutron transport codes [81]. The latest release – MCNP6 is a combination of MCNP5 and MCNPX, which can be used to simulate radiation protection and dosimetry, radiation shielding, radiography, medical physics, nuclear criticality safety, detector design and analysis, nuclear oil well logging, accelerator target design, reactor design, decontamination, and decommissioning [82].

3.5.2 TRIPOLI

TRIPOLI is a 3-D Monte Carlo particle transport code developed at CEA Saclay. It is capable to perform neutron, photon, electron, and positron transport calculations

for studies in shielding, reactor physics with depletion, criticality safety, and nuclear instrumentation [7]. This code is the reference in France for both industry and research activities.

The history of the TRIPOLI code family can date back to the late 1960s [83–85], while the current version TRIPOLI-4 has been developed since 1990s [86]. The TRIPOLI-4 is rewritten based on previous code versions by using the object-oriented C++ language. The entire code project consists of six components: one geometry component (written in C), one cross-section interpretation component connecting NJOY (written in Fortran), one memory management component, one simulation component and two components dedicated to parallel computing. The code is keeping maintained by the development team and updated with recent features.

3.5.3 PATMOS

PATMOS (PArticle Transport Monte Carlo Object-oriented System) is a new Monte Carlo neutron transport code developed at CEA [87]. The code is prototyped with the fast abstraction language – Python, and then written in the latest C++11/14 [88] for performance efficiency. Polymorphism is largely used in order to evaluate competing algorithms in the model, for example, PATMOS is capable to perform hybrid cross-section computation by combining pretabulated data with on-the-fly Doppler broadening.

The physics of PATMOS is simplified: only neutrons and mono-kinetic pseudo-particles and implemented in the code; only the most probable nuclear reactions are supported by the current implementation; though only simple geometries (box, sphere, cylinder) are implemented in native mode, it has been interfaced to the ROOT [89] geometric package for dealing with complex geometries. However, representative physics to perform a real simulation are fully featured in PATMOS thus it can run benchmark calculations like Hoogenboom-Martin [90]. Numerical validation is realized by comparisons with TRIPOLI-4 and MCNP5 [8]. The final aim is to carry out pin-by-pin full core depletion calculations (a great number of nuclides are present in each material) for large power reactors.

This new Monte Carlo code is dedicated to easily test new physics as well as new computing resources. The main purpose is to study the algorithm behavior with SIMD paradigms or accelerators like MICs and GPGPUs. Mutable and constant variables have been separated explicitly from the beginning of the code development, since in the shared-memory parallel model, one needs to well distinguish mutable and constant variables, and give them thread-private or global access respectively. The PATMOS

project has considered HPC notions all the way from its birth and evolves as a prototype to evaluate new features for the future Monte Carlo reference code in France.

3.6 HPC and Monte Carlo Transports

Algorithm 1 shows the general procedure of a history-based MC neutron transport calculation. The simulation can be loosely represented with four nested loops.

Algorithm 1: Overall procedure of history-based MC neutron transport calculations.

```
1  Distribute all particles in the simulation into N processes;
2  for Process i from 0 to N-1 do
3      Dispatch all particles in the current process into n batches;
4      for Batches j from 0 to n-1 do
5          ...
6          #pragma omp parallel for
7          for Every particle k in batch j do
8              ...
9              for All isotopes in material do
10                 Compute macroscopic cross-section...
11             end
12             ...
13         end
14         ...
15     end
16     ...
17 end
```

The parallelism of distributed systems usually takes place over the outermost loop (line 2 of Algorithm 1). Particles in the simulation system are evenly distributed into all MPI processes. Each process has its own copy of the basic global data thus MPI parallelization is generally not performed for on-node optimization for memory concern. Current PATMOS employs simplified algorithms in which there is no communication across different processes during the simulation. A large number of particles to calculate in each process cancel out the load balance issue of parallelism. It is only in the end that all processes will sum up to accumulate the final result. Thanks to the "embarrassingly parallel" nature of MC transports and all factors mentioned above, parallelism at MPI level can be expressed spontaneously without extra optimization needs. In the future, however, synchronous communication along with more accurate physical model may lead to new scalability problems, not to mention the "billion-way parallelism" brought by the *exascale computing* which will worsen this situation.

The second loop (line 4) is responsible for dividing the large particle pool into smaller batches (with a total number around 10^2). Just like the outermost loop, a balanced workload is still preserved for each iteration. In a history-based algorithm, the third loop (line 7) contains the largest number of loop iterations (between 10^5 and 10^6). Every neutron in a batch is tracked individually by one loop iteration from its birth to the disappearance. Though the *random walk* process of each *history* varies each time (for example, collision type, number of collisions and free-distance, etc.), choosing a reasonable granularity of thread task can help get rid of such unbalance.

In the current PATMOS implementation, we perform OpenMP parallelizations at this level due to its largest trip count among the four loops. Computing workloads are organized as threads and the total number of threads depends on the hardware. For example, on a dual-socket Sandy Bridge with sixteen hardware threads, the loop is parallelized statically with sixteen software threads. Such "thread parallelism" has been and is being proved efficient for MC calculations.

This natural threading approach inherits the typical strategy used for distributed memory systems. It is simple and direct to implement: no matter how many cores the architecture has, we use all of them to maximize the degree of parallelism. As hardware evolves constantly, however, the problem of this concept becomes serious as well. Compared to traditional CPUs with several dozen of cores in maximum, new accelerators like GPGPU can include up to several thousand of cores. As a consequence, the thread granularity varies greatly from one architecture to another. For the same simulation on both architectures, the workload of a GPGPU thread is significantly different from that of CPU which indicates that optimizations specific to certain thread granularity only make sense locally and can not be generalized for general use. For instance, computing cross-sections with the on-the-fly Doppler broadening leads to unbalanced workloads on MIC systems but not on conventional CPUs. Larger thread number results in fewer histories to be calculated in each thread so differences of runtime behavior between histories become non-negligible.

Future implementations could consider the "task parallelism" with which parallel workloads are organized by fixed size tasks. Though the capability of dealing with simultaneous tasks may differ from one architecture to another, the workload distributed to each task is always the same. Compared to the naive parallelism controlled haphazardly by the number of available hardware cores, uniform tasks are more friendly to code portability and sustainability.

Another issue coming along with load balancing is thread scheduling. As stated before, conventional MC transports hide unbalanced workload by launching a large number of neutrons in a single thread, characteristics of individual neutrons have little influence

on the overall behavior of the neutron population. With new algorithms like on-the-fly reconstruction, however, the algorithm complexity raises from $O(log(n))$ (energy lookup with binary search) to $O(n^2)$. The increase in computational cost results in the fact that large neutron populations can no longer cover the load balancing issue, so dynamic scheduling should be introduced for better execution performance.

One major problem of dynamic scheduling is that thread execution order depends on the runtime environment. Data reductions in random orders produce changeable numerical results which make big difficulties for code validation, especially for high-fidelity reactor simulations. In addition to concerns mentioned before, optimal scalability is and will be always visible but not attainable on shared-memory models. Even if the threading strategy is simple, obstacles like complicated cache hierarchies, NUMA systems, shared bus and so on are far more complex than the distributed ones and will always limit speedups. In this situation, code optimizations should go deeper down to the SIMD level in order to achieve better FLOP usage and better energy efficiency instead of seeking for ideal speedups.

The innermost loop (line 9 of Algorithm 1) referring to Equation 3.2 iterates over all nuclides present in the undergoing material to calculate the macroscopic total cross-section. In each iteration, a linear-interpolation (Equation 3.1) is used to get the cross-section corresponding to the given energy and then, isotopic cross-sections should be accumulated to obtain the material cross-section. Since the cross-section computation is typically the most computationally expensive hotspot in the simulation, one previous study [91] proposes a *fine-grained* threading technique to accelerate this process: based on the *coarse-grained* threading being described as "thread parallelism" in the previous paragraph, a part of threads are still preserved to parallelize the third loop (line 7 of Algorithm 1); while the rest is used to perform multi-threading for calculating cross-sections.

For modern architectures equipped with SIMD units, vectorization might be a better solution. According to Section 2.3, it is usually in the innermost loop we perform vectorization to achieve optimal execution performance. However, even where SIMD potentials can be identified and exposed, it is far from guaranteed that vectorizations can be carried out efficiently in a real application.

In the PATMOS code, the overall cross-section calculation is composed of a series of functions located in different objects within a deep code hierarchy due to C++ language features: the innermost loop is positioned at material level; the energy lookup and the linear interpolation belong to an energy grid object. Not to mention that there are several hierarchies like nuclide, Doppler broadening approach in the code structure between these two objects. Indirect external function calls present in the innermost

loop inhibit natural vectorization. Moreover, polymorphism widely used to evaluate competing algorithms complicates the situation due to its considerable call overhead.

As stated before, effective vectorization is a combination of efficient data movement and high SIMD identification. Typical cross-section data are organized by isotopes in MC calculations. It is a natural way to store the data since this is how data are obtained from physical experiments. As for calculation, however, required data are actually far from each other due to the traditional AoS organization.

How to organize data to aid natural vectorization in MC calculations is definitely worth investigating. Even applying vectorization in MC calculations is important, there is little reported on this subject. Liu [92] explores vector processing strategies for MC calculations with the XSBench code [93]. Results show that using SIMD technique can bring 11% speedup for energy lookups but data prefetch is the crucial (speedup around 43%) to achieve higher performance on CPU, MIC, and GPGPU.

This work confirms the fact that energy lookup algorithms are all memory-bound thus data prefetching to reduce cache misses brings more acceleration than vectorization. A paper by Ozog [94] presents two techniques to vectorize MC calculations. The first one, particle banking, can help the simulation code have higher SIMD potential. The second one is to optimize the conventional history-based algorithm with tunings like SIMD directives, intrinsics, MKL, data alignment and so on. Both papers mention that data prefetch can bring significant performance improvement.

3.6.1 History-Based and Event-Based

The conventional history-based Monte Carlo simulations do not fit well with novel architectures due to the lack of natural vectorization potential. In this algorithm, the basic computing unit always starts with calculating a physic event (for example, cross-section, free-distance, interaction type, etc.) and followed by a geometry event to determine where takes place the next physic event. The entire *history* continually repeats such event-switch between the physic and the geometry until the target neutron disappears in the system.

In order to have better data locality in each event, Brown [95] presents an event-based algorithm with the help of "shuffle" operations: calculations are organized as events thus parallel computing units always do the same operations at the same time. This vector-friendly was popular in the 1980s when supercomputers mainly relied on large vector machines but then disappeared from the scene along with the decline of vector

processors. Recently, it comes back in the spotlight due to the renaissance of SIMD technique.

One early attempt [73, 81] has been made for employing history-based algorithm in a continuous-energy MC code on GPGPU architectures. The WARP simulation code described in this work is not a GPGPU adaptation of some existing CPU code. It is an application targeting the GPGPU platform since its creation. Host CPU is responsible for initialization and collecting data, important computing tasks including the main transport loop are all offloaded to remote accelerators.

In another paper [94], it has been briefly explained that full event-based implementation is difficult to carry out due to obstacles like vector compatible data structure, unpredictable data transfers and choosing proper SIMD candidates. This opinion is confirmed by a recent work [96] in which a simplified event-based model is created to evaluate its pure improvement. Results show that event-based particle banking can achieve up to 90% of vector efficiency and a speedup of 4× over the traditional history-based algorithms. Considering this is a simplified ideal case for event executions, such performance improvement might never appear in a real-case simulation. Besides, it should be noted that achieving data-level parallelism is only one aspect of full hardware utilization.

For problems limited by memory latency or bandwidth, optimizations other than improving vector efficiency is more crucial. Even the advantage of event-based algorithms is widely reported by similar studies [97–100], the community always hesitates to spread them in real applications owing to various implementation difficulties. Finally, history-based algorithms are still of priority for energy lookup calculations.

3.6.2 Accelerators

Computing accelerators expose a manycore feature to perform massively parallel calculations. There has been much research done on applying them to accelerate Monte Carlo particle transport calculations which include PATMOS, WARP, XSBench, ARCHER [101], OpenMC, RMC [102, 103], and so on. It can be found that among these research work, GPGPUs are greatly studied for the MC calculation while only a few papers report on MIC evaluation. Implementations details for each project are not repeated here, but several thoughts about accelerator utilization will be discussed.

GPGPUs maximize parallelism at the expense of high data transfer overhead through I/O buses, so recognizing the proper computing kernel to offload to devices is important for execution efficiency. For Monte Carlo transport calculations, identifying the

candidate kernel is not evident. In a recent work of PATMOS [87], cross-section calculations have been ported on GPGPUs. This hotspot performs computationally expensive Doppler broadening on the fly thus it should benefit well from accelerators. Results show that the on-the-fly calculation with three Nvidia K80 GPUs can achieve the same performance as the binary lookup on one CPU in a real simulation. However, further evaluation indicates that one Knights Landing can outperform two K80s combined with one CPU.

The GPGPU code is carried out with the cooperation of Nvidia, various algorithms and manual tuning are evaluated to make an optimal implementation choice. Even with these efforts, the code efficiency is much lower than expected. One possible reason is that cross-section calculations are not the right hierarchy level to make full use of GPGPU for Monte Carlo transports. Exact performance analysis is necessary to determine whether the speedup brought by massive parallelism at this level is inhibited by the expensive data transfer between the host and the remote. Other GPGPU projects like WARP and ARCHER call the GPGPU kernel from the higher level transport loop, which could provide useful insights for future PATMOS development.

Modern CPUs tend to include more computing cores to improve performance, but this trend makes GPGPU accelerations more complicated to realize. Suppose we want to achieve full GPGPU utilization and in the meanwhile keep the host CPU busy for efficiency concern, multiple GPU kernels launched from different CPU cores will lead to serious concurrency problem. As far as we are concerned, this issue has not been reported in the MC community since common applications rarely perform multi-threading and GPU parallelism at the same time. On the flip side, simplex GPU parallelism wastes most CPU computing resources because only one out of a dozen cores is used. As can be imagined, effective hybrid parallelism requires considerable implementation efforts.

Not like so many attempts to vectorize (or parallelize) memory-bound energy lookups with accelerators, optimizing the on-the-fly Doppler broadening is rarely reported in the community. PATMOS carried out the first evaluation for the on-the-fly SIGMA1 on both GPGPU and MIC. Since the cross-section reconstruction should be totally CPU-bound and possess high SIMD potential, the corresponding vectorization on MIC and GPU is really worth investigating.

3.6.3 Cross-Section Computations

The energy lookup process in MC calculations typically employs the binary search. A number of methods have been proposed in the past as alternatives to improve lookup efficiency. Brown [104] describes a new hash-based energy lookup method performed

on a logarithmic scale. An unionized energy grid [105] presented by Leppänen shows significant speedups but require much more memory space. Lund [106] reports a similar fractional cascading algorithm to relieve memory requirement. One recent work [107] has compared a number of optimizations on unionized methods and hashing methods with OpenMC, and their results show that overall code speedup factors of 1.2-1.5× can be obtained compared to the conventional binary search. Apart from these existing algorithms, other variations of the binary search or table lookup schemes (for example, interpolation search, tree search, etc.) might be useful for cross-section calculations.

Doppler broadening of cross-section data has been an important problem in neutron transport simulations for a long time. Several algorithms have been developed in this category. The most well-known SIGMA1 [79] proposed by Cullen is the reference algorithm employed by preprocessing libraries like NJOY. The Serpent code [108] revisits and optimizes a target motion sampling temperature treatment method [109] proposed in the 1980s. This method can reconstruct cross-sections with an arbitrary base temperature at a cost of considerable computational time. Yesilyurt [110] introduces a regression model based on a series expansion of the multi-level Adler-Adler formalism with temperature dependence. It has been reported that this method has a relatively high performance but requires more memory space due to extra precomputed data. The multipole method introduced by Hwang [111] show a different way to represent resonance information in the cross-section tables: only a few thousand *poles* are required instead of hundreds of thousand pointwise energy values. A drawback is the accuracy degradation at higher temperatures, as well as the high computational cost. Despite these disadvantages, the multipole method has the lowest memory requirement among all methods and thus could make detailed simulations involving thermal-hydraulic feedback possible in a Monte Carlo code. A new windowed-multipole method proposed by Josey [112] simplifies the original physical model by ignoring certain resonance-interference. The optimized method is by itself 40% slower than the conventional table lookup and leads to 7.9% performance degradation in a real simulation. In addition to reconstructions of resolved resonance region, two methods to perform on-the-fly Doppler broadening of unresolved resonance region cross-sections are presented in a recent work by Walsh [113].

3.6.4 Shared-Memory Model

The parallelism of shared-memory systems is more complicated than that of distributed ones due to complex cache/memory subsystems. Therefore, multi-threading applications rarely get the same performance scaling that can be observed with MPI multi-processing. One former study [114] provides comprehensive insights into this topic. This paper shows that the reservation station (RS) stalls from high latency operations make computing

units doing nothing but wait for data in most of the time; both FLOP usage and memory-bandwidth are far from exhausted. As a conclusion, the calculation is basically limited by memory access latency.

Though increasing the RS size and reducing processor's clock rate can improve this situation, the paper suggests exploring new algorithms with more floating-point operations (for example, on-the-fly Doppler broadening) instead of memory accesses. Using hyper-threading can improve memory access capacity but such effort will also transform the problem from latency-bound to bandwidth-bound. In another paper [115] from the same group, an energy banding method has been introduced to improve cache efficiency for energy lookup calculations. By dividing the entire energy range into small segments, corresponding cross-section tables are light enough to fit into the last level cache (LLC). This method allows better data locality within the random walk process and efficiently reduces LLC misses.

Chapter 4

Energy Lookup Algorithms

It has been pointed out that the energy lookup in large cross section tables is the major performance hotspot of Monte Carlo calculations. This is the case in particular with criticality calculations involving many isotopes and few tallies. Because of the considerable variations between different isotopic cross section (Figure 3.2), finding a generic energy lookup solution well-suited for any isotope in any problem is not evident. Typically, a simple binary search is employed to perform the indexing in the cross section tables. However, retrieving data in large tables with the bouncing binary search results in high cache misses (65% at last level cache [114]) and therefore degrades efficiency. Moreover, this algorithm shows very little vectorization potential.

This part of work started with the PATMOS using the binary search and the isotope hashing as two energy lookup methods. A polymorphism interface had been preserved in order to switch between the two methods. At that time, the code had been parallelized on the shared-memory and distributed-memory systems with OpenMP and MPI respectively but no studies on vectorization had been carried out.

Previous studies focused on this problem have already proposed several alternatives, but none of them addressed the issue of evolving architectures and especially many-core ones. Therefore, in the first part of the thesis work, we decide to implement a large collection of competing energy lookup algorithms in PATMOS and investigate useful techniques for the performance improvement on both CPU and MIC.

4.1 Working Environments

Before introducing the implementation and optimization work, target architectures and the test case involved in this part of work will be detailed in this section.

4.1.1 Machines

The experimental environment for tests presented in this chapter is comprised of:

- `SB`: dual-socket Intel Xeon Sandy Bridge E5-2670 workstation with 2×8 cores running at 2.6 GHz hosting 4 MIC PCIe cards.
- `KNC`: Intel Xeon Phi coprocessor, pre-production of Knights Corner prototype C0, 61 cores running at 1.2 GHz, 16 GB GDDR5 memory with ECC enabled, MPSS version 3.3.
- `BDW`: dual-socket Intel Xeon Broadwell E5-2697v4@2.3GHz, 2×18 cores, hyper-threading & Intel Turbo Boost disabled, 256 GB DDR4.
- `KNL`: Knights Landing 7250@1.4GHz, 68 cores with 4 threads per core, 16GB MCDRAM, 96GB DDR4, clustering mode: quadrant, memory mode: flat.

In the beginning of the thesis, SB and KNC represented the latest architectures at that time thus all tests were initially performed on these two architectures. In mid-2016, BDW and KNL came out so the same evaluations are re-performed on the new machines as well.

4.1.2 `PointKernel` Benchmark

Unless otherwise specified, one single test case called `PointKernel` is applied for all performance analysis in this chapter.

The test case is the neutron simulation of a slowing down problem from a 2 MeV source in an infinite medium at a temperature of 900 K, composed of all the 390 isotopes of the nuclear data library ENDFBVII.0 [116]; the main components of the mixture are ^{1}`H` ($2 \times 1O^{22} atoms/cm^3$) and ^{238}`U` ($1O^{22} atoms/cm^3$) in order to have a classical Pressurized Water Reactor (PWR) spectrum, the other isotopes intervening as trace elements ($1O^{16} atoms/cm^3$). This simulation is a representative of PWR burn-up computations with 100,000 particles in each batch.

4.2 Porting and Profiling

4.2.1 Adaptations to KNC

KNC is a coprocessor computer architecture running its own operating system. Building around the standard x86-64 instruction set, it is expected to extend the parallel execution

infrastructure with little or no modification of source code. Compared to other accelerators like GPGPU, the main advantage of MIC is the simplicity of porting work [117]. Programmers do not have to learn a new programming language but may compile their source codes by specifying MIC as the target architecture. The classic programming languages (Fortran, C, C++), as well as parallel libraries like OpenMP and MPI, can be directly used regarding MIC as a "classic" x86 based many-core architecture.

The initial PATMOS code had been evaluated on CPU before porting on KNC, however, extra code adaptations to KNC are necessary since the C++ language is partially supported by the compiler on coprocessors. For example, using = *default* to explicitly declare member functions has been found not available on MIC. Besides, *Standard Template Library* (STL) containers (like *std::vector* and *std::list*) of user-defined objects without the default constructor can not be identified by the compiler thus a container of pointers pointing to objects instead of a container of objects should be used to solve the problem.

4.2.2 Code Profiling

Code profiling is the essential work before all optimizing efforts. After transforming the prototype on KNC, finding performance bottlenecks of the code becomes the major concern. The idea is to take the same simulation benchmark on both SB and KNC so as to make a comparative analysis. Profiling differences between the two tests provide a better understanding of these two architectures and guide on where to carry out following optimizations.

4.2.2.1 Profiling on Intel Sandy Bridge

Intel VTune is a user-friendly multi-thread profiling tool. With its practical graphic user interface, users can easily carry out profiling on different modes. By analyzing the conventional binary search method with the *Basic Hotspots* mode, it has been found that the calculation of material's total cross section can take up to 90% of overall simulation time in the *PointKernel* test case on SB. Results show that all top five performance hotspots are related to the calculation of macroscopic total cross section. Taking a more specific look at Figure 4.1, one can observe that the STL *std::lower_bound()* function (the binary search used to perform table lookups) takes about 64.1% of overall computing time. When counting on all operations involved with cross section computation, this figure will raise up to 95.8%. Such conclusion is confirmed by previous studies as well [104–106], thus how to optimize the cross section computation become a very important issue for any Monte Carlo neutronic codes.

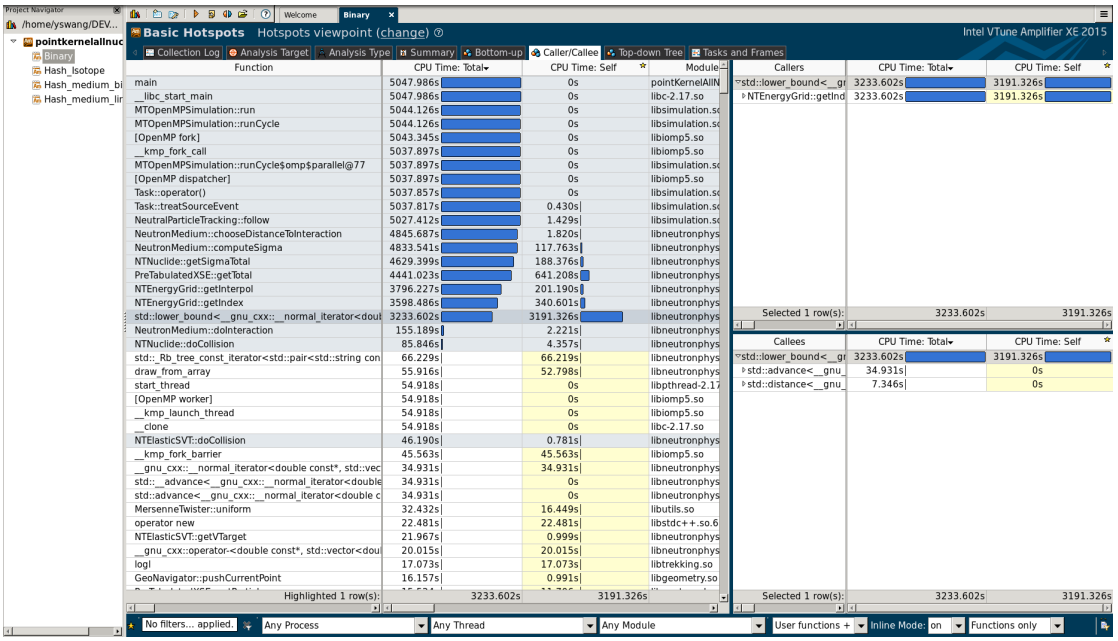


FIGURE 4.1: *Caller-Callee* view for `PointKernel` on Intel Sandy Bridge.

4.2.2.2 Profiling on KNC

Profiling work on KNC was more complicated than what one has been supposed to be. There were always certain problems to use VTune on the first generation KNC since it could not collect profiling data back to the host and therefore the profiling crashed every time. Under this circumstance, TAU was utilized as a temporary replacement. TAU is capable of gathering performance information from both OpenMP and MPI codes. Besides, it is supported to run profilings on KNC. TAU-specific compilation options should be added into the program CMake files. We verify that the TAU performance analysis corresponds well to that of VTune (Figure 4.2). However, this toolkit could not recognize new STL functions: loops with *std::generate()*, or *std::for_each()* declarations are treated as irregular events instead of loops. As a consequence, profiling results are far less readable and comprehensive compared to that of VTune. Finally, using VTune to perform code profiling on KNC has been realized with the relatively latest VTune release. Performance analysis of the binary search method shows that the cross-section computation is always the number one performance bottleneck on both SB and KNC.

4.3 Binary Search and Alternative Search Methods

The binary search method is the "reference" search scheme for Monte Carlo neutron transport codes. For every computation of the macroscopic cross-section, the binary search is applied respectively to the energy grid of each nuclide (390 in total in `PointKernel`).



FIGURE 4.2: TAU profiling for `PointKernel` on Intel Sandy Bridge.

No extra data need to be pre-computed or stored and therefore it suffers less memory penalty from complicated problems. Another consideration for using the binary search is that it preserves the computation accuracy without simplifying standard nuclear data. Due to these two reasons, this time-consuming method is still popular at present. Especially for reference codes like TRIPOLI and MCNP, which are required to afford large problem size and maintain full numerical accuracy.

4.3.1 Manual Binary Search

PATMOS employs the *std::lower_bound* function as the standard binary search method. Moreover, another manual binary search is also evaluated in PATMOS to improve lookup efficiency:

```
int lowerBound(const vector<double>& vec, double key) {
  int begin = 0, end = vec.size();
  while (begin < end) {
    int mid = (begin + end) >> 1; // bit shift instead of division
    if (key > vec[mid])
      begin = mid + 1;
    else
      end = mid;
  }
  return begin;
}
```

This function has two unconditional branches per iteration, one of which is essentially unpredictable. There is no way for the processor to know in which of the two halves of the search space the element we are looking for will be, which means that it expects to get on average one branch miss-prediction per two iterations [118]. The essential of this implementation is quite similar to the STL function, except that the STL uses templates such as *iterator*,*advance*, and *distance* instead of bit-set operations.

4.3.2 Vectorized N-ary Search

One natural idea to vectorize a binary search is to extend the conventional half-interval search to an N-interval search, or N-ary search. In theory, this method should not be more efficient than the binary search due to the gather operation. On the other hand, it is also a method which can benefit from vectorization since multiple border elements can be compared simultaneously against the key value compared to the one middle element of the binary search (Algorithm 2). In order to investigate the efficiency contribution of vectorization as well as the overhead related to the gather operation, the following N-ary search is evaluated in PATMOS:

Algorithm 2: N-ary search method.

```
1  index = 0;
2  Load 8 copies of key value to vec_key (vmovapd);
3  range = array_size ≫ 3;
4  while range > 1 do
5  |  Load (from array[index+range*0] to array[index+range*7]) to vec_boundaries;
6  |  Compare vec_boundaries with vec_key (vcmppd), store the result in cmp;
7  |  res: Count the number of bits set to 1 in cmp (popcnt);
8  |  index += (8-res-1)*range;
9  |  range = range ≫ 3;
10 end
11 while array[index] < key do
12 |  ++index;
13 end
14 return index;
```

When the lookup process arrives at the root of the tree structure where no more than two array elements are presented within each search interval, we then use a simple linear search to find the final index. It should be noted that a minimum of 16 elements in an array is needed to kick off the use of SIMD instructions, and for smaller arrays, the simple linear search is used for efficiency.

4.3.3 Vectorized Linear Search

Vectorizing the conventional binary search sacrifices the cache locality due to the gather operation. The linear search, however, performed on a contiguous memory space can be also vectorized in order to take advantage of the powerful VPU on MIC. Moreover, with the help of hashing (Section 4.6), the search interval is dramatically narrowed to a small range with often only a few elements, where a simple linear search may fully profit from cache effects and become more efficient than the binary search. Our optimizations follow the idea of one former study [118] which provides a comprehensive view of optimizing linear and binary search with SSE2 instructions.

Algorithm 3: Basic vectorized linear search. The `key` value is the target energy.

```
1  index = 0;
2  Load 8 copies of key value to vec_key (vmovapd);
3  for do
4  |  Load 8 (from array[index] to array[index+7]) array elements to vals (vmovapd);
5  |  Compare vals with vec_key (vcmppd), store the result in res;
6  |  if res ≠ 0 then
7  |  |  break;
8  |  end
9  |  index += 8;
10 end
11 Count trailing zero bits of res (tzcnt), store the result in offset;
12 return index + offset;
```

The basic SIMD algorithm for linear search is represented in Algorithm 3. The 512-bit VPU allows 8 simultaneous `double` operations. For each iteration in the loop, 8 elements in the array pointed by `index` are compared with the `key` value. Since all **vcmppd** operations in `IMCI` are already combined with a mask operation, the output of the comparison is a 8-bit `char` variable. Once the result is no longer zero, we determine the `offset` by counting trailing zero bits of this result. The final result is the accumulated `index` plus `offset`. In the basic algorithm, there is one conditional branch per 8 array elements, but such proportion can still be reduced by unrolling the loop (Algorithm 4). With the branchless method, each iteration makes 16 comparisons and the final `offset` is packed from 2 8-bit variables with a simple bit-set operation.

Further loop unrolling with 4 and 6 elements has been tried, but only minor improvement can be observed.

Algorithm 4: Branchless vectorized linear search.

```
1  index = 0;
2  Load 8 copies of key value to vec_key (vmovapd);
3  for do
4  |  Load 8 (from array[index] to array[index+7]) array elements to vals0
   |   (vmovapd);
5  |  Load another 8 (array[index+8] to array[index+15]) elements to vals1
   |   (vmovapd);
6  |  Compare vals with vec_key (vcmppd), store the result in res0;
7  |  Compare vals with vec_key (vcmppd), store the result in res1;
8  |  res = res0 + (res1≪8);
9  |  if res ≠ 0 then
10 |  |  break;
11 |  end
12 |  index += 2×8;
13 end
14 Count trailing zero bits of res (tzcnt), store the result in offset;
15 return index + offset;
```

4.3.3.1 Data Alignment for C++ Member Variables

One issue of using IMCI intrinsics is the restriction brought by data alignment. Most IMCI intrinsic instructions require aligned data as input. Utilizing such intrinsics with non-aligned data will result in segmentation fault during execution. In addition, aligned data would increase code performance due to the way CPUs handle memory accesses, thus data alignment is rather important and necessary for programming with intrinsics. PATMOS employs STL containers (or functions) as much as possible in order to facilitate implementation and guarantee code reliability, including `vector`, `map`, etc. Although `boost aligned_allocator` will handle any STL containers, it should be noted that such alignment only effects on the first element of containers but not each element of them. Moreover, the problem becomes a little more complicated in our case, cause except for the standard STL container, we also need align object member variables like `int`, `double`, or even `__m512d` for VPU register. Generally speaking, `__declspec(align(n))` instruction can be used to align member variables of a C++ object only if the object is allocated statically. In our code, however, STL containers are largely used and allocated dynamically thus the `__declspec(align(n))` instruction is no longer workable. On the other hand, `_mm_malloc()` only work for variables inside member functions but not for object member variables. One possible solution to align C++ member variables is rewriting `new` and `delete` operators:

```
void *operator new (size_t size)
{
  return _mm_malloc(size, 64); // or _aligned_malloc() depending on compiler
}
```

```
void *operator new[] (size_t size)
{
  return _mm_malloc(size, 64);
}

void operator delete (void *mem)
{
  _mm_free(mem);     // or _aligned_free() depending on compiler
}

void operator delete[] (void *mem)
{
  _mm_free(mem);
}
```

4.3.4 Comparison of Different Search Algorithms

Figure 4.3 represents timing results of different search algorithms performed on these two architectures respectively, while running on a single thread. The test result is the average time over ten million searches for retrieving the index corresponding to a random energy input. The points of the grid are randomly chosen and the array size is varied from 10 to 500 elements. SIMD optimization has significant speedup for the linear search. The vectorized branchless linear search can bring a factor of 10× improvement over the basic linear search on MIC. On CPU, however, it performs less efficiently than the basic vectorized linear search. For any array less than 200 elements, the optimized linear search turns out to be always more efficient than the binary search on MIC. Similar conclusions can be drawn on CPU as well. Results indicate that the threshold array size for choosing between the linear or the binary search is 200. Under this value, the linear search is more efficient.

As for the N-ary search, it does not seem to be able to outperform the binary search. Profiling tests indicate that gathering split array elements to prepare the comparison tree make the optimization counterproductive. Though larger width decreases the depth of a tree structure, the non-cache friendly memory access to separated array elements introduces a penalty which leads to worse lookup efficiency.

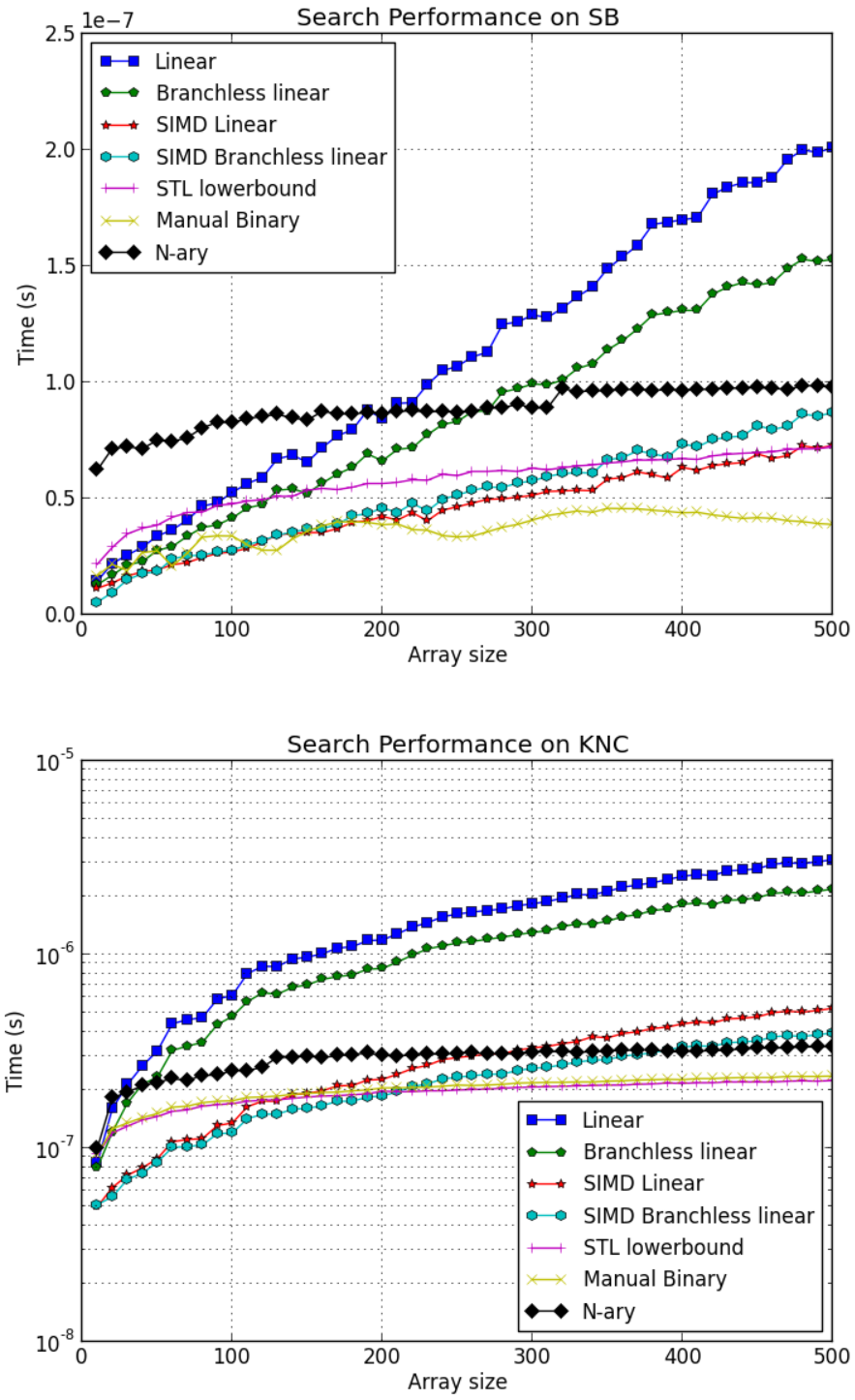


FIGURE 4.3: Performance of several versions of search method as a function of array size. The standard binary search has been tested in two different implementations, one coming from the STL *std::lowerbound* and one rewritten for this work (Manual Binary).

4.4 Unionized Energy Grid

The unionized energy grid is used to assemble cross-section datasets of all isotopes into one global table. Complementary energy points for isotopic datasets should be interpolated and supplemented in the unified table. Extra data lead to a huge increase in the memory demand. However, since only one lookup is performed for every macroscopic cross-section update, the speedup brought by the unionized grid can be also very impressive.

Leppänen figured out two methods to overcome the problem with insufficient memory: the grid thinning method and double indexing method. The grid thinning method preserves only important energy points (local min. and max. bound, minimum of threshold reactions, etc.) in the unified grid. This method indeed reduces memory requirement, but with a loss of numerical accuracy. The second solution is constructing an indexing table over the unified grid. For a target energy value: a first indexing provides its location on the unified table; a second indexing with the indexing table indicates its original locations in each isotopic cross section table. Compared to the original unionized grid, less extra energy points need to be pre-computed or stored before simulation and therefore this optimized method requires less memory space. In order to guarantee the simulation accuracy, the double indexing is chosen for PATMOS implementation.

4.4.1 Optimizations for the Unionized Method

4.4.1.1 Initialization

The global unionized table combines grids of all nuclides without duplicates. Algorithm 5 shows the naive serial algorithm to establish the unionized grid: we firstly check if one energy point already exists in the unionized grid (with user-defined `Contained()`); if it's true, we do nothing; otherwise, we will put this value as the last element of `u_grid` (with `std::remove_copy_if()`); at last, we sort `u_grid` to get the final unionized grid. Thanks to the use of STL functions, the implementation is direct and simple. But results shown in Table 4.1 indicate that such algorithm has rather poor performance. On one hand, we can certainly optimize the sequential implementation with parallelism but on the other hand, it should be noted that `std::remove_copy_if()` works well with an unsorted array, but maybe not proper for sorted ones in our case.

We then tried Algorithm 6 to improve this situation: the `std::merge()` is dedicated to combine two sorted containers into a new one with all its elements in order; in the

Algorithm 5: Initializing unionized grid with `std::remove_copy_if()`.

```
1 Initialize an empty unionized grid (std::vector<double>): u_grid;
2 for Nuclide i in all nuclides in the material do
3 |   Load energy grid of nuclide i: vals;
4 |   std::remove_copy_if(
  |    vals.begin(),vals.end(),
  |    std::back_inserter(u_grid),
  |    Contained(u_grid.begin(),u_grid.end())
  |    );
5 end
6 Sort the u_grid;
```

end, `std::erase()` and `std::unique()` are employed to remove duplicates. Such a few changes by using different STL functionalities bring significant speedup (see Table 4.1), which points out that choosing proper STL algorithm is very important for the implementation.

Algorithm 6: Initializing unionized grid with `std::merge()`.

```
1 Initialize two empty std::vector<double>: u_grid, temp_grid;
2 for Nuclide i in all nuclides in the material do
3 |   temp_grid = std::move(u_grid);
4 |   Load energy grid of nuclide i: vals;
5 |   std::merge(
  |    vals.begin(),vals.end(),
  |    temp_grid.begin(),temp_grid.end(),
  |    std::back_inserter(u_grid)
  |    );
6 end
7 u_gird.erase(std::unique(u_grid.begin(), u_grid.end()), u_grid.end());
```

Further optimization can be carried out with the concurrent container of Intel TBB library. Intel TBB is a C++ template library for shared memory parallel programming. Compared to the STL, TBB concurrent containers allow simultaneous data access and update by multiple threads. To create an unionized grid with TBB, we only need `push_back()` member function to insert every energy points into the concurrent vector: `u_grid`. The grid itself can automatically handle everything (e.g. vary container size, load new elements) in parallel. Results (Table 4.1) show that TBB containers bring impressive gain against the two previous methods.

	`std::remove_copy_if()`	`std::merge()`	TBB
CPU	453.47	4.32	1.06
KNC	3,269.35	36.93	4.65

TABLE 4.1: Time (s) to initialize an unionized grid with different methods.

4.4.1.2 Data Structure

In our preliminary implementation, an indexing table is constructed for each isotope; we then combine all the list into one 2D indexing table. For the benchmark `PointKernel`, for example, we have an indexing table with 390×3,574,598 elements. Profiling results show that access to the indexing table is one of the performance hotspots for the unionized method because the indexes for each isotope belong to different arrays. A better data structure can be accomplished by simply transposing the indexing table to 3,574,598×390 form: for each energy points in the unionized grid, the corresponding indexes of all 390 isotopes are contiguous in the memory space and can be loaded at once for all isotopes. Table 4.2 shows that such an optimization can bring a speedup between 20% and 50% for the simulation, but with worsening performance efficiency as the number of threads increases.

	SB		KNC	
	Serial	16 threads	60 threads	240 threads
Original	131.39 s	12.22 s	60.92 s	25.18 s
Optimized	85.06 s	10.23 s	39.56 s	20.63 s
Speedup	1.54×	1.20×	1.54×	1.21×

TABLE 4.2: Speedup of optimized unionized method for the *PointKernel* test case.

4.5 Fractional Cascading

Fractional cascading is a technique to speed up search operations for the same value in a series of relevant datasets [119]. Lund [106] firstly applied it in the OpenMC code to improve the energy lookup process for Monte Carlo transports. The basic idea is to build a unified grid for the first and second isotopes, then for the second and third, etc. When using this mapping technique, once we find the energy index in the first energy grid, all the following indexes can be read directly from the extra index tables without further computations. Compared to the global unionized method, the fractional cascading technique greatly reduces memory usage. The construction of cascading grid can be represented in following steps:

- For each isotopic energy grid E, create an extra grid M. The last extra grid M_k equals to E_k itself
- M_i is a sorted list containing every element of E_i and every other element of M_{i+1}

- For each element of M_i, a first pointer *p1* gives its index (std::lowerbound-1: last element less than the given value) in E_i; a second pointer *p2* gives its index (std::lowerbound: first element no less than the given value) in M_{i+1}

Given an energy e, the algorithm of calculating macroscopic cross section is shown in Algorithm 7.

Algorithm 7: Energy lookup with cascading grid

```
1  for all isotope in a material do
2      if first isotope then
3          get index i of e in M_0 with std::lowerbound;
4          get current isotopic energy grid index: pointers[i].current;
5          get next isotopic extra grid index: pointers[i].next;
6      else
7          // get current isotopic energy grid index with former next_index i_next;
8          if e ≤ M_i[i_next-1] then
9              get current isotopic energy grid index: pointers[i_next-1].current;
10             get next isotopic extra grid index: pointers[i_next-1].next;
11         else
12             get current isotopic energy grid index: pointers[i_next].current;
13             get next isotopic extra grid index: pointers[i_next].next;
14         end
15         calculate microscopic cross section;
16         update macroscopic cross section;
17     end
18 end
```

4.5.1 Reordered Fractional Cascading

The fractional cascading method builds augmented grids and associated indirection indexes for each isotope and the next. Since every isotope has an energy grid length largely differing from each other and we always perform a search process only in the first mapping table, the order in which the energy grids maps are generated may have a significant effect on searching performance.

	SB		KNC	
	Serial	16 threads	60 threads	240 threads
Alphabetical	135.09 s	14.94 s	74.94 s	28.45 s
Min-Max	124.21 s	14.21 s	65.65 s	26.19 s
Speedup	1.09×	1.05×	1.14×	1.09×

TABLE 4.3: Effects of isotope ordering when constructing fractional cascading maps for the *PointKernel* test case.

We thus tested several ordered approaches: a) alphabetical (this is the order in which the isotopes are actually read into memory; a) randomized; c) from longer to shorter grid; d) from shorter to longer grid. The first three approaches gave similar results, while the ordering of grids from shortest to longest gave a performance improvement of 10% (see Table 4.3 where we show the results for alphabetical and min-max orders).

4.6 Hash Map

Recently, Brown [104] revisited a hash map algorithm for Monte Carlo energy lookup. Results show that this method remarkably improved search efficiency without significant increase in memory requirements. Another feature of this scheme is that there is no data thinning, so numerical accuracy can be fully preserved. It has been pointed out that hash table is a useful scheme recommended for any Monte Carlo optimization efforts.

The original hashing energy lookup is based on a logarithmic scale energy grid. The entire energy range is divided up into N equal intervals, and for every individual isotope inside the material an extra table called *U_Grid* is established to store isotopic bounding indexes of each interval. The new search intervals are thus largely narrowed with respect to the original range and can be reached by a single floating-point division. The hashing can be also performed on a linear scale; the search inside each interval can be performed by a binary or linear search. In the original paper, a logarithmic hashing was chosen with $N = 8000$ as the best compromise between performance and memory usage. A *key* value computed with the input energy indicates the *minimum* and *maximum* of the narrowed search interval. In PATMOS, hashing functions characterized by the *U_Grid* are evaluated on two levels: one at the isotope level and another at the material level.

4.6.1 Isotope Hashing

Hashing at isotope level means each nuclide has its own *U_Grid*. these bounding information are valid only within isotope. In PATMOS, Constructing isotope hashing can be represented in following steps:

1. Take the first and the last element in the energy grid as *emin* and *emax*;
2. Determine the *U_Grid* size with a N value;
3. Make $du = \frac{\log(emax)-\log(emin)}{N}$;
4. For each energy value $E = emin \times \exp(n \times du)$ n=1,2...N, store its energy grid index in *U_Grid*.

Given an energy E, the lookup process is:

- Compute a key value for each isotope $u = (int)\frac{\log(E)-\log(emin)}{du}$;
- Use search schemes (linear search, binary search, etc.) to locate E in the energy grid between two bounding indexes: U_Grid[u] and U_Grid[u+1].

4.6.2 Material Hashing

Material hashing is quite similar to isotope hashing. The only difference is that U_Grid determines its minimum and maximum bound by checking all energy grids in the material. Isotopic cross-sections are hashed by the same U_Grid parameter, thus a key value at the material level will be valid for all isotopes within it. In this way, the key value computation will be performed only once at the beginning instead of being repeated for every isotope and therefore the computation is supposed to be more efficient.

4.6.3 Efficient Hashing Strategies

The efficiency of two hash-based methods depends on several factors. Unit tests are carried out respectively for each aspect in order to find the optimal solution.

4.6.3.1 Hashing Size

The hashing size `N` determines the number of energy points in the hash bins. A greater value of `N` means more bins and therefore fewer elements in each bin. In theory, larger hashing size requires more memory space but performs better. It has been mentioned [104] that dividing the whole energy range into $N \simeq 8000$ segments is a reasonable compromise between performance and memory usage.

Our first test case *PointKernelU238* is a variation of *PointKernel*, where the only isotope present is ^{238}`U`. We have carried out the test varying the hashing size `N` from 200 to 32,000 and found that a larger hashing size always provides better search efficiency. But when we switched to the *PointKernel* test, where all 390 isotopes are present, $N \simeq 500$ is observed to be the most efficient value with a gain of around 7% over the original optimal $N \simeq 8,000$. This may be due to the fact that the average energy grid size is around 12,000 (see Figure3.2) which is not much bigger than 8,000.

In the hashing method at isotope level, we can specify a different hashing strategy for each isotope, based on its own energy grid properties. Following this idea, we implemented a hashing method with $N = 500$ for energy grids with less than 30,000 elements

and $N = 10,000$ for the rest. Results show that such adapted hashing size provides a 5% speedup over the universal $N = 8,000$.

4.6.3.2 Logarithmic vs. Linear Hashing

The hashing method revisited by Brown [104] is based on a logarithmic scale. It is not obvious that such hashing organization is optimal for all cases. Therefore, Walsh [107] proposed to establish the hash function simply on a linear scale. In the benchmark `PointKernel`, we observed that linear scale takes up to 2.2× more time than the logarithmic scale and performs even slightly slower than the conventional binary search.

A detailed analysis of energy point distributions shows that while a logarithmic scale is the most appropriate to have balanced hash bins outside the resonance region when resonances are involved a linear scale may be more effective for bin balance. We thus implemented a mixed hashing, using linear scale between 1eV and the end of the resolved resonance region and a log scale otherwise. Timing results performed on the `PointKernelU238` test case shows that this mixed hashing method brings no acceleration on either CPU or MIC and it performs a little slower (no more than 10%) than the pure logarithmic method. Such performance degradation comes from algorithm branching when determining in which hashing region the lookup is to be carried on. Further optimization by varying hash size and reorganizing hashing region could still be worth exploring.

4.6.3.3 Search Efficiency within Hashing Bins

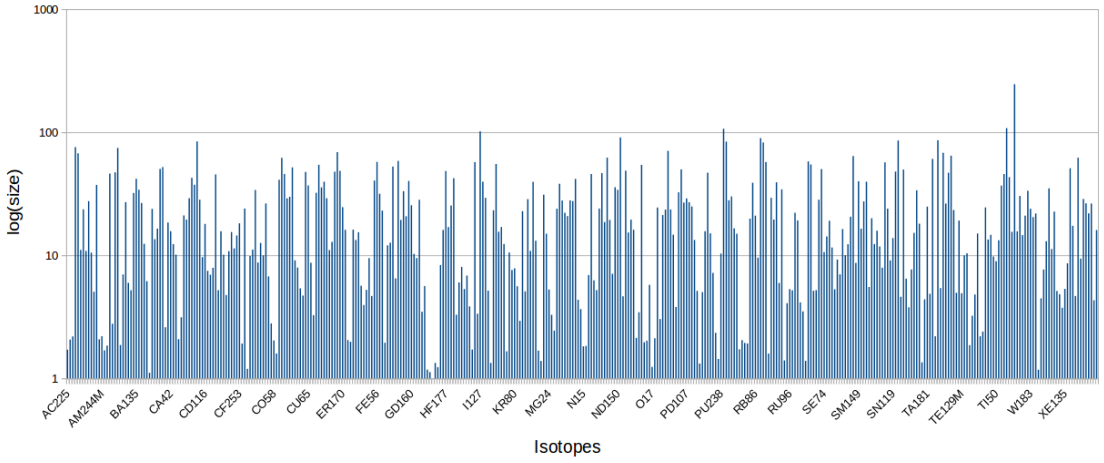


FIGURE 4.4: The average hash bin size for an isotope varies significantly from each other.

The energy point distributions are very irregular, due to the presence of resonances where the cross sections change of several order of magnitude in intervals of just 1 eV. Hashing the 390 isotopes with $N = 500$ for example, the average number of elements in each bin is around 21. But $\simeq 81\%$ of hash bins have no more than 3 elements, while in the resonance region isotopes like ^{239}PU and ^{238}U may have bins with up to 4000 energy points (Figure 4.4). Such unbalanced distribution leads to performance degradation. In order to investigate this issue, we measured average search time in each hash bin. The results confirm that larger hash bins take longer during indexing, but timing differences between isotopes are not very important (see Table 4.4). For example, the largest ^{238}U energy grid is 246× more voluminous than the smallest ^{3}H, but indexing energy points using linear search in a ^{238}U hash bin is on average only 1.5× longer than in ^{3}H.

Isotope	Energy grid size	Average bin size	Time (s)
^{3}H	469	1	2.473e-07
^{127}I	50,759	101.52	2.997e-07
^{239}PU	53,284	106.57	3.039e-07
^{235}U	53,936	107.87	3.058e-07
^{238}U	122,567	245.13	3.756e-07
Average:	10,761	21.52	2.49e-07

TABLE 4.4: Search performance in the hash bin (with N=500).

4.7 N-ary Map

After revisiting all existed energy lookup algorithms for Monte Carlo codes, we figure out that none of them is improved by the use of SIMD techniques. Since it has been mentioned above that vectorization is the key to evaluate algorithms for multi and many-core systems, so we propose a variation on N-ary tree [120] for its vectorization and cache-locality properties.

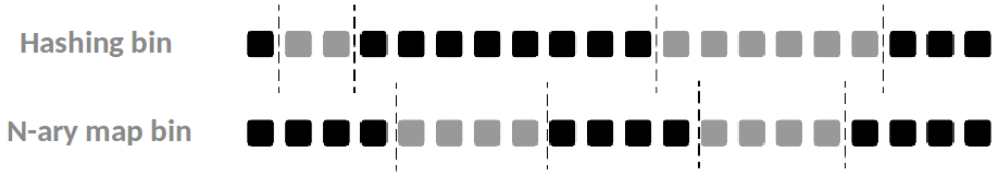


FIGURE 4.5: Balanced bin size with N-ary map method.

Directly vectorizing a binary search to an N-ary search (Subsection 4.3.2) can lead to a significant increase of discontiguous memory accesses. But if we pre-gather these separator values and store them in VPU registers before calculations, the overhead should be much less significant. Following this idea, we propose and evaluate an N-ary

energy lookup method: corresponding mapping points are pre-retrieved from the array and stored in VPU registers. SIMD instructions help to locate the interval between two mapping points; a binary search is then performed to carry out the lookup within mapping bins to find the final index.

Generally speaking, the width and the depth can affect the lookup efficiency of a tree structure, and this is also true for the N-ary energy lookup. Test results show that multi-level tree structures increase memory access overhead since indexing from one level to the next causes longer access latency. The deeper and wider a tree structure is, the more memory space is required and the more cache misses will be caused. More memory requirement also indicates that the program would have more *register splits*, because all mapping information is pre-computed and stored in VPU registers, and the number of 512-bit registers is rather limited for each core. Moreover, boundary points of one interval may be located in different branches of a tree, which can directly result in algorithm branching and enforce serialization inside SIMD instructions.

Finally, a flat N-ary map with only one level has been found the most suitable for the energy lookup. This method is similar to hashing, but the energy bins are not built on equal energy intervals but on an equal number of energy grid points. The algorithm is the following, where we use *IMCI* intrinsics:

1. For each isotope, divide the energy grid `E` into 32 segments with range $r = \frac{grid\ size}{32}$
2. Create an extra indexing list `I` containing $r \times j, (j = 0, 1, 2...31)$ and insert the index of last element in the energy grid at the end of the list
3. Store 32 variables `E[I[j]]` (`j = 0,1,2...31`) into 4 `__m512d` vector variables

Such a method creates only a few hundreds of bytes of extra mapping information for each isotope. The procedure for indexing an energy value (the key value) is represented by the following steps:

1. Load 8 copies of `key` value to `vec_key` (with instruction **vmovapd**)
2. Compare `vec_key` with 4 `__m512d` vector variables, store 4 comparison results in 4 8-bit variables: `cmp0`, `cmp1`, `cmp2`, `cmp3` (**vcmppd**)
3. Pack 4 results into one 32-bit unsigned variable: `res` = `cmp0` + (`cmp1`$\ll$8) + (`cmp2` + (`cmp3`$\ll$8))$\ll$16
4. Find the final index with a binary search in the interval between `E[I[res-1]]` and `E[I[res]]`

4.8 Full Simulation Results

In this section, we present the comparison of different lookup methods in terms of performance, scalability, and memory footprint. It should be noted that optimizations for each method have been already taken into account.

4.8.1 Performance

	SB (16 threads)			KNC (60 threads)			KNC (240 threads)		
	Time (s)	Speedup	Memory (MB)	Time (s)	Speedup	Memory (MB)	Time (s)	Speedup	Memory (GB)
Binary	32.90	1.00×	514	107.28	1.00×	970	35.76	1.00×	2.88
Cascading	14.21	2.32×	640	65.65	1.63×	1126	26.19	1.37×	2.9
HashIsotope	15.59	2.11×	527	69.81	1.54×	1030	25.86	1.38×	2.89
HashMaterial	14.05	2.34×	526	68.51	1.57×	1027	24.33	1.47×	2.89
N-ary map	-	-	-	84.96	1.26×	982	31.52	1.13×	2.88
Unionized	10.23	3.22×	5862	39.56	2.86×	6348	20.63	1.73×	8.0

TABLE 4.5: Performance and memory usage for energy lookup algorithms for the *PointKernel* test case.

As shown in Table 4.5, the reference binary search is the slowest method in the test. The unionized grid is the most efficient time-wise, but at the cost of a dramatic increase in memory foot-print (times 11× on CPU). The two variants of hashing methods and fractional cascading have nearly the same efficiency on both CPU and MIC.

A disappointing fact is that with all the algorithms tested the CPU always outperforms the KNC. This is true even for vectorized algorithms like the N-ary search, in spite of the fact that the theoretical vectorization speedup is twice as big in the KNC (×8) that in the CPU (×4).

4.8.2 Memory Optimization

As can be found in Table 4.5, certain data related to isotopic exiting distributions were still replicated for each working thread which requires about 10 MB of supplementary memory space per thread. While when working with CPU this is hardly noticeable, in MIC architecture the added cost in memory footprint is far from negligible (see Table 4.5).

We have therefore optimized the PATMOS data structures to reduce the memory foot-print of the duplicated objects to about 200KB∼500KB (depending on the method) per thread, which corresponds to a reduction factor of about 20 with respect to the previous version. Moreover, the code structure of material hashing and fractional cascading methods has also been re-designed to improve data locality. Results show that such

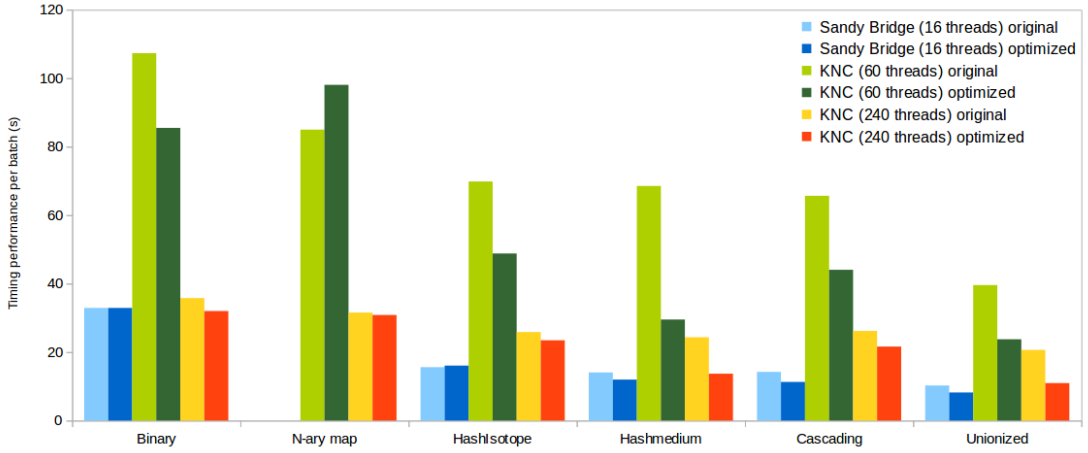


FIGURE 4.6: Performance comparison between naive and memory-optimized prototype (the lower the better).

reduction efforts bring considerable acceleration (Figure 4.6) to all methods but the N-ary search. Basically, both time performances and speedup against the reference binary search are improved (Table 4.6). Compared to the common multi-core architecture, we can conclude that the many-core benefits more from memory reduction. Among the methods, hashing at the material level shows the most improvement from this optimization, especially on KNC where it approaches the performance of the unionized grid. An exception to this general trend is the N-ary map on KNC, which is improved with hyper-threading (240 threads) but not without (60 threads). This behavior is not completely understood.

	Memory (MB)	SB (16 threads)		KNC (60 threads)		KNC (240 threads)	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Binary	367	32.88	1.00×	85.47	1.00×	31.98	1.00×
Cascading	502	11.27	2.91×	44.02	1.63×	21.61	1.47×
HashIsotope	368	16.05	2.04×	48.8	1.54×	23.44	1.36×
HashMaterial	368	11.96	2.74×	29.5	1.57×	13.67	2.34×
N-ary map	367	-	-	89.42	0.96×	27.04	1.18×
Unionized	5862	8.2	4.0×	23.74	3.6×	10.95	2.92×

TABLE 4.6: Performance and memory usage after memory optimization.

4.8.3 Scalability

As for algorithm scalability (Figure 4.7), we note that algorithm efficiency on MIC with 60 threads is much better than the efficiency on CPU with 16 threads. Though hyper-threading with 4 threads per core results in shorter computing times on MIC, the algorithm efficiency (compared to one single thread) degrades as well. Results on CPU are quite coherent with previous work: algorithms with better performance have

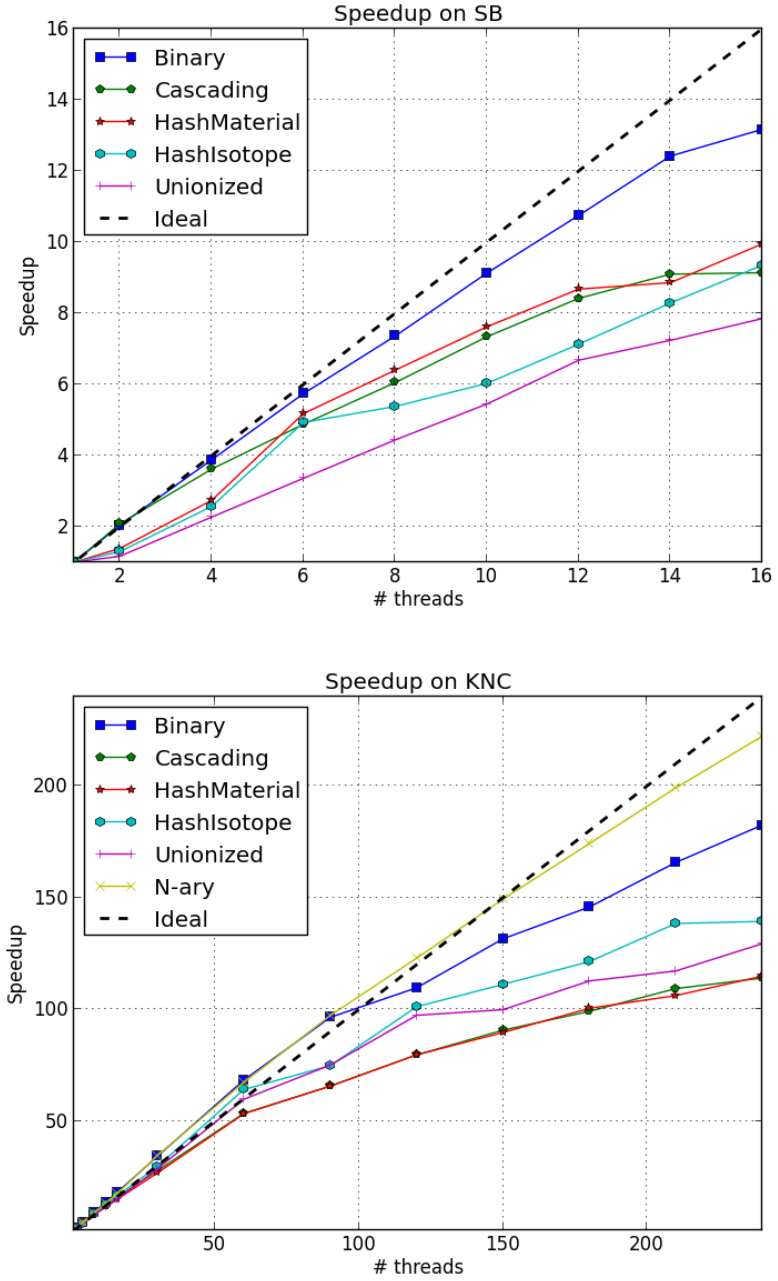


FIGURE 4.7: Speedup for energy lookup algorithms for the `PointKernel` test case.

a strong tendency to have worse scalability. Even after memory optimization, however, the material hashing and fractional cascading methods have poorer scalability on KNC than the unionized grid method. The N-ary map method has been found to have the best scalability on MIC architecture. The logarithmic material hashing method seems to retain the best balance between performance, scalability and memory footprint.

4.8.4 Results on Latest Architectures

In order to evaluate the contribution of latest architectures to the energy lookup algorithm, similar performance tests are repeated on BDW and KNL. The implementations used to evaluate different architectures are basically the same except few code adaptations for the intrinsic instructions.

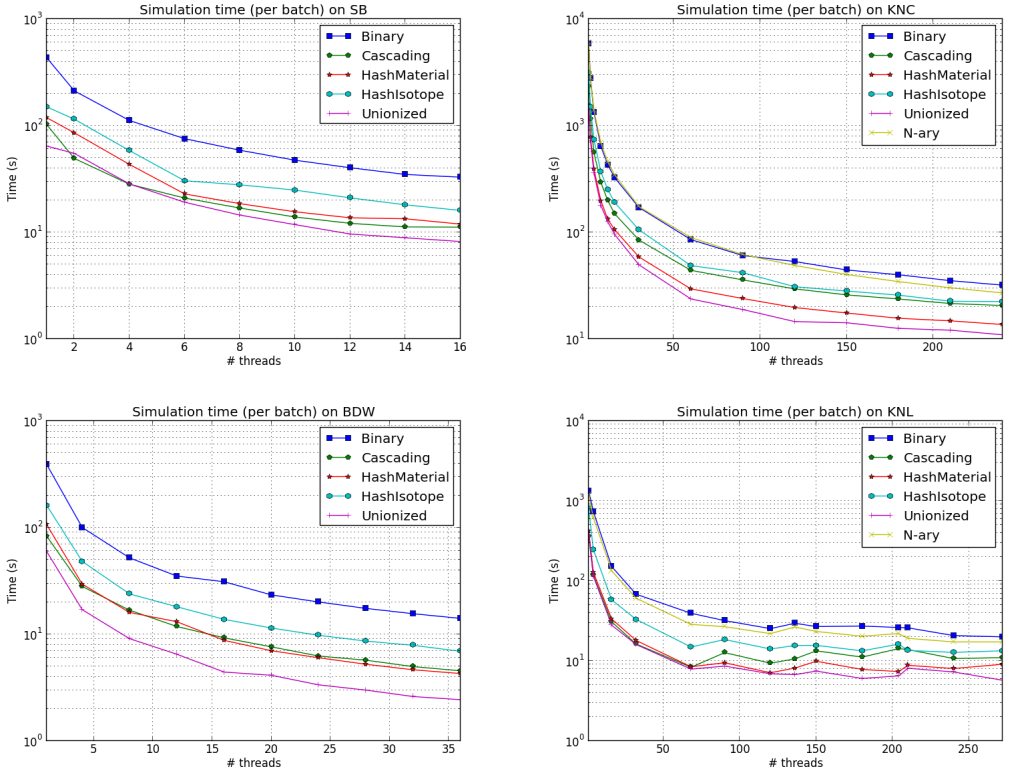


FIGURE 4.8: Execution performance of competing energy lookup algorithms.

As shown in Figure 4.8, execution performance on BDW is quite similar to that of SB. In both cases, the unionized grid has the best performance since it has relatively better cache utilization. Fractional cascading and material hashing have nearly the same performance since they both do the major computation at the material level (no need to perform calculations individually in each nuclide). Compared to the isotope hashing, computing at higher hierarchy allows these two methods to possess more data locality. As for many-core architectures, it can be observed that the material hashing approaches the performance of the unionized grid on KNL. Moreover, a significant difference between KNC and KNL is that all lookup algorithms benefit little from hyper-threading on the latest MIC processor. The previous KNC employs the in-order execution with which instructions are statically scheduled in a compiler-generated order. KNL, however, extends the instruction-level parallelism with the out-of-order execution where the program can still execute other instructions behind a stalled instruction. Therefore, the

hyper-threading on KNC can hide the frequent stall overhead during the random walk process while on KNL such overhead does not exist anymore due to the dynamically-scheduled instructions.

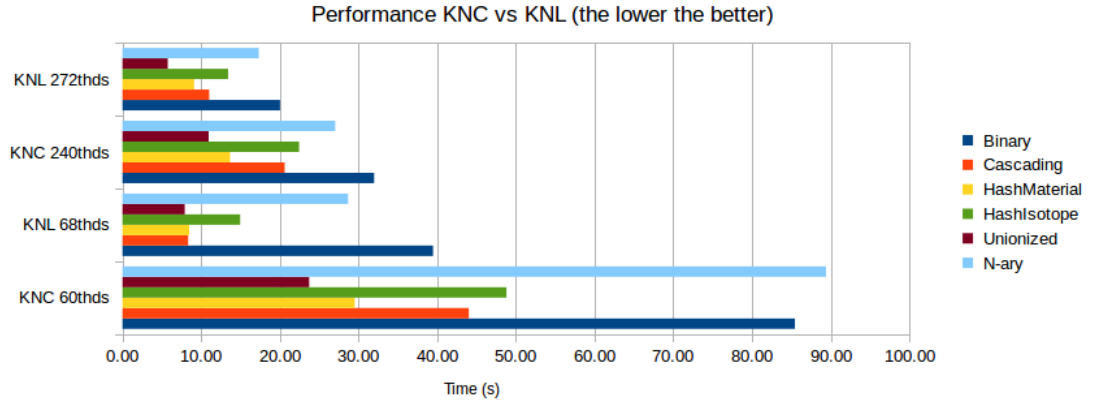


FIGURE 4.9: KNL shows 1.5× performance improvement over KNC for energy lookup algorithms by using hyper-threading.

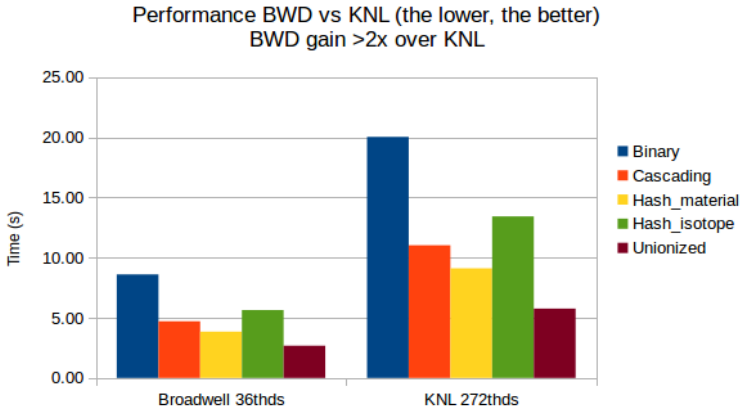


FIGURE 4.10: BDW shows > 2× speedup over KNL for energy lookup algorithms.

Figure 4.9 shows an overall performance comparison between these two architectures with and without hyper-threading. It can be observed that KNL has an overall 2.5× over KNC without using hyper-threading. By using all four hardware threads per core, however, KNL computing efficiency decreases more significantly than that of KNC which results in poorer performance improvement (around 1.5×). In conclusion, our implementation proposed for KNC works well on KNL. Porting codes to the latest architecture will directly have performance improvement. Results (Figure 4.10) show that even KNL outperforms the latest generation KNC with a factor of 2×, it is still 2× less efficient than the same generation CPU. This situation can be mitigated by finding new algorithms with more FLOP work.

Chapter 5

Cross-section Reconstruction

RECONS [121] is a mini-app dedicated to evaluating algorithms for the on-the-fly Doppler broadening. The code is developed by François-Xavier Hugot of CEA/DEN/SERMA. It performs direct reconstructions for the cross-sections located in the resolved resonance region (Figure 5.1); in other regions, cross-sections are computed with energy lookup methods. Cross-section data are from the library ENDF/B-VII.1 [75]. PREPRO [122] is used to transform the standard data to meet the calculation requirement of RECONS. STL functions and lambda expressions are largely used for the fast-implementation. Physical results can be evaluated by a comparative test between the reconstruction and the pretabulated data.

Unlike previous work in which different energy lookup methods are evaluated directly in PATMOS, RECONS is an independent code that should be integrated into our implementation. In this chapter, the algorithms used in RECONS, the implementation in PATMOS, and test results will be presented.

5.1 Working Environments

Before introducing algorithms and implementations, involved architectures and the test case will be detailed in this section.

5.1.1 Machines

The experimental environment for tests presented in this chapter is comprised of:

- `BDW`: dual-socket Intel Xeon Broadwell E5-2697v4@2.3GHz, 2×18 cores, hyper-threading & Intel Turbo Boost disabled, 256 GB DDR4.

- `KNL`: Knights Landing 7250@1.4GHz, 68 cores with 4 threads per core, 16GB MCDRAM, 96GB DDR4, clustering mode: quadrant, memory mode: flat.

This part of work started in mid-2016 when BDW and KNL were already available, so studies on SB and KNC were abandoned.

5.1.2 `PointKernel` Benchmark

The `PointKernel` applied in this chapter is different from previous ones. It is a neutron simulation in an infinite medium at a temperature of 300 K. 100,000 particles are calculated in each batch. The nuclear data library ENDFBVII.1 is used. The main components of the mixture are ^{1}`H` ($2 \times 1O^{22} atoms/cm^3$) and ^{240}`PU` ($1O^{22} atoms/cm^3$), other isotopes intervening as trace elements ($1O^{16} atoms/cm^3$). Interaction types are restricted in elastic scattering and radiative capture.

5.2 Algorithm

The on-the-fly Doppler broadening in RECONS employs formulas which are similar to the *multipole* method. It should be noted that only Single-Level Breit-Wigner and Multi-Level Breit-Wigner formalism have been implemented in the code; investigations on the Reich-Moore formalism are still ongoing.

5.2.1 Resolved Resonance Region Formula

For the resolved resonance region, cross sections of different reaction types are calculated with formulas presented in the ENDF manual [124, 125].

5.2.1.1 Single-Level Breit-Wigner

The elastic scattering cross section in the Single-Level Breit-Wigner (SLBW) formalism can be written as:

$$\sigma(E) = \sum_{l=0}^{NLS-1} \sigma^l(E) \tag{5.1}$$

where

$$\begin{aligned} \sigma^l(E) =& \frac{(2l+1)4\pi}{k^2} \sin^2 \phi_l \\ &+ \frac{\pi}{k^2} \sum_J g_J \sum_{r=1}^{NR_J} \frac{\Gamma_{nr}^2 - 2\Gamma_{nr}\Gamma \sin^2 \phi_l + 2(E - E_r')\Gamma_{nr} \sin(2\phi_l)}{(E - E_r')^2 + \Gamma_r^2/4} \end{aligned} \tag{5.2}$$

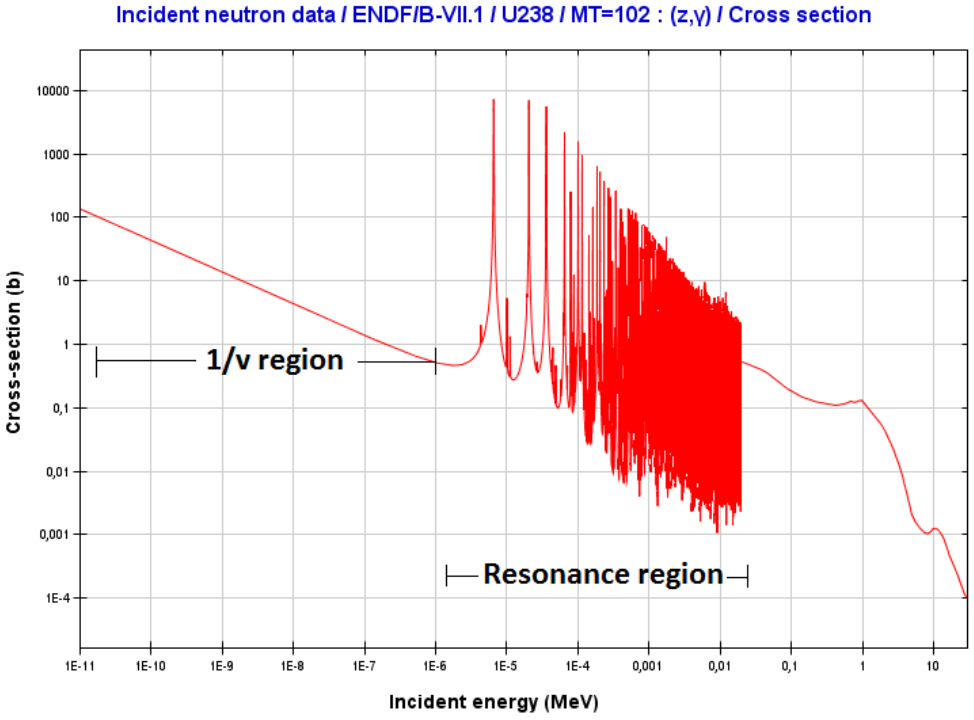


FIGURE 5.1: On-the-fly Doppler broadening applies on the resonance region [123].

In this formula:

- E: energy of the neutron
- l: relative neutron-nucleus angular momentum
- NLS: number of spin levels l
- J: angular momentum ("spin") of the resonance state
- NR_J: number of resonances for a given pair of l and J values.
- Γ_{nr}: neutron line width at energy E
- Γ_r: total resonance width at energy E

Besides, the statistical spin factor g_J is given by the following equation:

$$g_J = \frac{(2J+1)}{2(2I+1)} \tag{5.3}$$

where I is the angular momentum of the target nucleus.

The neutron wave number k at energy E in the center-of-mass system is calculated with the following relation:

$$k = 2.196771 \times 10^{-3} \frac{AWRI}{AWRI + 1.0} \sqrt{E} \tag{5.4}$$

where $AWRI$ is the ratio of the target isotope mass to the neutron mass.

The shifted resonance energy E_r' is defined as:

$$E_r' = E_r + \frac{S_l(|E_r|) - S_l(|E|)}{2P_l(E_r)}\Gamma_{nr}(|E_r|) \tag{5.5}$$

where S_l is the level-shift factor presented as a function of angular momentum:

$$\begin{aligned} S_0 &= 0 \\ S_1 &= -\frac{1}{1+\rho^2} \\ S_2 &= -\frac{18+3\rho^2}{9+3\rho^2+\rho^4} \\ S_3 &= -\frac{675+90\rho^2+6\rho^4}{225+45\rho^2+6\rho^4+\rho^6} \\ S_4 &= -\frac{44100+4725\rho^2+270\rho^4+10\rho^6}{11025+1575\rho^2+135\rho^4+10\rho^6+\rho^8} \end{aligned} \tag{5.6}$$

and the penetration factor P_l is defined as:

$$\begin{aligned} P_0 &= \rho \\ P_1 &= \frac{\rho^3}{1+\rho^2} \\ P_2 &= \frac{\rho^5}{9+3\rho^2+\rho^4} \\ P_3 &= \frac{\rho^7}{225+45\rho^2+6\rho^4+\rho^6} \\ P_4 &= \frac{\rho^9}{11025+1575\rho^2+135\rho^4+10\rho^6+\rho^8} \end{aligned} \tag{5.7}$$

The angular momentum hard-sphere phase shift ϕ_l at energy E is calculated with:

$$\begin{aligned} \phi_0 &= \hat{\rho} \\ \phi_1 &= \hat{\rho} - tan^{-1}\hat{\rho} \\ \phi_2 &= \hat{\rho} - tan^{-1}\frac{3\hat{\rho}}{3-\hat{\rho}^2} \\ \phi_3 &= \hat{\rho} - tan^{-1}\frac{\hat{\rho}(15-\hat{\rho}^2)}{15-6\hat{\rho}^2} \\ \phi_4 &= \hat{\rho} - tan^{-1}\frac{\hat{\rho}(105-10\hat{\rho}^2)}{105-45\hat{\rho}^2+\hat{\rho}^4} \end{aligned} \tag{5.8}$$

Parameters ρ and $\hat{\rho}$ are defined as $k \times a$, where a is channel radius defined as:

$$a = 0.123 \times AWRI^{1/3} + 0.08 \tag{5.9}$$

The radiative capture cross section in the SLBW formalism can be written as:

$$\sigma(E) = \sum_{l=0}^{NLS-1} \sigma_{\gamma}^{l}(E) \tag{5.10}$$

where

$$\sigma_{\gamma}^{l}(E) = \frac{\pi}{k^2} \sum_{J} g_J \sum_{r=1}^{NR_J} \frac{\Gamma_{nr}\Gamma_{\gamma r}}{(E - E_r')^2 + \Gamma_r^2/4} \tag{5.11}$$

and $\Gamma_{\gamma r}$ is the gamma width at the resonance energy.

The fission cross section in the SLBW formalism can be written as:

$$\sigma(E) = \sum_{l=0}^{NLS-1} \sigma_{f}^{l}(E) \tag{5.12}$$

where

$$\sigma_{f}^{l}(E) = \frac{\pi}{k^2} \sum_{J} g_J \sum_{r=1}^{NR_J} \frac{\Gamma_{nr}\Gamma_{fr}}{(E - E_r')^2 + \Gamma_r^2/4} \tag{5.13}$$

and Γ_{fr} is the fission width at the resonance energy.

5.2.1.2 Multi-Level Breit-Wigner

In the Multi-Level Breit-Wigner (MLBW) formalism, the equations are nearly the same as SLBW, except that a resonance-resonance interference term should be taken into consideration for the elastic scattering cross section. So the Equation 5.2 should be rewritten as:

$$\sigma^{l}(E) = \frac{\pi}{k^2} \sum_{J} g_J \sum_{r=1}^{NR_j} \frac{G_r\Gamma_r + 2H_r(E - E_r')}{(E - E_r')^2 + \frac{1}{4}\Gamma_r^2} \tag{5.14}$$

where

$$G_r = \frac{1}{2} \sum_{s=1;s\neq r}^{NR_J} \frac{\Gamma_{nr}\Gamma_{ns}(\Gamma_r + \Gamma_s)}{(E_r' - E_s')^2 + \frac{1}{4}(\Gamma_r + \Gamma_s)^2} \tag{5.15}$$

$$H_r = \sum_{s=1;s\neq r}^{NR_J} \frac{\Gamma_{nr}\Gamma_{ns}(E_r' - E_s')}{(E_r' - E_s')^2 + \frac{1}{4}(\Gamma_r + \Gamma_s)^2} \tag{5.16}$$

5.2.1.3 Doppler Broadening

All above equations are given for a particular isotope without Doppler broadening (at $0K$). In order to reconstruct the cross section at a higher temperature T, we can use the formula that says that if cross section value at $T = 0$ can be written as [126]:

$$\sigma(E, T = 0) = \sum \frac{A + B(E - E')}{(E - E')^2 + \Gamma^2/4} \tag{5.17}$$

then it follows that

$$\sigma(E,T) = \sum \frac{1}{\sqrt{\pi}\Delta} \left\{ \frac{2\pi A}{\Gamma} \Re\left[W(\xi)\right] - \pi B \Im\left[W(\xi)\right] \right\} \tag{5.18}$$

where

$$\xi = \frac{E' - E}{\Delta} + i\frac{\Gamma}{2\Delta}; \qquad \Delta(E,T) \sim \sqrt{TE}, \tag{5.19}$$

and W is the Faddeeva function (Section 5.2.2).

5.2.2 Faddeeva Function

Faddeeva function [127] is a complex error function involving an integral of e^{t^2}, which makes it closely related to the Fresnel and Dawson functions. The real and imaginary parts are called the Voigt functions. The functions are scaled by e^{-z^2}, which gives it favorable numeric properties that help to avoid numeric overflow.

Faddeeva function is defined as:

$$W(x) = e^{-x^2}(1 + \frac{2i}{\sqrt{\pi}} \int_0^x e^{t^2} dt) = e^{-x^2}[1 + erf(ix)] = e^{-x^2} erfc(-ix) \tag{5.20}$$

Integral representations:

$$W(x) = \frac{i}{\pi} \int_{-\infty}^{\infty} \frac{e^{-t^2}}{x - t} dt = \frac{2ix}{\pi} \int_0^{\infty} \frac{e^{-t^2}}{x^2 - t^2} dt \tag{5.21}$$

5.3 Implementations and Optimizations

RECONS has been integrated into PATMOS with considerable adaptation work. For calculations outside the resolved resonance region, the hashing isotope (Subsection 4.6.1) is used instead of the binary search for higher lookup efficiency. In this section, the implementation choice of the Faddeeva function, as well as optimizations of the reconstruction kernel, will be presented with details.

5.3.1 Faddeeva Implementations

The choice of the Faddeeva implementation in PATMOS is important since the function is not provided by standard performance libraries like MKL. In our cross-section reconstruction algorithm, it is the only external and computationally expensive function in the hotspot kernel. Efficient implementations should be used to obtain a good runtime

performance and also to preserve a high numerical accuracy. In the following of this section, two implementations evaluated in PATMOS will be presented.

5.3.1.1 ACM680 W

The default Faddeeva implementation in RECONS (referred to as `ACM680::W`) is developed on C++ by François-Xavier Hugot. The program is inspired by the ACM Algorithm 680 proposed by Poppe and Wijers [128]. For a given complex number z, the subroutine computes $exp(-z^2) \cdot erfc(-iz)$ where *erfc* is the complex complementary error function. The accuracy of the algorithm for z in the first and the second quadrant is fourteen significant digits; in the third and forth is thirteen significant digits outside a circular region. The calculation efficiency of the algorithm is enhanced by using a different approximation in the neighborhood of the origin, where the naive Gautschi algorithm [129] becomes ineffective.

The original ACM 680 implementation uses double precision, while `ACM680::W` evaluates both single and double precision in terms of performance and accuracy. It should be noted that for small $|z|$ values, `ACM680::W` evaluates the Faddeeva function by using a power-series (Equation 7.1.5 in [130]).

5.3.1.2 MIT W

Faddeeva Package [131] (or `MIT::W`) is an open-source C++ code developed by Steven G. Johnson from the Department of Mathematics of MIT (Massachusetts Institute of Technology). It has wrappers for other languages like C, Matlab, Python, and so on. Unlike the `ACM680::W` implementation, `MIT::W` uses a hybrid algorithm to compute the Faddeeva function: a continued-fraction expansion similar to Algorithm 680 and 363 [129] for sufficiently large $|z|$ values while a new Algorithm 916 [132] for small $|z|$ values. The package developer indicates that the continued-fraction expansion significantly outperforms the 916 for larger $|z|$ at the cost of certain accuracy loss. `MIT::W` uses Taylor expansions in certain regions to avoid cancellation errors and Chebyshev polynomial approximations to converge faster. Precomputed lookup tables are usually used in the implementation to transform the costly computing kernel to a simple memory read. According to the developer, significant speedup has been observed by using this technique. The overall accuracy is at least thirteen significant digits in both the real and imaginary parts [131].

5.3.2 Scalar Tuning

From this subsection, reconstruction algorithms and optimizations implemented in PATMOS will be presented in the order of scalar tuning, vectorization, and parallelism. As shown in the previous section, the calculation of elastic is more complicated and expensive than capture and fission. Furthermore, elastic of MLBW is much heavier than that of SLBW. Therefore, only MLBW elastic is discussed for brevity. Algorithm 8 shows the primitive computing kernel, which follows tightly the equations presented before(Equation 5.14). This implementation has a lot defects, such as duplicates, no vectorization, branching inside the loop, etc. So in the rest of section, our optimization work will be detailed.

Algorithm 8: Primitive MLBW $\sigma_{elastic}$ kernel

Input: Energy E, AoS $R[N]$
Output: Cross section σ_e

```
1  for resonance R[i] i ← 0 to N do
2      E'_r, Γ_nr, Γ_r ← E, R[i];                    // E'_r, Γ_nr, Γ_r of resonance i
3      for resonance R[j] j ← 0 to N do
4          if i ≠ j then                              // do not accumulate itself
5              E'_s, Γ_ns, Γ_s ← R[i], E;             // E'_s, Γ_ns, Γ_s of resonance j
6              g_r ← E'_s, Γ_ns, Γ_s, E'_r, Γ_nr, Γ_r;
7              G_r += g_r;                                          // accumulate G_r
8          end
9      end
10     for resonance R[j] j ← 0 to N do
11         if i ≠ j then                              // do not accumulate itself
12             E'_s, Γ_ns, Γ_s ← R[i], E;             // E'_s, Γ_ns, Γ_s of resonance j
13             h_r ← E'_s, Γ_ns, Γ_s, E'_r, Γ_nr, Γ_r;
14             H_r += h_r;                                          // accumulate H_r
15         end
16     end
17     if theta = 0 then                                  // temperature at 0K
18         σ ← E, R[i], E'_r, Γ_nr, Γ_r, H_r, G_r;
19         σ_e += σ;                                              // accumulate σ_e
20     else                                              // Doppler broadening
21         enrc ← (E, theta, E'_r, Γ_r);
22         w ← faddeeva(enrc);
23         σ ← E, R[i], E'_r, Γ_nr, Γ_r, H_r, G_r, w;
24         σ_e += σ;                                              // accumulate σ_e
25     end
26 end
```

Scalar tuning includes optimization efforts related to mono-thread sequential executions. In order to prepare the application for further vectorization or parallelization, an optimized scalar implementation can benefit more from subsequent enhancements.

Algorithm 9: Simplified MLBW $\sigma_{elastic}$ kernel (other than 0K)

Input: Energy E, AoS $R[N]$
Output: Cross section σ_e

```
1  initialize arrays: E'_r[N], Γ_nr[N], Γ_r[N];
2  for resonance R[i] i ← 0 to N do
3  |  E'_r[i], Γ_nr[i], Γ_r[i] ← E, R[i];                    // E'_r[i], Γ_nr, Γ_r of resonance i
4  end
5  for resonance R[i] i ← 0 to N do
6  |  for resonance R[j] j ← 0 to N do
7  |  |  if i ≠ j then                                       // do not accumulate itself
8  |  |  |  h_r, g_r ← Γ_nr[..], Γ_r[..], E'_r[..];
9  |  |  |  H_r += h_r; G_r += g_r;                          // accumulate H_r & G_r
10 |  |  end
11 |  end
12 |  enrc ← (E, theta, E'_r, Γ_r);
13 |  w ← faddeeva(enrc);
14 |  σ ← E, R[i], E'_r[i], Γ_nr[i], Γ_r[i], H_r, G_r, w;
15 |  σ_e += σ;                                             // accumulate σ_e
16 end
```

5.3.2.1 Algorithm Simplification

In a lot of cases, simulation codes are developed to completely reflect all details of the numerical model. This can leads to a fact that model-loyal implementations replicate a lot of calculations and therefore, degrade the execution performance. For example, computing intermediate variables like $E'_r, \Gamma_{nr}, \Gamma_r$ is repeated three times for each resonance calculation (lines 2,5,12 of Algorithm 8). To solve this problem, in place of calculating them on-the-fly whenever they are required, $E'_r, \Gamma_{nr}, \Gamma_r$ are precomputed only once in the beginning (lines 2-4 of Algorithm 9). Two inner loops (lines 3-9 and lines 11-15 in Algorithm 8) basically compute the same intermediate variables and only make a difference for the final accumulation. As a result, these two loops can be merged into one to avoid replications (lines 6-11 of Algorithm 9).

5.3.2.2 Code Reorganization

At the program structure level, alternative processes like SLBW or MLBW as well as base temperature or other temperatures are originally manipulated in the same C++ `class`. If-conditions are largely used to distinguish different processes at runtime. For optimization, inheritance is used to separate each individual properly in their own `class` to avoid runtime branch check.

To calculate the penetration and shift factor, the naive implementation uses a recursive algorithm as shown below:

```
std::pair<double, double> penetrability_shift(int l, double rho)
{
  if (l==0) return std::make_pair(rho, 0.);
  double rho2=sqr(rho);
  auto ps = penetrability_shift(l-1, rho);
  double plm=ps.first, slm=ps.second;
  double denum = square(l-slm)+square(plm);
  return std::make_pair(rho2*plm/denum, rho2*(l-slm)/denum-l);
}
```

This recursive algorithm will introduce problems for vectorization, thus we chose a straight-line implementation by following tightly Equation 5.6 and Equation 5.7. The switch-condition in these equations lead to algorithm branching again. Since the *l-value* is a constant for each resonance, the branching down at the function level can be lifted up to the initialization. A first attempt is to use polymorphism to determine the corresponding calculation for each individual *l-value*. For such light functions, the overhead of polymorphism may counteract the enhancement by removing branches. Further optimizations could use templates with explicit specialization, but initial tests show such metaprogramming prevents directive vectorization in PATMOS.

5.3.2.3 Strength Reduction

Strength reduction is an operation that replace expensive calculations with equivalent simpler ones. Such simplification can improve execution performance but usually results in accuracy loss. According to the Colfax training [133], operations like `exp2()`, `log2()`, and `sqrt()` are programmed directly on the processor without software calculations. So in PATMOS, `exp()` calls are all replaced by the hardware-supported `exp2()`. Generally for modern processors, addition, subtraction and multiplication all take about five clock cycles while a division operation may require up to several dozen clock cycles. In a consequence, divisions should be transformed to corresponding multiplications or bitwise operations wherever possible:

```
// replace exp() with equivalent exp2()
constexpr double fast_exp(double x) {return exp2(x*1.44269504089);}

// use multiplication instead of division
const float X_recip = 1.f/X;
for (int i=0; i<N; ++i)
    Y[i] *= X_recip;
```

```
// bit shift to calculate x = y / 8.
double x =>> 3.
```

5.3.2.4 STL Functions

Since C++11, a large number of new loop representations combined with lambda function has been added into the STL. Functionalities of these new functions can be known directly from their definitions. For example, `std::generate` is responsible to initialize arrays; `std::accumulate` is used to perform reductions. With a quick glance at the function name, developers can understand the global semantics of the target loop. The advent of these functions can largely enhance code readability and simplify code implementations. From the view of the compiler, however, identifying these new STL functions as vectorization candidates is not considered as a first priority. Even with the indication of SIMD directives, compilers are not capable yet to perform vectorizations on these loops. The naive reconstruction algorithm heavily employs these loop functions, thus all of them have been rewritten back to the standard `for(;;)` form for better compiler compatibility.

5.3.3 Vectorization

Effective vectorization comes from a complicated balance of data layout and arithmetic operation [58], thus this work require developers with deep understandings for both software and hardware. In order to maintain the readability and performance of the code, we choose to use SIMD directives instead of low-level intrinsics or libraries.

As shown in Algorithm 10, `omp simd` directives are used for explicit vectorization of each loop. More precise manual tunings involved in our work will be detailed in the following of this section.

5.3.3.1 Collapse

Vectorization is supposed to apply on the innermost loop of the computing kernel. For implementations like PATMOS in which a dozen levels of nested loops are present, vectorizing associated loops is desired. The `collapse` clause (line 6 of Algorithm 10) is responsible to unroll multi-level loops into one flattened loop body. The figure `2` indicates that two loops are associated with the SIMD construct.

Algorithm 10: Optimized MLBW $\sigma_{elastic}$ kernel (other than 0K)

Input: Energy E, SoA $A[N_p], B[N_p]$...
Output: Cross section σ_e

```
1  aligned initialization: E'_r[N_p], Γ_nr[N_p], Γ_r[N_p], temp_gr[N_p], G_r[N_p], H_r[N_p];
2  #pragma omp simd aligned(..)
3  for i ← 0 to N_p do
4  |  E'_r[i], Γ_nr[i], Γ_r[i], temp_gr[i], ← E, A[i], B[i]...;        // E'_r, Γ_nr, Γ_r of i
5  end
6  #pragma omp simd collapse(2) aligned(..)
7  for i ← 0 to N_p do
8  |  #pragma omp simd reduction(+:H_r[i], G_r[i]) aligned(..)
9  |  for j ← 0 to N_p do
10 |  |  h_r, g_r ← Γ_nr[..], Γ_r[..], E'_r[..];
11 |  |  H_r[i] += h_r; G_r[i] += g_r;                    // accumulate H_r & G_r
12 |  end
13 |  G_r[i] -= temp_gr[i];                    // subtraction to remove branching
14 end
15 #pragma omp simd reduction(+:σ_elastic) aligned(..)
16 for i ← 0 to N_p do
17 |  enrc ← (E, theta, E'_r, Γ_r);
18 |  w ← faddeeva(enrc);
19 |  σ ← E, E'_r[i], Γ_nr[i], Γ_r[i], H_r, G_r, w, A[i], B[i]...;
20 |  σ_e += σ;                                          // accumulate σ_e
21 end
22 ...
23 #pragma omp declare simd processor(mic_avx512)
24 faddeeva(enrc) ...
```

5.3.3.2 No Algorithm Branch

Algorithm branch inside a loop body can seriously decrease the vectorization efficiency. As shown in the inner loop (lines 6-11 of Algorithm 9), the influence of other resonances in the region should all be accumulated for the target central resonance. If-condition (line 7 of Algorithm 9) is used to not accumulate itself. It should be noted that such algorithm branch can be auto-vectorized with mask operations. However, the vectorization efficiency can not be high as long as branches present inside the loop. In order to solve this problem, the self-influence is precalculated (lines 3-5 of Algorithm 10) before the main loop thus the if-branch can be avoided by simply subtract the precalculated value at the end of the loop (line 13 of Algorithm 10). For implementations that difficult to remove algorithm branches, another possible solution is to move if-conditions outside of the loop body. In this situation, at least in each if-branch SIMD operations can be guaranteed.

5.3.3.3 Loop Splitting

We often see that vectorization may not apply for the entire loop in a real problem due to issues like non-straight codes, sequential executions, etc. In this situation, separating the original one unique loop into several smaller ones and vectorizing those who have SIMD opportunities can improve execution performance.

```
for (int i=0; i<N; ++i) {
    Z[i] = X[i] + Y[i];
    func(Z[i]);     // external function that can not be vectorized
}
```

The loop shown above consists of two major operations: one is array addition; another is external function call. With the help of loop splitting, the array addition can be vectorized by directives:

```
#pragma omp simd
for (int i=0; i<N; ++i)
    Z[i] = X[i] + Y[i];

for (int i=0; i<N; ++i)
    func(Z[i]);     // external function that can not be vectorized
```

Though split loops result in redundant work, such overhead turns out to be negligible if used properly. Another case to use loop splitting is when a huge amount of calculations present in the loop body. Even the whole loop can be vectorized, there are so many intermediate variables to calculate and to store for each iteration. Since KNL concludes thirty-two 512-bit vector registers for each computing unit, extra data which can not fit into these registers will be put to the main memory. This is what is called a *register spilling* which can lead to significant performance degradation. Here, loop splitting can relieve vector register pressure by avoiding the *register spilling*. According to profiling results, the second loop (lines 5-16) in Algorithm 9 are already cause register spilling on BDW. Therefore, we separate the loop of accumulation and the loop of Doppler broadening in the optimized implementation (lines 7-14 and lines 16-20 in Algorithm 10).

5.3.3.4 Declare SIMD Directives

The Faddeeva function (line 13 of Algorithm 9) present in the inner-loop prohibits natural vectorization. Before, one possible solution could be inlining external functions to make them locally to the loop body. In the case of Faddeeva function, however, inlining

is not practical cause the function concludes complex instructions like if-conditions, for-loops with several hundred lines of codes in total. Even developers can enforce inlining, related issues like branches inside the loop will result in performance degradation.

Thanks to the `declare simd` directive, such complex non-straight-line expressions is now vectorizable. By simply using `#pragma omp declare simd` during function declaration, developers can fully rely on the compiler for vectorization. Our tests show that if-conditions or loops inside the function negatively impact the efficiency, but jump instructions (e.g. *goto*) will completely stop vectorization. It should be noted that the current `declare simd` directive can only apply on functions with regular inputs and outputs (pointers, *int*, *float*, *double*, etc.). Using *classes* or STL containers will cause compilation error. Another important point is that the `declare simd` directive can not be applied on a pure virtual function. Moreover, specifying `CPUID` [134] when using Intel compiler is necessary for full utilization of SIMD opportunities (line 23 of Algorithm 10). Though compile options like `-xHost` can indicate the compiler the latest ISA supported by the processor, the efficiency of vectorized functions without explicit hardware specification is far from optimal.

5.3.3.5 Float

Generally, double precision floating point variables are used in MC transport to guarantee numerical accuracy. For the multipole method in PATMOS, the single precision is also tested to achieve better performance. Theoretically, using *float* instead of *double* can directly bring a 2× performance speedup. Along with a different data type, consistency of precision should be carefully handled as well. Here the consistency consists of two parts: variables and functions.

```
const float pi = 3.14159265358979     // excessive expression
const float pi = 3.141593f          // proper expression

long p = q * 2;          // 2 is "int", should use 2L
double y = x + 1;     // 1 is "int", should use 1.
float m = n * 2.;     // 2. is "double", should use 2.f
float a = b * 1e3.;     // 1e3 is "double", should use 1e3f
```

Appropriate variable declarations shown above can avoid the costly runtime type conversion. This tiny code change seems to be negligible for the entire application development, but it can become a serious issue that decreases performance. As for functions, one can choose to transform them manually from *double* to *float* (e.g. `sinf()` instead of `sin()`) or explicitly use C++ `std` functions instead of C ones (`std::sin()` instead of `sin()`).

5.3.3.6 SoA

The input of the naive computing kernel is a list of resonances: each resonance represents a data structure containing resonance features like resonance energy, "spin" of resonance, etc. For scalar executions, each iteration calculates one resonance where all required data are contiguous in the memory. As for vectorization, however, the multiple data needed for each single instruction is far from each other. After optimization, several lists of resonance variables replace the original list of resonances. This AoS to SoA conversion brings better memory access pattern for vectorization (`Input` of Algorithm 9 and Algorithm 10). Meanwhile, flattened C-style arrays take the place of original `std::vector` containers for efficiency purposes;

5.3.3.7 Data Alignment and Data Padding

Modern processors have multiple levels of cache/memory hierarchy to efficiently transfer the data. Data read action through all hierarchies is set to start at specific address boundaries. Misaligned accesses to these boundaries will overlap two cache lines, and this will require an entirely new cache read in order to obtain the data. It could miss all the way out to the DRAM.

Generally, the compiler will generate a peeled loop before the main loop body to deal with elements beyond the alignment boundary. Similarly, a remainder loop just after the main loop will be created when loop iteration is not a multiple of vector register size (VL). These extra loops normally with a few iterations might be more costly than the main loop since they are usually under scalar execution or poorly vectorized with mask operations. Thanks to the AVX-512CD, emerging architectures like KNL and Skylake can mechanically vectorize them. Still, their presence will introduce vectorization efficiency issues. In our implementation, aligned data is allocated dynamically by using `_mm_malloc`; `aligned` clauses (lines 2,6... of Algorithm 10) are used to get rid of runtime alignment check and help the compiler choose effective instructions without issues. On the other hand, we pad data for each array up to a VL-multiple size N_p (lines 1,3... of Algorithm 10). Padded data are set to be zero during accumulation in order not to change numerical results.

5.3.4 Threading

Generally, MC transport simulations divide input particles into batches and carry out calculations within several nested loops. Parallelism takes place at batch level due to the largest iteration number (between 10^5 and 10^6). Unbalanced workloads are hidden by

launching considerable particles for each thread task. Static scheduling is used to ensure the reproducibility of numerical results. This pattern works well with the "pretabulated" method since the efficiency of table lookups vary a little among different isotopes. In the case of the multipole, however, cross-section computing rates of isotopes significantly differ from one to another. According to profiling results, optimal OpenMP region can bring a further gain of 20% to our multipole implementation on KNL. Therefore, thread affinity, as well as OpenMP scheduling with smaller workloads for each thread, are explored to approach ideal performance.

5.4 Tests and Results

Several tests are carried out inside and outside the PATMOS in order to evaluate series of optimization efforts.

5.4.1 Unit Test of Faddeeva Functions

An effective Faddeeva algorithm is important for Doppler broadening within cross-section reconstruction. Profiling results show that this function can represent 70% of overall computing time for an SLBW cross-section. Two Faddeeva implementations mentioned in Section 5.2.2 are tested outside the PATMOS. Optimizations presented in Section 5.3 make sense for Faddeeva implementations as well thus a unit test is established to evaluate the effectiveness of these efforts. The test performs a loop with 100 million calls of the Faddeeva function on a single thread. The total calculation time is recorded for performance evaluation. As shown in Table 5.1, the baseline `ACM680::W`

	Baseline	Float	Strength R.	SIMD	CPUID	SoA	Alignment	MIT
BDW	33.86	30.74	30.73	25.56	14.47	9.82	**8.92**	18.61
KNL	113.14	111.52	110.21	104.25	24.63	9.50	**8.76**	67.58

Table 5.1: Elapsed time (s) of 100,000,000 calls of Faddeeva functions on a single thread with accumulated optimization efforts.

is much slower than `MIT::W`. It can be found that without vectorization, the speedup brought by *double* to *float* conversion is not as significant as assumed before. Another important point is specifying `CPUID` with `processor` clause: in the `SIMD` step, `declare simd` pragma is already disposed to vectorize the function call. However, without explicit `CPUID`, the Intel compiler seems not to handle vectorization with the hardware's latest instruction sets. This small change brings a speedup of 1.77× on BDW and 4.23× on KNL. With the help of AVX-512 [44], one many-core thread can have the same performance as one multi-core thread with much lower frequency. Same efforts have been put

in trying to optimize `MIT::W` as well, but jump instructions inside the code make the function unvectorizable. At last, the optimized `ACM680::W` outperforms `MIT::W` with a gain of 2.09× on BDW and 8.71× on KNL.

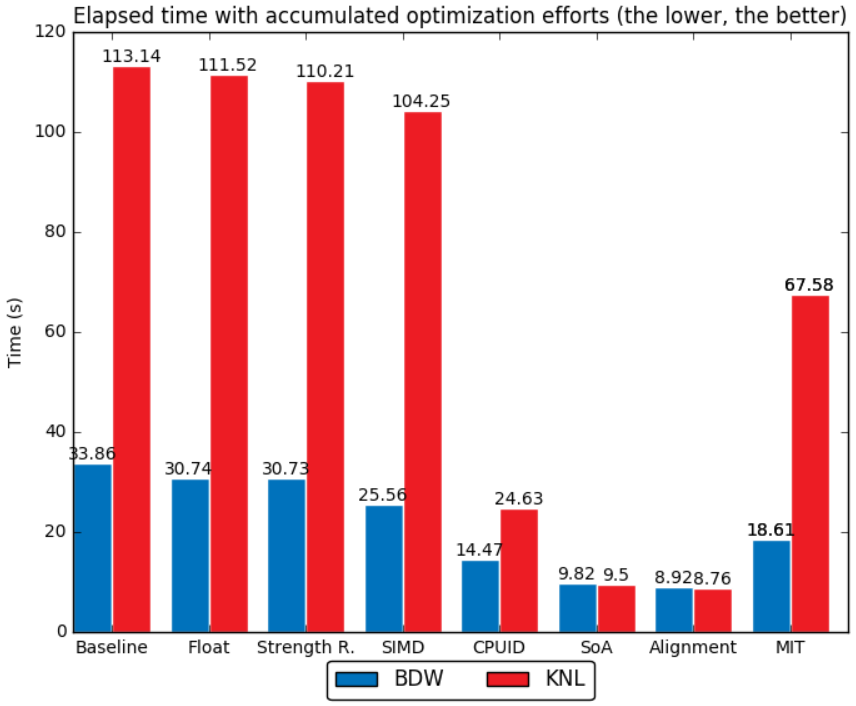


FIGURE 5.2: Performance tests of 100,000,000 calls of Faddeeva function on a single thread.

5.4.1.1 Preliminary Numerical Evaluation

As for numerical accuracy, comparisons of `ACM680::W` and `MIT::W` to some precalculated reference values are shown in Table 5.2. The original `ACM680::W` function has very a high quality compared to `MIT::W`. With the use of strength reduction and SIMD instructions, optimized `ACM680::W` functions lose considerable numerical accuracy. Though the fact that SIMD instructions will cause accuracy loss has been mentioned in the Intel manual [135], the exact numerical difference has been precised nowhere in accessible publications. Our test results show that the relative error by using SIMD instructions is around $1e-7$. With all optimizations taken in account, finally, we preserve two optimized `ACM680::W` implementations for the double and single precision respectively. Compared to the double precision implementation, accuracy loss brought by the single is really small. For MC transport calculations, an absolute difference no more than $1e{-}5$ obtained by the current single precision implementation is acceptable.

Implementations	Average $\Delta_{absolute}$	Max. $\Delta_{absolute}$
`MIT::W`	2.97e-09	1.65e-07
Original "double" `ACM680::W`	7.68e-16	8.68e-15
Optimized "double" `ACM680::W`	2.24e-07	6.19e-06
Optimized "float" `ACM680::W`	1.17e-06	7.34e-06

TABLE 5.2: Accuracy evaluation of Faddeeva implementations.

5.4.2 Reconstruction in PATMOS

Three main tests of the on-the-fly reconstruction are carried out with PATMOS: the first one is a unit-test to explore cross-section efficiency; the second one is the evaluation with all SLBW isotopes and only one MLBW in a simulation; At last, a comparative test between the reconstruction and energy lookups involving all SLBW and MLBW isotopes will be presented.

5.4.2.1 Unit Test of Cross Section Calculation

Before evaluating the reconstruction performance in a real simulation, we create a mini-test to explore the pure cross-section efficiency for the binary search and our on-the-fly method. A parallelized for-loop by using all 272 threads on KNL performs 100,000,000 cross-section calculations. The reason to employ all hardware threads is to have the same execution environment as how the cross section kernel called in a real simulation.

	^{240}`U` (68 resonances)		^{96}`ZR` (30 resonances)		^{240}`PU` (268 resonances)	
	Binary	Reconstruction	Binary	Reconstruction	Binary	Reconstruction
Performance (xs/sec.)	2.52e8	3.70e7	2.51e8	1.36e7	2.52e8	2.55e6
Average $\Delta_{relative}$	-	6.08e-4	-	6.72e-4	-	1.59e-3

TABLE 5.3: Evaluation of cross section calculation rate.

As shown in Table 5.3, the SLBW ^{240}`U` with sixty-eight resonances has the highest calculation rate among three isotopes. The MLBW ^{96}`ZR` with nearly half number of resonances is however, about 2.7× more time-consuming than ^{240}`U`. This proves that the MLBW calculation is more complex than the SLBW. Comparing the two MLBW isotopes: ^{96}`ZR` and ^{240}`PU`, we can easily find that the computation cost of the on-the-fly calculation is proportional to the number of resonances in the nuclide while the same cost is almost constant with the binary search. Therefore, it can be estimated that the on-the-fly calculation will introduce more load-balancing problems than energy lookups. More vectorized reduction operations in MLBW lead to more significant accuracy loss. Generally speaking, the on-the-fly reconstruction itself is about $10 \sim 10^2$ slower than the binary search depending on the number of resonances in the nuclide. For the worst case with ^{238}`U`, a much lower computing rate is expected.

5.4.2.2 Performance in `PointKernel`

The `PointKernel` test case presented in this section is a preliminary simulation with all SLBW isotopes and one MLBW isotope: ^{240}`PU`. All available hardware threads are employed (36 on BDW, 272 on KNL) since using hyper-threading can bring a 25% speedup in this benchmark.

	Baseline	Simplification	SIMD	No-Branching	Loop Splitting	Float	SoA	Alignment	Padding	Scheduling	Lookup
BDW	157.60	6.07	6.12	4.02	4.03	1.78	1.78	1.77	**1.75**	2.13	**0.48**
KNL	250.18	10.63	3.44	2.32	2.30	1.64	1.58	1.55	1.55	**1.51**	**1.26**

TABLE 5.4: Timing performance (s) per batch with accumulated optimizations.

Table 5.4 shows performance change brought by different efforts detailed in Section 5.3. Each optimization step has taken in account all steps before it. It should be noted that every step is totally independent of each other, thus changing orders of optimizations will not affect final efficiency. As shown in the table, KNL is 1.59× slower than BDW with the primitive code. SIMD directives bring direct speedup on KNL, but this is not the case with BDW. Following measures like removing if-branch inside the loop are necessary to help compiler vectorize on BDW. Another interesting point is that the AoS to SoA conversion brings only little speedup in our implementation. This may be due to the fact that our most time-consuming mid-loop already performs on intermediate SoA data. So changing data layout of the trivial pre-loop and post-loop from AoS to SoA does not have a big effect. Moreover, we can also find that dynamic scheduling negatively impacts the performance on BDW due to the already balanced workloads, but it can be anticipated that thread scheduling would be necessary for more complicated problems where much more MLBW isotopes are present. Finally, compared to the baseline code, the optimized implementation obtains a speedup of 74× on BDW and 166× on KNL and for numerical results, a maximum relative error of 10^{-4} is acceptable in MC simulations due to the cross-section uncertainty. A comparison between the conventional binary lookup and the reconstruction shows that the on-the-fly calculation is 3.65× slower on BDW but only 23% less efficient on KNL. These results are very encouraging, if we remember that with table lookup we can only treat a few temperatures, while our reconstruction can deal with an arbitrary number of temperatures.

Figure 5.3 shows the FLOP usage of the most important hotspot in each step of the optimization, the baseline implementation starts from a very low FLOP utilization. Then the situation has been significantly improved with the simplified algorithm. The `SIMD` step represents all vectorization work like vectorizing external function, loop collapse, reduction, etc. It can be observed that KNL recognizes well SIMD potentials indicated by directives and clauses while it is not the case for BDW. Speedups from the `Branching` step show that straight-line instructions are so important for vectorization

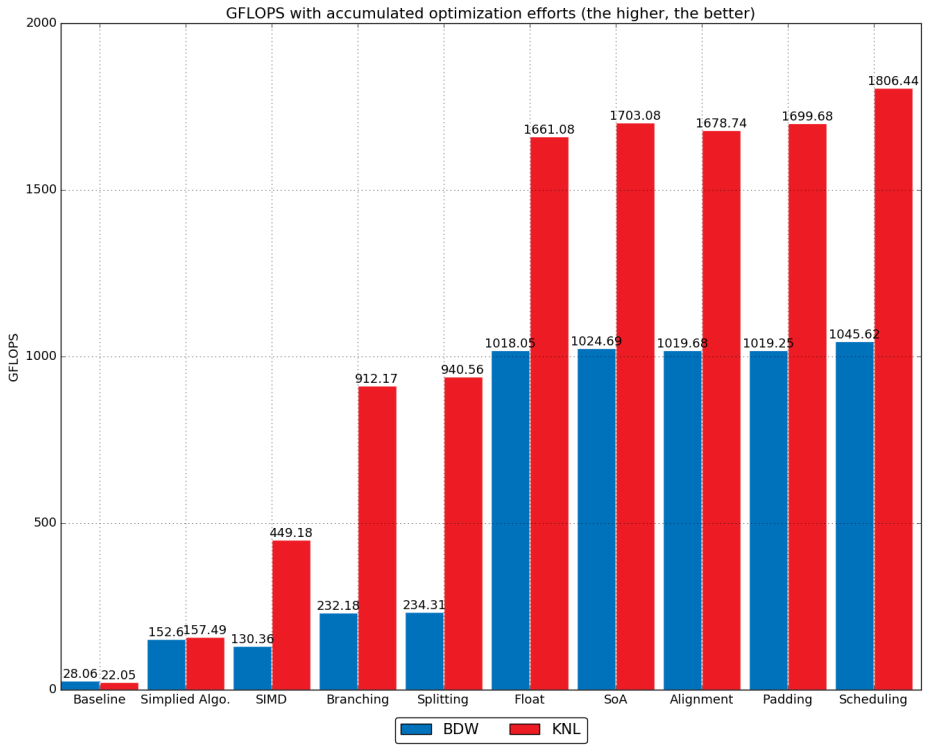


FIGURE 5.3: FLOP usage of the `PointKernel` benchmark on BDW and KNL.

efficiency. BDW can begin to benefit from vectorization only if the if-condition is removed from the loop body. Passing the calculation from *double* to *float* can directly double the FLOP usage. Other useful tunings further improving execution performance bring visible enhancement for FLOPS. After series of optimization work, we succeed to improve the FLOP usage from 28.06 and 22.05 GFLOPS (double precision) to 1045.62 and 1728.44 GFLOPS (single precision) on BDW and KNL, respectively.

5.4.2.3 Memory Requirement

Current cross section reconstruction model in PATMOS is a preliminary implementation of the multipole method. Compared to the real pole representations in the original model, our implementation requires more data as input. Each resonance is an *object* containing variables listed below:

- `ER`: resonance energy
- `J`: spin of resonance state
- `GT`: total width
- `GN`: neutron width

- `GG`: radiation width
- `GF`: fission width
- `GJ`: statistical factor $(2J+1)/[2(2I+1)]$

Besides, each *object* also has three precomputed constants which makes ten variables in total for every resonance (thus 80 bytes in *double* or 40 bytes in *float*). A complete memory footprint comparison is difficult to carry out because PATMOS still uses the binary search outside the resolved resonance region. When preparing data for the simulation, the entire pretabulated cross section tables are fully loaded into memory so there are duplicated data in the current implementation.

5.4.2.4 Roofline Analysis

Figure 5.4 shows Roofline analysis of our naive implementation. Red dots represent kernels (loops or functions) occupying more than 6% of overall computing time (green: less than 1%, yellow: between 1% and 6%). The entire simulation is overall memory-bound. More precisely, two kernels representing 70% of total computation time are limited by the L2 cache bandwidth of `KNL`; another two kernels representing 18% time are limited by the MCDRAM bandwidth. All four kernels have low arithmetic intensity (0.0001 - 0.1) and no vectorization is applied. As a result, optimizations by reducing memory access should be explored; computing-intensive kernels need to be extracted for vectorization.

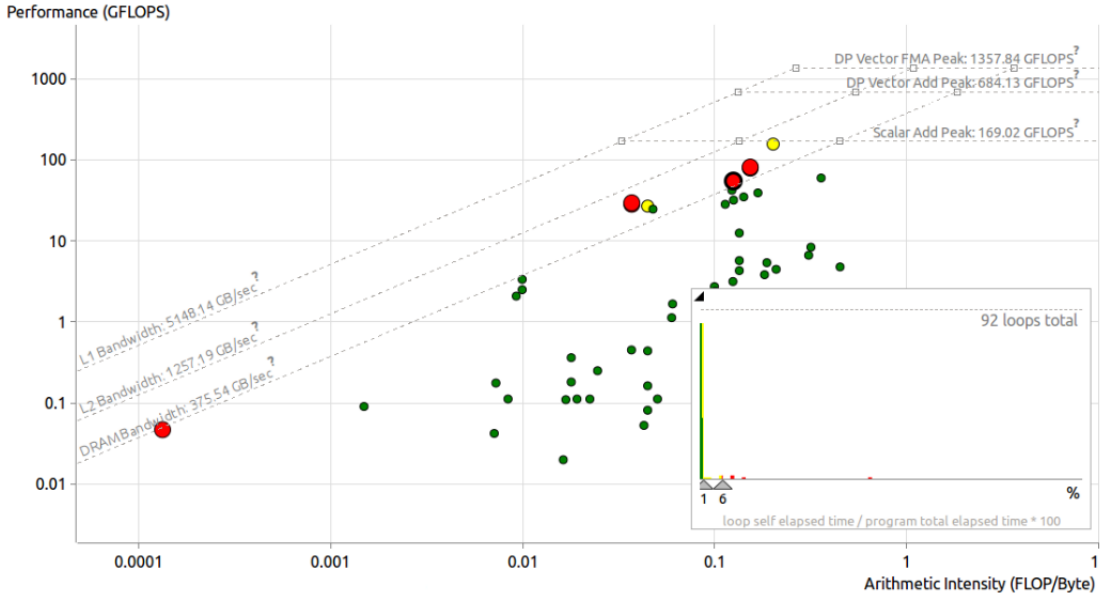


FIGURE 5.4: Roofline Analysis of the naive implementation on KNL.

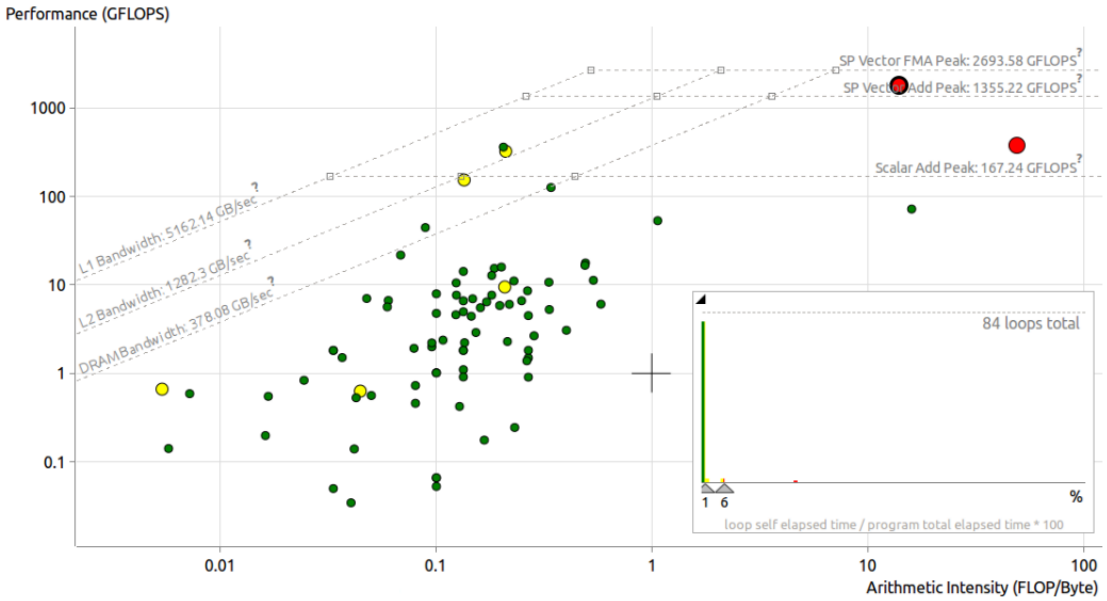


FIGURE 5.5: Roofline Analysis of the optimized implementation on KNL.

Figure 5.5 shows Roofline analysis of the optimized implementation. Two major hotspots (red dots) presented in the chart become both compute-bound. The most time-consuming kernel referring to the loop accumulating G_r and H_r (lines 7-14 of Algo. 10) achieves 1806 GFLOPS with vectorization. Another major hotspot referring to calls of the Faddeeva function has less FLOP usage due to algorithm branching inside the function. The chart indicates that the following work could focus on this branching issue as well as medium kernels (yellow dots) by exploring higher SIMD opportunities.

5.4.3 Energy Lookups vs. On-the-fly Reconstruction

The `PointKernel` test case presented in this section involves all SLBW and MLBW isotopes (267 isotopes in total in ENDF/B-VII.1). The main components of the mixture are ^{1}`H` and ^{240}`PU`. This test case is dedicated to evaluating the reconstruction with other

	Binary	Cascading	HashMaterial	HashIsotope	Unionized	Reconstruction
BDW	6.21	2.38	2.27	3.84	1.45	104.73
KNL (68 thds, flat)	14.14	4.24	3.83	9.35	2.89	89.89
KNL (272 thds, flat)	11.19	4.09	2.98	8.75	2.06	55.75
KNL (272 thds, cache)	15.275	5.08	3.94	9.83	3.28	69.89

TABLE 5.5: Timing performance (s) per batch of cross section computation methods.

energy lookup algorithms in terms of performance. As shown in Table 5.5, the unionized grid significantly outperforms other algorithms. The fractional cascading has nearly the same performance as hashing material on BDW but visibly slower on KNL. The hashing isotope is always less efficient than the two methods performed at material

level (hashing material and fractional cascading). The binary search has the lowest execution performance among all lookup algorithms. It can be observed that the on-the-fly calculation is much slower than all energy lookups. Even compared with the binary search, the reconstruction is 17× slower on BDW and 5× slower on KNL. Except for the reconstruction, all energy lookup algorithms show better timing performance (about 2×) on BDW than on KNL. The 512-bit AVX-512 greatly improves vectorization performance than the 256-bit AVX2 while even with this powerful SIMD support, the current on-the-fly reconstruction is still too time-consuming to use in real simulations. Compared to the result of Table 5.4, the reconstruction shows less efficiency because all MLBW isotopes are involved in the calculation. It should be noted that many voluminous Reich-Moore cross sections have not been considered yet in the current implementation. According to the conclusion of Subsection 5.4.2.1, performing on-the-fly calculations for these isotopes will result in further performance degradation. As a result, non-algorithmic optimizations like vectorization and threading tuning can not remove the performance issue of the on-the-fly reconstruction. New algorithms like the Windowed-Multipole method seems to be necessary for applying the on-the-fly Doppler broadening in real simulations.

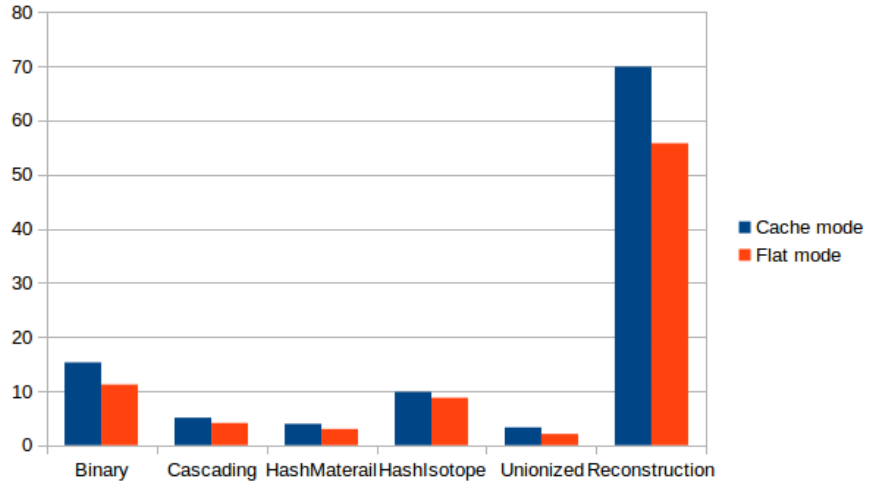


FIGURE 5.6: The performance effect of the memory mode on KNL.

In terms of hardware configuration, using hyper-threading brings a few performance improvement (around 20%) for energy lookups but about 62% for the reconstruction. The thread affinity control can bring a further 5% speedup for using 68 threads on KNL but makes no difference in the case of using all 272 hardware threads. Results shown in Figure 5.6 indicate that all cross-section algorithms benefit from the high bandwidth memory. Since all data involved in the `PointKernel` can fit into the 16 GB MCDRAM, using the *flat* memory mode will always get better execution performance.

Chapter 6

Conclusion and Perspective

6.1 Conclusion

The work presented in this thesis focuses on accelerating Monte Carlo neutron transport calculations with the help of emerging hardware accelerators, especially the Intel Many Integrated Core architecture. The main purpose is to identify and solve issues for MC simulations in terms of threading, vectorization, and memory organization.

Profiling results show that the cross-section computation is the major performance bottleneck of MC calculations. Typically, cross-section data are pre-calculated and stored into memory before simulations for each nuclide, thus during the simulation, only table lookups are required to retrieve data from memory and the compute cost is trivial. The binary search is usually used to locate the target cross-section data in large tables. In fact, this bouncing lookup method makes good sense in terms of the algorithm but is not friendly at all to the memory access pattern of modern computing processors. Random memory accesses result in high cache misses and moreover, there is little potential for vectorization.

In the first part of this thesis, we have investigated and proposed several SIMD-friendly alternatives of the traditional binary search. Then, a large collection of competing algorithms have been evaluated to accelerate the energy lookup process. In unit-tests outside the prototype, we were able to show that the vectorized linear search provides substantial gains on both CPU and MIC architecture. For an array of `double` variables, the SIMD-optimized linear search would be more efficient than the binary search when array size is less than 200. Vectorized N-ary search performs never better than the binary due to heavy discontiguous memory access. The speedup provided by MIC 512-bit VPU is relatively more significant than that of CPU due to the larger VPU

width. By using such optimized search scheme in the simulation, non-trivial additional speedup can be observed over existing solutions. A new SIMD-based lookup algorithm, N-ary map, has been proposed and tested for MIC architecture. It produces an 18% speedup over the conventional binary search with negligible increase in the memory footprint. This algorithm would be more favorable to the future architecture with more powerful vector units. Compared to the vectorization, memory optimization by reducing data duplication brings more acceleration in the simulation. This is due to the fact that all energy lookup methods are in principle a memory-bound problem. Regarding the performance of all tested algorithms, the unionized method is the fastest on both CPU and MIC architecture but at the cost of an order of magnitude increase in the memory footprint. On the other hand, the time-consuming binary search has the best scalability on CPU while on KNC the best is the similar N-ary map. Finally, it can be concluded that the logarithmic material hashing method is a good compromise between performance and memory foot-print on both CPU and MIC and could thus be the best choice for performing energy lookup in MC codes. Otherwise, the unionized grid can have the best performance if one can afford the extra memory footprint (∼6GB per temperature).

In order to solve the problem like restricted memory space and few vectorization of energy lookup methods, we have investigated another on-the-fly Doppler broadening method, which is a direct reconstruction for the cross-sections. This method is a variant of the multipole representation, the basic idea behind which is to do the Doppler broadening computation of cross sections each time a cross-section data is required. This method converts the problem from memory-bound to compute-bound: only several variables for each resonance are required instead of the conventional pointwise table covering the entire resolved resonance region. Such tremendous decrease of memory requirement makes it possible to use MC codes to perform simulations with the thermal-hydraulic feedback where thousands of temperature may be involved in the simulation. The major downside of this method is its considerable execution time. Previous studies [136] indicate that it can even be two orders of magnitude slower than the inefficient binary search.

Our naive reconstruction implementation shows a very low execution efficiency. Using this method in a simulation is about 200× slower than the binary search on both CPU and MIC. The FLOP usage is also extremely far from the peak even the computing kernel should be completely CPU-bound. A series of optimization efforts have been applied to improve this situation, such as loop splitting, data alignment, removing algorithm branching, and so on. Implementations and optimizations are then evaluated by some unit tests and a small benchmark in PATMOS on both multi-core and many-core systems. Through our progressive optimizations with scalar tuning, vector processing, and parallel adjustment, we found that this algorithm has high optimization

potential and offers abundant vectorization opportunities. Finally, the major computing kernel achieves 67% of Knights Landing's effective peak performance (1,806 GFLOPS / 2,693 GFLOPS in single precision). Compared to the classical memory-bound algorithm, important hotspots of the reconstruction all become limited by the capability of floating-point operations.

In this thesis, we succeeded to demonstrate the possibility to transform a problem from memory-bound to compute-bound in order to achieve higher hardware utilization on both CPU and MIC architectures. We proved that the vectorization during this process can bring significant performance enhancement. Even this transformation can not be completely applied to all kinds of algorithms, it is certainly worth investigating to explore alternative solutions for problems facing the *exascale computing* in which there will be more and more simplified computing cores with less enhancement on memory.

6.2 Future Work

The initial goals of the thesis have been completed, performance bottlenecks have been identified and corresponding optimizations are evaluated on modern computing processors, but there are still a lot of problems that should be further explored for the current work.

The current PATMOS code organizes the thread-task according to the available number of cores of the architecture. This leads to a fact that workload of a KNL thread is not the same as a BDW thread. Every time the platform is changed, series of tuning work is necessary since the content of each software thread is completely different. In order to have a better scaling on various computing architectures, the task-parallel model should be used to replace the current thread-parallel one.

The polymorphism is widely used in PATMOS to implement competing algorithms with same inputs and outputs. During the implementation of different energy lookup methods, however, each method has its own input and output requirements so the interface can be no longer uniform. A redesign of the code structure seems to be necessary if these algorithms will be preserved in PATMOS. Another problem is the function overhead because calling a polymorphism function usually takes much more time than a common function. For small polymorphism functions with only several instructions (for example, computing the *l-value*-depended penetration and shift factor with the on-the-fly reconstruction), the call overhead might become a serious performance issue.

The study of energy lookup algorithms has been well developed since a great deal of work has been reported to deal with the issue. There are actually few problems worthing further investigation. Many alternatives of the binary search can be applied to Monte Carlo calculations for higher lookup efficiency, but none of them could remove the memory-bound nature of table lookup. Though algorithms are already fixed, continuous re-architecturing or tuning work is certainly necessary to maintain the code efficiency. One former paper [92] shows that data prefetching brings considerable performance speedup. This simple work not evaluated in PATMOS is worth testing.

As for the cross-section reconstruction, this on-the-fly Doppler broadening still shows poor execution performance compared to the conventional binary lookup (5× slower in a simplified simulation). Other Faddeeva algorithms [137, 138] can be evaluated for the vector processing. Removing jump instructions inside the `MIT::W` could make the implementation vectorizable, but other factors like pre-calculated lookup tables, large function body, and so many if-conditions inside the function will greatly degrade vectorization efficiency. Further performance improvement is relatively limited with non-algorithmic optimizations thus new algorithms like the windowed-multipole is worth testing. Numerical validation of the on-the-fly reconstruction is crucial for the future work especially in order to decide whether it is possible to lower the precision for obtaining higher vectorization efficiency.

Preliminary GPGPU investigations have been started in this work. The idea is to perform shared-memory parallelism on the host CPU and offload the costly reconstruction kernel to remote devices. Since the kernel is completely compute-bound, porting it on the high FLOP-capability GPGPU should get a significant speedup. Programming models like StarPU [139], Kokkos [140] and OpenACC should be evaluated instead of the low-level CUDA library.

Bibliography

[1] Kenneth S Krane and David Halliday. *Introductory nuclear physics*, volume 465. Wiley New York, 1988.

[2] Duncan Burn. *Nuclear power and the energy crisis: politics and the atomic industry.* Springer, 1978.

[3] Intergovernmental Panel on Climate Change. *Climate Change 2014: Mitigation of Climate Change*, volume 3. Cambridge University Press, 2015.

[4] *Nuclear Power in France.* World Nuclear Association, 2017. URL `http://www.world-nuclear.org/information-library/country-profiles/countries-a-f/france.aspx`.

[5] Lyndon Evans and Philip Bryant. Lhc machine. *Journal of instrumentation*, 3 (08):S08001, 2008.

[6] Sea Agostinelli, John Allison, K al Amako, J Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G Barrand, et al. Geant4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[7] E Brun, F Damian, CM Diop, E Dumonteil, FX Hugot, C Jouanne, YK Lee, F Malvagi, A Mazzolo, O Petit, et al. Tripoli-4®, cea, edf and areva reference monte carlo code. *Annals of Nuclear Energy*, 82:151–160, 2015.

[8] Forrest B Brown et al. Mcnp–a general monte carlo n-particle transport code, version 5. *Los Alamos National Laboratory, Oak Ridge, TN*, 2003.

[9] David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[10] Sally Falk Moore. Law and social change: the semi-autonomous social field as an appropriate subject of study. *Law & Society Review*, 7(4):719–746, 1973.

[11] Alejandro Duran and Michael Klemm. The intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE, 2012.

[12] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[13] Michael Feldman. TOP500.org. URL `https://www.top500.org/news/tianhe-2-supercomputer-being-upgraded-to-95-petaflops/`.

[14] Takumi Maruyama, Toshio Yoshida, Ryuji Kan, Iwao Yamazaki, Shuji Yamamura, Noriyuki Takahashi, Mikio Hondou, and Hiroshi Okano. Sparc64 viiifx: A new-generation octocore processor for petascale computing. *IEEE micro*, 30(2), 2010.

[15] *The 48th edition of the TOP500 list.* TOP500, 2016. URL `https://www.top500.org/lists/2016/11/`.

[16] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162. ACM, 2004.

[17] *The Exascale Computing Project Awards $ 39.8M to 22 Projects.* Tiffany Trader, September 7, 2016. URL `https://www.hpcwire.com/2016/09/07/exascale-computing-project-awards-39-8m-22-projects/`.

[18] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[19] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[20] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pages 1–77, 2010.

[21] Pete Beckman. Looking toward exascale computing. In *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 3–3. IEEE, 2008.

[22] Forrest B Brown. Fundamentals of monte carlo particle transport. *Los Alamos National Laboratory, LA-UR-05-4983*, 2005.

[23] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[24] *Scientific Discovery through Advanced Computing (SciDAC) — Co-Design*. Office of Science, U.S. Department of Energy, 2016. URL `https://science.energy.gov/ascr/research/scidac/co-design`.

[25] Yunsong Wang, Emeric Brun, Fausto Malvagi, and Christophe Calvin. Competing energy lookup algorithms in monte carlo neutron transport calculations and their optimization on cpu and intel mic architectures. *Journal of Computational Science*, 20:94–102, 2017.

[26] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[27] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[28] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266. IEEE, 2011.

[29] Ankireddy Nalamalpu, Nasser Kurd, Anant Deval, Chris Mozak, Jonathan Douglas, Ashish Khanna, Fabrice Paillet, Gerhard Schrom, and Boyd Phelps. Broadwell: A family of ia 14nm processors. In *VLSI Circuits (VLSI Circuits), 2015 Symposium on*, pages C314–C315. IEEE, 2015.

[30] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[31] George Chrysos. Intel® xeon phi coprocessor (codename knights corner). In *Hot Chips 24 Symposium (HCS), 2012 IEEE*, pages 1–31. IEEE, 2012.

[32] Charlie Demerjian. Intel details knights corner architecture at long last. *URL:[http://semiaccurate.com/2012/08/28/intel-details-knights-corner-architecture-at-long-last/]*, 2012.

[33] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer, 2001.

[34] George Chrysos. Intel® xeon phi^{TM} coprocessor-the architecture. *Intel Whitepaper*, 176, 2014.

[35] Rezaur Rahman. Intel xeon phi coprocessor vector microarchitecture. *URL:[http://software. intel. com/en-us/articles/intel-xeon-phi-coprocessorvector-microarchitecture]*, 2014.

[36] Intel website. Intel powers the world's fastest supercomputer, reveals new and future high performance computing technologies. *URL:[https://newsroom.intel.com/news-releases/intel-powers-the-worlds-fastest-supercomputer-reveals-new-and-future-high-performance-computing-technologies/]*, 2013.

[37] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[38] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[39] David Kanter. Silvermont, intel's low power architecture. *RWT*, 2013.

[40] An Intel. Introduction to the intel quickpath interconnect. *White Paper*, 2009.

[41] Andrey Vladimirov and Ryo Asai. Clustering modes in knights landing processors: developer's guide. Technical report, Tech. rep., Colfax International, 2016.

[42] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.

[43] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[44] James Reinders. Avx-512 instructions. *Intel Corporation*, 2013.

[45] CUDA Nvidia. Programming guide, 2017.

[46] CUDA Nvidia. Nvidia tesla p100 – the most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. *Whitepaper*, 2016.

[47] Denis Foley. Nvlink, pascal and stacked memory: Feeding the appetite for big data. *Nvidia. com*, 2014.

[48] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1): 46–55, 1998.

[49] ARB OpenMP. Openmp application program interface, v. 3.0. *OpenMP Architecture Review Board*, 2008.

[50] ARB OpenMP. Openmp application program interface version 4.0, 2013.

[51] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44 (4):101–110, 2009.

[52] Chris J Newburn, Serguei Dmitriev, Ravi Narayanaswamy, John Wiegert, Ravi Murty, Francisco Chinchilla, Rajiv Deodhar, and Russ McGuire. Offload compiler runtime for the intel® xeon phi coprocessor. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225. IEEE, 2013.

[53] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[54] 2011-2017 OpenACC-standard.org. *OpenACC - More Sicence, Less Programming*, 2017. `https://www.openacc.org/`.

[55] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.

[56] M Deilmann. A guide to vectorization with intel c++ compilers. *Intel Corporation, April*, 2012.

[57] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi^{TM}*, pages 167–188. Springer, 2014.

[58] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition.* Morgan Kaufmann, 2016.

[59] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[60] Intel Corporation. *Intel Intrinsics Guide*, 2017. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[61] Arch D Robison. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18:25, 2012.

[62] Matthias Kretz and Volker Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[63] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 1–8. ACM, 2014.

[64] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[65] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[66] Bernd Mohr, Darryl Brown, and Allen Malony. Tau: A portable parallel program analysis environment for pc++. *Parallel Processing: CONPAR 94—VAPP VI*, pages 29–40, 1994.

[67] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2): 287–311, 2006.

[68] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[69] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[70] Martyn Corden. *Getting the Most out of your Intel® Compiler with the New Optimization Reports*, 2014. `https://software.intel.com/en-us/articles/getting-the-most-out-of-your-intel-compiler-with-the-new-optimization-reports`.

[71] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[72] TS Koskela and M Lobet. Roofline analysis in the intel® advisor to deliver optimized performance for applications on intel® xeon phi™ processor. 2017.

[73] Ryan M Bergmann and Jasmina L Vujić. Algorithmic choices in warp–a framework for continuous energy monte carlo neutron transport in general 3d geometries on gpus. *Annals of Nuclear Energy*, 77:176–193, 2015.

[74] N Soppera, M Bossant, and E Dupont. Janis 4: An improved version of the nea java-based nuclear data information system. *Nuclear Data Sheets*, 120:294–296, 2014.

[75] MB Chadwick, M Herman, P Obložinskỳ, Michael E Dunn, Y Danon, AC Kahler, Donald L Smith, B Pritychenko, Goran Arbanas, R Arcilla, et al. Endf/b-vii. 1 nuclear data for science and technology: cross sections, covariances, fission product yields and decay data. *Nuclear Data Sheets*, 112(12):2887–2996, 2011.

[76] Arjan Koning, Robin Forrest, Mark Kellett, Robert Mills, Hans Henriksson, Yolanda Rugama, et al. *The JEFF-3.1 nuclear data library*. OECD, 2006.

[77] Willis E Lamb Jr. Capture of neutrons by atoms in a crystal. *Physical Review*, 55 (2):190, 1939.

[78] George I Bell and Samuel Glasstone. Nuclear reactor theory. Technical report, Division of Technical Information, US Atomic Energy Commission, 1970.

[79] Dermott E Cullen and Charles R Weisbin. Exact doppler broadening of tabulated cross sections. *Nuclear Science and Engineering*, 60(3):199–229, 1976.

[80] Robert Macfarlane, Douglas W Muir, RM Boicourt, Albert Comstock Kahler III, and Jeremy Lloyd Conlin. The njoy nuclear data processing system, version 2016. Technical report, Los Alamos National Laboratory (LANL), 2017.

[81] Ryan Bergmann. *The Development of WARP-A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs*. PhD thesis, University of California, Berkeley, 2014.

[82] Tim Goorley. Mcnp6. 1.1-beta release notes. *Los Alamos National Laboratory Technical Report*, 2014.

[83] T Vergnaud. Overview of monte carlo methodologies in tripoli-3. In *Advanced Monte Carlo computer programs for radiation transport*. 1995.

[84] JC Nimal and T Vergnaud. Tripoli: a general monte carlo code, present state and future prospects. *Progress in Nuclear Energy*, 24(1-3):195–200, 1990.

[85] JP Both and Y Peneliau. The monte-carlo code tripoli-4 and its first benchmark interpretations. 1996.

[86] JP Both, YK Lee, A Mazzolo, Y Pénéliau, O Petit, and B Roesslinger. Tripoli-4: Code de transport monte carlo fonctionnalites et applications. In *Proceedings of 8th International Conference on Radiation Shielding (ICRS),(Arlington, Texas)*, page 373, 1994.

[87] Emeric Brun, Stéphane Chauveau, and Fausto Malvagi. Patmos: A prototype monte carlo transport code to test high performance architectures. In *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea*, 2017.

[88] ISO/IEC 14882:2014 – Information technology – Programming languages – C++ (2014). 2014.

[89] Ilka Antcheva, Maarten Ballintijn, Bertrand Bellenot, Marek Biskup, Rene Brun, Nenad Buncic, Ph Canal, Diego Casadei, Olivier Couet, Valery Fine, et al. Root—a c++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 182(6):1384–1385, 2011.

[90] J Eduard Hoogenboom and William R Martin. A proposal for a benchmark to monitor the performance of detailed monte carlo calculation of power densities in a full size reactor core. 2009.

[91] Andrew R Siegel, Kord Smith, Paul K Romano, Benoit Forget, and Kyle G Felker. Multi-core performance studies of a monte carlo neutron transport code. *The International Journal of High Performance Computing Applications*, 28(1):87–96, 2014.

[92] Tianyu Liu, Noah Wolfe, Christopher D Carothers, Wei Ji, and X George Xu. Optimizing the monte carlo neutron cross-section construction code xsbench for mic and gpu platforms. *Nuclear Science and Engineering*, 185(1):232–242, 2017.

[93] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[94] David Ozog, Allen D Malony, and Andrew R Siegel. A performance analysis of simd algorithms for monte carlo simulations of nuclear reactor cores. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 733–742. IEEE, 2015.

[95] Forrest B Brown and William R Martin. Monte carlo methods for radiation transport analysis on vector computers. *Progress in Nuclear Energy*, 14(3):269–299, 1984.

[96] Paul K Romano and Andrew R Siegel. Limits on the efficiency of event-based algorithms for monte carlo neutron transport. *Nuclear Engineering and Technology*, 2017.

[97] Steven P Hamilton, Thomas M Evans, and Stuart R Slattery. Gpu acceleration of history-based multigroup monte carlo. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 2016.

[98] John Apostolakis, René Brun, Federico Carminati, Andrei Gheata, and Sandro Wenzel. A concurrent vector-based steering framework for particle transport. In *Journal of Physics: Conference Series*, volume 523, page 012004. IOP Publishing, 2014.

[99] Xining Du, Tianyu Liu, Wei Ji, X George Xu, and Forrest B Brown. Evaluation of vectorized monte carlo algorithms on gpus for a neutron eigenvalue problem. In *Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013), Sun Valley, Idaho, USA*, pages 2513–2522, 2013.

[100] Tianyu Liu, Xining Du, Wei Ji, X George Xu, and Forrest B Brown. A comparative study of history-based versus vectorized monte carlo methods in the gpu/cuda environment for a simple neutron eigenvalue problem. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04206. EDP Sciences, 2014.

[101] X George Xu, Tianyu Liu, Lin Su, Xining Du, Matthew Riblett, Wei Ji, Deyang Gu, Christopher D Carothers, Mark S Shephard, Forrest B Brown, et al. Archer, a new monte carlo software tool for emerging heterogeneous computing environments. *Annals of Nuclear Energy*, 82:2–9, 2015.

[102] Kan Wang, Zeguang Li, Ding She, Qi Xu, Yishu Qiu, Jiankai Yu, Jialong Sun, Xiao Fan, Ganglin Yu, et al. Rmc–a monte carlo code for reactor core analysis. *Annals of Nuclear Energy*, 82:121–129, 2015.

[103] Qi Xu, Ganglin Yu, and Kan Wang. Research on gpu acceleration for monte carlo criticality calculation. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04210. EDP Sciences, 2014.

[104] Forrest B Brown. New hash-based energy lookup algorithm for monte carlo codes. *Trans. Am. Nucl. Soc*, 111:659–662, 2014.

[105] J. Leppänen. Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation. *Annals of Nuclear Energy*, 36(7):878–885, 2009.

[106] Amanda L Lund, Andrew R Siegel, Benoit Forget, Colin Josey, and Paul K Romano. Using fractional cascading to accelerate cross section lookups in monte carlo neutron transport calculations. 2015.

[107] J.A. Walsh et al. Optimizations of the energy grid search algorithm in continuous-energy Monte Carlo particle transport codes. *Computer Physics Communications*, 196:134–142, 2015.

[108] Jaakko Leppänen, Maria Pusa, Tuomas Viitanen, Ville Valtavirta, and Toni Kaltiaisenaho. The serpent monte carlo code: Status, development and applications in 2013. *Annals of Nuclear Energy*, 82:142–150, 2015.

[109] VN Ogibin and AI Orlov. Majorized cross-section method for tracking neutrons in moving media. *Voprosy Atomnoj Nauki i Techniki, Metodoki i Programmy*, 2 (16):6–9, 1984.

[110] Gokhan Yesilyurt, William R Martin, and Forrest B Brown. On-the-fly doppler broadening for monte carlo codes. *Nuclear Science and Engineering*, 171(3):239–257, 2012.

[111] Richard N Hwang. A rigorous pole representation of multilevel cross sections and its practical applications. *Nuclear Science and Engineering*, 96(3):192–209, 1987.

[112] Colin Josey, Pablo Ducru, Benoit Forget, and Kord Smith. Windowed multipole for cross section doppler broadening. *Journal of Computational Physics*, 307:715–727, 2016.

[113] Jonathan A Walsh, Benoit Forget, Kord S Smith, and Forrest B Brown. On-the-fly doppler broadening of unresolved resonance region cross sections. *Progress in Nuclear Energy*, 2017.

[114] John R Tramm and Andrew R Siegel. Memory bottlenecks and memory contention in multi-core monte carlo transport codes. *Annals of Nuclear Energy*, 82:195–202, 2015.

[115] Andrew Siegel, Kord Smith, K Felker, P Romano, Benoit Forget, and P Beckman. Improved cache performance in monte carlo transport calculations using energy banding. *Computer Physics Communications*, 185(4):1195–1199, 2014.

[116] MB Chadwick, P Obložinskỳ, M Herman, NM Greene, RD McKnight, DL Smith, PG Young, RE MacFarlane, GM Hale, SC Frankle, et al. Endf/b-vii. 0: next generation evaluated nuclear data library for nuclear science and technology. *Nuclear data sheets*, 107(12):2931–3060, 2006.

[117] Massimo Bernaschi, Mauro Bisson, and Francesco Salvadore. Multi-kepler gpu vs. multi-intel mic for spin systems simulations. *Computer Physics Communications*, 185(10):2495–2503, 2014.

[118] M. Probst. *Linear vs Binary Search*, 2010. `https://schani.wordpress.com/tag/c-optimization-linear-binary-search-sse2-simd`.

[119] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, 1986.

[120] James Andrew Storer. *An introduction to data structures and algorithms*. Springer Science & Business Media, 2012.

[121] François-Xavier Hugot. personal communication.

[122] Dermott E Cullen. Prepro 2007: 2007 endf/b pre-processing codes. *Rap. Tech. IAEA-NDS-39 Rev*, 13, 2004.

[123] *What is Compound Nucleus and what is Resonance?* Nuclear Power, 2017. URL `http://www.nuclear-power.net/nuclear-power/reactor-physics/nuclear-engineering-fundamentals/neutron-nuclear-reactions/compound-nucleus-reactions/what-is-nuclear-resonance-compound-nucleus/#prettyPhoto`.

[124] A Trkov, M Herman, DA Brown, et al. Endf-6 formats manual. *Data Formats and Procedures for the Evaluated Nuclear Data Files ENDF/B-VI and ENDF/B-VII, National Nuclear Data Center Brookhaven National Laboratory, Upton, NY*, pages 11973–5000, 2012.

[125] ME Dunn and NM Greene. Polident: A module for generating continuous-energy cross sections from endf resonance data. Technical report, Oak Ridge National Lab., TN (US), 2000.

[126] C Jammes and RN Hwang. Conversion of single-and multilevel breit-wigner resonance parameters to pole representation parameters. *Nuclear science and engineering*, 134(1):37–49, 2000.

[127] Vera Nikolaevna Faddeeva, Nikolai Mikhailovich Terent'ev, Vladimir Aleksandrovic Fok, and DG Fry. Tables of values of the function w (z)= e for complex argument. 1961.

[128] GPM Poppe and Christianus MJ Wijers. Algorithm 680: evaluation of the complex error function. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):47, 1990.

[129] Walter Gautschi. Efficient computation of the complex error function. *SIAM Journal on Numerical Analysis*, 7(1):187–198, 1970.

[130] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. Courier Corporation, 1964.

[131] *Faddeeva Package*. Massachusetts Institute of Technology, 2017. URL `http://ab-initio.mit.edu/wiki/index.php/FaddeevaPackage`.

[132] Mofreh R Zaghloul and Ahmed N Ali. Algorithm 916: computing the faddeyeva and voigt functions. *ACM Transactions on Mathematical Software (TOMS)*, 38 (2):15, 2011.

[133] Colfax International. *HOW Series Training on Performance Optimization*, 2017. `https://colfaxresearch.com/category/training/how/`.

[134] Intel Corporation. *Intel PROCESSOR Clause*, 2017. `https://software.intel.com/en-us/node/679659`.

[135] Martyn J Corden and David Kreitzer. Consistency of floating-point results using the intel® compiler. *Intel Corporation*, 2012.

[136] Colin Josey. *Windowed multipole: an efficient Doppler broadening technique for Monte Carlo*. PhD thesis, Massachusetts Institute of Technology, 2015.

[137] J Andre C Weideman. Computation of the complex error function. *SIAM Journal on Numerical Analysis*, 31(5):1497–1518, 1994.

[138] SM Abrarov and Brendan M Quine. Efficient algorithmic implementation of the voigt/complex error function based on exponential series approximation. *Applied Mathematics and Computation*, 218(5):1894–1902, 2011.

[139] 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*, 23(2): 187–198, 2011.

[140] H Carter Edwards and Christian R Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24. IEEE, 2013.

université PARIS-SACLAY

ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre : Optimisation du code Monte Carlo neutronique à l'aide d'accélérateurs de calculs

Mots-clés : Monte Carlo, transport de neutron, section efficace, algorithme de recherche, élargis-sement Doppler, programmation parallèle, vectorisation

Résumé : Les méthodes Monte Carlo (MC), très coûteuses en termes de temps de calcul, sont largement utilisées pour résoudre les problèmes de neutronique avec des approximations minimales. Le calcul des sections efficaces a été identifié comme le goulot d'étranglement principal pour les codes de MC neutronique. L'approche conventionnelle consiste à pré-calculer avant la simulation les sections efficaces. Ces données sont ensuite chargées en mémoire. Pendant le calcul, elles sont récupérées dans des tables et le caractère stochastique du transport MC induit des accès à la mémoire aléatoires. Afin de minimiser les conflits d'accès mémoire, nous avons étudié et optimisé une vaste collection d'algorithmes de recherche afin d'accélérer ce processus de récupération de données. Les résultats montrent qu'une accélération significative peut être obtenue sur des architectures modernes. Cependant, toutes ces solutions de recherche sont redoutablement inefficaces du fait de la saturation de la mémoire et du manque de vectorisation. Un autre problème majeur de ces méthodes est l'empreinte mémoire très importante pour les cas complexes. Afin de résoudre ce problème, nous avons étudié la reconstruction au vol des sections efficaces. Cet algorithme, très calculatoire, permet de passer d'un problème de type "memory-bound" à un problème de type "compute-bound" : l'espace mémoire est ainsi largement réduit. Après une série d'optimisations, les résultats montrent que le noyau de reconstruction bénéficie de la vectorisation et peut atteindre une utilisation élevée à la fois sur CPU classique et architecture many-cœurs.

Title: Optimization of Monte Carlo neutron transport simulations by using emerging architectures

Keywords: Monte Carlo, neutron transport, cross-section, energy lookup, Doppler broadening, cross-section reconstruction, parallel computing, vectorization

Abstract: Monte Carlo (MC) neutron transport simulations are widely used to perform reference calculations with minimal approximations. This method is time-consuming due to the law of large numbers. The cross-section computation has been identified as the major performance bottleneck for MC calculations. Typically, cross-section data are pre-calculated and stored into memory before simulations, thus during the simulation, only table lookups are required to retrieve data from memory and the compute cost is low. In the first part of the work, a large collection of lookup algorithms have been implemented and optimized to accelerate the data retrieving process. Results show that significant speedup can be achieved on modern architectures. However, since these energy lookup algorithms are naturally memory-bound, performance enhancement brought by vectorization is negligible. Moreover, considerable memory requirement along with energy lookups makes more complex problems unaffordable with current architectures. An on-the-fly cross-section reconstruction was then investigated to improve the situation. This method requires the minimum elementary data, and performs cross-section reconstruction and temperature treatment only when needed. It converts the problem from memory-bound to compute-bound: memory space is largely reduced, but the calculation is computationally expensive. After a series of optimizations, results show that the reconstruction kernel benefits well from vectorization and can achieve high hardware utilization on both multi-core and many-core systems.