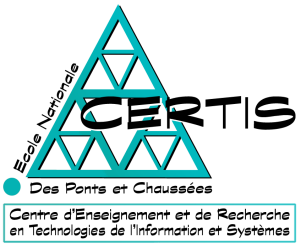
Quelques applications de la programmation des processeurs graphiques à la simulation neuronale et à la vision par ordinateur

Alexandre Chariot



Ecole des Ponts
Certis

Thèse

présentée pour l'obtention du grade de Docteur de l'Ecole des Ponts,
spécialité « Mathématiques et Informatique »

par

Alexandre Chariot

Quelques Applications de la Programmation des Processeurs Graphiques à la Simulation Neuronale et à la Vision par Ordinateur

Thèse soutenue le 16 décembre 2008 devant le jury composé de :

M.	Bernard Lapeyre	Professeur à l'Ecole des Ponts	(Président)
M.	Michel Barlaud	Professeur à l'Université de Nice-Sophia Antipolis	(Rapporteur)
M.	Stéphane Vialle	Professeur à Supélec	(Rapporteur)
M.	Mohamed Akil	Professeur à l'Ecole Supérieure d'Ingénieurs en Electronique et Electrotechnique	(Examinateur)
M.	Romain Brette	Maître de conférences à l'Ecole Normale Supérieure de Paris	(Examinateur)
M.	Luc Robert	Directeur technique, société Autodesk RealViz	(Examinateur invité)
M.	Renaud Keriven	Professeur à l'Ecole des Ponts	(Directeur)

A Papi et Mamie,
A Papa et Maman,
Et à Sirven

Titre Quelques Applications de la Programmation des Processeurs Graphiques à la Simulation Neuronale et à la Vision par Ordinateur

Résumé Largement poussés par l'industrie vidéoludique, la recherche et le développement d'outils matériels destinés à la génération d'images de synthèse, tels les cartes graphiques (ou GPU, *Graphics Processing Units*), ont connu un essor formidable ces dernières années.
L'augmentation de puissance et de flexibilité ainsi que le faible prix de ces GPU ont eu comme conséquence inattendue leur utilisation dans des domaines autres que graphiques. Cet usage détourné est nommé GPGPU, *General Purpose computation on GPU*, ou Programmation Générique sur GPU. Motivés par les besoins computationnels considérables liés aux deux domaines de recherche s'inscrivant dans les thématiques du Certis que sont les neurosciences et la vision par ordinateur, nous proposons dans cette thèse d'appliquer les concepts GPGPU à des applications spécifiques de ces domaines.
Premièrement, les idées clés de la programmation des processeurs graphiques et de leur dérivation sont exposées, puis nous présentons la diversité des applications possibles à travers un large panel de travaux existants.
Dans une deuxième partie, nous présentons un réseau de neurones impulsionnels simulé grâce au GPU et accéléré jusqu'à 18 fois par rapport à une implémentation CPU équivalente.
Dans une troisième partie, nous exposons une méthode variationnelle de reconstruction 3D par stéréovision dense, adaptée sur GPU, permettant de reconstruire précisément une carte de profondeur, à cadence vidéo.
Enfin, nous proposons une méthode d'appariements d'images sur GPU par mise en correspondance de points d'intérêts. A partir de ce procédé, nous avons développé une application apportant un soutien logistique lors de la capture photographique d'une scène large, par exemple à des fins de reconstruction 3D.

Mots-clés GPU, GPGPU, Programmation parallèle, Réseaux de neurones, Vision par Ordinateur, Stéréovision, Points d'intérêt, Mise en correspondance

Title Some Applications of Graphics Processors Programming to Neural Simulation and Computer Vision

Abstract Widely driven by the gaming industry, research and development of new hardware graphics equipments for the generation of images, such as graphics cards (or GPU, *Graphics Processing Unit*), have significantly risen these recent years.
The increased capacities and flexibility and low prices of these GPU had the unintended consequence of their use in areas other than graphics. This hijacking is named GPGPU, (*General Purpose computation on GPU*). Motivated by the computational requirements associated with two research areas within the CERTIS thematics (neurosciences and computer vision), we propose in this thesis to apply the GPGPU concepts to specific applications of these areas.
First, the key ideas of the graphics processors programming and their hijacking are exposed, then we present the diversity of possible applications across a wide range of existing work.
In a second part, we present a spiking neural network simulated through the GPU.
In a third part, we set out a variational method for 3D dense stereo reconstruction adapted on the GPU, to precisely reconstruct a depth map and in almost real time, up to video rate. Finally, we propose an image matching setup using GPU to match hundreds of images interactively. From this process we have developed an application providing logistical support during the photographic capture of a large scene, for example, for 3D reconstruction.

Keywords GPU, GPGPU, Parallel programming, Neural networks, Computer vision, Stereovision, Features, Matching

Remerciements

L'écriture de la page de remerciements n'est jamais l'exercice le plus facile au cours de la rédaction d'un rapport de thèse. Il faut s'assurer de n'oublier personne et de restituer au plus juste les apports et soutiens de chacun. D'avance, je remercie tous les exclus involontaires de cette liste.

En premier lieu, je tiens à adresser mon entière reconnaissance à mon directeur de thèse, Renaud Keriven, pour la confiance qu'il m'a accordé en m'accueillant au sein du Certis et par la suite, pour son indéfectible soutien pendant ces trois années ainsi que la motivation qu'il a su m'insuffler. Quelle que soit la situation, ses conseils avisés m'ont toujours permis d'avancer clairement dans mes travaux (et dans mes découvertes musicales).

Je tiens à présenter mes remerciements chaleureux aux personnes qui m'ont fait l'honneur d'être présentes le jour de ma soutenance en tant que membres du jury, en particulier à Michel Barlaud et Stéphane Vialle qui ont consenti à être rapporteurs de ce manuscrit, en dépit de la charge de travail que cela représente, à Bernard Lapeyre, président de ce jury, ainsi qu'à Mohamed Akil, Romain Brette et Luc Robert, examinateurs lors du jury de soutenance.

Je remercie spécialement l'Ecole des Ponts et l'Université Paris-Est pour avoir assuré l'organisation administrative et logistique de ma thèse (en particulier Alice Tran pour sa compétence et sa bonne humeur permanente), ainsi que le Rectorat de Paris, m'ayant fourni la bourse nécessaire au déroulement de la thèse.

Mes sincères remerciements vont aux membres permanents du Certis avec qui j'ai eu l'occasion de travailler ou que j'ai pu croiser lors de ces trois dernières années : tout d'abord à Brigitte Mondou pour sa grande efficacité et son incroyable patience face à nos besoins administratifs toujours plus originaux, à Jean-Yves Audibert, Florent Ségonne, Jean-Philippe Pons, Arnak Dalalyan et Pascal Monasse pour tous les échanges scientifiques fructueux que j'ai pu avoir avec eux, pour leurs compétences et leur gentillesse.

Un grand merci à tous les autres membres du Certis pour tout ce qu'ils m'ont apporté, aussi bien scientifiquement et humainement qu'à tout autre niveau : Anne-Laure et son infaillible entrain (et son envie de cuisiner, dont on aura largement profité !) ; Anne-Marie et son côté suisse forcément très chocolaté ; Cédric notre Attila des goûters (ne semant que quelques miettes sur son passage) ; Ehsan et sa démonstration de Setâr un certain midi ; Jaonary qui n'aura pas manqué de me transmettre son

virus (photographique !) ; Jérôme (c'est pour quand, cette partie de tennis ?) ; Hiep et sa jelly vietnamienne qui nous aura marqué ; Hichem, Hui et Zsolt nos trois post-doc' ; Lockman qui arrive toujours à des heures improbables ; Patrick L., Pierre et Mickaël, les trois compères parisiens (qu'on ne voyait que trop peu ici !) ; et Nicolas (alors, quand est-ce qu'on vient te voir à Vienne ?).

Je n'oublie pas non plus les "anciens" avec qui j'ai pu parcourir un bout de chemin très agréable : Patrick E., Maxime, Charlotte, Noura, Olivier, Romain, Geoffray, Nassim, Thomas, Victor et Julien M.

Mon entière gratitude va à Jorge Cham, pour ses grandes et perspicaces pensées à propos de son expérience personnelle, m'ayant été hautement précieuse.

Je ne saurai clairement exprimer toute ma gratitude envers mes parents et grand-parents, qui ont subi mes humeurs et mon manque de présence auprès d'eux, tout en m'ayant inconditionnellement soutenu, aussi bien moralement que financièrement. Merci à vous.

Un autre merci tout particulier aux amis et personnes proches qui se sont régulièrement enquis de mon avancée et de ma motivation. Le soutien d'Annie-Claude m'aura été plus que précieux, tout comme les entraides avec Aurore, Caro, Damien, Louis et Nico avec qui j'ai pu avoir de nombreuses discutions "doctorales". Je remercie également tous mes autres amis, m'ayant soutenu de près ou de loin ces trois années durant, non cité mais pour qui je ne manque aucunement d'avoir une pensée sincère, ils se reconnaîtront facilement.

Enfin, mes dernières et (donc) plus précieuses pensées et doux remerciements vont à Marie-Pierre, qui a métamorphosé ces derniers mois en une période plus qu'heureuse. Merci, ma Toi.

Noisy-le-Grand, le 12 mars 2009.

Table des matières

Liste des figures

Liste des Algorithmes

Introduction

C'est par la force des images que, par la suite des temps, pourraient bien s'accomplir les «vraies» révolutions.

André Breton
Les Nouvelles littéraires, Hommage à Saint-Pol Roux, 1925

Eléments de contexte

Le *siècle de l'image*

Indubitablement, l'image a pris le pas sur tout autre support d'information depuis déjà plusieurs siècles, nos passés artistiques et médiatiques en ayant témoigné à maintes reprises. En toute situation, nous sommes aujourd'hui plongés dans un milieu fait d'images, qu'elles soient dessinées, photographiées, filmées, artificiellement générées, qu'elles représentent la réalité, fidèlement ou non, ou bien des idées plus abstraites, et qu'on les utilise à des fins commerciales, artistiques, ludiques, politiques ou autres.

Ces dernières décennies ont vu un accroissement de l'utilisation de l'image, grâce au développement ou à l'apparition de sous-catégories ou de nouvelles branches des arts majeurs, telles la photographie (de reportage, de mode,...), le dessin (bande dessinée, design graphique,...), la peinture (*light painting*,...), ou encore la mise en scène (dispositifs interactifs et/ou multimédias,...), et bien plus particulièrement grâce à tous les progrès technologiques liés à l'univers du *numérique*, impliquées dans la majorité des domaines artistiques, industriels ou commerciaux.

Le numérique et l'image

En effet, s'il ne fallait retenir de ce début de siècle qu'une seule forme de l'image, ce serait sans nul doute son pendant numérique, ayant apporté aux artistes et industriels "*une nouvelle faculté : une malléabilité infinie*" [199], exploitée dans bien des domaines.

Si l'industrie cinématographique est l'une des plus friande en matière d'imagerie numérique (un des moteurs actuels de cette industrie étant en effet l'omniprésence d'effets spéciaux, de retouches numériques, et d'animations de synthèse), elle est depuis 2004 dépassée en termes budgétaire par l'industrie vidéoludique (cette tendance s'accroit depuis fortement, atteignant 1.20 milliards d'euros pour le cinéma contre 2.96 milliards d'euros pour le jeu vidéo, en 2004), ainsi devenu en France le premier loisir (figure 1).

FIGURE 1 – *Exemples d'images numériques récentes, illustrant les besoins calculatoires. Gauche : l'acteur Bill Nighy dont les mouvements sont numérisés pour être appliqués à un modèle totalement 3D du personnage du capitaine Davy Jones dont l'animation sera numériquement affinée, dans le film* Pirates of the Caribbean : Dead Man's Chest, *2006 ;* © Industrial Light & Magic/Walt Disney Pictures. *Droite : extrait du jeu vidéo* Little Big Planet *sur Playstation 3, 2008, présentant une scène entièrement générée numériquement ;* © Media Molecule.

Cette culture de l'image et les besoins computationnels sans cesse grandissants, aussi bien quantitatifs que qualitatifs qu'elle engendre, ont fait apparaitre de nouvelles méthodes de création d'images numériques, basées sur des améliorations matérielles : ils ont en effet suscité l'émergence d'équipements informatiques voués à la génération d'images numériques.

Les premiers matériels dédiés aux calculs graphiques sont dus à la société Silicon Graphics, Inc. (SGI), ayant introduit au début des années 80 la première station de travail 3D : IRIS 2000. D'autres gammes, à usages principalement professionnels, sont apparues dans les années suivantes, chez SGI (avec les générations succédant au IRIS 2000, comme les *Power Series*) et quelques autres industriels.

Graphics Processing Units

Après quelques essais de création de matériels additionnels, sous forme de cartes informatiques, destinés aux calculs graphiques (comme la société ATI avec la *VGA Wonder*) dans les années 80, c'est surtout à partir du milieu des années 90 que, poussés par l'industrie du jeu vidéo, plusieurs industriels ont commencé à proposer des cartes informatiques grand public prenant en charge toute la partie affichage et surtout la partie calculatoire liée au graphisme à afficher, on les nomma les *cartes graphiques*. La série de cartes *Voodoo Graphics* de la société *3dfx Interactive* en 1994 en est l'emblème le plus marquant. Depuis, deux principaux constructeurs de cartes graphiques ont subsisté, ATI (racheté par AMD) et NVidia, améliorant sans cesse leurs cartes graphiques respectives, embarquant des *processeurs graphiques*, ou *GPU* (*Graphics Processing Unit*), toujours plus performants.

Avec de tels processeurs, la volonté de ces constructeurs était originellement de proposer un outil matériel permettant de décharger le processeur central de la machine (ou *CPU*, *Central Processing Unit*), des calculs 3D, voire de tous les calculs liés aux graphismes. Le succès dans le milieu du jeu vidéo fut immédiat et si important que ces GPU sont depuis devenus un des éléments constitutifs les plus importants dans le choix d'un ordinateur, qu'il soit personnel ou professionnel.

Ceci s'explique par plusieurs points. En premier lieu, la puissance brute de calcul proposée par les GPU a largement dépassé depuis quelques années celle affichée par les CPU les plus performants (figure 2),

grâce à leur architecture parallèle hautement spécialisée et optimisée pour les opérations graphiques. Le regroupement possible des GPU en cluster démultiplie encore cette puissance de calcul. Ensuite, la flexibilité de la programmation des GPU, s'amplifiant rapidement, permet d'y adapter de plus en plus de types d'algorithmes. Enfin, le faible prix (400€ pour un modèle haut de gamme) assure une grande accessibilité.

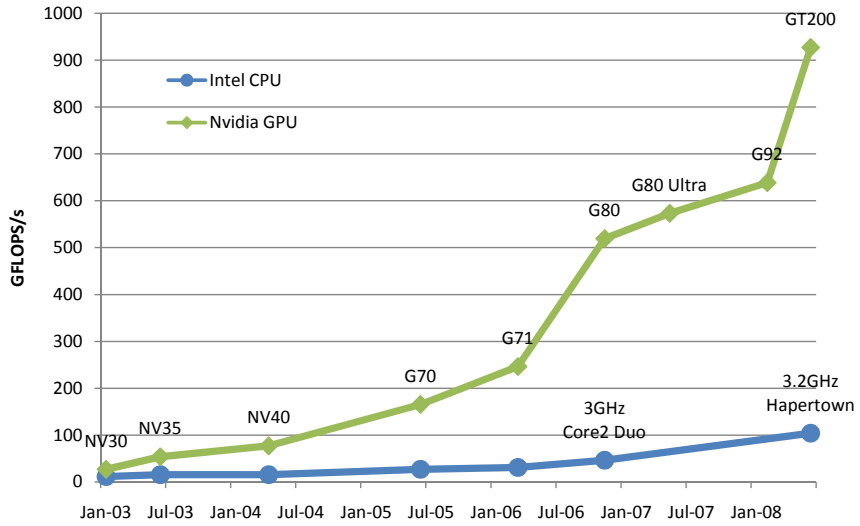


FIGURE 2 – *Puissances de calcul brutes comparées entre GPU NVidia et CPU Intel de 2003 à 2008. Dès 2003, les GPU ont pris l'ascendant sur les CPU. Malgré l'apparition des CPU multi-cœurs (Core2 Duo et Quad Core Hapertown), les architectures G80 et la toute récente G200 se démarquent plus que sensiblement, se montrant jusqu'à huit fois plus puissantes en terme de GFLOPS (Giga-FLOPS). Adapté de [172].*

Calcul généraliste par GPU

L'année 2002 a vu l'apparition d'une importante avancée dans la flexibilité de programmation des cartes graphiques. Elles devinrent suffisamment polyvalentes pour que la communauté scientifique commence à s'y intéresser de plus près et à y chercher des applications computationnelles *généralistes* potentielles, traditionnellement réalisées par le CPU plus polyvalent (figure 3).

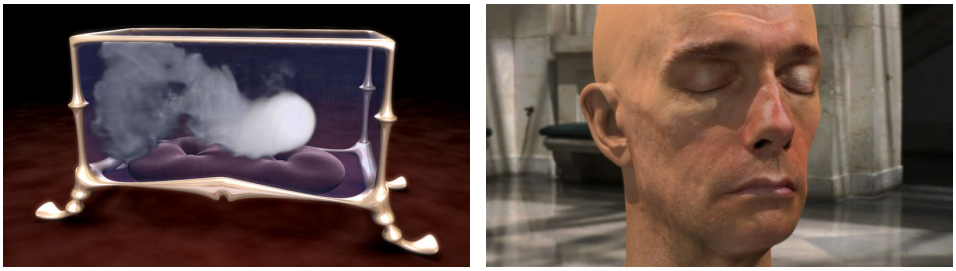


FIGURE 3 – *Exemples de simulations sur les GPU NVidia. Gauche : rendu de l'évolution d'un nuage de fumée dans une boîte. Les calculs physiques liés à l'écoulement du fluide, tout autant que les calculs graphiques, sont effectués par le GPU. Droite : rendu d'une tête humaine grâce à une modélisation complexe des différentes couches de l'épiderme et à une méthode de transluminescence (ou* subsurface scattering*, phénomène de pénétration de la lumière à travers la surface d'un objet translucide). Ces simulations sont exécutées en temps réel.*

Cette utilisation détournée des GPU est nommée *GPGPU* (*General Purpose computation on GPU*). Pour les chercheurs, l'intérêt de la programmation GPGPU est clair : pouvoir, à faible coût, utiliser des machines ou des clusters de machines parallèles, permettant d'accélérer grandement l'ensemble des calculs, avec des gains pouvant atteindre 2000%.

Les langages de programmation utilisés dans un cadre GPGPU ont évolué depuis 2002. Alors qu'on utilisait à ces débuts des *shading languages* (Cg, HLSL, GLSL,...), langages basés sur les capacités graphiques des cartes, de nouveaux langages GPU spécialisés dans le calcul généraliste ont depuis fait leur apparition, en particulier CUDA (*Computer Unified Device Architecture*) de NVidia et le très récent OpenCL [164, 234, 165] (officiellement soutenu par ATI, entre autres).

Malgré une encore faible implantation aujourd'hui de cette méthode de développement dans la communauté scientifique, un engouement croissant est à noter et pourrait à l'avenir changer certaines pratiques en recherche tout autant qu'en industrie. De plus, il est à noter qu'actuellement se pose la question de la convergence CPU/GPU en un seul processeur, les constructeurs ATI (processeur Fusion prévu pour la deuxième moitié de 2009) et Intel (processeur Larrabee, prévu pour fin 2009) s'y penchant fortement actuellement. Cela pourrait avoir pour même conséquence de modifier les habitudes de développement en recherche et en industrie.

Motivations

Conscient de la puissance proposée mais encore sous-exploitée des GPU, la ligne conductrice de cette thèse a été la volonté de développer de nouvelles méthodes et d'en adapter d'autres à une exploitation GPU, les domaines d'applications étant guidés par les différents projets auxquels le Certis, le laboratoire où cette thèse a été effectuée, est lié : en particulier le projet Facets, en lien avec l'équipe Odyssee, traitant de neurosciences, et le projet ANR Wired Smart.

En effet, deux des domaines d'applications qui nous ont concerné, la simulation de réseaux de neurones (chapitre 4) et l'appariement d'images (chapitre 6), ont tous deux comme défaut d'imposer des calculs très lourds, en grande quantité et sur des jeux de données toujours plus vastes (les réseaux de neurones simulées sont de plus en plus larges, et l'appariement d'images est fait à travers des ensembles de photographies de plus en plus grands), exigeant des temps de calculs bien trop importants pour des implémentations CPU standard.

L'utilisation de GPU nous parut alors comme une solution viable à explorer pour combler ce réel besoin de vitesse : par l'adaptation ou la création d'algorithmes de calcul sur GPU pour les domaines computationnels précités, il nous a semblé réaliste de pouvoir obtenir des facteurs de gains en temps notables.

L'emploi de plusieurs GPU conjointement sous la forme d'une grappe de calcul (cluster) est possible mais nécessite des méthodes de programmation spécifiques et les installations matérielles adéquates. Le projet ANR GCPMF (*Grilles de Calcul Pour les Mathématiques Financières*, regroupant l'équipe IMS de Supélec, le Cermics de l'Ecole des Ponts, les projets MATHFI, METALAU, OASIS et OMEGA de l'INRIA, le laboratoire PMA

de l'université Paris VI, le laboratoire MAS de Centrale Paris, ainsi que les sociétés BNP Paribas, Calyon, EDF, IXIS CIB, Misys Summit et Pricing Partners, et dont l'objectif est de mettre en valeur le potentiel du calcul parallèle appliqué aux mathématiques financières sur des infrastructures de grilles), dans lequel sont impliqués plusieurs personnes du Certis et du Cermics, étudie cette nouvelle forme de *grid computing* [237, 64]. Dans le cadre de cette thèse, nous avons choisi d'explorer l'utilisation non pas d'un tel cluster, mais d'un unique GPU, c'est-à-dire un seul nœud d'une potentielle grappe de calcul, plus accessible qu'une grille complète de calcul.

Il est enfin à noter que lors du début de cette thèse, la communauté GPGPU se créait tout juste, seuls les *shading languages* permettaient de développer sur GPU, c'est à partir de l'un deux, Cg (avec OpenGL comme interface graphique), que nous avons développé la librairie *CLGPU*, librairie de calcul généraliste sur GPU, décrite dans le chapitre 1. Depuis, le langage CUDA, dédié au calcul généraliste, a été introduit. La question de la réimplémentation de la CLGPU en CUDA ne se pose que depuis l'apparition il y a quelques mois (septembre 2008) de l'interopérabilité avec OpenGL. Apparemment, il semble possible de créer une interface utilisant Cg et CUDA sans que la différence soit perceptible au niveau utilisateur.

Organisation de ce rapport et contributions

Cette thèse est organisée de la façon suivante :

Première partie

Une première partie s'intéresse tout d'abord à définir ce qu'est la programmation GPU et plus particulièrement GPGPU, ses concepts et astuces, puis à en montrer un panel d'applications préexistantes.

Le chapitre 1 présente la programmation GPGPU : après avoir explicité quelques éléments de calcul parallèle, une évolution des GPU est montrée à travers un historique des différentes générations ayant existé ainsi que l'architecture globale des cartes actuelles. Ensuite nous détaillons les concepts inhérents au développement GPGPU, les contraintes impliquées, quelques astuces classiques de programmation, puis nous présentons (avec exemples) et comparons, dans une optique GPGPU, deux langages utilisables sur ces GPU : Cg et CUDA. Enfin, nous présentons une librairie destinée à du calcul généraliste sur GPU et développée par les membres du Certis, à des fins de recherche et d'enseignement : la *CLGPU*.

Le chapitre 2 répertorie de façon non-exhaustive les applications génériques, dont la partie computationnelle est réalisée sur GPU, en les classant par domaines : nous commençons par exposer quelques outils mathématiques adaptés sur GPU, puis nous présentons des applications dans les domaines de la simulation physique, du traitement de signal, différentes méthodes de rendu graphiques, de traitement d'image et de vision par ordinateur. Nous avons ici essayé d'exposer un panel d'applications représentatif des possibilités offertes par les GPU.

Deuxième partie

Une deuxième partie présente notre premier domaine d'application : les neurosciences, sciences étudiant le système nerveux (composé du cerveau, de la moelle épinière, des nerfs, des organes des sens et du système nerveux neuro-végétatif) dans son anatomie et son fonctionnement.

Plus précisément, nous nous intéressons aux réseaux de neurones impulsionnels. Un réseau de neurone est un modèle de calcul qui s'inspire de façon schématique du comportement de neurones physiologiques ; le neurone impulsionnel est une des modélisations possibles du neurone, cherchant à reproduire une des caractéristiques biologiques : le potentiel d'action. Nos études dans ce domaine, présentés dans le chapitre 4, ont été menées au sein de l'équipe Odyssee, basée à l'Inria Sophia-Antipolis, pour le projet Facets (*Fast Analog Computing with Emergent Transient States*) fondé par la Commission Européenne, dont le but est de créer des fondations théoriques et expérimentales pour la réalisation de nouvelles représentations computationnelles inspirées des systèmes nerveux biologiques.

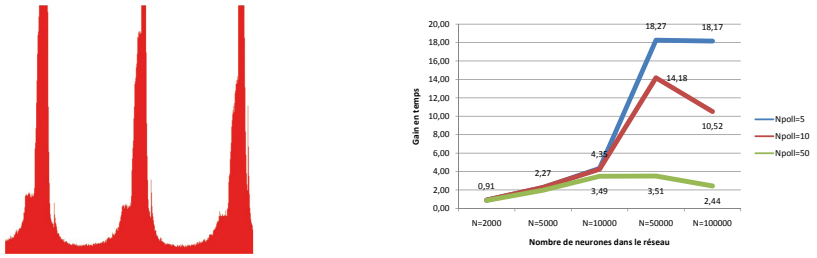


Figure 4 – *Simulation de réseaux de neurones. Gauche : histogramme de décharge de neurones impulsionnels. Droite : gains en temps observés dans l'une de nos applications. Pour de plus amples détails ; voir chapitre 4.*

Le chapitre 3 introduit quelques concepts et propriétés physiques propres aux neurones, puis définit quelques unes des modélisations de neurones les plus célèbres, cherchant ou non à retranscrire ces propriétés, en particulier la famille des modélisations impulsionnelles, ou *intègre-et-tire*.

Le chapitre 4 explicite les deux applications que nous avons développées en GPU dans le domaine neuroscientifique. La première est une simulation d'un réseau de neurones impulsionnels dans lequel la connexité entre neurones n'est pas locale. Ce travail a été publié à NeuroComp'06 [34]. La seconde application simule un autre réseau de neurones impulsionnels à connexité non locale, en introduisant la notion de *sondage des voisins*, permettant de grandement réduire la charge de calculs tout en conservant une précision très acceptable. L'ensemble de ces travaux résultent d'une collaboration avec Romain Brette.

Troisième partie

Une troisième et dernière partie traite d'un autre domaine computationnel, la Vision par Ordinateur, science des *machines qui voient*, dont le but est l'édification de théories et systèmes permettant la compréhension de l'information se trouvant à l'intérieur d'images de provenances

diverses (photographies, images vidéo, scanner IRM,. . .) et dont quelques unes des applications principales sont la segmentation d'objets, la reconstruction 3D de scènes ou d'objets, la reconnaissance de formes ou la classification par contenu.

FIGURE 5 – *Applications GPU en vision par ordinateur. Gauche et milieu : reconstruction 3D par stéréovision, en* wireframe *et avec texture ; voir chapitre 5. Droite : Appariements de points d'intérêts entre deux images d'une même scène ; voir chapitre 6.*

Le chapitre 5 propose premièrement une courte introduction au domaine de la Vision par Ordinateur, et en particulier au principe de la stéréovision. Nous exposons ensuite notre algorithme de reconstruction 3D par stéréovision dense, adaptée au calcul GPU. Cet algorithme est basé sur une fonction d'énergie comprenant un terme de régularisation que l'on minimise par descente de gradient. Ces travaux ont fait l'objet d'une publication à 3DPVT'06 [146].

Le chapitre 6 commence par couvrir différentes méthodes permettant de détecter et caractériser des *points d'intérêt* dans des images photographiques, c'est à dire de localiser les points pertinents, sur lesquels ou autour desquels se trouve l'information qui permettra de les caractériser, dans l'optique d'être également retrouvés dans d'autres images représentant le même objet. Ensuite, nous présentons un procédé général d'appariements de points d'intérêts, précédemment détectés, à travers un jeu d'images du même objet ou de la même scène, et proposons un algorithme d'appariements massifs de points d'intérêts, adapté sur GPU. Enfin, une application directe est montrée à travers l'ébauche d'un logiciel permettant d'assister un ou plusieurs utilisateurs lors de la capture photographique d'une scène large, en indiquant les zones de la scène trop peu capturées pour pouvoir être discernées. Les travaux présentés dans ce chapitre, publiés à ICPR'08 [33], ont été réalisés dans le cadre du projet *Wired Smart*, projet créé suite à l'appel RIAM (Recherche et Innovation en Audiovisuel et Multimédia) de l'ANR (Agence Nationale de la Recherche), cofinancé par cette dernière et le CNC (Centre National de la Cinématographie), regroupant le laboratoire I3S de l'université de Nice-Sophia Antipolis, le Département Informatique de l'ENS Ulm et les sociétés RealViz et Mikros Image, et ayant pour objectif la mise au point de solutions matérielles pour des architectures de suivi de contours.

Le chapitre 7 synthétise la méthodologie de développement exploitée lors des travaux présentés dans les chapitres précédents, en mettant en avant quelques conseils et interrogations à se poser avant un nouveau développement sur GPU, et en rappelant les principales différences entre les langages de programmation sur GPU orientés graphiques et ceux plus généralistes. Un questionnement est ensuite effectué sur les apports et contraintes d'une éventuelle réimplémentation des applications présen-

tées dans cette thèse avec un langage généraliste, à la place d'un langage orienté graphique comme nous l'avons fait.

Première partie

Concepts et applications GPGPU

Programmation générique sur GPU

1

Sommaire

Ce chapitre introductif va présenter quelques éléments de programmation générique sur GPU. Après une définition et l'exposition des motivations poussant la communauté à s'intéresser à cet outil, nous expliciterons l'organisation de ces cartes graphiques et nous détaillerons les concepts fondamentaux inhérents à leur programmation, en donnant et comparant des exemples en deux langages. Nous finirons par présenter la CLGPU, librairie informatique dédiée au calcul GPU, implémentée par nos soins et utilisée dans nos diverses implémentations d'algorithmes sur carte graphique.

1.1 Définition et Motivations

Au sein d'une architecture informatique, les processeurs centraux, *CPU*, ont originellement la charge de l'ensemble des calculs. Lorsque la nécessité de traiter des données de large volume (images, vidéo ou son) est apparue dans les années 80, leurs architectures se sont adaptées en proposant des jeux de fonctionnalités dédiées à ce traitement *multimédia* : *MMX* pour les processeurs de marque *Intel*, ou bien *3DNow !* pour ceux d'*AMD*, ont contribué au succès de ces gammes de processeurs.

Poussée par l'industrie vidéo-ludique, ayant récemment dépassé en terme de budget l'industrie cinématographique, l'émergence d'un type de matériel informatique dédié aux traitements graphiques déchargeant le CPU, nommé GPU, *Graphics Processing Unit*, est née, il y a une dizaine d'années, du besoin de plus en plus grand de spécialisation des architectures des processeurs, voire de multiplication de ceux ci. Les GPU sont ainsi des processeurs portés par une carte annexe, prenant à leur charge le traitement des images et des données 3D, ainsi que l'affichage.

L'implantation de ce type de matériel s'est aujourd'hui largement répandue, et a pour conséquence inattendue leur utilisation dans des domaines non graphiques, en particulier en simulation numérique. C'est ce type d'utilisation détournée que l'on nomme *GPGPU*, *General Purpose computing on Graphics Processing Units*, ou *Programmation Générique sur Carte Graphique*.

L'engouement de la communauté pour cette méthode de programmation est tel qu'aujourd'hui, ces cartes graphiques sont devenues de véritables outils computationnels professionnels. Etant architecturalement prévus pour un traitement massivement parallèle des données, ils devraient devenir dans un avenir très proche de véritables coprocesseurs utilisés par tout type d'application.

Les raisons d'un tel engouement pour la programmation GPGPU sont très nombreuses, nous nous proposons d'en citer les principales.

Puissance de calcul Les besoins toujours plus importants de réalisme pour le rendu d'images demandent une puissance de calcul croissant continuellement et ont naturellement poussés les industriels à multiplier les capacités matérielles de ces cartes. Que ce soit au niveau du nombre de processeurs parallèles contenus sur les cartes graphiques, du nombre de transistors, de la finesse de gravure ou de la quantité de mémoire disponible, leur évolution a été spectaculaire. La figure 1.1, adaptée de [172], présente l'évolution comparée des puissances brutes de calcul en terme de GFLOPS, entre les CPU Intel et les GPU NVidia.

Calcul parallèle massif Contenant jusqu'à 240 processeurs, les GPU sont conçus pour exécuter des tâches massivement parallèles, jusqu'à plusieurs milliers de threads. Pour cette raison, les GPU pourraient s'apparenter à des supercalculateurs débarrassés de structure complexe plutôt qu'à des CPU multi-cœurs, capable de traiter seulement quelques threads simultanément. A l'avenir, cela pourrait changer, mais à l'heure actuelle, en privilégiant la rapidité de calcul, les GPU sont préférables.

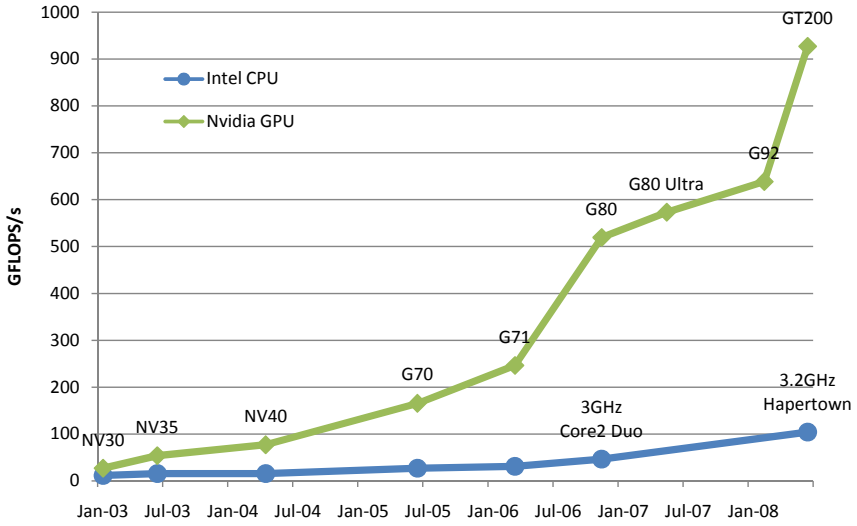


FIGURE 1.1 – *Puissances de calcul brutes comparées entre GPU NVidia et CPU Intel de 2003 à 2008. Dès 2003, les GPU ont pris l'ascendant sur les CPU. Malgré l'apparition des CPU multi-cœurs (Core2 Duo et Quad Core Hapertown), les architectures G80 et la toute récente G200 se démarquent plus que sensiblement, se montrant jusqu'à huit fois plus puissantes en terme de GFLOPS (Giga-FLOPS). Adapté de [172].*

Flexibilité Auparavant simplement paramétrables, les cartes graphiques permettent aujourd'hui une programmation flexible et avancée, en proposant de plus en plus de contrôle sur les caractéristiques toujours plus nombreuses, telle l'utilisation de types de données (en particulier le calcul flottant sur 32bits), de structures et d'instructions (branchements conditionnels) de plus en plus complexes. Il existe même dorénavant des outils de débogage destinés à certains langages utilisables en GPGPU. La programmation d'un GPU est ainsi de plus en plus semblable à une programmation séquentielle classique.

Plusieurs GPU Certains GPU, et en particulier les haut-de-gamme, peuvent être combinés en cluster, démultipliant encore la puissance de calcul. C'est le cas de la gamme Tesla, proposées par NVidia, qui associe jusqu'à 4 GPU et 16Go de mémoire dans un unique châssis, portant à 4 TFLOPS (4000 GFLOPS) la puissance de calcul de ce supercalculateur. Plusieurs calculateurs de ce type peuvent, de plus, être associés par un réseau classique.

Prix et disponibilité Pour environ 400€, il est possible de se procurer les GPU les plus puissants existants sur le marché actuel. Ces très faibles prix sont le résultat de l'existence d'un marché énorme et d'une concurrence rude entre les deux plus importants fabricants de cartes graphiques, NVidia et ATI, dont la cible commerciale principale est le grand public, pour la pratique des jeux vidéo. La composition d'un cluster à utilisation professionnelle est donc très raisonnablement envisageable. De plus, par ces faibles coûts, il est aujourd'hui rare pour une station de travail ou même un ordinateur grand public de ne pas disposer d'un GPU performant.

Ces arguments tendent à prouver que la puissance des GPU, longtemps négligée, est prête à être pleinement exploitée, dans tout domaine computationnel.

1.2 Eléments de parallélisme

Les capacités parallèles des GPU permettent d'exécuter un partitionnement d'une tâche complexe en une multitude de tâches élémentaires, pouvant être réalisées par ses différents processeurs travaillant simultanément, rendant l'exécution globale bien plus rapide qu'une exécution séquentielle. Des algorithmes de domaines variés (météorologie, finance, traitement d'image...), comme nous le montrerons dans le chapitre 2, se prêtent bien à cette découpe. On se propose de donner ici les définitions de quelques éléments de calcul parallèle, non spécifiques aux GPU.

1.2.1 Caractère SIMD/MIMD

Les architectures parallèles, composées en général d'un grand nombre d'unités de calcul, exploitent fortement les modes de fonctionnement SIMD / MIMD, autorisant l'exécution simultanée de plusieurs parties de code. Ces modes sont deux de ceux référencés dans la taxonomie de Flynn [56], classifiant les différents types d'architectures des systèmes informatiques parallèles toujours d'actualité.

Définition 1.1 *Le* **SIMD**, **Single Instruction on Multiple Data**, *est un mode de calcul parallèle dans lequel chaque unité de calcul reçoit la même suite d'instruction en plus de données propres, comme pour les processeurs vectoriels.*

Ce modèle est représenté en figure 1.2. Une exécution d'un code en SIMD peut être synchrone (on attend qu'une partie du code ait été exécuté sur toutes les unités de calcul. On parle de SIMD *vectoriel*, c'est par exemple le cas pour les GPU) ou asynchrone (chaque unité de calcul progresse indépendamment des autres. C'est alors un SIMD *parallèle*).

Les instructions SIMD sont accessibles dans de nombreux processeurs professionnels et grand public depuis 1997 : MMX [182] pour Intel, 3DNow ! [175] pour ATI, les jeux SSE (jusqu'à SSE4) pour les processeurs de type x86, AltiVec [45] pour Apple, IBM et Motorola.

Elles sont parfaitement adaptées, par exemple, au traitement de signal, particulièrement le traitement d'image, dans lequel on fait subir à chaque donnée élémentaire, le *pixel*, le même traitement, de façon indépendante. Dans ces cas, l'exécution du code est en général bien plus rapide qu'une implémentation classique SISD (*Single Instruction on Single Data*), dans lequel les instructions sont exécutées exclusivement séquentiellement.

Cependant, il existe des contreparties aux processeurs SIMD : leur programmation est souvent malaisée, un algorithme n'est pas forcément exécutable sur une machine SIMD, la gestion des registres est plus complexe,...

Définition 1.2 *Le mode* **MIMD**, **Multiple Instruction on Multiple Data** *permet d'exécuter des suites d'instructions différentes sur des données différentes, de façon asynchrone et indépendante.*

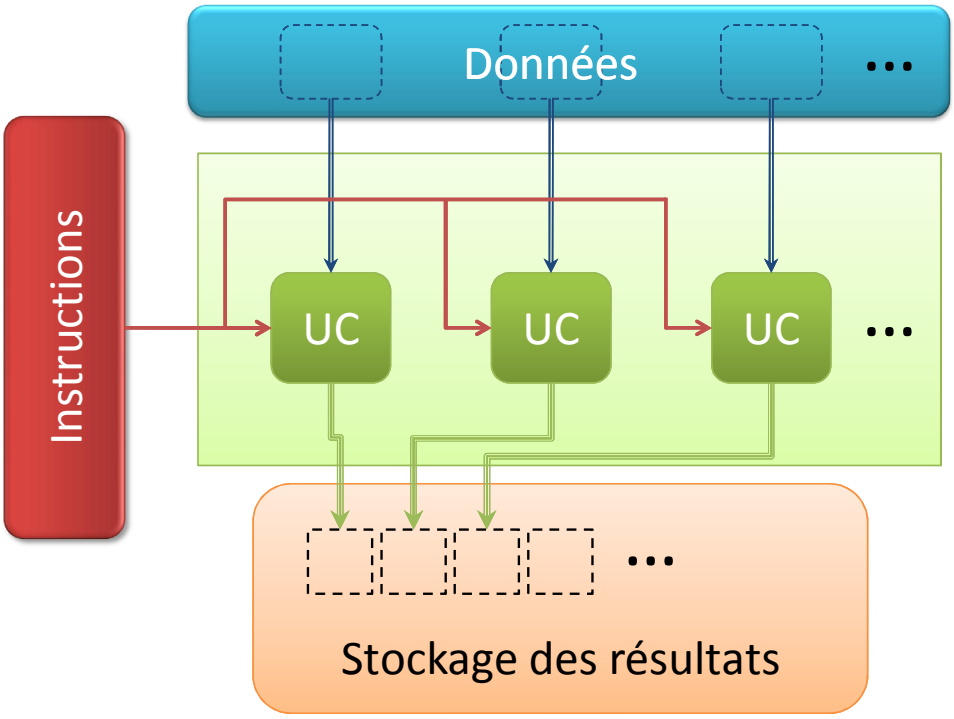


FIGURE 1.2 – *Le modèle parallèle* Single Instruction Multiple Data *: toutes les unités de calcul (UC) traitent simultanément, avec la même instruction, une partie des données qui leur est propre. Chacune de ces unités va ensuite ranger son résultat dans une partie de la mémoire.*

Ce modèle est représenté en figure 1.3. Chaque unité de calcul reçoit sa propre liste d'instructions et l'exécute sur ses propres données. Les machines MIMD sont plutôt destinées au monde professionnel. Les *Connection Machines 5* [235], de *Thinking Machines Corporation* et les *Transputers* [242] du défunt *Inmos* en sont deux des représentants.

On distingue deux types d'accès mémoire dans ce mode :

- *Mémoire distribuée* : chaque unité de calcul possède sa propre partie mémoire et ne peut pas accéder à celles des autres. Une communication est possible entre processeurs par le biais de *messages* ou d'*appels de procédures RPC*, mais nécessite de connecter ces unités MIMD de façon adéquate, sur le modèle d'une grille, par exemple.
- *Mémoire partagée* : aucune des unités de calcul n'a de mémoire propre mais toutes peuvent accéder à une même mémoire globale. Un changement effectué par une unité dans cette mémoire est donc visible depuis les autres unités. Pour éviter les problèmes de synchronisation, les principes classiques d'exclusion sont utilisés : sémaphores, mutex,...

1.2.2 PRAM

Définition 1.3 *On désigne par* **PRAM**, *ou* **Parallel Random Access Machine**, *un modèle de référence des machines à partage de mémoire, sur lesquels peuvent s'exécuter des algorithmes parallèles c'est le cas des GPU).*

Trois modèles sont distingués en fonction de leurs moyens d'accès à la mémoire, liés aux notions de *gather* et *scatter* (détaillées dans la section 1.2.3) :

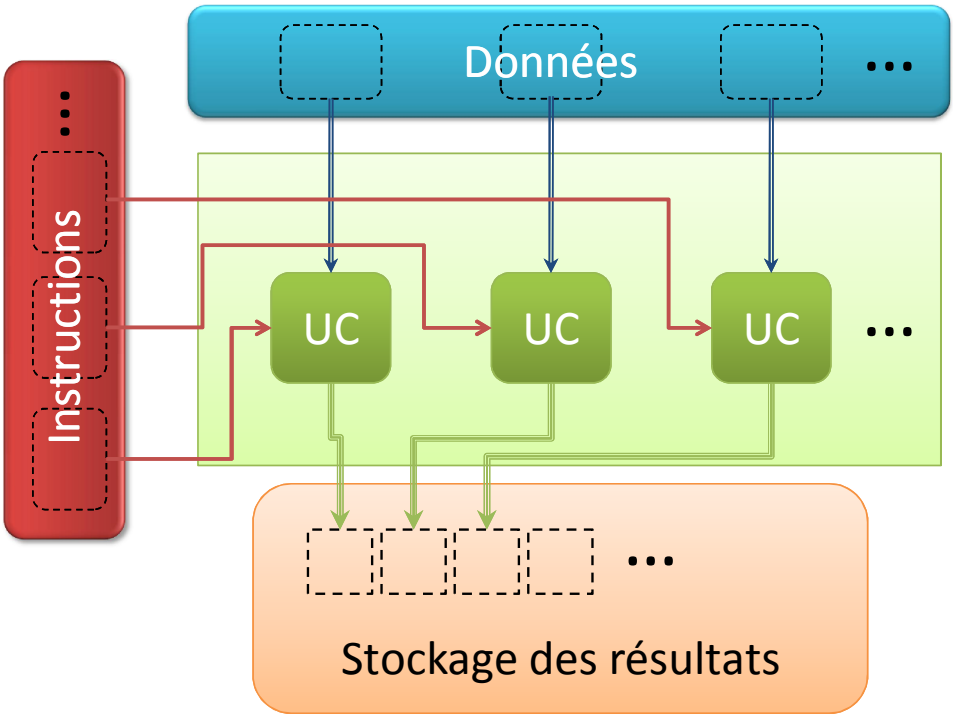


FIGURE 1.3 – *Contrairement au SIMD, le* Multiple Instruction Multiple Data *assigne à chaque unité de calcul (UC) une suite d'instruction spécifique. Ces unités reçoivent toujours une partie des données qui leur est propre et stockent leurs résultats dans une même partie de mémoire.*

1. **EREW**, *Exclusive Read Exclusive Write* : chaque processeur ne peut lire ou écrire à un endroit mémoire que si aucun autre n'y accède au même moment ;
2. **CREW**, *Concurrent Read Exclusive Write* : chaque processeur peut lire à n'importe quel endroit en mémoire, mais plusieurs processeurs ne peuvent écrire simultanément au même endroit.
3. **CRCW**, *Concurrent Read Concurrent Write* : chaque processeur peut lire et écrire où il le souhaite en mémoire. Dans ce cas, on distingue trois sous-cas :
   - CRCW commun : si plusieurs processeurs écrivent la même valeur au même endroit, l'opération réussit. Sinon, elle est considérée illégale ;
   - CRCW arbitraire : si plusieurs processeurs écrivent au même endroit, un des essais réussit, les autres sont avortés ;
   - CRCW prioritaire : si deux processeurs écrivent au même endroit, leur rang de priorité détermine le seul qui réussit.

1.2.3 Aptitude à supporter les opérations de Gather et Scatter

Définition 1.4 **Gather** *et* **Scatter** *désignent la possibilité, pour une unité de traitement, de lire, respectivement d'écrire, des données à différents endroits de la mémoire, adressés indirectement.*

Gather et scatter sont des opérations fondamentales dans tout traitement informatique. La figure 1.4 illustre ces deux principes. Dans un langage ressemblant au C, une opération de lecture type gather s'écrit `u=d[i]`, où `d` est une structure de données stockée en mémoire, `i` un

index, et `u` une variable de stockage. Au contraire, l'opération de scatter s'écrit `d[i]=u`.

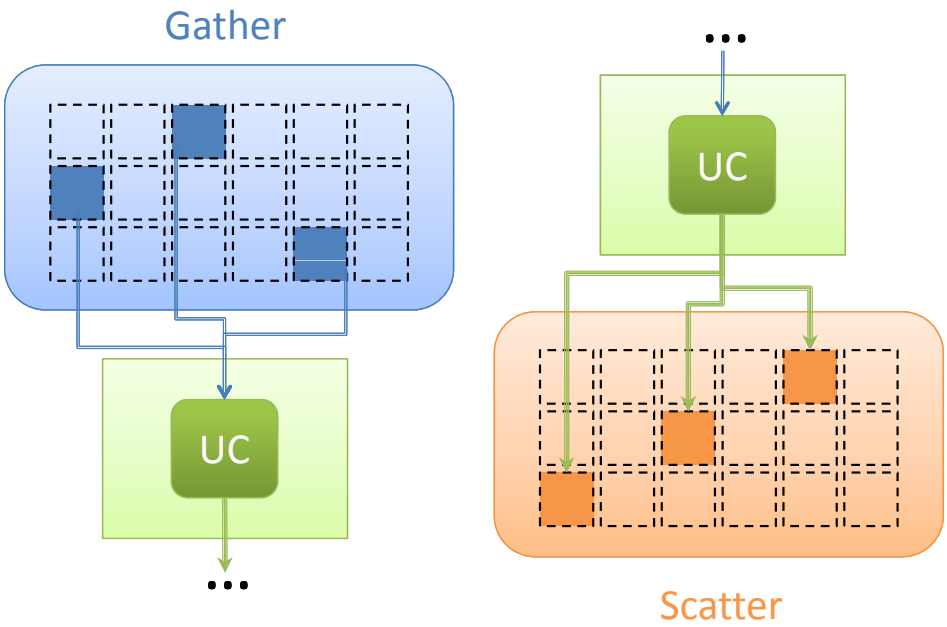


FIGURE 1.4 – *Par gather, l'unité de calcul a un accès indirect en lecture à la mémoire dans laquelle sont stockées les données. Par scatter, cette unité peut écrire ses résultats à des endroits de la mémoire adressés indirectement.*

1.3 Historique des GPU et Organisation interne

Depuis l'apparition dans les années 80 des accélérateurs graphiques matériels, leur évolution n'a cessé de s'accélérer. Les premières cartes graphiques accessibles au grand publiques sont apparues dans le milieu des années 90, avec en particulier les *Voodoo Graphics*. C'est NVidia qui a introduit le terme *GPU*, remplaçant le terme *VGA Controller*, alors insuffisant pour désigner l'ensemble des possibilités offertes par ces cartes.

L'industrie du jeu vidéo, en particulier, n'a depuis cessé de motiver le développement de nouveaux processeurs dédiés au traitement graphiques. Cette évolution peut être catégorisée en différentes générations [52], selon leurs capacités, sur lesquelles nous nous proposons de dévoiler quelques aspects. Une description plus détaillée de l'architecture des dernières générations, permettant de comprendre leur utilisation dans nos applications, est ensuite proposée.

1.3.1 Les différentes générations de GPU

Les forces ayant poussé les industriels à améliorer sans cesse ces cartes graphiques sont dues principalement à un aspect économique de compétitivité, pour faire face à un appétit de plus en plus grand de complexité visuelle ou de réalisme dans les représentations cinématographiques ou les simulations vidéo-ludiques, poussé par une volonté assurément humaine de divertissement.

Suivant la loi de Moore [159], faite pour les CPU qui ne la respectent plus aujourd'hui, et stipulant que le nombre de transistors sur un processeur double tous les 18 mois, les architectures des GPU ont évolué depuis

une décennie, franchissant parfois des sauts technologiques importants. Nous nous proposons ici de classer ces architectures en différentes générations.

Avant les GPU Dans les années 80, des sociétés comme *Silicon Graphics* ou *Evans and Sutherland* proposèrent des solutions matérielles extrêmement couteuses, réservé à quelques professionnels spécialisés, et ne pouvant effectuer que quelques opérations graphiques très simples comme des transformations de vertex ou des applications de textures. De façon générale, le reste des opérations graphiques étaient accomplies par le CPU.

Première génération La première génération de GPU à proprement parler a débuté avec l'arrivée des *Voodoo Graphics* de *3dfx Interactive* en 1996, et dura jusqu'en 1999. Les principales cartes de cette génération sont les TNT2 de NVidia (architecture NV5), les Rage d'ATI et les Voodoo3 de 3dfx. Les premières opérations graphiques disponibles sont la rasterization de triangles et l'application de texture. Ces cartes implémentent également le jeu d'instruction de DirectX 6, API (*Application Programming Interface*, *Interface de Programmation*) alors standard. Cependant, elles souffrent de certaines limitations, principalement la grande faiblesse du jeu d'instructions mathématiques et l'impossibilité de transformer matériellement des vertex, opération lourde encore à la charge du CPU.

Deuxième génération Les premières GeForce 256 de NVidia (NV10) font leur apparition en 1999, à peu près en même temps que les Radeon 7500 d'ATI (architecture RV200) et Savage 3D de S3. Ces cartes permettent maintenant une prise en charge complète de la transformation des vertex et du calcul des pixels (*Transform and Lightning*, *T&L*). Les deux API principales, DirectX 7 et OpenGL, sont maintenant supportées par ces cartes. Les améliorations apportées au jeu d'instructions rendent ces cartes plus facilement configurables, mais pas encore programmable : les opérations sur les vertex et les pixels ne peuvent être modifiés par le développeur.

Troisième génération Dès cette génération, les constructeurs NVidia et ATI se partagent la quasi-totalité du marché, NVidia ayant fait l'acquisition de 3dfx. Les Geforce 3 (NV20) en 2001 et GeForce 4 Ti (NV25) en 2002 de NVidia, la Radeon 8500 d'ATI (R200) en 2001 forment cette génération de GPU, permettant enfin au développeur de diriger la transformation des vertex par une suite d'instructions qu'il spécifie. Néanmoins, la programmabilité des opérations sur les pixels n'est toujours pas possible. DirectX 8 et quelques extensions à OpenGL permettent tout de même une plus grande souplesse de configuration dans le traitement de ces pixels.

Quatrième génération Courant 2002, ATI propose sa Radeon 9700 (R300), et NVidia sa GeForce FX (NV30) à la fin de la même année. Ces deux cartes, en plus de supporter le jeu d'instructions DirectX 9 et de nouvelles extensions OpenGL, apportent de plus la possibilité de programmer le traitement des pixels. C'est à partir de cette quatrième génération de cartes que les premières opérations GPGPU ont pu se faire.

Cinquième génération Les GeForce 6 (NV40, avril 2004) et GeForce 7 (G70, juin 2005) de NVidia, les Radeon X800 (R420, juin 2004) et X1800 (R520, octobre 205) et dérivées composent cette génération. Ces cartes permettent quelques fonctions intéressantes, en particulier en calcul GPGPU : l'accès aux textures lors de la transformation des vertex, le rendu dans différentes textures (*Multiple Render Target*, *MRT*) et le branchement dynamique, uniquement pour la transformation des sommets, améliorant grandement la vitesse d'exécution. Le traitement des pixels ne supporte pas ce branchement dynamique.

Sixième génération C'est la génération en plein essor actuellement, équipant la plupart des ordinateurs sur le marché. Ses limites sont plus floues, chaque constructeur apportant ses innovations propres. Elle est composée des GeForce 8 (G80) et GeForce 9 (G92), lancées respectivement en 2006 et 2008, apportant des modifications dans la façon de concevoir le pipeline graphique. Entre la transformation des vertex et la rasterization des primitives (leur transformation en fragments, ou pixels), une étape supplémentaire est insérée, permettant de modifier la géométrie du maillage des primitives : insertions et suppression de vertex sont possibles, ce qui ouvre de nouvelles voies au calcul GPGPU. L'ajout du type `int` utilisable en interne (registres) et surtout en externe (textures) a également contribué à élargir le nombre d'applications possibles. De plus, sur les cartes GeForce 8, il n'existe plus de différence physique entre les différents processeurs. L'architecture matérielle modifiée est dite *unifiée*, cette caractéristique étant exploitée par le langage de programmation CUDA, de NVidia, utilisable avec des cartes à architectures G80 ou supérieures. La série GeForce 9 ne connait pas un succès véritable, ayant moins de mémoire et ses performances étant égales voire inférieures aux cartes équivalentes de la série GeForce 8, pour un prix semblable. Les séries de cartes d'ATI de cette génération sont les Radeon HD2000 et Radeon HD3000 (R600) , lancées en 2007, pour contrer la série GeForce 8. Malgré quelques améliorations intéressantes proposées par ce constructeur (support de DirectX 10.1, de la norme Shader 4.1 par exemple), ces cartes peinent à s'imposer à cause d'un prix trop élevé, de pièces bruyantes et de puces chauffant beaucoup.

Septième génération La dernière génération en date n'est pas encore répandue à l'heure actuelle. Les GeForce 200 (G200), lancées en juin 2008, apportent des améliorations principalement techniques : augmentation de la mémoire disponible, du nombre de processeurs, des fréquences mémoire et GPU, de la bande passante par élargissement du bus de données, pour des performances annoncées jusqu'à deux fois supérieures aux séries précédentes. Pour ATI, les Radeon HD4000, en juin 2008 également, sont censées rivaliser avec la série GeForce 9, mais les performances des modèles haut de gamme les mettent au même niveau que les GeForce 200, proposant les mêmes types d'améliorations techniques, mais affichant des tarifs plus attractifs et une consommation électrique revue à la baisse.

Rétrocompatibilité Il est important de noter que les programmes écrits pour un type ou une génération de cartes restent utilisable avec les générations futures de cartes, même si toutes les capacités ne sont plus néces-

sairement exploitées. C'est, par ailleurs, un problème important de développement que de devoir faire avec les différentes générations de cartes présentes sur le marché.

1.3.2 Organisation d'un GPU

Deux aspects importants permettent de caractériser l'organisation des GPU : le contenu matériel orienté calcul parallèle et son organisation adéquate en *pipeline*.

1.3.2.1 Composants matériels

La carte graphique est un des organes de l'ordinateur, chargé du traitement graphique et de l'affichage. Elle se compose d'une zone mémoire et de processeurs, auxquels viennent s'ajouter divers registres et chipsets de communication.

Ce qu'on appelle GPU, au sens strict, est l'ensemble des processeurs graphiques contenus sur cette carte, c'est-à-dire les unités opérant les calculs. Par abus de langage, on nommera également GPU la carte entière.

La mémoire disponible varie aujourd'hui jusqu'à 768Mo, cadencée à 1080MHz, accessible en lecture et en écriture par les processeurs de la carte mais aussi par le CPU.

Les GPU embarquent jusqu'à 128 (G80) ou 240 processeurs de flux chacun (G200), cadencés jusqu'à 1500MHz, et répartis en différentes catégories, abordés dans la section 1.3.2.2. Ces processeurs sont nommés *processeurs de flux* en raison de leurs natures : ils sont en effet capables de traiter, de façon parallèle, un ensemble de données élémentaires présenté sous la forme d'un flux.

Chacun de ces processeurs fonctionne en mode SIMD parallèle / CREW, il reçoit donc un ensemble d'instructions qui sera exécuté sur la totalité des données du flux. Ils sont de plus tous capables de gather, mais pas de scatter : ils peuvent accéder à n'importe quelle adresse mémoire en lecture, mais pas en écriture (voir sections 1.2.1 et 1.2.3).

Les processeurs d'un GPU sont organisés en *pipeline* que nous nous proposons de décrire.

1.3.2.2 Pipeline graphique actuel

Nous donnons tout d'abord quelques définitions utiles.

Définition 1.5 *Un* **vertex** *est un point géométrique 3D, sommet d'un ou plusieurs polygones.*

Définition 1.6 *Un* **fragment** *est une partie élémentaire d'une scène en cours de rendu qui pourrait occuper l'espace d'un pixel dans l'image finale. La différence entre pixel et fragment est une vue d'esprit : alors que le pixel est un "point" de couleur visible par l'utilisateur, le fragment est le "point" de couleur que le programme manipule, provenant de différentes textures, et dont le traitement produira un pixel.*

Définition 1.7 *Un* **shader** *est un programme à vocation graphique, généralement court, que le développeur peut spécifier en différents langages, maniant des vertex ou des fragments et permettant de contrôler un sous-ensemble des processeurs d'un GPU.*

Définition 1.8 *Un* **pipeline** *est une séquence ordonnée de différents étages. Chaque étage récupère ses données de l'étage précédent, effectue une opération propre et renvoie ses résultats à l'étage suivant.*

Un pipeline est dit *rempli* lorsque tous ses étages sont mis à contribution simultanément, c'est son utilisation optimale, diminuant le nombre d'étapes globales d'exécution d'un algorithme par rapport à une version séquentielle classique.

Le pipeline des GPU actuels est représenté figure 1.5. Il est composé de trois étages programmables, pilotés par les *shaders*, et d'un étage non-programmable, le *rasterizer*.

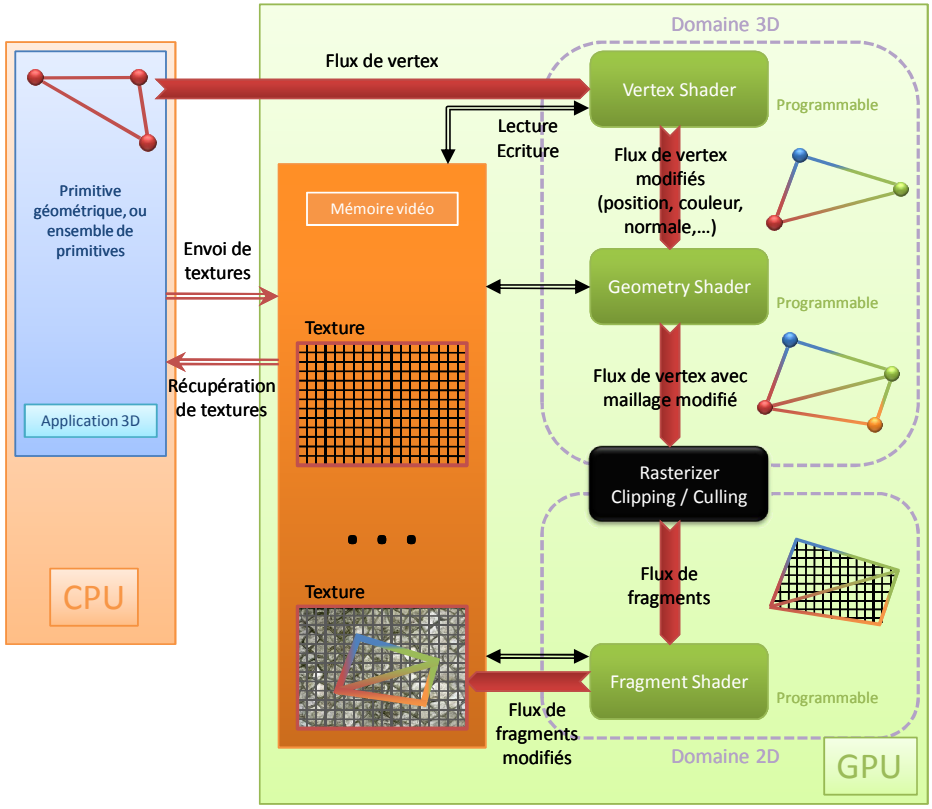


FIGURE 1.5 – *Pipeline GPU. Les primitives géométriques sont envoyées sous la forme d'un flux de vertex au* Vertex Shader*, programmé par le développeur, et pouvant modifier les attributs des vertex. Le flux sortant est dirigé dans le* Geometry Shader*, deuxième shader programmable, ayant la capacité de modifier le maillage, donc le nombre de vertex. Le* Rasterizer *transforme ensuite ces vertex et primitives en une image composée de* fragments*. Le flux de fragments résultant est pris en charge par le* Fragment Shader*, troisième et dernier shader du pipeline programmé par le développeur, qui se charge de donner une couleur finale aux fragments / pixels. Le résultat est écrit en mémoire vidéo, dans une texture ou dans le framebuffer. Le CPU a la possibilité de lire et d'écrire dans cette mémoire, suivant certaines restrictions.*

Pour traiter la géométrie d'une image, les données sont premièrement envoyées au GPU. Ces données sont les primitives géométriques de la scène à rendre, décrite via les commandes d'une API graphique, et représentés sous la forme d'un flux de *vertex*.

Ces vertex sont les données d'entrée du premier étage du pipeline, le *Vertex Shader*. Celui ci a pour vocation de modifier les différents attributs des vertex : il peut les changer et leur en donner de nouveaux, comme couleur et normale. Le flux sortant est de même taille et est composé des

mêmes vertex modifiés. Il a accès à la mémoire de la carte en lecture et peut y écrire ses résultats. Sa principale utilité est de déplacer les objets pour donner l'impression de mouvement d'une frame à une autre.

Le flux sortant du précédent shader est le flux d'entrée pour l'étage suivant du pipeline, le *Geometry Shader*. Ce shader permet de modifier la géométrie de la scène, ici le maillage géométrique passé à la carte. Il peut en effet ajouter ou supprimer des vertex et en modifier les attributs. Le flux de sortie contient un ensemble de vertex de taille potentiellement différente à celle du flux d'entrée. Il a également accès à la mémoire en lecture, et en écrire pour ses résultats. Ce shader est en particulier utilisé pour les méthodes de raffinement de maillage ou d'augmentation de détails sur les modèles géométriques, comme le *Displacement Mapping*, technique permettant de créer le relief en surface d'un objet en déplaçant certains points de cette surface en fonction de valeurs contenues dans une texture de hauteur (*displacement map*).

L'étage suivant dans le pipeline est le *Rasterizer*. A partir du maillage composé des vertex sortant du shader précédent, la géométrie finale est projetée sur une grille de la taille de l'image de sortie, et les primitives géométriques sont divisées en *fragments*. C'est à cet étage du pipeline qu'on réalise également quelques opérations supplémentaires améliorant la vitesse d'exécution des calculs suivants, telles que *clipping* (suppression des objets extérieurs au cône de vision donc non visibles) ou *back-face culling* (suppression de polygones suivant leur orientation par rapport à la caméra : un polygone présentant son "dos" à la caméra n'est pas censé être visible). Les fragments résultants possèdent des coordonnées, correspondant à la position finale dans l'image. Le flux de fragment sortant est ensuite dirigé vers le dernier étage du pipeline.

Le *Fragment Shader* est le troisième et dernier étage programmable du pipeline. C'est le shader le plus important car pour chaque fragment de son flux d'entrée, il lui attribue sa couleur finale, en fonction de l'éclairage, des réflexions et réfractions lumineuses et de diverses techniques de rendu pouvant interroger des textures tierces. Comme les shaders précédents, il peut consulter la mémoire de la carte mais ne peut y écrire n'importe où. Il a seulement l'autorisation d'écrire aux coordonnées du fragment qu'il est en train de traiter, coordonnées qu'il ne peut donc modifier.

Le flux de fragments calculés, ou *shadés*, est écrit en mémoire. Classiquement, la structure accueillant ces données est le framebuffer, directement affiché à l'écran, image ne nécessitant pas un transit supplémentaire par le CPU. Néanmoins, il est possible d'écrire cette image dans une texture, pouvant être utilisée par un autre shader ou récupérée sur CPU.

Jusqu'à la cinquième génération de cartes graphiques, les processeurs embarqués étaient catégorisés, certains servant au traitement de vertex, les *Vertex Units*, d'autres à celui des fragments, les *Fragment Units*. Cela a pour inconvénient de pouvoir créer un *bottleneck* : lorsque les Vertex Units sont surchargés, l'utilisation des Fragment Unit n'est pas optimale et peut même être extrêmement diminuée, et réciproquement. Avec la mise à disposition de la sixième génération de cartes graphiques (GeForce 8), les processeurs ne sont plus spécifiques, ils peuvent maintenant servir aussi

bien à la gestion des vertex qu'à celle des fragments. Cela permet de remplir au mieux le pipeline et d'éviter le bottleneck précédent. L'architecture de ces cartes est dite *unifiée*.

1.4 Comment programmer en GPGPU ?

Adapter un algorithme générique pour un calcul sur matériel spécialisé, tel un GPU, demande de prendre en compte quelques aspects spécifiques de ces architectures.

Originairement, les GPU sont destinés au traitement et à l'affichage de graphismes ; ils manient de façon naturelle vertex et pixels, sont capables de rasterizer des primitives et d'utiliser quelques structures simples de données tout au mieux. Les libertés dont jouissent les développeurs sur CPU, concernant types complexes de données, gestion de mémoire ou richesse des jeux instructions, sont alors drastiquement réduites par les limitations matérielles inhérentes aux architectures de ces cartes. Il a donc été nécessaire de repenser le portage d'algorithmes génériques sur GPU. Un modèle de programmation classique en GPGPU en a émergé, ainsi que différentes astuces largement employées.

Dans cette section, après un tour d'horizon des langages de programmation disponibles sur GPU, nous exposons ce modèle de programmation GPGPU, ainsi que les contraintes et astuces liées à ce type de programmation. Nous continuons avec une rapide présentation de deux langages utilisés en GPU, Cg et CUDA, puis les comparons sur l'implémentation de quelques techniques usuelles de calcul parallèle.

Dans les chapitres 4, 5 et 6, nous détaillerons trois applications de différents domaines et d'importante ampleur, utilisant les méthodes et astuces décrites ci-dessous pour contourner les contraintes de programmation sur GPU afin d'obtenir des gains en vitesse significatifs.

1.4.1 Langages

Le but initialement graphique des GPU a bien entendu dirigé les constructeurs à initialement proposer des langages destinés à ce type de calcul, c'est-à-dire dont les instructions utilisent les branchements matériels spécifiques, destinés à des opérations de type *mult-add*, largement utilisées en synthèse d'images. Même si d'autres langages plus généralistes ont depuis été présentés, le choix reste bien plus restreint sur GPU que sur CPU. On s'intéresse ici à faire un tour d'horizon non-exhaustif des langages existants sur GPU. En plus des langages bas niveau type Assembleur dont il ne sera pas fait mention dans cet exposé, manipulant directement le contenu de registres des unités programmables sur la carte, les langages de plus haut niveaux disponibles sur GPU se distinguent en deux grandes catégories : les *Shading Languages*, dont les instructions sont principalement destinées aux calculs graphiques, et ceux dont l'objectif est le calcul générique, que l'on nommera *langages GPGPU*.

1.4.1.1 *Shading Languages*

Ces langages ont pour point commun de proposer au développeur des jeux d'instructions spécifiques à la génération d'images, ils sont orientés matériel. Ils permettent d'écrire des *shaders*. Ceux ci seront exécutés par

les unités matérielles spécifiques, *Vertex Unit* et *Fragment Unit*, ces deux types d'unités étant unifiés dans les dernières générations de cartes.

Cg, HLSL, GLSL Ces trois langages, respectivement proposés par NVidia [147, 52], Microsoft [178, 154] et 3DLabs [195], ont été développés conjointement et sont donc très similaires. Ce sont les *shading languages* les plus usités. Les syntaxes et sémantiques proposées ont été délibérément fixées proches des instructions du C++, par soucis de simplicité. Les éléments qui ont effectivement motivé ce choix sont la portabilité entre différents systèmes d'exploitation et/ou modèles de GPU et la possibilité d'un développement rapide. Un compromis nécessaire entre ces éléments et une volonté de performance a amené ces langages à posséder des instructions exploitant directement les unités matérielles de la carte, proposées via une syntaxe de type C et permettant de réaliser les opérations de bases et d'autres plus spécifiques de la génération d'images de synthèse. Ces langages sont compilables et multi programmes : le développeur doit écrire un *shader* pour chaque étage programmable du pipeline : *Vertex Shader*, *Geometry Shader* et *Fragment Shader*. Chacun a ses particularités, en particulier les types de données entrant et sortant. L'ensemble de ces shaders est encastré dans un code C++ englobant, pilotant la carte graphique. Les rares différences entre ces langages se résument à deux principaux points. Premièrement, ils ne supportent pas tous les mêmes API : HLSL est exclusivement utilisable avec DirectX, GLSL avec OpenGL, et Cg a la capacité d'utiliser les deux. Le second point est lié à la gestion des différents modèles de cartes : alors que GLSL ne propose pas de sous-ensemble d'instructions dédiés à chaque modèle, Cg et HLSL exploitent cette idée, sous la notion de *profil*, abordée dans la section 1.4.5.2.

Sh Tout comme Cg, HLSL et GLSL, Sh [151], développé par le *Computer Graphics Lab* de l'*University of Waterloo*, est un langage encastré dans le C++, avec une syntaxe proche du C permettant d'écrire des shaders paramétrés. Il est à la fois orienté vers le calcul graphique et la programmation générique. Sa particularité la plus notoire est d'être basé sur le traitement de flux de données. RapidMind [107] est le successeur commercial de Sh, il est utilisé dans un nombre important de domaines nécessitant une certaine puissance : $3D$, traitement de signaux, finance,...

1.4.1.2 Langages GPGPU

L'utilisation des *shading languages* pour de la programmation générique n'est pas des plus aisée. Le développeur les employant est contraint d'adapter ses algorithmes à un modèle de calcul basé sur des primitives géométriques et des textures, ce qui n'est pas toujours facile ni même possible. Des moyens pour programmer les GPU sans pour autant avoir besoin de connaissances en graphisme ont naturellement suscité un intérêt grandissant. Différents langages ont émergés de projets industriels ou universitaires.

Tous ces nouveaux langages ont comme dénominateur commun d'être de plus haut niveau que les *shading languages*, en proposant plus de possibilités et en se séparant totalement des notions graphiques : *texture*, *shader*, *vertex* ou *fragment* n'y ont plus de sens. Les plus importants langages GPGPU sont présentés dans la suite.

CUDA CUDA [25, 168], *Compute Unified Device Architecture*, est le dernier né des langages de NVidia, pouvant être utilisé à partir de la sixième génération de cartes graphiques de ce constructeur (architectures G80 et plus récentes). Quelques unes de ses particularités sont d'être encastré dans du code C++ en en définissant seulement quelques extensions, de mettre à disposition une mémoire partagée et rapide, ou de supporter différents types d'opérations scalaires, en particulier les opérations sur entiers et bit à bit. C'est aussi le premier langage à exploiter l'unification des shaders sur les architectures G80 de NVidia. Dans le domaine du traitement d'images, il est de plus en plus employé ; Young et Jargstorff proposent dans [254] quelques implémentations astucieuses d'algorithmes classiques de ce domaine. Une présentation plus détaillée de CUDA est proposée dans la section 1.4.6.

OpenCL Le très récent *OpenCL* [164, 234, 165] (*Open Computing Language*) a été annoncé par *Apple* au sein du *Compute Working Group*, formé par le *Khronos Group* (regroupant *3DLabs*, *Apple*, *AMD*, *NVidia*, *ARM*, *Ericsson* et d'autres universitaires et industriels). Ces partenaires souhaitent mettre à disposition un langage *open source*, dans la veine d'OpenGL [214] et OpenAL [39] et ont pour ambition de faire d'OpenCL le standard libre pour le calcul GPGPU, avec les importants avantages d'être multiplateforme (cartes AMD/ATI et NVidia) et de permettre une programmation homogène

BrookGPU, Brook+ *BrookGPU* [82] est une implémentation GPU du langage *Brook*, tous deux développés par le *Stanford University Graphics Lab*. C'est un langage basé sur la gestion de flux de données. AMD/ATI a également proposé *Brook+*, une amélioration de BrookGPU pouvant être utilisé uniquement sur leurs cartes. *Folding@home*, projet mondial de calcul distribué simulant les repliements de protéines dans le but d'en tirer des solutions médicales, utilise en partie Brook+.

Scout Proposé par le *Los Alamos National Laboratory*, *Scout*[153, 152] est un langage GPGPU destiné à l'analyse et à la visualisation scientifique, dont beaucoup de techniques sont basées sur l'utilisation de mappings, exprimés sous forme de fonctions mathématiques et transformant des données en image affichables. Scout a notablement été utilisé pour la simulation et la détermination de caractéristiques du courant côtier *El Ninõ*.

Accelerator *Microsoft Research* a présenté *Accelerator* [226], destiné à simplifier la programmation GPGPU en fournissant un modèle de calcul parallèle accessible simplement à travers d'autres langages. Les opérations parallèles y sont compilées à la volée et optimisées pour les *fragment shaders*.

CGis Le langage *CGiS* [60], développé par l'*Universität des Saarlandes*, est un autre langage parallèle, similaire à Brook et Accelerator, manipulant également des objets de type flux de données, mais se démarquant par l'absence de notion de *kernel computationnel* (équivalent de *shader* pour du calcul uniquement GPGPU), remplacé par un mécanisme de boucle globale `forall`, chère au calcul parallèle.

1.4.2 Modèle de programmation

La programmation GPGPU est basée sur quatre concepts principaux, que nous proposons d'expliciter. On évoque également deux autres façons, liées, d'interpréter la notion de complexité en calcul sur GPU.

1.4.2.1 Tableaux = Textures

Sur CPU, une des structures de données la plus utilisée est le tableau $1D$. Les tableaux de dimensions supérieures sont représentés dans un tableau $1D$, les données y étant stockées à la suite pouvant être accédés par des offset sur leurs indices.

En GPU, les données exploitables sont nécessairement stockées dans des *textures* (donc en $2D$), on y accède par leurs *coordonnées*. Pour y représenter des structures d'autres dimensions, il est utile de passer par une réorganisation vers un tableau $2D$, transmis à la carte dans une texture. Le GPU peut alors lire l'élément (i, j) du tableau en consultant le pixel (i, j) de cette texture.

Dans le cas d'un tableau $1D$ stocké dans une texture de taille $N \times N'$, l'élément k a pour coordonnées dans la texture $(k \mod N, E(\frac{k}{N}))$, où $E(x)$ est la fonction partie entière de x.

Les tailles maximales pour les textures sont de 8192×8192 avec l'API DirectX, ou bien de 4096×4096 pour les versions moins récentes. Bien qu'il n'y ait *a priori* pas de contrainte sur le nombre de textures instanciées simultanément, c'est souvent la quantité de mémoire sur la carte qui va limiter ce nombre, s'élevant aujourd'hui jusqu'à 768Mo par carte grand public, ou 1Go pour des cartes professionnelles.

Chaque élément de texture peut contenir jusqu'à 4 scalaires (correspondant aux canaux rouge, vert, bleu et alpha d'un pixel à afficher). Les opérations scalaires se font simultanément sur ces 4 valeurs. Les types de scalaires utilisables pour les éléments de texture peuvent être entiers ou flottants. Ces éléments peuvent être vectoriels de dimension 1 à 4, voire matriciels de dimensions allant jusqu'à 4×4.

1.4.2.2 Kernel = Fragment Shader

La programmation parallèle a introduit la notion de *kernel computationnel*, brique calculatoire de base, équivalent d'une fonction mathématique, pouvant être appliqué à un ensemble de données de façon parallèle.

En programmation GPGPU, ces kernels sont les fragment shaders. Un tel shader comporte donc une suite courte d'instructions opérant sur une donnée (ou un petit ensemble de données). Il est à noter qu'aucun vertex shader ou geometry shader n'est en général implémenté. Il est possible de ne pas spécifier de shader, les unités de traitement correspondantes se comportant alors comme des *passthrough*, ne modifiant aucun attribut des données.

1.4.2.3 Calcul = Rendu graphique

Une fois le fragment shader implémenté, une fois l'environnement convenablement préparé (textures créées aux bonnes dimensions, données d'entrée envoyées à la mémoire GPU, paramètres supplémentaires assignés, environnement OpenGL ou DirectX correctement initialisé,...),

l'exécution du calcul décrit dans le fragment shader se fait en utilisant la totalité du pipeline graphique, par invocation du rendu d'un simple rectangle de taille adaptée aux données de sortie. En effet, un tel rectangle rasterizé est comparable à une grille de taille fixée, structure la plus pratique à disposition pour la réalisation de calculs parallèles. Cela justifie la non-modification des sommets, et donc la non-utilisation des vertex et geometry shaders. Ces étapes sont résumées sur la figure 1.6. Harris présente également quelques principes de ce type de programmation dans [89].

Le lancement d'un rendu avec un shader est également appelé *passe* du shader.

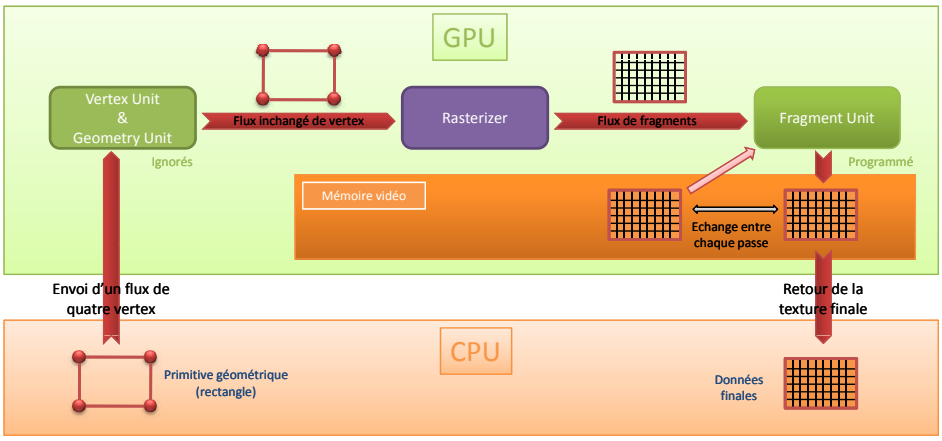


FIGURE 1.6 – *Modèle classique de programmation GPGPU : les quatre sommets d'un rectangle sont passés à la carte via l'API de son choix. Le vertex shader et le geometry shader ne les modifient pas. Le rasterizer divise le rectangle en fragments. Chaque fragment est traité par un des fragment unit. Le résultat est enregistré dans une texture de sortie, de la taille du rectangle passé en entrée. Cette texture peut être réutilisée dans une passe ultérieure du fragment shader, ou bien être récupérée par le CPU.*

1.4.2.4 *Feedback*

Lors d'un calcul destiné à faire un affichage à l'écran, l'image générée est stockée dans le *framebuffer*, buffer dédié à cet effet. Pour un calcul GPGPU, le résultat du calcul n'est pas stocké dans ce *framebuffer*, mais dans une texture, par un moyen technique nommé *Render-To-Texture*, RTT [248].

Cette texture peut alors être utilisée comme donnée d'entrée pour une passe ultérieure, permettant ainsi de segmenter tout algorithme à implémenter en GPU en différentes passes. Lorsqu'un algorithme nécessite plusieurs passes successives, une astuce simple et classique, appelée *ping-pong*, est d'utiliser une paire de textures, une pour l'entrée de l'algorithme, une pour sa sortie, que l'on permute entre chaque rendu. Cela permet de ne pas avoir à transférer les données du GPU vers le CPU et réciproquement entre chaque passe.

1.4.2.5 Complexité GPGPU

La complexité d'un algorithme parallèle GPU se mesure différemment d'un algorithme séquentiel classique. On peut vouloir se reposer sur le

nombre total de fragments à calculer, on parle alors de *complexité en fragment* C_f. On peut également vouloir déterminer le nombre de rendus nécessaires pour obtenir le résultat final, c'est alors la *complexité en rendus* C_r. Ces deux complexités sont liées par un facteur multiplicatif, la première étant égale à la seconde multipliée par la taille N des données : $C_f = C_r \times N$.

1.4.3 Contraintes matérielles

Connaître le modèle de programmation des GPU n'est pas suffisant pour produire un programme efficace et optimisé. Il est également nécessaire de prendre en compte les contraintes imposées par l'architecture matérielle des cartes graphiques.

1.4.3.1 Accès mémoire

Aptitude à supporter les opérations de Gather et Scatter Sur GPU, les processeurs sont capables de retrouver en mémoire des informations ailleurs que sur le pixel i_0 sur lequel ils s'exécutent, ils sont donc capables de *gather*. Ce procédé est nécessaire pour la mise à jour de valeurs en fonction de celles des voisins spatiaux, par exemple. Par contre, le *scatter* leur est impossible, ils ne peuvent pas modifier un autre pixel que i_0. Matériellement, les processeurs des GPU ne permettent pas le *scatter*, celui ci étant beaucoup plus difficile à implémenter de par la conception des fragments, ayant chacun une localité dans la texture de sortie et ne pouvant donc pas être adressé indirectement en écriture.

Cette restriction implique d'importantes incommodités de programmation, en particulier dans l'adaptation d'algorithmes devant propager des données dans différents endroits de la mémoire, par exemple pour des mises à jours de valeurs en fonction d'une certaine connectivité. Le développeur est alors amené à utiliser diverses astuces [180] : redéfinition de l'algorithme en plusieurs passes GPU, utilisation du *vertex shader* et/ou du *geometry shader* comme outil de *scatter* ; taguer les données avec les adresses mémoires et faire un rendu simple pour ensuite trier selon ces adresses ; reformuler le problème et les algorithmes en termes de *gather*, comme nous le verrons dans le chapitre 4.

De plus, les toutes dernières générations de cartes graphiques, en particulier à partir des GeForce 8, autorisent nativement les opérations de scatter avec certains langages (comme CUDA), par une refonte de l'organisation des processeurs et de la mémoire, détaillée dans la section 1.4.6.

Texture Indirection Une *texture indirection* est une lecture dans une texture à des coordonnées dépendant d'un calcul ou d'une autre lecture dans une texture. Le nombre de *texture indirections* successives par programme est limité, voire très limitée sur les anciennes cartes ATI, pour des raisons internes au développement des cartes. La création de structures de type *look-up table*, ou *table de correspondance*, est ainsi parfois incommode.

1.4.3.2 Bande passante

L'envoi des données au GPU ainsi que le retour des résultats sont des opérations couteuses en temps. Bien que la communication entre la

carte graphique et le reste du système soit assurée par des ports dédiés (actuellement *AGP* et *PCI Express*), cela reste insuffisant par rapport à la rapidité d'échanges d'informations entre la mémoire de la carte et ses processeurs, faisant de la communication entre CPU et GPU le goulet principal. De plus, les ports *AGP* ont le désavantage d'être asymétrique, le temps GPU vers CPU étant plus long que son contraire. Il est alors important de minimiser ces transferts, en conservant les données dans la mémoire de la carte le plus longtemps possible. Ceci est d'autant plus avantageux lorsque le CPU n'a pas à attendre le résultat d'une passe GPU.

1.4.3.3 Autres

D'autres difficultés liées au matériel existent, bien que moins embarrassantes aujourd'hui.

Taille des programmes Avec la norme *Pixel Shader* 2.0, le nombre d'instruction par shader était limité à 96. Ce plafond est passé à 65535 avec la norme *Pixel Shader* 3.0. Ces quotas étaient parfois très vite atteints, après déroulages de boucles, restreignant les développeurs. La norme actuelle, utilisée par les cartes graphiques récentes (sixième génération et plus), *Pixel Shader* 4.0, autorise des shaders de taille quelconque. Néanmoins, lors du développement d'un programme devant également s'exécuter sur d'anciennes générations de cartes, il est important de prendre en compte ces limitations.

Formats et précisions En interne, les formats et la précision des données ne sont pas constants entre les générations de cartes. Les générations plus anciennes proposent des calculs sur flottants en 16bits, voire 8bits, et les plus récents sur 32bits, flottants ou entiers (en particulier Geforce 8 et plus récents). Une fois encore, un développeur souhaitant exécuter un programme sur différentes cartes devra s'assurer de la compatibilité des formats de données utilisés par son programme avec ceux disponibles sur la génération de cartes utilisée.

1.4.4 Astuces de programmation

Nous présentons ici quelques utilisations astucieuses de propriétés matérielles disponibles sur les cartes graphiques actuelles et plus anciennes, permettant un gain en facilité ou en temps de calcul et largement employées en programmation GPU avec *shading language*, généraliste ou non.

1.4.4.1 Stencil Buffer

Le *Stencil Buffer* est un buffer disponible sur la carte, binaire, de la taille de l'image de sortie, permettant de spécifier un *masque* de rendu : seuls les pixels (x, y) de l'image de sortie qui sont non masqués, c'est-à-dire pour lesquels la valeur en (x, y) dans le Stencil Buffer vaut 0, seront calculés. L'application principale est de limiter la zone de rendu d'une image.

L'avantage apporté au calcul généraliste est qu'il est facile de limiter le calcul à une partie des données d'origine, amenant à un gain de temps pouvant être considérable.

1.4.4.2 Z-Buffer et Early Z-Cull

Le *Z Buffer* est un autre buffer disponible sur la carte, de la taille de l'image de sortie, contenant des valeurs réelles correspondant à la profondeur des objets dans la scène $3D$. Il permet, par un test de profondeur, de gérer les problèmes de visibilité consistant à déterminer quels objets doivent être rendus, lesquels sont cachés par d'autres, et dans quel ordre doit se faire l'affichage.

Le *Early Z-Cull* est une méthode utilisant le Z Buffer et les tests de profondeur pour abandonner certains fragments avant même qu'ils soient traités par le fragment shader. Le fait de pouvoir mettre de côté les fragments non pertinents avec le calcul en cours apporte un substantifique gain de temps pour l'exécution globale. Des utilisations avancées et conjointes du Z-Buffer et du Stencil Buffer permettent entre autres un large nombre d'effets graphiques, tels ombrages (par *Shadow volume* [48]) ou réflexions lumineuses.

Dans une optique GPGPU, on utilise le Z Buffer en y stockant une valeur artificielle qui ne représente pas la profondeur mais quelque chose d'adapté au calcul que l'on souhaite réaliser. On adapte ainsi le test de profondeur câblé matériellement (donc très rapide) à un test généraliste, applicable sur l'ensemble de nos données.

1.4.4.3 Instruction `discard()`

Tout comme le Early Z-Cull, l'instruction `discard()` va permettre de mettre de côté des fragments dont la détermination n'est pas pertinente. Cette instruction est utilisable à l'intérieur des shaders, permettant ainsi de mettre de côté un fragment après un test dépendant d'un calcul préalable, interne au shader (et non d'une valeur externe comme pour le Z Buffer), pouvant apporter un gain de temps substantiel. Un fragment ayant subi cette instruction `discard()` n'est pas considéré par le pipeline graphique comme rendu.

1.4.4.4 Occlusion Queries

Les *Occlusion Queries* sont un mécanisme permettant de questionner la carte graphique sur le nombre de fragments rendus lors du dessin d'une primitive ou d'un groupe de primitives géométriques. Traditionnellement, cela est utilisé pour déterminer la visibilité d'un objet.

Il est souvent intéressant, particulièrement en GPGPU, de savoir combien de fragments ont été rendus. Employé conjointement avec l'instruction `discard()` utilisée sur des fragments que l'on ne souhaite pas compter, il est facile et rapide de déterminer ce nombre, pouvant servir de critère d'arrêt pour nombre d'algorithmes.

1.4.5 Introduction à Cg

Comme annoncé dans la section 1.4.1.1, Cg est un *shading language*, développé par NVidia et initialement destiné aux calculs et rendus gra-

phiques de scène, ce qui comprend la détermination de la géométrie des objets décrits aussi bien que leur illumination. Il permet de spécifier les différents shaders pilotant les unités programmables des cartes, comme vu dans la section 1.3.2.2. Ce sont des instructions des API (OpenGL ou DirectX) qui permettent au sein du programme CPU l'initialisation des données, des paramètres et de l'environnement nécessaire au lancement des shaders Cg.

1.4.5.1 Structures de données et types de variables disponibles

Parce que Cg opère spécifiquement sur des vertex et des fragments, données élémentaires en graphisme 3D, il ne propose pas de manier des structures complexes ou des notions d'abstraction comme la programmation orientée objet. Il ne connait pas la notion de pointeur, ne propose pas d'allocation de zones en mémoire et n'autorise pas le passage en paramètre de structures simples, au sens C. Il est néanmoins possible de définir des structures à l'intérieur de programme Cg.

De plus, Cg sait manier les structures de données les plus usitées en graphisme (tableaux, vecteurs, matrices de taille maximale 4×4), et connait les opérations mathématiques et géométriques fondamentales à ce domaine (fonctions d'interpolation, produit scalaire, fonctions trigonométriques et hyperboliques, multiplication matricielles, calcul de distance, rayon réfléchi, consultation de textures...). Boucles et appels de fonctions sont disponibles, les boucles pouvant ou non être déroulées et les fonctions étant inlinés, pour des raisons de performance. Les branchements conditionnels sont également utilisables, à éviter au possible, étant donné le caractère SIMD vectoriel des GPU (voir section 1.2.1) obligeant à attendre qu'une partie du code ait été exécuté par toutes les unités de calcul : les différents chemin conditionnels sont exécutés séquentiellement pour l'ensemble des unités de calcul, et non parallèlement.

1.4.5.2 Profil

Cg inclut la notion de *profil matériel*, à spécifier lors de l'écriture du programme en Cg. Chaque profil correspond à l'association d'une architecture GPU et d'une API, apportant des limitations sur le nombre maximal d'instructions, le jeu d'instructions disponible, le nombre maximal de textures,... Ce mécanisme va à l'encontre de la programmation générique sur CPU, où un programme peut se compiler, à quelques restrictions près, avec n'importe quel CPU. Ici, chaque programme est spécifique. La raison de telles limitations est historique, les premières architectures GPU acceptant Cg ne supportant pas toutes les mêmes caractéristiques. Néanmoins, malgré les apparentes limitations que cette notion apporte, son intérêt est de forcer le développeur à limiter la taille des programmes : plus ceux ci sont petits, plus ils s'exécutent rapidement, ce qui est primordial en graphisme temps-réel. De plus, ces limitations portent sur le matériel disponible et non sur le langage Cg.

Les profils actuels et futurs sont et seront des super-ensembles de ces anciens profils, permettant une rétrocompatibilité complète de ces anciens profils avec les prochaines générations. Lors de ses prochaines évolutions, Cg permettra des opérations de plus en plus complexes.

1.4.5.3 Exemples de code

Nous présentons ici deux exemples de programme Cg, le premier pour introduire l'aspect graphique auquel est destiné Cg, le second pour illustrer les capacités GPGPU de ce langage.

Exemple graphique Ce programme simple interpole les couleurs des sommets d'une primitive géométrique en $2D$, ici un triangle, puis l'affiche. Il utilise pour cela un vertex shader et un fragment shader.

Il est adapté d'un tutoriel complet disponible dans [52].

Pour réaliser cela, le code du vertex shader, écrit dans le fichier `VSPosCol.cg`, est le suivant :

```
/***********************************/
/*          VS: Varying            */
/*                      VSPosCol.cg */
/***********************************/

// Structure de sortie, contenant les paramètres
//  de chaque vertex
struct VSOut {
  float2 position : POSITION;   // Position
  float4 color    : COLOR;      // Couleur
};

/* Fonction principale */
VSOut VSmain( float2 position : POSITION,
              float4 color    : COLOR)
{
  // Déclaration de la structure de sortie
  VSOut OUT;

  // Assignation des paramètres de sortie
  OUT.position = position;
  OUT.color    = color;

  // Retour
  return OUT;
}
```

Il est possible en Cg de définir des structures comme en C++. Ici, la structure définie est utilisée comme sortie de la fonction principale, et contient deux vecteurs :

- `position`, de dimension 2, contenant la position $2D$ du vertex traité ;
- `color`, de dimension 4, représentant sa couleur, codée en RGBA.

La fonction principale s'exécutant sur chaque vertex du flux d'entrée prend comme paramètres deux vecteurs identiques aux précédents

Chacun de ces vecteurs est associé à une *sémantique*, ici `POSITION` ou `COLOR`. Les sémantiques sont ce qui permet de lier à une variable du programme une caractéristique d'un flux de données. Ici, Les paramètres d'entrée sont associés aux positions et couleurs des vertex passés par l'API graphique (donc par le programme principal), et les paramètres de sortie sont associés aux positions et couleurs des vertex du flux calculé. Ces données pourront être récupérées dans les étages suivants du pipeline.

Le calcul des paramètres de sortie est ici canonique : on associe les sorties aux entrées, les caractéristiques du flux d'entrée sont donc directement recopiées dans le flux de sortie. En bref, notre vertex shader ne fait rien. On aurait pu laisser le vertex shader par défaut, ou bien modifier la couleur ou la position par un calcul. Dans la suite, comme annoncé dans la section 1.4.2.2, on n'utilisera plus de vertex shader (ni de geometry shader).

Le fragment shader utilisé, dans le fichier `FSPassthru.cg` présenté ci-dessous, se comporte en *pass-through*, ne faisant rien d'autre qu'assigner à chaque fragment du flux sortant la couleur récupérée dans le flux d'entrée. C'est le comportement par défaut du GPU lorsqu'aucun fragment shader n'est spécifié.

```
/***********************************/
/*        FS: Passthrough          */
/*                    FSPassthru.cg */
/***********************************/

// Structure de sortie, contenant les paramètres
//  de chaque fragment
struct FSOut {
  float4 color : COLOR; // Couleur finale (unique paramètre)
};

// Fonction principale
FSOut FSmain(float4 color : COLOR)
{
  // Déclaration de la structure de sortie
  FSOut OUT;

  // Assignation de la couleur
  OUT.color = color;

  // Retour
  return OUT;
}
```

La structure de sortie ne contient plus qu'un seul paramètre, représentant la couleur finale du fragment. En effet, le flux de vertex sortant du vertex shader a été transformé en flux de fragments par le rasterizer. Il est à noter que dans le cas général, ce shader n'est pas un pass-through, c'est même lui qui réalise les calculs les plus lourds.

Par soucis de clarté, le programme principal C++, contenant les initialisations de Cg et des données et paramètres du programme, les fonctions de transferts de ces données et les instructions de rendu, n'est pas présenté ici (un tutoriel d'initialisation Cg est proposé dans [40]). En assumant que ce programme envoie à la carte graphique un triangle dont les sommets sont colorés respectivement en rouge, vert et bleu, on obtient le résultat présenté figure 1.7. Chacun des pixels voit sa couleur interpolée entre celles des sommets de ce triangle, suivant sa position. On note que seuls les pixels du triangle ont été traités et non ceux du fond de l'image, dont la couleur avait au préalable été fixée à blanc.

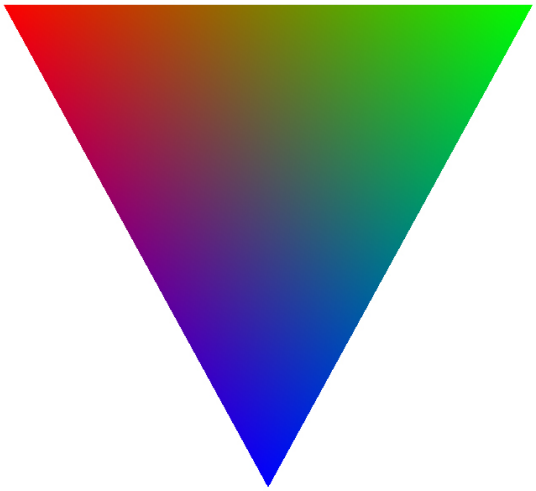


FIGURE 1.7 – *Chaque pixel du triangle rendu s'est vu assigné une couleur correspondant à l'interpolation, sur sa position dans le triangle, des couleurs appliquées aux sommets.*

Exemple GPGPU Le second exemple présente l'addition de deux matrices carrées de taille $N \times N$: $C = A + B$. Suivant les principes de programmation GPGPU explicités dans la section 1.4.2,

- le rendu d'un carré de taille $N \times N$ pixels nous assure d'avoir en sortie un tableau de la bonne taille ;
- il n'est pas nécessaire de spécifier un vertex shader car le calcul est pris en charge par le fragment shader ;
- enfin, l'écriture du résultat se fait dans une texture et non dans le framebuffer.

On suppose que le programme principal C++ initialise le GPU pour un rendu dans une texture d'une primitive adéquate : un rectangle de la taille de la texture de sortie (ainsi, un pixel de cette texture correspondra exactement à un fragment, facilitant la structuration des données).

Un fragment shader possible implémentant ces opérations est le suivant :

```
1  /*****************************/
2  /*      FS: AddMatrices      */
3  /*****************************/
4  // Structure de sortie, contenant la valeur
5  //  de chaque élément de la matrice de sortie
6
7  struct FSOut {
8    float value : COLOR;  // Valeur finale
9  };
10
11 // Fonction principale
12 FSOut FSAddMatrices(in float2 coords : TEXCOORD0,
13                      uniform samplerRECT hA,
14                      uniform samplerRECT hB)
15 {
16   // Déclaration de la structure de sortie
17   FSOut OUT;
18
19   // Lecture des éléments des matrices d'entrée
20   float valueA = f1texRECT(hA, coords);
21   float valueB = f1texRECT(hB, coords);
```

```
22
23    // Calcul de la somme
24    OUT.value = valueA + valueB;
25
26    // Retour
27    return OUT;
28  }
```

Semblablement à l'exemple précédent, la structure de sortie contient une variable `value` associée à la sémantique `COLOR`. Ainsi, la couleur d'un fragment (i, j) de la texture de sortie représente la valeur de l'élément (i, j) de la matrice résultat.

La fonction principale `FSAddMatrices()` n'a pas ici besoin de connaitre la couleur interpolée des fragments, assignée par le rasterizer. Par contre, la connaissance des coordonnées est nécessaire pour l'interrogation des matrices en entrée, d'où la variable d'entrée `coords`.

Ces matrices sont représentées en mémoire par des textures A et B. L'accès à ces textures se fait via des *handlers*, ici les paramètres `hA` et `htB`, de type `samplerRect` (désignant un handler sur une texture rectangulaire) et déclarées `uniform`. Le qualificatif `uniform` spécifie que la valeur initiale de la variable dont il est question provient de l'extérieur du programme Cg, soit de l'environnement.

Par un appel à la fonction `f1texRECT()`, on consulte la valeur contenue dans la première matrice/texture A via le handler `hA` aux coordonnées `coords`. Il existe d'autres fonctions de consultation des textures, à utiliser en fonction du type de texture en présence.

La somme des éléments des matrices en entrée est stockée dans la structure de sortie, qui est alors retournée, écrivant le résultat dans la texture de rendu C aux bonnes coordonnées. Le programme principal C++ s'occupe de la récupération de cette texture sur CPU.

Comme pour l'exemple précédent, ce programme principal n'est montré pour des soucis de clarté. Néanmoins, nous en donnons la structure principale. Dans le `main()` C++ :

1. Initialisation des trois textures A, B et C ;
2. Transfert vers le GPU des textures A et B ;
3. Définition des options des samplers (pas d'interpolation,...) ;
4. Appel d'un rendu de primitive sur GPU, de la taille de la texture de sortie C ;
5. Transfert vers le CPU de C.

1.4.6 Introduction à CUDA

CUDA, pour *Computer Unified Device Architecture*, est à la fois un framework et un langage de programmation de NVidia, proche du C++, permettant d'exploiter les capacités des GPU d'architecture G80 et plus de NVidia (à partir des GeForce 8800 et déclinaisons), et exploite les ressources matérielles des GPU d'une façon différente à Cg, en particulier en ce qui concerne la gestion de la mémoire et l'organisation des traitements. C'est aussi le premier langage à exploiter l'unification des shaders : les processeurs de la carte ne sont pas différentiés en unités pour le traitement des vertex ou des fragments, chacun de ceux ci peut être assigné à n'importe quelle tâche.

Avec l'arrivée de CUDA, NVidia a dévoilé, à propos de ses cartes récentes, certaines caractéristiques qu'il n'était pas nécessaire de connaitre auparavant. En CUDA, certaines prennent un sens, c'est le cas de l'organisation des processeurs sur la carte, ainsi que celle de la mémoire disponible par processeur.

1.4.6.1 Structures de données et variables disponibles

Etant proche du langage C++, CUDA permet au développeur d'utiliser et de passer en paramètre aux programmes écrits un large panel de structures, en particulier les pointeurs ou les tableaux à plusieurs dimensions, ce qui était impossible en Cg. L'allocation de structures est par exemple possible dans différentes zones de la mémoire, partagée à différents degrés (voir section 1.4.6.5).

1.4.6.2 Organisation matérielle

Dans la section 1.3.2.2, il a été dit que les processeurs d'un GPU se répartissent en deux catégories : vertex unit et fragment unit. Ceci était un abus de langage concernant l'architecture G80 de NVidia, dite *unifiée* : tous les processeurs sont identiques et peuvent exécuter toutes les instructions destinées à ces traitements. Le langage CUDA exploite cette caractéristique, se détachant totalement des notions de vertex, maillage et fragment.

CUDA tire également parti de la disposition des processeurs des architectures unifiées (G80, G92 et G200 pour l'instant). Sur les architectures G80 et G92, ces processeurs sont répartis en 16 groupes de 8 processeurs, ces groupes étant nommés *multiprocesseurs*. Sur les architectures G200, il existe 10 *grappes* de 3 multiprocesseurs de 8 processeurs. Ce type de regroupement permet de faciliter les échanges pour un travail à haute fréquence (jusqu'à 1500Hz) des processeurs d'un multiprocesseur.

Il est à noter que cette organisation matérielle n'est pas en opposition avec le modèle de pipeline graphique présenté dans la section 1.3.2.2, c'est ce pipeline graphique qui repose sur cette organisation. En effet, dans un schéma de programmation graphique, ou GPGPU avec *shading language*, chaque processeur est dédié soit au traitement des vertex, soit à celui de la géométrie, soit à celui des fragments, rôle qui peut varier d'un calcul à l'autre. Dans ce cas, ils sont contrôlés par les shaders correspondants. Cela justifie l'abus de langage réalisé.

1.4.6.3 Kernels

Au contraire de Cg, CUDA ne dépend d'aucune API graphique tierce. Il propose sa propre API haut niveau, consistant en quelques extensions au langage C et dont la prise en main ne nécessite pas de connaissances approfondies dans le domaine graphique. En effet, CUDA est exclusivement dédié au calcul GPGPU : texture, fragment et pixel n'y ont pas de réelle signification. Les données ne sont plus stockées dans des textures, mais dans des flux, classiquement des tableaux. Les routines pilotant les processeurs sont des *kernels* comme définis dans la section 1.4.2.2, appliqués à tous les éléments du flux entrant ; ce sont les équivalents des *fragment shaders*. L'exécution du code se fait par une *invocation de kernel*,

équivalent d'un *rendu* en Cg, et écrivant ses résultats dans un flux de sortie.

1.4.6.4 Organisation des threads

L'exécution de programmes CUDA repose sur la notion de *thread*, similaire à celle de threads sur CPU, processus légers pouvant s'exécuter en parallèle. Ces threads sont regroupés en *warp*, des ensembles de 32 threads ; c'est la taille minimale que le GPU peut traiter en SIMD. En pratique, le développeur ne manipule pas ces warps, mais des *blocs*, qui sont également des réunions de 64 à 512 threads (de 2 à 16 warps), organisés dans un espace de une à trois dimensions. Ces blocs sont enfin regroupés par *grid*, ou grille, dans un espace de même type. Tous les threads contenus dans une grille sont pilotés par un même kernel. Une invocation de kernel est donc une exécution d'un kernel sur tous les threads d'une grille. Le schéma 1.8 résume l'organisation de ces groupements.

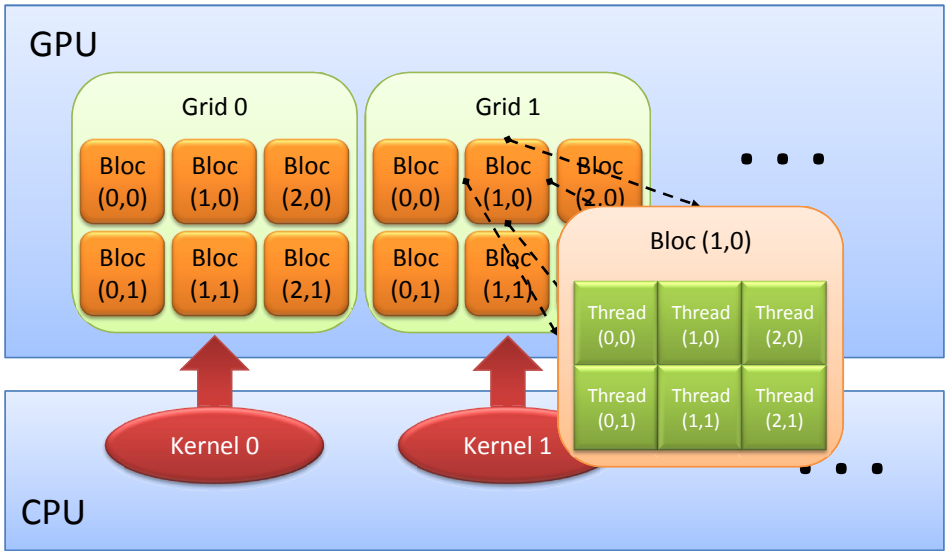


FIGURE 1.8 – *Organisation des unités logiques de traitement pour CUDA. Les blocs sont ici organisés en deux dimensions dans chaque grille, tout comme les threads le sont dans chaque bloc.*

La communication entre les threads d'un même bloc est permise par une mémoire partagée par le bloc. Il est possible de synchroniser tous les threads d'un même bloc.

1.4.6.5 Organisation mémorielle

Chaque processeur, exécutant un thread, a un accès physique à plusieurs endroits mémoriels. Les deux plus larges zones mémoire sont les mémoires locale sur GPU et globale sur CPU, les processeurs du GPU ayant en effet accès à une partie de la mémoire du reste du système. Ces accès sont lents (entre 200 et 300 cycles d'horloge) et ne sont de préférence utilisés que lorsqu'ils peuvent être masqués par d'autres calculs. Une partie de la mémoire globale du GPU peut être mise en cache sur un multiprocesseur, et accessible aux huit processeurs de ce multiprocesseur ; ce sont les caches pour les constantes et pour les unités de texture, de 8Ko chacun, accessibles uniquement en lecture. Chaque multiprocesseur propose également une mémoire partagée entre ses processeurs, de 16Ko,

permettant l'échange d'informations rapidement entre eux et économisant la bande passante. Cette mémoire n'est cependant partagée qu'entre les threads d'un même bloc. Enfin, chaque processeur dispose de quelques registres qui lui sont propres. La figure 1.9 résume cette organisation des espaces mémoire utilisée en CUDA.

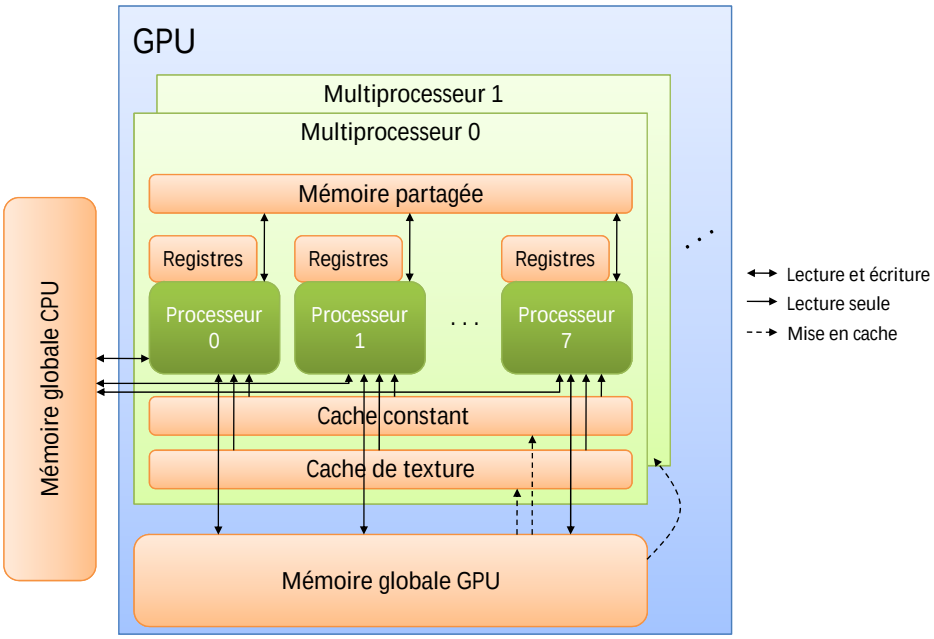


FIGURE 1.9 – *Chaque processeur a accès à ses propres registres, en plus d'une mémoire partagée et de caches des mémoires constante et de texture, sur le multiprocesseur auquel il appartient. Un accès à la mémoire globale est effectuable, mais lent.*

Ce schéma mémoriel est l'un des atouts majeurs de CUDA, car les différents accès en lecture et écriture à ces zones autorisent les opérations de scatter, permettant des portages beaucoup plus simples d'algorithmes sur GPU.

1.4.6.6 Exécution

Le nombre de blocs pouvant être exécuté simultanément est conditionné par l'architecture présente, un modèle de GPU disposant de plus ou moins de *multiprocesseurs*. L'inclusion d'un grand nombre de blocs dans des grilles permet de s'abstraire de cette contrainte matérielle et d'exécuter un kernel sur un grand nombre de threads en une seule invocation. De plus, CUDA se charge de répartir les ressources disponibles : sur les GPU disposant d'un nombre important d'unités de traitement, les blocs de threads seront exécutés simultanément sur ces unités ; dans le cas contraire, une exécution séquentielle aura lieu. Chaque grille est assignée à un multiprocesseur. Les 16Ko de mémoire du multiprocesseur sont donc partagés par tous les blocs que ce multiprocesseur exécute. Ce qui reste à la charge du développeur est l'arrangement des threads par bloc et des blocs par grille, en en spécifiant les nombres, généralement dépendant de la taille des données en entrée plutôt que du nombre de processeurs disponibles sur la carte.

Le modèle de programmation en CUDA est un une extension du modèle GPGPU présenté en section 1.4.2 : ce qu'il est possible de faire en

programmation GPGPU classique (avec un shading language) est *a priori* possible en CUDA.

1.4.6.7 Exemple de code

Programmer en CUDA, c'est écrire un ou plusieurs kernels et les invoquer à partir d'un programme principal. L'exemple canonique de démonstration du fonctionnement de ce langage est l'addition de deux matrices, que nous prendrons ici carrées. Nous présentons donc ici l'implémentation d'un kernel et du code C++ associé.

Le kernel se définit comme une fonction C++ standard, précédé de la déclaration `__global__`, indiquant que la fonction est à exécuter sur le GPU et non le CPU. Dans notre cas, on peut l'écrire de cette façon :

```
__global__ void add_matrix(float A[N][N],
                           float B[N][N],
                           float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}
```

Le kernel `add_matrix` prend en argument trois matrices carrées de taille N, `A` et `B` étant les deux matrices à additionner dans `C`.

Chaque thread lancé est doté d'un identificateur, `threadIdx`, a accès à sa position dans le bloc courant, `blockIdx`, et aux dimensions de ce bloc, `blockDim`. Les variables `threadIdx`, `blockIdx` et `blockDim` sont des vecteurs de dimension 3, car les threads et les blocs peuvent être organisés sur une, deux ou trois dimensions. Cela permet le lancement d'un calcul sur des vecteurs, des matrices ou des champs tensoriels de façon naturelle.

`i` et `j` représentent ici les indices des éléments à additionner des matrices `A` et `B`. Le résultat est stocké à la même place dans la matrice `C`. `i` et `j` sont calculés en fonction de l'identifiant du thread courant, mais également de l'identifiant du bloc contenant ce thread : on va en effet assigner à chaque bloc le traitement d'une partie déterminée des matrices, ce qui nécessite pour chaque thread de savoir dans quel bloc il s'exécute.

L'invocation du kernel se fait dans le reste du programme C de la façon suivante :

```
int main()
{
  // Taille des matrices
  int N = 4096,

  // Déclaration de matrices A, B et C
  float A[N][N];
  float B[N][N];
  float C[N][N];

  // Remplissage des matrices A et B
  /*  ...  */

  // Invocation du kernel
```

```
15   dim3 dimBlock(16, 16);
16   dim3 dimGrid( (N + dimBlock.x - 1) / dimBlock.x,
17                 (N + dimBlock.y - 1) / dimBlock.y);
18   add_matrix<<<dimGrid, dimBlock>>>(A, B, C);
19 }
```

Les deux variables `dimGrid` et `dimBlock` permettent respectivement de spécifier les dimensions de la grille et des blocs sur lequel le calcul va s'exécuter. Ici, un bloc peut exécuter $16 \times 16 = 256$ threads, et la grille est choisi avec assez de blocs pour avoir un thread par élément de matrice. Le nombre de threads par bloc est généralement induit par la taille des données à traiter plutôt que par le nombre de processeurs disponibles, qui est la plupart du temps largement dépassé.

L'invocation du kernel se fait comme un appel de fonction en C++, à la différence près qu'on spécifie en plus les paramètres précédents entre `<<< , >>>`.

Pour de plus amples détails, le lecteur est invité à consulter le guide de programmation CUDA [172].

1.4.7 Quelques techniques classiques vues sur ces deux langages

Nous nous proposons ici de décrire trois algorithmes parallélisables pouvant être efficacement adaptés sur GPU, pour s'en servir comme exemples illustrant les différences entre deux langages GPU, l'un à vocation graphique, Cg, l'autre destiné au calcul généraliste, CUDA. Nous avons souhaité faire un tel parallèle pour montrer les atouts et inconvénients de ces deux types de langages.

1.4.7.1 Quelques techniques classiques de programmation parallèle

Parmi les méthodes essentielles en informatiques, le *mapping*, la *réduction* et le *scan* sont trois des techniques ayant été adaptées ou développées pour une exécution sur machine parallèle, en particulier sur GPU. Nous donnons ici un aperçu de ces trois méthodes. Pour de plus amples détails sur les algorithmes parallèles, nous invitons le lecteur à se référer à [100].

Définition 1.9 *Le* **mapping** *consiste à appliquer une même fonction mathématique sur tout un jeu de données.*

C'est le traitement parallèle le plus simple, par lequel chaque unité de calcul parallèle va effectuer la même opération (ou suite d'opérations) sur un morceau des données entrantes.

Un exemple simple de *mapping* est présenté en figure 1.10

Définition 1.10 *La* **Réduction de données** *permet de calculer, à partir du flux de données entrant, un sous-flux ou un élément unique ayant une ou plusieurs particularités désirées.*

La *réduction* est un processus itératif. A chaque itération, la taille des données, initialement n, est divisée par m (un entier, généralement une puissance de 2 si n l'est aussi, et dépendant de la dimension des données en entrée) , le flux des données entrantes est découpé en morceaux de taille m, chacune de ces parties étant envoyée à une unité de calcul parallèle, qui va exécuter sur ces données la suite d'instructions qu'on

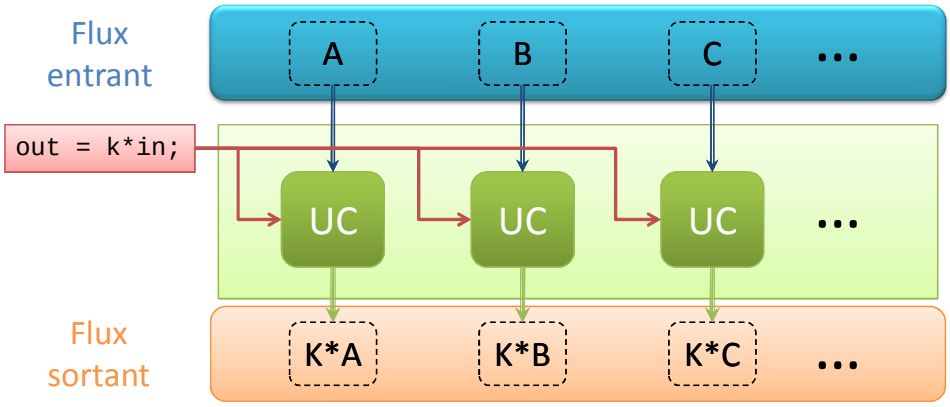


FIGURE 1.10 – *Exemple de* mapping *: multiplication par un scalaire k d'un flux de scalaires entrant : chaque unité de calcul lit un scalaire dans le flux entrant, le multiplie par k, puis écrit le résultat à la bonne place dans le flux de sortie.*

lui a confié, puis va écrire un résultat à l'endroit idoine dans le flux sortant. Lorsqu'une unité a fini de traiter sa partie courante, elle en reçoit une nouvelle, et ce jusqu'à épuisement du flux entrant, ce qui marque la fin de l'itération. Le flux sortant est pris comme entrée dans l'itération suivante. Après au maximum $log_m(n)$ itérations, la sortie est réduite au sous-flux ou à l'élément voulu. La complexité temporelle d'une telle réduction est de l'ordre de $O(\frac{n}{p} \log n)$, avec p le nombre d'unités de calcul disponibles sur le matériel. Séquentiellement, la même réduction aurait une complexité de l'ordre de $O(n)$. Pour des problèmes en dimension supérieure, ce principe de réduction est facilement transposable, en augmentant m. En dimension deux, on réduit la taille du flux par quatre à chaque itération ; c'est le cas le mieux adapté dès que l'on a à manipuler des images 2D.

Un exemple de *réduction* est présenté en figure 1.11.

Définition 1.11 *Le* **Scan** *est un algorithme qui, appliqué à une séquence, en calcule une autre de même taille, dans laquelle chaque élément est la somme de tous les éléments d'index inférieurs dans la séquence d'entrée.*

Malgré une nature apparemment séquentielle, il existe différents algorithmes parallèles efficaces pour réaliser cette opération, également appelée *all-prefix-sum*. Le scan fut popularisé par Blelloch qui le décrit comme une primitive importante dans le calcul parallèle adapté aux machines de l'époque, en particulier les *Connection Machines* [14]

L'idée naïve d'implémentation séquentielle est un parcours simple des données avec mise à jour d'une somme partielle, l'algorithme 1 le présente. Sa complexité est en $O(n)$.

Algorithme 1 : Scan naïf

ENTRÉES : S : scalaire ; x : tableau de données

```
1 S ← x[0];
2 pour i = 1 à n − 1 faire
3 |   S ← S + x[i];
4 |   x[i] ← S;
5 fin pour
```

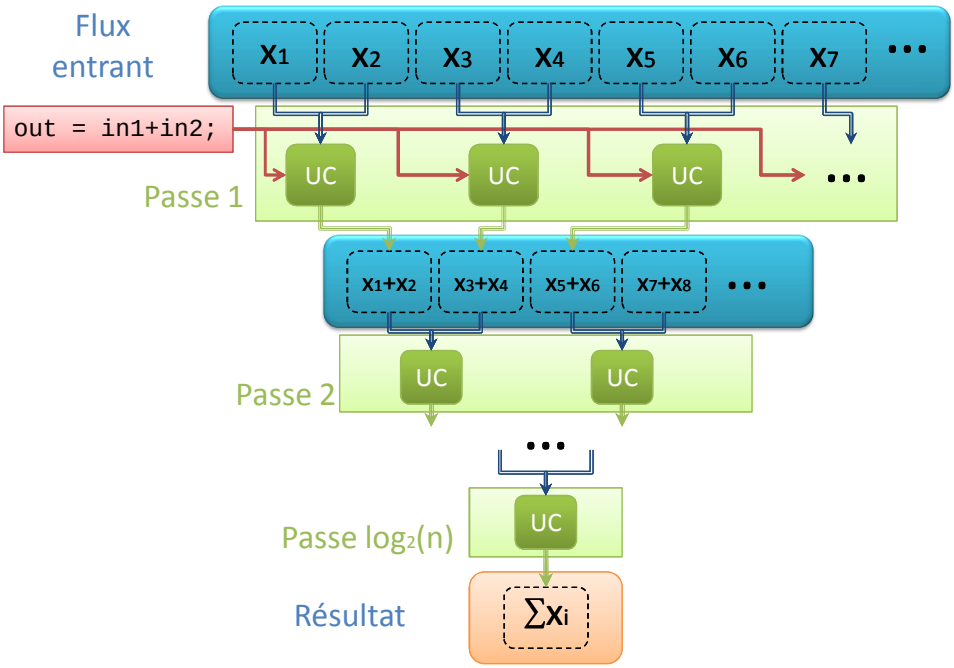


FIGURE 1.11 – *Exemple de Réduction : somme sur un tableau. Les données sont placées dans un flux en entrée. A chaque itération du processus, la taille du tableau est divisée par deux. Un élément d'index i du flux de sortie F_s contient la somme suivante de valeurs du flux d'entrée F_e : $F_s(i) = F_e(2i) + F_e(2i+1)$. Après $log_m(n)$ itérations, on obtient un nouveau flux de taille 1 contenant la somme de tous les éléments initiaux.*

Hillis *et al.* [99] furent dans les premiers à proposer une implémentation parallèle de l'algorithme de *scan*, ou *all partial sums* sur un tableau. Un pseudo-code basé sur leur implémentation est présenté par l'algorithme 2 et la figure 1.12. La complexité de cet algorithme est en $O(n \log_2 n)$, avec n la taille de la séquence d'entrée, ce qui est moins bon que la complexité de l'algorithme séquentiel 1. La parallélisation de l'algorithme introduit ici une baisse théorique de performance, compensée en pratique par l'exécution parallèle de ce code.

Algorithme 2 : Scan parallèle

ENTRÉES : x, tableau de données

1 **pour** $i = 1$ **à** $\log_2 n$ **faire**
2 **pour tout** k **faire en parallèle**
3 **si** $k \geq 2^i$ **alors**
4 $x[k] \leftarrow x[k - 2^{i-1}] + x[k]$;
5 **fin si**
6 **fin pour tout**
7 **fin pour**

Blelloch [14], Horn [104] et Sengupta *et al.* [208, 207] ont montré que l'algorithme de scan et le modèle de données qu'il manie ont des utilisations potentielles dans des domaines variés : détection de collision, surfaces de subdivision, résolution d'équations de récurrence, algorithmes de tri, algèbre linéaire,...

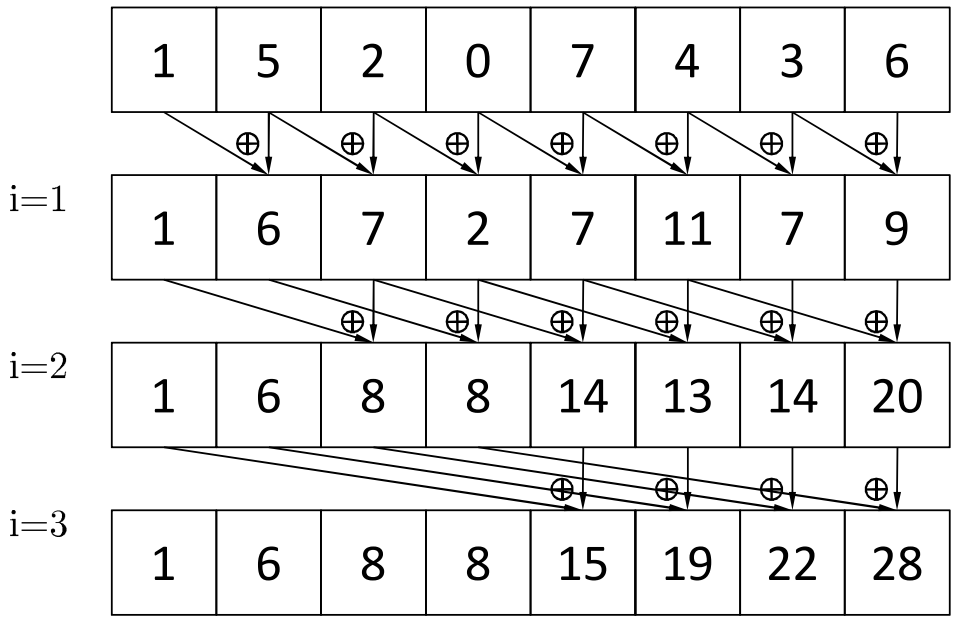


FIGURE 1.12 – *Scan parallèle : Exemple de calcul de sommes partielles sur un tableau de 8 éléments. Pour chaque i, on additionne des couples de valeurs suivant ce schéma*

1.4.7.2 Comparaison de Cg et CUDA

Les trois techniques venant d'être exposées sont parallélisables sur GPU aussi bien en Cg, avec utilisation des notions de textures et fragments, qu'en CUDA, par programmation directe. Nous proposons ici pour chaque méthode présentée ci-dessus un bref aperçu des implémentations existantes.

Mapping C'est l'opération naturelle des GPU, équivalent à l'application d'une fonction mathématique. Aucune implémentation notable n'est donc à relever.

En Cg, chaque pixel/fragment de la texture d'entrée est modifié par la même suite d'opérations, encodée dans un simple shader. Une seule passe sera appliquée, avec un seul shader. La figure 1.10 illustre cette opération.

En CUDA, chaque thread a à traiter un élément du flux en entrée en lui appliquant la fonction mathématique, contenue dans un unique kernel. Une seule invocation de ce kernel est nécessaire.

Pour des opérations simples de type mapping, aucun de ces deux langages n'est spécialement préférable.

Réduction En Cg, la réduction est réalisable en itérant les rendus, avec un seul shader : chaque processeur du GPU va recevoir une partie des données (généralement deux éléments) pour retourner un élément unique. Les données étant stockées dans des textures, il est alors aisé de réaliser des réductions en dimension 1 (c'est le cas proposé sur la figure 1.11, chaque ligne de la texture d'entrée étant un flux entrant indépendant des autres) et en dimension 2. Dans ce dernier cas, à chaque itération, il est possible de diviser par deux les deux côtés de la texture d'entrée, on divise alors par quatre la taille du flux entrant. En un pixel (i, j) de la texture de sortie T_s, on stockera $T_s(i,j) = f(T_e(2i,2j), T_e(2i+1,2j), T_e(2i,2j+1), T_e(2i+1,2j+1))$. En pratique, on utilise deux textures en *ping-pong* (voir section 1.4.2.4)

En CUDA, la réduction est un exemple astucieux des possibilités du langage. Une implémentation est proposée dans les exemples du SDK de CUDA [90]. La difficulté principale d'un tel algorithme sur les machines parallèles que sont les GPU est la synchronisation globale des threads et les communications entre threads. CUDA ne disposant pas de mécanisme de synchronisation globale pour des raisons d'implémentation matérielle, le problème est contourné par la décomposition de la charge en plusieurs kernels, dont les invocations servent de synchronisation. Dans [90] est également présentée une suite d'optimisations d'implémentation et algorithmiques permettant d'améliorer encore d'un facteur 30 une implémentation parallèle naïve de cet algorithme.

La réduction est une opération en général simple à implémenter en Cg. En CUDA, ce n'est pas beaucoup plus complexe pour le cas naïf, mais requiert quelques connaissances en programmation parallèle. La simplicité de Cg et le fait que ce type d'opération se prête bien à ce langage joue en sa faveur.

Scan Le scan est un bon exemple d'algorithme ayant été implémenté en Cg, en particulier le scan de Hillis [99] implémenté par Horn [104] et repris par Hensley *et al.* [97]. C'est une implémentation directe de l'algorithme 2, utilisant la technique du *ping-pong*, tout comme pour la *réduction*. Sengupta *et al.* [208] ont depuis présenté un nouvel algorithme, implémenté en Cg, utilisant au mieux la paralléllisation proposée par ce langage et s'exécutant en $O(n)$, avec n la taille des données en entrée. Cette amélioration théorique provient d'un compromis astucieux entre une décomposition du *scan* en deux parties (une *réduction* pour calculer certaines sommes partielles, suivie d'une recomposition, appelée *down-sweep*, permettant de recalculer des valeurs intermédiaires) et un retour à la méthode de Horn dans les itérations où la taille des données est suffisamment petite. Les étapes de réduction et de *down-sweep* sont telles qu'elles permettent d'occuper efficacement l'ensemble des processeurs, optimisant la parallélisation du calcul.

Peu après la version Cg [208] du scan, Harris *et al.* ont présenté une version implémenté en CUDA [93], adaptant l'algorithme efficace de Blelloch et la précédente implémentation GPU en Cg [208] pour une plus grande efficacité avec CUDA. Sengupta *et al.* ont ensuite proposé une autre amélioration, nommée *segmented scan* [207], autorisant un découpage plus arbitraire du flux de données en entrée. Cette version fut appliquée à des tris de type quicksort ou à des multiplications de matrices creuses. Leur implémentation s'appuie sur certaines propriétés spécifiques à CUDA, en particulier la mémoire partagée.

Pour une utilisation plus approfondie, Cg montre quelques limitations. Ce qu'il est possible de faire en quelques invocations de kernels en CUDA nécessite un nombre important de rendus en Cg. De par les limitations SIMD/CREW de Cg, le partitionnement de certains algorithmes en plusieurs passes (rendus ou kernels) ne peut se faire pareil dans les deux langages.

Comparaison globale de Cg et CUDA Des trois exemples précédents, nous avons vu qu'il est parfois préférable d'utiliser Cg à CUDA et réci-

proquement, et plus globalement des shading language face aux langages GPGPU, cela dépendant de l'application à implémenter.

Nous proposons ici une comparaison succincte des atouts et faiblesses de chacun de ces deux langage pour de la programmation généraliste ; ces remarques sont généralisables

1. Arguments Pour Cg
   - Apprentissage assez simple à partir du C/C++ (nécessite quelques notions de programmation graphique)
   - Peut être utilisé sur n'importe quelle carte NVidia ;
   - Astuces simples et utiles en algorithmique (Early Z Cull, `discard()`, Occlusion Queries,...) ;
   - Proche d'autres langages de shading (HLSL, GLSL), donc portage possible sur cartes ATI avec peu de modifications ;
   - Dans le cas d'applications graphique, permet facilement d'afficher les résultats ;
   - Pas besoin de gestion des threads et bloc, le driver s'en charge.
2. Arguments Contre Cg
   - Nécessite de connaître un minimum les API graphiques (OpenGL, D3D) ;
   - Bottlenecks possibles sur le pipeline (en cas de surcharge des vertex unit, les fragment units ne sont pas exploitées à leur plein potentiel, ou réciproquement) ;
   - Opérations de scatter impossibles : nécessité de repenser certains algorithmes pour les porter en Cg ;
   - Pour de nombreux algorithmes, nécessité de plusieurs passes, pouvant être consommatrices de bande passante ;
   - Modèle mémoriel contraint, pas de mémoire partagée rapide.
3. Arguments Pour CUDA
   - Apprentissage très simple à partir du C/C++, pas besoin de notion en API graphique (CUDA propose sa propre API) ;
   - Scatter possible ;
   - Possibilité d'écrire plus d'un pixel par thread ;
   - Utilisation de l'architecture unifiée des G80 et plus, permettant un meilleur remplissage du pipeline ;
   - Modèle mémoriel plus performant qu'en Cg : mémoire globale, mais aussi mémoire partagée entre les threads d'un bloc ;
   - Synchronisation des threads possible pendant leur exécution ;
   - Dans la plupart des cas, possibilité de baisser le nombre de rendus/invocations nécessaires par rapport à Cg, ce qui implique une réduction de l'overhead driver/matériel ;
4. Arguments Contre CUDA
   - Ne proposait pas, jusqu'à très récemment, de passerelle vers Cg (impossibilité de faire cohabiter un shader Cg et un kernel CUDA), ce qui est problématique si l'on a des besoins graphiques (problèmes de visibilité,...). Désormais, il existe une interopérabilité entre CUDA et OpenGL [70], permettant de partager des données entre CUDA, OpenGL et Cg, ainsi que de mixer les appels provenant de ces différents langages ;
   - Nécessite une carte NVidia récente (architecture G80 ou plus récente) ;

- Aucune garantie sur l'ordre d'exécution des blocs, pas de communication entre bloc ;
- Préférable d'exécuter au minimum 32 threads (un warp) ;

Pour les applications présentées dans les chapitres qui vont suivre, nous aurions pu indifféremment utiliser Cg ou CUDA. Notre choix, porté sur Cg, a été motivé par la création d'une librairie de calcul sur GPU (voir section suivante), que nous avons voulu utilisable dans le maximum d'applications, dont la plupart en vision par ordinateur nécessitant une gestion de graphismes.

1.5 CLGPU

Lors de nos premières implémentations d'algorithmes sur GPU, en Cg, il s'est avéré que nous utilisions répétitivement les mêmes fonctions d'initialisation, de création de textures et de passage de paramètres. L'idée d'une librairie dédiée à faciliter l'utilisation de ces instructions nous parût parfaitement adaptée, d'où le développement de la *CLGPU*, librairie d'appels Cg, au sein de la *CertisLib* [32].

1.5.1 CertisLib

La CertisLib [32] est un ensemble de librairies développées par les membres du Certis et pour leurs besoins de recherche et d'enseignement. Basées sur le langage C++, ces librairies thématiques permettent de manipuler facilement l'ensemble des types de données et d'outils utilisés en analyse d'images et vision par ordinateur : images, caméras, champs de Markov, levels sets, points caractéristiques, stéréovision,... ainsi que les outils d'affichage nécessaires. Elles proposent également un ensemble de classes dédiées à l'algèbre linéaire.

Parmi ces librairies se trouve la CLGPU, dédiée au calcul généraliste sur GPU, écrite en C++, basée sur l'API OpenGL, et pour une utilisation du langage Cg. Dans la suite de cette section, nous explicitons le choix de ce langage par rapport à d'autres, ainsi que le fonctionnement général de la CLGPU.

1.5.2 Motivation : choix de Cg pour la CLGPU

L'idée d'utiliser un langage tel que Cg peut paraitre singulier d'un point de vue actuel, voire handicapant, surtout dans un contexte de programmation générique que nous nous sommes fixés. Néanmoins, plusieurs raisons nous ont pourtant poussés à ce choix.

- La première est matérielle : nous avons dû faire le choix d'un constructeur. Ce choix s'est porté sur NVidia [169], leader à cette date sur le marché des cartes graphiques, proposant des produits répondant à nos attentes et innovant de façon continue.
- La seconde raison est historique : lors du début du développement de la CLGPU, les cartes NVidia disponibles à l'époque étaient celles de la série 7 (architectures G70 à G73). Or, CUDA n'était pas encore disponible, ne fonctionnant qu'à partir des architectures G80. De plus, le choix se limitait aux shading languages : Cg, HLSL, GLSL, Sh,... Le choix de Cg fut assez évident, étant donné le choix du

constructeur qu'on a retenu : Cg ayant été spécialement développé pour ces générations de cartes, ses instructions y sont optimisées.
- Ensuite, bien que seule l'utilisation GPGPU nous intéressait à ce début, il n'était pas exclu qu'un usage plus graphique, en accord avec d'autres librairies de la CertisLib, soit fait de la CLGPU et de ces cartes. Nous voulions donc nous baser sur un langage autorisant les manipulations d'éléments graphiques, de vertex, de pixels, de buffers d'affichage,...
- Cg possède également quelques avantages techniques destinés principalement au calcul graphique mais dont le détournement pour un usage générique est possible et potentiellement bénéfique. C'est en particulier le cas des *occlusion queries* et du *Early Z-Cull*, comme nous l'avons vu dans la section 1.4.4.
- Enfin, par soucis de simplicité, nous voulions un langage simple d'apprentissage, répondant au moins à nos besoins. A ce niveau, Cg est aussi bon que CUDA.

Ces conditions, ainsi que les quelques défaut de CUDA relevés (les threads devant être exécutés par groupe de 32 –un warp– au minimum, l'impossibilité de rendre dans une texture,...) ont donc fait de Cg le candidat idéal.

Par la suite, en particulier dès que CUDA fut disponible, la question de réécrire la CLGPU ne s'est pas réellement posée. La simplicité de Cg, les possibilités que ce langage offre et la mise à disposition en même temps que CUDA d'une nouvelle génération de cartes graphiques (apportant des changements particulièrement intéressants pour Cg) ont suffit à couvrir nos besoins. Les quelques essais que nous avons effectué en CUDA ne nous apportant pas de gain de temps substantiel, nous nous sommes donc arrêtés à Cg.

En conclusion, le choix de Cg par rapport à un autre langage de shading n'a aucun impact, les algorithmes seraient les mêmes et offriraient les mêmes performances. Nous ne nous poserons la question de CUDA que lorsqu'il y aura des passerelles avec les applications graphiques, car nos applications le sont aussi.

1.5.3 Fonctionnement de la CLGPU

Le développement de cette librairie a été pensé de façon à masquer à l'utilisateur les instructions techniques d'initialisation et d'appel de rendus Cg. Cet utilisateur n'a plus qu'à apprendre le contenu même de Cg. Son fonctionnement est simple, la CLGPU ne contenant que quelques classes C++ et peu de méthodes. Pour être utilisée, elle ne nécessite que de très simples connaissances en graphisme.

L'exemple canonique d'utilisation de cette librairie est l'application d'un filtre à une image. Il est donné dans la suite de listings suivante. On souhaite appliquer un filtre gaussien à une image en niveau de gris. Le code complet est divisé en deux parties : d'un côté, les appels CPU pour initialiser et lancer le calcul, et de l'autre côté le calcul même sur GPU.

Les appels CPU sont présentés dans une fonction `main()` C++ :

```
1 /*****************************************/
2 /*                MAIN C++               */
3 /*****************************************/
4 int main(int argc, char **argv)
```

```
5  {
6    // === Definition et chargement des donnees pour une
7    // ===  image CertisLib
8    Image<byte,2> Img;
9    load(Img,SRCPATH("ksmall.jpg"));
10   int w=Img.width(),h=Img.height();
11
12   // === Initialisation du GPU
13   GpuInit(argc,argv,NV80);
14
15   // === Declaration et initialisation du framebuffer
16   FrameBuffer fb;
17   fb.Init();
```

Cette première partie d'initialisations déclare une image `Img` de type définie dans la CLGraphics (une des librairies de la CertisLib) et y charge une image présente sur disque (ligne 9). La CLGPU nécessite d'être initialisée par l'emploi de quelques instructions, regroupées en un seul appel ligne 13, auquel on peut spécifier le modèle de carte utilisé, ici NV80 correspondant à la huitième génération de cartes NVidia. Enfin, un objet appelé *framebuffer* doit être instancié et initialisé. Cet objet est traditionnellement le buffer de sortie du GPU destiné à l'affichage, mais il va ici aider à gérer les liens entre les textures et le GPU. Un et un seul framebuffer est nécessaire.

```
18   // ==== Transfert de l'image vers une texture GPU
19   Texture InTex;            // Texture GPU d'entree
20   InTex.Init(w,h,FLOAT1);   // Initialisation de la texture
21   Image<float,2> I(Img);    // Cast des donnees en float
22   InTex.Set(I.data());      // Transfert vers GPU
23
24   // === Declaration et initialisation de la texture
25   // === de sortie
26   Texture OutTex;
27   OutTex.Init(w,h,FLOAT1);
```

Dans cette deuxième phase, on instancie dans la mémoire de la carte graphique, aux bonnes dimensions, une texture, `InTex`, dans laquelle on va envoyer le contenu de l'image `Img`, puis une texture `OutTex`, aux mêmes dimensions, dans laquelle sera récupérée l'image en sortie. Le transtypage des données de l'image d'entrée est nécessaire pour des raisons de compatibilités avec la CertisLib.

```
28   // === Declaration et initialisation du shader a partir
29   // === d'un fichier externe
30   FragmentShader smooth;
31   smooth.Init(SRCPATH("shaders/smooth.cg"));
32
33   // === Passage d'un parametre au shader et association
34   // === d'une texture
35   smooth.SetInt("n",3);
36   smooth.SetTexture("I",InTex);
```

Avant de lancer le calcul GPU, il faut spécifier le *shader* à utiliser. On en instancie un, `smooth`, auquel on associe le code contenu dans le fichier `smooth.cg` (lignes 30 et 31). A ce shader, on passe les paramètres qu'il demande, ici un entier n de valeur 3, et une texture `I` à laquelle on associe la texture d'entrée `InTex`.

```
37    // === Rendu
38    fb.Render(smooth,OutTex);
```

Après les précédentes phases d'initialisation et de préparation des données, le calcul sur GPU (le *rendu*) est lancé, par cette simple ligne 38. La fonction lancée est une méthode (au sens C++) du framebuffer, à laquelle on passe en argument la texture de sortie, ici `OutTex`. Cette fonction exécute le rendu d'un rectangle de mêmes tailles que celles de `OutTex`.

```
39    // === Recuperation des donnees et affichage
40    fb.Get((void*)I.data());
```

Les résultats peuvent ensuite être récupérés du GPU vers une image sur CPU, `I`.

```
41    // === Detachement du framebuffer, destruction des
42    // === textures et framebuffer, et cloture des librairies
43    fb.UnBind();
44    InTex.Delete();
45    OutTex.Delete();
46    Click();
47    fb.Delete();
48    GpuTerminate();   // Cloture de la CLGPU
49    Terminate(false); // Cloture de la CLGraphics
50
51    return 0;
52  }
```

Avant la fin du programme, il est nécessaire de *détacher* le framebuffer, c'est-à-dire le libérer de ses liens avec les textures auxquelles il a été associé, puis de détruire ces textures, pour pouvoir retrouver l'état OpenGL initial.

En parallèle de ce `main`, le code du shader `smooth.cg` exécuté est le suivant.

```
1  /*****************************************/
2  /*                MAIN Cg                */
3  /*****************************************/
4  float main (
5    in float2 coords : TEXCOORD0,
6    uniform int n,
7    uniform samplerRECT TIn
8              ): COLOR
9  {
10   float J = 0;
11   int a,b;
12   for (a=-n;a<=n;a++)
13     for (b=-n;b<=n;b++) {
14       float2 c=coords+float2(a,b);
15       J+=f1texRECT(TIn,c);
16     }
17   return J/((2*n+1)*(2*n+1));
18  }
```

Ce simple exemple de shader Cg permet d'illustrer les principaux éléments du langage. Un fragment shader est intrinsèquement une fonction, ici `main()`, avec différents types d'arguments :

– un vecteur de deux float, ici `coords` (ligne 5), associé à la *sémantique* `TEXCOORDS0`, contenant les coordonnées dans la texture de sortie du fragment courant. C'est le rôle des sémantiques que de lier une variable Cg à un registre, reflétant une propriété du pipeline graphique : position, couleur, coordonnées dans des textures,...
– un type scalaire, `int` (ligne 6), que l'on déclare `uniform`, signifiant que la valeur proviendra d'un programme extérieur. Ici, l'entier `n` prend sa valeur 3 par le programme C++ principal montré précédemment (ligne 35), grâce à la méthode `SetInt`. Il est également possible de déclarer des scalaires de type `float` grâce à la CLGPU.
– une référence vers une texture contenant nos données, TIn. Cette référence est associée à une texture en dehors du shader, également dans le programme principal (ligne 36), par la méthode `SetTexture`. Ces références sont ici de type `samplerRECT`, désignant une texture dont les dimensions ne sont pas spécifiquement des puissances de 2. Il existe d'autres types de qualifieurs pour les textures : 1D, 2D, 3D, texture pour du *cube mapping* [170].

Le filtre à appliquer est défini par :

$$\forall(i,j), Out(i,j) = \left(\sum_{k=-n}^{n} \sum_{l=-n}^{n} In(i+k, j+l) \right) / (2n+1)^2$$

Les lignes 10 à 17 implémentent directement ce calcul. On voit ici que Cg est très similaire au C/C++ (donc facile d'apprentissage). L'accès à la texture d'entrée est possible par la fonction `f1texRECT()`, à laquelle on passe la référence sur texture et les coordonnées de l'élément à lire. Ainsi, on peut facilement récupérer les valeurs de n'importe quel élément de la texture, et plus particulièrement des voisins spatiaux à partir des coordonnées courantes contenues dans `coords`. De plus, les problèmes de bords sont évités par une extension de type OpenGL de la texture sur ses bords.

Pour des détails approfondis sur le langage Cg, le lecteur est invité à se reporter à [52].

Conclusion du chapitre

Dans ce chapitre, nous avons introduit le concept de *programmation générique parallèle sur carte graphique*, ou *GPGPU* (*General Purpose computing on Graphics Processing Units*) en explicitant son intérêt croissant pour la communauté scientifique, en exposant les notions de parallélisme sur lesquels il repose et qu'il est nécessaire de connaitre dans ce cadre, puis nous avons présenté un historique des générations successives de cartes graphiques et l'architecture matérielle des plus récentes, le modèle de programmation GPGPU que cette architecture impose ainsi que ses contraintes et quelques astuces classiques. Nous avons présenté plus en détail deux langages de programmation GPU utilisés en GPGPU, Cg et CUDA, ainsi que la CLGPU, librairie de calcul GPGPU basée sur Cg et développée par nos soins, en justifiant le choix de Cg.

Dans le chapitre suivant, nous répertorions un nombre important d'utilisations des GPU comme outil computationnel, dans un panel significatif de domaines.

Applications GPGPU

2

Sommaire

L'utilisation détournée du matériel graphique dans les ordinateurs pour du calcul générique a commencé dès la fin des années 70 avec des machines telles les *Ikonas* [47] et les *Pixel Machines* [186]. Plus récemment, les machines *Pixel-Planes 5* [190] et *PixelFlow* [177, 158] ont contribué au développement de techniques telles textures procédurales et ombrages. Les premiers GPU proposaient un pipeline fixe, leur programmabilité effective n'est apparue qu'avec la version 1 des *pixel shaders* de Microsoft, avec Direct3*D* 8.0 en 2000, autorisant de nouveaux types de calculs, détaillés dans [233] par Trendall et Stewart.

Mais c'est surtout lors de ces quelques dernières années, grâce à la mise à disposition de langages informatiques haut niveau et plus faciles d'accès, le coût de plus en plus faible du matériel et l'accroissement de la flexibilité et de la puissance de ces cartes que s'est développé le calcul GPGPU, incitant les chercheurs et les développeurs à s'y intéresser de plus en plus près. On a alors vu émerger un large spectre d'applications computationnelles utilisant ces cartes graphiques comme outil pour le calcul générique.

Dans ce chapitre, nous présentons quelques unes des applications orientées GPGPU parmi les plus récentes. Il serait prétentieux de vouloir être exhaustif étant donné l'engouement de la communauté scientifique pour cet outil ces dernières années ; nous nous bornerons donc à certains domaines significatifs : nous présentons tout d'abord certains outils et méthodes de calcul améliorés grâce aux GPU, puis des applications physiques utilisant ou non ces outils. Différentes applications dans le domaine financier seront brièvement exposées, suivies de méthodes de traitement de signaux puis de synthèse d'images par illumination. Enfin, quelques méthodes en traitement d'images et en vision par ordinateur seront présentées.

La plupart de ces applications sont également répertoriées dans [180] ou [76].

2.1 Outils

Une des premières et principales utilisations des cartes graphiques en calcul générique est le développement ou l'implémentation d'*outils de calculs*. Deux domaines ont particulièrement intéressé les chercheurs ces dernières années : l'algèbre linéaire et la résolution d'équations aux dérivées partielles. Nous citons également quelques applications algorithmiques.

2.1.1 Algèbre linéaire

L'adaptation d'algorithmes du domaine de l'algèbre linéaire s'est faite aussi récemment que rapidement, les chercheurs ayant constaté que beaucoup de ces algorithmes peuvent s'adapter au schéma de calcul SIMD proposé par les GPU. Les premières implémentations de calcul matriciel y ont été réalisées par Larsen et McAllister [133], utilisant les fonctions de *blending* de textures disponibles sur les cartes, consistant à mélanger deux textures ou plus. Un an plus tard, Thompson *et al.* ont proposé un framework de calcul généraliste sur GPU [229], incluant la possibilité d'exécuter des opérations algébriques, et montrant pour la première fois que les GPU peuvent rivaliser et dépasser les CPU en temps de calcul sur des opérations matricielles.

Bolz *et al.* ont présenté une implémentation de matrices creuses [15], utilisée dans un résolveur de gradient conjugué, ainsi qu'un résolveur multigrilles sur grilles régulières. Goodnight *et al.* ont également proposé un résolveur multigrilles différent [73], destiné à résoudre les problèmes de frontières. Moravánszky a introduit sur GPU un système algébrique linéaire reposant sur une représentation de matrices denses [46]. Avec une approche plus large, Krüger et Westermann ont proposé un framework complet d'algèbre linéaire, avec des représentations optimisées pour GPU des types vecteurs, matrices pleines et creuses [127]. La figure 2.1 montre la représentation d'une matrice pour GPU, tandis que la figure 2.10 montre le résultat de l'utilisation de ce framework. Ces outils ont particulièrement été utilisés pour la résolution d'équations aux dérivées partielles, comme nous le verrons dans la section 2.1.2.

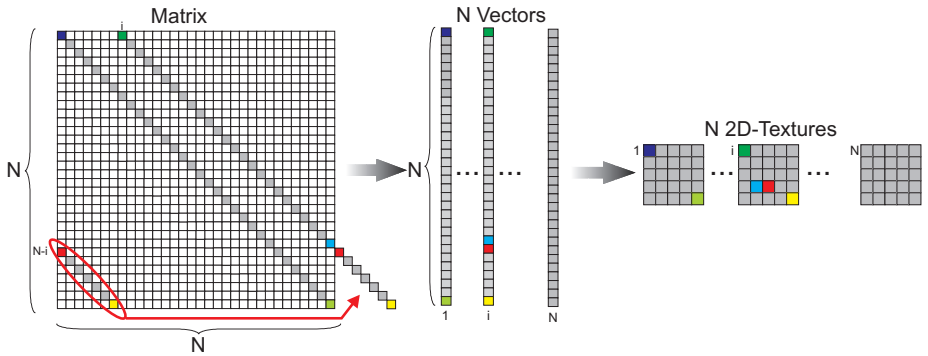


Figure 2.1 – *Représentation d'une matrice en GPU sous la forme d'un ensemble de vecteurs diagonaux, puis d'un ensemble de textures. Issue de [127].*

Galoppo *et al.* ont introduit des algorithmes adaptés aux GPU permettant de résoudre efficacement des systèmes linéaires denses [63], en stockant les matrices entièrement dans des textures 2*D* et en adaptant les opérations de décomposition matricielle en terme de *rasterisation*. Récemment, Barrachina *et al.* ont également proposé deux API, basées sur

la librairie *CUBLAS* [171], pour la résolution de systèmes linéaires [6], en apportant des amélioration algorithmiques et techniques, comme une répartition de la charge entre CPU et GPU. Ils se sont particulièrement intéressés à l'implémentation de variantes de la factorisation de Cholesky [7].

Les propriétés des systèmes linéaires tridiagonaux les rendent implémentables de façon non-itérative, comme l'ont proposé Kass *et al.* avec leur résolveur [115], appliqué dans un calcul temps réel de l'approximation de la profondeur de champ sur une image (voir figure 2.2). Ces résultats ont été obtenus par application d'un filtre

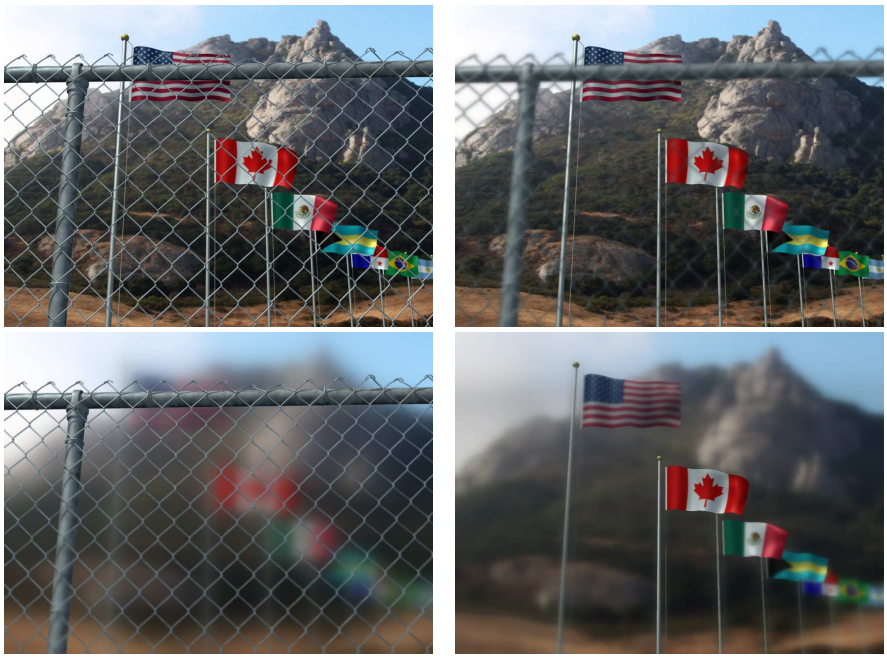


FIGURE 2.2 – *Différents rendus de profondeur de champ. En haut à gauche : image nette de la scène. En haut à droite : ouverture petite et plan focal à mi-distance. En bas à gauche : grande ouverture et plan focal éloigné. En bas à droite : grande ouverture et plan focal proche. Issu de [115].*

De façon plus générale, l'appropriation des GPU pour le calcul algébrique a été étudiée par Fatahalian *et al.* [49]. Ils se sont surtout intéressés aux multiplications matrice-matrice, qu'ils ont montré limitées par le manque de bande passante pour l'accès au données en cache. Pour remédier à cela, ils ont également proposé des changements dans l'architecture des prochains modèles de GPU.

2.1.2 Equations aux Dérivées Partielles (EDP)

L'importance et l'efficacité des équations différentielles dans presque tous les domaines scientifique est telle que pouvoir accélérer leurs calculs, de plus en plus complexes, est devenu un enjeu important. C'est pourquoi il est naturel de voir apparaître dans la littérature de plus en plus d'articles traitant d'implémentations GPU de résolveur d'équations aux dérivées partielles, plus difficile à adapter correctement sur GPU que les équations différentielles ordinaires (EDO), dont l'implémentation est assez directe après discrétisation.

Les méthodes numériques employées pour résoudre ces EDP sont nombreuses. Parmi celles employées sur GPU, on trouve par exemple une

méthode basée sur le gradient conjugué (Krüger et Westermann [127], Bolz *et al.*[15]), une méthode multi-grilles (Goodnight *et al.* [73] , Bolz *et al.*[15]), ou bien la méthode de Jacobi (Harris *et al.* [91], un résultat de leur implémentation est présenté en figure 2.11).

Les premiers essais menés pour résoudre ces EDP avec l'aide des GPU furent de Rumpf et Strzodka [197, 196], appliqués à de la diffusion non-linéaire (voir figure 2.3) et à de la segmentation dans une image par level-set. Ils ont fait correspondre d'une part des structures de données telles matrices et vecteurs à des textures, et d'autre part des opérateurs algébriques à des fonctionnalités GPU telles que le *blending*.

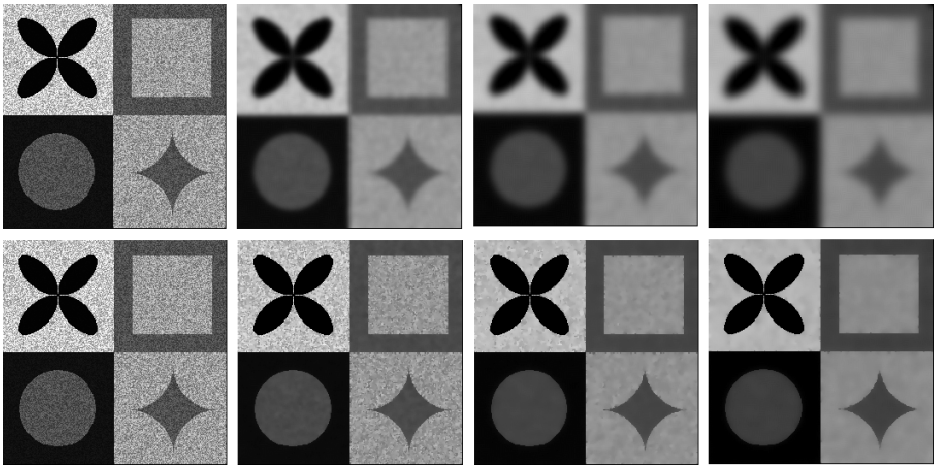


FIGURE 2.3 – *Diffusion non linéaire. Comparaison d'une diffusion linéaire (haut) et d'une diffusion non-linéaire (bas) en GPU. Issu de [197]*

Lefohn et Whitaker ont utilisés les GPU pour résoudre des systèmes creux d'équations aux dérivées partielles non linéaires [134, 136]. De plus, Lefohn a également défini une nouvelle structure de données pour un résolveur d'EDP s'adaptant aux régions d'intérêt [135].

Plus récemment, Rumpf et Strzodka ont étudié l'implémentation de méthodes par éléments finis sur GPU plus en détail [198]. Zhao s'est appuyé sur le modèle de Lattice-Boltzmann, initialement concu pour des problèmes de dynamique des fluides, pour construire un résolveur d'EDP sur GPU [260].

2.1.3 Algorithmique

Tris L'une des opérations fondamentale en algorithmique est le tri d'éléments. Un large panel de méthodes variant en complexité existe, *a priori* peu adaptées à des implémentations GPU. C'est surtout l'algorithme de tri bitonique [8] qui a été implémenté sur GPU.

Buck et Purcell ont proposé une première implémentation [26] de ce tri, présenté en figure 2.4. Par la suite, Kipfer et Westermann reprennent cette implémentation et l'améliorent en utilisant au mieux les ressources disponibles sur ces cartes [120], et propose dans le même article un autre algorithme de tri respectant l'ordre des éléments dans des tableaux partiellement triés, dans des étapes intermédiaires du tri, ce qui peut être un avantage dans les situations où une légère erreur est tolérable.

Greb et Zachmann améliorent l'algorithme bitonique par une nouvelle approche théorique [77], atteignant la complexité minimale de $O(n \log n)$ pour le tri. Par rapport à une version CPU optimisée (fonction `sort()`

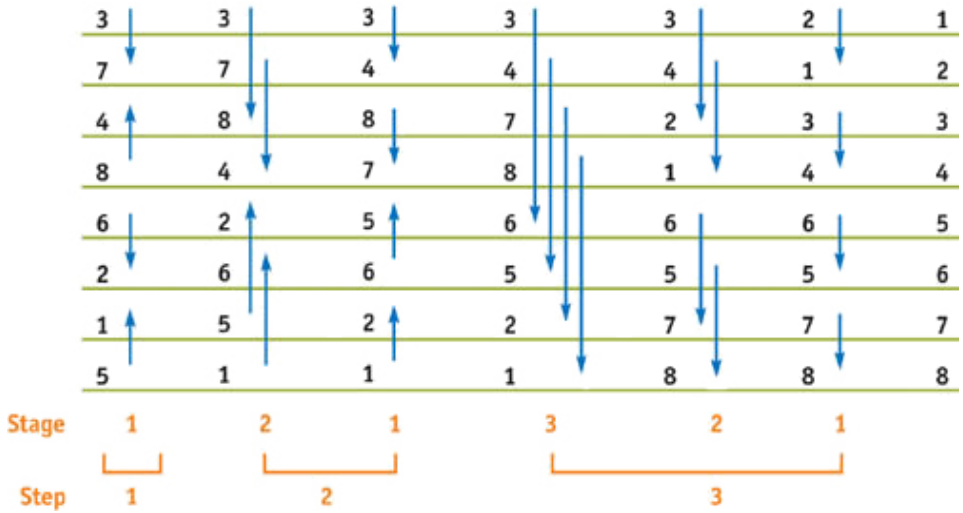


FIGURE 2.4 – *Tri bitonique sur 8 éléments. Ce tri nécessite trois étapes, chacune composée de une à trois phase. Chaque flèche est une comparaison à faire entre deux éléments, le plus petit devant se trouver* a posteriori *à la queue de la flèche. Issu de [26].*

de la *STL*), leur gain est seulement de 1.10 à 1.7 dans le meilleur des cas présentés, sur une GeForce6800.

Leur implémentation est, de plus, rapide d'un point de vue pratique.

Graphes Malgré la nature non ordonnée des graphes et la nécessité d'avoir un modèle de programmation parallèle avec des données correctement organisées, certains algorithmes basés sur ces graphes ont été adaptés, voire développés, pour être calculés sur GPU. Dans ce sens, Frishman et Tal ont proposé un schéma global de partitionnement sur plusieurs niveaux, réalisable sur GPU [58] (voir figure 2.5) ; ils ont également présenté une méthode de représentation de graphes par GPU [59]. Harish et Narayanan ont implémenté plusieurs algorithmes de recherche sur de larges graphes (parcours en profondeur d'abord, plusieurs recherche de plus courts chemins,...) en CUDA [85], obtenant des facteurs de gain intéressants, variant entre 20 et 50.

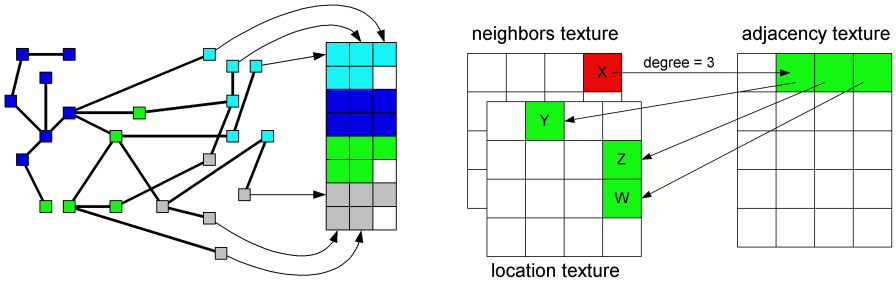


FIGURE 2.5 – *Représentation d'un graphe sur GPU. Gauche : le graphe est partitionné spatialement, pour être correctement représenté dans une texture. Droite : Représentation des arêtes du graphe : le noeud X a ici trois voisins : Y, Z et W. Issu de [58].*

Autres Silberstein *et al.* ont présenté une technique de description d'algorithmes consommateurs de mémoire sur GPU, par gestion manuelle du cache disponible, en CUDA [213].

2.2 Simulations physiques

Les outils mathématiques présentés dans la section précédente ont rapidement intéressé les physiciens souhaitant réaliser des simulations nécessitant de résoudre des systèmes linéaires ou équations différentielles.

2.2.1 Automates, *Coupled Map Lattice* et dérivés

Les premières travaux GPGPU menés en simulations physiques concernèrent des techniques cellulaires, en particulier les *automates cellulaires*. Greg James a par exemple implémenté le *jeu de la vie* [109].

Harris *et al.* ont plus tard utilisé des CML (*Coupled Map Lattice*) pour la simulation de phénomènes descriptibles par EDP [92] : cela inclut la simulation de phénomènes d'ébullition, de convection et de réaction de diffusion du type équation de la chaleur, que l'on observe en figure 2.6. Wu *et al.* ont poursuivi cette démarche, résolvant des problèmes de dynamique des fluides en accélérant leurs calculs par d'astucieux regroupement de leurs variables scalaires et vectorielles [246]. Li *et al.* ont appliqué les LBM (*Lattice Boltzmann Methods*) à bon nombre de problèmes de fluides [139, 138].

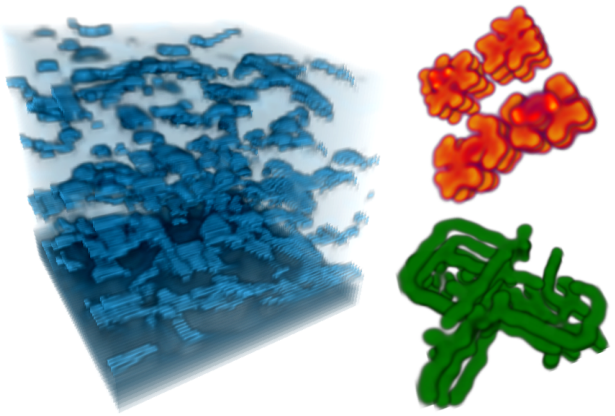


Figure 2.6 – *Représentation de* Coupled Map Lattice *en* 3*D. Gauche : ébullition. Droite : réactions de diffusion. Issue de [92].*

Kim et Lin ont simulé la croissance de cristaux de glace avec un outil du même type, les *phase field models*, modifié pour une manipulation esthétique de cette croissance [118], comme on peut le voir en figure 2.7

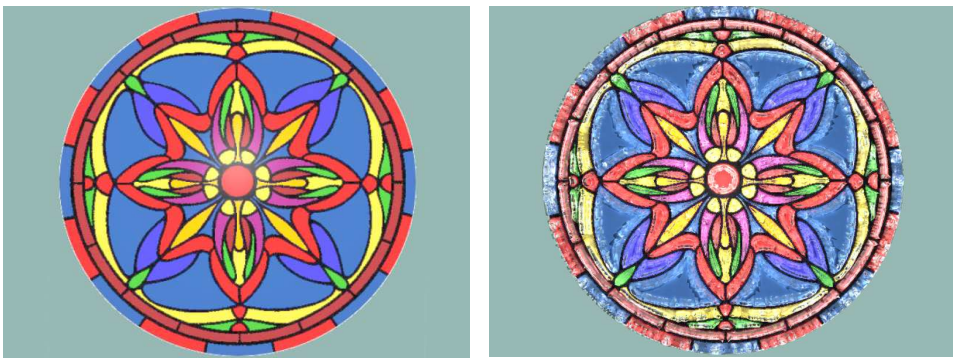


Figure 2.7 – *Croissance de cristaux de glace sur vitrail. Gauche : vitrail original. Droite : cristaux de glace ayant crû à partir des bords du cadre, par processus itératif. Issue de [118].*

2.2.2 Systèmes de particules

Les systèmes de particules proposent de gérer les propriétés (position, vitesse, masse,...) d'un grand nombre d'éléments simples en fonction des forces locales et globales en présence. Ils sont par exemple largement utilisés dans les simulations physiques de phénomènes naturels : feu, explosion, fumée, pluie, surface liquides, chute de neige, mais aussi cheveux, fourrure, herbe,...ces simulations représentent des enjeux importants pour l'industrie cinématographique, par exemple.

L'accélération de ces simulations grâce aux GPU a été étudiée par Kipfer *et al.* [119], proposant une méthode permettant également de classer rapidement les particules afin de déterminer les collisions potentielles entre elles. Kolb *et al.* ont parallèlement implémenté un système de particule calculant avec précision les collisions de particules avec le reste de la géométrie de la scène [123] (voir figure 2.8). Green a proposé un simple système particulaire dans les exemples du SDK de NVidia [80], étendu par Nyland *et al.* pour le calcul des forces gravitationnelles à n corps [173]. Ils améliorèrent cette implémentation quelques temps plus tard à l'aide de CUDA [174]. Krüger *et al.* ont proposé un système de particules, accéléré par GPU, pour la visualisation interactive de flux $3D$ sur grilles uniformes [126]. Parallèlement, Potiy et Anikanov ont présenté la même année une méthode similaire [185].

FIGURE 2.8 – *Système de particules. Collisions entre particules et différentes géométries. Gauche : lapin de Stanford sous la neige. Milieu et droite : fontaine de Vénus. Issue de [123].*

Utilisant les systèmes à particules, les simulations physiques de mouvements de tissus sont de plus en plus étudiées, ces recherches étant guidées par les besoins des industries du cinéma d'animation et du jeu vidéo. Pour ces simulations, les particules sont les sommets des maillages représentant le tissu, soumis aux forces exercées par les sommets voisins. Green a montré un simple exemple d'une telle simulation [79], utilisant l'intégration de Verlet pour le calcul de trajectoires, grâce à la manipulation de vertex shaders : chaque sommet du maillage y est un vertex du flux d'entrée, c'est d'ailleurs un des rares exemples de calcul orienté GPGPU réalisé par vertex shader et non fragment shader. Zeller a étendu ces travaux en ajoutant des contraintes de cisaillement du tissu permettant un meilleur réalisme par son déchirement en plusieurs morceaux [257, 258] (voir figure 2.9).

2.2.3 Simulation de fluides

La simulation d'écoulement de fluides nécessite de résoudre les *équations de Navier-Stokes* [227]. Bolz *et al.* [15], Goodnight *et al.* [73], Harris [87], Harris *et al.* [91] et Krüger et Westermann [127] ont introduit ces équations

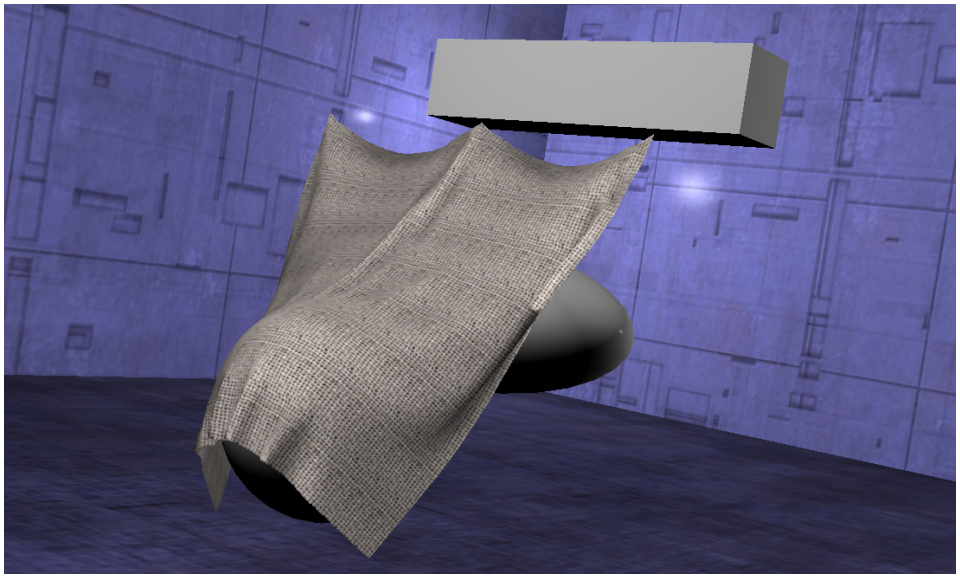


FIGURE 2.9 – *Simulation de mouvements de tissus. Exemple de déformation possible. Obtenu à partir de [258].*

sur GPU. La figure 2.10 nous montre le rendu d'une surface liquide en temps réel, et la figure 2.11 le rendu dynamique de nuages.

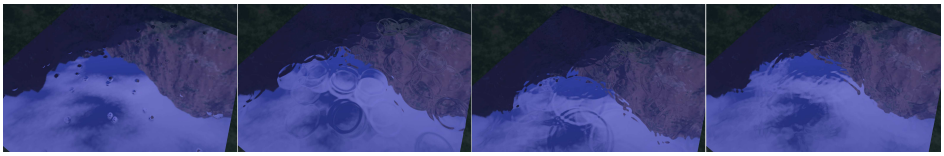


FIGURE 2.10 – *Simulation de la surface d'un liquide en temps réel, par résolution d'un ensemble d'équations algébriques. Issue de [127].*

FIGURE 2.11 – *Simulation dynamique de nuages au sein d'une application interactive. Issue de [91].*

Hagen *et al.* ont simulé la dynamique de gaz idéaux décrits par des équation d'Euler, sur GPU [83], en deux et trois dimensions.

La simulation de flux autour d'obstacles a fait l'objet de plusieurs études : Bolz *et al.* [15], Krüger et Westermann [127], Krüger *et al.* [126] (voir figure 2.12), Liu *et al.* [142] et Sander *et al.* [200] s'y sont particulièrement intéressés, Harris proposant une implémentation de cette technique, appliquée à la dynamique de nuages [88].

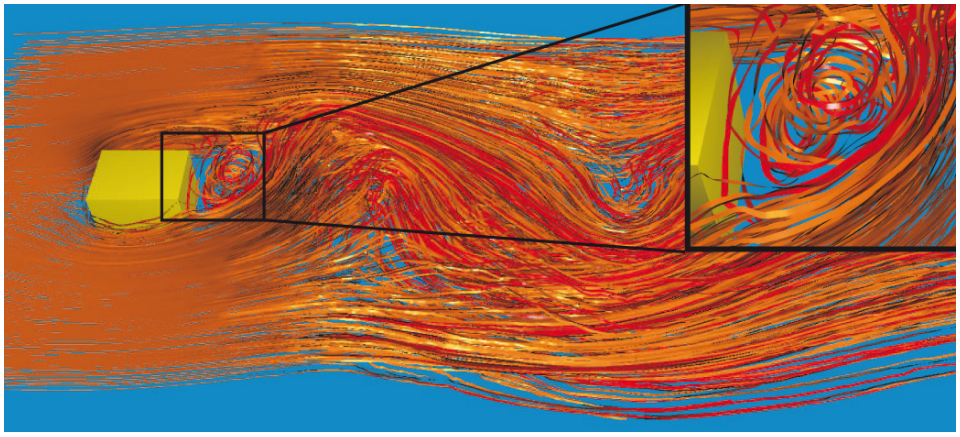


FIGURE 2.12 – *Simulation de lignes de flux. Autour du cube, ces lignes sont perturbées. La simulation fonctionne en temps réel. Issue de [126].*

Crane *et al.* ont récemment proposé un simulateur complet pouvant s'intégrer dans des applications temps réel [117].

Bien que ne pouvant pas atteindre l'apparence réaliste obtenue par une réelle simulation de fluides, l'utilisation des systèmes à particules pour émuler les effets des fluides a tout de même été un sujet d'attraction pour certains chercheurs, tels Iwasaki *et al.* [108], Kolb et Cuntz [124], Hegeman *et al.* [96] ou bien Green [81].

2.3 Finance

Les besoins computationnels du monde financier font que ce dernier s'intéresse de plus en plus aux outils dédiés à l'accélération de calculs ; les arguments des GPU en font d'excellents candidats. Néanmoins, la littérature de ce domaine reste assez pauvre pour l'instant, les recherches effectuées étant loin d'être systématiquement publiées.

Un des prérequis essentiels en finance est la génération de nombres aléatoires entiers, auquel se sont récemment attelés Sussman *et al.* [224], Howes et Thomas [106] ou bien Mascagni [149].

Les *options* et autres *produits dérivés* sont des outils largement employés pour éviter les risques associés aux investissements, en prédisant les valeurs futures d'un actif financier. Kolb et Phar ont présenté des implémentations GPU de ces options [125], utilisant deux modèles de *pricing* différents : le modèle de *Black-Scholes* et les *lattice models*. Ces deux approches s'adaptent bien au modèle computationnel des GPU. Surkov a également proposé une adaptation du modèle de *Black-Scholes* sur GPU [184]

De plus, Surkov a également présenté des algorithmes utilisant la FST (*Fourier Space Time-stepping*) pour l'évaluation de différent types d'options [223].

D'autres outils matériels d'accélération sont également utilisés : Agarwal *et al.* [1] ont implémenté, sur le *Cell Broadband Engine*, processeur multicœur de Sony, Toshiba et IBM, des méthodes de génération de nombres aléatoires et des simulations de Monte Carlo pour calculs financier.

2.4 Traitement de signaux

L'un des outils les plus utilisés en traitement du signal, et néanmoins coûteux, est la FFT (*Fast Fourier Transform*, *Transformée Rapide de Fourier*), permettant les analyses de divers signaux dans le domaine fréquentiel et différents types de compression de données.

Plusieurs projets de développement d'algorithmes de FFT sur GPU sont apparus dès 2002 : Mitchell *et al.* [157], Moreland et Angel [161] (voir figure 2.13), Buck *et al.* [24], Schiwietz et Westerman [203], ou bien Sumanaweera et Liu [222] avec application dans le milieu médical. Govindaraju *et al.* se sont concentrés sur l'utilisation effective de la mémoire cache, des possibilités vectorielles des textures et du rendu dans plusieurs textures (*Multiple Render Target*, MRT) [75]. Horn a proposé une librairie open-source permettant des opérations de FFT sur GPU [105]. Son implémentation est considérée comme très performante, même si Owens *et al.* en ont démontré les limitations [179] en une dimension, en terme de bande passante, gestion du *write-back* vers la mémoire cache ou d'*overhead*.

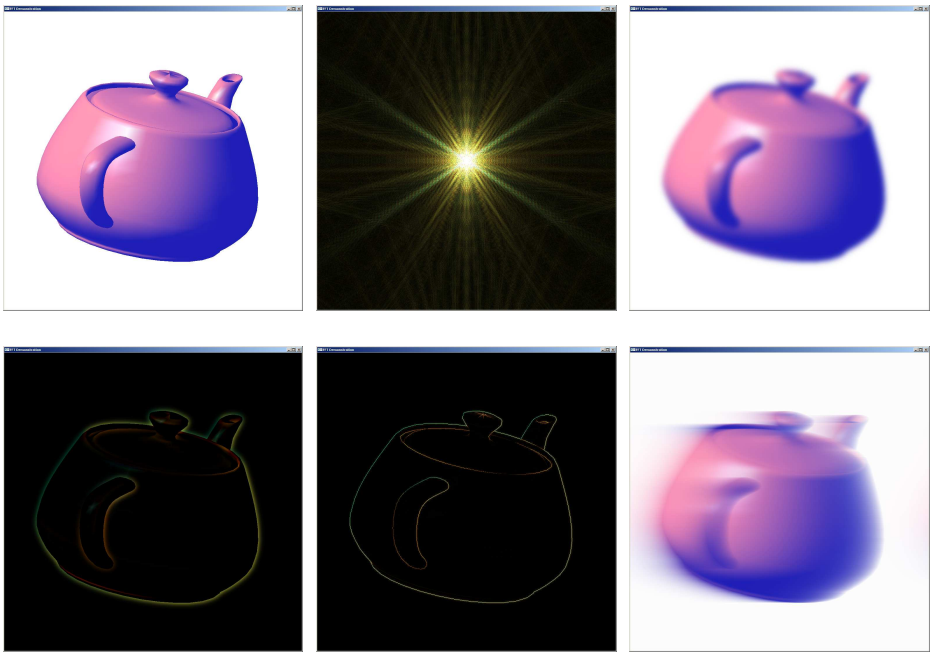


Figure 2.13 – Fast Fourier Transform *et applications. Haut : image originale ; fréquences de cette image ; avec un filtre passe-bas. Bas : avec un filtre passe-haut ; avec un filtre laplacien ; avec flou cinétique non réaliste. Issue de [161].*

Un dérivé de la FFT, la DCT (*Discrete Cosine Transform*, *Transformée en Cosinus Discrète*), notamment utilisé dans la compression d'images en JPEG, a également été étudiée par GPU. Green en propose une implémentation [78]. Wang *et al.* ont également proposé un portage de la DWT (*Discrete Wavelet Transform*, *Transformée en Ondelette Discrète*), autre type de transformation utilisée en traitement de signal, sur GPU [240].

Les deux types de filtres numériques linéaires, à réponse impulsionnelle finie (*Finite Impulse Response*, FIR), respectivement infinie (*Infinite Impulse Response*, IIR), autant utilisés en traitement du signal que la FFT et ses dérivés, ont été adaptés sur GPU par Smirnov et Chuieh [216], respectivement Green [78], ce dernier ayant implémenté des IIR $2D$ séparables. Kass *et al.* ont postérieurement présenté une version plus efficace du filtre IIR

2D [115], en utilisant une réduction cyclique basée sur un algorithme de type *scan* (voir section 1.4.7.1). Une application est présentée en figure 2.2 avec un calcul de profondeur de champ en temps réel.

2.5 Illumination

Le but premier des GPU reste d'améliorer la qualité et la rapidité des rendus graphiques. Nous présentons brièvement ici trois des techniques de génération d'images qui vont dans ce sens, adaptées au GPU : le *ray tracing*, le *photon mapping* et le *rendu par radiosité*.

2.5.1 Ray Tracing

Le *ray tracing*, ou *lancer de rayon*, est une technique de rendu simulant l'interaction de la lumière avec la description géométrique de la scène, en calculant la couleur de chacun des pixels par le lancer de rayons de la caméra vers la scène [69]. Les algorithmes de ray tracing vont donc à l'encontre du schéma classique de rendu, objet par objet, pour lequel le GPU est utilisé. Néanmoins, ce fut une des premières techniques de rendu à avoir été implémenté en GPGPU.

Purcell [187] et Purcell *et al.* [188] ont montré qu'il est possible d'adapter l'intégralité des étapes nécessaires au ray tracing sur GPU : génération des rayons, calcul de leurs trajectoires, intersections avec les objets de la scène et détermination de la couleur résultante. Ils ont de plus montré que l'utilisation de structures d'accélération (structures de données permettant de réduire sévèrement le nombre de tests d'intersections nécessaires par rayon) est possible dans ces implémentations GPU.

Les utilisations de ces structures accélératrices, les grilles uniformes pour Purcell [187, 188], les volumes englobants hiérarchisés pour Thrane et Simonsen [230] et les *k-d tree* pour Foley et Sugerman [57] (voir figure 2.14), ont été comparées par Thrane et Simonsen dans cette même étude.

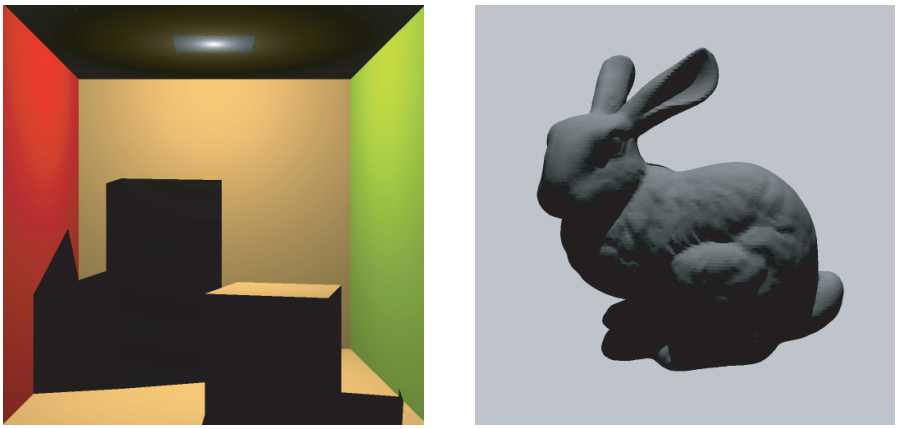


Figure 2.14 – *Rendu par* ray tracing. *Gauche : Cornell box. Droite : lapin de Stanford. Issue de [57].*

Bien qu'apportant un considérable gain, ces structures ne permettaient que la génération d'images statiques par ray tracing. Carr *et al.* [31] ont décrit une nouvelle structure accélératrice, basée sur des volumes englobants hiérarchisés, autorisant la génération dynamique d'images par ray

tracing. Woop *et al.* [225] avaient proposé, un an auparavant, une architecture complète, basée sur celle des GPU, permettant un rendu en temps réel d'une scène par ray-tracing, en utilisant une ou plusieurs cartes de type FPGA.

Weiskopf *et al.* [241] ont proposé une implémentation GPU d'un algorithme de ray tracing non linéaire, permettant de modéliser les milieux d'indices de réfraction variables (mirages) ou les phénomènes gravitationnels (trous noirs). Ces chemins courbes sont ici approximés par morceaux linéaires.

2.5.2 Photon Mapping

Le *photon mapping*, ou *placage de photon* [112], est une méthode de rendu d'images de synthèse s'appuyant sur l'illumination globale de la scène. Cette technique est composée de deux étapes. Durant la première, on simule l'émission de photons à partir des sources lumineuses vers la scène et leurs interactions avec les surfaces, en mémorisant les contributions de ces photons dans une structure adaptée, la *photon map*. La seconde étape est un rendu dans lequel la précédente structure est utilisée pour déterminer la lumière en chaque point des surfaces visibles.

Purcell *et al.* ont proposé différentes techniques de création et d'interrogation de cette photon map en GPU [189], tâches plus difficiles que sur CPU. Larsen et Christensen ont implémenté un algorithme de photon mapping en temps réel sur GPU, en partageant la charge entre GPU et CPU et en exploitant une cohérence temporelle entre images successives [132].

Le rendu réaliste de caustiques, enveloppe des rayons lumineux réfléchis ou réfractés, a suscité plusieurs travaux, Wyman et Davis [247] (voir figure 2.15) et Shah *et al.* [209] s'y sont par exemple intéressés.

FIGURE 2.15 – *Rendu de caustiques réflectives d'un anneau en métal et d'un dragon. Issue de [247]*

2.5.3 Radiosité

La *radiosité* est un autre algorithme de rendu ressemblant au photon mapping, dans lequel l'énergie est envoyée à la scène à partir des sources lumineuses, mais où elle n'est pas stockée dans une structure annexe ; la géométrie de la scène est à la place découpée en éléments, conservant

l'énergie y arrivant. [74]. C'est une application de la méthode des éléments finis, utilisée ici pour résoudre l'*équation de rendu* [114], équation intégrale donnant la quantité d'énergie en un point d'une surface.

Carr *et al.* ont implémenté sur GPU [30] l'algorithme classique de radiosité [74], avec transluminescence (voir figure 2.16), ainsi que Coombe *et al.* [38]. Basé sur la méthode de *radiosité progressive* [35], Coombe et Harris ont proposé une autre implémentation GPU [37].

FIGURE 2.16 – *Rendu par radiosité d'une scène à géométrie complexe. Issue de [38].*

2.6 Traitement d'images et Vision par Ordinateur

Le traitement d'images et la Vision par Ordinateur étant deux domaines proches, nous avons choisi de présenter leurs applications ensembles.

2.6.1 Traitement d'images

Le traitement d'images a pour objet l'étude et la transformation des images numériques pour en dégager de l'information, en interpréter le contenu. Les quelque méthodes présentées ici, parmi les plus classiques, permettent de segmenter, filtrer ou corriger des images.

2.6.1.1 Segmentation

La segmentation est l'un des outils essentiels dans le domaine du traitement d'images ($2D$ ou $3D$). La possibilité de segmenter des objet, c'est-à-dire de les séparer les uns des autres, d'en identifier les contours de façon automatique, permet de précieuses applications, particulièrement dans le domaine médical, où l'identification des structures en présence

(par radiographie, résonance magnétique, magnétoencéphalographie, tomographie,...) est nécessaire. Les algorithmes employés dans ce domaine ont naturellement été portés sur GPU.

Seuillage Le seuillage est une des méthodes de segmentation les plus simples, requérant peu de calcul : suivant sa valeur, chaque pixel est étiqueté comme faisant partie ou non de la structure à segmenter. Son accélération sur GPU est néanmoins appréciable lorsqu'appliqué sur de larges images. Yang et Welch ont implémenté un tel seuillage [253], et les travaux de Viola *et al.* ont permis un seuillage en $3D$ [238] (voir figure 2.17).

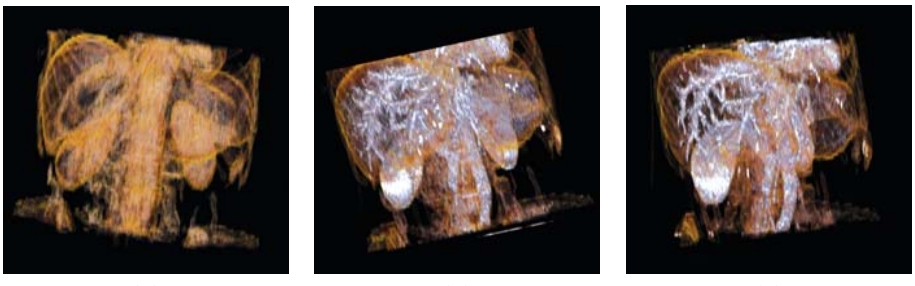


FIGURE 2.17 – *Segmentation par seuillage. Gauche : jeu de données de foie. Milieu et droite : deux segmentations du réseau des vaisseaux sanguins, suivant un masque médian, respectivement bilatéral. Issue de [238]*

Level-set Une deuxième méthode de segmentation, plus efficace mais plus couteuse, a également été implémentée en GPU, les *level-sets*, consistant en une évolution de surfaces implicites représentant le contour des objets à segmenter. Rumpf et Strzodka ont proposé une telle méthode en $2D$ [196]. Une implémentation $3D$ a été présentée par Lefohn et Whitaker [134] (voir figure 2.19), permettant également un traitement du bruit. Un perfectionnement notable a été proposé par Lefohn *et al.* [136], améliorant la vitesse de calcul en n'effectuant les calculs que sur les pixels susceptibles d'être intégrés à la surface (méthode des *narrow band*). Sherbondy *et al.* ont présenté une autre implémentation de cette segmentation [210]. Plus récemment, Schenke *et al.* ont analysé les différentes implémentations existantes et ont proposé un modèle hybride de segmentation $3D$, par seuillage et croissance de région [202]. Enfin, Labatut *et al.* [128] ont présenté une implémentation des level-sets utilisée pour un algorithme de reconstruction $3D$ par stéréovision multi-caméras (voir figure 2.18).

Segmentation morphologique Brambor a exposé quelques méthodes de segmentation à partir d'outils morphologiques, sur GPU et autres plateformes dédiées au traitement de flux [20].

2.6.1.2 Filtrage

Les filtres sont un autre outil de base en traitement d'image, permettant d'obtenir divers effets, sur les tons, les couleurs, la netteté.

Colantoni *et al.* ont implémenté sur GPU divers algorithmes de filtrage, de conversion d'espaces de couleur, de diffusion anisotrope ou d'analyse en composantes principales [36]. Strengert *et al.* ont utilisé les GPU pour

FIGURE 2.18 – *Stéréovision multi-vues par* level-set. *Haut : extrait du jeu de données (temple). Bas : temple reconstruit avec angles de vues correspondants. Le jeu de données contient* 24 *images de* 256×256 *pixels. La reconstruction s'effectue en* 420*s sur GPU, contre* 1550*s pour une version CPU. Issu de [128].*

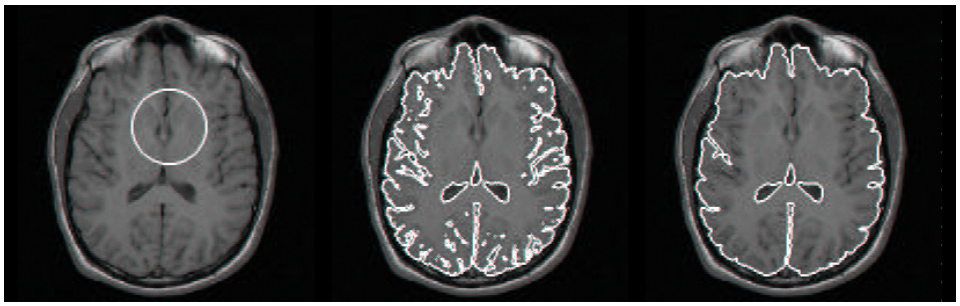


FIGURE 2.19 – *Segmentation par* level set *d'une coupe IRM de cerveau. Gauche : contour initial. Milieu : une étape de l'évolution de la surface. Droite : résultat final. Issue de [134].*

réaliser des filtres bilinéaires par des méthodes pyramidales [220]. James a développé un filtre ajoutant un effet de *glow* à l'image (objets lumineux provoquant un halo autour d'eux), pouvant s'exécuter en temps réel [110] (voir image 2.20). Bjorke a implémenté en GPU plusieurs algorithmes de correction des couleurs, ajustement des contrastes, saturations,... [13] Jargstorff a proposé un framework complet de traitement d'image sur GPU, offrant la possibilité de créer et d'enchaîner divers filtres définis par l'utilisateur [111]. Plus récemment Novosad s'est intéressé au développement de filtre d'interpolation et aux problèmes d'antialiasing [167].

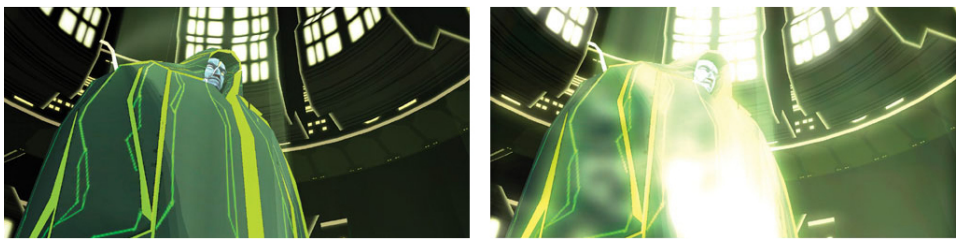


FIGURE 2.20 – *Filtre* glow, *calculé en temps réel. Gauche : image originale. Droite : image avec glow. Issue de [110].*

2.6.1.3 Tone Mapping

En traitement d'image, le *tone mapping* désigne une technique qui adapte des images à hautes gammes dynamiques (*High Dynamic Range Imaging*, *HDR* [42]) à la plus petite gamme de valeurs affichable, ce qui permet d'obtenir des images avec un haut niveau de détail dans tous les tons. Une implémentation temps-réel du tone mapping a été réalisée par Goodnight *et al.* [72] (voir figure 2.21), puis par Woetzel *et al.* [245].

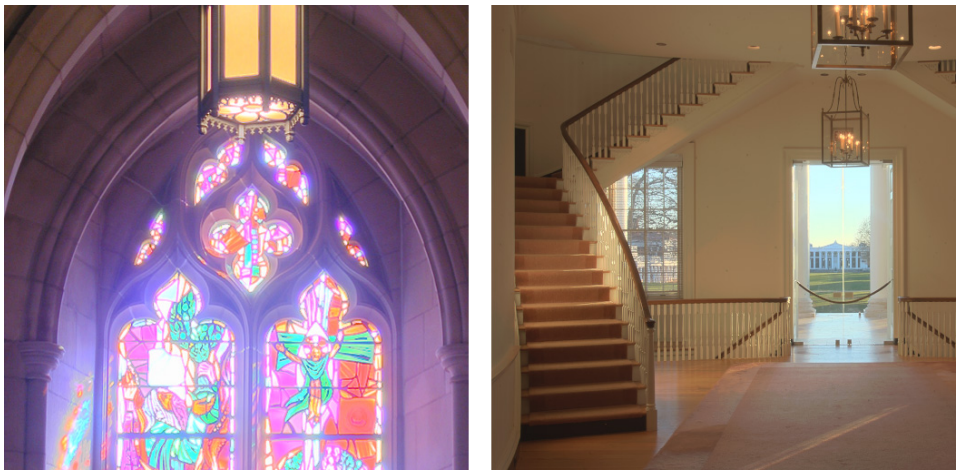


FIGURE 2.21 – *Exemples de* tone mapping *à partir d'images HDR. Les détails dans les tons sombres sont conservés tout autant que ceux dans les tons clairs. Issue de [72].*

2.6.2 Vision

Comparables par certains aspects aux tâches du traitement d'image, celles traîtées en *Vision par Ordinateur*, ou *Computer Vision*, sont liées à l'interprétation automatique du contenu d'une ou plusieurs images. Les problèmes posés par ce domaine exigent de considérables calculs, qu'il est alors intéressant de pouvoir accélérer. Beaucoup d'algorithmes utilisés en vision s'adaptent bien au schéma computationnel des GPU.

Fung a présenté différentes méthodes de vision [61], dont un exemple est présenté en figure 2.22, en utilisant *OpenVidia*[62], projet implémentant en GPU des algorithmes de ce domaine : détection de contours, détection et descriptions de points d'intérêts, suivi de structures, création de panoramas,... Les applications possibles sont nombreuses, nous nous bornerons ici à en exposer seulement quelques unes. D'autres applications, avec nos contributions, seront couvertes aux chapitres 5 et 6.

2.6.2.1 Diagramme de Voronoï et Triangulation de Delaunay

Le diagramme de Voronoï [5] fait partie des structures fondamentales en géométrie de la vision par ordinateur, utilisées dans de nombreux algorithmes de recherche de plus proches voisins ou de trajectoires, de détection de collision, de retouche d'image, de partitionnement de structures,... Ce diagramme est un partitionnement d'un espace métrique $\mathcal{E}$, déterminé par les distances à un sous ensemble discret $\mathcal{F}$ de cet espace, chaque point de $\mathcal{E}$ portant le même label que son plus proche voisin dans $\mathcal{F}$. Le calcul d'un tel diagramme approximé sur GPU a été proposé par Hoff *et al.* [103] (voir figure 2.23) en se basant sur les unités de rasterization

FIGURE 2.22 – *Une application de* Computer Vision *: création de panorama. Par une projection adéquate, les images s'assemblent sans défaut. Issue de [61].*

des cartes graphiques. Rong et Tan ont proposé une implémentation GPU par *jump flooding* [192], algorithme de propagation. Fischer et Gotsman ont implémenté un algorithme utilisant les plans tangents pour déterminer les diagrammes de Voronoï d'ordres supérieurs [53]. Nielsen a enfin présenté une étude des différents algorithmes GPU de détermination de diagramme de Voronoï [166].

Le dual du diagramme de Voronoï est la triangulation de Delaunay, permettant entre autre de déterminer des maillages d'objets. Cette méthode a également été implémentée sur GPU, par exemple par Rong *et al.* [193].

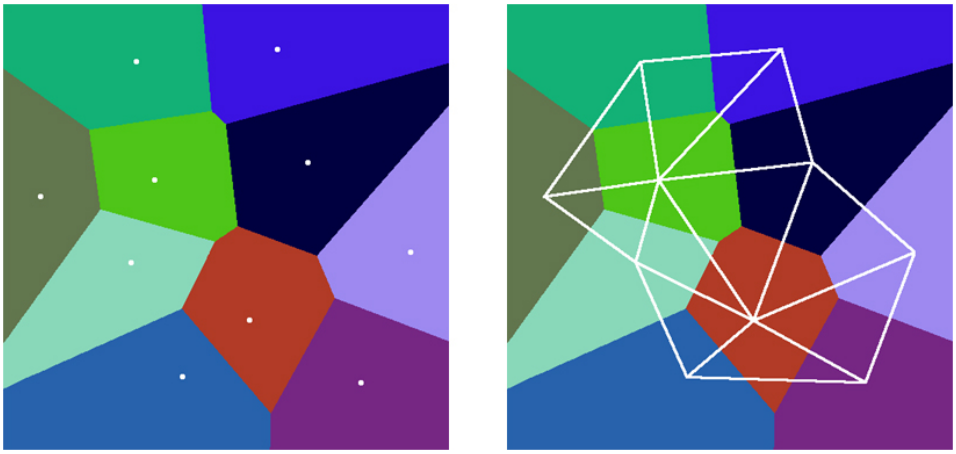


FIGURE 2.23 – *Diagramme de Voronoï et triangulation de Delaunay associée. Gauche : chaque zone colorée correspond aux points les plus proches de la graine blanche associée. Droite : deux graines dont les régions correspondantes sont adjacentes forment une des arêtes de la triangulation de Delaunay. Issue de [103].*

2.6.2.2 Transformée en distance

La transformée en distance est une application associant à chaque point d'un espace la distance au point obstacle le plus proche. Cette transformée sert par exemple à la détermination de squelettes de formes.

Elle est proche du diagramme de Voronoï, il est alors normal que les mêmes personnes aient voulu implémenter cette transformée en distance sur GPU, comme Rong et Tan [192] ou Fischer et Gotsman [53].

Strzodka et Telea ont présenté un framework permettant de calculer ces transformées généralisées ainsi que les squelettes de formes [221].

Peikert et Sigg ont implémenté un algorithme de calcul de carte de distances sur GPU [181] utilisé avec des boîtes englobantes (autour des cellules de Voronoï), spécialement déterminées dans ce cadre.

2.6.2.3 Raffinement de maillages

Un *maillage polygonal*, ou *mesh*, est une représentation discrétisée d'un objet ou d'une scène, permettant un traitement informatique approprié via les API graphiques. Pour une plus grande précision, ou pour une optimisation du temps de rendu, il est parfois nécessaire de modifier le maillage des objets d'une scène. Cela implique des méthodes de vision par ordinateur.

Shiue *et al.* ont par exemple implémenté un algorithme de subdivision adaptative de maillage [211] (voir figure 2.24). Schneider et Westermann se sont intéressés à la génération sur GPU de terrains avec hiérarchisation des niveaux de détails [205]. Boubekeur et Schlick ont proposé un shader générique permettant d'effectuer un remaillage à la volée d'un maillage, sans contrainte sur la topologie de l'objet [18].

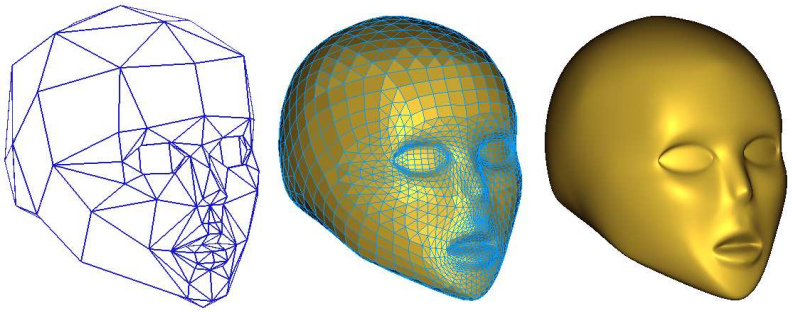


FIGURE 2.24 – *Raffinement de maillage. Gauche : maillage initial. Milieu : maillage raffiné. Droite : rendu lissé à partir du maillage raffiné. Issue de [211].*

2.6.2.4 Stéréovision

Par *stéréovision*, on désigne un ensemble de méthodes permettant, à partir de prises de vue multiples d'un objet sous différents angles, de retrouver sa forme, ses dimensions, sa position. Toutes les étapes de ces algorithmes nécessitent d'importants calculs, pour lesquels les GPU se sont une fois de plus avérés très utilisés.

Yang *et al.* ont proposé un algorithme de stéréo s'exécutant intégralement sur GPU, basé sur une mesure classique de dissimilarité sur des fenêtres de corrélations [249]. Un an plus tard, ils proposent une version améliorée de leur travail, utilisant sur des fenêtres adaptatives et employant au mieux les capacités du GPU [251]. Woetzel et Koch ont implémenté un algorithme d'estimation de carte de profondeur dense à partir d'images multiples, sur GPU [244]. Labatut *et al.* ont présenté le portage d'un algorithme variationnel sur GPU, en utilisant au mieux les techniques GPGPU [128] (voir figure 2.18).

Nous avons ensuite implémenté une méthode variationnelle de stéréovision basée sur des modèles déformables, donnant des résultats précis et denses [146]. Un exemple de résultat est montré en figure 2.25. L'explication de cette méthode fera l'objet du chapitre 5.

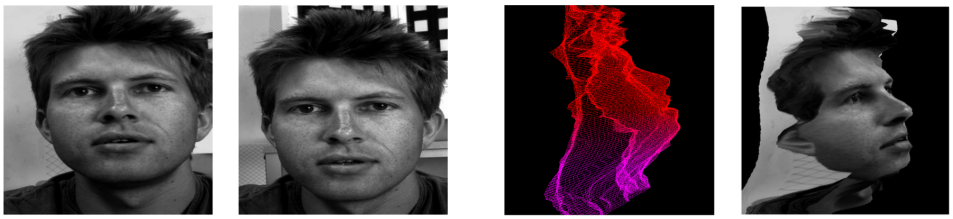


FIGURE 2.25 – *Exemple de résultat de stéréovision. Les deux figures de gauche forment le jeu de données à partir duquel a été calculé le maillage présenté sur les deux figures de droite, en fil de fer et avec placage de texture. Issue de [146]*

Depuis, Brunton *et al.* se sont intéressé aux problèmes de la stéréovision avec utilisation des *belief propagations* (*propagations des croyances bayésiennes*) [23], algorithme itératif calculant des probabilités à partir de modèles graphiques.

Dans un registre très similaire à la stéréovision, Kaufman a, dans sa thèse, développé et décrit sur GPU un algorithme pour un système autostéréoscopique [116], donnant l'illusion du relief d'une scène 3D lors d'un affichage sur un écran.

Conclusion du chapitre

Dans ce chapitre nous avons répertorié des applications de divers domaines scientifiques, physiques ou mathématiques, ayant pour trait d'union l'utilisation des cartes graphiques comme alternative computationnelle. Dans la large majorité des cas, bien que compliquant l'implémentation, parfois grandement, le gain en vitesse est significatif, ces programmes pouvant s'exécuter jusqu'à 100 fois plus vite.

Dans la suite de l'exposé, nous aborderons deux domaines d'études par deux parties distinctes, dans lesquels nous avons nous mêmes répondu à des besoins computationnels par utilisation des GPU. Dans une deuxième partie, nous aborderons la simulation de réseaux de neurones impulsionnels, puis dans une troisième partie nous étayerons la présentation d'applications GPU en Vision par Ordinateur, en développant plus particulièrement des algorithmes de stéréovision dense et d'appariements de points d'intérêts.

Deuxième partie

Neurosciences Computationnelles

Introduction aux neurosciences computationnelles

3

Sommaire

Sous le terme *neurosciences* sont regroupées les disciplines étudiant l'anatomie et l'activité du système nerveux chez les êtres vivants pourvus de nerfs, de cerveau ou de moelle épinière. Parmi ces disciplines, les *neurosciences computationnelles* traitent de la modélisation du fonctionnement du système nerveux par des simulations sur ordinateur. Cette science récente suscite un engouement grandissant dans la communauté scientifique par le défi qu'elle se propose de relever : interpréter les processus neurologiques (et plus particulièrement cognitifs) et s'en inspirer pour le développement d'applications autres.

Ce chapitre a pour but de présenter au lecteur les bases nécessaires à la compréhension des mécanismes biologiques, ainsi qu'à leur traduction en modèles de computation *biologiquement plausibles*, ce qui facilitera la lecture du chapitre suivant. Une première partie présentera les propriétés biologiques et les processus de transmission de l'information neurologique. Une deuxième partie exposera les différents types de modèles de neurones ainsi qu'un modèle computationnel, le modèle impulsionnel. Une troisième partie présentera un modèle particulier : le modèle intègre-et-tire. Enfin, une quatrième partie présentera succinctement le formalisme des réseaux de neurones, forme sous laquelle les neurones intègre-et-tire seront largement exploités dans le chapitre 4. La majorité des notions présentées dans ce chapitre sont issues de [21], [41] et [3], nous invitions le lecteur à s'y référer pour de plus amples détails.

3.1 Le neurone et ses propriétés

3.1.1 Présentation

Le *neurone* est l'un des deux types principaux de cellules trouvées dans le système nerveux central, avec les *cellules gliales*. Par un signal électrique nommé *influx nerveux*, il véhicule à travers le système nerveux central l'information permettant la communication intercellulaire et autorise ainsi l'interaction entre le cerveau et le reste du corps. Les neurones reçoivent l'information perçue par les différents organes, l'acheminent jusqu'au cerveau, y traitent l'information pour produire une réponse adaptée, qui sera communiquée aux fibres musculaires et aux glandes de notre corps. Ils sont ainsi les cellules essentielles à tout processus cognitif.

Un cerveau humain possède environ cent milliards de neurones, distingués en une dizaine de milliers de catégories, aucune de ces catégories n'étant propre à l'homme. Ces types sont réparties en 3 classes : les *neurones moteurs*, convoyant l'information motrice, les *neurones sensoriels*, acheminant l'information liée aux sens, et les *interneurones*, réalisant une connexion entre deux neurones proches.

3.1.2 Morphologie

La fonction de transmission d'influx nerveux du neurone est exprimée par sa structure morphologique (voir figure 3.1), dédiée à ce traitement. Il se compose de quatre parties principales. Un *corps cellulaire*, nommé *soma* ou *péricaryon*, contient un noyau et tous les autres organites permettant de produire de l'énergie et de synthétiser les protéines nécessaires au bon fonctionnement du neurone. De ce soma partent l'*axone* et les *dendrites*. L'axone est un prolongement long (de quelques millimètres à plus d'un mètre) et mince (entre 1 et 15μm) du soma, conduisant les influx électriques en dehors de celui ci, jusqu'à son *arborisation terminale* ; il est pratiquement entièrement recouvert d'une gaine de myéline isolant électriquement la fibre nerveuse. Les dendrites sont d'autres expansions du soma, longs de 1mm, très ramifiées, jamais recouvert de myéline et dont la fonction est de recevoir des influx nerveux par contact avec les axones d'autres neurones. Enfin, les *synapses* sont les jonctions qui sont les lieux de ces échanges, elles se trouvent entre l'arborisation terminale de l'axone et les récepteurs des dendrites. Suivant le signal propagé à ces jonctions, elles peuvent jouer un rôle *exitateur* ou *inhibiteur*. Nous reviendrons sur le rôle des synapses dans la section 3.1.5.2. On estime à 10^{15} le nombre de connections synaptiques entre neurones dans un cerveau, à raison de quelques centaines à plusieurs milliers par neurone.

D'un type de neurone à l'autre, la morphologie des cellules neuronales peut varier énormément en terme de forme d'arbre somato-dendritique ou de taille du soma. La figure 3.2 montre (à gauche) une cellule de Purkinje dont l'arbre dendritique reçoit plus de 100000 entrées synaptiques, alors que pour les cellules pyramidales (à droite), les dendrites n'en reçoivent que quelques milliers. La morphologie du neurone joue donc un rôle essentiel dans la détermination de sa fonction.

Ce sont, de plus, des cellules incapables de mitose, ayant donc une très grande longévité, et requérant un apport constant en glucose et oxygène.

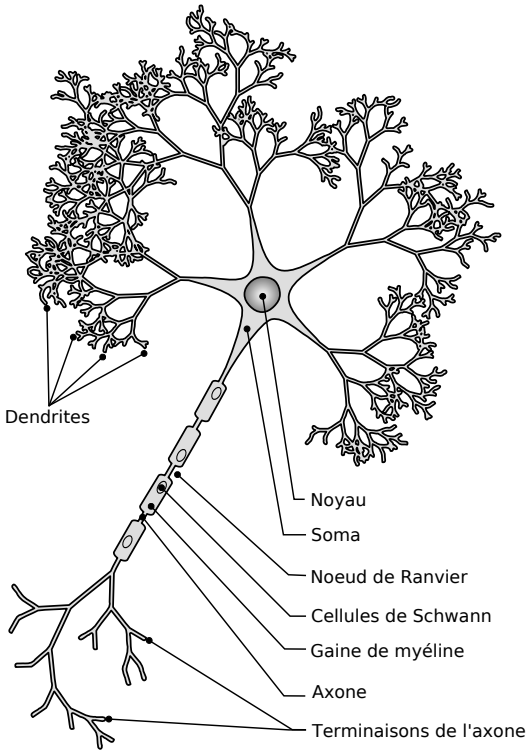


FIGURE 3.1 – *Morphologie d'un neurone. Le soma contient le noyau de la cellule. C'est de ce soma que partent les dendrites, formant un arbre plus ou moins dense, et l'axone, recouvert d'une gaine de myéline et dont les terminaisons sont pourvues de synapses. Le long de l'axone se trouvent des cellules de Schwann synthétisant la myéline, et les noeuds de Ranvier, lieux de régénération de l'influx nerveux sur l'axone (voir section 3.1.5.1).*

3.1.3 Propriétés physiologiques

En plus des propriétés morphologiques conditionnant l'utilité des neurones, ces derniers possèdent quelques particularités physiologiques, quantifiables, caractérisant leur mode de fonctionnement, en particulier la propagation d'information par *potentiel d'action*.

3.1.3.1 Canaux ioniques

Certaines protéines présentes sur la membrane, appelées *canaux ioniques*, laissent passer spécifiquement certains ions vers l'intérieur ou l'extérieur de la cellule. Ces canaux ioniques contrôlent le flux d'ions (principalement les ions sodium Na^+, potassium K^+, calcium Ca^{2+} et chlorure Cl^-) en s'ouvrant ou se fermant, en fonction des variations électriques et de signaux internes comme externes. Ce mécanisme a une importance capitale dans la régulation du voltage interne de la cellule, comme nous le verrons dans le paragraphe suivant.

3.1.3.2 Potentiel de membrane, potentiel de repos

Le signal électrique pertinent pour une cellule du système nerveux est la différence de potentiel existante entre l'intérieur de cette cellule et le milieu extracellulaire, dûe à des écarts de concentrations ioniques. On dit de la cellule qu'elle est *polarisée* et possède un *potentiel membranaire*.

Lorsque le neurone est au repos, cette différence de potentielle est appelée *potentiel de repos* et est de l'ordre de -70mV (par rapport au milieu

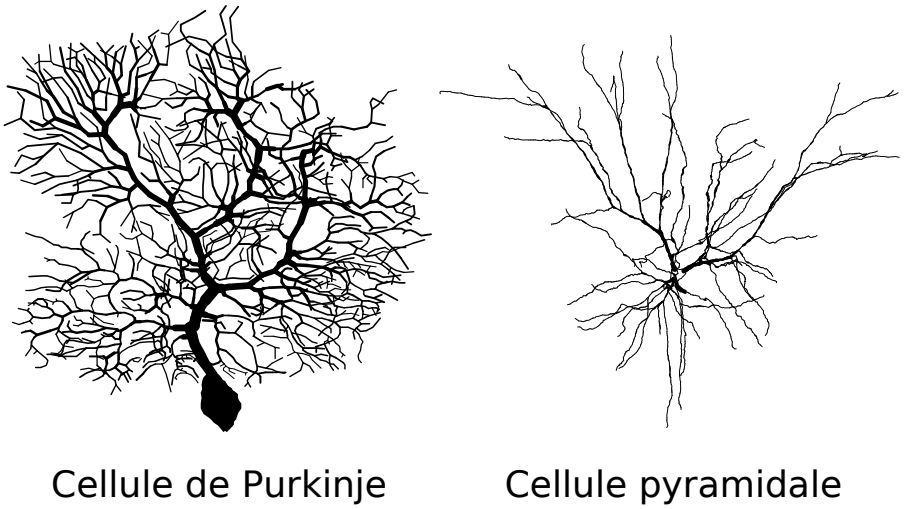


FIGURE 3.2 – *Deux types de neurones. Gauche : cellule de Purkinje, neurone spécialisé prédominant dans le cervelet. Droite : neurone pyramidal, présent dans certaines couches du néocortex et certaines régions de l'hippocampe. La différence de forme de l'arbre dendritique est significatrice de fonctions différentes. Issu de [122].*

extérieur à la cellule, par convention à 0mV). C'est par certaines protéines de la membrane, appelées *pompes à ions*, que sont maintenus les gradients de concentration : au repos, la membrane du neurone est perméable uniquement aux ions potassium, faisant tendre le potentiel de repos vers le potentiel d'équilibre de cet ion, de l'ordre de -80mV. Les ions sodium, majoritairement présents à l'extérieur, ont un potentiel d'équilibre de l'ordre de 50mV.

Les potentiels d'équilibre E sont donnés par l'équation de Nernst :

$$E = \frac{k_B T}{q} \ln \frac{[ext]}{[in]}$$

avec k_B la constante de Boltzmann, T la température, q la charge de l'ion considéré, et $[ext]$ et $[in]$ les concentrations respectivement extérieure et intérieure de l'ion considéré.

3.1.3.3 Hyperpolarisation, dépolarisation

Lorsque les canaux ioniques permettant de faire passer les ions positifs à l'extérieur ou les ions négatifs à l'intérieur s'ouvrent, cela rend le potentiel de repos d'autant plus négatif, on parle d'*hyperpolarisation*.

A l'inverse, une ouverture des canaux ioniques permettant les flux d'ions positifs vers l'intérieur ou d'ions négatifs vers l'extérieur contribue à faire tendre le potentiel de repos vers une valeur moins négative voire positive. Ce processus est appelé *dépolarisation*.

3.1.4 Potentiel d'action

Lorsqu'un neurone subit une dépolarisation assez forte pour élever son potentiel au dessus d'une valeur seuil, généralement -55mV, il se produit un processus en trois phase, nommé *potentiel d'action* (*spike*), durant de 2 à 3ms (voir figure 3.3) :

– une dépolarisation rapide d'une amplitude d'environ 100mV, amenant le potentiel à $+30$mV. Cette amplitude est la même pout tout potentiel d'action ;
– une repolarisation et hyperpolarisation rapides de la membrane du neurone, voyant son potentiel diminuer jusqu'à -80mV ;
– une légère dépolarisation, lente, ramenant le potentiel de la membrane à son état de repos, -70mV. Cette phase est appelée *période réfractaire* car il est impossible ou au moins très difficile d'y susciter un nouveau potentiel d'action. L'apparition de ce phénomène dépend donc également du passé du neurone en question.

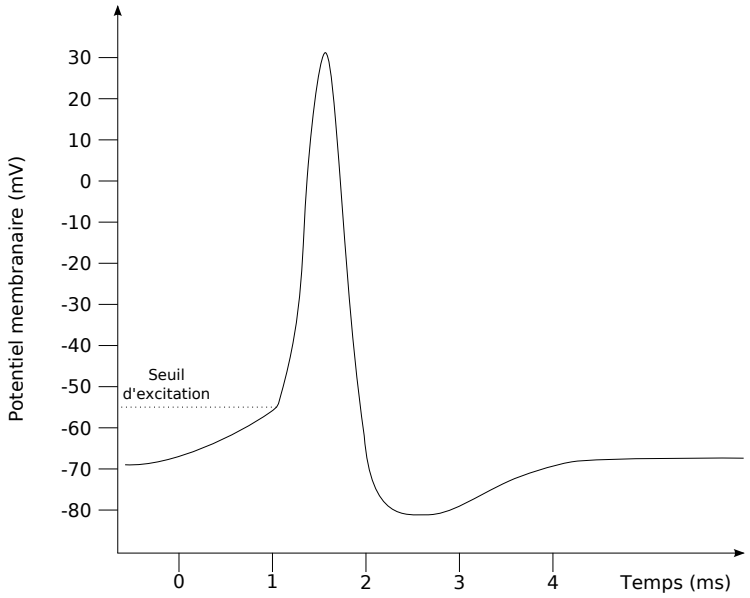


FIGURE 3.3 – *Enregistrement postsynaptique d'un potentiel d'action. Lorsque le potentiel membranaire est élevé au dessus du seuil d'excitation, un potentiel d'action est généré : une forte dépolarisation a lieu, jusqu'à atteindre rapidement* $30mV$*, elle est suivie d'une brusque repolarisation atteignant* $-80mV$*, puis d'une lente dépolarisation, jusqu'au retour au potentiel de repos.*

Un tel processus est possible par une modification des concentrations ioniques dans les milieux intra et extracellulaire, par un jeu complexe d'ouvertures et fermetures de canaux ioniques spécifiques, principalement les canaux sodium et potassium.

Les potentiels d'actions sont d'une importance capitale car ce sont les influx nerveux codant l'information. Il est donc nécessaire de pouvoir les propager à travers le système nerveux central.

3.1.5 Propagation des potentiels d'action

Le signal se propage à l'intérieur des cellules à une vitesse de 100m.s^{-1}, et entre les cellules au niveau des synapses. Ces deux modes de progression sont assurés par les deux mécanismes présentés ci-dessous.

3.1.5.1 Régénération des potentiels d'action le long de l'axone

Toute variation du potentiel membranaire tel un potentiel d'action est propagée à l'intérieur de la cellule neuronale, le long de l'axone, jusqu'aux

synapses. La fibre nerveuse étant mauvaise conductrice, cette variation s'estompe rapidement. Pour palier à cela, les *cellules de Schwann*, présentes dans le milieu extracellulaire, produisent une gaine de *myéline* autour de l'axone, augmentant considérablement sa conductivité (voir figure 3.1).

Néanmoins, ce mécanisme n'est pas suffisant lorsqu'il est nécessaire de propager le signal sur de longues distances (certains neurones sont long de plus d'un mètre). La myélinisation des axones n'est jamais complète, il existe certains points présentant une faible résistance électrique au niveau de laquelle à peu près tous les canaux Na+ de l'axone sont concentrés, ce sont les *nœuds de Ranvier* (voir figure 3.1). C'est dans ces lieux que les potentiels d'action vont pouvoir se régénérer. Ce mode de propagation saltatoire (de nœud en nœud) permet à la cellule nerveuse de conserver son énergie, car l'excitation active nécessaire à la propagation des potentiels d'action est restreinte aux régions nodales.

3.1.5.2 Transmission synaptique

La synapse est le nom de la jonction entre les arborisations terminales d'un axone et les dendrites d'un autre neurone. C'est par elle que va transiter l'influx électrique. Il en existe deux types.

Pour une *synapse électrique*, les deux neurones sont reliés par une *jonction communicante*, ou *jonctions gap*, assurant une continuité de conduction électrique. Les ions passent librement d'une cellule à l'autre, permettant la continuité des potentiels d'action. Ces synapses électriques se trouvent principalement entre les neurones inhibiteurs.

Les synapses les plus fréquentes sont les *synapses chimiques*, que l'on peut voir en figure 3.4. Le mécanisme de transmission de l'influx, où *neurotransmission*, est plus complexe que pour le premier type de synapse. Ici, le neurone émetteur et le neurone récepteur de cet influx sont séparés par une fente synaptique. Le signal électrique afférent va provoquer l'ouverture de canaux calcium, entrainant la fusion de la membrane d'un terminal dendritique du neurone présynaptique avec celle des vésicules qu'elle contient. Ces vésicules vont libérer dans la fente synaptique des molécules appelées *neurotransmetteurs* ou *neuromédiateurs*, c'est le phénomène d'*exocytose*. Les neurotransmetteurs vont se mouvoir dans cette fente par simple diffusion et se fixer par sur des récepteurs spécifiques de la membrane du neurone postsynaptique, entrainant l'ouverture de canaux ionique et ainsi la création de courants propagés dans le neurone postsynaptique. Une synapse peut être dite *excitatrice* si le signal transmis est dépolarisant, ou *inhibitrice* s'il est hyperpolarisant. Ce rôle est fonction de la nature des neurotransmetteurs relâchés (il a été récemment prouvé que de nombreux neurones peuvent libérer deux types de neurotransmetteurs ou plus, contrairement au principe de Dale auparavant énoncé, stipulant qu'il n'y a qu'un neurotransmetteur par neurone).

3.2 Modélisation biophysique d'un neurone

Pour pouvoir étudier, analyser, simuler, recréer un neurone et surtout prédire son comportement, éventuellement au sein d'un réseau, il est nécessaire d'en fournir une représentation, fonction des différentes théories que l'on souhaite appliquer et des mécanismes physiques que l'on sou-

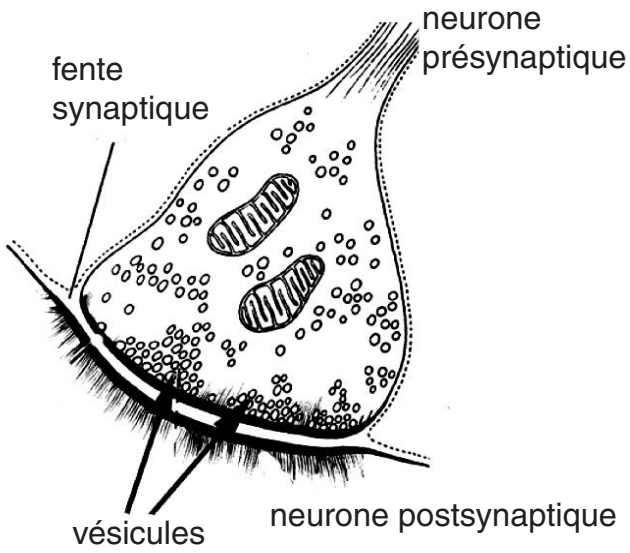


FIGURE 3.4 – *Schéma d'une synapse chimique. Le signal électrique afférent provoque l'ouverture des vésicules et le déversement des neurotransmetteurs qu'elles contiennent dans la fente synaptique. Ces molécules viennent se fixer sur des récepteurs de la membrane postsynaptique, entrainant la création d'un signal électrique qui sera propagé dans la cellule efférente. Extrait de [21]*

haite voir reproduits. La connaissance des propriétés biologiques du neurone permet la création de modèles dont le comportement se rapproche de celui d'un neurone réel.

Selon le point de vue adopté, on peut chercher à modéliser le plus fidèlement possible l'ensemble des propriétés et variables biologiques d'une cellule neuronale dans un modèle biophysique, ce que nous traitons dans cette section, ou bien considérer le neurone comme un élément de base dans le traitement de l'information, en l'idéalisant avec un modèle computationnel, dont nous présentons un exemple particulier dans la section 3.3.

Un modèle biophysique de neurone va chercher à reproduire sous la forme d'équations l'évolution des variables biologiques significatives, en particulier le potentiel membranaire V.

Bien que V soit la variable la plus pertinente, cette grandeur dépend, dans un neurone réel, d'autres grandeurs, qui peuvent être utilisées dans les différentes modélisations. Nous noterons ces grandeurs :

- V_0 le potentiel de repos du neurone ;
- E_u le potentiel d'équilibre d'un ion u ;
- C_m la capacité membranaire (polarisé, le neurone se comporte en effet comme un condensateur, avec courant de fuite) ;
- g_m la conductance membranaire :
- g_u la conductance d'un canal ionique u, et $\bar{g}_u$ sa valeur maximale ;
- g_l la conductance de fuite, et $\bar{g}_l$ sa valeur maximale ;
- τ_m la constante de temps membranaire, $\tau_m = C_m/g_m$.

3.2.1 Modèle de Hodgkin et Huxley

En 1952, Hodgkin et Huxley ont découvert comment est généré le potentiel d'action dans l'axone du calmar. Leurs travaux multidisciplinaires,

récompensés d'un prix Nobel, ont montré que le courant membranaire I peut être vu comme la somme d'un courant capacitif de capacité C et de courants ioniques dépendants de la différence entre le potentiel membranaire et le potentiel d'équilibre du canal considéré [102]. La dynamique du potentiel membranaire d'un neurone est ici décrite par le système différentiel non linéaire suivant :

$$\begin{cases} C_m \frac{dV}{dt} &= -\bar{g}_l(V - V_0) - \bar{g}_K n^4 (V - E_K) \\ & \quad -\bar{g}_{Na} m^3 h (V - E_{Na}) + I \\ \\ \frac{dw}{dt} &= (1 - w)\alpha_w(V) - w\beta_w(V), \text{ avec } w = n, m \text{ ou } h \end{cases}$$

avec $\bar{g}_l$ la conductance maximale de fuite. Les fonctions auxiliaires m, n et h correspondent à des probabilités d'ouverture ou d'activation des canaux ioniques. Elles sont toutes trois régies par une équation différentielle du même type, avec $\alpha_w(V)$ et $\beta_w(V)$ les fonctions indiquant les vitesses d'ouvertures des canaux ioniques, calculées empiriquement par Hodgkin et Huxley et dont des approximations sont données dans [102].

L'élévation à la puissance 4 de la fonction n correspond à une approximation acceptable de la conductance du potassium par une équation de premier ordre élevée à la puissance 4, à la place d'une équation du quatrième ordre. Cela représente la subdivision du canal potassium en 4 sous-canaux identiques, devant être ouvert en même temps.

Il en est de même pour le canal sodium avec l'élévation à la puissance 3 de la fonction m. La présence simultanée des fonctions m et h sur ce canal modélise la superposition de deux canaux ayant chacun leur propre dynamique ; cela représente la propriété du canal sodium de pouvoir être activé ou pas en plus d'être ouvert ou fermé.

Ce modèle, en expliquant plus qu'en copiant les mécanismes neuronaux, est le modèle de référence pour les processus membranaires, les autres modèles n'arrivant pas à obtenir la même précision dans la simulation de la variation du potentiel membranaire ou la génération de potentiel d'action.

Une autre des raisons de son succès est le fait qu'il a été le premier à introduire une modélisation des conductances des canaux ioniques, et a engendré une série de modèles dérivés décrivant cette propriété : ce sont les *modèles à conductance*, ou *conductance-based models*, les plus simples représentations biophysiques d'un neurone, dans lesquelles les canaux ioniques sont représentés par des conductances et la membrane cellulaire de lipides par une capacité.

3.2.2 Autres modèles à conductances

Comme nous venons de l'énoncer, les résultats obtenus à partir du modèle de Hodgkin et Huxley en font une référence. Néanmoins, sa grande complexité a favorisé l'introduction de modèles plus simples mais aussi plus approximatifs, prenant également en compte les conductances des ions. Ils sont souvent décrits par un système différentiel cette fois ci linéaire, de deux équations, autorisant un traitement analytique.

3.2.2.1 Modèle de Morris-Lecar

Le modèle de Morris-Lecar [162] est l'un des plus utilisés en simulation neuronale. Il met en jeu deux variables d'état englobant pour l'une

les grandeurs à dynamiques rapides et pour l'autre celles à dynamiques lentes. Le système différentiel qui en résulte est plus simple à résoudre que celui provenant du modèle de Hodgkin et Huxley. Les équations de ce système sont les suivantes :

$$\begin{cases} C_m \frac{dV}{dt} &= -\bar{g}_{Ca} m_\infty (V - E_{Na}) - \bar{g}_K W (V - E_K) - \bar{g}_l W (V - E_l) + I_{ext} \\ \frac{dW}{dt} &= \phi \cosh(\frac{V - V_3}{2V_4})(w_\infty - W) \end{cases}$$

avec ϕ une constante, $\bar{g}_{Ca}$, $\bar{g}_K$ et $\bar{g}_l$ les conductances maximales respectives du canal calcium du canal potassium, et de fuite. Les fonctions m_∞ et w_∞ représentent, comme pour le modèle de Hodgkin et Huxley, une probabilité d'ouverture des canaux ioniques correspondant, et sont régis par les équations suivantes :

$$\begin{cases} m_\infty &= (1 + \tanh(\frac{V - V_1}{V_2}))/2 \\ w_\infty &= 1 + \tanh(\frac{V - V_3}{2V_4}) \end{cases}$$

V_1, V_2, V_3 et V_4 sont des paramètres secondaires constants choisis en fonction des propriétés souhaitées pour le système.

V est la variable rapide, assimilable au potentiel membranaire, présentant une non linéarité cubique permettant un rétrocontrôle positif ; W est une variable de recouvrement, de dynamique lente et linéaire, apportant une rétroaction négative ; I_{ext} est le courant extérieur appliqué à la membrane.

3.2.2.2 Modèle de FitzHugh-Nagumo

C'est l'un des plus célèbres modèles [55]. Il considère également un ensemble de deux variables d'état catégorisant les dynamiques du système en deux types. Le système différentiel linéaire dirigeant ces variables est le suivant :

$$\begin{cases} C_m \frac{dV}{dt} &= AV(V - \alpha)(1 - V) - W + I_{ext} \\ \frac{dW}{dt} &= \epsilon(V - \gamma W) \end{cases}$$

avec A, ϵ, α, et γ des constantes, ϵ est "assez petit", $0 < \alpha < \frac{1}{2}$, et γ tel que $A(V - \alpha)(1 - V) - \frac{1}{\gamma} = 0$ n'ait pas de solution réelle (ce qui permet d'avoir une seule solution stationnaire).

V, W et I_{ext} ont les mêmes significations que dans le modèle de Morris-Lecar. Ces équations sont adaptées d'un oscillateur de Van der Pol.

3.2.2.3 Autres

D'autres modélisations à conductance existent dans la littérature, variant en complexité et utilisation. On relèvera particulièrement le modèle de Hindmarsh-Rose [101], utilisant trois variables dans un système différentiel non linéaire. Ces variables correspondent au potentiel membranaire de la cellule, à la dynamique globale rapide des ions sodium et potassium et à la dynamique plus lente des autres ions.

Le modèle de McKean [231] est un exemple de modification du modèle de FitzHugh-Nagumo et de simplification des conductances. Les canaux ne sont pas modélisés ; les variations brutes des concentrations permettant la génération d'un potentiel d'action sont remplacées par une fonction potentielle, linéaire par morceaux. Par ceci, les propriétés du comportement neuronal sont préservées et un calcul explicite peut être effectué.

Cela fait du modèle de McKean un *intègre-et-tire*, catégorie de modèle que nous décrivons dans la section suivante.

3.3 Une famille de modèles computationnels particuliers : les Intègre-et-tire

Alors que le but des modèles biophysiques est l'étude du comportement neuronal pour en comprendre les mécanismes, on définit des modèles dit *computationnels* dans lesquels on considère un neurone comme une unité logique de calcul, capable de résoudre un problème donné, et dont le comportement ne reflète pas nécessairement toutes les propriétés des neurones biologiques. L'élément de communication entre neurones, le potentiel d'action, est ici idéalisé et considéré comme instantané, permettant des simulations bien plus rapides.

Deux classes de modèles computationnels peuvent être distinguées, suivant la variable principale considérée comme pertinente.

Dans la première classe, on s'intéresse à la fréquence de décharge des neurones, c'est-à-dire à la fréquence des potentiels d'action qu'ils émettent. Dans ces *modèles fréquentiels*, on considère que cette fréquence suit une loi de Poisson non homogène. L'entrée d'un neurone (les potentiels d'actions reçus) suit également un processus de Poisson non homogène, en tant que somme des sorties d'autres neurones, eux aussi considérés comme soumis à cette distribution poissonienne. Ces modèles sont largement étudiés à travers la littérature (voir [12], [145] , ou [212]) et sont utilisés dans des structures de type perceptron (voir section 3.4) ou des algorithmes d'apprentissage qui ont des applications informatiques importantes (principalement classification et reconnaissance de formes).

La seconde classe privilégie la détermination de l'instant des impulsions au lieu de leur fréquence, on parle de *modèle impulsionnel* ou *modèle intègre-et-tire* [236] (*Integrate-and-fire*). On ne cherche pas à étudier la création du potentiel d'action, mais à décrire la création d'une série d'impulsions en fonction de l'entrée du neurone.

La suite de cette section va couvrir cette seconde classe. Les contradictions qu'impliquent les hypothèses sous-jacentes des modèles fréquentiels, à savoir l'obtention d'une fréquence de décharge non poissonienne et la synchronisation des fréquences de sorties (déjà observée dans [239]) de plusieurs neurones *a priori* indépendants, ont amené la communauté à étudier cette autre classe de modélisation de cellules neuronales. Pour de plus amples détails sur ces hypothèses, le lecteur est invité à se reporter à [21].

3.3.1 Définition formelle

Au lieu d'expliquer des processus physiologiques tels la génération de potentiel d'action et son intégration par une synapse, un modèle intègre-et-tire va chercher à quantifier ces phénomènes en les remplaçant par des règles simples. Pour cela, l'activité interne du neurone est mesurée simplement par son potentiel membranaire en dessous du seuil, comprenant une partie intégratrice correspondant à la somme des entrées synaptiques (arbre dendritique) et dont la variation est modélisée par une fonction linéaire par morceaux, à laquelle on ajoute un mécanisme représentant la création et la propagation d'un potentiel d'action :

$$\begin{cases} \frac{dV}{dt} & = & f(V,t) \\ \\ V(t) & > & V_{seuil} \Rightarrow \text{Génération d'un potentiel d'action} \end{cases}$$

avec $f(V,t)$ la fonction linéaire par morceaux. Les parties non-linéaires simulent la génération d'un potentiel d'action par l'émission d'une impulsion de Dirac : ce mécanisme, déclenché lorsque le potentiel membranaire dépasse la tension V_{seuil}, suppose alors une brusque montée et une réinitialisation instantanées de la valeur de ce potentiel (éventuellement une inhibition du neurone pendant une durée correspondant à la période réfractaire) ainsi que la propagation d'un message à tous les neurones en aval. La figure 3.5 représente un potentiel d'action idéalisé dans ce cadre.

C'est de ce mécanisme que vient le nom *intègre-et-tire* du modèle : les entrées sont intégrées jusqu'à ce que le seuil de création du potentiel d'action soit dépassé par une variable représentant le potentiel membranaire, on dit que le neurone *tire*.

Les données d'étude sont les instants de décharge du modèle, que l'on notera $\{t_k\}_{k\in\mathbb{N}}$. Ce train d'impulsions est également une entrée synaptique possible pour tout autre neurone du réseau modélisé : chaque émission d'une impulsion par neurone va provoquer chez ses voisins efférents une augmentation du courant d'entrée I.

Dans la suite, les dépendances temporelles du potentiel membranaire V et des différents courants I seront sous-entendues.

3.3.2 Première formulation : modèle de Lapicque

Le premier modèle de type intègre-et-tire a été présenté par Lapicque [130] en 1907. Il permet de transformer une entrée analogique (I) en une série d'instants de décharge pour lesquels on approxime la brusque variation du potentiel par le mécanisme précédemment montré, simple mais efficace, se rapprochant de façon satisfaisante des résultats du modèle de Hodgkin-Huxley. Les potentiels d'actions sont ici représentés par des Diracs, comme le montre la figure 3.5.

Lapicque considéra que le comportement du neurone hors potentiel d'action est assimilable à celui d'un circuit électrique intégrateur avec fuite, dont le schéma électrique équivalent est donné en figure 3.6. Une approximation forte est ici faite : les conductances ioniques sont toutes ignorées, la conductance de la membrane est représentée par une unique conductance de fuite $\bar{g}_l = 1/R_m$.

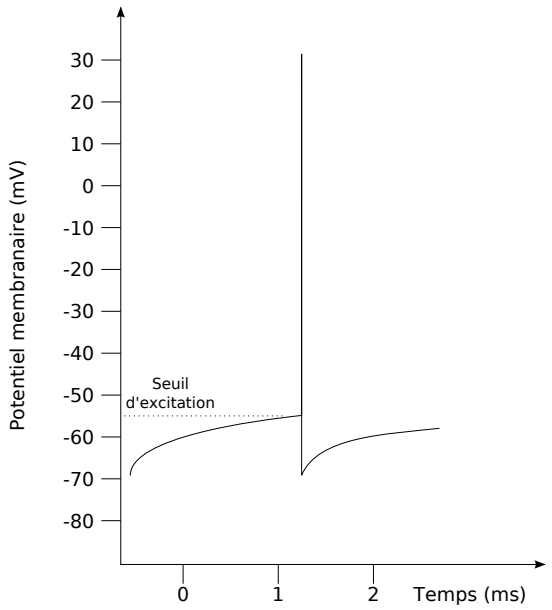


FIGURE 3.5 – *Potentiel d'action idéalisé dans le modèle intègre-et-tire de Lapicque. Dès que le potentiel membranaire dépasse le seuil d'excitation, une impulsion de Dirac est générée et la valeur du potentiel est réinitialisée.*

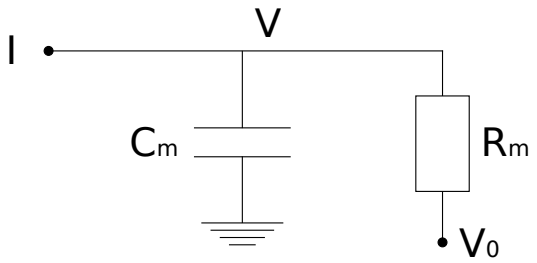


FIGURE 3.6 – *Schéma électrique équivalent à un neurone intègre-et-tire en comportement linéaire (hors potentiel d'action). C_m et R_m sont respectivement la capacité et la résistance membranaires, V est le potentiel membranaire dépendant du temps, V_0 est le potentiel de repos, et I est le courant injecté, pouvant également dépendre du temps. Adapté de [41]*

Par cette description, on peut écrire que le courant d'entrée I est la somme de deux courant :

$$I = I_{R_m} + I_{C_m}$$

avec I_{R_m} le courant passant dans la résistance R_m et I_{C_m} celui passant par la capacité C_m. Par la loi d'Ohm, il vient $I_{R_m} = (V - V_0)/R_m$. On a également $I_{C_m} = C_m \frac{dV}{dt}$, soit :

$$I = \frac{V - V_0}{R_m} + C_m \frac{dV}{dt}$$

La constante de temps membranaire τ_m définie par $\tau_m = R_m C_m$ permet de donner une première formulation de l'équation différentielle du premier ordre régissant ce modèle :

$$\tau_m \frac{dV}{dt} = -(V - V_0) + R_m I$$

On préfèrera cependant l'écrire en faisant apparaitre la conductance membranaire de fuite $\bar{g}_l$, ce qui, avec le mécanisme de réinitialisation, et en notant $\{t_k\}_{k\in\mathbb{N}}$ la suite des moments des impulsions générées par le

neurone, donne :

$$\begin{cases} C_m \frac{dV}{dt} & = & -\bar{g}_l(V - V_0) + I \\ \\ \text{Si } V & > & V_{seuil} \text{ alors } \begin{cases} t_k & \leftarrow & t \\ V & \leftarrow & V_{raz} \end{cases} \end{cases}$$

avec V_{raz} la valeur de réinitialisation de V, souvent choisie proche de V_0.

Un exemple d'évolution temporelle du potentiel V de la membrane d'un neurone intègre-et-tire en fonction de l'entrée synaptique I est donné en figure 3.7.

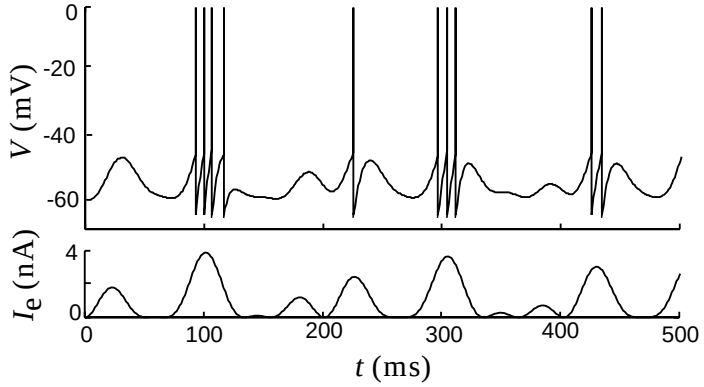


FIGURE 3.7 – *Evolution temporelle du potentiel membranaire d'un neurone intègre-et-tire passif. Haut : potentiel membranaire. Bas : courant injecté. Les potentiels d'actions sont générés dès que $V > V_{seuil}$. Extrait de [41].*

En faisant ce choix, et dans le cas particulier d'un courant injecté constant I_0, il est possible de déterminer explicitement le comportement du neurone dans les phases linéaires. En supposant qu'au temps initial t_0 un potentiel d'action vient tout juste d'être généré, il vient :

$$V = V_0 + \frac{I_0}{\bar{g}_l}[1 - \exp(-\frac{t}{\tau_m})]$$

Le prochain potentiel d'action sera généré à l'instant t_1 pour lequel :

$$V = V_{seuil} = V_0 + \frac{I_0}{\bar{g}_l}[1 - \exp(-\frac{t_1}{\tau_m})]$$

soit

$$t_1 = \tau_m \ln(\frac{I_0}{I_0 - (V_{seuil} - V_0)\bar{g}_l})$$

et se répètera de façon régulière à une fréquence $f_{I_0} = \frac{1}{t_1}$.

Ceci n'est vrai que si $I_0 > (V_{seuil} - V_0)\bar{g}_l$. Dans le cas contraire, le modèle prédit toujours une variation négative du potentiel membranaire, qui ne pourra pas dépasser V_{seuil}. Pour des grandes valeurs de I_0, on peut alors montrer que la fréquence de décharge est une fonction linéaire du courant d'entrée I_0, constant dans le temps.

3.3.3 Modèles dérivés

Il existe une grande variété de modèles intègre-et-tire, comme celui de McKean [231], précédemment cité.

La plupart de ces modèles dérivent cependant de celui introduit par Lapicque. Certains rendent variables certaines quantités que le modèle de Lapicque considérait fixe, comme le seuil V_{seuil} ou bien la valeur de réinitialisation V_{raz}.

D'autres, de plus grand intérêt, modifient la forme des équations d'évolution. Nous en décrivons quelques uns dans les paragraphes suivants.

3.3.3.1 Intégrateur parfait

L'intégrateur parfait est un modèle encore plus simple que le modèle de Lapicque, ne prenant pas en compte le courant de résistance I_{R_m}. L'équation régissant le potentiel du neurone est alors :

$$C_m \frac{dV}{dt} = I$$

Le neurone y est vu comme un accumulateur sans fuite, sommant les entrées synaptiques et générant un potentiel d'action lorsque le potentiel accumulé dépasse un seuil, avec un mécanisme identique à celui du modèle de Lapicque. Romain Brette [21] a montré qu'un tel modèle peut être vu comme une implémentation impulsionnelle d'un modèle fréquentiel. Malgré sa simplicité, son étude suscite un certain intérêt par les propriétés qu'il peut mettre en évidence sur des modèles plus complexes.

3.3.3.2 Intègre-et-tire à conductances synaptiques

Alors que le modèle précédent et le modèle de Lapicque ne prennent pas en compte les courants dus aux canaux ioniques (et pour cause, l'existence de ces canaux n'était pas connu à cette date), d'autres modèles les intègrent. On y prend donc en compte ces courants, internes au neurone, générés par les potentiels d'action au niveau présynaptique en introduisant des variables supplémentaires.

Song *et al.* ont par exemple utilisé de tels modélisations pour de l'apprentissage par modèle Hebbien [218], avec le modèle de neurone suivant :

$$\tau_m \frac{dV}{dt} = -(V - V_0) - g_{ex}(t)(V - V_{ex}) - g_{in}(t)(V - V_{in})$$

avec V_{ex} et V_{in} les potentiels de réversion synaptique respectivement pour les synapses excitatrices et inhibitrices, et g_{ex} et g_{in} représentant les conductances synaptiques respectivement excitatrices et inhibitrices, régies par des fonctions exponentielles décroissantes :

$$\begin{cases} \tau_{ex} \frac{dg_{ex}}{dt} &= -g_{ex} \\ \tau_{in} \frac{dg_{in}}{dt} &= -g_{in} \end{cases}$$

avec τ_{ex} et τ_{in} les constantes de temps associées.

On remarque que I est lié à V, ce qui en fait un cas à part du modèle de Lapicque.

Destexhe a également étudié la modélisation des courants synaptiques dans les modèles intègre-et-tire [44], en approximant les variables α_w et β_w, pour $w = n$,n ou h du modèle de Hodgkin et Huxley (voir section 3.2.1) par des impulsions de Dirac générées lorsque le neurone tire.

3.3.3.3 Intègre-et-tire non linéaire

L'équation générale régissant un intègre-et-tire non linéaire est :

$$\tau \frac{dV}{dt} = F(V) + G(V)I$$

avec F et G deux fonctions pouvant apparaitre sous plusieurs formes. Le même mécanisme de génération de potentiel d'action que pour le modèle de Lapicque est utilisé.

Par rapport au modèle de Lapicque, la fonction G peut être vue comme une résistance dépendant du potentiel membranaire, et $-\frac{F(V)}{V-V_0}$ comme correspondant à une constante de dégradation, fonction elle aussi du potentiel membranaire.

Un exemple spécifique est l'intègre-et-tire quadratique, utilisé par exemple par Hansel et Mato [84] pour la recherche d'états stables dans de grands réseaux de neurones, ou bien par Brunel et Latham [22] pour calculer les taux de décharges de neurones dont l'entrée est soumise à un bruit coloré, bonne approximation du bruit dû aux potentiels d'action afférents.

3.3.3.4 *Spike Respond Model*

Tout comme les modèles d'intègre-et-tire non linéaires, les *Spike Respond Models* [66] (*SRM*) sont des généralisations de l'intégrateur à fuite de Lapicque. Néanmoins, deux aspects différencient ces deux généralisations : alors que les modèles intègre-et-tire définissent toujours le comportement du neurone par des équations différentielles et rendent dépendantes du potentiel membranaire certaines variables, les SRM les rendent dépendantes du moment $\hat{t}$ du dernier potentiel d'action, et expriment le potentiel de la membrane en fonction d'une intégrale temporelle et de noyaux.

L'équation générale d'un SRM est la suivante :

$$V = \eta(t - \hat{t}) + \int_{-\infty}^{\infty} \kappa(t - \hat{t}, s) I(t - s) ds$$

avec κ le noyau d'intégration modélisant le courant externe, et η le noyau rendant compte de la forme du potentiel d'action lui même. Le mécanisme de génération de potentiel d'action et de réinitialisation est toujours présent, avec un seuil qui peut dépendre du temps t.

Ce type de modèle a entre autres été étudié par Jolivet *et al.* [113], qui l'a utilisé sur des données artificielles avec une quantité minimale d'informations *a priori*.

3.4 Mise en réseau de neurones

3.4.1 Présentation formelle

Ce qu'on appelle *réseau de neurone artificiel*, abrégé en *réseau de neurone* ou *réseau neuronal*, est un modèle mathématique ou computationnel inspiré de réseaux de neurones biologiques, en conservant tout ou partie de

leurs propriétés. Il consiste en un ensemble de neurones modélisés et interconnectés en une structure de graphe, qui, à l'instar des réseaux biologiques, traitent un ensemble d'informations par une approche connexionniste pondérée. Plus formellement, les réseaux de neurones sont des outils de modélisation de données statistiques non linéaires, pouvant être utilisés pour modéliser des relations complexes entre entrées et sorties d'un modèle. Leur objet d'étude n'est plus l'activité du modèle d'un neurone formel, mais celle de la population considérée.

Par leur capacité d'induction, les réseaux de neurones permettent de construire un système de décision par confrontation à un ensemble de situations dont la généralité est fonction de la quantité et de la diversité des cas rencontrés, ils sont donc capable d'*apprendre*. Ces réseaux formels sont utilisés en apprentissage par approximation *parcimonieuse* (nécessitant moins de paramètres ajustables), pour de la classification, de la reconnaissance de motifs [12], pour de l'approximation de fonction, en finance,... C'est cette parcimonie, et la relative facilité de simulation qui en découle, qui donne aux réseaux de neurones leur intérêt industriel.

Il existe un nombre conséquent de formalisations de tels réseaux, différant par plusieurs paramètres comme la topologie des connexions, le modèle de neurone utilisé et donc les paramètres de ces modèles. On distingue parmi les plus célèbre le perceptron de Rosenblatt [194], le plus simple des réseaux de neurones formels, et sa généralisation le perceptron multicouche [95]. Ces deux exemples proposent un apprentissage supervisé mais ne permettent pas la rétropropagation du gradient. Une large gamme de modélisations à apprentissage supervisés sans et avec rétropropagation du gradient, ou à apprentissage non supervisé, reflète la diversité des possibilités offertes par cet outil de modélisation.

Une description plus complète des différents types de réseaux de neurones et de leurs utilisations dépasse cependant le cadre de cet exposé. Pour cela, nous référons le lecteur à [3].

3.4.2 Réseaux de neurones impulsionnels

Bien que capable d'apprendre et largement utilisé dans des problèmes de classification, les modèles de type perceptron sont trop éloignés de modèles *biologiquement plausibles* pour intéresser la communauté neurocomputationnelle. Pour les recherches concernant le fonctionnement du cerveau, les modèles intègre-et-tire et le fait qu'ils reproduisent quelques phénomènes biologiques (modélisation du potentiel membranaire, des canaux ioniques, des potentiels d'actions, de la fréquence de décharge, de la synchronisation des décharges) ont naturellement suscité un meilleur intérêt.

Dans le cas des neurones formels intègre-et-tire, Gerstner et Kistler recensent et explicitent différents types de réseaux [67], en variant le modèle impulsionnel utilisé (intègre-et-tire à fuite, SRM,...) ainsi que la modélisation des entrées synaptiques (constante, bruitée, dépendante des sorties des autre neurones,...). La population de neurones y est décrite par des équations de densité probabiliste, en introduisant une densité de potentiel membranaire $p(u,t)$, densité de neurones ayant leur potentiel membranaire égal à u au temps t. Ils décrivent également l'activité $A(t)$ d'un réseau comme le flux à travers le seuil de génération de potentiel

d'action. Ils montrent que les variations des grandeurs $p(u,t)$ et $A(t)$ peuvent se décrire sous la forme d'équations régissant la variation du potentiel membranaire dans le cas d'un neurone intègre-et-tire seul.

Conclusion du chapitre

Ce chapitre a premièrement mis en avant les différentes propriétés biologiques et physiologiques caractérisant l'état et l'évolution d'un neurone, et a décrit le mécanisme de transmission de l'information, le potentiel d'action, ainsi que ses méthodes de propagation à l'intérieur d'un neurone et entre deux neurones. Une seconde partie a introduit plusieurs modélisations biophysiques classiques d'un neurone, avec en particulier le modèle de Hodgkin et Huxley, toujours considéré comme une référence. Une troisième partie a décrit une classe de modélisations nommée *intègre-et-tire*, dans laquelle un mécanisme non linéaire modélise avec une précision acceptable la génération de potentiels d'action, et pour laquelle quelques modèles classiques sont brièvement présentés. Enfin, une brève quatrième partie a présenté le formalisme des réseaux neuronaux.

Dans le chapitre suivant, nous exposons deux de nos méthodes de simulation d'un réseau de neurones intègre-et-tire simple et leurs résultats, avec utilisation des processeurs de cartes graphiques comme outil de computation *générique*, avec pour objectif principal d'améliorer les temps de calcul.

4 Simulations sur GPU de réseaux de neurones impulsionnels

Sommaire

Les neurosciences computationnelles reposent en partie sur la simulation de grands ensembles de neurones interconnectés (voir par exemple [232]). Le temps d'exécution mis par ces simulations, croissant généralement de façon quadratique avec le nombre de neurones simulés, constitue un des freins à la simulation de réseaux à larges populations de neurones.

Dans ce chapitre, nous exposons deux nouvelles implémentations de réseaux de neurones impulsionnels, parallélisées grâce au GPU, pouvant émuler jusqu'à 2.5×10^5 neurones (nombre maximal en raison des limitations mémorielles des GPU), en proposant un gain en temps pouvant aller jusqu'à 18.

L'ensemble des travaux ici présentés ont été effectués dans le cadre du projet FACETS, avec l'équipe ODYSSEE. Les résultats présentés dans la section 4.2 ont fait l'objet d'une publication à NeuroComp'06 [34], en collaboration avec Renaud Keriven et Romain Brette. Les résultats de la section 4.3, non publiés, sont également issus d'une collaboration avec Romain Brette et Renaud Keriven.

4.1 Travaux antérieurs

La simulation de réseaux de neurones est un domaine que les chercheurs utilisant l'outil GPGPU ont assez peu exploré, généralement réservé à des machines parallèles de type grappe de calcul, ou MIMD à mémoire partagée distribuée, tels les travaux de Boniface *et al.* [16, 17]. La première tentative d'implémentation sur GPU d'un tel réseau a été réalisée en 2004 par Oh *et al.* [176]. Oh *et al.* y ont porté un réseau de neurones de type *perceptron multicouche* [95], à des fins de détection de texte. Dans ce genre de réseau, habituellement totalement connecté entre couches adjacentes, la couche de neurones en entrée reçoit les données qui transitent par la suite de couche en couche jusqu'à celle de sortie. Chaque couche de neurones exécute la même suite d'opérations : un produit scalaire entre leurs données d'entrée et les poids synaptiques, suivi d'une fonction non linéaire. La plupart de ces produits scalaires peuvent être remplacés par des multiplications matricielles, qu'ils ont adapté sur GPU grâce à la méthode de Moravánszky [46], atteignant un gain en temps de 20 par rapport à une implémentation CPU équivalente.

Par la suite, Bernhard et Keriven [11] ont été les premiers à implémenter des réseaux de neurones impulsionnels sur carte graphique, y transcrivant deux algorithmes à base de neurones utilisés pour segmenter des images, respectivement de Campbell *et al.* [28] et de Buhmann *et al.* [27]. Ils ont également présenté un algorithme plus généraliste pouvant être adapté à plusieurs types de réseaux neuronaux, en particulier des réseaux intègre-et-tire oscillants. Alors que les algorithmes précédents ne proposaient qu'une connexité locale entre les neurones (ce qui n'est pas représentatif d'un réseau de neurones biologiques), ce nouvel algorithme a la particularité de permettre une *connexité non locale*, en stockant astucieusement les voisins de chaque neurone dans une texture supplémentaire. Néanmoins, les limitations techniques des cartes graphiques disponibles, en particulier la taille maximale des textures et la quantité de mémoire embarquée, ont limité sur cette implémentation le nombre de neurones représentables et le nombre de voisins par neurone (ou plus précisément le produit de ces deux grandeurs).

Dans ce chapitre, nous exposons deux nouvelles façons de simuler sur carte graphique un réseau de neurones impulsionnels génériques à connexité non locale, pouvant émuler jusqu'à $2.5\ 10^5$ neurones pour 250 connexions aléatoires par neurone, en tenant compte des délais de transmission des influx nerveux le long des axones et à travers les synapses.

4.2 Réseaux de neurones impulsionnels à connexité non locale

Le premier algorithme présenté permet de simuler un ensemble de neurones impulsionnels reliés suivant une connexité non nécessairement locale. Le modèle intègre-et-tire utilisé est volontairement très simple. Ceci se justifie par l'objectif de ces travaux, à savoir améliorer la rapidité d'exécution de simulations de réseaux impulsionnels, et non interpréter les résultats obtenus (synchronisme,...) en vue d'en déterminer des caractéristiques.

L'implémentation de cet algorithme est hybride CPU/GPU : les équa-

tions régissant l'évolution des neurones sont résolues sur GPU, alors que les communications entre neurones sont effectuées sur CPU. Nous décrivons en détail cette mixité dans la suite de ce chapitre. Nous avons ici employé un GPU bas de gamme (GeForce 7800GTX) avec un CPU standard (Pentium 3GHz).

4.2.1 Modèle de neurone utilisé

Dans ce premier modèle, nous considérons un réseau de N neurones excitateurs, chacun étant connecté à $n_v = 250$ voisins tirés au hasard, ce sont les neurones efférents. Le nombre de neurones successeurs (ou postsynaptiques, ou afférent) est fixe, mais pas le nombre de neurones prédécesseurs (ou présynaptiques, ou efférent).

Nous avons choisi pour les neurones un modèle impulsionnel simple, un intègre-et-tire basé sur le modèle de Lapicque (voir section 3.3.2), avec des courants synaptiques excitateurs exponentiels. Ainsi l'état de chaque neurone i est donné par la valeur des variables V_i, le potentiel membranaire, et I_i, l'entrée synaptique totale, qui sont régies par les équations différentielles suivantes :

$$\begin{cases} \tau_V \frac{dV_i}{dt} &= -V_i + R(I_i + I_0) \\ \tau_I \frac{dI_i}{dt} &= -I_i \end{cases}$$

avec $\tau_V = 10$ ms la constante de temps membranaire, $\tau_I = 3$ ms la constante de temps synaptique, $I_0 = 1.1$ une entrée synaptique constante, et R la résistance membranaire, également constante.

Une impulsion est produite lorsque V_i atteint à l'instant t le seuil V_{seuil}. Par convention, on prend ce seuil égal à 1. V_i est alors réinitialisé à 0 et l'impulsion est transmise aux voisins après un délai constant d fixé : à l'instant $t + d$, $I_i \longleftarrow I_i + c_0$ pour chaque neurone cible i, avec $c_0 = 0.5/250$. On note que les neurones déchargent spontanément, il s'agit donc d'un réseau d'oscillateurs couplés.

L'objet de l'étude n'étant pas la qualité de l'approximation, les équations différentielles sont implémentées simplement selon un schéma d'Euler (pour ce modèle linéaire, une intégration exacte est possible, mais ce n'est pas le cas en général), en choisissant Δt comme pas de temps. Par intégration d'Euler, le système d'équations différentielles précédent devient, avec les dépendances temporelles :

$$\begin{cases} V_i(t + \Delta t) &= V_i(t) + \frac{\Delta t}{\tau_V}(-V_i + R(I_i(t) + I_0)) \\ I_i(t + \Delta t) &= I_i(t)(1 - \frac{\Delta t}{\tau_I}) \end{cases}$$

avec $V_i(t)$ le potentiel membranaire à l'instant t et $I_i(t)$ le courant synaptique d'entrée à l'instant t.

4.2.2 Implémentation

L'algorithme 3 implémente le schéma précédent avec les quelques notations supplémentaires suivantes : la quantité $m = d/\Delta t$ est supposée entière ; la variable T mémorise les temps d'impulsions ; X est une variable

intermédiaire ; les Y^k cumulent les effets des impulsions pour l'instant futur $t + k\Delta t$ et C est la matrice de connexion, dont les coefficients C_{ij} sont donnés par :

$$C_{ij} = \begin{cases} c_0 & \text{si } i \text{ est une cible de } j \\ 0 & \text{sinon} \end{cases}$$

De plus, la résistance membranaire R, égale à 1, n'est pas apparente dans le schéma de calcul.

Algorithme 3 : Schéma d'Euler avec délai

```
   Entrées : V et I : tableaux de données initiales (taille N, nombre de
             neurones) ; Y : tableau des impulsions (taille N × m)
1  pour k = 1 à m faire
2  |  Y^k ⟵ 0;
3  fin pour
4  boucler jusqu'à interruption
5  |  Transférer Y vers GPU;
6  |  T_i ⟵ 0 ;                                  // Début boucle GPU
7  |  pour k = 1 à m faire
   |  |  // Calcul des équations d'évolution
8  |  |  pour i = 0 à N faire en parallèle
9  |  |  |  V_i ⟵ V_i + (Δt/τ_v)(−V_i + I_i + I_0);
10 |  |  |  I_i ⟵ I_i + (Δt/τ_i)(−I_i + Y_i^k);
11 |  |  |  si V_i > 1 alors
12 |  |  |  |  V_i ⟵ 0, T_i ⟵ k;
13 |  |  |  fin si
14 |  |  fin pour
15 |  fin pour
   |                                              // Fin boucle GPU ;
16 |  Transférer T vers CPU;
17 |  pour k = 1 à m faire                        // Début boucle CPU
   |  |  // Transmission des potentiels d'action
18 |  |  pour i = 1 à N faire
19 |  |  |  X_i ⟵ {T_i = k};
20 |  |  fin pour
21 |  |  Y^k ⟵ C.X;
22 |  fin pour
   |                                              // Fin boucle CPU ;
23 fin boucle
```

Les lignes 6 à 15 sont effectuées sur le GPU, les lignes 17 à 22 sur le CPU.

Le caractère hybride de cette implémentation est nécessaire : en effet, les contraintes matérielles imposées par la programmation GPGPU (voir la section 1.4.3 pour de plus amples précisions sur ces restrictions) empêchent chaque neurone de prévenir ses successeurs (impossibilité d'effectuer des opération de *scatter*). La structure ici employée pour contourner ce problème, une matrice de connexité C de taille $N \times n_v$, est trop grande pour que toutes les données puisse être contenues dans la mémoire des GPU : en effet, pour $N = 2.5\ 10^5$ neurones et $n_v = 250$ voisins, il faut stocker 6.25×10^7 connections, chacune étant représentée par un entier codé sur 8o. Cela représente 476Mo de RAM sur les 512 présents sur

les meilleurs cartes alors disponibles (GeForce 7900), ce qui ne laisse pas assez de place pour les données restantes. Ce manque de mémoire contribue à rendre les opérations matricielles difficilement implémentables sur GPU, en particulier pour des matrices de tailles importantes.

Le découpage de l'algorithme 3 en deux parties, CPU et GPU, est alors déterminé par la nécessité de transférer le moins fréquemment possible des données entre le CPU et le GPU. D'où le principe consistant à calculer m pas de temps du schéma d'Euler, avant de gérer les impulsions. Ici les transferts sont limités à :

1. Envoyer Y^k vers le GPU à la ligne 5 ;
2. Récupérer T sur le CPU à la ligne 16.

Suivant le délai d et le pas de temps Δt nécessaire à l'intégration, il se peut que m devienne trop grand pour pouvoir mémoriser tous les Y^k. Nous avons donc aussi testé un schéma simplifié (algorithme 4) dans lequel l'instant exact k d'impulsion n'est pas pris en compte. Moins consommateur en mémoire, ce schéma rend compte, sur notre exemple, du comportement qualitatif du réseau. Les lignes 4 à 15 sont effectuées par le GPU, la ligne 17 par le CPU (nous utilisons ici une structure de type matrice creuse pour la matrice de connexité C).

Algorithme 4 : Délais de transmission approchés

```
   Entrées : V et I : tableaux de données initiales (taille N) ; Y :
             tableau des impulsions (taille N)
 1 Y ⟵ 0;
 2 boucler jusqu'à interruption
 3 |  Transférer Y vers GPU;
 4 |  I ⟵ I + Y ;
 5 |  X ⟵ 0 ;                                  // Début boucle GPU
 6 |  pour k = 1 à m faire
   |  |  // Calcul des équations d'évolution
 7 |  |  pour i = 0 à N faire en parallèle
 8 |  |  |  V_i ⟵ V_i + (Δt/τ_v)(−V_i + I_i + I_0);
 9 |  |  |  I_i ⟵ I_i + (Δt/τ_i)(−I_i);
10 |  |  |  si V_i > 1 alors
11 |  |  |  |  V_i ⟵ 0;
12 |  |  |  |  X_i ⟵ 1;
13 |  |  |  fin si
14 |  |  fin pour
15 |  fin pour
   |                                          // Fin boucle GPU ;
16 |  Transférer X vers CPU;
   |  // Transmission des potentiels d'action
17 |  Y ⟵ C.X ;
18 fin boucle
```

Les implémentations de ces algorithmes ont été effectuées sur des cartes GeForce 7900, disposant de 512 Mo de RAM. Au moment de ces implémentations, les cartes plus récentes (notamment les GeForce 8800) n'étaient pas encore disponibles.

4.2.3 Résultats

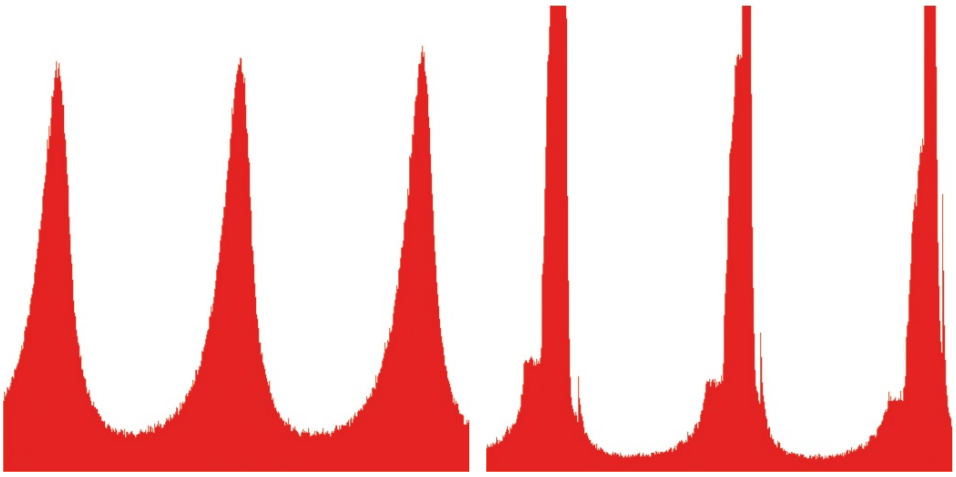


FIGURE 4.1 – *Profil des fréquences de décharge en régime stable du réseau simulé. Gauche : algorithme 3 (avec délais réels). Droite : algorithme 4 (avec délais approchés)*

La figure 4.1 montre la fréquence de décharge du réseau au cours du temps. Qualitativement, il apparait que les neurones déchargent de manière régulière et corrélée, comme observé dans [239] pour une architecture complète. Une mesure quantitative montre que les deux algorithmes retrouvent la même fréquence, bien que les profils soient naturellement un peu différents.

L'intérêt de notre étude porte sur le facteur de gain en temps obtenu par une implémentation GPU.

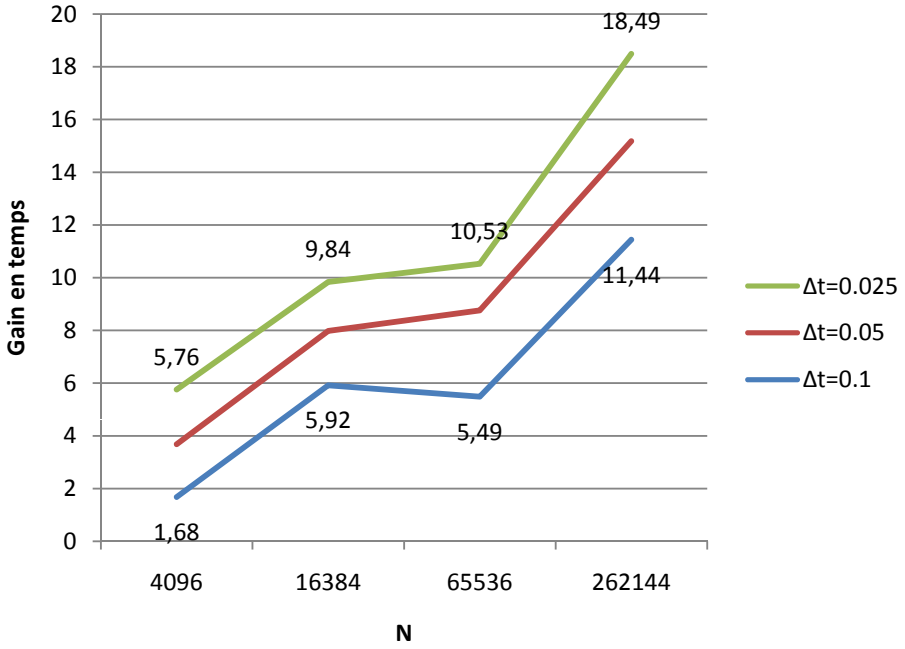


FIGURE 4.2 – *Gain en temps de calcul de l'implémentation GPU par rapport à une implémentation CPU de la partie schéma d'Euler de la simulation. Un gain d'environ* 20 *est observable entre une simulation GPU sur une carte bas de gamme et la simulation CPU de référence.*

La figure 4.2 montre, pour différentes valeurs de N et de Δt (donc du

nombre $m = d/\Delta t$ de boucles internes), le gain en temps pour la partie schéma d'Euler (hors calcul des Y, donc), obtenu par notre implémentation GPU par rapport à une implémentation CPU de référence, que ce soit pour l'algorithme 3 (lignes lignes 6 à 15) ou pour l'algorithme 4 (lignes 4 à 15). Le gain moyen est d'environ 20, ce qui est bien la valeur attendue en cas d'utilisation efficace du GPU.

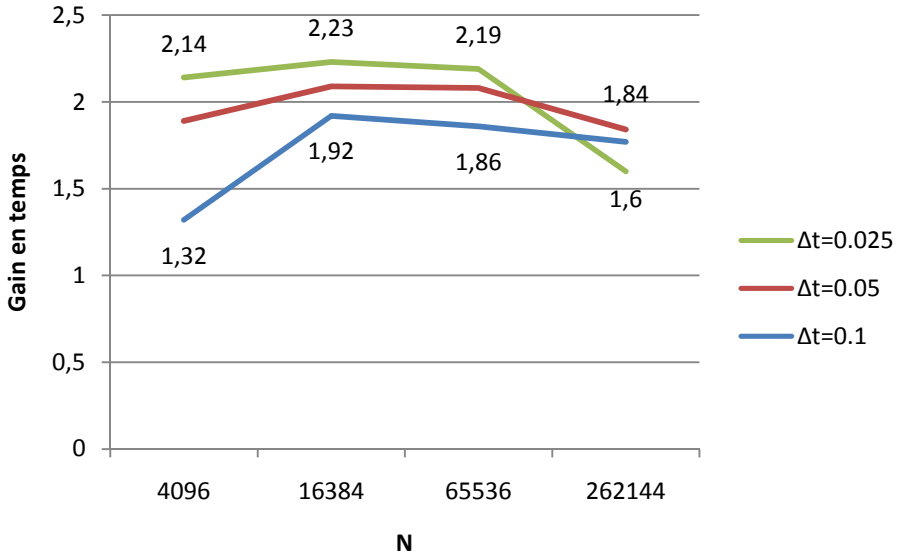


FIGURE 4.3 – *Gain en temps de calcul de notre implémentation GPU de l'algorithme 4 complet par rapport à une implémentation CPU de référence, en fonction du nombre de neurones simulés. Pour cet algorithme, le gain obtenu n'est que de* 2*. Un modèle plus complexe de neurone répartira cette charge davantage sur le GPU, augmentant ce gain en conséquence.*

La figure 4.3 montre, dans les mêmes conditions, le gain en temps obtenu par notre implémentation GPU de l'algorithme 4 au complet, par rapport à une implémentation CPU similaire. Le gain tombe à 2 environ. Ce gain est prometteur et requiert des commentaires :

1. Etant donné la complexité du calcul de Y et la simplicité du modèle de neurone choisi, ce gain est prévisible. En effet, la majeure partie du temps de calcul est passée sur CPU (ligne 17 de l'algorithme 4), la partie sur GPU étant bien plus rapide. Plus le modèle sera complexe, plus le gain sera important ;
2. Les transferts de données entre CPU et GPU occupent également une part non négligeable des temps de simulations.

C'est donc le caractère mixte CPU/GPU de l'implémentation qui bride les résultats de notre simulation. Avec la mise à disposition de GPU plus récents, plus puissants et disposant de plus de mémoire, en particulier les GeForce 8800, il est envisageable d'effectuer le calcul des paramètres Y entièrement sur GPU.

Néanmoins, la voie que nous avons choisi d'explorer par la suite est quelque peu différente : au lieu d'envisager les possibilités offertes par les générations futures de cartes graphiques, nous avons cherché à modifier notre algorithme pour pouvoir l'exécuter entièrement sur GPU. Nous décrivons ainsi dans la section suivante un nouvel algorithme, basé sur un principe de sondage de neurones.

4.3 Réseau de neurones impulsionnels par sondage

La seconde modélisation proposée est une amélioration de la précédente, présentée en section 4.2. On se propose ici de ne pas considérer les impulsions de tous les neurones prédécesseurs, mais d'en sonder quelques uns parmi ceux-ci pour une estimation de l'entrée synaptique globale.

La modélisation précédente supposait, implicitement, qu'un neurone communique avec ses neurones successeurs uniquement lorsqu'il génère un potentiel d'action qu'il souhaite leur transmettre. Le problème soulevé dans ce cas est l'impossibilité, en GPGPU, de modifier les valeurs des neurones successeurs directement (voir la section 1.4.3 pour de plus amples précisions sur les restrictions de la programmation GPGPU).

Le principe de notre second modèle est une approche inverse : chaque neurone va estimer le nombre global de neurones prédécesseurs ayant émis un potentiel d'action à l'instant précédent (discrétisé), en sondant un sous-ensemble *seulement* de ces neurones afférents, choisi aléatoirement, et ce *à chaque pas de temps*. Ce modèle a été établi en collaboration avec Romain Brette.

4.3.1 Modèle de neurone utilisé

Le modèle de neurone employé diffère légèrement de celui utilisé dans le précédent algorithme (voir section 4.2). Il est également basé sur un intègre-et-tire de Lapicque mais se différencie par deux points :

1. Un bruit suivant un processus brownien, modélisant le "bruit de fond" observé pour des neurones biologiques, remplace l'entrée synaptique constante I_0 ;
2. L'entrée synaptique I_i est la moyenne pondérée des courants provenant des neurones prédécesseurs sondés, tirés au hasard.

Les équations régissant les évolutions de ces variables sont les suivantes :

$$\begin{cases} \tau_V \frac{dV_i}{dt} &= -V_i + RI_i + bruit \\ \\ I_i &= \frac{N_{neighb}}{N_{poll}} w \sum_{k=1}^{m} J_{n_k} \\ \\ \tau_J \frac{dJ_i}{dt} &= -J_i \text{ avec } J_i \longleftarrow J_i + \epsilon_i \text{ à chaque impulsion} \end{cases}$$

avec $\tau_V = 10$ ms la constante de temps membranaire, $\tau_J = 3$ ms la constante de temps synaptique (ces deux constantes sont identiques à celles utilisées dans le modèle décrit dans la section 4.2), R la résistance membranaire, N_{neighb} le nombre total de prédécesseurs, N_{poll} le nombre de prédécesseurs sondés, w_i le poids synaptique, $\{n_k\}_{k\geq 1}$ une famille de prédécesseurs sondés aléatoirement, J_i le courant généré par les propres impulsions du neurone, et $\epsilon_i = \pm 1$. Le signe de ϵ_i dépend du caractère excitateur ou inhibiteur du neurone prédécesseur.

Par schéma d'Euler, en choisissant Δt pour pas de temps, le jeu d'équations différentielles précédent devient, avec les dépendances temporelles :

$$\begin{cases} V_i(t+\Delta t) &= V_i(t) + \frac{\Delta t}{\tau_V}(-V_i + RI_i(t) + bruit) \\ I_i(t+\Delta t) &= \frac{N_{neighb}}{N_{poll}} w_i \sum_{k=1}^{m} J_{n_k}(t) \\ J_i(t+\Delta t) &= J_i(t)(1 - \frac{\Delta t}{\tau_J}) \text{ avec } J_i \longleftarrow J_i + \epsilon_i \text{ à chaque impulsion} \end{cases}$$

avec $V_i(t)$ le potentiel membranaire à l'instant t et $I_i(t)$ le courant synaptique d'entrée à l'instant t. $I_i(t+\Delta t)$ est vu comme la somme pondérée des courants J_{n_k} des neurones prédécesseurs sondés, à l'instant t et non à l'instant $t+\Delta t$, ce qui correspond au délai de propagation de l'influx nerveux.

Deux problèmes implicites émergent de cette modélisation. Tout d'abord, la création d'un bruit brownien nécessite la manipulation de variables aléatoires, distribuées par un générateur de nombres aléatoires. Un tel générateur manipule des nombres entiers, type de données tout juste disponible au début de l'implémentation de cet algorithme, avec les cartes GeForce 8800.

Ensuite, comme nous l'avons énoncé, il est difficile de stocker le réseau entier dans la mémoire de la carte. Il est alors nécessaire de pouvoir recréer ce réseau, c'est-à-dire pour chaque neurone, être capable de régénérer l'ensemble de ses neurones prédécesseurs ou successeurs.

Ces deux problèmes sont abordés dans les deux paragraphes suivants, avant une description de l'implémentation et des résultats.

4.3.2 Génération de nombres aléatoires et bruit brownien

La génération de nombres aléatoires peut être extrêmement fastidieuse si l'on recherche de bonnes propriétés de distributions des variables. Les implémentations sur GPU sont peu nombreuses. On dénombre au moins deux essais : Howes et Thomas [106] utilisant le langage CUDA, et Sussman *et al.* [224].

Dans notre cadre, nous avons privilégié l'utilisation d'un type simple de générateur, facilement implémentable sur GPU et dont les propriétés sont satisfaisantes pour notre application : un générateur congruentiel linéaire. Couplé à une transformation de Box-Müller, une distribution gaussienne peut être produite, à partir de laquelle on détermine aisément une variable suivant une loi brownienne.

4.3.2.1 Générateur congruentiel linéaire

Les générateurs congruentiels [121] délivrent une suite d'entiers positifs $\{x_n\}_{n>0}$ du type :

$$x_{n+1} = ax_n + c \mod m$$

avec a, c et m les paramètres du générateur. a est appelé le multiplicateur, c l'incrément, m le module. Le premier élément x_0 est appelé la graine du générateur.

Nous avons opté pour les valeurs suivantes pour ces paramètres, utilisées par *DEC* pour leurs architectures VAX :

$$\begin{cases} a &= 69069 \\ c &= 1 \\ m &= 2^{32} \end{cases}$$

Les entiers étant codés sur 32bits sur GPU, l'opération modulo 2^{32} sera économique car automatiquement réalisée.

Ce générateur produit des entiers répartis sur $[0, 2^{32}[$. Etant donné la simplicité du générateur, le caractère uniforme de cette répartition n'est pas réellement assuré, ce qui n'est cependant pas essentiel pour notre étude.

De plus, un des défauts reconnus des générateurs congruentiels linéaires est de produire des entiers dont les bits de poids faibles, c'est-à-dire les chiffres les plus à droite dans l'écriture binaire, sont corrélés. Knuth démontre cela dans [121]. Implicitement, nous ne considérons donc que les bits de poids fort dans nos simulations.

4.3.2.2 Transformée de Box-Müller et distribution gaussienne

La transformée de Box-Müller [19] a la particularité de générer des paires de nombres aléatoires à distribution normale centrée réduite (ou distribution gaussienne), à partir de nombres aléatoires de loi uniforme. En notant (x, y) une telle paire de nombres (déterminée à partir de deux entiers dans $[0, 2^{32}[$ obtenus au paragraphe précédent, en les ramenant par division dans $[0, 1[$), la transformée de Box-Müller s'écrit de la façon suivante :

$$\begin{cases} X &= \sqrt{-2 \ln x} \cos 2\pi y \\ \\ Y &= \sqrt{-2 \ln x} \sin 2\pi y \end{cases}$$

avec (X, Y) la paire de variables indépendantes à distribution normale centrée réduite voulue.

A partir de ce résultat, une variable G suivant une distribution normale d'écart type σ est obtenue par :

$$\begin{aligned} G &= \sigma\, X \\ &= \sigma \sqrt{-2 \ln x}\, \cos 2\pi y \end{aligned}$$

La variable aléatoire Y de la transformée de Box-Müller n'est pas utilisée dans nos simulations.

4.3.2.3 Bruit brownien

L'obtention d'un bruit brownien B à partir d'une variable gaussienne G est classiquement réalisée par :

$$B = \sqrt{dt}\, G$$

avec G la gaussienne obtenue dans le paragraphe précédent, et dt l'intervalle entre deux dates consécutives où l'on effectue les simulations, c'est-à-dire le pas de temps.

Finalement, on obtient une variable B suivant un mouvement brownien par :

$$B = \sqrt{dt}\ \sigma\ \sqrt{-2\ln x}\ cos(2\pi y)$$

avec x et y deux variables aléatoires uniformes sur $]0,1]$.

Pour notre application, ce bruit brownien modélise le bruit de fond caractéristique des enregistrements de potentiels de neurones biologiques.

4.3.3 Régénération du réseau

Le second problème à résoudre est la régénération du réseau. Nous cherchons à trouver une représentation des prédécesseurs, ou des successeurs, de chaque neurone i.

Dans notre cadre GPGPU, nous avons vu qu'il est difficile pour un neurone de prévenir ses successeurs de son état et qu'il lui est préférable de consulter l'état de tous ses prédécesseurs ; nous cherchons donc à déterminer, pour un neurone i, une représentation de l'ensemble de ses prédécesseurs, soit une fonction $f(i)$ telle que :

$$\forall i \in [\![1, N]\!], f(i) = \{\text{Prédécesseurs de } i\}$$

N étant le nombre de neurones total dans le réseau.

De plus, comme nous l'avons vu cité au préalable, il est impossible de stocker une représentation complète du réseau, il est donc nécessaire de trouver une formulation réduite de cet ensemble. La solution que nous avons adoptée est la suivante : à chaque neurone i est associé une graine fixe $seed_i$ et une graine évoluant à chaque itération $seed_i^{(tmp)}$ initialisée à la même valeur que $seed_i$. En notant N_{neighb} le nombre fixe de voisins (prédécesseurs) pour chaque neurone, on détermine l'ensemble des prédécesseurs de i comme la suite d'entiers $\{p_k^{(i)}\}_{k\in[\![1,N_{neighb}]\!]}$ produite par le même type de générateur congruentiel linéaire que celui utilisé dans la section 4.3.2.1 :

$$\begin{cases} p_{k+1}^{(i)} &= a p_k^{(i)} + c \mod N, \forall k \geq 0 \\ p_0^{(i)} &= seed_i^{(tmp)} \end{cases}$$

avec les mêmes valeurs pour les paramètres a et c : $a = 69069$ et $c = 1$. Ainsi, tous les prédécesseurs du neurone i sont accessibles par répétition d'un calcul élémentaire.

Néanmoins, l'ensemble des prédécesseurs ne sera pas complètement régénéré à chaque itération.

En effet, le principe du sondage est justement d'interroger une partie seulement de cet ensemble pour évaluer l'entrée synaptique globale du neurone. Notons N_{poll} le nombre (fixé) de prédécesseurs sondés par neurone, et $turn$ une variable globale initialisée à 0. Alors, l'algorithme 5 de détermination du sous-ensemble des prédécesseurs sondés du neurone i, représentant une étape d'une itération dans le processus global (voir l'im-

plémentation de l'algorithme complet 6 dans la section 4.3.4) s'écrit de la façon suivante :

Algorithme 5 : Prédécesseurs à sonder du neurone i

ENTRÉES : $seed_i$: graine fixe

SORTIES : $\{p_k^{(i)}\}_{k\in[\![1,N_{poll}]\!]}$: sous-ensemble des prédécesseurs de i à sonder

1 **pour** $k = 1$ **à** N_{poll} **faire**
2 $\quad p_k \longleftarrow seed_i^{(tmp)} \mod N$;
3 $\quad seed_i^{(tmp)} \longleftarrow (a\, seed_i^{(tmp)} + c) \mod m$;
4 **fin pour**
5 **si** $turn = 0$ **alors**
6 $\quad seed_i^{(tmp)} \longleftarrow seed_i$;
7 **fin si**

On prend $m = 2^{32}$. Les variables $seed_i^{(tmp)}$ sont initialisées une unique fois dans le programme principal, elles ne sont pas réinitialisées au début de chaque itération de l'algorithme 5. De même, la variable *turn* n'est pas modifiée dans cet algorithme, mais le sera dans l'algorithme principal 6, entre chaque itération de la boucle principale.

De cette façon, on constate qu'il existe, pour chaque neurone, un nombre restreint de sous-ensembles générés (autant que d'état que peut prendre la variable *turn*) et qu'ils sont réutilisés de façon cyclique. Dans le programme principal, ce nombre sera de N_{neighb}/N_{poll}, permettant de varier au maximum les pools de neurones sondés. Dans le cadre de nos simulations, il est suffisant pour assurer une assez grande diversité dans les échantillons sondés.

4.3.4 Implémentation

La programmation GPGPU utilisant un *shading language* tel Cg oblige souvent à partitionner l'algorithme à implémenter. C'est le cas ici ; notre algorithme est basé sur plusieurs noyaux computationnels (plusieurs boucles parallèles). Cet algorithme, présenté en 6, implémente le modèle de réseau neuronal exposé jusqu'ici dans la section 4.3.

Initialisations Les lignes 3 à 8 correspondent à l'initialisation des variables : V_i le potentiel membranaire ; J_i le courant membranaire ; l_i un indice pour reporter dans le tableau T_i les instants de décharge ; *turn* un entier utile à la régénération du réseau (pour la réinitialisation des graines des générateurs aléatoires).

La boucle principale (lignes 10 à 41) est répartie en deux noyaux successifs, chacun s'exécutant sur tous les neurones en parallèles :

Mise à jour de V_i Lignes 13 à 30 : Le potentiel membranaire V_i est mise à jour par :

1. Inférence de l'entrée synaptique globale I comme somme pondérée des courants synaptiques de quelques prédécesseurs choisis aléatoirement (sondage), lignes 14 à 23 : les prédécesseurs choisis sont les N_{poll} prochaines itérations du générateur congruentiel de graine

Algorithme 6 : Simulation par sondage

ENTRÉES : $\{seed_i\}_{i\in[\![1,N]\!]}$: graines fixes pour la régénération du réseau ; $\{seed_i^{(br)}\}_{i\in[\![1,N]\!]}$: graines pour le bruit brownien
SORTIES : $\{T_i\}_{i\in[\![1,N]\!]}$: tableaux contenant les instants de décharge de chaque neurone

Initialisations;
$turn \longleftarrow 0$;
pour $i = 0$ **à** N **faire**
  $V_i \longleftarrow 0$;
  $J_i \longleftarrow 0$;
  $l_i \longleftarrow 0$;
  $seed_i^{(tmp)} \longleftarrow seed_i$;
fin pour
Transférer V,J,I,$turn$,$seed$,$seed^{(tmp)}$ **vers** GPU;
boucler jusqu'à interruption
  $turn \longleftarrow (turn + 1) \mod N_{neighb}/N_{poll}$;
  $t = t + \Delta t$;
  `// Calcul des équations d'évolution`
  pour $i = 0$ **à** N **faire en parallèle**
    $J \longleftarrow 0$;
    pour $l = 0$ **à** N_{poll} **faire**
      $p \longleftarrow seed_i^{(tmp)} \mod N$;
      $seed_i^{(tmp)} \longleftarrow (a\, seed_i^{(tmp)} + c) \mod m$;
      $J \longleftarrow J + J_p$;
    fin pour
    si $turn = 0$ **alors**
      $seed_i^{(tmp)} \longleftarrow seed_i$;
    fin si
    $I \longleftarrow \frac{N_{neighb}}{N_{poll}} w_i\, J$;
    $seed_i^{(br)} \longleftarrow a\, seed_i^{(br)} + c \mod m$;
    $x \longleftarrow \frac{1}{m}(seed_i^{(br)} + 1)$;
    $seed_i^{(br)} \longleftarrow a\, seed_i^{(br)} + c \mod m$;
    $y \longleftarrow \frac{1}{m}(seed_i^{(br)} + 1)$;
    $g \longleftarrow \sigma\sqrt{-2\log(x)}\cos(2\pi y)$;
    $V_i = V_i + \frac{\Delta t}{\tau_V}(-V_i + R\, I + \sqrt{\Delta t}\, g)$;
  fin pour
  `// Gestion des potentiels d'action`
  pour $i = 0$ **à** N **faire en parallèle**
    si $V_i > V_{seuil}$ **alors**
      $V_i \longleftarrow 0$;
      $J_i \longleftarrow J_i + 1$;
      $T_i[l_i] \longleftarrow t$;
      $l_i \longleftarrow l_i + 1$;
    fin si
    $J_i \longleftarrow J_i(1 - \frac{\Delta t}{\tau_J})$;
  fin pour
  Transférer V **vers** CPU;
fin boucle

$seed_i^{(tmp)}$, réinitialisée à $seed_i$ tous les N_{neighb}/N_{poll} tours de boucle principale, ces tours étant comptés par la variable $turn$;

2. Génération d'un bruit gaussien g à partir de la méthode exposée dans la section 4.3.2.3, lignes 24 à 28 : deux nombre aléatoires x et $y \in]0,1]$ sont déterminés et utilisés dans une transformée de Box-Müller (voir section 4.3.2.2) ;
3. Mise à jour du potentiel membranaire V_i en fonction de I et du bruit brownien $\sqrt{\Delta t}\, g$, en respectant l'équation régissant ce potentiel.

Potentiels d'actions Lignes 31 à 39 : Une fois que tous les potentiels membranaires ont été mis à jour, on vérifie si chaque neurone a vu son potentiel dépasser le seuil V_{seuil}. Dans ce cas, on émet un potentiel d'action : réinitialisation du potentiel à 0, augmentation brusque du courant membranaire (modélisation des effets du pic du potentiel dû au potentiel d'action), et sauvegarde de l'instant de décharge. Puis le courant membranaire est mis à jour en respectant l'équation différentielle le régissant.

Néanmoins, en pratique, cet algorithme ne peut être implémenté tel quel, avec seulement deux shaders Cg. Chaque noyau computationnel est dirigé par un shader, ne pouvant écrire en sortie qu'un seul type de données. Ainsi, dans la première boucle (lignes 13 à 30), il est impossible de modifier à la fois les valeurs des potentiels V_i, réelles, et des graines $seed_i^{(tmp)}$ et $seed_i^{(br)}$, entières. La solution adoptée est alors de reporter la mise à jour de ces graines dans un troisième noyau computationnel, exécuté après les deux premiers, en répétant autant de fois que nécessaire l'itération du générateur congruentiel. Dans le premier noyau, ce sont donc des copies des graines qui sont utilisées.

4.3.5 Résultats

Il est nécessaire de valider la méthode de simulation par sondage pour pouvoir l'utiliser dans des cas pratiques. Nous commençons donc par une justification expérimentale de la méthode, pour ensuite revenir sur les résultats obtenus pour les gains de temps.

4.3.5.1 Justification expérimentale

Notre modèle de simulation de réseau de neurones par sondage est valide si la fréquence de décharge obtenue, en régime stationnaire, correspond à celle donnée par le même réseau (avec les mêmes paramètres) sans sondage.

Les paramètres de notre simulation sont les suivants :

- Le nombre de neurones du réseau : $N = 1024 \times 10$ que nous supposons constant jusqu'à mention contraire ;
- Le nombre de neurones prédécesseurs de chaque neurone : $N_{neighb} = 500$;
- Le nombre de neurones sondés pour chaque neurone : N_{poll}, variable (afin tester de l'influence du pourcentage de neurones sondés)
- Le seuil de déclanchement d'un potentiel d'action : $V_{seuil} = 0.015mV$;

- Le pas de temps de la discrétisation d'Euler des schémas numériques : $\Delta t = 0.1ms$;
- Le nombre d'itérations de la boucle globale : $Nb_{iter} = 30000$;
- La durée simulée : $T = \Delta t \ Nb_{iter} = 3s$ (pour être en régime stable, approximativement atteint à $0.5s$) ;
- Le poids des connexions synaptiques exitatrices : $w_{exc} = 3 \times 10^{-12}$;
- Le poids des connexions synaptiques inhibitrices : $w_{inh} = -1.2 \times 10^{-11}$;
- La constante de temps membranaire : $\tau_V = 10ms$;
- La constante de temps synaptique : $\tau_I = 5ms$;
- La résistance membranaire : $R = 1\Omega$, non représentée.

La figure 4.5 montre les résultats de notre analyse sur la précision de la fréquence de décharge en régime stationnaire lors d'une simulation sur GPU, par sondage, comparée avec une simulation sans sondage sur CPU, notre implémentation directe de référence que nous nommerons dans la suite simplement version CPU. Les différents graphes constituant cette figure présentent les résultats d'une même simulation avec une variation sur le pourcentage de neurones excitateurs. La variation de cette proportion modifie la fréquence de décharge en régime stationnaire de la version CPU. La figure 4.4 montre l'évolution de la fréquence de décharge, variant dans le même sens que le pourcentage de neurones excitateurs : un plus grand nombre de neurones excitateurs va entrainer une augmentation des émissions de potentiels d'action sur un labs de temps donné, d'où une augmentation de la fréquence de décharge.

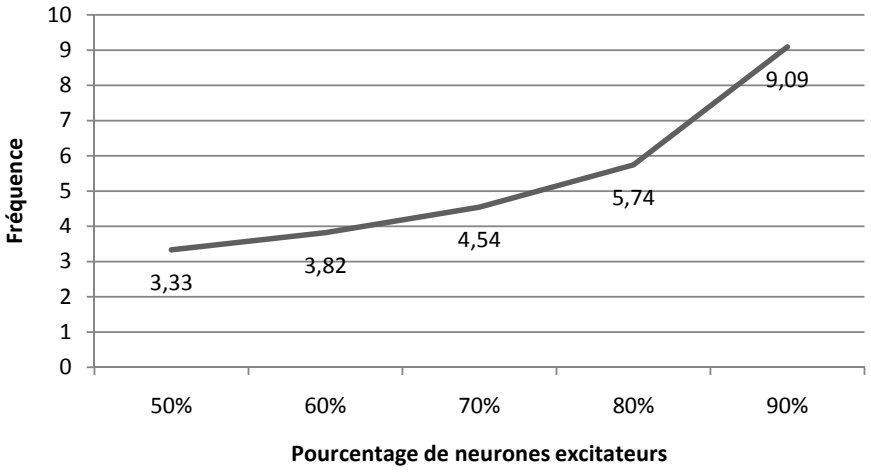


FIGURE 4.4 – *Fréquence de décharge (en Hz) en régime stationnaire en fonction du pourcentage de neurones excitateurs présents dans le réseau. On observe que la fréquence varie dans le même sens que le pourcentage de neurones excitateurs.*

Comme attendu, quel que soit le pourcentage de neurones excitateurs existants dans le réseau, en augmentant le nombre de neurones présynaptiques sondés, la fréquence de décharge en régime stationnaire obtenue par simulation par sondage converge, aux erreurs d'arrondis près, vers celle obtenue par simulation sans sondage. Pour une proportion de 80% de neurones excitateurs, l'erreur relative commise par sondage, par rapport à la version de référence, est de l'ordre de 0.7%, et ce dès que seulement 5 prédécesseurs sont sondés à chaque pas de temps. Lorsque

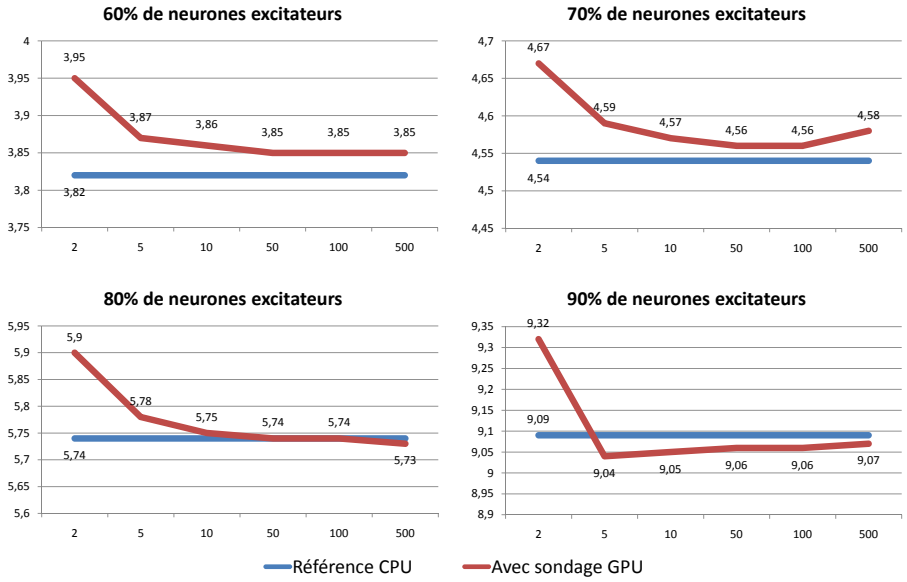


FIGURE 4.5 – *Comparaison des fréquences de décharges (en Hz) en régime stationnaire pour des simulations sans et avec sondage en fonction de N_{poll}. En faisant varier la proportion de neurones excitateurs dans le réseau, on modifie la fréquence de décharge pour l'algorithme de référence, c'est-à-dire sans sondage (voir l'évolution de la fréquence de décharge en fonction du pourcentage de neurones excitateurs en figure 4.4). Quelle que soit cette proportion, plus le nombre de prédécesseurs se rapproche de $N_{neighb} = 500$, plus la fréquence de décharge obtenue dans les versions avec sondage se rapproche de celle de référence, ce qui est le comportement attendu. A partir de 5 prédécesseurs sondés (soit ici 1% de l'ensemble des prédécesseurs), l'erreur relative est inférieure à 1.3% (voir le tableau de la figure 4.1).*

l'on augmente le nombre de neurones sondés, cette erreur continue de baisser, comme le montre la table 4.1.

N_{poll}	Pourcentage de neurones excitateurs			
	60%	70%	80%	90%
2	3.29	2.78	2.71	2.46
5	1.29	1.08	0.69	0.55
10	1.03	0.65	0.17	0.44
50	0.77	0.43	0	0.33
100	0.77	0.43	0	0.33
500	0.77	0.87	0.17	0.22

TABLE 4.1 – *Erreurs relatives en pourcentages sur les fréquences observées dans les simulations de référence et avec sondage, en fonction du nombre N_{poll} de neurones sondés parmi $N_{neighb} = 500$ et du pourcentage de neurones excitateurs.*

Ces résultats montrent que le profil de décharge peut être reconstituée à partir d'un petit nombre de neurones présynaptiques avec une très bonne précision : dès que 5 neurones sont sondés parmi les prédécesseurs, l'erreur relative commise sur la fréquence de décharge n'excède jamais 1.3%, quel que soit la proportion de neurones excitateurs dans le réseau (des proportions inférieurs à 60% ne sont pas biologiquement plausibles). On remarque qu'en sondant la totalité des prédécesseurs (soit 500), les erreurs relatives obtenues ne sont pas précisément nulles. Ceci est probablement dû aux différences de précision entre types de données CPU et

GPU (les nombres réels des GPU NVidia ne respectent en effet pas encore la norme IEEE)

4.3.5.2 Performances GPU

Nous étudions ici l'apport de notre implémentation GPU de la simulation d'un réseau de neurones (avec sondage) à l'aide de notre librairie CLGPU (voir section 1.5), en comparaison à la version CPU (sans sondage). La machine utilisée pour ces tests est dotée d'un processeur Intel Xeon, de 1Go de RAM et d'une carte graphique GeForce8800 GTX (maintenant dépassée) embarquant 769Mo de RAM .

De par la nécessité d'aller sonder *à chaque instant* les prédécesseurs, au lieu de prévenir ses successeurs uniquement lors de l'émission d'un potentiel d'action dans l'algorithme CPU, la simulation de l'algorithme à sondage sur CPU (dont l'implémentation a été réalisée à des fins de validation de l'algorithme GPU) est extrêmement lente comme le présente la figure 4.6, ce qui rend inadaptée son utilisation.

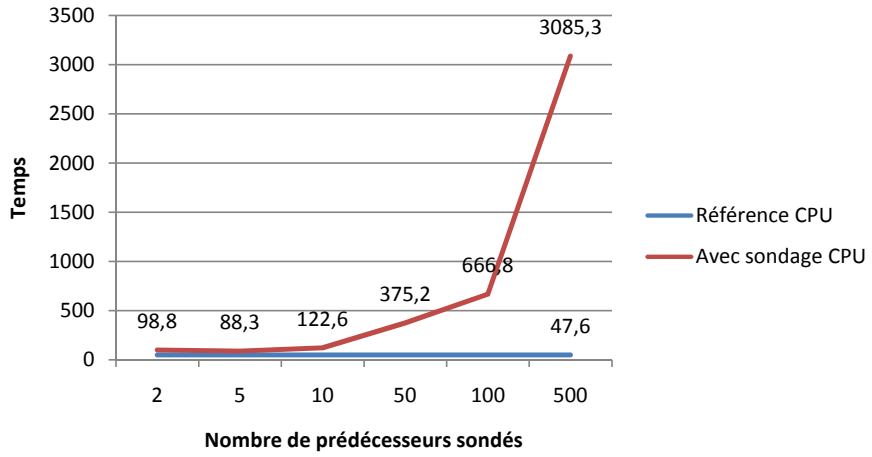


FIGURE 4.6 – *Comparaison des temps d'exécution (en s) pour les algorithmes CPU et CPU avec sondage, en fonction du nombre N_{poll} de neurones sondés. Pour ce graphe, la proportion de neurones excitateurs est de 80%. L'algorithme à sondage est particulièrement lent, les temps de simulation augmentant de façon exponentielle avec N_{poll}, car tous les neurones doivent, à chaque pas de temps, sonder leur prédécesseurs, alors que dans l'algorithme CPU, ils ne préviennent leurs successeurs que lorsque nécessaire, soit lors du déclenchement d'un potentiel d'action.*

La figure 4.7 révèle cependant que le temps obtenu avec l'algorithme CPU peut largement être amélioré grâce à notre implémentation GPU. Cette figure montre, pour quatre proportions différentes de neurones excitateurs, les temps mis pour le déroulement des algorithmes CPU et GPU. On remarque tout d'abord que le temps mis par la version CPU augmente avec la proportion de neurones excitateurs. En effet, plus il y a de neurones excitateurs, plus les neurones vont générer des potentiels d'actions, ce qui alourdit la charge de cet algorithme, dans lequel un neurone doit prévenir ses successeurs lors de l'émission d'un potentiel d'action.

Pour des nombres raisonnables de prédécesseurs sondés, la version GPU présente un gain pouvant aller jusqu'à 4.3 par rapport à la version CPU (voir figure 4.9). A partir de 100 prédécesseurs sondés, l'algorithme GPU montre cependant une perte de vitesse importante, ce qui est dû

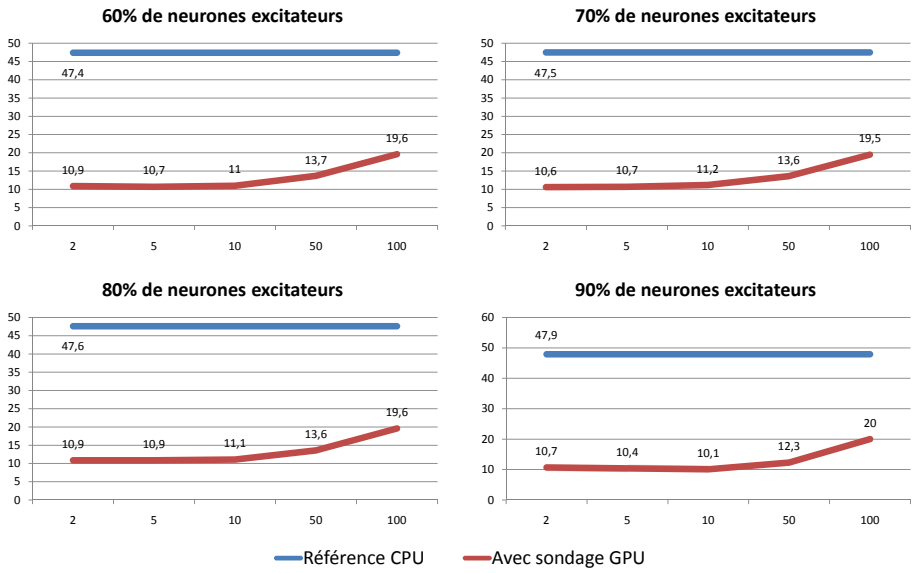


FIGURE 4.7 – *Comparaison des temps d'exécution (en s) pour les algorithmes CPU et GPU en fonction du nombre N_{poll} de neurones sondés. Les quatre profils sont similaires. L'algorithme avec sondage sur GPU propose un temps d'exécution jusqu'à 5 fois plus faible.*

aux boucles de calcul de type boucle `for()` nécessaires pour scruter les prédécesseurs, et pour lesquelles le GPU n'est pas optimisé.

Ce gain de 4.3 est obtenu avec la simulation d'un réseau impulsionnel comportant une population de seulement $N = 10000$ neurones. Ce nombre est d'une part petit face aux réalités biologiques, et d'autre part trop faible pour que le caractère *parallèle* de l'implémentation soit utilisé à son plus haut potentiel. Pour un nombre supérieur de neurones, le rapport des temps d'exécution est encore plus favorable à notre méthode : nous retranscrivons dans la table de la figure 4.8 différents temps de simulation pour d'autres valeurs de N et pour plusieurs valeurs de N_{poll} pour la version GPU.

N	CPU	GPU		
		$N_{poll} = 5$	$N_{poll} = 10$	$N_{poll} = 50$
2000	9.8	10.8	10.8	11.3
5000	24.7	10.9	10.8	12.5
10000	47.4	10.9	11.1	13.6
50000	253.9	13.9	17.9	72.3
100000	521.6	28.7	49.6	213.7

FIGURE 4.8 – *Temps d'exécution (en s) des algorithmes CPU et GPU, en fonction de N et pour différentes valeurs de N_{poll}. La proportion de neurones excitateurs a été ici fixée à 80%.*

La figure 4.9 présente les gains de temps réalisé par notre implémentation GPU par rapport à la version CPU, pour les valeurs de N et N_{poll} utilisées dans la table de la figure 4.8. Nous montrons qu'il est ainsi possible de simuler un réseau de neurone impulsionnel comportant un grand nombre de neurones (au moins 50000) jusqu'à 14 fois plus rapidement qu'avec une implémentation CPU, tout en ayant une erreur qualitative relative maximale de l'ordre de 1%. Ce gain peut augmenter jusqu'à 18 si l'on tolère une erreur relative de l'ordre de 1.3%.

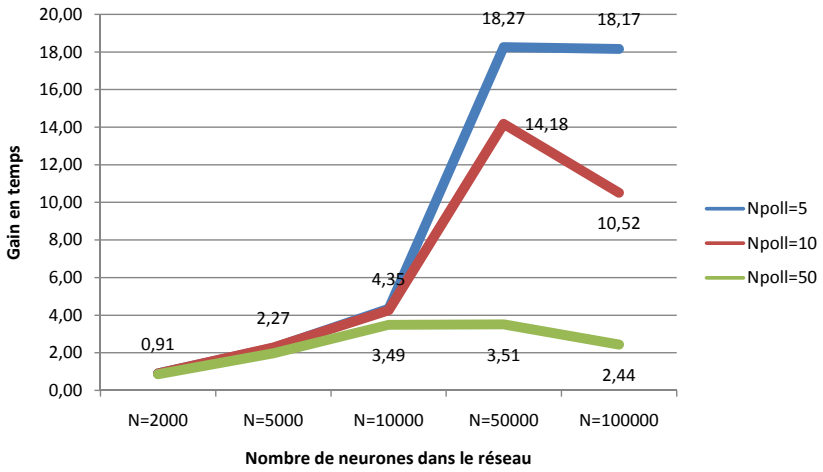


FIGURE 4.9 – *Gains en temps de l'algorithme GPU par rapport à l'algorithme CPU pour différentes valeurs de N. Plus le nombre total de neurones à simuler est grand, plus l'implémentation GPU est efficace, pouvant être jusqu'à* 18 *fois plus rapide que la version CPU, pour une simulation d'au moins* 50000 *neurones, en sondant* 5 *prédécesseurs.*

De plus, tout comme pour les résultats présentés en section 4.2.3, il est important de noter que le modèle de neurone utilisé est d'une simplicité ne favorisant pas les calculs massifs dont sont capables les GPU. L'utilisation de modèles neuronaux plus complexes, exploitant mieux les capacités computationnelles des GPU, ainsi que de cartes graphiques de générations plus récentes, permettrait d'augmenter encore ce gain.

Conclusion du chapitre

Ce chapitre a présenté nos différentes applications en neurosciences avec utilisation des GPU comme outil d'accélération des calculs par parallélisation. Un premier type de réseau de neurone impulsionnel à connexité éparse a été présenté formellement, puis sous forme algorithmique. Le gain en temps obtenu, prometteur mais faible, nous a poussés à développer une nouvelle méthode de simulation de réseau neuronal impulsionnel.

Ainsi, un deuxième type de réseau de neurone a été proposé, dans lequel chaque neurone va constamment vérifier l'état d'une partie aléatoire de ses prédécesseurs (simulation par sondage). Une implémentation algorithmique a ensuite été présentée, permettant un gain en temps pouvant aller jusqu'à 18, pour une erreur commise inférieure à 1.3%.

Il est à noter que ces gains temporels peuvent largement augmenter avec la complexité du modèle de neurone retenu. En effet, plus le modèle est complexe, plus la charge de travail sera lourde, et plus le rapport des temps d'exécution des simulations CPU et GPU grandira, en faveur du GPU. De plus, notre objectif n'a pas été ici de valider nos méthodes sur des dérivés applicatifs, mais de proposer un outil de simulation rapide et précis à la communauté scientifique. Il serait à présent intéressant de connaître les implications des erreurs introduites par nos modèles.

Les deux chapitres de la partie suivante s'intéressent à un tout autre domaine d'application, la Vision par Ordinateur.

Troisième partie

Vision par Ordinateur

Stéréovision variationnelle sur GPU

5

Sommaire

La vision par ordinateur est définie comme la science et la technologie des *machines qui voient*. Elle est un domaine computationnel cherchant à extraire de l'information dans des images, qu'elles soient séquences vidéo, photographies ou autres. Les systèmes décrits en vision ont des applications nombreuses et peuvent par exemple servir dans des procédés industriels de contrôle, pour de la vidéosurveillance, de la modélisation d'objets, de scènes ou d'environnements, du traitement d'images,...

Ce chapitre va introduire brièvement ce domaine et les problèmes qu'il tente de résoudre, avant de présenter plus en détail une méthode de reconstruction d'objets en 3D, la stéréovision, ainsi qu'un algorithme de stéréovision dense par méthode variationnelle.

Les travaux présentés dans les sections 5.2 et 5.3 de ce chapitre ont fait l'objet d'une publication à 3*DPVT*'06 [146], en collaboration avec Julien Mairal et Renaud Keriven.

5.1 Vision par ordinateur

5.1.1 Introduction

L'espace tridimensionnel nous entourant et les objets qu'il contient sont facilement descriptibles et interprétables par l'homme. L'information se formant sur la rétine n'est pourtant composée que de points, environ un million, chacun renseignant sur la quantité de lumière et la couleur de l'espace environnant projeté sur la rétine. Les objets observés n'existent pas sur la rétine, et pourtant leur vision et leur interprétation est possible : cela est le résultat du processus visuel, associant les informations issues de l'observation et les connaissances *a priori*.

Depuis longtemps déjà, la vision suscite l'intérêt de nombreux scientifiques : mathématiciens, géomètres, biologistes ont essayé de comprendre les mathématiques de la vision, l'anatomie et le fonctionnement de l'œil et du cerveau, et ont découvert un système d'une extrême complexité, dont les limites ne sont pas mêmes clairement définies.

Avec le développement des capacités des machines computationnelles, ces scientifiques ont voulu résoudre le problème de la vision quantitativement, en essayant de définir un modèle algorithmique pour la perception visuelle : sans nécessairement chercher à comprendre comment fonctionne la vision biologique, ils ont voulu créer un modèle qui, d'un point de vue extérieur, a des propriétés semblables.

La mise au point de ce modèle a été le fondement d'une *théorie de la vision par ordinateur*, tentant de résoudre des problèmes spécifiques. Elle est un processus de traitement d'information, prenant en entrée des images, apportant certaines connaissances préalables et donnant une modélisation de l'entrée sous la forme d'objets et de relations entre ces objets. Un tel processus calculatoire pour le traitement et la représentation de l'information visuelle a été présenté par David Marr au début des années 80 [148].

Ce *paradigme de Marr* est fondé sur trois étapes :

1. la segmentation des images, à la base de tout système de vision, dont le but est l'extraction d'entités dans les images d'entrée, par détection de points d'intérêts, de contours (discontinuités des niveaux colorimétriques dans l'image ou approximation par représentation analytique), ou extraction de régions homogènes ;
2. la reconstruction de la géométrie d'une scène (points 3D et leurs caractéristiques physiques), induite des images en entrée, et produisant des modélisations facilement manipulables, par stéréoscopie, par *structure from motion* (structure à partir du mouvement) ou par *shape from shading* (forme à partir des ombrage) ;
3. la reconnaissance, consistant essentiellement en la comparaison d'indices visuels en deux ou trois dimensions avec les indices d'objets à reconnaitre, préalablement connus.

Ces trois étapes définissent trois grands champs disciplinaires en vision par ordinateur, ayant chacun introduit de nombreux algorithmes et ayant trouvé la grande quantité d'applications que chacun sait.

Notre application d'intérêt ici, appartenant à l'étape de reconstruction, est la stéréovision, que nous présentons ci-dessous.

5.1.2 Stéréovision

La stéréovision est une méthode de vision par ordinateur permettant, à partir d'au moins deux prises de vues d'un même objet se recoupant, de reconstituer cet objet sous la forme d'un maillage 3D.

On distingue deux types de problèmes possibles pour cette reconstruction.

Premièrement, la capture de la géométrie complète d'un objet est l'objectif de la stéréovision multivue, utilisant un certain nombre de photos de l'objet, prises à partir de points de vues connus. Plusieurs algorithmes existant pour cela sont référencés et comparés qualitativement par Seitz *et al.* dans [206]. Seitz *et al.* proposent également une taxinomie de ces algorithmes, une méthodologie d'évaluation de ceux ci ainsi que plusieurs jeux de données. Plusieurs portages sur GPU ont été réalisés, Labatut *et al.* [128] ont notamment implanté un tel algorithme de stéréovision multivue par évolution variationnelle de surface.

Le second problème, plus ancien, est celui de la reconstruction à partir de deux images, présenté avec la figure 5.1. C'est l'un des domaines de prédilection des chercheurs en vision par ordinateur. Scharstein et Szeliski ont proposé une taxinomie des nombreux algorithmes existants pour résoudre ce problème, ainsi qu'une comparaison de quelques méthodes existantes [201]. Encore plus que pour la stéréovision multivue, de nombreuses solutions ont été proposées. Le problème se ramène toujours au calcul d'un décalage (une disparité) entre les deux images. Cette modélisation n'est pas toujours adaptée et on trouve à l'opposé d'autres méthodes, dites variationnelles, mettant en jeu des équations aux dérivées partielles, telle celle de Stretcha [219], permettant d'estimer une carte de profondeur de la scène.

Jusqu'à la fin de ce chapitre, nous présentons un tel algorithme variationnel de stéréovision dense, implémenté entièrement sur GPU. Cet algorithme est basé sur le travail de Keriven et Faugeras [51], considérant le problème de la stéréovision à n vues, déterminant la surface d'un objet par minimisation d'une certaine énergie incluant un terme de consistance photométrique et des contraintes de régularisation.

Ici, nous nous restreignons aux cas à deux et trois caméras et modélisons la surface comme une carte de profondeur vue à partir d'une caméra de référence. Bien que proposant des résultats précis et lisses, une telle approche est généralement négligée à cause de son manque d'efficacité. C'est ce qui a motivé nos travaux. En effet, alors que la plupart des approches sont basées sur la recherche d'une carte de disparité, nous procédons en deux étapes :

1. Estimation des correspondances (disparité) entre deux images préalablement rectifiées (deux images sont dites rectifiées lorsqu'elles ont subies une transformation affine qui amène leurs droites épipolaires parallèles et alignées. Cela permet de simplifier grandement les recherches ultérieures des correspondances) ;
2. Reconstruction tridimensionnelle de l'objet.

Dans ce contexte, les auteurs de [68], [244], [255], [243], [252], [250] et [71] ont développé des algorithmes en temps-réel ou semi temps-réel sur GPU. Celui présenté par Yang et Pollefeys [250] peut évaluer jusqu'à

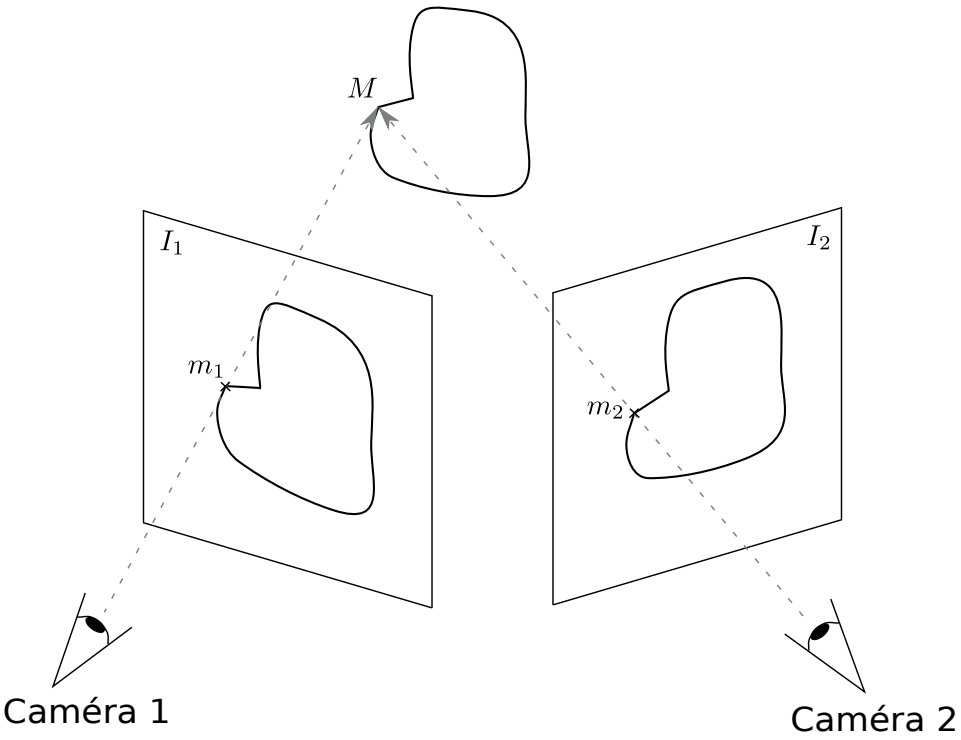


FIGURE 5.1 – *Modèle stéréoscopique à deux caméras. Un point M d'un objet réel est observé à travers deux caméras et se projette en m_1 sur l'image I_1 de la caméra 1 et en m_2 sur l'image I_2 de la caméra 2. La prise de vue sera suivie des étapes de calibration des caméras, de rectification des images, de corrélation pour apparier m_1 et m_2, de détermination de la position spatiale de M puis de reconstruction de la surface.*

289 millions de disparités par seconde, sur une ATI Radeon 9800. Néanmoins, ces résultats ne sont pas dédiés à une reconstruction *précise*. Notre but est d'obtenir aussi rapidement que possible une surface cohérente et précise, à partir de deux ou trois caméras, ou d'un flux vidéo. Un premier essai d'utilisation des GPU pour le même problème a été réalisé par Zach *et al.* [256], présentant des résultats visuellement satisfaisants. Ils utilisent cependant une différence de niveaux de gris non-robuste comme critère de consistance photométrique là où nous utilisons une corrélation croisée normalisée. La principale différence est que nous utilisons une descente de gradient mathématiquement établie là où ils laissent leur surface se mouvoir à vitesse constante, cherchant une décroissance d'énergie.

Bien que basée sur une corrélation croisée, notre méthode ne nécessite aucune rectification des images et n'a pas besoin que ces images soient orientées dans le même sens : elles sont rétro-projetées sur la surface et corrélées directement sur elle. Pour améliorer la convergence et gérer les minima locaux, notre algorithme est multi-échelle en terme de maillage et d'image. La prise en compte des occlusions et la sélection de caméra sont basées sur les normales à la surface.

5.2 Stéréovision à deux caméras

Nous considérons premièrement le cas où deux caméras sont utilisées. Le cas à trois caméras sera couvert dans la section 5.3. L'approche se fera par descente de gradient sur une énergie dont l'expression sera donnée dans la suite.

5.2.1 Modélisation

5.2.1.1 Modèle choisi

En considérant deux caméras totalement calibrées et en prenant la caméra 1 comme référence, nous modélisons l'objet à reconstituer comme la triangulation régulière d'une surface déformable S, sur laquelle chacun de ses vertex M se situe sur un rayon fixe issu du centre optique $O1$ de la caméra 1, comme le montre la figure 5.2. La déformation du maillage se fera par mouvement de chaque vertex le long du rayon auquel il est associé.

Bien qu'asymétrique, cette représentation (maillage régulier et directions de déplacement fixées) simplifie grandement les calculs et permettra de s'adapter facilement à une implémentation GPGPU.

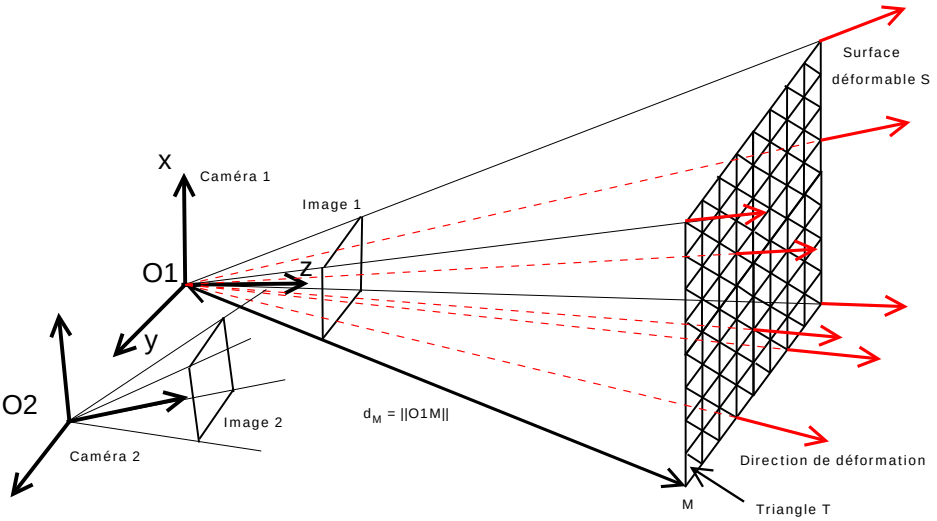


FIGURE 5.2 – *Modèle à deux caméras. La caméra* 1*, de centre optique* $O1$ *est la caméra de référence. Pour chaque point* M *de* S *est lancé un rayon à partir de* $O1$*, ce rayon détermine la direction de déplacement pour* M*.*

Dans la suite, nous noterons d_M la distance du centre $O1$ à un point M du maillage que l'on souhaite faire évoluer. Nous allons maintenant décrire la fonction d'énergie $E(S)$, dont la minimisation sera effectuée par descente de gradient.

5.2.1.2 Formulation de la fonction d'énergie

La formulation la plus simple de l'énergie pourrait être la somme des différences au carré des niveaux de gris :

$$E(S) = \int_S (I_2 \circ \Pi_2(m) - I_1 \circ \Pi_1(m))^2 \, dS(m)$$

avec I_i l'image i, Π_i la projection associée à la caméra i, et $\circ$ désignant la composition de fonction.

Cette énergie semble robuste par le fait que la somme couvre toute la surface, comme mentionné dans [256]. Cependant, nos tests sur des images réelles étant restés peu convaincants, nous avons préféré utiliser une énergie plus robuste, basée sur une corrélation croisée normalisée, où les images sont reprojetées sur la surface (ou bien un plan tangent à S en

m) et corrélé sur un voisinage de m :

$$E(S) = \int_S (1 - \rho(I_1 \circ \Pi_1, I_2 \circ \Pi_2, m))\, dS(m)$$

Une version discrétisée de cette énergie est utilisée, en calculant les corrélations sur chaque triangle T de S :

$$E(S) = \sum_T E_T = \sum_T (1 - \rho_T(I_1 \circ \Pi_1, I_2 \circ \Pi_2)) \tag{5.1}$$

avec, en omettant les dépendances en T dans les notations :

$$\begin{aligned}
\rho_T &= \frac{< I_1, I_2 >}{|I_1|.|I_2|} \qquad (5.2)\\
< I_1, I_2 > &= \frac{1}{A_T} \int_T (I_1 \circ \Pi_1(m) - \overline{I_1 \circ \Pi_1})(I_2 \circ \Pi_2(m) - \overline{I_2 \circ \Pi_2})\, dm \\
\overline{I_i \circ \Pi_i} &= \frac{1}{A_T} \int_T I_i \circ \Pi_i(m)\, dm \\
|I_i| &= \sqrt{< I_i, I_i >}
\end{aligned}$$

A_T représente l'aire du triangle T. Notons qu'aucune métrique sur les triangles n'est utilisée.

Multiplier la corrélation par l'aire de la surface du triangle conduirait à la plus petite surface possible, c'est-à-dire à la convergence de tous les points du maillage vers le centre optique $O1$. C'est un comportement classique d'une méthode basée sur les contours actifs, habituellement contourné par l'addition d'une force ballon. Néanmoins, nous n'utilisons pas une telle force, préférant ajouter à notre énergie un terme de régularisation que nous décrivons par la suite.

5.2.1.3 Minimisation par descente de gradient

La minimisation de l'énergie $E(S)$ se fait par la méthode classique de descente de gradient, ici à pas fixe, avec un schéma multi-résolution, autant en terme de maillage que de tailles d'image, pour traiter les minima locaux autant que possible. L'énergie $E(S)$ étant une fonction des distances $\{d_M\}_{M \in S}$, nous devons calculer son gradient par rapport à ces distances.

Notons $V(M)$ l'ensemble des triangles auxquels appartient le vertex M. Nous pouvons écrire :

$$\frac{\partial E(S)}{\partial d_M} = \sum_{T \in V(M)} \frac{\partial E_T}{\partial d_M}$$

Le calcul de ces quantités lorsque M est un des vertex de T est direct et produit des expressions plutôt longues. En omettant les dépendances en T, on peut dans ce cas écrire :

$$\begin{aligned}
\frac{\partial E_T}{\partial d_M} &= \frac{\partial < I_1, I_2 >}{\partial d_M}\frac{1}{|I_1|.|I_2|} + \frac{< I_1, I_2 >}{|I_1|}\frac{\partial}{\partial d_M}(\frac{1}{|I_2|}) \\
&\quad + \frac{< I_1, I_2 >}{|I_2|}\frac{\partial}{\partial d_M}(\frac{1}{|I_1|}) \\
\frac{\partial}{\partial d_M}\frac{1}{|I_j|} &= -\frac{1}{|I_j|^2}\frac{\partial |I_j|}{\partial d_M} \\
\frac{\partial |I_j|}{\partial d_M} &= \frac{1}{2|I_j|}\frac{\partial < I_1, I_2 >}{\partial d_M}
\end{aligned}$$

Le moyen classique et néanmoins approprié de paramétrer m sur un triangle $T = (M_1, M_2, M_3)$ est de l'écrire sous cette forme :

$$m(\lambda, \gamma) = M_2 + \lambda\overrightarrow{M_2M_1} + \gamma\overrightarrow{M_2M_3}$$

avec $0 \leq \lambda + \gamma \leq 1$. Il est alors possible d'écrire :

$$\begin{aligned}
< I_1, I_2 > &= 2\int_{0\leq\lambda+\gamma\leq 1} (I_1 \circ \Pi_1(m(\lambda,\gamma)) - \overline{I_1 \circ \Pi_1}) \\
&\qquad (I_2 \circ \Pi_2(m(\lambda,\gamma)) - \overline{I_2 \circ \Pi_2})\, d\lambda d\gamma \\
\overline{I_i \circ \Pi_i} &= 2\int_{0\leq\lambda+\gamma\leq 1} I_i \circ \Pi_i(m(\lambda,\gamma))\, d\lambda d\gamma \\
\frac{\partial < I_1, I_2 >}{\partial d_M} &= 2\int_{0\leq\lambda+\gamma\leq 1} \Big((\frac{\partial I_1 \circ \Pi_1(m(\lambda,\gamma))}{\partial d_M} - \frac{\partial \overline{I_1 \circ \Pi_1}}{\partial d_M}) \\
&\qquad \big(I_2 \circ \Pi_2(m(\lambda,\gamma)) - \overline{I_2 \circ \Pi_2}\big)\Big) \\
&\quad + \Big((\frac{\partial I_2 \circ \Pi_2(m(\lambda,\gamma))}{\partial d_M} - \frac{\partial \overline{I_2 \circ \Pi_2}}{\partial d_M}) \\
&\qquad \big(I_1 \circ \Pi_1(m(\lambda,\gamma)) - \overline{I_1 \circ \Pi_1}\big)\Big)\, d\lambda d\gamma \\
\frac{\partial \overline{I_i \circ \Pi_i}}{\partial d_M} &= 2\int_{0\leq\lambda+\gamma\leq 1} I_i \circ \Pi_i(m(\lambda,\gamma))\, d\lambda d\gamma
\end{aligned}$$

Pour discuter de l'implémentation GPU des quantités (5.1) et (5.2), il est important de noter qu'elles se résument à des sommes doubles (les valeurs moyennes étant encapsulées dans la somme principale) dépendant des quantités suivantes :

$$\frac{\partial I_i \circ \Pi_i(m)}{\partial d_M} \tag{5.3}$$

où m est un point de T, et M un de ses vertex. Développons cette équation.

Nous reprenons les notations suivantes, introduites dans [50] : P_i est une matrice 3×3, p_i un vecteur à 3 composantes. Alors h désignera la fonction qui convertit un point en coordonnées homogènes en un point à coordonnées dans l'image :

$$\begin{aligned}
h(x, y, w) &= (x/w, y/w), \text{ si } w \neq 0 \\
\widetilde{\Pi_i}(m) &= P_i m + p_i \\
\Pi(m) &= h \circ \widetilde{\Pi_i}(m)
\end{aligned}$$

La paramétrisation de m peut également être écrite de la façon suivante :

$$m = \alpha_1 M_1 + \alpha_2 M_2 + \alpha_3 M_3$$

Cela permet de représenter la quantité (5.3) de la façon suivante :

$$\begin{aligned} m_i &= \widetilde{\Pi_i}(m) = (m_{i,x}, m_{i,y}, m_{i,z}) \\ P_i &= \begin{pmatrix} P_{i,1} \\ P_{i,2} \\ P_{i,3} \end{pmatrix} \\ \frac{\partial I_i \circ \Pi_i(m)}{\partial d_{M_k}} &= \frac{\alpha_k}{m_{i,z}^2 d_{m_k}} \Big(\nabla_y I_i(h(m_i)), -\nabla_x I_i(h(m_i)) \Big) \\ &\quad \Big((P_{i,2} M_k - m_{i,z})(P_{i,3} M_k - m_{i,y}), \\ &\quad (P_{i,1} M_k - m_{i,z})(P_{i,3} M_k - m_{i,x}) \Big)^T \end{aligned} \tag{5.4}$$

Dans notre implémentation, nous utilisons une forme variant légèrement pour ces équations, permettant de tirer profit des capacités vectorielles des GPU, mais plus difficile à lire.

5.2.2 Implémentation GPU

Nous allons maintenant voir comment est adapté ce calcul sur GPU.

5.2.2.1 Discrétisation

Lors de l'implémentation des quantités décrites ci-dessus sur GPU, les intégrales de la forme $\int_T F(m)dm$, pour une certaine fonction F, sont remplacées par des sommes discrètes de la forme $\sum_p F(p)dm(p)$, où p est déterminé par le rasterizer, et chaque fragment shader va calculer $F(p)$.

Soit T le triangle (M_1, M_2, M_3), p un barycentre tel que $p = \alpha_1 M_1 + \alpha_2 M_2 + \alpha_3 M_3$ avec $\alpha_i \geq 0$ et $\alpha_1 + \alpha_2 + \alpha_3 = 1$. Alors, nous devons calculer la quantité (5.3), à savoir :

$$D_{i,k}(\alpha_1, \alpha_2, \alpha_3) = \frac{\partial I_i \circ \Pi_i(m)}{\partial d_{M_k}}$$

pour $k \in [\![1, 3]\!]$ et tous les triplets $(\alpha_1, \alpha_2, \alpha_3)$ déterminés par le rasterizer. Un calcul direct impliquant les projections de caméras donne :

$$D_{i,k}(\alpha_1, \alpha_2, \alpha_3) = g_i(\alpha_k f_i(M_k), \widetilde{\Pi_i}(m), (\nabla I_i) \circ \Pi_i(p)))$$

avec

$$f_i(M_k) = P_i M_k$$

et g_i facilement déductible de (5.4). On remarque que lorsqu'un fragment shader est appelé pour un point p, les valeurs α_k ne sont pas connues. Ainsi, il est impossible d'obtenir les $\alpha_k f_i(M_k)$ requis. Néanmoins, l'ajout d'un attribut virtuel aux vertex auquel on donne la valeur $f_i(M_k)$ pour M_k et 0 pour $M_j, j \neq k$, fournit automatiquement la quantité interpolée $\alpha_k f_i(M_k)$ lors du calcul de m. De plus, le rasterizer peut calculer $\Pi_i(m)$ car

$$\Pi_i(m) = \alpha_1 \Pi_i(M_1) + \alpha_2 \Pi_i(M_2) + \alpha_3 \Pi_i(M_3)$$

est une valeur interpolée entre trois valeurs qui dépendent seulement d'un vertex. Un des avantages est que les quantités dépendant des vertex M_k sont calculées seulement une fois par le vertex shader et non une fois par point m.

Nous remarquons également que lors du rendu d'un triangle $T = (M_1, M_2, M_3)$, les trois valeurs $D_{i,1}$, $D_{i,2}$ et $D_{i,3}$ peuvent être calculées simultanément pour chaque point m grâce aux capacités vectorielles des GPU.

En choisissant d'effectuer le rendu depuis la caméra 1, nous obtenons que, pour un point donné $m \in T$, la valeur de $I_1 \circ \Pi_1(m)$ est indépendante des positions des vertex de T, soit :

$$D_{1,k}(\alpha_1, \alpha_2, \alpha_3) = 0$$

5.2.2.2 Sommation

Après le calcul de $F(p)$, $p \in T$, pour une certaine fonction F, il est nécessaire de calculer la somme $\sum_{p \in T} F(p) dm(p)$. Une telle réduction est un problème classique, comme nous l'avons vu dans la section 1.4.7.1, aboutissant à un calcul de complexité logarithmique. Ici, nos triangles possèdent un petit nombre de pixels grâce à notre approche multi-échelle, adaptant la taille du maillage aux dimensions de l'image. Ainsi, un simple rendu GPU dans lequel chaque pixel shader s'occupe d'un triangle, en parcourant par une boucle tous ses points, est bien plus efficace.

Pour chaque triangle, il est également nécessaire de sommer, pour chaque vertex, des quantités sur les triangles auxquels il appartient (voir équation (5.1)). Une fois encore, chaque pixel shader va traiter un vertex par une boucle sur les triangles auxquels il appartient.

5.2.2.3 Régularisation

Comme annoncé préalablement, nous n'avons pas utilisé de métrique de surface et devons ajouter un terme de régularisation spatiale à l'énergie. Un mouvement par courbure moyenne aurait pu ici être employé (ou, d'une façon équivalente, un ajout de l'aire de la surface à l'énergie). Nos résultats ont néanmoins été meilleurs par lissage direct d_M, en ajoutant au gradient le terme :

$$K\left((\frac{1}{|N(M)|} \sum_{M' \in N(M)} d_{M'}) - d_M \right)$$

avec $N(M)$ l'ensemble des voisins du vertex M, et K une constante adapté au niveau de détail considéré.

5.2.2.4 Critère d'arrêt

Un critère d'arrêt simple basé sur la valeur maximale du gradient donne de bons résultats. Cependant, fixer le nombre d'itération pour chaque niveau de détail donne des résultats similaires, tout en accélérant la convergence.

5.2.3 Schéma computationnel complet

La figure 5.3 décrit une itération complète du schéma de calcul

1. Assignation des vertex de la surface aux vertex GPU, rendu, calcul de $D_{2,k}(p), k = 1, 2, 3$ pour tout point p ;
2. Assignation des triangles aux pixel shaders, rendu, calcul des doubles sommes avec les quantités $\frac{\partial E_T}{\partial d_M}$;
3. Assignation des vertex de la surface aux pixels shaders, rendu, calcul de la somme de voisinnage menant au gradient $\frac{\partial E(S)}{\partial d_M}$. Calcul des positions des vertex d_m en fonction de ce gradient ;
4. Mis à jour des positions des vertex (stockés dans un buffer particulier) et éventuellement test d'arrêt.

5.2.4 Résultats

Nos expériences ont été réalisées avec un PC standard dont le CPU est cadencé à 3GHz et d'une carte graphique Geforce 7800GTX, maintenant largement démodée.

I_1, I_2 et ∇I_2 sont précalculés lors d'une première phase. Les textures utilisées sont en 16bits (apportant un gain en temps de 40%). Nous avons obtenu une convergence rapide avec une moyenne de 10 itérations par niveau de détail. Il s'est avéré que notre approche multi-résolution empêche de tomber dans des minima locaux.

Pour effectuer des comparaisons, nous avons développé une version CPU de référence, minimisant la même énergie que la version GPU. Cette version CPU a été précautionneusement écrite et compilée avec toutes les optimisations disponibles. Le tableau 5.1 montre les gains obtenus par la version GPU sur la version CPU. On observe un gain moyen de l'ordre de 10 à 15. Cela montre que notre implémentation utilise au mieux le pipeline graphique.

Image	Maillage	GPU	CPU	Gain
64^2	5^2	1.60 kHz	555 Hz	2.9
128^2	9^2	1.33 kHz	116 Hz	11.5
256^2	17^2	464 Hz	28.6 Hz	16.2
512^2	33^2	102 Hz	7.5 Hz	14.1
512^2	65^2	89.4 Hz	7.3 Hz	12.2
512^2	129^2	67.9 Hz	7.2 Hz	9.0

TABLE 5.1 – *Fréquences observées pour* 1000 *itérations à un niveau de détail donné, en fonction des tailles des images et du maillage. Le gain moyen est entre* 10 *et* 15*.*

Les figures 5.4 et 5.5 montrent les jeux de donnés utilisés. Les résultats correspondants, ainsi que quelques temps de calculs sont donnés dans les figures 5.6, 5.7, 5.8 et 5.9. Pour chaque figure présentée, nous avons exécuté 8 itérations pour chaque niveau de détail, excepté pour les deux derniers niveaux, exécutés respectivement 4 et 2 fois. Un temps d'exécution total de 250ms est observé pour une convergence globale. Comme attendu, notre algorithme n'est pas aussi rapide que ceux basés sur des cartes de disparités. Cependant, les applications visées ne sont pas les mêmes.

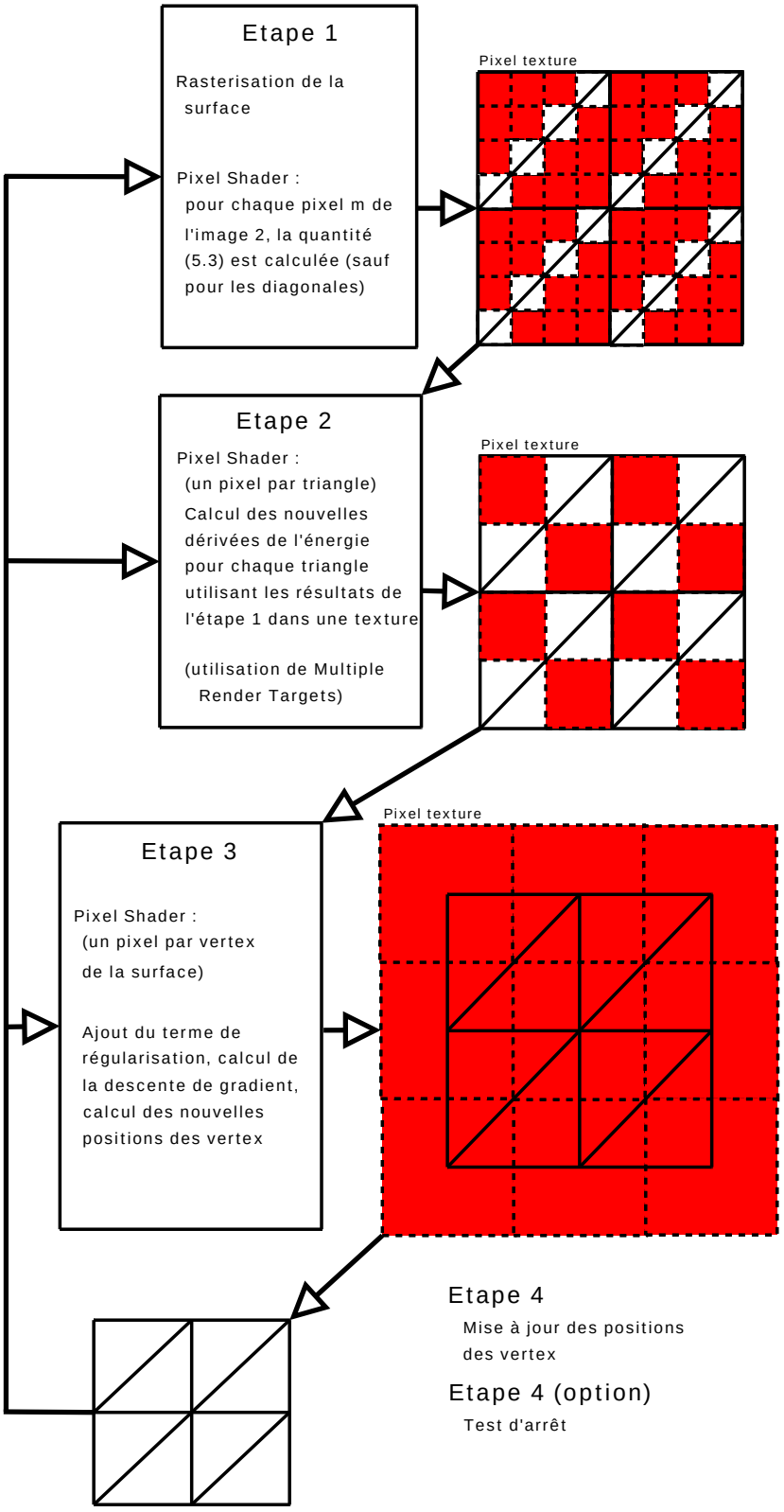


FIGURE 5.3 – *Schéma de calcul complet pour deux caméras. Exemple d'itération sur un maillage de* 3×3 *vertex et une image de taille* 8×8*. Notre implémentation débute avec un maillage* 3×3 *et une image* 32×32*. Un carré rouge représente un pixel effectivement calculé. Les lignes pleines représentent le maillage.*

Un des inconvénients de cette méthode est que les occlusions ne sont pas gérées. Nous pouvons cependant facilement interdire l'algorithme de

FIGURE 5.4 – *Premier et deuxième jeux de données*

FIGURE 5.5 – *Troisième et quatrième jeux de données*

considérer les triangles occlus, avec l'utilisation d'une troisième caméra. C'est ce que nous présentons dans la section suivante.

5.3 Stéréovision à trois caméras

Notre modèle peut facilement être étendu pour gérer trois caméras. Notons la caméra gauche L, la centrale C et celle de droite R. La caméra de référence sera C, tenant le rôle précédemment tenu par la caméra 1 dans la section précédente. La corrélation sera calculée entre les caméras C et L ou entre les caméras C et R. Les seuls critères d'assignation des caméras sont les suivants :

1. La surface est modélisée à partir du centre optique de la caméra C ;
2. La corrélation entre les caméras L et R n'est pas prise en compte.

Ce choix fait, nous divisons à chaque itération les triangles en deux ensembles S_L et S_R, S_L (respectivement S_R) étant l'ensemble des triangles corrélant entre les caméras C et L (respectivement entre les caméras C et R). L'énergie associée est alors similaire à celle donnée dans l'équation

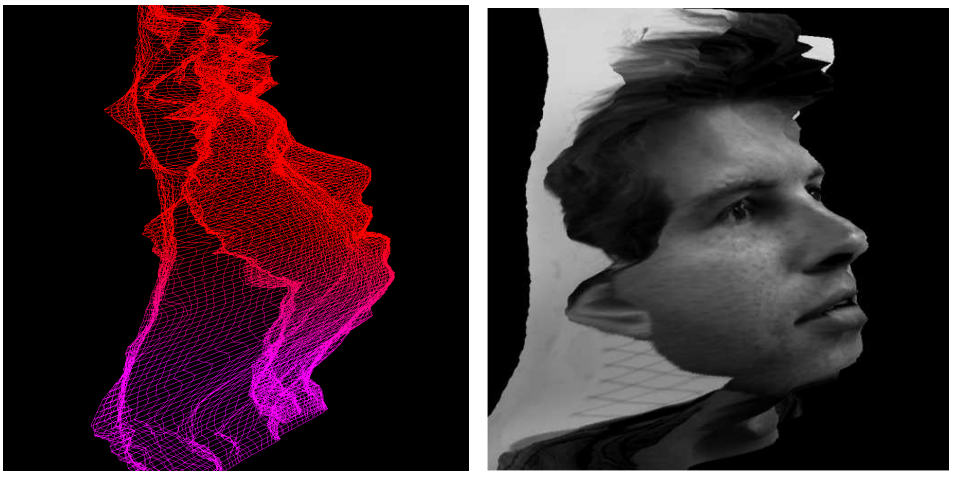


FIGURE 5.6 – *Surface obtenue à partir du premier jeu de données. Temps de calcul :* 240*ms. Deux caméras.*

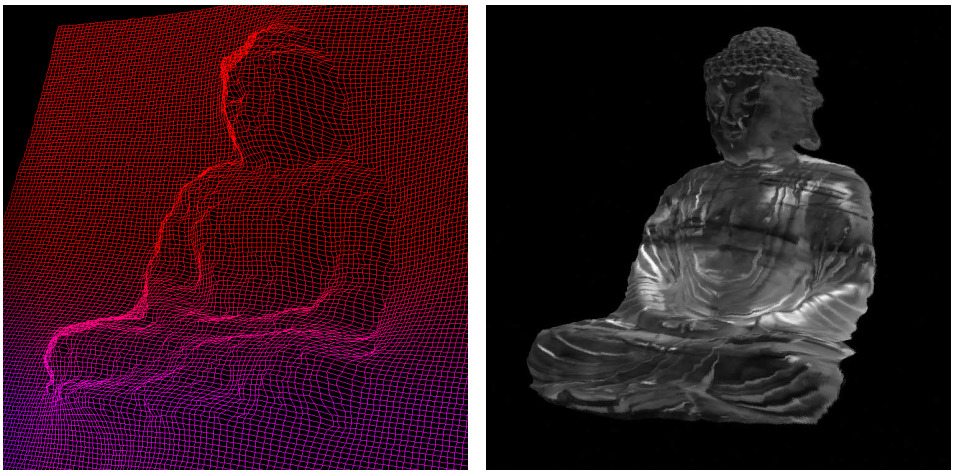


FIGURE 5.7 – *Surface obtenue à partir du deuxième jeu de données. Temps de calcul : 250ms. Deux caméras. Ces images sont la propriété du Pr. Kyros Kutulakos, University of Toronto.*

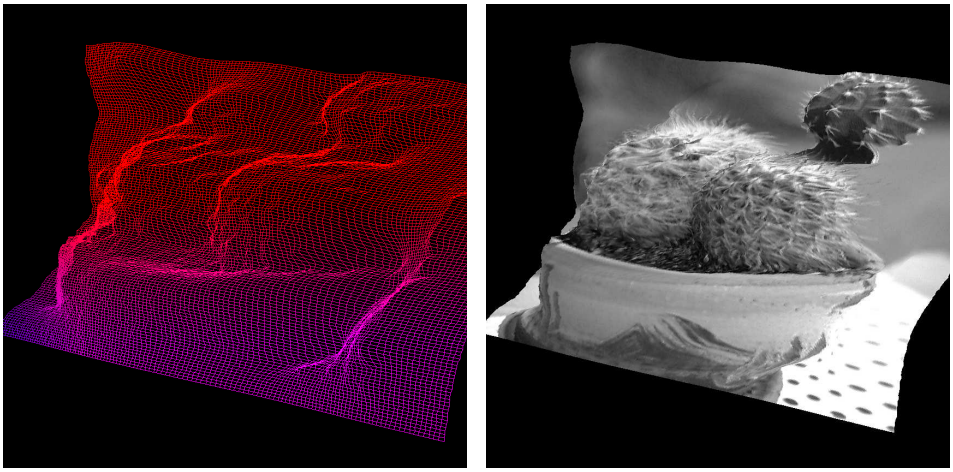


FIGURE 5.8 – *Surface obtenue à partir du troisième jeu de données. Temps de calcul : 230ms. Deux caméras. Ces images sont la propriété du Pr. Kyros Kutulakos, University of Toronto.*

(5.1) :

$$\begin{aligned} E(S) &= \sum_{T \in S_L} (1 - \rho_T(I_C \circ \Pi_C, I_L \circ \Pi_L)) \\ &\quad + \sum_{T \in S_R} (1 - \rho_T(I_C \circ \Pi_C, I_R \circ \Pi_R)) \end{aligned} \tag{5.5}$$

Dans nos tests, nous utilisons simplement un test sur les normales aux triangles pour déterminer la meilleure caméra secondaire entre L et R. De plus, les occlusions peuvent facilement être traitées. En déterminant la surface à partir de la caméra secondaire, on détermine si un triangle est vu ou non. Lorsqu'un triangle n'est vu par aucune caméra, il n'est assigné ni à S_L, ni à S_R. Ensuite, en gardant l'énergie donnée par l'équation (5.5), un triangle qui n'est pas vu par au moins C et une des deux autres caméras n'est pas pris en compte dans cette énergie. En fait, nous ignorons également les triangles qui ne sont pas assez "de face", en un certain sens. Cela apporte une première façon de gérer les discontinuités, un point important que notre modèle ne prenait pas en compte jusqu'ici.

Nous avons implémenté cet algorithme à la fois sur CPU et GPU. La version CPU ne prend pas en compte les occlusions, un test de visibilité

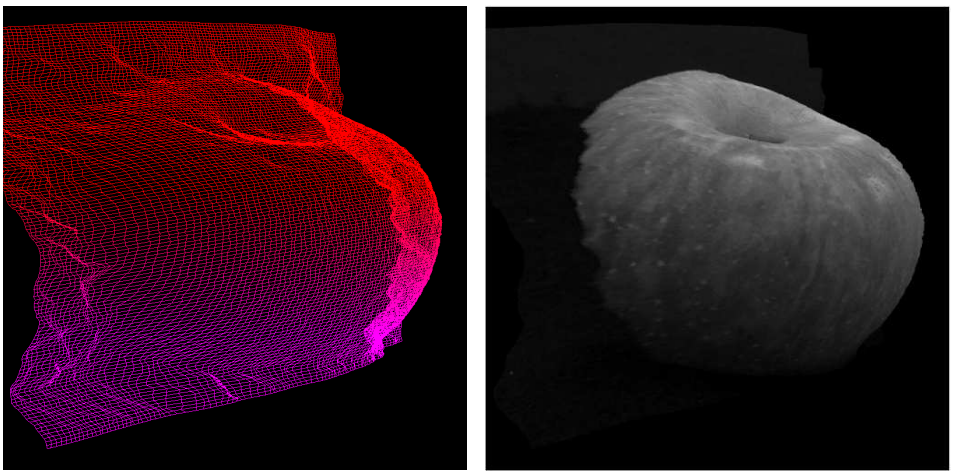


FIGURE 5.9 – *Surface obtenue à partir du quatrième jeu de données. Temps de calcul :* 220*ms. Deux caméras.*

serait trop lent. Cette version s'exécute seulement 9% plus lentement que la version à deux caméras.

Pour implémenter la version GPU, nous utilisons un *stencil buffer* (voir section 1.4.4.1) pour classifier les triangles en deux groupes, pour pouvoir ensuite travailler séparément sur chaque groupe. On mémorise donc grâce à ce buffer l'appartenance de chaque triangle à S_L ou S_R, en marquant un de ces deux groupes par un masque. Il sera simplement nécessaire d'inverser le masque lors du changement de groupe. Ce test est directement câblé sur la carte graphique et est ainsi très rapide. Une vue d'ensemble de l'algorithme pour la version GPU est donnée dans la figure 5.10. Si les triangles étaient aléatoirement distribués entre S_L et S_R, notre algorithme pourrait mener à des problèmes de cache. Ce n'est heureusement pas le cas ici. Nos premiers résultats montrent une surcharge supplémentaire de 30% par rapport à la version à deux caméras.

La figure 5.11 montre le jeu de données utilisé pour notre algorithme à trois caméras. Quelques résultats graphiques sont présentés en figures 5.12 et 5.13.

Grâce à la gestion des occlusions et à une plus grande précision lorsque les trois caméras observent le triangle courant, la reconstruction à trois caméras (figure 5.13) est plus exacte et plus précise que celle à deux caméras (figure 5.12).

5.4 VIDÉO

Nous avons testé notre algorithme sur des séquences vidéo. Pour la première image, la convergence est obtenue en environ 250ms. Par continuité temporelle, nous prenons la surface reconstruite à l'image f comme valeur initiale pour la convergence à l'image $f+1$. Quelques tests préliminaires consistant principalement en un ajustement des choix des niveaux de détails et du nombre d'itérations ont montré des résultats prometteurs en atteignant une fréquence de 8 images par seconde pour l'algorithme à deux caméras. Avec les cartes les plus récentes actuellement, on peut espérer atteindre environ 20 images par secondes. De plus, il est à noter que notre approche peut facilement être transformée en une version utilisant deux cartes graphiques couplées, avec un minimum de transferts entre

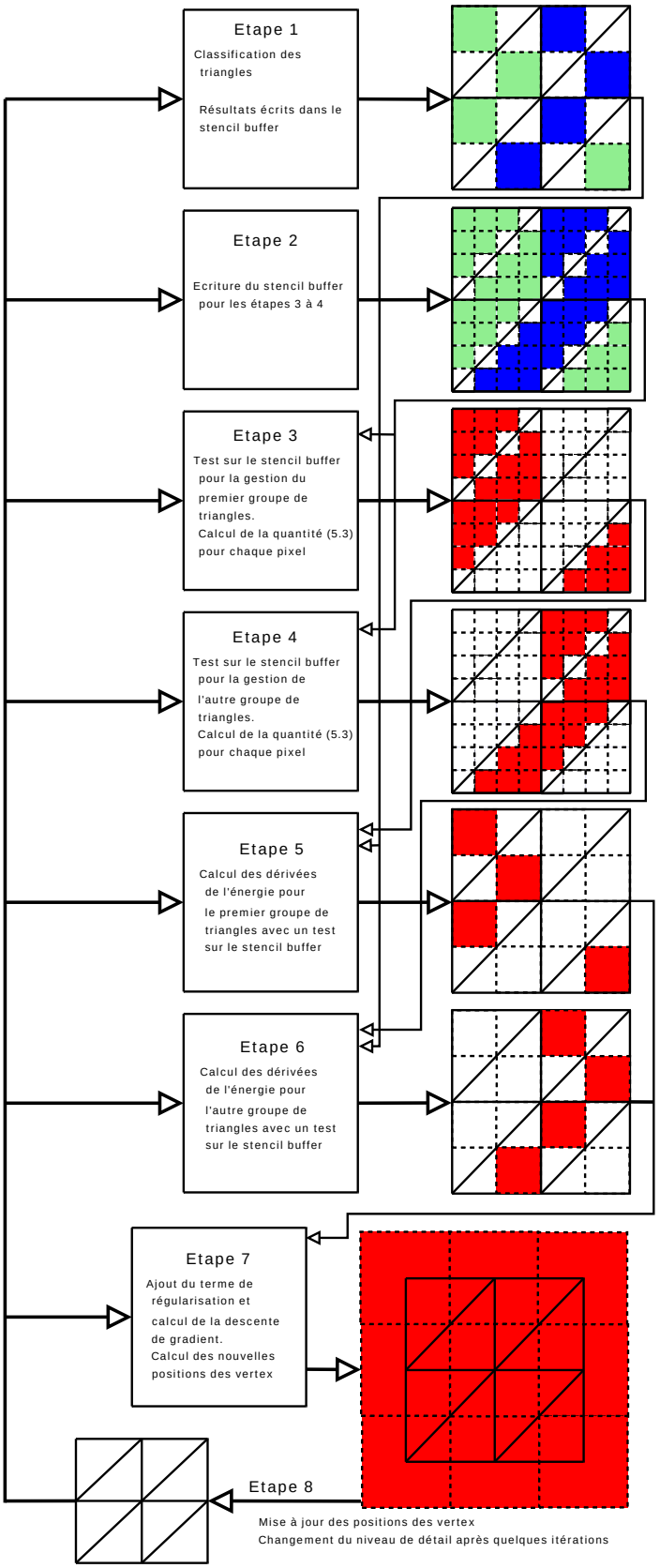


FIGURE 5.10 – *Schéma de calcul complet pour trois caméras.*

les deux, ce qui permettrait d'atteindre facilement le temps réel, tout en restant financièrement très abordable.

FIGURE 5.11 – *Jeu de données pour l'algorithme à trois caméras. Issu de [137].*

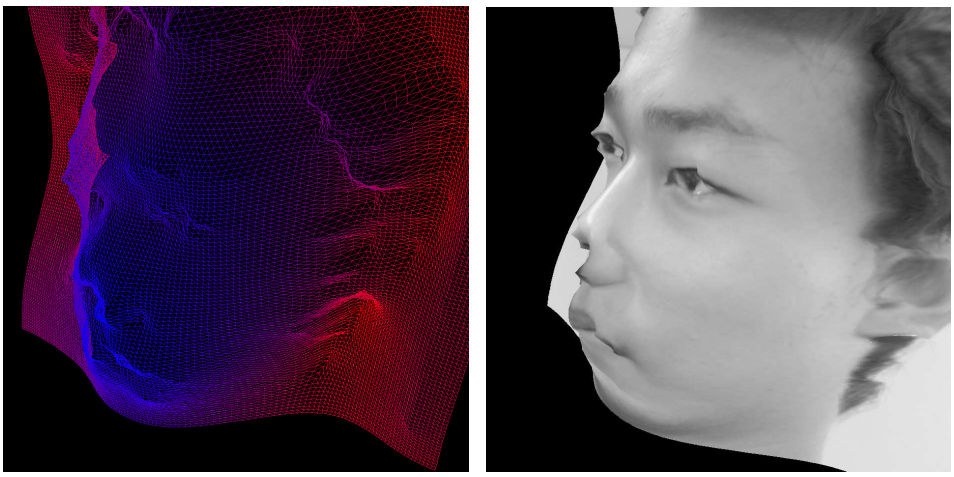


FIGURE 5.12 – *Reconstruction à partir de deux caméras avec zones occludées. Le nez et la partie gauche du visage ne sont pas correctement reconstruits.*

CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons présenté un algorithme de stéréovision dense sur GPU, basé sur des principes variationnels, capable de gérer les occlusions, prenant deux ou trois caméras en entrée et proposant un gain en temps de l'ordre de 10 à 15 et une reconstruction précise des objets observés, ceci grâce à un GPU aujourd'hui totalement dépassé techniquement, bien plus lent que les GPU actuels et obligeant à calculer en plusieurs passes ce qui pourrait l'être directement.

Dans le prochain chapitre, nous décrivons un algorithme de mise en correspondance de points d'intérêts pouvant servir de base aux algorithmes de reconstitution 3D.

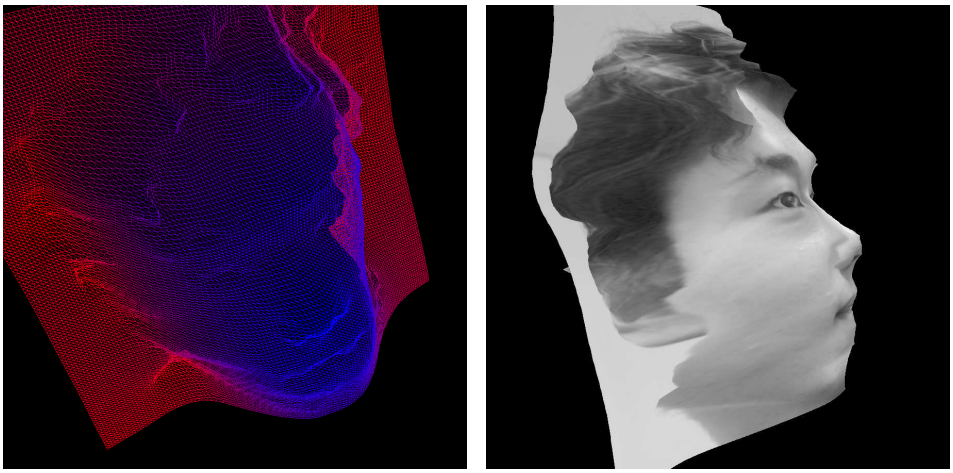


FIGURE 5.13 – *Reconstruction avec trois caméras. La reconstruction est correcte.*

Appariement d'images par GPU

6

Sommaire

L'appariement d'une image donnée avec une seconde image ou un ensemble d'autres images, est classiquement réalisé en repérant et en mettant en correspondance différents indices présents dans ces images. C'est un pré-requis essentiel à de nombreux algorithmes en vision par ordinateur, tel le *Structure From Motion* [94], la reconnaissance d'objet ou la classification. Les indices usuellement utilisés sont des *points d'intérêt*, détectés dans des images, puis caractérisés localement et appariés à travers l'ensemble de ces images. Avec de nombreuses images en entrée, pouvant contenir chacune plusieurs milliers de points d'intérêt, la mise en correspondance rapide de ces points est une étape très consommatrice en temps de calcul, même avec les plus récents CPU, et souvent sous-estimée. Dans les applications concernées, c'est en effet un des bottlenecks majeurs.

Dans ce chapitre, nous présentons dans une première partie quelques unes des méthodes classiques pour détecter et caractériser par un *descripteur* ces points d'intérêt, en particulier les *Scale Invariant Feature Transforms* (*SIFT*), et les *Speeded-Up Robust Features* (*SURF*). Dans une deuxième partie, nous exposons d'abord le procédé usuel d'appariement (*matching*), basé sur une comparaison de *distances entre descripteurs*, puis proposons ensuite une méthode de mise en correspondance rapide de deux images, implémentée sur GPU, s'exécutant classiquement en 20*ms* pour une paire d'images contenant un millier de points d'intérêt chacune. L'objet de notre troisième partie est la présentation d'une application de cette méthode, traitant un large ensemble d'images pour apporter une aide à l'utilisateur lors d'un processus d'acquisition photographique d'une scène, et pouvant indiquer quelles parties de la scène n'ont pas encore été "suffisamment" observées pour pouvoir être reconstituée, une nouvelle image étant appariée de façon interactive avec toutes les précédentes.

Les travaux présentés dans ce chapitre, réalisés dans le cadre du projet ANR *Wired Smart*, ont fait l'objet d'une publication à *ICPR*'08 [33], en collaboration avec Renaud Keriven.

6.1 Détection et caractérisation de points d'intérêt

L'appariement d'images est basé sur la détection, la caractérisation et la mise en correspondances d'*indices* présents dans les images, les *features*, pouvant être en pratique trois primitives principales :

1. Une région d'intérêt, détectée dans l'image, particularisant par exemple une zone homogène ou une zone à texture similaire. La méthode la plus connue pour ce faire, nommée MSER (*Maximally Stable Extremal Regions*) a été introduite par Matas *et al.* [150]. La notion locale de *région extrême* y est définie et utilisée pour maximiser un critère de stabilité ;
2. Une arête, représentant par exemple un contour d'un des objets de l'image. L'opérateur de Sobel [217], consistant en l'application de deux filtres de convolution successivement, fut le premier détecteur de ces contours, par combinaison des gradients horizontaux et verticaux. Canny proposa un nouvel opérateur [29], basé sur une première phase de lissage puis sur une convolution plus adaptée, offrant ainsi la détection de plus d'arêtes et une meilleure location de celles ci. Cet opérateur fut repris par Deriche [43] (sous la forme d'un filtre récursif, plus rapide, nommé filtre de Canny–Deriche) puis par Lindeberg [140] (par une approche différentielle prenant en compte la notion d'échelle) ;
3. Un point, dit *point d'intérêt* (ou pixel dans une image), dénotant par exemple les *coins* de l'image, c'est-à-dire les points de discontinuité dans au moins deux directions de la fonction d'intensité ou de ses dérivées. Moravec a introduit un des premiers détecteurs de points [160], en considérant les variations de la valeur moyenne de l'intensité sur une fenêtre localement glissante. Ce détecteur a ensuite été repris par Harris et Stephens [86], puis par Mikolajczyk et Schmid [156], introduisant la notion d'échelle (le détecteur est alors nommé Harris Laplace) ; ces deux améliorations sont présentées dans la suite. Une autre façon de détecter de tels points est d'utiliser des opérateurs différentiels, tel les *Difference of Gaussian* (DoG) ou les *Hessian of Gaussian* (HoG). Ils permettent une localisation en espace et en échelle et sont entre autres utilisés pour les SIFT, décrits plus loin.

Schmid *et al.* ont proposé une comparaison des différents types de détecteurs de points dans [204].

En pratique, les features les plus utilisées sont les points d'intérêt, détectés en espace et en échelle. Plus faciles et rapides à manier que les autres types de features, ils permettent d'obtenir *in fine* des résultats tout aussi satisfaisants.

Une fois localisés, ces points peuvent être exploités après une étape de caractérisation. Pour ce faire, Lowe a introduit avec les SIFT [143] la notion de *descripteur local*, notion que nous décrivons dans la suite après avoir exposé différents détecteurs de points d'intérêt.

6.1.1 Détection

Nous présentons ici, dans l'ordre chronologique, quatre méthodes de localisation de points d'intérêt, dont nous pouvons voir un résultat classique sur la figure 6.1.

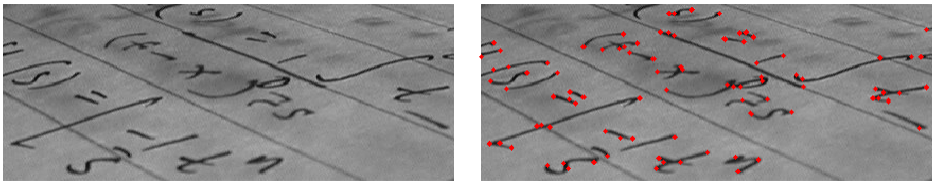


FIGURE 6.1 – *Détection de points d'intérêt. Gauche : image originale. Droite : image originale avec superposition des points détectés.*

6.1.1.1 Harris

Le détecteur de coins de Harris et Stephens [86] est une amélioration de la méthode de Moravec [160]. L'idée de base de ce détecteur est que la variation d'intensité aux alentours d'un coin est importante, alors qu'elle est presque nulle dans les zones homogènes, et nulle dans une direction seulement si le point courant se trouve sur une arête. La figure 6.2 illustre cela.

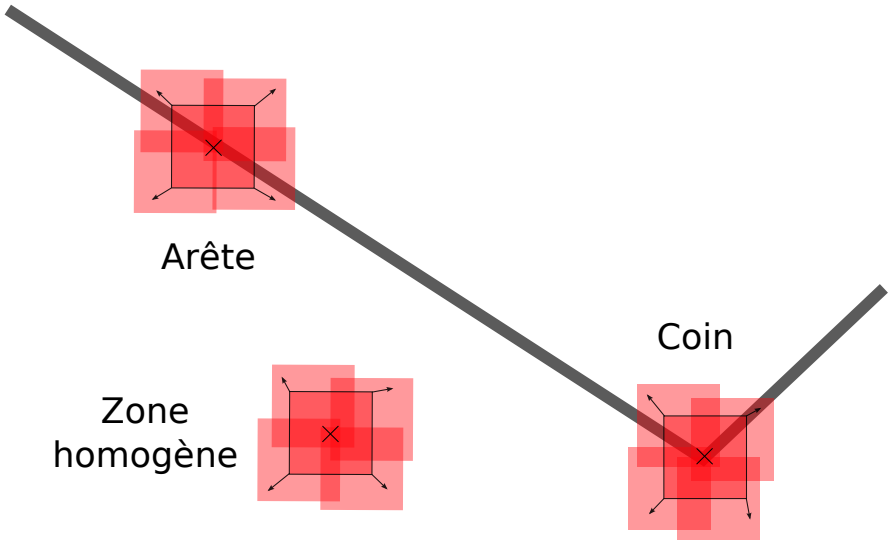


FIGURE 6.2 – *Détecteur de coins de Harris. Pour un point se trouvant dans une zone homogène, la variation d'intensité pour une fenêtre se déplaçant dans un voisinage local est presque nulle. Pour un point sur une arête, elle est nulle dans la direction de l'arête et plus importante dans les autres. Pour un coin, cette variation est importante, quelle que soit la direction de déplacement de la fenêtre.*

En notant I l'intensité et en considérant un patch rectangulaire couvrant une région (u,v), le changement E produit par un déplacement (x,y) est donné par :

$$E(x,y) = \sum_{u,v} w(u,v)[I(u+x,v+y) - I(u,v)]^2$$

où w spécifie la fenêtre de déplacement ; cette fonction vaut 1 dans une fenêtre rectangulaire et 0 ailleurs.

Ce détecteur induit une réponse anisotrope, car les valeurs prises par (x, y) appartiennent à un ensemble discret correspondant à des décalages de 45° en 45°. De plus petits déplacements peuvent être employés grâce à une extension analytique de la quantité $E(x, y)$, par développement limité de l'intensité I, que Harris et Stephens ont proposé : $E(x, y)$, par développement limité de l'intensité I :

$$I(x+u, y+v) = I(u, v) + x\frac{\partial I}{\partial x} + y\frac{\partial I}{\partial y} + O(x^2, y^2)$$

Ils réécrivent alors la quantité $E(x, y)$ de la façon suivante, avec une approximation bilinéaire (6.1) valable pour de petites variations de (x, y) :

$$\begin{aligned} E(x,y) &= \sum_{u,v} w(u,v)[x\frac{\partial I}{\partial y} + y\frac{\partial I}{\partial y} + O(x^2, y^2)]^2 \\ &\approx (x,y) \cdot M \cdot (x,y)^t \end{aligned} \tag{6.1}$$

avec

$$M = \sum_{u,v} w(u,v) \begin{bmatrix} \frac{\partial I}{\partial x}^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \frac{\partial I}{\partial y}^2 \end{bmatrix}$$

M est appelée la matrice des seconds moments, ou matrice d'autocorrélation.

En choisissant la fonction w gaussienne et non plus en créneau comme précédemment, on lisse la détection, autrement trop bruitée :

$$w(u, v) = e^{-\frac{u^2+v^2}{2\sigma^2}}$$

avec σ donné.

E est ainsi proche d'une fonction d'auto-corrélation locale. Harris et Stephens montrent [86] qu'un point peut être considéré comme un coin lorsque les valeurs propres de M sont toutes deux grandes, et définissent une mesure R de réponse au coin, donnée par :

$$R = \mathrm{Det}(M) - k\,\mathrm{Tr}(M)^2$$

avec $k = 0.04$, déterminé empiriquement.

Cette mesure permet une classification directe : un point avec une valeur R négative se trouve sur une arête, une valeur faible de R indique une localisation dans une zone homogène, et une valeur positive révèle un coin. On considère que les points de Harris sont ceux pour lequel R est supérieur à un seuil défini.

Ce détecteur a la propriété d'être invariant par rotation et partiellement invariant par changement affine d'intensité, mais n'est pas invariant par changement d'échelle ; cela sera corrigé par le détecteur présenté dans le paragraphe suivant.

6.1.1.2 Harris-Laplace

Le détecteur de Harris-Laplace, ou Harris multi-échelle, introduit par Mikolajczyk et Schmid [156], s'appuie fortement à la fois sur la façon dont sont localisés les points de Harris, présentée précédemment, et sur

une représentation dans un espace d'échelle gaussien. Pour une image I, la représentation en espace d'échelle en un point (x, y) est donnée par la famille des signaux dérivés $L(x, y, \sigma_t)$ définie comme la convolution de la fonction d'intensité de l'image par un noyau gaussien $g(x, y, \sigma_t)$:

$$L(x, y, \sigma_t) = g(x, y, \sigma_t) * I(x, y) \tag{6.2}$$

avec le noyau :

$$g(x, y, \sigma_t) = \frac{1}{2\pi\sigma_t} e^{-\frac{x^2+y^2}{2\sigma_t}}$$

La convolution n'est effectuée que sur les dimensions spatiales x et y. Elle a pour effet de lisser l'image en utilisant une fenêtre de la taille du noyau. Cela apporte au détecteur de Harris-Laplace l'invariance par changement d'échelle qui faisait défaut jusqu'ici. Il est en effet possible de modifier la matrice des seconds moments M pour insérer cette invariance dans le détecteur. On note alors :

$$M = \mu(x, y, \sigma_s, \sigma_t) = \sigma_t^2 g(x, y, \sigma_s) * \begin{bmatrix} L_{xx}(x, y, \sigma_t) & L_{xy}(x, y, \sigma_t) \\ L_{xy}(x, y, \sigma_t) & L_{yy}(x, y, \sigma_t) \end{bmatrix}$$

avec $L_{xx} = \frac{\partial^2 L}{\partial x^2}$, $L_{yy} = \frac{\partial^2 L}{\partial y^2}$ et $L_{xy} = \frac{\partial^2 L}{\partial x \partial y}$. Les facteurs σ_s et σ_t sont deux échelles différentes pour le noyau gaussien : σs est l'échelle pour le noyau d'intégration, et σt celle du noyau de dérivation utilisé dans l'équation (6.2).

A partir de cette matrice M, la détection de Harris-Laplace consiste en deux phases présentées ci-dessous.

Détection des points à différentes échelles Elle est faite à des échelles prédéfinies dans le but de réduire l'espace de recherche. Mikolajczyk et Schmid utilisent la suite $\{k_1^i \sigma_0\}_{i \in [\![0,n]\!]}$ avec $k_1 = 1.4$ pour les échelles d'intégration σ_s. Ils choisissent l'échelle de différentiation $\sigma_t = k_2 \sigma_s$ en prenant $k_2 = 0.7$. La mesure R de réponse en coin est calculée de façon similaire à précédemment :

$$R = \mathrm{Det}(\mu(x, y, \sigma_s, \sigma_t)) - k\,\mathrm{Tr}(\mu(x, y, \sigma_s, \sigma_t))^2$$

De la même façon, les points retenus sont les maxima locaux pour cette quantité, au dessus d'un seuil donné.

Choix de l'échelle caractéristique Un algorithme itératif, basé sur les travaux de Lindeberg [141], permet de localiser les coins et de sélectionner l'échelle caractéristique de chacun d'entre eux à partir des points initialement détectés à l'échelle σ_s. A chaque étape i de l'algorithme, (i) on choisit l'échelle $\sigma_s^{(i+1)}$ qui maximise le *LoG*, ou *Laplacian of Gaussian* (soit l'opérateur laplacien appliqué à la fonction L précédente : $\nabla^2 L = \frac{\partial^2 L}{\partial x^2} + \frac{\partial^2 L}{\partial y^2}$), (ii) on utilise cette échelle pour la localisation spatiale du coin, maximisant la mesure de réponse au coin R sur un voisinnage local, (iii) on test le critère d'arrêt (convergence en position et en échelle). Les points satisfaisant ce critère d'arrêt sont les coins recherchés, maximisant le LoG à travers les échelles ainsi que la mesure de réponse au coin R.

6.1.1.3 DoG et SIFT

Introduits par Lowe [143] en 1999, les *SIFT*, *Scale-Invariant Feature Transform*, ont connu un succès notable en tant que détecteur de points d'intérêts : invariants par rotation, par changement d'échelle, par changement d'illumination, en partir par changement de point de vue, ils présentent de plus une résistance au bruit, ce que les précédents détecteurs ne supportaient pas. En plus de localiser les points d'intérêts, les SIFT permettent également une caractérisation robuste de ces points. Nous abordons cette caractérisation dans la section 6.1.2.1.

Pour la détection des points d'intérêts, SIFT utilise les DoG, *Difference of Gaussian*. En pratique on procède en deux étapes (deux autres étapes seront nécessaires à la caractérisation) : un repérage des extrema dans un espace d'échelle suivi d'une localisation précise de certains de ces points candidats passant différents tests.

Détection des extrema dans l'espace d'échelle Comme pour Harris-Laplace 6.1.1.2, les SIFT sont basés sur un espace d'échelle construit par convolution de l'image d'origine I avec un filtre gaussien g :

$$L(x, y, k\sigma) = g(x, y, k\sigma) * I(x, y)$$

Pour une détection efficace de points stables dans l'espace d'échelle, on cherche les extrema de la fonction *DoG*, différence de gaussiennes convoluée avec l'image :

$$\begin{aligned} D(x, y, \sigma) &= (g(x, y, k_i\sigma) - g(x, y, k_j\sigma)) * I(x, y) \\ &= L(x, y, k_i\sigma) - L(x, y, k_j\sigma) \end{aligned}$$

avec $g(x, y, \sigma)$ un noyau gaussien d'échelle σ.

Ainsi, cette DoG est la simple soustraction de deux images préalablement convoluées avec le filtre gaussien, pour deux échelles distinctes. Les différentes images obtenues sont regroupées par *octave*, une octave correspondant à doubler la valeur de σ l'échelle. La valeur de k_i est choisie telle qu'on obtient un nombre fixe d'images par octave. On calcule les DoG en prenant deux images adjacentes au sein d'une même octave (voir figure 6.3).

Les extrema sont recherchés dans les images comme les extrema locaux sur un 26-voisinage dans l'espace d'échelle, comme indiqué sur la figure 6.4.

Décrit ainsi, on remarque que l'approche par DoG est une approximation des *LoG* normalisés $\sigma^2\nabla^2 g$, pour lesquels les extrema représentent les poins les plus stables, d'après Mikolajczyk [156]. En effet, l'équation de la chaleur (paramétrée en σ, et non en $t = \sigma^2$) amène :

$$\frac{\partial g}{\partial \sigma} = \sigma\nabla^2 g$$

En approchant cette équation aux dérivées partielle par une différence finie pour des échelles σ et $k\sigma$, on obtient :

$$\frac{\partial g}{\partial \sigma} \approx \frac{g(x, y, k\sigma) - g(x, y, \sigma)}{(k-1)\sigma}$$

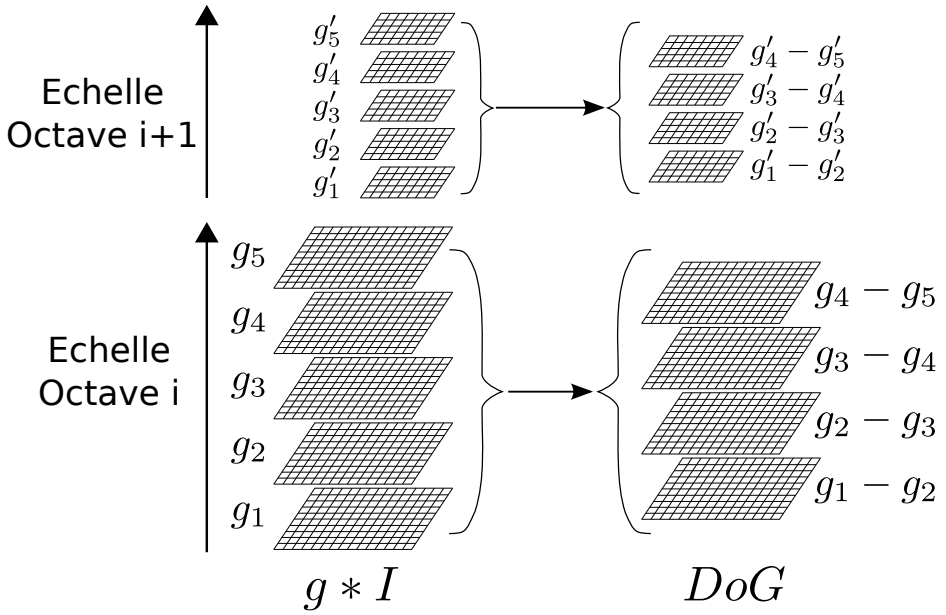


FIGURE 6.3 – *Calcul des images de la* Difference of Gaussian, DoG. *Pour chaque octave de l'espace d'échelle, les images résultant de la convolution de l'image originale par des gaussiennes d'échelles croissantes sont soustraites les unes aux autres, une image étant soustraite à ses images adjacentes, pour donner les composants de la DoG. Entre chaque octave, les tailles de l'image sont divisées par deux pour accélérer les calculs, et le processus est répété. Adapté de [144].*

d'où :

$$g(x, y, k\sigma) - g(x, y, \sigma) \approx (k-1)\sigma^2\nabla^2 g$$

Ainsi, la fonction DoG pour laquelle les échelles varient d'un facteur constant k contiennent déjà la normalisation d'échelle σ^2, requise pour que le Laplacien soit invariant par changement d'échelle. Le facteur $(k-1)$, constant, n'influence pas la détection des extrema. Pour k tendant vers 1, l'erreur d'approximation tend vers 0. Lowe trouve empiriquement que cette approximation n'a aucun impact sur la stabilité des extrema détectés.

Localisation des points d'intérêt La détection précédente dans l'espace d'échelle repère beaucoup trop de points susceptibles d'être des points d'intérêts. Une étape d'adaptation des points aux données environnantes, pour une meilleure localisation dans l'espace d'échelle et pour la courbure principale, est réalisée, permettant alors de rejeter ceux qui ont un contraste trop faible (extrema instables) ou qui sont localisés sur des arêtes.

Premièrement, chaque point candidat voit sa position interpolée avec ses voisins. Une expansion de Taylor de la fonction d'échelle DoG est opérée :

$$D(\mathbf{x}) = D + \frac{\partial D^t}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^t\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

D est ici évalué au point courant et $\mathbf{x} = (x, y, \sigma)$ est l'offset à partir de ce point. L'extremum $\hat{\mathbf{x}}$ est déterminée en prenant la dérivée de cette fonction par rapport à $\mathbf{x}$ égale à 0. Si cet offset $\hat{\mathbf{x}}$ est supérieur à 0.5 dans les trois dimensions, alors l'extremum est plus proche d'un point voisin, la position du point candidat courant est alors changée et l'interpolation à

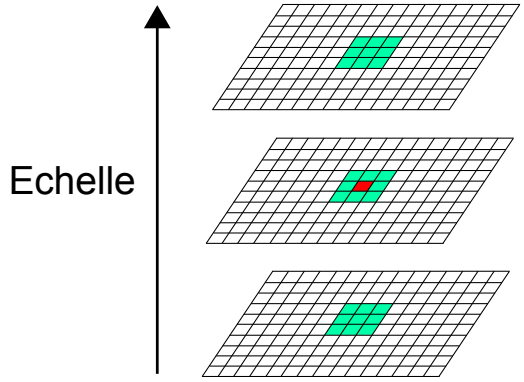


FIGURE 6.4 – *Détermination des extrema dans la DoG. On cherche ces extrema en comparant une valeur (pixel en rouge) à celle contenues dans un voisinage* $3 \times 3 \times 3$*(en vert), en espace et en échelle. Adapté de [144].*

nouveau faite à partir de ce point. L'offset final $\hat{\mathbf{x}}$ est ajouté à la position du point courant pour obtenir l'estimation interpolée de la position de l'extremum.

Ensuite, les points de faible contraste sont supprimés en estimant la quantité :

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D}{\partial \mathbf{x}}^t \hat{\mathbf{x}}$$

avec $\hat{\mathbf{x}}$ l'offset estimé précédemment. Tous les points avec une telle valeur inférieure à un seuil donné sont rejetés. Lowe utilise 0.03 comme seuil (avec pour hypothèse que les intensités dans l'image sont comprises dans l'intervalle $[0, 1]$).

Enfin, on cherche à éliminer les points mal localisés et ayant une forte réponse en arête à la fonction DoG, pour des raisons de stabilité. Un tel point aura dans la DoG une courbure principale forte dans la direction perpendiculaire à l'arête, et une faible courbure principale le long de l'arête. Un rapport de la première à la deuxième courbure principale suffisamment proche de 1 assure alors la présence d'un coin et non d'une arête. Ces deux courbures principales peuvent être déterminées à partir de la matrice hessienne $\mathbf{H}$ de taille 2×2 calculée à la position et à l'échelle du point courant :

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

avec $D_{xx} = \frac{\partial^2 D}{\partial x^2}$, $D_{yy} = \frac{\partial^2 D}{\partial y^2}$ et $D_{xy} = \frac{\partial^2 D}{\partial x \partial y}$.

Lowe montre [144] que pour obtenir un rapport de ces courbures inférieur à un seuil r, il n'est pas nécessaire de déterminer les valeurs propres de $\mathbf{H}$, mais qu'il est suffisant de vérifier l'inégalité suivante :

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r+1)^2}{r}$$

avec

$$\begin{aligned} \mathrm{Tr}(\mathbf{H}) &= D_{xx} + D_{yy} \\ \mathrm{Det}(\mathbf{H}) &= D_{xx}D_{yy} - D_{xy}^2 \end{aligned}$$

Le test à effectuer ne nécessite ainsi que très peu de calculs. Lowe utilise la valeur de seuil $r = 10$.

6.1.1.4 SURF

Les *SURF*, *Speeded Up Robust Features*, présentés par Bay *et al.* [10], sont d'autres détecteurs (et descripteur) de points d'intérêts dans des images utilisés en vision par ordinateur. Largement inspiré des SIFT, ils ont été récemment prouvés plus efficaces que ces derniers, en terme de nombre de points détectés et caractérisés par unité de temps [9]. La partie détecteur des SURF est également basée sur l'utilisation de la matrice hessienne, ayant de bonnes performances aussi bien en terme de précision que de temps de calcul. Néanmoins, au lieu d'utiliser des mesures différentes pour sélectionner la position et l'échelle, les SURF utilisent le déterminant de la matrice hessienne pour les deux. Ce détecteur est également appelé *Fast Hessian Detector*.

Pour un point (x, y) à l'échelle σ, la matrice hessienne $\mathbf{H}$ est définie de la façon suivante :

$$\mathbf{H}(x, y, \sigma) = \begin{bmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{bmatrix}$$

avec, comme précédemment, $L_{xx} = \frac{\partial^2 L}{\partial x^2}$, $L_{yy} = \frac{\partial^2 L}{\partial y^2}$ et $L_{xy} = \frac{\partial^2 L}{\partial x \partial y}$, et $L(x, y, \sigma)$ définie comme dans la section 6.1.1.2. Le noyau gaussien utilisé est, en pratique, discrétisé et réduit, comme le montre la figure 6.5 partie gauche. Après le succès des approximation des LoG par Lowe (voir section 6.1.1.3), Bay *et al.* poussent l'approximation encore plus loin, en utilisant cette fois ci des *box filters*, présentés en figure 6.5 partie droite.

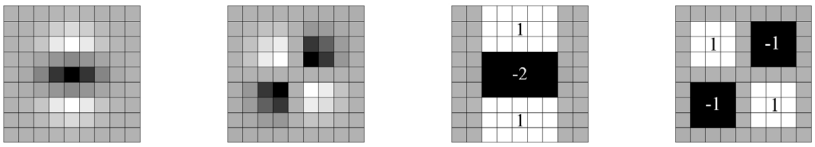


FIGURE 6.5 – *Filtres d'approximation du noyau gaussien avec* $\sigma = 1.2$. *De gauche à droite : les discrétisations réduites des dérivées secondes en yy et en xy, et les approximations de celles ci par les* box filters *de Bay* et al. *Les zones en gris sont égales à* 0. *Extrait de [10]*

Quelque soit la taille des filtres, ces approximations par box filters peuvent être très facilement calculées à partir des images intégrales, précalculées. Une image intégrale F est déterminée à partir d'une image I de la façon suivante :

$$F(x, y) = I(x, y) + F(x - 1, y) + F(x, y - 1) - F(x - 1, y - 1)$$

Chaque pixel de cette image contient ainsi la somme de tous les pixels au dessus et à gauche de sa position dans l'image initiale. Dans notre cas, calculer la somme S des pixels sur une zone rectangulaire $ABCD$ est alors très simple :

$$\begin{cases} A & = & (x_A, y_A), \ldots, D = (x_D, y_D) \\ x_A & = & x_D < x_B = x_C \\ y_A & = & y_B < y_C = y_D \end{cases}$$

alors :

$$S = F(C) + F(A) - F(B) - F(D)$$

Pour obtenir la réponse aux box filters, il suffit de calculer trois ou quatre sommes de ce type, de les pondérer en accord avec la figure 6.5 et de les sommer.

Alors que les SIFT sous-échantillonnent les images après convolution avec le noyau gaussien, les SURF procèdent autrement : grâce aux box filters et aux images intégrales, il n'est pas nécessaire d'appliquer le même filtre à la couche précédente, il est possible d'appliquer de tels filtres de n'importe quelle taille à la même vitesse directement sur l'image originale. Il n'est donc plus nécessaire de réduire itérativement l'image, il suffit d'agrandir la taille du filtre d'un incrément dépendant de l'octave. En prenant pour filtres de base les filtres 9×9 présentés en figure 6.5, les filtres suivants de la première octave auront pour tailles 15×15, 21×21, 27×27 et ainsi de suite.

Entre chaque changement d'octave, l'incrément de taille sur les filtres est doublé. S'il est de 6 pour la première octave (filtres 9×9, 15×15,...), il sera alors de 12 pour la deuxième (9×9, 21×21,...), 24 pour la troisième,...

La localisation des points d'intérêt est alors faite similairement à celle des SIFT : suppression des non-extrema locaux, sur un voisinage $3 \times 3 \times 3$ (espace et échelle), puis interpolation en espace et en échelle des extrema du déterminant de la matrice hessienne, de façon similaire aux SIFT.

6.1.2 Caractérisation par descripteur

Une fois les points d'intérêt localisés dans l'image, éventuellement dans l'espace d'échelle (à chaque point est alors associé un facteur d'échelle σ), il est nécessaire de caractériser ces points, sur un voisinage local, dans le but de pouvoir apparier correctement un point d'intérêt d'une image I_1 avec un autre d'une image I_2, tout en s'assurant autant que possible que ces deux points représentent le même point physique de la scène capturée.

Nous présentons ici les méthodes de caractérisation utilisées par Lowe pour les SIFT, et par Bay *et al.* pour les SURF. Ces méthodes cherchent toutes deux à décrire le point par un histogramme de l'orientation des gradients, dans un voisinage local.

6.1.2.1 SIFT

L'algorithme de Lowe [143] opère en deux étapes pour le calcul d'un descripteur local : assignation d'une orientation puis construction d'un descripteur.

Assignation d'une orientation A chaque point d'intérêt retenu est affectée une orientation. L'invariance par rotation n'est pas atteinte avec une mesure adaptée, mais par l'assignation de cette orientation. L'approche est la suivante : on considère l'échelle σ du point pour déterminer l'image $L(x, y, \sigma)$ à partir de laquelle se fait l'ensemble des calculs. Par cette image (que l'on nomme à présent simplement $L(x, y)$), on précalcule la magnitude $m(x, y)$ et l'orientation $\theta(x, y)$ du gradient pour tous les pixels au

voisinage du point d'intérêt :

$$\begin{aligned} m(x,y) &= \sqrt{(L(x+1,y)-L(x-1,y))^2+(L(x,y+1)-L(x,y-1))^2} \\ \theta(x,y) &= \tan^{-1}\left(\frac{L(x,y+1)-L(x,y-1)}{(L(x+1,y)-L(x-1,y)}\right) \end{aligned}$$

Un histogramme des orientations sur le voisinage du point d'intérêt est construit, chaque échantillon ajouté à l'histogramme étant pondéré par sa magnitude de gradient et par une fenêtre gaussienne circulaire avec un facteur de 1.5σ. Les pics de cet histogramme correspondent aux orientations principales. Le plus haut de ces pics est détecté et retenu. Tout autre pic au moins égal à 80% du premier est également retenu, ce qui produit des orientations multiples. Dans ce cas, de nouveaux points d'intérêts sont créés, avec la même localisation et la même échelle, mais variant en orientation.

Construction du descripteur Une fois la position, l'échelle et l'orientation d'un point d'intérêt déterminées, la prochaine étape est de calculer pour ce point un descripteur aussi résistant que possible aux variations encore non prises en compte, telles le changement du point de vue et de l'illumination. L'idée est, comme pour l'assignation des orientations, de déterminer les magnitudes et orientations du gradient sur un voisinage déterminé en suivant l'orientation précédemment assignée au point en question. Une fonction de pondération par un noyau gaussien de taille 1.5σ est appliquée à chaque pixel de ce voisinage.

Le descripteur du point est alors construit comme l'ensemble des histogrammes sur les 16 régions de tailles 4×4 composant le voisinage 16×16 orienté du point, chacun de ces histogrammes reflétant 8 directions (ces tailles sont celles pour lesquelles Lowe obtient les meilleurs résultats dans ses expériences [143]). Un descripteur d'un point est alors représenté sous la forme d'un vecteur à $16 \times 8 = 128$ dimensions. La figure 6.6 illustre la construction de ce descripteur. Une interpolation trilinéaire est effectuée sur les valeurs des gradients pour une distribution spatiale plus lisse sur les 16 histogrammes générés. Ceci permet d'éviter les changements trop brusques d'une région à une autre.

Le vecteur de description obtenu est finalement normalisé afin d'obtenir une invariance aux changements affines d'illumination (contraste et luminosité). Afin de réduire l'impact des variations non-linéaires pouvant modifier de façon importante les magnitudes de gradients, un seuillage est ensuite appliqué à ce vecteur normalisé, supprimant toutes les magnitudes au dessus de 0.2, puis en renormalisant le vecteur résultant. La valeur 0.2 a été déterminée empiriquement.

Plusieurs améliorations des SIFT ont été proposées, entre autres par Sinha *et al.* [215], portant partiellement l'algorithme sur GPU (série GeForce 7) : la quasi-totalité de la détection des points y est opérée, alors que les calculs des histogrammes de gradients et la construction du descripteur sont laissés au CPU, un portage GPU de cette phase n'étant pas avantageux. Un gain en temps de 10 est obtenu, permettant à des applications de traiter des images vidéo. Une autre implémentation GPU, très récente, a été proposée par Heymann *et al.* [98], portant l'intégralité de l'algorithme, adapté au GPU (NVidia Quadro FX 3400) et arrivant à des

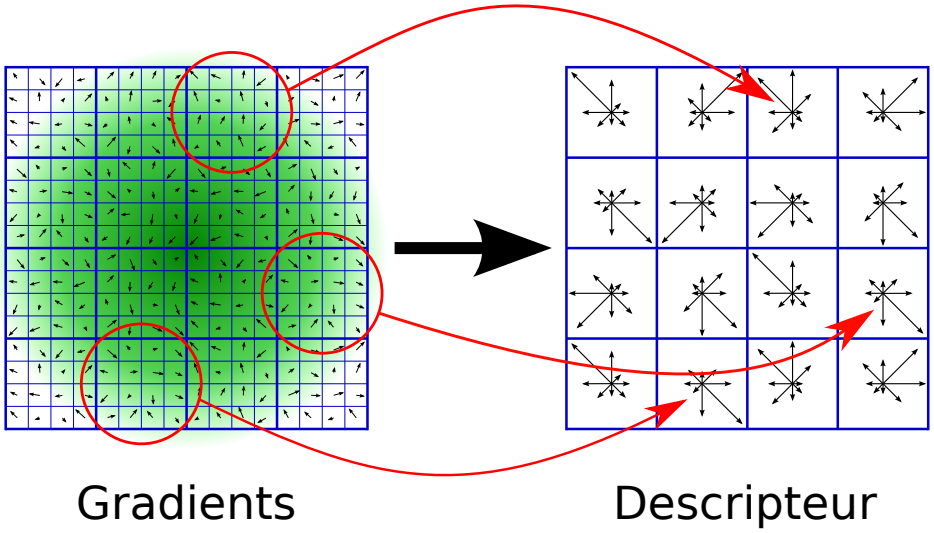


FIGURE 6.6 – *Construction du descripteur SIFT. Les magnitudes et orientations du gradient sont d'abord calculées en chaque pixel du voisinage du point d'intérêt considéré centré, puis pondérées par un noyau gaussien, représenté en vert (figure de gauche). Pour chaque sous-région de taille* 4×4 *de ce voisinage, on construit un histogramme des orientations, représenté sous forme d'étoile d'orientations dans chacune des cases de la figure de droite, et pour lequel la longueur des flèches correspond à la somme des magnitudes proches de la direction correspondante. Le vecteur descripteur résultant est de dimensions* $16 \times 8 = 128$*. Adapté de [144].*

gains de 5 par rapport à une version optimisée CPU, pour l'extraction et la description des points d'intérêt d'images vidéo (640×480 pixels).

6.1.2.2 SURF

Malgré les très bonnes performances des SIFT, leur haute dimensionnalité est leur principal inconvénient, rendant la phase d'exploitation (appariement des descripteurs, voir section 6.2) peu commode en terme de calculs. Avec les SURF, Bay *et al.* procèdent de façon similaire, en deux étapes, en exploitant les mêmes propriétés mais en proposant une charge de calcul bien inférieure.

Orientation Pour conserver l'invariance par rotation, une orientation est assignée à chaque point retenu. Pour cela, les réponses aux ondelettes de Haar 2D (*Haar wavelet*, voir figure 6.7 gauche) en x et y sont premièrement déterminées à l'échelle σ (l'échelle à laquelle a été détectée le point), plus simples à calculer que les magnitudes de gradients, sur tous les points d'un voisinage du point d'intérêt, ce voisinage étant circulaire de rayon 6σ.

A haute échelle, la taille des ondelettes est grande, le filtrage peut alors être effectué rapidement grâce aux images intégrales ; cela ne nécessite que quelques opérations. En reprenant les notations introduites dans la partie droite de la figure 6.7, et en notant $R(\mathbf{x})$ la réponse en un point $\mathbf{x}$ et

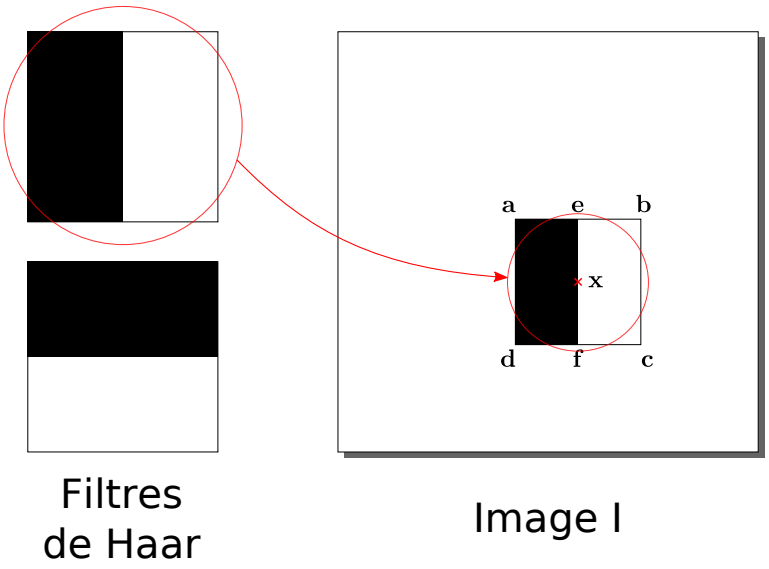


FIGURE 6.7 – *Ondelettes de Haar utilisées et calcul de la réponse en un point. Gauche : ondelettes de Haar utilisées en x (haut) et en y (bas). La partie blanche est pondérée à* 1*, la partie noire à* -1*. Droite : calcul de la réponse en x en un point* **x** *de l'image I. Les points* **a** *à* **f** *sont utilisés pour calculer la réponse en* **x** *de l'image I, donnée à l'équation (6.3).*

$F(\mathbf{u})$ l'intensité de l'image intégrale en un point $\mathbf{u}$, on a :

$$\begin{aligned} R(\mathbf{x}) &= -\big(F(\mathbf{f}) + F(\mathbf{a}) - F(\mathbf{e}) - F(\mathbf{d})\big) + \big(F(\mathbf{c}) + F(\mathbf{e}) - F(\mathbf{b}) - F(\mathbf{f})\big) \\ &= -F(\mathbf{a}) - F(\mathbf{b}) + F(\mathbf{c}) + F(\mathbf{d}) + 2F(\mathbf{e}) - 2F(\mathbf{f}) \end{aligned} \tag{6.3}$$

Une fois les réponses à ces ondelettes calculées et pondérées par une gaussienne d'échelle $2.5\,\sigma$ centrée sur le point d'intérêt, les réponses sont représentées comme des vecteurs dans un espace $2D$ et l'orientation principale est alors estimée en sommant toutes les réponses sur une fenêtre glissante (dont la taille est un paramètre à spécifier).

Construction du descripteur Similairement aux SIFT, pour la construction du descripteur d'un point d'intérêt d'échelle σ et d'orientation θ, une région d'intérêt centrée sur ce point et orienté suivant θ est considérée. Cette région carrée est séparée en sous-régions de taille 4×4. Les réponses d_x et d_y aux ondelettes de Haar, calculées suivant θ, sont pondérées par un noyau gaussien d'échelle $3.3\,\sigma$. Ces réponses sont sommées sur chaque sous-région et forment les premières composantes du vecteur décrivant cette sous-région. Une information à propos de la polarité des variations est ajoutée en déterminant la somme des valeurs absolues des réponses d_x et d_y.

Ainsi, chaque sous-région est décrite par un vecteur à quatre composantes : $\mathbf{v} = \big(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|\big)$. Le descripteur du point d'intérêt est, au final, de dimension $4 \times 4 \times 4 = 64$.

En tant qu'approximation de gradients, la réponse aux ondelettes assure une robustesse aux offsets sur la luminosité, et l'invariance aux changements affines de contraste est obtenue en normalisant le vecteur descripteur.

U-SURF Bay *et al.* décrivent également un algorithme simplifié pour la description de point d'intérêt, nommé *U-SURF* (*Upright SURF*), omettant

la détermination de l'orientation principale et donc l'alignement de la région d'intérêt sur cette orientation. Ce descripteur n'est ainsi pas robuste aux rotations, mais il est très rapide à calculer, et bien adapté aux applications où la caméra conserve un alignement à peu près horizontal tout au long de la prise de vue.

Terriberry *et al.* [228] ont très récemment proposé une implémentation quasi intégralement GPU des SURF, atteignant des gains en temps variant de 3.6 à 5.5 par rapport à l'implémentation GPU des SIFT de Heymann *et al.* [98] en fonction des paramètres utilisés, permettant de traiter avec une GeForce 8800GTX des images à haute définition en un temps raisonnable (jusqu'à 30Hz pour 1280×960 pixels).

6.2 Appariement de deux images

L'appariement de deux images I et J représentant la même scène, communément utilisé dans des algorithmes de reconstruction 3D ou de *stitching* (combinaison de plusieurs photographies avec des zones de recoupement pour en créer une seule plus grande), est une des étapes de l'extraction d'informations des images. Une fois les vecteurs de description calculés pour tous les points d'intérêts de ces deux images, cet appariement consiste à déterminer, pour chaque point d'intérêt de I, l'éventuel point correspondant dans J, et réciproquement.

6.2.1 Méthode d'appariement

Pour un point i de I, l'idée est de déterminer le point j de J le plus proche (ou bien les points j_k de J les plus proches), au sens d'une distance dans l'espace des descripteurs (*feature space*), puis d'appliquer différents filtrages pour conserver le maximum de bons appariements tout en supprimant le maximum de mauvais. La figure 6.8 montre un exemple de paires de points bien appariés.

Figure 6.8 – *Exemple d'appariements corrects de points d'intérêts. A chaque paire de points appariés, on a associé une couleur. Les tailles des cercles représentent l'échelle à laquelle les points ont été détectés.*

6.2.1.1 Distances

Entre les points d'intérêts $i \in I$ et $j \in J$, on établit une distance $d(i, j)$, calculée à partir de leurs descripteurs respectifs dans $\mathbb{R}^d$, avec d la dimension du descripteur employé (128 pour SIFT, 64 pour SURF).

La question du choix de la distance ne se pose que peu. En effet, c'est une distance euclidienne, donnant les résultats les plus probants, qui est

utilisée dans la large majorité des cas (par exemple [144, 10]) :

$$d_E(i,j) = \sqrt{\sum_{k=1}^{d} (i_k - j_k)^2}$$

avec i_k la $k^{\text{ième}}$ composante du vecteur descripteur de i.

D'autres types de distances sont parfois employées, par exemple une distance de Mahalanobis modifiée [155]. De façon générale, toute norme définie sur l'espace des descripteurs (de dimension finie) peut être utilisée.

6.2.1.2 Plus proche voisin approché (ANN)

Pour déterminer le plus proche voisin de i parmi les points détectés dans J, il est nécessaire de calculer les distances de i à tous ces points, ce qui est la tâche la plus consommatrice en temps de calcul, proposant une complexité quadratique en $O(m^2)$, m étant le nombre moyen de points d'intérêt détectés dans une image.

Néanmoins, si l'utilisateur est prêt à tolérer de légères erreurs dans la recherche (retournant un point qui n'est pas le plus proche voisin, mais qui n'en n'est pas éloigné de façon significative), il est possible de grandement accélérer ces calculs, en utilisant les *ANN* (*Approximate Nearest Neighbors*) [4], offrant une complexité réduite à $O(m \times \log(m))$.

Ce gain est dû à l'utilisation que font les ANN de structures de clustering de type *kd-trees* ou *box-decomposition trees*, une décomposition de l'espace à d dimensions, dans laquelle les points sont classés afin de restreindre l'espace de recherche. Pour un point d'interrogation i, le plus proche ou les k plus proches voisins peuvent alors être déterminés en un nombre bien plus restreint d'opérations. N'importe quelle métrique de Minkowski peut être spécifiée pour le calcul des distances entre points.

6.2.1.3 Filtrage

Le choix du plus proche voisin au sens euclidien n'est pas un critère suffisant pour obtenir de bons appariements. Cet unique critère amène à un nombre important d'appariements faux, en particulier sur des images représentant des motifs qui se répètent, les descripteurs se ressemblant alors fortement. De plus, la mise en correspondance induite n'est pas nécessairement réciproque : si le point $j \in J$ est le plus proches voisins de $i \in I$, il n'est généralement pas vrai que i est le plus proche voisin de j.

Pour palier à ces défauts, une ou plusieurs étapes de filtrage des appariements sont nécessaires. On présente ici deux filtrages différents : un simple et efficace, puis un autre plus robuste mais plus lourd. Ces deux filtrages peuvent être utilisés conjointement.

Deux plus proches voisins Une première idée naïve serait de défausser les paires dont la distance est inférieure à un seuil donné. Cela est cependant peu efficace, étant donné que certains descripteurs sont plus discriminants que d'autres. Lowe [144] propose une façon plus efficace et peu contraignante, nécessitant tout de même de déterminer pour un point $i \in I$ les deux plus proches voisins j et $j' \in J$, j étant le plus proche, de comparer leurs distances à i, puis de supprimer les appariements pour

lesquels le rapport de ces deux distances $\frac{d(i,j)}{d(i,j')}$ est inférieur à un seuil donné. Lorsqu'un point a deux voisins proches dans l'espace des descripteurs, mais potentiellement éloignée dans l'espace de l'image, il est préférable de lever l'ambigüité en supprimant l'appariement. Lowe propose d'utiliser la valeur 0.8 comme seuil. Ses expériences montrent qu'avec ce filtrage, 90% des mauvais appariements sont supprimés, alors que 5% des bons seulement le sont. Cette efficacité est due au fait que lorsqu'un point est mal apparié, il arrive fréquemment que la distance au second plus proche voisin soit très similaire à celle au plus proche voisin.

Ce filtrage peut être vu comme une estimation de la densité des mauvais appariements dans cette partie de l'espace des descripteurs.

RANSAC L'algorithme itératif *RANSAC* [54], pour *RANdom SAmple Consensus*, est une méthode d'estimation des paramètres d'un modèle à partir d'un jeu d'observations contenant des *outliers*, échantillons aberrants, ainsi que des *inliers*, échantillons correctes. L'algorithme suppose qu'il existe une procédure de détermination des paramètres d'un modèle à partir d'un sous-ensemble, généralement très réduit, des échantillons. L'exemple canonique d'utilisation du RANSAC est la régression linéaire, l'approche d'un jeu d'échantillons par une droite.

Itérativement, l'algorithme sélectionne au hasard un sous-ensemble d'échantillons, considérés comme des hypothétiques inliers, détermine les paramètres d'un modèle, puis teste tous les autres échantillons avec ces paramètres : si un point correspond bien au modèle, il est lui aussi considéré comme un possible inlier. Si un pourcentage suffisant des échantillons est labellisé comme hypothétiques inliers, le modèle est considéré comme raisonnablement bon. Il est alors ré-estimé à partir de tous les inliers potentiels, et l'erreur globale des inliers relativement au modèle est évaluée. Toutes ces étapes sont répétées un nombre fini de fois, chaque itération produisant un modèle soit rejeté (trop peu d'inliers), soit retenu, avec une erreur d'approximation. Au final, le modèle proposant la plus petite erreur est retenu.

Dans notre cas, on cherche à estimer la matrice fondamentale caractérisant la relation existante entre les points correspondants dans une paire d'images. Il existe des méthodes exactes de calcul d'une telle matrice fondamental à partir de 7 paires de points [259], les sous-ensembles utilisés à chaque itération du RANSAC sont donc de taille 7. Les appariements considérés comme des outliers pour le modèle estimé sont rejetés, ce qui permet de supprimer jusqu'à 95% des mauvais appariements, tout en conservant la quasi-totalité des bons.

6.2.2 Algorithme GPU d'appariement de deux images

Malgré l'emploi de techniques et outils réduisant la complexité algorithmique de l'appariement de deux images tels les ANN (voir section 6.2.1.2), la mise en correspondance de N images, de complexité algorithmique $O(N^2)$, représente une charge de calculs encore trop lourde pour pouvoir être utilisé de façon interactive dans des applications de reconstruction 3D (ou du type de celle présentée dans la section 6.3).

Dans cette section, nous présentons une méthode implémenté sur GPU permettant d'accélérer, par rapport à une version CPU optimisée,

Résolution image	HD (1920×1080)	960×540
CPU	3	12.3
GPU	8.5	30.2
GPU4 $\times$ 4	15.0	60.1
Gain GPU/CPU	2.8	2.5
Gain GPU4 $\times$ 4/CPU	5	5

TABLE 6.1 – *Vitesse de calcul (en Hz) et gain pour notre implémentation GPU des SURF, pour différentes méthodes et différentes tailles d'images. Seul le temps de détection des points d'intérêt et de leur échelle est pris en compte. La version CPU a été réimplémentée par nos soins ; la version GPU calcule le Hessien sur GPU en traitant les images les unes après les autres ; la version GPU*4 $\times$ 4 *fait de même en prenant les images* 4 *par* 4*.*

les temps de calcul pour la mise en correspondance de N images (N pouvant aller jusqu'à plusieurs centaines), chacune pouvant contenir plus de 1000 points d'intérêt.

6.2.2.1 Implémentation

Nous avons repris l'algorithme SURF de Bay *et al.* [10], en implémentant le calcul des descripteurs et la détermination des appariements sur GPU.

Détection et description Bien que ce ne soit pas notre contribution principale, nous avons développé une version des SURF mixte CPU/GPU (les travaux de Terriberry *et al.* [228] n'étaient alors pas encore disponibles). Comme dans [215], les dérivées des images (les Hessiens) sont calculées sur GPU, alors que les descripteurs sont évalués sur CPU, à l'aide de la librairie originale [10]. Il est à noter que nous ne considérons pas le temps de calcul des descripteurs comme un *bottleneck*, étant donné que ces calculs ne sont fait qu'une fois par image, et que le problème auquel nous nous sommes attachés est d'accélérer l'appariement d'un ensemble d'images.

La table 6.1 présente quelques temps classiques pour l'extraction de points d'intérêt et le calcul de l'échelle par SURF, pour deux tailles d'images et pour différentes méthodes. Le temps de calcul du descripteur n'est pas pris en compte. CPU correspond à une version intégralement CPU des SURF, réimplémentée par nos soins. La version GPU est notre première version sur GPU dans laquelle les images sont traitées une par une ; ses temps prennent en compte le transfert de chaque image du CPU vers le GPU, le calcul du Hessien sur GPU, le transfert retour et la fin des calculs (sauf calcul du descripteur). La version GPU4 $\times$ 4 est la même que la précédente à la différence près que les images sont traitées quatre par quatre, y compris pour les transferts. On observe un gain de 2.5 en faveur de notre implémentation GPU par rapport à une implémentation CPU. Ce gain peut monter jusqu'à 5 en traitant les images quatre par quatre : Cg est en effet adapté à de tels calculs grâce à ses variables de type `float4`, vecteurs à 4 dimensions sur lesquelles une même opération peut être appliquée parallèlement. De plus, en traitant les images quatre par quatre, on réduit le nombre de transferts (sans diminuer la quantité de données à transférer).

Appariement La méthode utilisée, de type *brute force*, est décrite avec la figure 6.9.

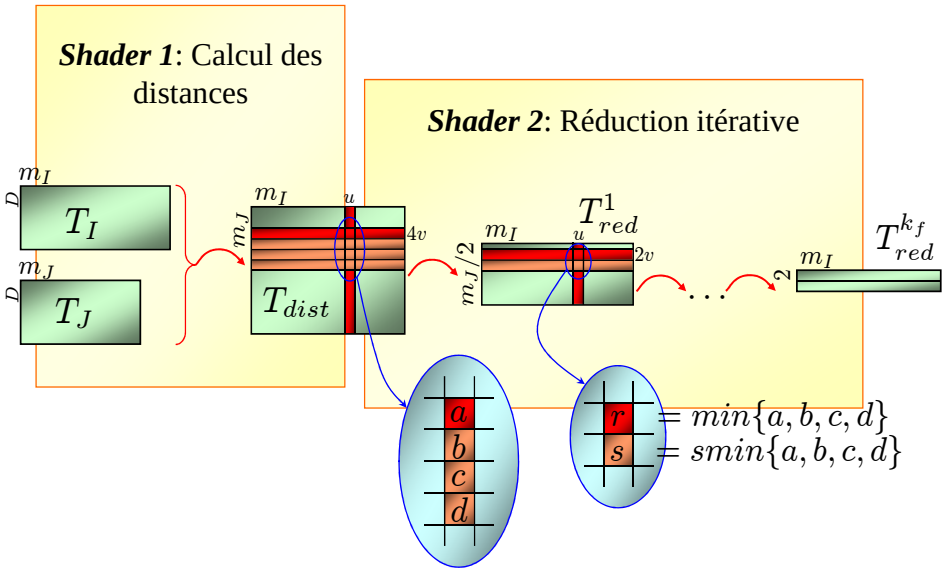


FIGURE 6.9 – *Méthode d'appariement de deux images sur GPU. Les m_I descripteurs des points détectés dans l'image I sont stockés en colonne dans la texture T_I ; il en est de même pour les m_J descripteurs des points de l'image J stockés dans T_J. Toutes les distances euclidiennes $d(i,j)$ pour $i \in [\![1, m_I]\!]$ un point d'intérêt de I et $j \in [\![1, m_J]\!]$ un point d'intérêt de J sont calculées et stockées dans une texture T_{dist}, avec les index des points j. Par réductions successives sur T_{dist}, on détermine les deux plus proches voisins de i dans J, j_i et j'_i, ainsi que leurs index. La même suite d'opération est effectuée en inversant les rôles de I et J.*

Une fois l'ensemble des descripteurs calculés et stockés dans la mémoire du GPU sous forme de textures (une par image en entrée), l'appariement de deux images I et J suit le principe suivant :

1. Le GPU calcule en parallèle les distances entre chaque paire de descripteurs $(i,j) \in I \times J$;
2. Dans un second temps, dans le but d'utiliser par la suite un filtrage par les deux plus proches voisins (voir section 6.2.1.3), le GPU détermine pour chaque descripteur $i \in I$ les deux plus proches voisins j_i et j'_i par réduction sur T_{dist}. L'opération similaire en inversant les rôles de I et J est réalisée simultanément, elle ne sera pas décrite.

Plus précisément, pour chaque image I, notons m_I le nombre de points d'intérêts détectés dans I, et D la dimension de l'espace des descripteurs retenus ($D = 64$ pour les SURF). Une texture T_I de taille $M_I \times D$ est créée et remplie avec les composantes des m_I descripteurs, chaque descripteur occupant une colonne. Dans un but de clarté, nous supposons que les points d'intérêts sont indexés par des entiers, et nous noterons indifféremment i le point d'intérêt, son descripteur ou son index.

Pour deux images I et J données, une nouvelle texture T_{dist} de taille $m_I \times m_J$ est créée et remplie avec un premier shader calculant les distances euclidiennes $d(i,j)$. Comme l'appariement doit également fournir les index des points appariés j_i et j'_i et pas seulement leur distances à i, T_{dist} stocke en réalité un couple distance×index. Cela est facilement géré par le GPU, capable de manier nativement des types vectoriels (de dimension

4 au maximum) :

$$T_{dist}(i,j) = \left(\sum_{k=1}^{D} \left(T_I(i,k) - T_J(j,k)\right)^2 \ , \ j \right)$$

L'étape suivante est une réduction itérative sur la texture T_{dist}, opérée avec un second shader, pour lequel la hauteur de la texture de sortie sera réduite à 2, pour garder les deux plus proches voisins (avec leurs index). Dans le cas où m_J n'est pas un multiple de 2, on aura artificiellement augmenté cette valeur jusqu'à la prochaine puissance de 2, en remplissant les lignes correspondantes dans T_{dist} par des codes d'erreur. Cette pyramide de textures va de $T^0_{red} = T_{dist}$ à $T^{k_f}_{red}$, où k_f est dépendant de m_J : $k_f = \log_2 m_J - 1$. La $k^{\text{ième}}$ itération va remplir la texture T^k_{red} à partir de la texture T^{k-1}_{red} de cette façon :

$$\begin{cases} T^k_{red}(u,2v) &= min\{ \ T^{k-1}_{red}(u,4v),\ T^{k-1}_{red}(u,4v+1), \\ & \qquad T^{k-1}_{red}(u,4v+2),\ T^{k-1}_{red}(u,4v+3) \ \} \\ T^k_{red}(u,2v+1) &= smin\{ \ T^{k-1}_{red}(u,4v),\ T^{k-1}_{red}(u,4v+1), \\ & \qquad T^{k-1}_{red}(u,4v+2),\ T^{k-1}_{red}(u,4v+3) \ \} \end{cases}$$

avec $min\{\cdot\}$ (respectivement $smin\{\cdot\}$) désignant le premier (respectivement le second) minimum des quatre paires, en comparant les premiers éléments de ces paires (les distances calculées). Finalement, $T^{k_f}_{red}$ est récupéré dans la mémoire CPU, donnant pour chaque point d'intérêt i de I les deux plus proches voisins j_i et j'_i leur distances à i :

$$\begin{cases} T^{k_f}_{red}(i,1) = \left(d(i,j_i) \ , \ j_i \right) \\ \\ T^{k_f}_{red}(i,2) = \left(d(i,j'_i) \ , \ j'_i \right) \end{cases}$$

Pour une optimisation supplémentaire, nous utilisons des textures de vecteurs 4D au lieu de vecteurs 2D. Ainsi, nous regroupons $T^k_{red}(u,2v)$ et $T^k_{red}(u,2v+1)$ au pixel (u,v) d'une texture plus petite, et plus important, nous calculons $min\{\cdot\}$ et $smin\{\cdot\}$ simultanément.

6.2.2.2 Résultats

Tous les tests ont été effectués sur un CPU Xeon $3GHz$, équippé d'une carte NVidia GeForce 8800 Ultra. La table 6.2 montre les temps pour les appariements par paire d'images, en ms. Ni la détection des points d'intérêts, ni le calcul des descripteurs, ni la construction de la structure utilisée avec les ANN n'est pris en compte ici, étant donné que ces opérations sont réalisées une seule fois par image lors d'un prétraitement. Nous comparons dans cette table notre méthode à une implémentation naïve CPU en $O(m^2)$ (m étant le nombre moyen de points d'intérêt par image) et à une version ANN. Nous avons effectués ces tests sur des données synthétiques et des données réelles sans constater de différence pertinente.

m	512	1024	2048	4096
Temps CPU	160	660	4230	24220
Temps ANN	73	290	1200	7990
Temps GPU	6.8	19	71	270
Rapport CPU/ANN	2.2	2.3	3.5	3.0
Rapport CPU/GPU	23.5	34.6	59.6	89.7
Rapport ANN/GPU	10.7	15.3	16.9	29.6

TABLE 6.2 – *Temps en ms par appariement de paire d'images et gains pour trois algorithmes, en fonction du nombre m de points d'intérêts, appliqués sur des données réelles ou synthétiques. Les trois algorithmes sont CPU (naif, en $O(m^2)$), ANN (voisins approchés), GPU (notre méthode). Les temps de calculs sur des données réelles ou générées aléatoirement sont similaires.*

Comme prévu, notre méthode GPU et la méthode CPU naïve croissent de façon quadratique avec m. Néanmoins, les problèmes de cache, la rapidité de la mémoire GPU et la latence due aux transferts CPU/GPU font que les temps de calcul croissent plus rapidement que prévu sur CPU, et moins rapidement que prévu pour notre version GPU. Cela explique le gain en temps variant entre 23 et 90. Typiquement, on cherchera à mettre en correspondance des images contenant au plus 1000 points d'intérêt, ce que notre méthode fait en 20ms pour une paire d'images.

Notre implémentation ANN utilise la librairie de référence ANN-Lib [163, 4]. Bien que plus efficace qu'une approche naïve sur CPU, les temps de cette version ANN se suivent pas exactement la complexité prévue. Cela peut être expliqué par le petit nombre de points traités par rapport à la haute dimension ($d = 64$). Ainsi, on observe, de la version ANN à notre version GPU, un gain en temps variant entre 11 et 30, suivant le nombre de points, notre méthode étant d'autant plus efficace qu'il y a de points. Pour une utilisation classique (environ 1000 points à apparier), notre méthode s'exécute en 15 fois moins de temps qu'une version ANN, qui elle prend 2 fois moins de temps qu'une version naïve.

En ce qui concerne la recherche des plus proches voisins, Garcia *et al.* [65] et nous [33] avons indépendemment eu la même idée. Néanmoins, les applications sont différentes : alors qu'ils cherchent à réaliser un tracking d'objets, notre objectif est d'apparier des images. Etant donné que les gains en temps sont très liés au matériel utilisé (CPU et GPU en particulier) ainsi qu'aux différentes bibliothèques d'implémentation (ANN,...), il serait intéressant de mener une comparaison et de comprendre si les différences de résultats s'expliquent par le choix du langage, du matériel, des bibliothèques,...

6.3 Notre application

Forts des temps de calcul rapides présentés dans la section précédente, il nous a paru envisageable de développer une première application d'aide à la prise de vue lors de la capture photographiques de larges scènes pour une reconstruction 3D du type [129]. Le procédé est décrit avec la figure 6.10.

Avec notre configuration portable, un ou plusieurs appareils photos Reflex communiquent via Wi-Fi avec un ordinateur portable (équipé d'un

GPU GeForce 8800M). Pour chaque nouvelle photo prise et transférée, l'application détecte et caractérise les points d'intérêt et stocke ces résultats dans la mémoire GPU, puis compare la nouvelle photo avec toutes les autres précédemment traitées, gardant en mémoire la liste des appariements filtrée par deux plus proches voisins et par RANSAC (voir section 6.2.1.3). Une heuristique supplémentaire de filtrage est alors appliquée par le CPU, basée sur l'hypothèse réaliste qu'un point apparié n'est pas isolé au milieu des points d'intérêts relevés : tous les points appariés qui n'ont pas dans leur voisinage un pourcentage supérieur à un seuil déterminé de points également appariés sont supprimés. Nous utilisons un tel seuil à 20%. Cette heuristique, simple à appliquer grâce à la librairie ANNLib [163], permet d'éliminer les appariements aberrants.

Une fois les différents filtrages des appariements effectués, un graphe de connexité entre images est créé. Deux images traitées y sont reliées lorsqu'elles partagent un nombre suffisant d'appariements. A la suite de cela, en affichant les points d'intérêts appariés et non appariés, il est possible de rapidement discerner les zones de la scène encore trop peu photographiés.

De plus, l'ensemble des calculs et la mise à jour des résultats sont effectués dynamiquement, l'application étant automatiquement prévenue lors d'une nouvelle prise de vue. Notre méthode d'appariement avec filtrage, s'exécutant typiquement en 25*ms* par paire d'image, assure un temps d'attente raisonnable entre deux prises de vues, permettant une interactivité confortable avec le système, et ceci même pour des images de grandes tailles (résolutions HD), comportant chacune jusqu'à plusieurs milliers de points d'intérêt.

Bien que notre application n'ait pas pour but la reconstruction 3D mais la capture photographique multiple d'une scène pour une reconstruction offline, un futur développement possible et envisagé est l'édification d'un système complet de reconstruction 3D, dont cette application constituerait justementle premier maillon. Les étapes manquantes seraient alors la calibration du jeu d'images, la détermination des positions 3D des points d'un maillage et la reprojection des textures. Ces étapes dépassant néanmoins le cadre de cette thèse.

Conclusion du chapitre

Nous avons, dans ce dernier chapitre, présenté différentes méthodes de détection et de caractérisation de points d'intérêts, principalement basées sur la description locale de points à haute dimensionnalité, telles les SIFT et les SURF. Ensuite, nous avons exposé le principe d'appariements de points d'intérêt avec différents moyens de filtrage des correspondances obtenues. Nous avons par la même occasion présenté notre algorithme de mise en correspondance de points d'intérêt par GPU de type *brute force*, qui s'avère plus efficace qu'une version utilisant des structures ANN, de complexité moindre mais s'exécutant sur CPU. Ainsi, nous pouvons apparier deux images, c'est-à-dire mettre en correspondance l'ensemble des points d'intérêt d'une image avec ceux d'une autre, en 20*ms* pour une moyenne de 1000 points par image.

Encouragés par ces résultats, nous avons développé une application apportant une aide à la prise de vue pour la capture photographique

dense de larges scènes, pouvant s'exécuter sur une plateforme mobile équipée d'un GPU. Pour une nouvelle photographie, cette application effectue tous les calculs liés aux détections et caractérisations de points d'intérêt, aux appariements et au filtrage des résultats, puis affiche, pour une image, l'ensemble des images qui représentent la même portion de la scène. Nous envisageons de rendre notre application disponible sur le web.

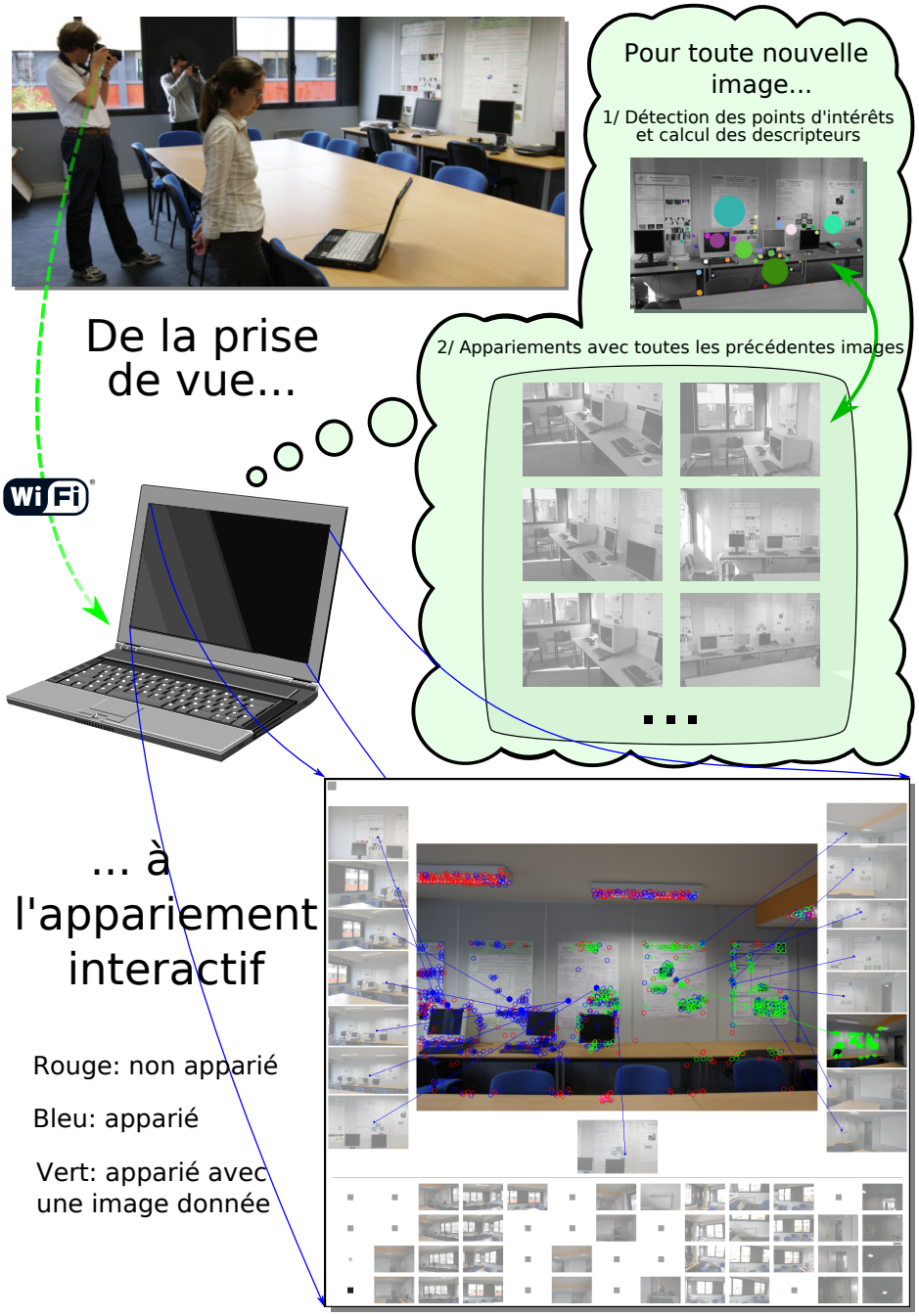


FIGURE 6.10 – *Description de notre application portable : aide à l'acquisition photographique dense de larges scènes. Plusieurs utilisateurs peuvent de concert photographier la scène d'intérêt. Un contrôle de l'avancée de l'acquisition est réalisé sur un ordinateur portable équipé d'un GPU GeForce 8800M. A chaque nouvelle prise de vue, (1) la photographie est transférée par Wi-Fi au module portable, (2) les points d'intérêts, détectés et caractérisés en espace et en échelle, sont conservés en mémoire GPU, (3) cette nouvelle image est appariée avec toutes les images précédemment traitées, suivant l'algorithme présenté dans la section 6.2.2 et en appliquant les filtres des deux plus proches voisins et du RANSAC, (4) un graphe de connexité entre les différentes images est créé (deux images sont reliées si elles partagent suffisamment de mises en correspondances de points d'intérêt) et (5) le résultat est montré dynamiquement sur l'écran du module portable, permettant de se déplacer dans le graphe de correspondance, les images autour de l'image courante étant ses voisines dans ce graphe, et indiquant pour la photo courante les points d'intérêts appariés au moins une fois dans ce graphe (rouge), appariés avec un point d'intérêt une image pointée (vert) et non appariés avec aucun point d'intérêt d'une autre image du graphe. Les zones à fortes concentration de points bleus sont donc celles pour lesquelles la prise de vue n'est pas encore assez dense.*

Synthèse

Introduction

Les travaux présentés dans ce manuscrit s'inscrivent dans l'évolution récente des capacités à la fois logicielles et matérielles des processeurs de cartes graphiques en tant qu'entités computationnelles généralistes (programmation GPGPU), demandant une adaptation continuelle de la part de l'utilisateur.

Il nous paraît intéressant de donner une lecture transverse des résultats obtenus, et plus particulièrement des moyens adoptés pour y parvenir, de synthétiser ces connaissances acquises, illustrées par des rappels des travaux précédemment exposés, pour en faire une base de réflexion possible à des travaux futurs, ou bien pour satisfaire la curiosité des intéressés.

Pour cela, une synthèse méthodologique est premièrement présentée, exposant les questions que l'on peut se poser avant de débuter un développement en GPGPU. Ensuite, une hypothétique réimplémentation de nos applications en CUDA, langage généraliste pour GPU, est évoquée, précisant les avantages ou inconvénients inhérents à ces applications.

Synthèse méthodologique : Penser GPGPU

Déterminer une méthodologie complète de développement GPGPU est un travail qui pourrait occuper une thèse entière. Nous avons préféré donner des indications plus synthétiques au lecteur voulant se lancer dans la programmation GPGPU, sous la forme d'une série organisée de questionnements importants à envisager avant même le début d'un développement. Nous avons classé ces questions en deux catégories, à laquelle nous rajoutons un rappel des avantages et inconvénients de l'utilisation généraliste des GPU.

Possibilité d'adaptation ou de formulation d'un nouvel algorithme sur GPU

- Les premières et principales interrogations face à un algorithme à développer sur GPU sont les suivantes : est-il parallélisable pour une implémentation GPU ? En d'autres termes, les calculs qu'il implique peuvent-ils être traités de façon indépendante, ou bien suffisamment indépendante pour ne pas trop pénaliser l'exécution globale ? Tous les processeurs effectueront-t-il les mêmes calculs sur des données différentes ? Si non, les divergences seront-elles suffisamment restreintes ? L'algorithme cherchera-t-il à propager des données ou à utiliser un aspect *scatter* ? Si oui, peut-on en imaginer une reformulation ?

- Dans le cas où l'algorithme est parallélisable, existe-il un mapping simple des données du problème vers un tableau 2D, structure principalement utilisée ? Les opérations ou routines que l'algorithme nécessite existent-elles sur GPU ? Si non, sont-elles assez facilement réimplémentables, de façon optimisée ? Les types de données dont l'algorithme a besoin existent-ils sur GPU (en particulier, a-t-on besoin de réels double précision), ou peuvent-ils être adaptés à des types GPU (en particulier les vecteurs) ?
- Lorsqu'une version optimisée d'un algorithme n'est pas adaptable sur GPU, doit-on pour autant s'arrêter à une version CPU algorithmiquement meilleure ? Suivant la taille du problème envisagé, il est possible qu'une version plus naïve sur GPU soit plus efficace au niveau des temps de calculs (c'est le cas pour notre application d'appariements d'image décrite dans le chapitre 6) ; cette éventualité est donc à ne pas négliger.
- Le temps d'apprentissage d'un des langages GPU est également à prendre en compte : pour une utilisation ponctuelle, le choix du GPU n'est peut être pas le plus pertinent.

Comment adapter sur GPU ?

La possibilité d'implémentation d'un algorithme sur GPU est parfois fonction de choix que l'on a à faire.

- Le premier de ces choix est celui du constructeur de la carte graphique. Le portage d'un programme d'une carte d'un constructeur à une carte d'un autre constructeur n'est pas nécessairement assuré et dépend du langage de programmation utilisé. Pour l'heure, c'est NVidia qui supporte la plus large gamme de langage, ayant de plus proposé un langage graphique reconnu, Cg, ainsi qu'un langage généraliste, CUDA, de plus en plus utilisé à des fins de recherche et développement. Une fois le choix du constructeur effectué, le choix du modèle est rapide, étant donné les prix très abordables de ces cartes. Un tout nouveau langage généraliste, OpenCL, a la particularité d'être indépendant du modèle de carte et du constructeur, rendant les choix précédents moins décisifs.
- Le choix du langage est des plus important et dépend des besoins de l'algorithme à implémenter, ainsi que des savoir-faire du développeur. Selon le langage choisi, on rencontrera certains avantages et difficultés. C'est en particulier le cas entre un shading language et un langage généraliste : un shading language impose de raisonner en terme de fragments et de processeurs CREW, mais libère de la contrainte de répartition des charges et permet l'utilisation d'astuces purement hardware liées au pipeline graphique. Au contraire, un langage généraliste permettra une flexibilité accrue de programmation, au prix d'une organisation mémorielle plus complexe.

Avantages et inconvénients d'un développement sur GPU

Comme nous l'avons vu au cours de ce document, l'emploi de GPU dans un contexte généraliste peut s'avérer très rentable en terme de vitesse de calcul. Néanmoins, il ne faut pas perdre de vue qu'il existe des

contreparties parfois sévères. Nous proposons ici un rappel des atouts et désavantages de l'emploi de ces GPU.

Avantages

- Actuellement, les cartes les plus puissantes de NVidia possèdent 240 processeurs travaillant en parallèle, pour une puissance brute de presque 1TFLOPS, soit 8 fois plus que les CPU quad core actuels ;
- La portabilité des algorithmes développés sur des GPU actuels vers des GPU futurs est assurée. Sans modifier le code, le gain (GPU comparé à CPU) s'amplifiera en changeant simplement de carte. Ceci est assuré par l'évolution plus rapide des GPU. Il est à noter que de nouveaux types de CPU pourraient venir modifier ce résultat (architectures *many core*, projet *Tera-Scale*), ainsi que des processeurs hybrides CPU/GPU (Larrabee d'Intel, par exemple) ;
- Il est possible d'associer facilement plusieurs GPU en cluster, démultipliant la puissance de calcul ;
- De génération en génération, la flexibilité de programmation s'accroît, les GPU tendent à devenir aussi facilement programmables que les CPU ;
- Prochainement, une plus grande transparence et compatibilité entre marques seront proposées, le projet OpenCL allant entièrement dans ce sens ;
- Un GPU haut de gamme coûte dans les 400€, mettant à la portée budgétaire de toute institution des machines équivalentes à des supercalculateurs.

Inconvénients

- L'impossibilité d'écriture arbitraire en mémoire (scatter non autorisé), lors de l'utilisation d'un shading language, peut se révéler gênante, nécessitant de réorganiser tout ou partie d'algorithme ;
- La rétrocompatibilité des programmes n'est présentement pas totale, ce qui peut poser certains problèmes lors d'une distribution ;
- Actuellement, la portabilité entre cartes de différents constructeurs n'est pas encore assurée. Il faudra probablement attendre les avancées d'OpenCL pour voir apparaître ces possibilités ;
- Les compilateurs de shader ou kernel sont encore obscurs car non livrés à la communauté. De légers artefacts peuvent parfois intervenir, sans raison apparente ;
- Un investissement initial est exigé pour la maîtrise des concepts GPU/GPGPU, pénalisant les cycles de développement courts ;
- Bien que la tendance disparaisse très vite, le GPU jouit encore, dans la communauté scientifique, d'une réputation d'outil peu sérieux.

Shading language ou langage généraliste ?

Comme nous l'avons vu au cours de ce manuscrit, l'évolution technique des cartes graphiques a été fulgurante, passant en cinq ans d'une puissance de quelques GFLOPS à aujourd'hui un millier. Pour exploiter ces capacités, les moyens logiciels ont dû s'adapter. En ce qui concerne

la manipulation des cartes NVidia, on est donc passé d'une simple configurabilité à une véritable programmation, tout d'abord grâce au *shading language* proposé, Cg [147] (avec de fortes contraintes à ses débuts, en particulier sur le nombre maximal d'instructions, puis sans cette restriction, et enfin avec l'ajout de nouveaux types entiers), dont la communauté scientifique s'est emparé pour effectuer des calculs généralistes, puis à un langage spécifique à ces calculs, CUDA [25, 168], que la communauté semble utiliser principalement aujourd'hui. De plus, dans l'avenir, il est probable que le successeur de CUDA (et autres langages généralistes) soit OpenCL [164, 234], ayant les deux importantes ambitions d'être *open source* et d'être multi-plateforme.

Les travaux effectués s'inscrivent dans les débuts de cette évolution, avec l'utilisation des différentes version de Cg, et ont commencé bien avant que CUDA ne soit disponible. Aujourd'hui, lorsqu'on souhaite commencer la programmation généraliste, on aurait tendance à se tourner directement vers CUDA, paraissant par sa définition plus adapté, jusqu'à rendre caduque l'utilisation des *shading language* dans ce cadre. Mais est-ce vraiment le cas ?

Supposons que nous souhaitions réimplémenter des applications Cg en CUDA. La première remarque est qu'il n'y a aucun changement matériel, on utilise bien le même GPU, même si le modèle de programmation est différent : il n'y a donc pas de puissance de calcul potentielle supplémentaire. Seul l'apport logiciel est à considérer pour CUDA.

Ensuite, une utilisation efficace et optimale de Cg est simple à obtenir, en adaptant correctement ses données à des textures 2D. Pour CUDA, le problème est plus difficile à résoudre : des pertes d'efficacités sont en effet à déplorer si l'on ne respecte pas l'architecture matérielle utilisée (en particulier les communications entre les différentes parties de l'organisation mémorielle, ainsi que la répartition de la charge sur les différents processeurs).

Envisageons maintenant une réimplémentation en CUDA des applications présentées au cours de ce manuscrit :

- Le cas de la simulation d'un réseau de neurones sans sondage, présenté dans la partie 4.2 est probablement le plus complexe. La communication entre neurones serait agréablement facilité par la possibilité d'écriture arbitraire, mais ceci au prix de problèmes de synchronisations entre threads, d'écritures multiples et de remaniement de l'organisation en mémoire (due à des mémoires partagées trop petites pour contenir le réseau de neurones en entier). Le gain d'une version CUDA est hypothétique, seule une réimplémentation permettrait de trancher ;
- La simulation d'un réseau de neurone avec sondage, présenté dans la partie 4.3 ne nécessite aucune des particularités apportées par CUDA. L'analyse du problème serait la même que pour une implémentation Cg, et les résultats identiques ;
- L'algorithme de stéréovision présenté dans le chapitre 5 n'est pas adapté à une implémentation CUDA car il nécessite un affichage 3D des résultats stockés dans des textures. On préfère donc ici Cg à CUDA. De plus, une réimplémentation avec les versions récentes de Cg serait bien plus concise et claire ;
- Comme pour la simulation de réseau de neurones par sondage, notre algorithme d'appariement d'images n'a pas besoin des parti-

cularités de CUDA. Ses données sont de plus très bien adaptées à celles manipulées par Cg. Une implémentation CUDA est possible et serait réalisée de façon similaire à Cg, n'apportant donc pas de réel gain de temps.

Ainsi, pour des personnes connaissant au préalable Cg, il est possible de réaliser avec un peu d'astuce presque tout type de calcul généraliste sans pour autant réapprendre un nouveau langage tel CUDA.

Conclusion

Dans cette courte synthèse, nous avons présenté quelques pistes de réflexion pour orienter tout nouveau développement GPU pour du calcul généraliste, sous la forme d'une série organisée de questionnements possibles, ainsi que d'un rappel des atouts et désagréments, matériels et logiciels, de l'utilisation de GPU pour de tels calculs. Ensuite, une discussion sur l'emploi de langages de programmation orientés graphique ou généraliste est exposée, s'appuyant sur l'hypothèse d'une réimplémentation des travaux présentés dans ce manuscrit à l'aide du langage CUDA.

Ceci pourrait servir à l'établissement d'une méthodologie complète d'implémentation GPGPU.

Conclusion générale

Récapitulatif des contributions

Au cours de cette thèse, nous nous sommes intéressés à l'utilisation générique des cartes graphiques (GPU) en tant que machine de calcul parallèle, en explicitant leur structure et les caractéristiques de leur programmation. Leur emploi croissant dans ce cadre a été exposé à travers un large panel d'application de domaines diversifiés.

Grâce à ces GPU, nous avons répondu aux besoins computationnels engendrés par différentes applications dans deux domaines de recherche liés aux thématiques du Certis.

Premièrement, nous avons proposé un guide de méthodes et astuces de programmation GPU/GPGPU par shading language, ainsi qu'une librairie GPU de calcul généraliste. Un état de l'art des applications sur GPU a également été présenté.

Dans un deuxième temps, nous avons accéléré la simulation de réseaux de neurones impulsionnels en définissant de nouveaux algorithmes adaptés à la structure des GPU et exploitant leurs capacités. Nous avons introduit la notion de simulation par sondage, permettant d'obtenir des facteurs de gain significatifs par rapport à une implémentation de référence sur CPU, ceci pour un faible taux d'erreur. De plus, il reste une large marge de progression pour ce gain avec l'utilisation de modèles de neurones plus complexes.

Dans un troisième temps, nous avons proposé une adaptation GPU d'un algorithme de reconstruction par stéréovision dense gérant les occlusions, basé sur des principes variationnels. Nous avons obtenu des gains en temps probants par rapport à une implémentation CPU optimisée, permettant d'atteindre une cadence vidéo.

En restant dans le même domaine computationnel, la vision par ordinateur, nous avons par la suite adapté sur GPU l'algorithme SURF puis conçu un algorithme d'appariements de points d'intérêt dédié à la mise en correspondance de larges ensembles d'images pouvant chacune contenir des milliers de points d'intérêt. A notre connaissance, c'est le premier algorithme GPU à réaliser cela. Par notre méthode, le temps moyen de mise en correspondance de deux telles images est de l'ordre de 20ms. Forts de ces résultats, nous avons proposé une application permettant d'aider un utilisateur désireux de capturer photographiquement une scène de façon dense, à des fins de reconstruction 3D, en lui indiquant interactivement les zones insuffisamment observées (trop peu d'appariements).

Ces travaux ont menés à différentes publications dans des conférences internationales :

– Alexandre Chariot, Renaud Keriven et Romain Brette : Simulation rapide de modèles de neurones impulsionnels sur carte gra-

phique. *In 1ère conférence francophone de neurosciences computationnelles*, Pont-à-Mousson, Oct. 2006 ;
– Julien Mairal, Renaud Keriven et Alexandre Chariot : Fast and efficient dense variational stereo on GPU. *In Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, Juin 2006 ;
– Alexandre Chariot et Renaud Keriven : GPU-boosted online image matching. *In 19th International Conference on Pattern Recognition*, Tampa, US, Déc. 2008

Perspectives

Dans le futur, les méthodes et applications développées au sein de cette thèse pourront être intégrées à des ensembles plus larges et des applications complètes.

La simulation GPU de réseaux de neurones proposée pourrait ainsi être intégrée à un simulateur de plus grande envergure, tel Mvaspike [191]. Par ailleurs, d'autres projets Odyssee utilisent à présent le GPU comme machine de calcul.

La détection de points d'intérêts avec l'adaptation GPU des SURF, ainsi que la méthode d'appariement d'images développée, sont utilisées par différents partenaires du projet ANR *Wired Smart*.

De plus, dans le cadre d'un projet de reconstruction 3D de bâtiments remarquables, le Certis se munit d'un ballon captif permettant de transporter, orienter et commander le déclenchement d'un appareil photo jusqu'à plusieurs dizaines de mètres de hauteur, photographiant en haute résolution.

Figure 6.11 – *Ballon captif. Gauche : arrimage de l'appareil photo au ballon. Droite : vue du ballon en vol. Issues de [183].*

Ce ballon est équipé d'un retour vidéo retransmettant en temps réel aux utilisateurs se trouvant au pied du ballon un flux, ce qui permet un contrôle de l'orientation des prises de vues. A ce point, notre application d'aide à la capture photographique de scènes sera utilisée pour être certain de disposer, pour la reconstruction 3D ultérieure, d'un ensemble assez dense de clichés.

Ce type de reconstruction est actuellement déjà réalisée au Certis, mais à partir de clichés non haute résolution, les prises de vues étant effectuées au sol ou en hélicoptère.

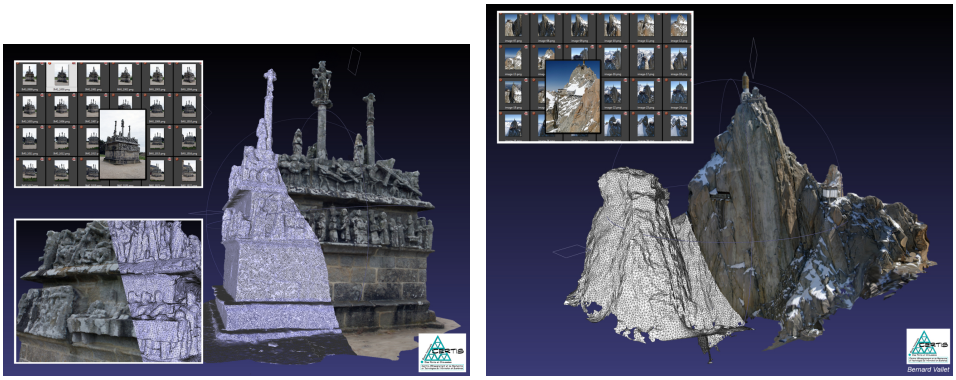


FIGURE 6.12 – *Résultats du* CERTIS, *reconstructions 3D. Gauche : calvaire de Tronoën. Droite : Aiguille du Midi,* ©*Bernard Vallet*

OUVERTURE

Dans les quelques prochaines années, il est indéniable que dans le monde computationnel, les GPU prendront de plus en plus d'importance, ou que leurs structures inspireront davantage les fabricants de CPU.

D'un côté, l'unification des processeurs (pour NVidia, à partir de l'architecture G80) et la mise à disposition de langages orienté GPGPU (CUDA, mais également OpenCL [164, 234, 165] tendant à en devenir un concurrent sérieux) facilitent amplement la tâche des développeurs, rendant de plus en plus transparent l'orientation graphique. Le GPU sera ainsi de plus en plus vu comme un réel coprocesseur "délocalisé", avec des capacités plus spécifiques mais une puissance accrue.

D'un autre côté, une des autres pistes explorées par les développeurs de processeurs centraux, forcés par la croissance des GPU, est la convergence de ceux-ci et des CPU, leur fusion en une seule unité de traitement. Parallèlement, Intel et AMD développent actuellement de telles architectures : Intel va proposer le Larrabee [131] et AMD le Fusion [2] (dont le développement a commencé à la suite du rachat d'ATI, fabriquant exclusivement des cartes graphiques), l'unité résultante serait un CPU disposant de plusieurs cœurs (pour le Larrabee) ou bien une architecture mêlant un processeur classique multi-cœurs et un processeur graphique dans une même puce (pour le Fusion). En procédant ainsi, la volonté d'Intel et ATI est de redéfinir un nouveau standard de pipeline graphique, inclus parmi les autres tâches du CPU et réparties sur l'ensemble de ses cœurs.

Bien que très prometteuses, les premières versions de cette nouvelle architecture qu'est le Larrabee, n'embarquant que 10 cœurs et sur laquelle plane une incertitude sur modèle de programmation et sa facilité d'utilisation, laissent pour le moment douter de la supériorité annoncée par Intel. NVidia annonce même lors de la *NVISION*'08, de façon cinglante, que l'architecture de ces processeurs s'apparenterait à des GPU datant de 2006, donc déjà dépassés.

Néanmoins, il est encore trop tôt pour se positionner clairement sur le succès de ce Larrabee ou d'autres architectures unifiant CPU et GPU, étant donné les changements qu'ils impliqueront, notamment dans les modèles de programmation (d'un type *top-to-bottom* vers un type flux), chose à laquelle les programmeurs sont en général assez réticents. Ce ne sera probablement que dans quelques années que ces changements

architecturaux importants pourront potentiellement devenir de nouveaux standards.

Bibliographie

[1] V. Agarwal, L.K. Liu et D.A. Bader : Financial modeling on the cell broadband engine. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008. (Cité page 62.)

[2] AMD : The Industry-Changing Impact of Accelerated Computing. Whitepaper, 2008. (Cité page 165.)

[3] M.A. Arbib : *The Handbook of Brain Theory and Neural Networks*. Bradford Books, 2003. (Cité pages 76 et 91.)

[4] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman et A.Y. Wu : An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998. (Cité pages 148 et 153.)

[5] Franz Aurenhammer : Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991. (Cité page 69.)

[6] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo et E.S. Quintana-Orti : Solving dense linear systems on graphics processors. Rapport technique, Technical Report ICC 02-02-2008, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, February 2008. (Cité page 56.)

[7] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo et Enrique S. Quintana-Ort´ : GLAME@lab : An M-script API for Linear Algebra Operations on Graphics Processors, February 2008. (Cité page 56.)

[8] K.E. Batcher *et al.* : Sorting networks and their applications. *Proceedings of the AFIPS Spring Joint Computer Conference*, 32:307–314, 1968. (Cité page 57.)

[9] J. Bauer, N. Sunderhauf et P. Protzel : Comparing several implementations of two recently published feature detectors. *International Conference on Intelligent and Autonomous Systems*, 2007. (Cité page 142.)

[10] H. Bay, T. Tuytelaars et L. Van Gool : SURF : Speeded Up Robust Features. *Lecture notes in computer science*, 3951:404, 2006. (Cité pages 142, 148 et 150.)

[11] F. Bernhard et R. Keriven : Spiking neurons on gpus. *In International Conference on Computational Science. Workshop General purpose computation on graphics hardware (GPGPU) : Methods, algorithms and applications*, Readings, UK, May 2006. (Cité page 95.)

[12] C.M. Bishop : *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1995. (Cité pages 85 et 91.)

[13] Kevin Bjorke : Color controls. *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 363–373, 2004. (Cité page 68.)

[14] Guy E. Blelloch : *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990. (Cité pages 42 et 43.)

[15] Jeff Bolz, Ian Farmer, Eitan Grinspun et Peter Schröder : Sparse matrix solvers on the gpu : conjugate gradients and multigrid. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 171, New York, NY, USA, 2005. ACM. (Cité pages 55, 57, 60 et 61.)

[16] Y. Boniface, F. Alexandre et S. Vialle : A bridge between two paradigms for parallelism : neural networks andgeneral purpose MIMD computers. *In Neural Networks, 1999. IJCNN'99. International Joint Conference on*, volume 4, 1999. (Cité page 95.)

[17] Y. Boniface, F. Alexandre et S. Vialle : A Library to Implement Neural Networks on MIMD Machines. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 935–938, 1999. (Cité page 95.)

[18] Tamy Boubekeur et Christophe Schlick : Generic adaptive mesh refinement. *GPU Gems 3*, pages 93–104, 2007. (Cité page 71.)

[19] G.E.P. Box et M.E. Muller : A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958. (Cité page 103.)

[20] Jaromír Brambor : *Algorithmes de la morphologie mathématique pour les architectures orientées flux*. Thèse de doctorat, ENSMP - CMM Centre de Morphologie Mathématique, Juillet 2006. (Cité page 67.)

[21] Romain Brette : Modèles Impulsionnels de Réseaux de Neurones Biologiques. *These de doctorat en Neurosceinces computationneles. Universite Pierre et Marie Curie-Paris VI*, 2003. (Cité pages 76, 82, 85 et 89.)

[22] N. Brunel et P.E. Latham : Firing Rate of the Noisy Quadratic Integrate-and-Fire Neuron. *Neural Computation*, 15(10):2281–2306, 2003. (Cité page 90.)

[23] A. Brunton, C. Shu et G. Roth : Belief Propagation on the GPU for Stereo Vision. *The 3rd Canadian Conference on Computer Robot Vision*, pages 76–82, 2006. (Cité page 72.)

[24] I. Buck, K. Fatahalian et P. Hanrahan : GPUBench : Evaluating GPU performance for numerical and scientific applications. *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004. (Cité page 63.)

[25] Ian Buck : GPU computing with NVidia CUDA. *In SIGGRAPH '07 : ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM. (Cité pages 26 et 160.)

[26] Ian Buck et Tim Purcell : A toolkit for computing on gpus. *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 621–636, 2004. (Cité pages 57 et 58.)

[27] J.M. Buhmann, T. Lange et U. Ramacher : Image Segmentation by Networks of Spiking Neurons, 2005. (Cité page 95.)

[28] S.R. Campbell, D.L.L. Wang et C. Jayaprakash : Synchrony and Desynchrony in Integrate-and-Fire Oscillators, 1999. (Cité page 95.)

[29] J. Canny : A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986. (Cité page 135.)

[30] N. Carr, J. Hall et J. Hart : Gpu algorithms for radiosity and subsurface scattering, 2003. (Cité page 66.)

[31] Nathan A. Carr, Jared Hoberock, Keenan Crane et John C. Hart : Fast gpu ray tracing of dynamic meshes using geometry images. *In GI '06 : Proceedings of Graphics Interface 2006*, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society. (Cité page 64.)

[32] Certis : Certislib. Bibliothèque informatique, http://certis.enpc.fr/~keriven/CertisLibs/. (Cité page 47.)

[33] Alexandre Chariot et Renaud Keriven : GPU-boosted online image matching. *In 19th International Conference on Pattern Recognition*, Tampa, US, Dec 2008. (Cité pages 7, 134 et 153.)

[34] Alexandre Chariot, Renaud Keriven et Romain Brette : Simulation rapide de modèles de neurones impulsionnels sur carte graphique. *In 1ère conférence francophone de neurosciences computationnelles*, Pont-à-Mousson, Oct 2006. (Cité pages 6 et 94.)

[35] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace et Donald P. Greenberg : A progressive refinement approach to fast radiosity image generation. *SIGGRAPH Comput. Graph.*, 22(4):75–84, 1988. (Cité page 66.)

[36] P. Colantoni, N. Boukala et J. Da Rugna : Fast and accurate color image processing using 3d graphics cards. *Proceedings Vision, Modeling and Visualization 2003*, 133, 2003. (Cité page 67.)

[37] Greg Coombe et Mark J. Harris : Global illumination using progressive refinement radiosity. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 635–647, 2005. (Cité page 66.)

[38] Greg Coombe, Mark J. Harris et Anselmo Lastra : Radiosity on graphics hardware. *Proceedings of the 2004 conference on Graphics interface*, pages 161–168, 2004. (Cité page 66.)

[39] Creative Technology : Site internet d'OpenAL. http://www.openal.org/. (Cité page 26.)

[40] Alex D'Angelo : Hello, Cg ! Introductory Tutorial. Whitepaper, 2004. (Cité page 34.)

[41] Peter Dayan et Laurence F. Abbott : *Theoretical Neuroscience*. MIT Press, 1999. (Cité pages 76, 87 et 88.)

[42] P. Debevec : HDRI and image-based lighting. *SIGGRAPH'03, Course Notes 19*, 2003. (Cité page 69.)

[43] R. Deriche : Using Canny's criteria to derive a recursively implemented optimal edge detector. *International Journal of Computer Vision*, 1(2):167–187, 1987. (Cité page 135.)

[44] Alain Destexhe : Conductance-based integrate-and-fire models. *Neural Comput.*, 9(3):503–514, 1997. (Cité page 89.)

[45] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung et Hunter Scales : Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000. (Cité page 15.)

[46] Ádám Moravánszky : Dense Matrix Algebra on the GPU. *ShaderX2 : Shader Programming Tips and Tricks with DirectX*, 9:352–380, 2003. (Cité pages 55 et 95.)

[47] J. N. England : A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12(3):336–340, 1978. (Cité page 53.)

[48] Cass Everitt et Mark J. Kilgard : Practical and robust stenciled shadow volumes for hardware-accelerated rendering, 2003. http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0301002. (Cité page 31.)

[49] K. Fatahalian, J. Sugerman et P. Hanrahan : Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. *In HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM. (Cité page 56.)

[50] Olivier Faugeras : *Three-Dimensional Computer Vision : A Geometric Viewpoint*. Mit Press, 1993. (Cité page 122.)

[51] Olivier Faugeras et Renaud Keriven : Variational principles, surface evolution, PDE's, level set methods and the stereo problem. *Biomedical Imaging, 2002. 5th IEEE EMBS International Summer School on*, 2002. (Cité page 118.)

[52] Randima Fernando et Mark J. Kilgard : *The Cg Tutorial : The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cité pages 18, 25, 33 et 51.)

[53] I. Fischer et C. Gotsman : Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools*, 11(4):39–60, 2006. (Cité pages 70 et 71.)

[54] M.A. Fischler et R.C. Bolles : Random sample consensus : a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981. (Cité page 149.)

[55] R. FitzHugh : Impulses and Physiological States in Theoretical Models of Nerve Membrane. *Biophysical Journal*, 1(6):445, 1961. (Cité page 84.)

[56] Michael J. Flynn : Some computer organizations and their effectiveness. *IEEE Trans. Comput*, C-21(9):948–960, September 1972. (Cité page 15.)

[57] Tim Foley et Jeremy Sugerman : Kd-tree acceleration structures for a gpu raytracer. *In HWWS '05 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM. (Cité page 64.)

[58] Yaniv Frishman et Ayellet Tal : Multi-Level Graph Layout on the GPU. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, pages 1310–1317, 2007. (Cité page 58.)

[59] Yaniv Frishman et Ayellet Tal : Online dynamic graph drawing. *Proc. Eurographics/IEEE VGTC Symp. on Visualization (EuroVis' 07)*, 2007. (Cité page 58.)

[60] Nicolas Fritz, Philipp Lucas et Philipp Slusallek : CGiS, a new Language for Data-Parallel GPU Programming. *In* Bernd Girod, Hans-Peter Seidel et Marcus Magnor, éditeurs : *Proceedings of the 9th International Workshop "Vision, Modeling, and Visualization" (VMV'04)*, pages 241–248, 2004. (Cité page 26.)

[61] James Fung : Computer vision on the GPU. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 649–666, 2005. (Cité pages 69 et 70.)

[62] James Fung et Steve Mann : Openvidia : parallel gpu computer vision. *In MULTIMEDIA '05 : Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM. (Cité page 69.)

[63] N. Galoppo, NK Govindaraju, M. Henson et D. Manocha : LU-GPU : Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 3–3, 2005. (Cité page 55.)

[64] V. Galtier : Expérimentation et performances de picsougrid sur grid'5000. Invited Talk for ANR GCPMF, 2006. (Cité page 5.)

[65] V. Garcia, E. Debreuve et M. Barlaud : Fast k nearest neighbor search using gpu. *In CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA, June 2008. (Cité page 153.)

[66] W. Gerstner : A Framework for Spiking Neuron Models : The Spike Response Model. *Neuro-informatics and Neural Modelling*, 2001. (Cité page 90.)

[67] W. Gerstner et W.M. Kistler : *Spiking Neuron Models : Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. (Cité page 91.)

[68] I. Geys, T.P. Koninckx et L. Van Gool : Fast Interpolated Cameras by Combining a GPU based Plane Sweep with a Max-Flow Regularisation Algorithm. *International Symposium on 3D Data Processing, Visualization and Transmission*, pages 534–541, 2004. (Cité page 118.)

[69] Andrew S. Glassner, éditeur. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989. (Cité page 64.)

[70] Michael Gold : OpenGL and CUDA. NVISION 2008 presentation, 2008. http://developer.nvidia.com/object/nvision08-opengl4.html. (Cité page 46.)

[71] M. Gong et Y.H. Yang : Near real-time reliable stereo matching using programmable graphics hardware. *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, 1, 2005. (Cité page 118.)

[72] Nolan Goodnight, Rui Wang, Cliff Woolley et Greg Humphreys : Interactive time-dependent tone mapping using programmable graphics hardware. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 180, New York, NY, USA, 2005. ACM. (Cité page 69.)

[73] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke et Greg Humphreys : A multigrid solver for boundary value problems using programmable graphics hardware. *In HWWS '03 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on*

Graphics hardware, pages 102–111, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité pages 55, 57 et 60.)

[74] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg et Bennett Battaile : Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, 1984. (Cité page 66.)

[75] Naga K. Govindaraju, Scott Larsen, Jim Gray et Dinesh Manocha : A memory model for scientific algorithms on graphics processors. *In SC'06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM. (Cité page 63.)

[76] GPGPU.org : Site internet. http://www.gpgpu.org. (Cité page 54.)

[77] A. Greb et G. Zachmann : Gpu-abisort : optimal parallel sorting on stream architectures. *ipdps*, 0:27, 2006. (Cité page 57.)

[78] S. Green : Image processing tricks in OpenGL. *Game Developers Conference*, 2005. (Cité page 63.)

[79] Simon Green : Cloth sample, NVidia SDK sample, 2003. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html. (Cité page 60.)

[80] Simon Green : Particle system, NVidia SDK sample, 2004. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html. (Cité page 60.)

[81] Simon Green : Particle-based fluid simulation for games, February 2008. Game Developers Conference 2008 Presentation. (Cité page 62.)

[82] Stanford University Graphics Group : BrookGPU. http://graphics.stanford.edu/projects/brookgpu/. (Cité page 26.)

[83] T.R. Hagen, K.A. Lie et J.R. Natvig : Solving the Euler Equations on Graphics Processing Units. *Computational Science–ICCS 2006*, 3994: 220–227, 2006. (Cité page 61.)

[84] D. Hansel et G. Mato : Existence and stability of persistent states in large neuronal networks. *Phys. Rev. Lett.*, 86(18):4175–4178, Apr 2001. (Cité page 90.)

[85] Pawan Harish et P.J. Narayanan : Accelerating large graph algorithms on the GPU using CUDA. *In* Springer Berlin / Heidelberg, éditeur : *High Performance Computing - HiPC 2007*, volume 4873/2007, pages 197–208. Springer Berlin, 2007. (Cité page 58.)

[86] C. Harris et M. Stephens : A combined corner and edge detector. *Alvey Vision Conference*, 15:50, 1988. (Cité pages 135, 136 et 137.)

[87] Mark J. Harris : Fast fluid dynamics simulation on the gpu. *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 637–665, 2004. (Cité page 60.)

[88] Mark J. Harris : Fluid code sample, NVidia SDK sample, 2005. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html. (Cité page 61.)

[89] Mark J. Harris : Mapping computational concepts to GPUs. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM. (Cité page 28.)

[90] Mark J. Harris : *Optimizing Parallel Reduction in CUDA*, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html. (Cité page 45.)

[91] Mark J. Harris, William V. Baxter, Thorsten Scheuermann et Anselmo Lastra : Simulation of cloud dynamics on graphics hardware. *In HWWS '03 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité pages 57, 60 et 61.)

[92] Mark J. Harris, Greg Coombe, Thorsten Scheuermann et Anselmo Lastra : Physically-based visual simulation on graphics hardware. *In HWWS '02 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. (Cité page 59.)

[93] Mark J. Harris, Shubhabrata Sengupta et John D. Owens : Parallel Prefix Sum (Scan) with CUDA. *In* Hubert Nguyen, éditeur : *GPU Gems 3*, chapitre 39. Addison Wesley, août 2007. (Cité page 45.)

[94] R. Hartley et A. Zisserman : *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003. (Cité page 133.)

[95] S. Haykin : *Neural Networks : A Comprehensive Foundation*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1998. (Cité pages 91 et 95.)

[96] Kyle Hegeman, Nathan A. Carr et Gavin S.P. Miller : Particle-based fluid simulation on the gpu. *In* Peter M.A. Sloot Vassil N. Alexandrov, Geert Dick van Albada et Jack Dongarra, éditeurs : *Computational Science – ICCS 2006*, volume 3994 de *LNCS*, pages 228–235. Springer, 2006. (Cité page 62.)

[97] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh et Anselmo Lastra : Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24:547–555(9), September 2005. (Cité page 45.)

[98] S. Heymann, K. Maller, A. Smolic, B. Froehlich et T. Wiegand : SIFT implementation and optimization for general-purpose GPU. *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2007. (Cité pages 144 et 147.)

[99] W. Daniel Hillis et Jr. Guy L. Steele : Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986. (Cité pages 43 et 45.)

[100] W.D. Hillis et G.L. Steele Jr : Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. (Cité page 41.)

[101] J.L. Hindmarsh et R.M. Rose : A model of neuronal bursting using three coupled first order differential equations. *Proceedings of the Royal society of London. Series B. Biological sciences*, 221(1222):87–102, 1984. (Cité page 84.)

[102] A.L. Hodgkin et A.F. Huxley : A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117(4):500–544, 1952. (Cité page 83.)

[103] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha et Tim Culver : Fast computation of generalized Voronoi diagrams using graphics hardware. *In SIGGRAPH '99 : Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. (Cité pages 69 et 70.)

[104] Daniel Horn : Stream Reduction Operations for GPGPU Applications. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 573–589, 2005. (Cité pages 43 et 45.)

[105] Daniel Horn : libgpufft, 2006. http://sourceforge.net/projects/gpufft/. (Cité page 63.)

[106] L. Howes et D. Thomas : Efficient random number generation and application using CUDA. *GPU Gems 3*, pages 805–830, 2007. (Cité pages 62 et 102.)

[107] RapidMind Inc. : RapidMind. http://www.rapidmind.net/. (Cité page 25.)

[108] Kei Iwasaki, Moro Yoshitaka, Dobashi Yoshinori et Nishita Tomoyuki : Particle-Based Fluid Simulation on the GPU. *IPSJ SIG Technical Reports*, pages 121–126, 2005. (Cité page 62.)

[109] Greg James : Game of life, NVidia SDK sample, 2001. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html. (Cité page 59.)

[110] Greg James : Real-time glow. *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 343–362, 2004. (Cité page 68.)

[111] Frank Jargstorff : A framework for image processing. *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 445–467, 2004. (Cité page 68.)

[112] Henrik Wann Jensen : Global illumination using photon maps. *In Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag. (Cité page 65.)

[113] R. Jolivet, T.J. Lewis et W. Gerstner : The Spike Response model : A framework to predict neuronal Spike trains. *Lecture notes in computer science*, pages 846–853, 2003. (Cité page 90.)

[114] James T. Kajiya : The rendering equation. *In SIGGRAPH '86 : Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM. (Cité page 66.)

[115] M. Kass, A. Lefohn et J. Owens : Interactive Depth of Field Using Simulated Diffusion on a GPU. Rapport technique, Pixar Animation Studios, 2006. (Cité pages 56 et 64.)

[116] Benoit Kaufman : *Spécification et Conception d'un système autostéréoscopique multi-vues pour l'affichage tri-dimensionnel*. Thèse de doctorat, Université de Paris-Est Marne-la-Vallée, september 2006. (Cité page 72.)

[117] Ignacio Llamas Keenan Crane et Sarah Tariq : Real time simulation and rendering of 3d fluids. *GPU Gems 3*, pages 633–675, 2007. (Cité page 62.)

[118] Theodore Kim et Ming C. Lin : Visual simulation of ice crystal growth. *In SCA '03 : Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 86–97, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité page 59.)

[119] Peter Kipfer, Mark Segal et Rüdiger Westermann : Uberflow : a gpu-based particle engine. *In HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM. (Cité page 60.)

[120] Peter Kipfer et Rüdiger Westermann : Improved GPU sorting. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746, 2005. (Cité page 57.)

[121] D.E. Knuth : The art of computer programming. Vol. 2 : Seminumerical algorithms. *Reading*, 1981. (Cité pages 102 et 103.)

[122] C. Koch et I. Segev : The role of single neurons in information processing. *NATURE NEUROSCIENCE*, 3:1171–1177, 2000. (Cité page 79.)

[123] A. Kolb, L. Latta et C. Rezk-Salama : Hardware-based simulation and collision detection for large particle systems. *In HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM. (Cité page 60.)

[124] Andreas Kolb et Nicolas Cuntz : Dynamic particle coupling for gpu-based fluid simulation. *Proc. 18th Symposium on Simulation Technique*, pages 722–727, 2005. (Cité page 62.)

[125] Craig Kolb et Matt Phar : Option Pricing on the GPU. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 719–731, 2005. (Cité page 62.)

[126] Jens Krüger, Peter Kipfer, Polina Kondratieva et Rudiger Westermann : A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005. (Cité pages 60, 61 et 62.)

[127] Jens Krüger et Rüdiger Westermann : Linear algebra operators for gpu implementation of numerical algorithms. *In SIGGRAPH '03 : ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM. (Cité pages 55, 57, 60 et 61.)

[128] P. Labatut, R. Keriven et J.P. Pons : A GPU Implementation of Level Set Multiview Stereo. *International Conference on Computational Science (4) pp*, pages 212–219, 2006. (Cité pages 67, 68, 71 et 118.)

[129] P. Labatut, J.-P. Pons et R. Keriven : Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts. *In IEEE International Conference on Computer Vision*, Rio de Janeiro, Brazil, Oct 2007. (Cité page 153.)

[130] L. Lapicque : Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen*, 9:620–635, 1907. (Cité page 86.)

[131] Larry Seiler and Doug Carmean and Eric Sprangle and Tom Forsyth and Michael Abrash and Pradeep Dubey and Stephen Junkins and Adam Lake and Jeremy Sugerman and Robert Cavin and Roger Espasa and Ed Grochowski and Toni Juan and Pat Hanrahan : Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008. (Cité page 165.)

[132] B.D. Larsen et N.J. Christensen : Simulating photon mapping for real-time applications. *Rendering Techniques*, pages 123–132, 2004. (Cité page 65.)

[133] E. Scott Larsen et David McAllister : Fast matrix multiplies using graphics hardware. *In Supercomputing '01 : Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM. (Cité page 55.)

[134] A. E. Lefohn et R. T. Whitaker : A GPU-based, three-dimensional level set solver with curvature flow. *University of Utah tech report UUCS-02-017, December*, 2002. (Cité pages 57, 67 et 68.)

[135] Aaron E. Lefohn : A dynamic adaptive multi-resolution GPU data structure : adaptive shadow maps, octree 3D paint, adaptive PDE solver. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 165, New York, NY, USA, 2005. ACM. (Cité page 57.)

[136] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen et Ross T. Whitaker : A streaming narrow-band algorithm : interactive computation and visualization of level sets. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 243, New York, NY, USA, 2005. ACM. (Cité pages 57 et 67.)

[137] M. Lhuillier et L. Quan : A quasi-dense approach to surface reconstruction from uncalibrated images. Site internet, 2005. http://www.cs.ust.hk/~quan/WebPami/pami.html. (Cité page 131.)

[138] Wei Li, Zhe Fan, Xiaoming Wei et Arie Kaufman : GPU-Based Flow Simulation with Complex Boundaries. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 747–764, 2005. (Cité page 59.)

[139] Wei Li, Xiaoming Wei et Arie Kaufman : Implementing lattice boltzmann computation on graphics hardware, 2003. (Cité page 59.)

[140] T. Lindeberg : Edge Detection and Ridge Detection with Automatic Scale Selection. *International Journal of Computer Vision*, 30(2):117–154, 1998. (Cité page 135.)

[141] T. Lindeberg : Feature Detection with Automatic Scale Selection. *International Journal of Computer Vision*, 30(2):79–116, 1998. (Cité page 138.)

[142] Youquan Liu, Xuehui Liu et Enhua Wu : Real-time 3d fluid simulation on GPU with complex obstacles. *In PG '04 : Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pages 247–256, Washington, DC, USA, 2004. IEEE Computer Society. (Cité page 61.)

[143] D.G. Lowe : Object recognition from local scale-invariant features. *International Conference on Computer Vision*, 2:1150–1157, 1999. (Cité pages 135, 139, 143 et 144.)

[144] D.G. Lowe : Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. (Cité pages 140, 141, 145 et 148.)

[145] D.J.C. MacKay : *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003. (Cité page 85.)

[146] Julien Mairal, Renaud Keriven et Alexandre Chariot : Fast and efficient dense variational stereo on GPU. *Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, pages 97–104, 2006. (Cité pages 7, 72 et 115.)

[147] William R. Mark, R. Steven Glanville, Kurt Akeley et Mark J. Kilgard : Cg : A system for programming graphics hardware in a C-like language. *In SIGGRAPH '03 : ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM. (Cité pages 25 et 160.)

[148] D. Marr et W.H. Freeman : Vision. *San Francisco*, pages 116–124, 1982. (Cité page 117.)

[149] M. Mascagni : Random Number Generation for serial, parallel, distributed, and Grid-based financial computations. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–1, 2008. (Cité page 62.)

[150] J. Matas, O. Chum, M. Urban et T. Pajdla : Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, 2004. (Cité page 135.)

[151] Michael McCool et Stefanus Du Toit : *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004. (Cité page 25.)

[152] Patrick S. McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth et Sharen Cummins : Scout : a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11):648–662, 2007. (Cité page 26.)

[153] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen et Greg Roth : Scout : A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. *In VIS '04 : Proceedings of the conference on Visualization '04*, pages 171–178, Washington, DC, USA, 2004. IEEE Computer Society. (Cité page 26.)

[154] Microsoft : Introduction au langage HLSL. http://msdn.microsoft.com/fr-fr/library/ms810449.aspx. (Cité page 25.)

[155] K. Mikolajczyk et J. Matas : Improving Descriptors for Fast Tree Matching by Optimal Linear Projection. *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, 2007. (Cité page 148.)

[156] K. Mikolajczyk et C. Schmid : An Affine Invariant Interest Point Detector. *Lecture notes un Computer Science*, pages 128–142, 2002. (Cité pages 135, 137 et 139.)

[157] J.L. Mitchell, M.Y. Ansari et E. Hart : Advanced Image Processing with DirectX®9 Pixel Shaders. *ShaderX*, 2004. (Cité page 63.)

[158] Steven Molnar, John Eyles et John Poulton : PixelFlow : high-speed rendering using image composition. *In SIGGRAPH '92 : Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240, New York, NY, USA, 1992. ACM. (Cité page 53.)

[159] G.E. Moore : Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. (Cité page 18.)

[160] Hans P. Moravec : Towards automatic visual obstacle avoidance. *In Proceedings of the 5th International Joint Conference on Artificial Intelligence*, page 584, August 1977. (Cité pages 135 et 136.)

[161] Kenneth Moreland et Edward Angel : The FFT on a GPU. *In HWWS'03 : Proceedings of the ACM SIGGRAPH EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité page 63.)

[162] C. Morris et H. Lecar : Voltage oscillations in the barnacle giant muscle fiber. *Biophysical Journal*, 35(1):193–213, 1981. (Cité page 83.)

[163] D.M. Mount et S. Arya : ANN : A library for approximate nearest neighbor searching. *In CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997. http://www.cs.umd.edu/~mount/ANN/. (Cité pages 153 et 154.)

[164] Aaftab Munshi : OpenCL Parallele Computing on the GPU. Présentation à SIGGRAPH2008, 2008. (Cité pages 4, 26, 160 et 165.)

[165] Aaftab Munshi : The OpenCL Specification. Documentation, december 2008. (Cité pages 4, 26 et 165.)

[166] Frank Nielsen : An interactive tour of voronoi diagrams on the gpu. *In ShaderX^6 : Advanced Rendering Techniques*. Charles River Media (http://www.charlesriver.com/), 2008. (Cité page 70.)

[167] J. Novosad : Advanced high-quality filtering. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 417–435, 2005. (Cité page 68.)

[168] NVidia : CUDA Zone – Site internet. http://www.openal.org/. (Cité pages 26 et 160.)

[169] NVidia : Site internet. http://www.nvidia.fr. (Cité page 47.)

[170] NVidia : Perfect reflections and specular lighting effects with cube environment mapping. Rapport technique, NVidia Corporation, August 1999. Whitepaper. (Cité page 51.)

[171] NVidia : Cublas library. Rapport technique, NVidia Corporation, January 2007. Whitepaper. Part of CUDA Toolkit. (Cité page 56.)

[172] NVidia : *CUDA Programming Guide, version 2.0*, 2008. http://www.nvidia.com/object/cuda_home.html. (Cité pages 3, 13, 14 et 41.)

[173] L. Nyland, M.J. Harris et J. Prins : N-body simulations on a GPU. *Proceedings of the ACM Workshop on General-Purpose Computation on Graphics Processors*, 2004. (Cité page 60.)

[174] L. Nyland, M.J. Harris et J. Prins : Fast n-body simulation with CUDA. *GPU Gems 3*, pages 677–695, 2007. (Cité page 60.)

[175] Stuart Oberman, Greg Favor et Fred Weber : AMD 3DNow ! Technology : Architecture and Implementations. *IEEE Micro*, 19(2):37–48, 1999. (Cité page 15.)

[176] Kyoung-Su Oh et Keechul Jung : GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004. (Cité page 95.)

[177] Marc Olano et Anselmo Lastra : A shading language on graphics hardware : the PixelFlow shading system. *In SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168, New York, NY, USA, 1998. ACM. (Cité page 53.)

[178] Michael Oneppo : HLSL Shader Model 4.0. *In SIGGRAPH '07 : ACM SIGGRAPH 2007 courses*, pages 112–152, New York, NY, USA, 2007. ACM. (Cité page 25.)

[179] J.D. Owens, S. Sengupta et D. Horn : Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications. Rapport technique, Technical Report ECE-CE-2005-3, Department of Electrical and Computer Engineering, University of California, Davis, October 20005. (Cité page 63.)

[180] John D. Owens, David Luebke, Naga Govindaraju, Mark J. Harris, Jens Krüger, Aaron E. Lefohn et Timothy J. Purcell : A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. (Cité pages 29 et 54.)

[181] R. Peikert et C. Sigg : Optimized bounding polyhedra for gpu-based distance transform. *Scientific Visualization : The visual extraction of knowledge from data*, 2005. (Cité page 71.)

[182] Alex Peleg et Uri Weiser : MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996. (Cité page 15.)

[183] Phodia : Site internet. http://www.phodia.com. (Cité page 164.)

[184] Victor Podlozhnyuk : Black-scholes option pricing. Rapport technique, NVidia Corporation, June 2007. Whitepaper. Part of CUDA SDK. (Cité page 62.)

[185] O.A. Potiy et A.A. Anikanov : 3D Flow visualization using GPU-driven particle system. *GraphiCon 2005 Proceedings*, 2005. (Cité page 60.)

[186] Michael Potmesil et Eric M. Hoffert : The pixel machine : a parallel image computer. *SIGGRAPH Comput. Graph.*, 23(3):69–78, 1989. (Cité page 53.)

[187] Timothy J. Purcell : Ray tracing on a stream processor. Mémoire de D.E.A., Stanford University, 2004. Adviser-Patrick M. Hanrahan. (Cité page 64.)

[188] Timothy J. Purcell, Ian Buck, William R. Mark et Pat Hanrahan : Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002. (Cité page 64.)

[189] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen et Pat Hanrahan : Photon mapping on programmable graphics hardware. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Courses*, page 258, New York, NY, USA, 2005. ACM. (Cité page 65.)

[190] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann et Amitabh Varshney : Real-time procedural textures. *In SI3D '92 : Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 95–100, New York, NY, USA, 1992. ACM. (Cité page 53.)

[191] O. Rochel et D. Martinez : An event-driven framework for the simulation of networks of spiking neurons. *In Proc. 11th European Symposium on Artificial Neural Networks*, pages 295–300, 2003. (Cité page 164.)

[192] Guodong Rong et Tiow-Seng Tan : Jump flooding in gpu with applications to voronoi diagram and distance transform. *In I3D '06 : Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, New York, NY, USA, 2006. ACM. (Cité pages 70 et 71.)

[193] Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao et Stephanus : Computing two-dimensional delaunay triangulation using graphics hardware. *In SI3D '08 : Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97, New York, NY, USA, 2008. ACM. (Cité page 70.)

[194] F. Rosenblatt : The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. (Cité page 91.)

[195] Randi J. Rost : *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005. (Cité page 25.)

[196] Martin Rumpf et Robert Strzodka : Level set segmentation in graphics hardware. *In Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, volume 3, pages 1103–1106, 2001. (Cité pages 57 et 67.)

[197] Martin Rumpf et Robert Strzodka : Nonlinear diffusion in graphics hardware. *In Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '01)*, pages 75–84, 2001. (Cité page 57.)

[198] Martin Rumpf et Robert Strzodka : Graphics processor units : New prospects for parallel computing. *In* Are Magnus Bruaset et Aslak Tveito, éditeurs : *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 de *Lecture Notes in Computational Science and Engineering*, pages 89–134. Springer, 2005. (Cité page 57.)

[199] Michael Rush : *Les nouveaux médias dans l'art*. Thames and Hudson, oct 2005. (Cité page 1.)

[200] Pedro V. Sander : Explicit early-z culling for efficient fluid flow simulation and rendering. Rapport technique, ATI Research, August 2004. Technical Report. (Cité page 61.)

[201] D. Scharstein et R. Szeliski : A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision*, 47(1):7–42, 2002. (Cité page 118.)

[202] S. Schenke, B.C. Wunsche et J. Denzler : GPU-Based Volume Segmentation. *Proceedings of IVCNZ 2005, Dunedin, New Zealand*, pages 171–176, 2005. (Cité page 67.)

[203] T. Schiwietz et R. Westerman : GPU-PIV. *Proceedings of the Vision, Modeling, and Visualization Conference (VMV'04)*, pages 151–158, 2004. (Cité page 63.)

[204] C. Schmid, R. Mohr, C. Bauckhage et M. INRIA : Comparing and evaluating interest points. *In Computer Vision, 1998. Sixth International Conference on*, pages 230–235, 1998. (Cité page 135.)

[205] Jens Schneider et Rüdiger Westermann : GPU-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006. (Cité page 71.)

[206] S.M. Seitz, B. Curless, J. Diebel, D. Scharstein et R. Szeliski : A comparison and evaluation of multi-view stereo reconstruction algorithms. *Int. Conf. on Computer Vision and Pattern Recognition*, pages 519–528, 2006. (Cité page 118.)

[207] Shubhabrata Sengupta, Mark J. Harris, Yao Zhang et John D. Owens : Scan Primitives for GPU Computing. *In Graphics Hardware 2007*, pages 97–106. ACM, août 2007. (Cité pages 43 et 45.)

[208] Shubhabrata Sengupta, Aaron E. Lefohn et John D. Owens : A work-efficient step-efficient prefix sum algorithm. *In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, mai 2006. (Cité pages 43 et 45.)

[209] Musawir A. Shah, Jaakko Konttinen et Sumanta Pattanaik : Caustics mapping : An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, 2007. (Cité page 65.)

[210] Anthony Sherbondy, Mike Houston et Sandy Napel : Fast volume segmentation with simultaneous visualization using programmable graphics hardware. *In VIS '03 : Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 23, Washington, DC, USA, 2003. IEEE Computer Society. (Cité page 67.)

[211] L.J. Shiue, V. Goel et J. Peters : Mesh mutation in programmable graphics hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–24, 2003. (Cité page 71.)

[212] H.T. Siegelmann et E.D. Sontag : Analog computation via neural networks. *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pages 98–107, 1993. (Cité page 85.)

[213] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney et John D. Owens : Efficient computation of sum-products on gpus through software-managed cache. *In ICS '08 : Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM. (Cité page 58.)

[214] Silicon Graphics : Site internet d'opengl. http://www.opengl.org/. (Cité page 26.)

[215] S. Sinha, J.M. Frahm, M. Pollefeys et Y. Genc : GPU-based Video Feature Tracking and Matching. *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, 278, 2006. (Cité pages 144 et 150.)

[216] A. Smirnov et T. Chiueh : An Implementation of a FIR Filter on a GPU, 2005. (Cité page 63.)

[217] I. Sobel et G. Feldman : A 3x3 isotropic gradient operator for image processing. *Never published but presented at a talk at the Stanford Artificial Project*, 1968. (Cité page 135.)

[218] S. Song, K.D. Miller et LF Abbott : Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3:919–926, 2000. (Cité page 89.)

[219] C. Strecha, T. Tuytelaars et L. Van Gool : Dense matching of multiple wide-baseline views. *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1194–1201, 2003. (Cité page 118.)

[220] M. Strengert, M. Kraus et T. Ertl : Pyramid Methods in GPU-Based Image Processing. *Vision, Modeling, and Visualization 2006 : Proceedings, November 22-24, 2006, Aachen, Germany*, 2006. (Cité page 68.)

[221] Robert Strzodka et Alexandru Telea : Generalized Distance Transforms and skeletons in graphics hardware. *In Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '04)*, pages 221–230, 2004. (Cité page 71.)

[222] Thilaka Sumanaweera et Donald Liu : Medical image reconstruction with the FFT. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 765–784, 2005. (Cité page 63.)

[223] Vladimir Surkov : Parallel option pricing with fourier space time-stepping method on graphics processing units. *In Parallel and Distributed Processing*, pages 1–7. IEEE, 2008. (Cité page 62.)

[224] M. Sussman, W. Crutchfield et M. Papakipos : Pseudorandom number generation on the gpu. *In GH '06 : Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 87–94, New York, NY, USA, 2006. ACM. (Cité pages 62 et 102.)

[225] Jörg Schmittler Sven Woop et Philipp Slusallek : RPU : A Programmable Ray Processing Unit for Realtime Ray Tracing. *In Proceedings of ACM SIGGRAPH 2005*, July 2005. (Cité page 65.)

[226] David Tarditi, Sidd Puri et Jose Oglesby : Accelerator : using data parallelism to program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, 2006. (Cité page 26.)

[227] R. Temam : Navier-Stokes Equations. *Interaction*, 1989. (Cité page 60.)

[228] Timothy B. Terriberry, Lindley M. French et John Helmsen : GPU Accelerating Speeded-Up Robust Features. *3DPVT'08 - the Fourth International Symposium on 3D Data Processing, Visualization and Transmission*, 2008. (Cité pages 147 et 150.)

[229] Chris J. Thompson, Sahngyun Hahn et Mark Oskin : Using modern graphics architectures for general-purpose computing : a framework and analysis. *In MICRO 35 : Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. (Cité page 55.)

[230] N. Thrane et LO Simonsen : A comparison of acceleration structures for GPU assisted ray tracing. *Master's thesis, University of Aarhus*, 2005. (Cité page 64.)

[231] A. Tonnelier : The McKean's Caricature of the Fitzhugh-Nagumo Model I. The Space-Clamped System. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 63(2):459–484, 2003. (Cité pages 85 et 88.)

[232] R.D. Traub, D. Contreras, M.O. Cunningham, H. Murray, F.E.N. LeBeau, A. Roopun, A. Bibbig, W.B. Wilent, M.J. Higley et M.A. Whittington : Single-Column Thalamocortical Network Model Exhibiting Gamma Oscillations, Sleep Spindles, and Epileptogenic Bursts. *Journal of Neurophysiology*, 93(4):2194–2232, 2005. (Cité page 93.)

[233] Chris Trendall et A. James Stewart : General calculations using graphics hardware with applications to interactive caustics. *In Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 287–298, London, UK, 2000. Springer-Verlag. (Cité page 53.)

[234] Neil Trevett : OpenCL - Heterogeneous Parallel Programming. SIGGRAPH'08 BOF slides, 2008. (Cité pages 4, 26, 160 et 165.)

[235] Lewis W. Tucker et George G. Robertson : Architecture and applications of the connection machine. *Computer*, 21(8):26–38, 1988. (Cité page 16.)

[236] H.C. Tuckwell : *Introduction to Theoretical Neurobiology*. Cambridge University Press, 1988. (Cité page 85.)

[237] Stéphane Vialle : Fine grained computations and interactive distributed applications : Two important issues in parallel, distributed and grid computing. Invited talk. In ricordo di Amelia. Giornata in memoria della dot.ssa Amelia De Vivo. University of Potenza, Italy, 2007. (Cité page 5.)

[238] Ivan Viola, Armin Kanitsar et Meister Eduard Gröller : Hardware-based nonlinear filtering and segmentation using high-level shading languages. *In* K. Moorhead G. Turk, J. van Wijk, éditeur : *Proceedings of IEEE Visualization 2003*, pages 309–316. IEEE, octobre 2003. (Cité page 67.)

[239] C. Vreeswijk : Partial synchronization in populations of pulse-coupled oscillators. *Physical Review E*, 54(5):5522–5537, 1996. (Cité pages 85 et 99.)

[240] J. Wang, T.T. Wong, P.A. Heng et C.S. Leung : Discrete Wavelet Transform on GPU. *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, 2004. (Cité page 63.)

[241] D. Weiskopf, T. Schafhitzel et T. Ertl : GPU-Based Nonlinear Ray Tracing. *Computer Graphics Forum*, 23(3):625–633, 2004. (Cité page 65.)

[242] Colin Whitby-Strevens : Transputers – past, present and future. *IEEE Micro*, 10(6):16–19, 76–82, 1990. (Cité page 16.)

[243] J. Woetzel et R. Koch : Multi-camera real-time depth estimation with discontinuity handling on PC graphics hardware. *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, 1, 2004. (Cité page 118.)

[244] J. Woetzel et R. Koch : Real-time multi-stereo depth estimation on GPU with approximative discontinuity handling. *Visual Media Production, 2004. CVMP 2004. First European Conference on*, pages 245–254, 2004. (Cité pages 71 et 118.)

[245] Jan Woetzel, Patrick Fittkau et Reinhard Koch : Real-time Adaptive Tone Mapping for Monitoring High Contrast Hemispherical Image Capture with the GPU. *Visual Media Production, 2006. CVMP 2006. 3rd European Conference on*, pages 197–197, November 2006. (Cité page 69.)

[246] Enhua Wu, Youquan Liu et Xuehui Liu : An improved study of real-time fluid simulation on GPU : Research articles. *Comput. Animat. Virtual Worlds*, 15(3-4):139–146, 2004. (Cité page 59.)

[247] Chris Wyman et Scott Davis : Interactive image-space techniques for approximating caustics. *In I3D '06 : Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 153–160, New York, NY, USA, 2006. ACM. (Cité page 65.)

[248] C. Wynn : OpenGL Render-to-Texture. *Nvidia Corporation, white paper Edition*, 2004. (Cité page 28.)

[249] R. Yang et M. Pollefeys : Multi-resolution real-time stereo on commodity graphics hardware. *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, 1, 2003. (Cité page 71.)

[250] R. Yang et M. Pollefeys : A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11(1):7–18, 2005. (Cité page 118.)

[251] R. Yang, M. Pollefeys et S. Li : Improved real-time stereo on commodity graphics hardware. *Proc. of CVPR Workshop on Real-time 3D Sensors and Their Use*, 2004. (Cité page 71.)

[252] R. Yang, M. Pollefeys, H. Yang et G. Welch : A unified approach to real-time, multi-resolution, multi-baseline 2D view synthesis and 3D depth estimation using commodity graphics hardware. *International Journal of Image and Graphics*, 4(4):627–651, 2004. (Cité page 118.)

[253] R. Yang et G. Welch : Fast image segmentation and smoothing using commodity graphics hardware. *J. Graph. Tools*, 7(4):91–100, 2002. (Cité page 67.)

[254] Eric Young et Frank Jargstorff : Image Processing and Video Algorithms with CUDA. NVISION 2008 presentation, 2008. http://developer.nvidia.com/object/nvision08-ImageVideoCUDA.html. (Cité page 26.)

[255] C. Zach, K. Karner et H. Bischof : Hierarchical Disparity Estimation with Programmable 3D Hardware. *WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision), Short Communications*, pages 275–282, 2004. (Cité page 118.)

[256] C. Zach, A. Klaus, M. Hadwiger et K. Karner : Accurate dense stereo reconstruction using graphics hardware. *EUROGRAPHICS 2003, Short Presentations*, 2003. (Cité pages 119 et 120.)

[257] Cyril Zeller : Cloth simulation on the GPU. *In SIGGRAPH '05 : ACM SIGGRAPH 2005 Sketches*, page 39, New York, NY, USA, 2005. ACM. (Cité page 60.)

[258] Cyril Zeller : Cloth simulation. *Nvidia Corporation, white paper Edition*, February 2007. (Cité pages 60 et 61.)

[259] Z. Zhang : Determining the Epipolar Geometry and its Uncertainty : A Review. *International Journal of Computer Vision*, 27(2):161–195, 1998. (Cité page 149.)

[260] Ye Zhao : Lattice boltzmann based PDE solver on the GPU. *Vis. Comput.*, 24(5):323–333, 2008. (Cité page 57.)

Ce document a été préparé à l'aide de l'éditeur de texte TeXnicCenter et du logiciel de composition typographique LaTeX 2ε.