Enjeux de conception des architectures GPGPU : unités arithmétiques spécialisées et exploitation de la régularité

Caroline Collange

▶ To cite this version:

Caroline Collange. Enjeux de conception des architectures GPGPU : unités arithmétiques spécialisées et exploitation de la régularité. Réseaux et télécommunications [cs.NI]. Université de Perpignan, 2010. Français. NNT : . tel-00567267

HAL Id: tel-00567267
https://theses.hal.science/tel-00567267

Submitted on 19 Feb 2011

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

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

UNIVERSITÉ DE PERPIGNAN VIA DOMITIA
Laboratoire ELIAUS :
Électronique, Informatique, Automatique et Systèmes

THÈSE

Présentée le 30 novembre 2010 par

Caroline COLLANGE

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DE PERPIGNAN
Spécialité **INFORMATIQUE**

Au titre de l'école doctorale 305 : E²

Enjeux de conception des architectures GPGPU : unités arithmétiques spécialisées et exploitation de la régularité

soutenue devant la commission d'examen formée de

M. :	Dominique	LAVENIER	Président
MM. :	Daniel	ETIEMBLE	Rapporteurs
	Mateu	SBERT	
MM. :	Florent	DE DINECHIN	Examinateurs
	Ronan	KERYELL	
	Marc	DAUMAS	Directeur
	David	DEFOUR	Co-directeur

GPU shader engines are not your family's vector processor
Neither Uncle Seymour (Cray)
Nor Uncle Steve (Wallach)
Andy Glew [Gle09]

Remerciements

Je tiens à remercier l'ensemble de mon jury de thèse pour avoir accepté de venir jusqu'à Perpignan pour m'écouter et me transmettre leurs commentaires. Merci à Dominique Lavenier pour avoir accepté de présider ce jury, et à Daniel Etiemble et Mateu Sbert pour leur relecture attentive du manuscrit. Merci également à Florent de Dinechin et Ronan Keryell pour leurs questions et nombreuses remarques constructives.

Je remercie Marc Daumas pour avoir assumé la tâche délicate et parfois ingrate de diriger cette thèse. Merci à David Defour pour m'avoir accompagné scientifiquement et humainement depuis mon stage de fin de Master. Merci David pour ton soutien sans faille, et pour avoir toléré mon entêtement dans nos discussions scientifiques.

Je remercie également les chercheurs de tous horizons avec qui j'ai eu la chance et le plaisir de travailler. Un grand merci à Mark pour avoir m'avoir accordé une confiance et un soutien sans faille dès mon premier stage à Lehigh, et pour m'avoir fait partager sa passion pour les « règles à calcul électroniques ». Merci également à Arnaud pour son soutien et les discussions enrichissantes que nous avons mené. Merci à Jorge, à Yogi et à Yao pour leurs contributions respectives à ce travail.

Un autre grand merci aux membres de l'équipe DALI – Philippe, Bernard, David D. et David P., Christophe, Matthieu, Guillaume pour leur aide et pour les discussions constructives que j'ai pu mener avec chacun. Merci à mes co-bureaux successifs Nicolas, Pascal, Éric, Arnault, Laurent, puis David et Álvaro à Lyon pour avoir toujours maintenu une ambiance conviviale dans le bureau. Merci aussi à Benaoumeur, Mourad, Ali et Chen pour leurs encouragements.

Je tiens aussi à remercier tous les membres de l'équipe Arénaire qui m'ont accueilli lors de ma rédaction de thèse. Je remercie Florent pour son soutien continu malgré ma trahison (partir à Perpignan), mon prédécesseur Nicolas L. pour m'avoir transmis patiemment sa riche expérience aussi bien à Perpignan qu'à Lyon, Nathalie et Claude-Pierre pour leurs commentaires constructifs avant ma soutenance, Damien, Gilles, Guillaume, Jean-Michel, Nicolas B., Serge, Vincent, Adrien, Álvaro, Andy, Bogdan, Christophe, David, Diep, Erik, Ioana, Ivan, Xavier pour leur soutien.

Merci à tous ceux que j'ai eu l'occasion de croiser durant et/ou avant ma thèse et qui m'ont apporté leurs encouragements, notamment Jérémie, Stef, Pierre, Mehdi, Nicolas Estibals, Nicolas Brunie, Marthe et certainement d'autres que j'oublie.

Je remercie Cyril Z. pour la confiance qu'il m'a accordée en m'invitant en stage à Devtech, et pour m'y avoir abrité des tracas administratifs. Merci à Paulius pour son encadrement et ses conseils, ainsi qu'à l'ensemble de Devtech pour leur accueil. Merci à Norbert pour avoir su répondre à toutes les questions que je me posais et d'autres encore, et à Tim M. pour son soutien. Je remercie également les membres d'ARG et notamment Ronny et James pour leurs commentaires sur mon travail.

Merci également aux membres du forum NVIDIA, et notamment à Greg Diamos pour les discussions enrichissantes. Coin aussi aux X86-Adv et autres canards du forum CPC, à qui je dois les données de la figure 1.7 mais aussi et surtout beaucoup d'inspiration.

Merci aux membres de Vélo en Têt, avec qui j'ai pu pratiquer des activités aussi diverses que la peinture et la mécanique. Merci enfin à mes parents pour leur soutien. J'adresse mes plus plates excuses aux personnes que j'ai oubliées.

Table des matières

1 État de l'art du calcul sur GPU **11**
1.1 Environnements logiciels 11
1.1.1 API graphiques 12
1.1.2 Le modèle de programmation 14
1.1.3 JIT ou micro-architecture ? 16
1.1.4 Travaux indépendants 18
1.2 Architecture 18
1.2.1 Architectures considérées 18
1.2.2 Applications cibles 19
1.2.3 Une architecture orientée débit 20
1.2.4 Une perspective historique 21
1.2.5 Le modèle d'exécution : SIMT 22
1.2.6 Mémoires externes 23
1.2.7 Mémoires internes 24
1.2.8 Unités spécialisées pour le rendu graphique 26
1.3 Enjeux 28
1.3.1 Approche statique ou approche dynamique ? 28
1.3.2 Matériel spécialisé ou implémentation logicielle ? 32

2 Étude de cas : l'architecture Tesla **35**
2.1 Jeu d'instructions 36
2.2 Organisation générale 38
2.3 Gestion des tâches 40
2.3.1 Processeur de commandes 40
2.3.2 Ordonnancement à gros grain 41
2.4 SM 42
2.4.1 Lecture des instructions 42
2.4.2 Ordonnancement des warps 43
2.4.3 Banc de registres 45
2.4.4 Unités d'exécution 47
2.5 Hiérarchies mémoire 48
2.5.1 Mémoires internes 49
2.5.2 Mémoires niveau cluster 52

2.5.3 Traduction d'adresse . . . 53
2.5.4 Réseau d'interconnexion et ROP . . . 54
2.5.5 Contrôleur mémoire . . . 54
2.6 Gestion de l'énergie . . . 55
2.6.1 Protocole de test . . . 55
2.6.2 Tests applicatifs . . . 56
2.6.3 Ordonnancement . . . 57
2.6.4 Instructions . . . 57
2.6.5 Mémoire . . . 58
2.7 Conclusion . . . 59

3 Exploiter les spécificités arithmétiques du GPU 61
3.1 Introduction à l'arithmétique virgule flottante . . . 62
3.1.1 Système de représentation . . . 62
3.1.2 Unités de calcul . . . 63
3.1.3 Modélisation et analyse d'erreurs . . . 65
3.2 Test des unités de calcul . . . 66
3.2.1 Historique . . . 66
3.2.2 Architecture . . . 67
3.2.3 Caractéristiques des unités arithmétiques . . . 67
3.2.4 Évolution des fonctionnalités arithmétiques . . . 72
3.3 Détourner les unités graphiques . . . 74
3.3.1 Application : transferts radiatifs . . . 74
3.3.2 Détourner le pipeline graphique . . . 76
3.3.3 Résultats . . . 78
3.4 Évaluation de fonction par filtrage . . . 81
3.4.1 Détourner l'unité de textures . . . 81
3.4.2 Application : système logarithmique . . . 83
3.4.3 Résultats et validation . . . 84
3.5 Arithmétique d'intervalles sur GPU . . . 86
3.5.1 Arithmétique d'intervalles . . . 86
3.5.2 Simuler les modes d'arrondis manquants . . . 88
3.5.3 Arrondis dirigés corrects . . . 93
3.5.4 Résultats . . . 96

4 Barra, simulateur d'architecture CUDA 99
4.1 Simulateurs d'architectures parallèles . . . 99
4.2 Environnement de simulation . . . 101
4.2.1 Pilote . . . 101
4.2.2 Modules . . . 102
4.3 Simulation fonctionnelle . . . 102
4.4 Parallélisation de la simulation . . . 105
4.5 Validation et applications . . . 107
4.6 Caractérisation . . . 107

4.6.1 Vitesse de simulation 107
4.6.2 Précision 109

5 Tirer parti de la régularité parallèle 113
5.1 Introduction 114
5.1.1 Régularité séquentielle 114
5.1.2 Régularité parallèle 115
5.1.3 Sources de régularité 116
5.1.4 Applications 117
5.2 Régularité séquentielle 118
5.2.1 Caches conventionnels sur GPU 118
5.2.2 Caches de valeurs 119
5.2.3 Implications 119
5.3 Régularité de contrôle 120
5.3.1 Problématique 120
5.3.2 Reconvergence implicite 122
5.3.3 Validation 123
5.4 Vecteurs uniformes et affines 125
5.4.1 Intérêt 126
5.4.2 Scalarisation 127
5.4.3 Gestion du flot de contrôle 129
5.4.4 Problèmes et solutions 130
5.4.5 Résultats 132
5.5 Conséquences sur l'architecture et la micro-architecture 137
5.5.1 Approche statique 137
5.5.2 Approche dynamique 137
5.5.3 Conséquences sur les unités de calcul 140
5.6 Partage de tables pour l'évaluation de fonctions 142

Bibliographie 161

Introduction

Les microprocesseurs superscalaires se sont progressivements imposés depuis les années 1990 dans la quasi-totalité des segments de marché liés au calcul : ordinateurs personnels, serveurs, supercalculateurs, voire informatique embarquée [HPAD07]. Ils ont été suivis par les processeurs multi-cœurs au cours des années 2000. Leur production de masse permet des économies d'échelle qui rendent leur emploi plus économique que celui d'alternatives spécialisées.

L'évolution des performances de ces architectures se heurte cependant à plusieurs limites. D'une part, les gains permis par le parallélisme d'instructions s'amenuisent à mesure de son exploitation. Le parallélisme de tâches que les multi-cœurs exploitent en complément présente quant-à-lui des difficultés de programmation. D'autre part, le fossé de performance entre les unités de calcul et la mémoire s'accroît de manière continuelle, et devient de plus en plus difficile à combler tout en présentant au logiciel l'illusion d'une unique mémoire cohérente. Enfin, l'énergie consommée et dissipée par les processeurs représente le nouveau facteur limitant la performance, devant la surface de silicium et la vitesse de commutation des transistors [ABC+06].

En parallèle au développement des multi-cœurs, on constate l'émergence des processeurs graphiques (GPU) dédiés au rendu d'images de synthèse. Conçus à l'origine comme des accélérateurs spécialisés, ils sont devenus des architectures parallèles à grain fin entièrement programmables. Leur rôle est complémentaire à celui des processeurs généralistes. Alors que les superscalaires multi-cœur sont optimisés pour minimiser la latence de traitement d'un faible nombre de tâches séquentielles, les GPU sont conçus pour maximiser le débit d'exécution d'applications présentant une grande quantité de parallélisme.

Le marché de masse que constitue le jeu vidéo a permis de concevoir et de produire en volume des GPU dont la puissance de calcul est nettement supérieure à celle des processeurs multi-cœurs [MM05].

Cette puissance disponible à faible coût a suscité l'intérêt de la communauté scientifique qui y a vu l'occasion d'exploiter le potentiel des GPU pour d'autres tâches que le rendu graphique. Ainsi, les GPU ont été proposés pour accélérer des applications de calcul scientifique haute performance, telles que des simulations physiques, ou des applications multimédia, telles que du traitement d'image et de vidéo [OLG+07, GLGN+08].

Les constructeurs de GPU ont vu dans ces travaux académiques une opportunité de s'ouvrir au marché du calcul scientifique, et ont commencé à intégrer des fonctionnalités matérielles non liées au rendu graphique dans leurs processeurs respectifs [LNOM08, NVI09b]. Ce double engouement pour le calcul généraliste sur processeur graphique (GPGPU) soulève plusieurs questions.

Les GPU sont décrits dans la littérature tantôt comme des architectures totalement nouvelles [LNOM08], tantôt comme des architectures parallèles classiques qui n'auraient de nouveau que le vocabulaire qui leur est associé [VD08, Kan08]. Ces deux visions extrêmes ne représentent chacune qu'une part de la réalité. En effet, de nombreux aspects des GPU actuels sont hérités des architectures SIMD [HPAD07]. On peut notamment retrouver des similarités dans la façon dont est conçue l'architecture, dans les langages et environnements de programmation utilisés, ainsi que dans les algorithmes parallèles suivis. En contrepartie, les GPU ont aussi adopté des éléments issus du rendu graphique, voire des idées nouvelles. Quelles sont les connaissances et méthodes issues des processeurs SIMD qui sont également applicables aux GPU, et quelles sont les différences qu'il faut prendre en compte ? Nous tenterons de répondre à ces questions en dressant un état de l'art des environnements de programmation et des architectures dédiées au GPGPU au début du chapitre 1.

Les différents constructeurs de GPU ont des points de vue différents sur le niveau d'abstraction à considérer pour les architectures et les langages de programmation. Par exemple, AMD met l'accent sur la simplicité et l'efficacité du matériel en déléguant les tâches d'ordonnancement au compilateur, tandis que les architectures NVIDIA fournissent des mécanismes matériels transparents pour masquer la complexité interne. De même, ces constructeurs proposent des environnements de programmation se plaçant à différents niveaux d'abstraction. Doit-on adopter des mécanismes d'optimisation statiques ou dynamiques ? Nous évaluerons les avantages et les inconvénients de chaque approche dans la suite du premier chapitre.

Les programmeurs ont accès uniquement à une vue partielle des mécanismes mis en œuvre en matériel, ainsi qu'aux interactions entre le programme, le compilateur et l'architecture. Les principaux environnements de programmation pour GPU se comportent comme des boîtes noires, ce qui rend l'optimisation du code délicate. Ainsi, le manque de documentation précise semble être l'un des principaux griefs des programmeurs sur GPU à l'encontre des constructeurs [VD08, WPSAM10]. Nous désirons également comprendre et étudier les enjeux de conception des GPU. Pour répondre à ces deux questions, nous chercherons à déterminer plus précisément le fonctionnement des architectures actuelles au travers de tests synthétiques que nous appliquerons au cours du chapitre 2.

Les architectures parallèles promettent des gains en efficacité énergétique par rapport aux processeurs superscalaires. Ainsi, Nebulae, le deuxième supercalculateur du classement Top500 de juin 2010 construit à base de GPU fournit une puissance de calcul de 1,27 petaflops sous le test Linpack pour une consommation annoncée à 2,5 MW. Par comparaison, son concurrent direct Jaguar à base de processeurs multi-cœurs atteint 1,75 petaflops pour une consommation de 7 MW [Mer10]. Cependant, il est difficile de quantifier précisément l'impact que les décisions de programmation et de conception auront sur la consommation. Nous présenterons des mesures de consommation afin d'orienter ces choix à la fin du chapitre 2.

Les GPU disposent d'unités spécialisées pour effectuer des tâches liées au rendu graphique, telles que les unités de rastérisation, de filtrage de textures ou de fusion de fragments. Celles-ci permettent des gains en performance, mais surtout en consommation. Par exemple, le système sur puce NVIDIA Tegra 2 destiné aux appareils multimédia mobiles emploie des accélérateurs spécialisés distincts pour l'encodage vidéo, le décodage vidéo, le traitement d'images et de son et le rendu graphique, alors que ses processeurs généralistes ARM Cortex-A9 n'incluent délibérément pas d'extensions SIMD virgule-flottante [NVI10c]. La définition même d'une unité

spécialisée semble exclure toute utilisation en dehors de son domaine d'application. Qu'en est-il réellement ? Nous explorerons des possibilités de détourner ces unités pour effectuer des calculs associés à plusieurs applications non-graphiques au cours du chapitre 3.

Les unités de calcul généralistes en virgule flottante des GPU diffèrent également par rapport à celles des CPU, voire entre générations et entre constructeurs de GPU. Nous ferons une étude détaillée du comportement arithmétique d'une génération de GPU au moyen de tests synthétiques, et présenterons un récapitulatif des spécificités des unités arithmétiques de l'ensemble des GPU de ces cinq dernières années.

Cette étude nous révélera que les unités arithmétiques des GPU offrent des fonctionnalités supplémentaires par rapport à leurs équivalents sur les CPU les plus répandus. Nous tirerons parti de certaines de ces possibilités pour construire une bibliothèque d'arithmétique d'intervalles efficace dans la suite du chapitre 3.

Les GPU représentent une architecture encore en mutation. Leur étude du point de vue de l'architecture des ordinateurs est nécessaire pour effectuer des choix de conception pertinents. Elle requiert un modèle réaliste du matériel permettant l'exploration micro-architecturale. Nous présenterons au chapitre 4 un simulateur de l'architecture NVIDIA Tesla qui est à même de répondre à ce besoin.

Les faibles performances que les GPU obtiennent sur du code parallèle irrégulier constitue le principal reproche que leurs détracteurs leur accordent [Pfi08]. Cette faiblesse des GPU constitue aussi leur force : l'exploitation de régularité présente dans les applications parallèles est une des raisons qui explique l'efficacité des architectures GPU. Faire évoluer les GPU pour les rendre aussi souples que les CPU scalaires ferait perdre une grande partie de leur efficacité sur les applications régulières. L'enjeu consiste plutôt à établir un compromis entre la flexibilité et l'exploitation de la régularité. Au delà des mécanismes d'exécution existants, nous chercherons à explorer certains de ces compromis et nous considérerons davantage de moyens de tirer parti de la régularité dans le chapitre 5.

La régularité de contrôle représente une première manifestation de la régularité. Les systèmes matériels existants nécessitent d'inclure des informations particulières au niveau du jeu d'instructions pour communiquer les points d'indépendance de contrôle. Nous considérerons un mécanisme transparent identifiant la régularité de contrôle de manière dynamique qui permette de s'affranchir de ces informations. Ainsi, l'impact sur le jeu d'instructions et sur le compilateur est éliminé.

La régularité sur les données se manifeste par de de la redondance dans les calculs et en mémoire. Nous déterminerons dans la suite du chapitre 5 des moyens d'éliminer cette redondance en exploitant la régularité. La détection peut être réalisée en matériel à l'exécution ou lors de la compilation. Cette optimisation vise aussi bien une réduction de la consommation qu'une amélioration des performances.

Chapitre 1

État de l'art du calcul sur GPU

Le domaine du calcul généraliste sur GPU est actuellement en phase de transition depuis des prototypes de recherche vers une utilisation dans l'industrie. Il reste moins bien maîtrisé que la programmation sur des architectures séquentielles, voire des architectures parallèles plus conventionnelles. Programmer efficacement un GPU réclame une compréhension du fonctionnement des couches logicielles impliquées autant que de celui des architectures matérielles sous-jacentes.

Ainsi, les deux questions qui se posent sont les suivantes.

– Comment exploiter la puissance de calcul des GPU pour accélérer des applications de calcul généraliste ?
– Comment concevoir un GPU capable d'exécuter efficacement aussi bien les tâches graphiques que les tâches généralistes ?

Pour répondre à la première question, nous présenterons dans la section 1.1 une étude comparée des outils de programmation existants, en mettant en perspective leur évolution dans le temps.

La seconde question fera l'objet d'une étude des architectures GPU au travers d'exemples actuels dans la section 1.2. Nous en extrairons les principaux enjeux qui sous-tendent la conception d'un GPU.

1.1 Environnements logiciels

Le GPU est conçu à l'origine comme un coprocesseur spécialisé dédié à accélérer les calculs intervenant dans le rendu interactif d'images de synthèse. Il prend place au sein d'un système *hôte*, constitué d'un ou plusieurs CPU, d'un espace mémoire partagé et d'autres périphériques.

En tant que périphérique de calcul, le GPU utilise un environnement d'exécution et un pilote de périphérique. Un programme utilisant le GPU sera divisé en deux parties : un programme principal exécuté par le processeur hôte, dont une des tâches consistera à configurer le GPU, et éventuellement un ou plusieurs *noyaux de calcul* exécutés par le GPU.

Les modèles de programmation associés aux noyaux de calcul GPU sont tous dérivés du modèle SPMD (Single Program, Multiple Data). Le code du noyau est exécuté de manière

concurrente par un grand nombre de *threads*, de l'ordre de la dizaine de milliers. Tous les threads exécutent le code du même noyau. Afin de distinguer les threads entre eux, chacun dispose d'un *identifiant* unique. Ainsi, le noyau correspond au corps d'une boucle parallèle, tandis que l'identifiant du thread correspond au compteur de la boucle parallèle.

Les distinctions entre les modèles de programmation résident dans les possibilités offertes concernant la communication et les synchronisations entre threads, leur souplesse et leur granularité. La thèse de Göddeke fournit une vue d'ensemble des environnements GPGPU dans son chapitre 3 [Gö10].

Nous considérerons ici les environnements basés sur les bibliothèques de rendu graphique historiques, puis les environnements dédiés à la programmation généralistes.

1.1.1 API graphiques

La manière traditionnelle de configurer et programmer un GPU consiste passer par l'intermédiaire d'environnements de programmation dédiés au rendu graphique. Les environnements de programmation graphique actuellement utilisés sont Direct3D de Microsoft [Mica] et les implémentations de la norme OpenGL du consortium Khronos [OGL].

Le pipeline graphique Les algorithmes de rendu graphique direct opèrent selon une succession d'étapes. Nous considérerons ici le pipeline de rendu graphique de l'environnement d'OpenGL 2.0 et Direct3D 9, tel qu'il existait en 2005. Notons que des étapes supplémentaires que nous n'aborderons pas liées à l'instanciation dynamique de primitives et à la tessalation ont été introduites ultérieurement. Les étapes de ce pipeline de rendu sont représentées sur la figure 1.1.

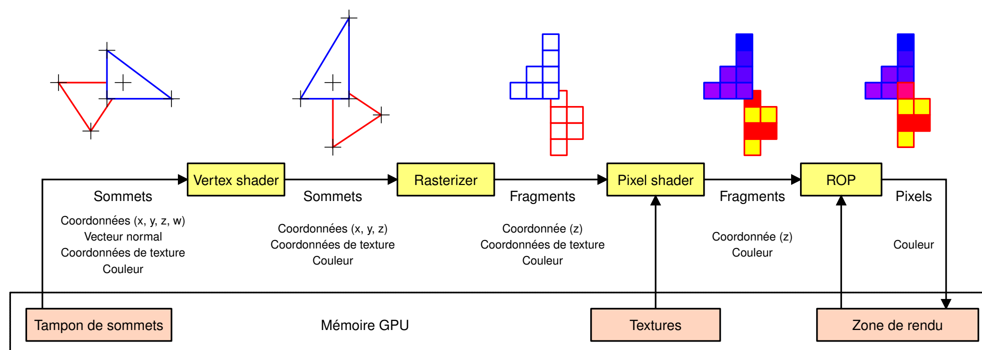


FIGURE 1.1 – Pipeline de rendu graphique Direct3D 9.

Les données en entrée provenant du CPU sont des listes de *sommets* composant des *primitives* géométriques, telles que des triangles ou des lignes. Des *attributs* sont associés à chaque sommet pour représenter ses coordonnées dans l'espace, sa couleur, ses coordonnées de texture ou son vecteur normal.

Un premier programme nommé *vertex shader* est appliqué de manière indépendante à

chaque sommet. Il se charge de modifier les attributs, typiquement en appliquant des transformations géométriques, afin de calculer les coordonnées du sommet dans l'espace de l'écran.

Les primitives sont ensuite assemblées à partir des sommets les composant. L'unité de *rastérisation* détermine les pixels (ou *fragments*) couverts par chaque primitive.

Des attributs de fragments correspondant aux attributs de sortie du vertex shader sont calculés par interpolation entre les valeurs des attributs des sommets de la primitive. Les coordonnées de chaque fragment sont fixées définitivement à cette étape.

La seconde étape programmable, le *fragment shader* ou *pixel shader* consiste à calculer la couleur résultante de chaque fragment.

Ces deux types de shader peuvent lire depuis des images rangées en mémoire, nommées *textures*. Le processus d'accès à la texture est dit *échantillonnage*.

Enfin, le fragment doit rejoindre l'image de destination. Une étape d'arbitrage et de fusion est réalisée par les unités ROP (Raster Operations). Il permet de gérer les situations d'occlusion et de transparence qui se présentent lorsque plusieurs fragments se trouvent aux mêmes coordonnées.

Détournement des API graphiques Les deux fonctionnalités qui ont permis la programmation généraliste sur GPU sont d'une part, la possibilité d'effectuer le rendu dans une zone mémoire, puis de réutiliser cette zone comme texture dans une étape ultérieure, et d'autre part, les unités de shaders programmables.

De nombreuses techniques ont été mises au point afin d'exprimer les algorithmes parallèles classiques tels que les réductions, les calculs de préfixes ou les tris de manière graphique. Owens et al. présentent un état de l'art détaillé de ces techniques [OLG$^+$07].

Les calculs généralistes sont typiquement effectués dans le fragment shader. Chaque fragment représente alors un thread indépendant. L'identifiant du thread est constitué par ses coordonnées dans la zone de rendu. Le nombre de threads à exécuter est déterminé par les dimensions d'un rectangle à dessiner, qui englobe généralement l'ensemble de la zone de rendu. Les lectures en mémoire sont opérées par des échantillonnages de textures, ce qui permet d'adresser des valeurs arbitraires dans un table. En revanche, les écritures doivent s'opérer uniquement aux coordonnées de destination dans la zone de rendu, c'est-à-dire au pixel correspondant à l'identifiant du thread.

Devoir détourner les modèles de programmation et les environnements dédiés au rendu graphique rend la programmation complexe. Des langages de plus haut niveau ont été développés pour pallier à cette difficulté, notamment le langage du projet Brook [BFH$^+$04]. Néanmoins, ces langages ne résolvent pas les problèmes que posent les contraintes logicielles et matérielles sur la souplesse de programmation.

Les contraintes logicielles proviennent du surcoût qu'engendre l'utilisation d'un environnement dédié au rendu graphique pour effectuer les calculs. Au niveau matériel, l'absence de gestion matérielle du *scatter*, c'est-à-dire la possibilité d'écrire à une adresse arbitraire en mémoire est restée un frein au développement du GPGPU.

Afin de contourner la pile logicielle graphique et permettre l'opération scatter, ATI (aujourd'hui AMD) a développé l'environnement CTM [PSG06]. Il permet l'accès direct au GPU au niveau assembleur. Cependant, la difficulté de programmation due au positionnement bas-

niveau du langage lui a valu un succès mitigé auprès des développeurs. Cette expérience a fait apparaître la nécessité d'offrir une pile logicielle complète.

L'impossibilité d'effectuer des communications locales à grain fin limite également l'efficacité de ces approches. De fait, la génération suivante d'environnements de développement a intégré des langages de plus haut niveau, ainsi que des mémoires locales partagées en matériel.

1.1.2 Le modèle de programmation

L'arrivée de l'environnement de développement CUDA de NVIDIA a marqué un pas significatif dans la démocratisation du GPGPU [NVI10b]. Cet environnement combine un modèle de programmation relativement accessible au travers d'un langage niveau C, et une architecture matérielle permettant les lectures et écritures mémoire arbitraires et offrant un accès à des mémoires locales et des synchronisations entre threads.

L'API CAL d'AMD est basée sur des concepts similaires, et fournit une fondation sur laquelle peuvent s'appuyer des langages de plus haut niveau tels que Brook+, une version reprise par AMD du projet Brook [AMD09a].

Ces API propriétaires ont inspiré des normes permettant la portabilité des applications entre les GPU des différents constructeurs. L'environnement OpenCL du consortium Khronos établit un norme multi-plate-forme principalement inspiré par CUDA [Mun09]. Microsoft inclut également les Compute Shaders dans son environnement de programmation graphique Direct3D à partir de la version 11 [Mica]. Cette interface est quant-à-elle plus proche de CAL.

Tous ces environnements ont pour point commun d'être basés sur un modèle de programmation similaire. Ce modèle tire son inspiration du modèle *bulk-synchronous programming* (BSP), qui considère des sections parallèles où les threads sont indépendants, séparées par des barrières de synchronisation globales [Val90]. Cependant, les synchronisations globales sont trop intrusives pour permettre le passage à l'échelle.

Le modèle de programmation des environnements GPU distingue deux niveaux d'ordonnancement imbriqués, qui fonctionnent chacun suivant le modèle BSP à sa propre échelle (figure 1.2).

Le programmeur découpe le domaine des threads de manière régulière en blocs nommés CTA (Cooperative Thread Arrays). Il peut placer des barrières de synchronisation locales entre tous les threads d'un CTA, et utiliser une mémoire locale à chaque CTA, nommée mémoire partagée. Cette mémoire est allouée de manière statique.

Entre l'exécution de deux noyaux de calcul, il est possible d'opérer une synchronisation globale.

Notons qu'il n'existe pas de consensus sur la terminologie liée à ce modèle. En effet, chaque acteur du calcul sur GPU a adopté son propre vocabulaire. La table 1.1 récapitule les principales différences. Ainsi, des expressions telles que « local memory » peuvent avoir un sens totalement différent suivant le contexte. Nous nous basons dans cette thèse sur une version francisée de la nomenclature NVIDIA.

TABLE 1.1 – Correspondance entre les terminologies des principaux constructeurs

NVIDIA	AMD	Intel	OpenCL	Ce document
Thread	Thread	Strand / Channel	Work item	Thread
Warp	Wavefront	Thread / Vector		Warp
CTA/Block	Thread group	Fiber	Work group	CTA
Scalar processor (SP) / CUDA Core	SPU	Lane	Processing Element	SP
Streaming multiprocessor (SM)	SIMD processor	Core / EU	Compute Unit	SM
Shared memory	Local data share (LDS)		Local memory	Mémoire partagée
Local memory			Private memory	Mémoire locale
Global memory	Local memory / Global buffer		Global memory	Mémoire globale
Stream	Queue		Command queue	File de commandes
Kernel	Kernel		Program	Noyau de calcul

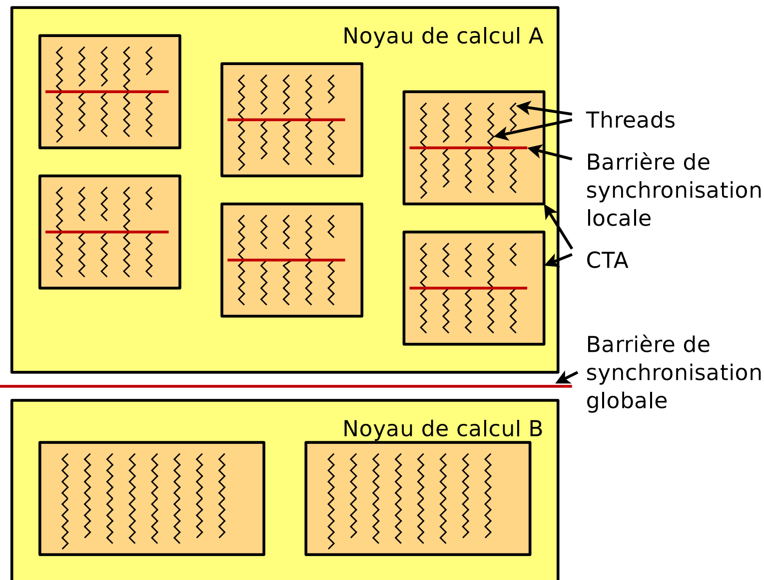


FIGURE 1.2 – Répartition des threads en CTA dans le modèle SIMT.

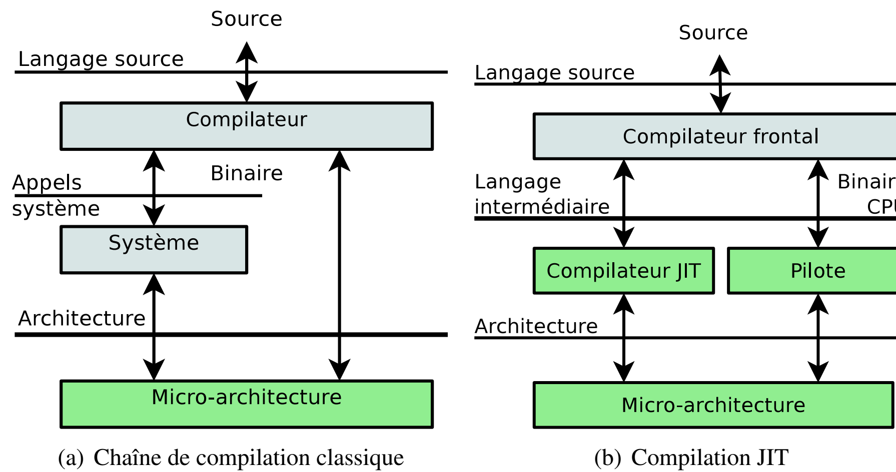
1.1.3 JIT ou micro-architecture ?

Les environnements de développement pour GPU que nous avons abordés sont des systèmes complets. Ils regroupent à la fois un langage, un compilateur, une interface de bibliothèque (API), un pilote et une architecture matérielle. Cela représente une intégration verticale.

Or, dans une chaîne de compilation traditionnelle, les langages, compilateurs et systèmes d'exploitation forment un écosystème qui n'est pas lié au matériel. Le fabricant de processeurs se doit de définir et documenter précisément *l'architecture*, qui forme l'interface entre le logiciel et le matériel. Ainsi, on effectue la distinction entre l'architecture, qui définit le jeu d'instructions et sa sémantique, et la micro-architecture, qui est l'implémentation d'une architecture donnée. Dans le cadre classique, l'architecture est aussi l'interface entre les applications et le système (figure 1.3(a)).

Dans le cadre des GPU, l'architecture a moins de poids. L'interface entre les applications et le système est constituée par un langage intermédiaire (HLSL assembly pour Direct3D [Micb], NVIDIA PTX pour CUDA [NVI10d], AMD IL pour CAL [AMD09a]), voire un langage de haut-niveau (GLSL pour OpenGL, C pour OpenCL). Le constructeur maîtrisant l'ensemble de la chaîne de compilation, il n'est pas tenu de documenter la façon d'accéder au matériel. AMD et Intel publient des documentations succinctes du jeu d'instructions et des registres de configuration de leurs GPU respectifs [AMD09b, Int09b] et NVIDIA fournit aux développeurs une documentation simplifiée du jeu d'instructions Tesla depuis août 2010. Ces efforts de documentation sont cependant loin d'égaler ceux des jeux d'instructions des CPU [Int10a].

L'organisation des couches logicielles est similaire à celle des environnements fonctionnant à base de compilation à la volée (JIT) tels que Sun/Oracle Java ou Microsoft .Net. En addition, le code s'exécutant sur CPU communique avec le pilote du GPU par une des API graphiques ou



(a) Chaîne de compilation classique (b) Compilation JIT

FIGURE 1.3 – Compilation statique et compilation JIT.

GPGPU que nous avons abordées précédemment (figure 1.3(b)). Par exemple, CUDA fournit l'accès aux fonctionnalités de base par l'API *CUDA Driver*, et offre une API de plus haut niveau nommée *CUDA Runtime*. Une telle organisation présente à la fois des opportunités et des difficultés.

Opportunités D'une part, cette organisation permet de maintenir la compatibilité ascendante sans conséquence sur le matériel. Les constructeurs ont la possibilité de modifier le jeu d'instructions des GPU d'une génération sur l'autre. La compatibilité avec les applications existantes est conservée, car celles-ci seront recompilées à la volée.

D'autre part, les applications existantes peuvent profiter des améliorations du compilateur à chaque mise à jour des pilotes. Cet aspect est particulièrement manifeste dans le cadre des applications graphiques. Par exemple, chaque mise à jour des pilotes graphiques s'accompagne d'une annonce de gains de performance dans les jeux récents.

Cette approche minimise aussi les conséquences d'éventuels bogues matériels. Par exemple, le bogue de la division du Pentium d'Intel en 1995 a nécessité le retour des processeurs affectés, coûtant au constructeur près de 500 millions de dollars [Coe95]. Du côté des GPU, la documentation du R600 d'AMD mentionne un bogue causant l'exécution incorrecte d'un mode d'adressage indirect [AMD09d]. Ce problème pourtant sérieux a pu être contourné par logiciel et serait passé entièrement inaperçu s'il n'avait pas été documenté.

Enfin, ce modèle permet la génération dynamique de code spécialisé. Cette technique est utilisée par des applications graphiques.

Difficultés En retour, le processus de compilation et d'exécution est opaque. L'environnement ne fournit pas de retour précis sur l'efficacité de la compilation et de l'exécution. Il est donc difficile d'effectuer des optimisations fines spécifiques à l'architecture.

La recompilation à la volée fait perdre également en déterminisme. Des régressions accidentelles en termes de performance et de fonctionnalités sont toujours possibles. De plus,

une nouvelle version de l'environnement peut rendre visible des erreurs de programmation qui étaient tolérées auparavant, brisant la compatibilité avec des programmes existants. L'application CUDA Badaboom s'est par exemple révélée incompatible avec les GPU basés sur l'architecture Fermi [Bad10].

1.1.4 Travaux indépendants

La disponibilité d'environnements bas-niveau tels que CUDA a encouragé l'émergence de travaux académiques et industriels se basant sur ces environnements. On trouve notamment des outils de compilation, des langages de plus haut niveau et des environnements d'exécution.

Parmi ceux-ci, Ocelot est un environnement de compilation pour les programmes CUDA qui opère sur une représentation intermédiaire basée sur le langage PTX [DKYC10]. Il inclut un environnement d'exécution reproduisant les fonctionnalités de l'API CUDA Runtime. Les noyaux de calcul peuvent être émulés sur CPU par un interpréteur de la représentation intermédiaire, traduits à la volée vers un code CPU par LLVM [LA04], ou exécutés sur GPU.

Cet environnement offre un ensemble de structures de données permettant la transformation et l'analyse de code, tels que des graphes de flot de contrôle et de données et une allocation de registres virtuels sous forme SSA. Il fournit également des étapes de compilation telles qu'une allocation de registres au niveau PTX, et permet de générer des traces et des statistiques à partir de l'interpréteur.

L'objectif principal du projet est de fournir une machine virtuelle assurant la compilation JIT pour diverses architectures. Alors que l'environnement CUDA n'utilise le JIT que pour assurer la portabilité, Ocelot prévoit d'utiliser les données retournées par l'analyse dynamique de performances pour optimiser la compilation.

1.2 Architecture

Un GPU doit être à même d'exécuter efficacement un champ d'applications plus large que le strict rendu graphique. Ces applications ont pour point commun de permettre l'exploitation de parallélisme de données. Une architecture parallèle optimisée pour le débit permet de les exécuter efficacement.

1.2.1 Architectures considérées

Nous ferons référence dans la suite de cette thèse à un certain nombre de GPU commerciaux. Nous considérerons les gammes des constructeurs NVIDIA, AMD et Intel commercialisés entre 2005 et 2010. La chronologie de ces architectures est représentée sur la figure 1.4. Les nouvelles générations de GPU coïncident généralement avec les révisions de l'API Microsoft Direct3D.

Les noms des produits qui s'appuient sur ces GPU sont les gammes GeForce, Quadro et Tesla [1] de NVIDIA, les gammes Radeon, FireGL et FirePro/FireStream d'AMD et GMA

1. Le nom Tesla présenté sur la figure 1.4 correspond au nom de code associé à une architecture, et n'a pas de relation directe avec le nom commercial Tesla désignant la gamme destinée au calcul scientifique du même constructeur. Lorsque nous utiliserons le nom Tesla par la suite, nous ferons référence à l'architecture.

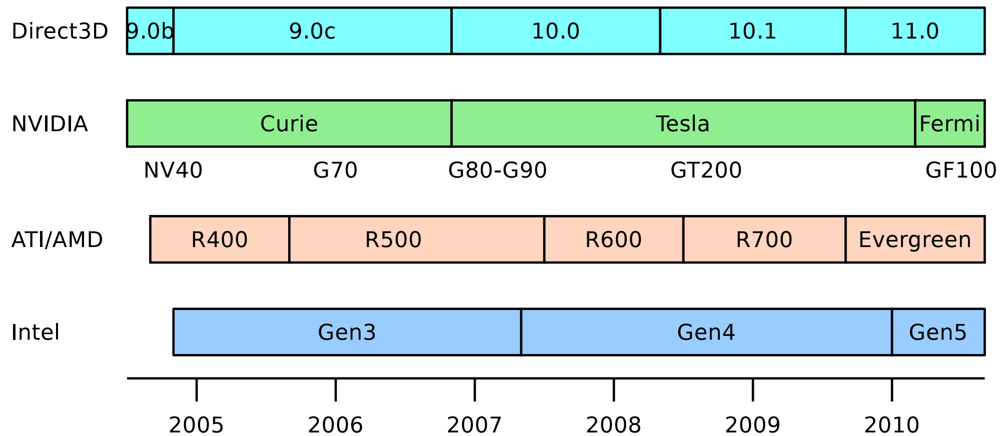


FIGURE 1.4 – Générations successives des architectures GPU NVIDIA, AMD et Intel considérées ici, mis en relation avec les versions de Microsoft Direct3D.

d'Intel.

Nous classerons également le projet Larrabee d'Intel parmi les GPU [SCS+08]. Bien qu'aucune version commerciale ne soit disponible à ce jour et que le projet de coprocesseur graphique Larrabee ait été suspendu au profit de l'architecture parallèle MIC [Int10b], les enjeux de conception de cette architecture sont similaire à ceux des GPU actuels. En effet, nous avons affaire dans les deux cas à une architecture parallèle généraliste initialement optimisée pour le rendu graphique.

1.2.2 Applications cibles

L'application historique des GPU est la synthèse d'images en temps réel pour les jeux vidéo. Le poids de ce marché a permis de justifier l'investissement dans la conception de coprocesseurs spécialisés (ASIC) basés sur des technologies de l'état de l'art.

En parallèle de l'augmentation de la puissance de calcul, l'industrie du jeu vidéo tend à réclamer plus de flexibilité. Cela a conduit entre autre à l'adoption des unités de shader programmables. Des programmeurs de jeux réclament encore aujourd'hui une flexibilité accrue, et la possibilité de s'abstraire du modèle de pipeline graphique traditionnel [Swe09]. Ainsi, les fonctionnalités récemment introduites visant la programmation généraliste peuvent également profiter aux moteurs de rendu des jeux vidéo.

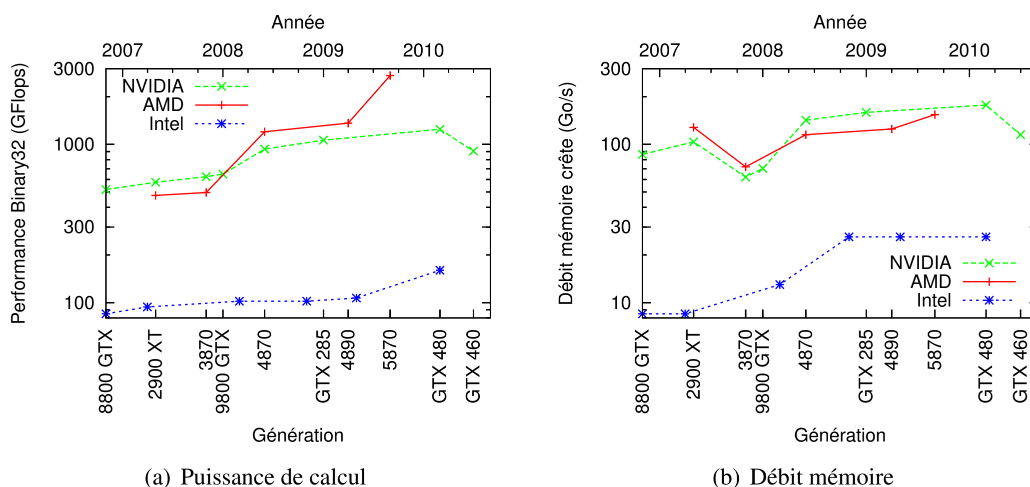
Les *shaders* utilisés dans les moteurs de rendu actuels peuvent s'avérer plus complexes en termes de dépendances, d'instructions et de contrôle de flot que des applications GPGPU régulières telles que de l'algèbre linéaire. Ainsi, les modèles d'exécution de Direct3D et OpenGL imposent un ordre de traitement des primitives strictement séquentiel [SWND07]. Par ailleurs, Norman Rubin mettait en évidence en 2008 une évolution exponentielle de la longueur des shaders, et une utilisation accrue de structures de contrôle [Rub08].

La diversification des applications que ciblent les GPU est de fait inévitable. Une architecture GPU est aujourd'hui conçue dès l'origine pour exécuter efficacement un panel d'applica-

tions présentant du parallélisme de données et de la régularité :
- le rendu 3D interactif pour les jeux vidéo,
- le rendu pour applications de conception assistée par ordinateur et de modélisation,
- les applications multimédia (image, son, vidéo),
- les éléments de jeux vidéo non liés au rendu (physique),
- le calcul scientifique.

1.2.3 Une architecture orientée débit

Considérons les puissances de calcul et les débits mémoire crêtes des architectures GPU de ces trois dernières années, comparés aux CPU de même génération sur la figure 1.5. Les processeurs comparés ici sont tous issus du segment haut de gamme du marché grand public.



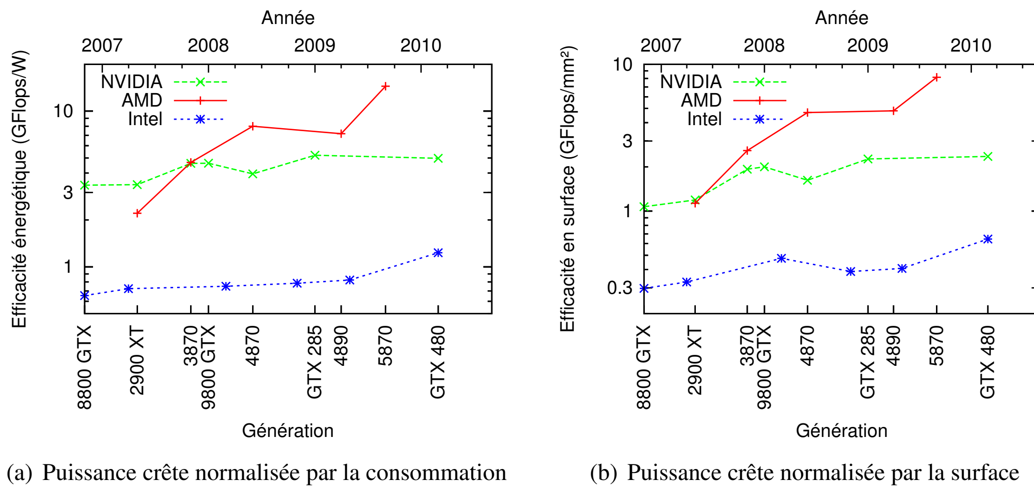
(a) Puissance de calcul (b) Débit mémoire

FIGURE 1.5 – Comparaison des puissances de calcul et débits mémoire crête des GPU NVIDIA et AMD, et des CPU Intel.

La performance crête des GPU reste supérieure d'un facteur 8 à 10 par rapport à celle des CPU. La tendance est à une augmentation de 25 % par an sur les CPU Intel, contre respectivement +35 % et +105 % par an sur les GPU NVIDIA et AMD. Le fossé des performances crêtes tendrait donc à se creuser dans l'immédiat. L'élargissement des unités SIMD et la généralisation de l'opérateur de multiplication-addition fusionnée (FMA) qu'offriront les extensions de jeux d'instructions tels qu'AVX pourraient compenser cette tendance [Int09a].

L'évolution du débit de la mémoire externe est plus modeste. Si l'on ne considère pas l'impact de l'intégration du contrôleur mémoire, il augmente de 20 % par an sur les CPU. Celui des GPU est plus élevé d'un facteur 6 à 7, et évolue dans les mêmes proportions. Cette tendance étant dépendante des contraintes technologiques liées à la DRAM, il est peu probable qu'il y ait d'évolution sur ce point dans un futur proche.

Normalisons maintenant les valeurs de performances crêtes par la consommation et la surface figure 1.6. Les valeurs de consommation sont issues de l'enveloppe thermique (TDP)



(a) Puissance crête normalisée par la consommation (b) Puissance crête normalisée par la surface

FIGURE 1.6 – Comparaison de l'efficacité théorique en énergie et en surface des GPU NVIDIA et AMD, et des CPU Intel.

fournie par le constructeur [2]. L'écart de performances entre les deux types d'architectures se réduit pour atteindre un facteur de 4 à 9. L'évolution de l'efficacité énergétique des architectures NVIDIA est similaire à celle des multi-cœurs (+15 %/an).

Plusieurs raisons expliquent ces différences de performances crêtes :

– l'exploitation du parallélisme de données par le GPU pour masquer les latences, tandis que le CPU se base sur le parallélisme d'instructions qui est plus coûteux à extraire au-delà d'un certain point [HPAD07],
– l'accent mis sur le calcul virgule flottante sur le GPU, aux dépens des formats entiers et à virgule fixe,
– le profil de réutilisation des données dans les applications sur GPU qui rend envisageable l'utilisation de caches de faible taille, libérant des ressources pour des unités d'exécution,
– la régularité des applications considérées, qui permet de simplifier les mécanismes de contrôle.

1.2.4 Une perspective historique

La conception du GPU tel qu'on le connaît aujourd'hui a été influencée par des travaux sur les architectures parallèles dans les années 1990. De nombreux aspects associés aux GPU se retrouvent dans cette littérature. Par exemple, l'organisation interne des registres et unités d'exécution de l'architecture NVIDIA Tesla est étonnamment proche de celle du projet Torrent-0 développé à Berkeley à la fin des années 1990 [Asa98].

Les architectures parallèles peuvent être classées grossièrement en plusieurs catégories, comprenant notamment :

2. Notons que l'enveloppe thermique des CPU n'inclut pas la consommation de la mémoire, ni celle du contrôleur mémoire sur les processeurs Core 2. En revanche, celle des GPU inclut la consommation sur l'ensemble de la carte graphique. Cette différence à l'avantage des CPU n'affecte pas les tendances globales.

– le modèle SIMD, qui consiste à exécuter chaque instruction simultanément sur plusieurs unités de calcul sur des données différentes,
– le modèle MIMD, où plusieurs instructions indépendantes sont exécutées sur autant d'unités de calcul [Fly72].

Les architectures de type SIMD opèrent sur des registres vectoriels. Pour permettre un traitement différencié entre les composantes individuelles de ces registres, les jeux d'instructions offrent de la prédication et des opérations *gather* et *scatter* [Asa98]. Ces mécanismes opèrent respectivement sur le contrôle et sur les données.

La prédication simule un flot de contrôle différencié sur chaque voie SIMD en permettant de désactiver de manière sélective le calcul sur certaines composantes d'un vecteur. Elle peut être complétée par des mécanismes à base de piles ou de compteurs pour gérer des structures de contrôle imbriquées arbitraires [LP84, Ker92].

L'opération *gather* prend en entrée un vecteur d'adresses mémoire et se charge de récupérer l'ensemble des données pointées par ces adresses pour les placer chacune dans la composante correspondante du registre vectoriel destination. L'opération *scatter* fonctionne de façon analogue sur les écritures mémoire, en répartissant un vecteur de données à des emplacements décrits par un vecteur d'adresses [Asa98].

Néanmoins, les GPU actuels présentent d'autres spécificités que les machines parallèles traditionnelles pour répondre à des contraintes différentes. D'un côté, la densité de logique à surface de silicium constante a été multipliée par 30 en dix ans, permettant l'intégration de supercalculateurs complets sur une puce. En revanche, la latence des accès à la mémoire externe n'a pas connu de tel saut quantitatif [Pat04]. La consommation des circuits devient également le principal facteur limitant, devant la surface de silicium. Ainsi, les architectures parallèles de la génération suivante ont opté pour une organisation hiérarchique et adopté de nombreux mécanismes destinés à optimiser les mouvements de données [KDR+02, KBH+04].

1.2.5 Le modèle d'exécution : SIMT

Le modèle de programmation présenté section 1.1.2 considère des CTA composés chacun d'un nombre de threads de l'ordre de la centaine. Les threads peuvent effectuer des communications et des synchronisations à grain fin à l'intérieur de chaque CTA.

Les CTA étant indépendants entre eux, ils peuvent être ordonnancés librement sur un ensemble de multiprocesseurs, ou SM. Chaque SM se charge d'exécuter les threads d'un ou plusieurs CTA. L'ordonnancement des CTA sur les SM est laissé à la liberté de l'environnement, permettant d'ajuster l'équilibre entre parallélisme et localité en fonction de la cible matérielle. Le modèle de programmation garantit que tous les threads d'un CTA seront en vie simultanément sur un cœur unique.

Chaque SM offre une mémoire partagée à faible latence qui est partitionnée de manière statique entre les CTA en cours d'exécution. Cette mémoire permet les communications entre les threads au sein d'un CTA.

Une solution naïve consisterait à concevoir chaque SM comme un processeur MIMD, composés de cœurs indépendants exécutant chacun un ou plusieurs threads. Cette approche nécessite de répliquer la logique de contrôle : caches d'instructions, mécanismes de lecture et de décodage d'instructions. Or, les applications que nous considérons exhibent de la régularité

d'instructions : plusieurs threads peuvent être synchronisés de manière à exécuter la même instruction au même moment.

Une telle instruction opérant pour le compte de plusieurs threads peut être traitée comme une instruction SIMD, à la manière des architectures décrites dans la section précédente. Ainsi, le modèle SIMT (single instruction, multiple threads) rassemble les threads d'un CTA en groupes de taille fixe nommés *warps*. Les threads d'un warp restent mutuellement synchronisés, et partagent le même mécanisme de lecture et de décodage d'instructions.

Le modèle SIMT est similaire au MIMD du point de vue du programmeur : des mécanismes d'exécution transparents exécuteront cependant le code sur des unités SIMD. À la différence des architectures SIMD conventionnelles, la vectorisation s'effectue lors de l'exécution plutôt qu'à la compilation.

D'un autre point de vue équivalent, les instructions opèrent toutes sur des vecteurs. Les instructions de lecture/écriture mémoire deviennent alors des instructions gather/scatter. Les instructions de comparaison et de branchement reproduisent de manière transparente une exécution proche d'un modèle MIMD.

Toujours selon ce point de vue, le jeu d'instructions n'offre aucune instruction de calcul scalaire, ni de registre scalaire, ni même d'opération de lecture ou écriture mémoire ; il est entièrement vectoriel. C'est paradoxalement cette généralisation vectorielle qui permet à ce type d'architecture d'exposer un modèle de programmation SPMD et d'être qualifiée de *scalaire* par NVIDIA [LNOM08].

Dans ce modèle, les données sont naturellement manipulées sous la forme de structures de tableaux (SoA), par opposition aux architectures SIMD à vecteurs courts capables de traiter des données sous forme de tableaux de structure (AoS). En SIMT, ce sont les opérations gather et scatter qui permettent d'effectuer les conversions entre SoA et AoS.

1.2.6 Mémoires externes

Les applications traditionnelles de rendu graphique opèrent sur des jeux de données plus importants que la quantité de mémoire qu'il est possible d'intégrer sur une puce.

De fait, l'architecture des GPU met l'accent sur la bande passante vers la mémoire graphique. Ces optimisations se font généralement au détriment de la latence. En effet, considérons figure 1.7 les latences mémoires mesurées sur plusieurs générations de GPU NVIDIA. Nous observons que la latence reste stable autour de $340\ \mathrm{ns}$, avec une légère augmentation de $2\ \%$ par an. Par comparaison, la latence mémoire d'un CPU avec contrôleur intégré est de l'ordre de $50\ \mathrm{ns}$, soit inférieure d'un facteur 7, et décroît de $6\ \%$ par an [Pat04].

Les processeurs graphiques des années 1980 à 1990 utilisaient des technologies de mémoires à double ports (VRAM), puis des mémoires exotiques supportant directement l'accélération des opérations graphiques 2D (SGRAM). Ces solutions sont devenues obsolètes en raison de l'augmentation de la vitesse des SDRAM, des avancées des contrôleurs mémoire, de la régularité moindre des accès et de facteurs économiques. Ainsi, les mémoires actuelles (GDDR5) sont basées sur la même technologie que les mémoires conventionnelles telles que la DDR3, mais bénéficient d'optimisations quantitatives pour améliorer leur vitesse au détriment du coût et des marges d'erreur. Les normes et protocoles employés étant très proches, l'avantage de cette approche est qu'un même contrôleur mémoire peut être aisément conçu pour être

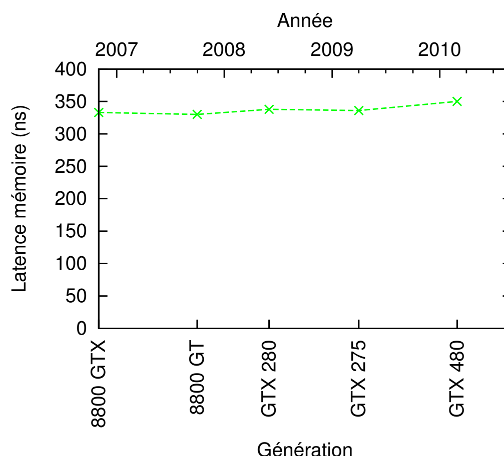


FIGURE 1.7 – Évolution des latences mesurées sur les GPU NVIDIA.

compatible avec les mémoires DDR2 et DDR3 employées sur les cartes graphiques d'entrée de gamme.

Les débits mémoire mis en œuvre sont comparables à ceux des architectures serveur multiprocesseur mais les enjeux sont différents. Dans le cas des GPU, les exigences sur la capacité sont plus réduites, mais les contraintes sur les coûts de production sont plus fortes.

Ainsi, les configurations mémoire choisies visent à maximiser la largeur des bus, tout en minimisant le nombre de puces. La capacité mémoire obtenue est principalement un effet secondaire des exigences en termes de débit. La largeur des bus mémoire de tous les GPU haut de gamme considérés varie entre 256 et 512 bits.

1.2.7 Mémoires internes

La figure 1.8 représente l'évolution de la taille des mémoires internes des mêmes architectures que précédemment. Les valeurs présentées incluent les bancs de registres et les caches.

Contrairement à une idée reçue, un GPU n'est pas constitué majoritairement d'unités de calcul, mais inclut également des quantités conséquentes de mémoires internes. La taille des bancs de registres et caches est en progression constante, et se rapproche des tailles des caches des CPU.

En effet, nous avons observé au travers des figures 1.5(b) et 1.7 que la bande passante mémoire b_m des GPU était en augmentation tandis que la latence t_m restait stable. Or, le volume de données en vol dans le pipeline mémoire d_m s'exprime par le produit entre la latence et le débit moyens, selon la loi de Little sur les files d'attentes [Lit61] :

$$d_m = b_m t_m. \tag{1.1}$$

Les GPU s'appuient sur le parallélisme de données pour masquer les latences. Le nombre de threads en vol n_{threads} est proportionnel au nombre de transactions mémoire en vol :

$$n_{\text{threads}} = \frac{d_m}{p \cdot d_{\text{trans}}}, \tag{1.2}$$

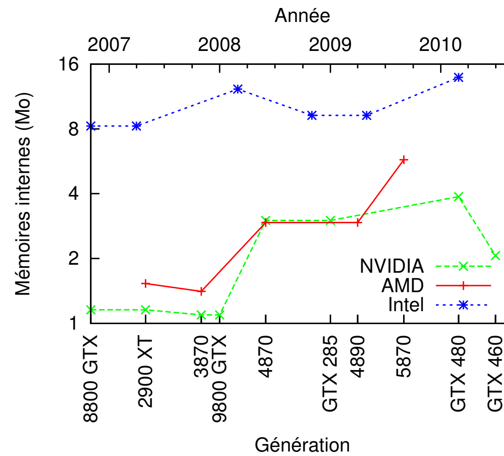


FIGURE 1.8 – Évolution de la capacité des mémoires internes des GPU et CPU.

où d_{trans} est la taille d'une transaction et p le nombre de transactions en vol par thread, obtenues en exploitant le parallélisme d'instructions.

Or, si chaque thread nécessite un contexte d'exécution de taille d_c, alors la taille totale des mémoires internes sera de :

$$d_{\text{total}} = \frac{d_c b_m t_m}{p \cdot d_{\text{trans}}}. \tag{1.3}$$

La taille du contexte d_c est dépendante de la complexité de l'application, du taux de partage entre threads et du ratio entre puissance de calcul et débit mémoire.

Par exemple, considérons les mesures de débit effectuées par Volkov sur GT200 en fixant la taille de transaction d_{trans} à 8 octets et en faisant varier p et n_{threads} [Vol10b]. Les résultats obtenus montrent que pour atteindre 85 % du débit crête avec $p = 1$, il est nécessaire de maintenir au moins 9600 threads en vol. Dans ces conditions, chaque thread doit se limiter à un contexte de 250 octets.

L'exploitation du parallélisme de données a donc un impact négatif sur la localité.

Des solutions possibles sont d'extraire davantage de parallélisme d'instructions pour augmenter p, augmenter la granularité des communications d_{trans}, ou ajuster dynamiquement l'équilibre entre le nombre de threads n_{threads} et la taille de leur contexte d'exécution d_c. Nous verrons par la suite que ces trois solutions sont employées par les GPU actuels.

La mémoire interne est majoritairement constituée de bancs de registres. Ces mémoires doivent fournir une bande passante importante, mais tirent avantage de la régularité de l'ordonnancement des threads. Nous étudierons une telle structure plus en détail dans la section 2.4.3.

Pour permettre des communications à faible latence entre threads, les cœurs disposent de mémoires locales contrôlées de manière logicielle.

Enfin, des caches permettent d'assurer les rôles combinés de mémorisation de données locales, de tampon d'envoi et de réception sur la mémoire externe et de tampon de communication entre threads. Ils permettent aussi la mémorisation de tableaux locaux accédés avec indirection ou de taille dynamique telle que la pile d'appel. Leur gestion est assurée de manière

transparente par le matériel, et ils permettent d'unifier et d'équilibrer les ressources mémoire entre chaque type d'usage. Cependant, ils ont un coût en énergie, latence et surface supérieur aux mémoires spécialisées, à bande passante équivalente.

L'architecture Fermi et le projet Larrabee affichent une tendance au déplacement des données locales depuis le banc de registres vers le cache L1.

1.2.8 Unités spécialisées pour le rendu graphique

Rastérisation L'étape de *rastérisation* consiste à identifier les fragments recouverts par une primitive. Les environnements graphiques imposent que l'ordre de traitement des primitives soit respecté. En effet, les opérations de fusion des fragments en cas de recouvrement ne sont pas nécessairement associatives, et le résultat dépendra de l'ordre de traitement des primitives.

Cette opération est donc effectuée de manière séquentielle sur la plupart des GPU, une primitive après l'autre. La difficulté de parallélisation a justifié son implémentation en matériel sur tous les GPU commercialisés. Fermi parallélise cette opération en découpant l'espace de la zone de rendu en tuiles [Tri10]. Larrabee est décrit comme suivant une approche hiérarchique en logiciel [Abr09].

Filtrage de textures Le placage de texture consiste à disposer une image, ou *texture* sur la face d'un objet géométrique, en tenant compte de la perspective. On appelle *texels* les éléments de couleur dans le plan de la texture, et pixels ceux du plan de l'écran.

La problématique consiste à calculer la couleur de chaque pixel en fonction du ou des texels le recouvrant. Un filtre est appliqué afin d'éviter les effets de crénelage ou de repliement. Le filtrage est assisté par une unité matérielle sur tous les GPU car jugé trop coûteux à réaliser en logiciel [For09].

Les textures telles qu'elles sont exposées dans les langages graphiques et GPGPU sont paramétrables par plusieurs attributs. Une texture peut être définie comme un tableau uni-, bi- ou tri-dimensionnel, et sera respectivement indexée au moyen d'une, deux ou trois coordonnées de textures. Chaque texel contient un vecteur de 1 à 4 composantes, chacune d'elles pouvant être représentée dans l'un des nombreux formats virgule fixe ou virgule flottante disponibles.

Lorsqu'aucun filtre n'est appliqué, l'unité de filtrage de textures retourne la valeur du texel le plus proche des coordonnées calculées.

Le mode de filtrage de base disponible en matériel est le filtrage bilinéaire. Il effectue une interpolation entre les quatre texels encadrant les coordonnées, selon le schéma représenté figure 1.9.

Soit T une texture bi-dimensionnelle de $N \times M$ texels accédée par des coordonnées de textures u et v telles que $u \in [0, N]$ et $v \in [0, M]$. L'unité de filtrage renvoie la valeur V suivante :

$$\begin{aligned} V = \; & (1-\alpha)\cdot(1-\beta) & T[i,j] \\ & +(1-\alpha)\cdot\beta & T[i+1,j] \\ & +\alpha\cdot(1-\beta) & T[i,j+1] \\ & +\alpha\cdot\beta & T[i+1,j+1], \end{aligned} \tag{1.4}$$

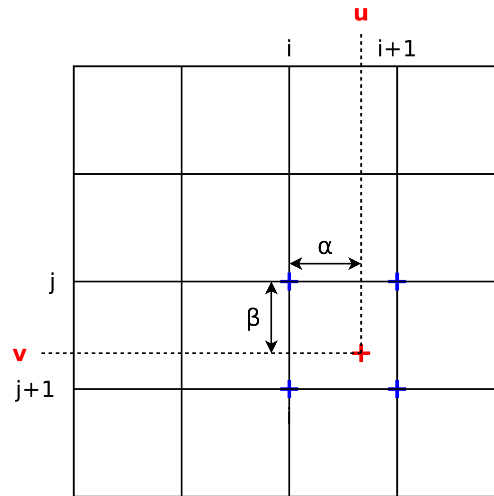


FIGURE 1.9 – Filtrage bilinéaire d'une texture à deux dimensions.

avec $i, j \in \mathbb{Z}$ et $\alpha, \beta \in [0, 1[$ calculés tels que $u = i + \alpha + \frac{1}{2}$ et $v = j + \beta + \frac{1}{2}$ [3].

De nombreux autres modes de filtrage sont supportés de manière logicielle ou matérielle. Nous nous limiterons ici à l'utilisation du filtrage bilinéaire 2D à coordonnées non normalisées.

Cache de textures Le rôle d'un cache traditionnel est d'exploiter la localité spatiale et la localité temporelle offertes par un code séquentiel. Il part du principe qu'un thread accédera successivement à des adresses égales ou proches dans un intervalle de temps court.

Dans le cas d'une architecture parallèle à grain fin, la localité se manifeste entre threads adjacents. Par exemple, un *fragment shader* pourra appliquer une texture pour les informations de couleur, une autre pour les informations de relief (*displacement map*), et une troisième pour les informations d'éclairement (*light map*). Chacune de ces textures réside à un emplacement différent dans la mémoire : il n'existe pas de localité entre les accès successifs aux textures. En revanche, les pixels adjacents accéderont à des texels proches dans chacune des textures. Les textures étant généralement bi-dimensionnelles, le mode d'adressage et la structure du cache sont adaptés à la localité spatiale dans un espace 2D.

Dans ce contexte, il peut être avantageux de maintenir une synchronisation artificielle entre les warps pour limiter la compétition sur le cache. Des techniques basées sur l'ajout de barrières de synchronisation ont été développées pour améliorer l'efficacité du cache de textures [MLC$^+$08].

La gestion de la cohérence mémoire s'effectue de manière manuelle, par invalidation explicite des caches. Cette approche se justifie par la taille réduite des caches et la faible localité temporelle. Dans la pratique, la mémoire de textures est considérée comme accessible en lec-

3. Le décalage d'un demi-texel appliqué aux coordonnées est dû au problème classique des piquets de clôture : les coordonnées de texture u et v peuvent atteindre respectivement les valeurs N et M, alors que le dernier texel est situé en $(N - 1, M - 1)$.

ture seule pendant tout le temps d'exécution d'un noyau de calcul sur le GPU.

ROP L'unité ROP se charge de l'arbitrage entre les fragments situés aux mêmes coordonnées. Le test d'occlusion détermine quel fragment masque l'autre dans le cas de fragments opaques. Le fragment le plus proche est déterminé au moyen d'un tampon de profondeur (*Z-buffer*) contenant la valeur de profondeur de la dernière valeur écrite dans la zone de rendu aux coordonnées considérées.

Elle gère également les effets de transparence, en permettant d'effectuer une interpolation paramétrable entre la couleur en mémoire et la couleur du fragment à fusionner. L'équation d'interpolation est de la forme suivante :

$$x' = \alpha x + \beta y \tag{1.5}$$

où α et β sont des vecteurs constants, x est la couleur précédemment en mémoire, y la couleur à écrire et x' la nouvelle valeur de couleur.

1.3 Enjeux

Si les générations actuelles de GPU partagent un certain nombre de points communs, tels que ceux que nous avons abordés dans la section précédente, les GPU des différents constructeurs possèdent chacun leurs spécificités propres. Ainsi, il n'existe pas actuellement de consensus sur la direction à privilégier en ce qui concerne l'évolution future de ces architectures.

1.3.1 Approche statique ou approche dynamique ?

Dimensions de parallélisme Tout comme les processeurs multi-cœurs, les GPU tirent parti du parallélisme à plusieurs niveaux de granularité. Considérons table 1.2 une comparaison quantitative des mécanismes d'extraction du parallélisme mis en œuvre dans les différents GPU ainsi qu'un CPU. Les types de parallélisme exploités sont le parallélisme de tâches (TLP), le parallélisme de données (DLP, prépondérant dans le cadre des GPU), et le parallélisme d'instructions (ILP). Les architectures GPU considérées sont celles que nous avons abordé section 1.2.1. L'architecture Intel Nehalem est présente dans les CPU Intel Core i7 en 2009.

Du grain le plus gros au grain le plus fin, les mécanismes permettant l'extraction de ces formes de parallélisme sont :

- l'exécution concurrente de tâches sur plusieurs cœurs ou SM dotés de mécanismes d'exécution indépendants (CMP),
- le multithreading à basculement sur événement (SoEMT), consistant à basculer d'un warp à l'autre lorsqu'un événement à latence longue survient,
- le multithreading à grain fin (FGMT), consistant à entrelacer l'exécution de différents warps au sein des unités d'exécution, permettant de masquer les latences courtes,
- l'exécution en pipeline d'une même instruction sur des données différentes,
- l'exécution en parallèle d'une même instruction sur des données différentes (SIMD),
- le démarrage en parallèle de plusieurs instructions indépendantes par cycle (degré superscalaire ou VLIW).

TABLE 1.2 – Comparaisons des dimensions de chaque mécanisme parallèle des architectures GPU.

Architecture	CMP	Multithreading		Vecteurs implicites		V. explicites	Degré SS
		SoEMT	FGMT	Pipeline	SIMD	SIMD	
Tesla	30	1	32	4	8	1	1
Fermi	16	1	48	2	16	1	2/3+1
Evergreen	20	6	4	4	16	1	5
GenX	12	1	4	1 ou 2	8 ou 2	1 ou 4	1
Larrabee	24	8	4	1	1	16	1+1
Nehalem	4	1	2	1	1	4	3+3
Parallélisme	TLP			DLP			ILP

TABLE 1.3 – Classification des mécanismes d'extraction du parallélisme entre statique et dynamique.

Architecture	Multithreading (TLP)	Vecteurs (DLP)	Superscalaire (ILP)
Tesla/Fermi	Dynamique	Dynamique	Mixte
Evergreen	Mixte	Dynamique	Statique
GenX	Dynamique	Mixte	Statique
Larrabee	Mixte	Statique	Mixte
Nehalem	Dynamique	Statique	Dynamique

Ces mécanismes peuvent également se partager entre techniques statiques et techniques dynamiques. La table 1.3 présente un tel classement. Ainsi, si l'on retrouve les mêmes niveaux de granularité de parallélisme dans toutes les architectures, les approches suivies pour extraire le parallélisme varient entre méthodes statiques et méthodes dynamiques. Considérons maintenant tour à tour chacune des trois granularités de parallélisme.

Ordonnancement des warps Maintenir plusieurs warps en parallèle par SM en les faisant partager les mêmes unités de calcul permet de masquer les latences sans pour autant affecter le débit d'exécution crête. Ces latences sont de deux types : des latences courtes et prévisibles pour les instructions de calcul, et des latences longues et non-déterministes pour les lectures en mémoire.

Une technique consiste à mettre en œuvre des mécanismes distincts pour masquer les latences longues et les latences courtes. Ainsi, les architectures d'AMD et Larrabee se basent sur du multithreading à basculement sur événement pour masquer les latences mémoire. Ce choix vise à effectuer un compromis dynamique entre parallélisme et localité.

En effet, il est possible de partager des ressources entre les warps inactifs. Considérons un exemple sur la figure 1.10. Quatre warps W1 à W4 sont répartis en deux groupes G1 et G2. Les warps de deux groupes différents peuvent être exécutés simultanément par du multithreading à

grain fin. Le basculement entre deux warps d'un même groupe intervient lors des opérations à latence longue. Comme ces basculements de contextes sont uniquement permis en des points connus du compilateur, il est possible de scinder le contexte associé aux warps (registres) entre une partie privée et une partie partagée. La partie privée survit aux basculements de contexte, tandis que la partie partagée est écrasée par les données du ou des autres warps du groupe.

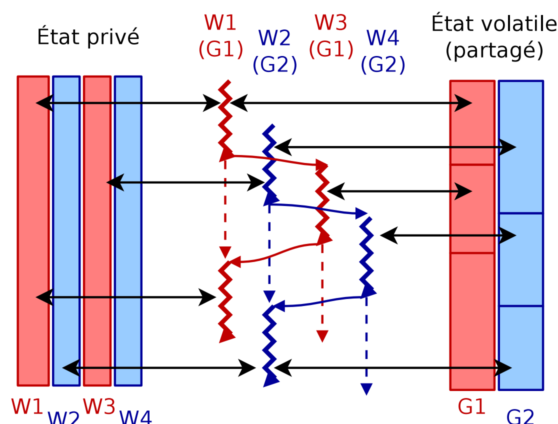


FIGURE 1.10 – Exemple d'ordonnancement à deux niveaux.

Ce type de multithreading à basculement sur événement nécessite l'intervention du compilateur sur toutes les architectures considérées. Il peut éventuellement être réalisé de manière logicielle comme sur Larrabee. Les décisions se rapportant à la sauvegarde du contexte d'exécution lors des basculements de contexte incombent au compilateur et sont établies de manière statique.

À l'inverse, le multithreading à grain fin est géré entièrement en matériel, de manière dynamique. Notons que les architectures NVIDIA se reposent uniquement sur ce multithreading dynamique, tandis que les autres architectures GPU suivent une approche mixte.

Extraction de vecteurs Nous effectuons une distinction dans la table 1.2 entre les mécanismes basés sur des vecteurs implicites de type SIMT et les vecteurs explicites visibles au niveau architectural. Les vecteurs implicites sont contrôlés par le matériel de manière dynamique, tandis que l'exploitation des vecteurs explicites nécessite un travail de vectorisation préalable de la part du compilateur voire du programmeur. Les GPU basés sur le SIMT suivent l'approche dynamique, tandis que les CPU et Larrabee dépendent d'une vectorisation statique.

Parallélisme d'instructions Les GPU exploitent également du parallélisme d'instructions. Les architectures NVIDIA et Intel Larrabee sont de type superscalaire dans l'ordre, tandis que l'architecture AMD est un processeur de type VLIW. Dans ce dernier cas, l'effort de conception est déporté sur le compilateur [Leu02].

Une technique appliquée par tous les GPU pour recouvrir les latences mémoire est de rendre les lectures de textures asynchrones par rapport aux calculs, en permettant à un warp

de continuer à exécuter des calcul non-dépendants d'une lecture mémoire en cours. Ce recouvrement peut être explicite dans le jeu d'instruction comme dans le cas des architectures AMD [AMD09b], ou être géré dynamiquement par un *scoreboard* à l'instar des architectures NVIDIA [CMOS08].

De manière plus originale, le jeu d'instructions d'Evergreen présente des aspects se rapprochant d'architectures *transport triggered* [Cor97]. Contrairement à un VLIW traditionnel où chaque instruction d'un groupe gouverne une unité indépendante, les unités de calcul d'Evergreen sont disposées en cascade (figure 1.11). Les connexions entre ces unités sont paramétrables par des champs dans le mot d'instruction [AMD09b]. Les chemins de données représentés en gras sur la figure 1.11 sont les résultats exacts des multiplications flottantes. Des unités de calcul de minimum/maximum, négation et valeur absolue (non présentes sur le schéma) peuvent également être utilisées. Ainsi, des opérations arithmétiques relativement complexes peuvent être décrites au moyen d'une unique instruction.

De tels mécanismes d'extraction de l'ILP sont rendus possibles par la tolérance à la latence des instructions et à la localité dans le cache d'instructions qu'apporte le modèle SIMT.

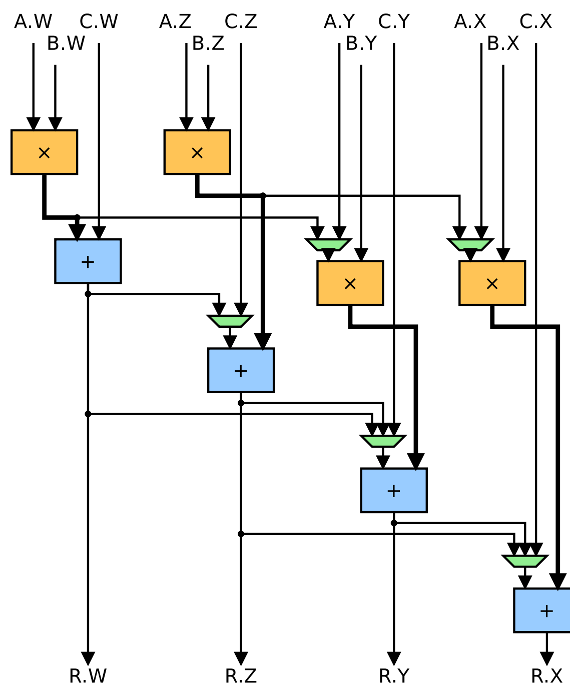


FIGURE 1.11 – Organisation générale de l'unité virgule flottante VLIW de l'architecture Evergreen.

Enfin, notons que les frontières entre les trois formes de mécanismes d'extraction du parallélisme que nous avons présentées peuvent devenir floues. Ainsi, la notion de warp pourrait être amenée à s'effacer pour laisser place à des techniques plus flexibles pour grouper des instructions depuis différents threads, à mi-chemin entre le multithreading traditionnel et le SIMT [Fun08, Gle09].

1.3.2 Matériel spécialisé ou implémentation logicielle ?

Les spécificités des GPU actuels sont-elles amenées à se résorber, où bien seront-elles intégrées à des processeurs généralistes ? La tendance ayant accompagné l'évolution des GPU jusqu'à aujourd'hui est de remplacer des unités spécialisées en unités programmables. Cette tendance à la généricité s'illustre par l'apparition des vertex shaders et des fragment shaders puis leur unification, ainsi que les autres types de shaders introduits par Direct3D 10 et 11.

De même, des unités spécialisées sont délaissées au profit d'unités génériques pour permettre la consolidation des ressources de calcul, même lorsque cela n'est pas requis par l'API graphique. Par exemple, l'architecture Evergreen effectue les interpolations d'attributs en utilisant les unités de calcul généralistes [AMD09b], contrairement aux générations antérieures qui disposaient d'unités dédiées [AMD09c].

Le projet Larrabee met également l'accent sur cette approche de généralisation, en proposant de réaliser la plupart des étages du pipeline de rendu graphique de manière logicielle, y compris la rastérisation [SCS+08]. Seul le filtrage de textures bénéficie d'une unité matérielle spécialisée.

Les avantages de cette approche sont une souplesse de programmation accrue et la possibilité d'un équilibrage de charge dynamique adapté à chaque application. Cependant, le rendement énergétique d'une unité dédiée reste bien meilleur que celui d'une implémentation logicielle.

La tendance pourrait donc s'inverser, alors que la consommation énergétique devient un enjeu plus important que la surface de silicium. Notons que l'ensemble des GPU actuels intègrent des processeurs spécialisés pour le décodage vidéo. Les GPU destinés au marché embarqué disposent également d'architectures spécialisés avec notamment des vertex shaders et fragment shaders séparés [NVI10c].

Conclusion

Nous avons pu constater que les modèles de programmation pour le GPGPU étaient issus à la fois des modèles employés par la communauté graphique (shaders) et des modèles des machines parallèles pour le calcul haute performance. Ils partagent certaines caractéristiques avec leurs prédécesseurs, telles que la compilation à la volée des shaders ou les mémoires locales gérées par le programmeur.

L'architecture des GPU reflète également ce double héritage, qui s'illustre par l'irruption de mécanismes issus des architectures parallèles dans le pipeline de rendu graphique. Cette architecture est optimisée pour maximiser le débit des unités de calcul et de la mémoire, en tirant parti du parallélisme de données, du faible partage, de la localité et de la régularité des applications. L'apport des architectures graphiques se manifeste par la présence d'unités spécialisées, dédiées à des tâches particulières du rendu graphique, qui pourraient ouvrir la voie au développement d'autres types d'unités spécialisées, plus efficaces en énergie que les processeurs conventionnels.

L'évolution rapide rencontrée dans le domaine du GPGPU au cours de ces cinq dernières années, aussi bien au niveau matériel que logiciel, montre que cette plate-forme n'est pas encore figée à l'heure actuelle. Elle offre des opportunités pour proposer de nouvelles idées de modèles

de programmation et d’architectures.

Chapitre 2

Étude de cas : l'architecture Tesla

Nous avons entrevu au chapitre 1 un aperçu des architectures GPU des différents constructeurs. Nous nous intéresserons dans ce chapitre à étudier de manière plus précise un exemple, l'architecture NVIDIA Tesla. Notre étude détaillée de cette famille de GPU répond à trois problématiques.

D'une part, l'optimisation fine des applications GPGPU nécessite une connaissance détaillée de l'architecture matérielle et des facteurs susceptibles de limiter les performances. Notre étude répond à ce besoin en fournissant plus de détails que ceux qui sont disponibles dans la documentation officielle.

D'autre part, étudier un processeur actuel nous permet de mieux comprendre les enjeux de conception auxquels ses architectes ont dû faire face. Cette expérience qui est rarement publiée est cruciale pour orienter la recherche sur les problèmes actuels.

Enfin, l'architecture Tesla nous fournit une base de travail pour explorer différentes idées architecturales. Le fait de partir d'une architecture existant réellement nous garantit que le modèle de base est représentatif. Ainsi, nous tirerons parti de l'analyse présentée ici pour concevoir un simulateur de GPU dans le chapitre 4.

En tant que partie intégrante de la plate-forme CUDA, l'architecture Tesla a eu une influence majeure sur le développement du GPGPU, au point de devenir une norme *de facto*. En novembre 2009, NVIDIA recensait ainsi près de 700 travaux académiques et industriels basés sur CUDA et donc Tesla [Cud]. Cela en fait l'architecture privilégiée à analyser, modéliser et étendre.

La documentation officielle présente un modèle simplifié de l'architecture, mais ne décrit pas son fonctionnement précis. Un article des concepteurs fournit également un aperçu plus détaillé [LNOM08]. S'il est défendable que ces ressources soient suffisantes pour développer des applications ciblant l'architecture Tesla, il n'en va pas de même pour les travaux visant à analyser et modéliser ce type d'architecture.

J'ai mis au point des tests entre 2007 et 2010 pour comprendre plus en profondeur le fonctionnement de ces GPU, et en inférer leurs enjeux de conception. D'autres travaux expérimentaux tentent de reconstituer les caractéristiques de certains GPU de l'architecture Tesla

TABLE 2.1 – Révisions de l'architecture.

Compute model	Fonctionnalités majeures
1.0	Architecture de base Tesla
1.1	Instructions atomiques en mémoire globale
1.2	Atomiques en mémoire partagée, instructions de vote
1.3	Double précision

au moyen de tests synthétiques. Volkov et Demmel [VD08] ont étudié les latences et les débits des transferts mémoire et des unités de calcul pour optimiser des algorithmes d'algèbre linéaire, sur plusieurs GPU NVIDIA du G80 au GT200. Wong et al. [WPSAM10] ont fait une étude détaillée des hiérarchies mémoires sur le GT200. Ces travaux offrent une compréhension qualitative des structures architecturales internes du GPU.

Les projets de pilote graphique libre Nouveau [Nou] et de pilote GPGPU de Pathscale [Psc] comportent un effort de rétro-ingénierie sur les architectures NVIDIA pour reconstituer leurs spécifications. Ces tests sont orientés vers l'interface entre le matériel et le pilote plutôt que sur le fonctionnement interne du GPU. Les résultats obtenus ne sont pas publiés.

Nous tenterons ici d'offrir une vision d'ensemble, qui combine un récapitulatif des données présentes dans la littérature avec nos propres résultats expérimentaux et hypothèses. Nous nous attacherons à étudier les aspects liés à la performance, mais aussi à la consommation d'énergie.

Nous commencerons par une description au niveau architectural dans la section 2.1. Nous offrirons ensuite une vue d'ensemble de l'organisation des GPU Tesla section 2.2. Nous considérerons en détail chacun des principaux composants : le frontal de gestion des tâches (section 2.3), les cœurs de calcul ou SM (section 2.4) et la hiérarchie mémoire (section 2.5). Enfin, nous étudierons plus particulièrement les aspects liés à la consommation et dissipation énergétique section 2.6.

2.1 Jeu d'instructions

Au niveau architectural, les unités de calcul de tous les GPU Tesla reconnaissent le même jeu d'instructions de base, mais des extensions sont introduites ou retirées en fonction des révisions de l'architecture ou de la gamme de produit. Ces révisions de l'architecture sont désignées sous le nom de *Compute Model* par NVIDIA et sont représentées sous forme d'un numéro de version. Les versions associées à l'architecture Tesla sont listées table 2.1.

La documentation officielle du jeu d'instructions Tesla n'est pas publiquement disponible à ce jour. Cependant, la majorité du jeu d'instructions a été retrouvée par analyse des binaires produits par le compilateur NVIDIA, notamment par l'intermédiaire du projet *decuda* dès 2007 [vdL], ou à *nv50dis* associé au projet *Nouveau* [Env] à partir de 2009. J'ai contribué à ces projets sur les aspects arithmétiques.

Ce jeu d'instructions suit le modèle d'exécution SIMT. Il partage de nombreuses similitudes avec les langages intermédiaires issus de la compilations des shaders de Direct3D (assem-

bleur HLSL), OpenGL (ARB Fragment/Vertex/...Program [ARB]) ou CUDA (PTX [NVI10d]).

Il s'agit d'un jeu d'instructions à 4 adresses. Il est orthogonal et offre un niveau d'abstraction élevé : il n'expose pas d'aspect lié à l'ordonnancement des instructions. Le modèle d'exécution de chaque thread est strictement séquentiel. La plupart des instructions sont codées sur 64 bits, tandis que certaines peuvent être codées sur 32 bits. Les instructions sur 64 bits devant être alignées en mémoire, les instructions sur 32 bits sont toujours groupées par paires.

Pour conserver le modèle de programmation SIMT et abstraire la largeur des warps, Tesla n'expose pas d'instruction de type *swizzle* ou autres modes de communication directe entre voies SIMD, à l'exception d'une instruction de réduction de booléens (vote) introduite avec la révision 1.2, et d'un support minimal pour le calcul des dérivées partielles entre pixels adjacents (instructions `dsx` et `dsy` de Direct3D [Mica]). L'aspect multithreading simultané est également masqué, chaque thread possédant son propre ensemble de registres architecturaux privés.

Plusieurs types de registres architecturaux spécialisés sont accessibles :

- les registres généraux,
- les registres de drapeaux,
- les registres d'adresse.

Les registres de drapeaux permettent de conserver le résultat d'une comparaison ou la classification d'un résultat (zéro, négatif, retenue, débordement). Ces registres sont vectoriels, et peuvent contenir une valeur différente pour chaque voie SIMD. Associés à un code de condition, ils peuvent être utilisés par les instructions de branchements, mais aussi par la plupart des autres instructions. Les instructions peuvent être prédiquées de manière généralisée par un registre de drapeaux et un code de condition. Ce système agit en complément de la prédication implicite associée au SIMT. La souplesse offerte vient au prix de 10 bits réservés dans le mot d'instruction [vdL]. Ainsi, le jeu d'instructions Tesla ne semble pas significativement optimisé pour la densité du code.

Mémoire Au niveau architectural, l'espace mémoire de Tesla est hétérogène, divisé en espaces séparés. Chaque espace est accessible par des instructions de lecture-écritures spécifiques. La décision de placement des données ne peut donc être faite qu'au moment de la compilation. En particulier, le compilateur doit être capable d'inférer statiquement l'espace mémoire désigné par chaque pointeur.

Ces espaces logiques comprennent :

- la mémoire de constantes,
- la mémoire de textures,
- la mémoire globale,
- la mémoire locale,
- la mémoire partagée.

Les mémoires de textures et de constantes sont héritées des langages de shaders pour la programmation graphique, qui y font référence explicitement. Effectuer la distinction entre ces espaces mémoire au niveau du jeu d'instructions est une décision naturelle. Elle permet de séparer les chemins d'accès à la mémoire en dimensionnant de manière adéquate chaque unité d'accès. Le compilateur n'a pas à faire de traduction entre les espaces mémoire du modèle de

programmation et les espaces mémoire de l'architecture. En revanche, la séparation entre les mémoires partagée, globale et locale destinées principalement au calcul généraliste nécessite un effort supplémentaire de la part du compilateur (en C pour CUDA) ou du programmeur (en OpenCL).

En contrepartie, cette distinction permet de séparer explicitement les zones mémoires en lecture seule (constantes, textures), locale à un thread (locale), ou à latence courte et déterministe (partagée). La latence de chaque type de mémoire peut être estimée précisément de manière statique. Des accès à des zones distinctes peuvent également être réordonnées sans risque d'interférence. Le compilateur bénéficie de ces informations supplémentaires pour effectuer le choix et l'ordonnancement des instructions. Ainsi, on observe que le compilateur CUDA déplace les lectures en mémoire globale au plus tôt, quitte à nécessiter plus de registres, tandis qu'à l'inverse il fusionne les lectures en mémoire partagée avec des instructions de calcul, voire duplique ces lectures pour réduire l'utilisation des registres.

2.2 Organisation générale

La partie calcul est composée d'une hiérarchie de cœurs [LNOM08, Kan08]. Au niveau le plus élevé, le GPU contient jusqu'à 10 TPC (*texture / processor clusters*) connectés aux autres composants par un réseau d'interconnexion de type *crossbar*. Chaque TPC contient une unité d'accès à la mémoire.

Domaines d'horloge Le GPU est divisé entre plusieurs domaines d'horloges, présentés table 2.2. Le temps de cycle de chaque domaine est choisi en fonction du compromis souhaité entre latence, débit, surface et consommation pour les composants matériels qu'il contient. Les unités de calcul (SP) fonctionnent à la fréquence la plus haute (fréquence SP). Le reste des SM est animé par une horloge de fréquence divisée par deux (fréquence SM). Les autres unités du GPU telles que le réseau d'interconnexion et l'ordonnanceur global fonctionnent à une fréquence indépendante, généralement inférieure (fréquence cœur). Enfin, les contrôleurs mémoire suivent l'horloge de la mémoire externe, en général de type GDDR3. Nous nous référerons par la suite aux périodes de chaque horloge par « cycles SM » ou « cycles SP » pour lever l'ambigüité.

Malgré la présence de plusieurs domaines d'horloge, les GPU Tesla disposent d'un unique domaine de tension en plus du domaine de tension de la mémoire. Ainsi, les gains d'énergie restent plus limités que sur les CPU munis de domaines de tension distincts entre les cœurs et le reste du circuit.

Souplesse et passage à l'échelle Les différentes instances de l'architecture Tesla ont suivi des évolutions progressives. La stratégie suivie consiste à introduire une nouvelle révision majeure dans le segment de haut de gamme, pour ensuite réutiliser sa micro-architecture pour les segments de milieu et d'entrée de gamme. Le nombre de TPC et de partitions mémoire est alors progressivement réduit. Des révisions mineures peuvent êtres introduites à cette occasion, mais également des pertes de fonctionnalités. Ainsi, les SM des dérivés du GT200 ne possèdent pas les unités de calcul virgule flottante Binary64. Bien que la plupart des composants peuvent être réutilisés, cela implique de concevoir et maintenir deux modèles de SM différents. La

TABLE 2.2 – Répartition des domaines d'horloge de l'architecture Tesla.
L'exemple choisi est la GeForce 9800 GX2.

Horloge	Composants	Fréquence typique
SP	Chemins de données des unités de calcul	1,5 GHz
SM	Contrôle des instructions, registres, caches internes	750 MHz
Cœur	Unités de textures, réseau d'interconnexion	600 MHz
RAM	Contrôleurs mémoire, entrées-sorties mémoire	1 GHz

TABLE 2.3 – Modèles de GPU de l'architecture Tesla.

GPU	TPC	SM/TPC	Partitions	Sous-architecture	Représentant
G80	8	2	6	1.0	GeForce 8800 GTX
G84	2	2	2	1.1	GeForce 8600 GTS
G86	1	2	2	1.1	GeForce 8500 GT
G92	8	2	4	1.1	GeForce 8800 GT
G94	3	2	4	1.1	GeForce 9600 GT
G96	2	2	2	1.1	GeForce 9500 GT
G98	1	2	1	1.1	GeForce G 100
GT200	10	3	8	1.3	GeForce GTX 280
GT215	4	3	2	1.2	GeForce GTS 260M
GT216	2	3	2	1.2	GeForce GT 240M
GT218	1	2	1	1.2	GeForce G 210M

table 2.3 liste l'ensemble des GPU dérivés de l'architecture Tesla. Au delà des différences dues au choix du GPU, il est possible de produire d'autres modèles de cartes graphiques en désactivant certains TPC et partitions mémoire et en variant les différentes fréquences. Cela permet d'améliorer les rendements de production ou réduire les autres coûts de fabrication de la carte graphique (circuit imprimé, mémoire).

Les contraintes économiques nécessitent que l'architecture soit capable de passer à l'échelle vers le bas : les dérivés doivent rester compétitifs pour un coût très réduit. Les unités et partitions mémoires doivent pouvoir être désactivées et le réseau d'interconnexion doit pouvoir être reconfiguré de manière transparente.

Cela contraste avec les objectifs de conception des calculateurs parallèles traditionnels, qui sont prévus pour passer à l'échelle vers le haut en formant des grappes de calcul ou des supercalculateurs.

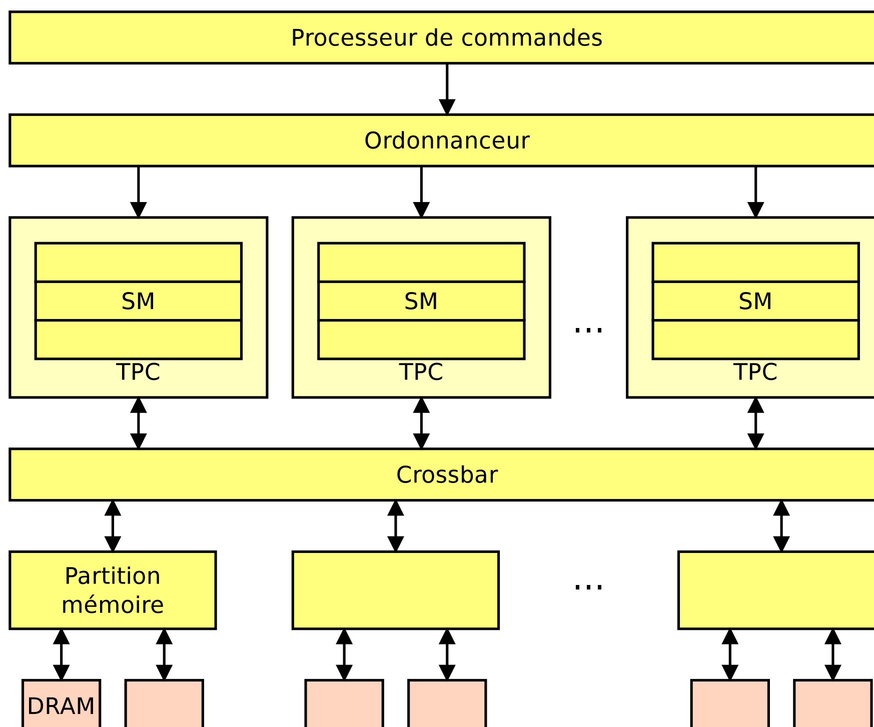


FIGURE 2.1 – Vue générale de l'architecture Tesla.

2.3 Gestion des tâches

Le Tesla étant un coprocesseur spécialisé, il n'est pas capable de fonctionner de manière autonome. Il est dirigé par un pilote s'exécutant sur le ou les CPU de l'hôte. Ce pilote contrôle l'état (registres de configuration) du GPU, lui envoie des commandes, initie des transferts de données, et peut recevoir des interruptions indiquant la terminaison d'une tâche.

2.3.1 Processeur de commandes

Le GPU doit pouvoir fonctionner de manière asynchrone par rapport aux CPU. Il est contrôlé au travers d'une file de commandes selon un schéma producteur-consommateur classique. Cette file est typiquement placée dans la mémoire du système hôte. Le pilote graphique se charge d'ajouter des commandes dans la file, tandis que le processeur de commandes du GPU retire les commandes à l'autre extrémité de la file et les exécute.

Les commandes possibles comprennent des commandes de configuration, des copies mémoire, ou des lancements de noyaux de calcul. Pour permettre à plusieurs contextes d'exécution de coexister, et pour exploiter du parallélisme en recouvrant copies mémoire et exécution sur le GPU, plusieurs files de commandes virtuelles peuvent être utilisées. Le basculement du GPU d'un contexte d'exécution à l'autre se fait par exécution de microcode sur un microcontrôleur

dédié [1].

2.3.2 Ordonnancement à gros grain

L'ordonnanceur est une unité du GPU chargée de répartir les CTA (blocs de threads décrits dans la section 1.1.2) à exécuter sur les multiprocesseurs. Une telle unité est décrite dans [NNB09]. Avant d'envoyer un signal de début d'exécution aux multiprocesseurs, il procède à l'initialisation des registres et des mémoires partagées.

En particulier, il doit initialiser :

- les arguments du noyau,
- les coordonnées du CTA,
- la taille des CTA et de la grille,
- les coordonnéees du thread à l'intérieur du CTA.

Les arguments et les coordonnées du CTA sont communs à tous les threads qu'il contient. Ils peuvent donc être placés en mémoire partagée. Ainsi, les 16 premiers octets de la mémoire partagée sont initialisés par 8 entiers de 16 bits contenant les informations de dimension. Les arguments sont placés aux adresses immédiatement supérieures. Certaines de ces données sont constantes, et en tant que telles, pourraient être passées par la mémoire de constantes. Le choix qui a été fait s'explique probablement par le coût d'une initialisation de la mémoire de constantes, qui oblige à interrompre le pipeline et invalider les caches de constantes [Bly06].

Des tests effectués en lançant des CTA de nombre et de durée d'exécution variables nous montrent que la politique d'ordonnancement des CTA est de type tourniquet (*round-robin*), et que l'ordonnanceur global attend la terminaison de tous les CTA en cours d'exécution sur le GPU avant d'ordonnancer le groupe de CTA suivant. Il n'y a pas de fonctionnement de type pipeline. Cependant, lorsqu'un TPC n'a plus de travail à exécuter, son compteur d'horloge cesse de s'incrémenter, ce qui laisse supposer la présence de *clock gating* à gros grain pour minimiser la consommation d'énergie des TPC inactifs.

La stratégie d'ordonnancement des CTA est différente entre le G80 et les G92 et GT200. Celle des GPU les plus modernes est de réaliser l'ordonnancement en largeur, en distribuant les CTA sur des TPC différents, favorisant la bande passante disponible et l'équilibrage du réseau d'interconnexion. À l'inverse, celle du G80 est un ordonnancement en profondeur. Elle distribue les CTA consécutifs en priorité sur les SM d'un même TPC.

La politique d'ordonnancement choisie est très simple, et offre une marge de progression pour des optimisations futures. Ces opportunités ont été exploitées dans l'architecture Fermi, qui offre une politique d'ordonnancement plus souple [NVI09b]. En effet, on peut considérer que l'ordonnanceur de CTA est la première unité dédiée uniquement au GPGPU à avoir été introduite dans l'architecture Tesla. Maintenir la première itération de cette unité la plus simple possible permet de minimiser la prise de risque pour le constructeur.

1. `http://nouveau.freedesktop.org/wiki/CtxInit`

2.4 SM

La capacité à masquer les latences des diverses opérations tout en conservant un débit élevé repose sur l'exploitation par les multiprocesseurs de multithreading à grain fin (FGMT). Ainsi, chaque SM du G80 et chaque SM du GT200 maintiennent respectivement 24 warps et 32 warps en vol simultanément.

Contrairement à l'approche suivie par les architectures graphiques d'AMD [AMD09c], Tesla n'utilise pas de multithreading à basculement sur évènement. Une fois encore, il s'agit probablement d'une volonté de conserver un modèle simple au niveau architectural.

Des processeurs superscalaires conventionnels gèrent le FGMT, mais se limitent à deux voire quatre threads. Au delà, du multithreading à basculement sur événement est préféré [KAO05]. Pour atteindre efficacement un tel niveau de parallélisme, l'architecture des SM est conçue dès l'origine pour le FGMT. En contrepartie, cette architecture n'est efficace que lorsque le degré de parallélisme de données est suffisant. Ainsi, lorsqu'un seul warp est en cours d'exécution, le SM exécute une instruction tous les 8 cycles SP, contre 2 cycles par instruction lorsque le SM est saturé. De plus, Tesla se limite à l'exécution de code SPMD, qui offre plus de régularité et de localité que l'exécution simultanée de threads arbitraires. Plutôt que d'ajouter de la complexité au système, le FGMT apporte ici à l'inverse de la régularité et du parallélisme qui permettent de simplifier la micro-architecture.

La figure 2.2 présente une vue fonctionnelle du pipeline d'exécution d'un SM. Les étages ne correspondent pas nécessairement aux cycles physiques.

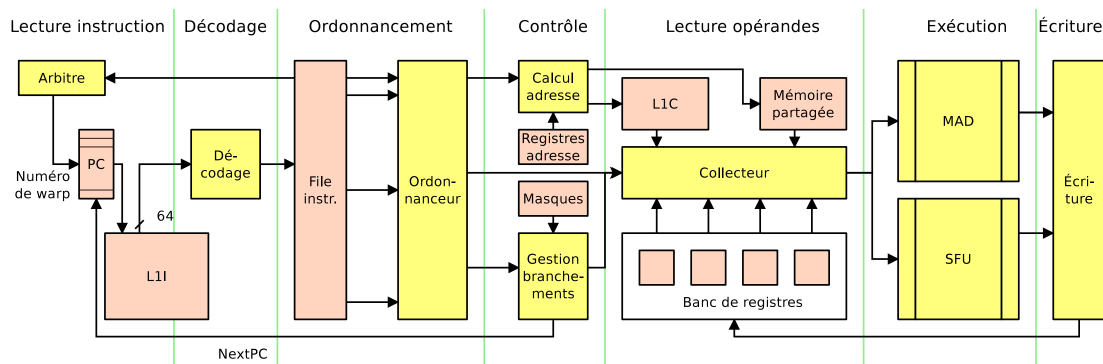


FIGURE 2.2 – Vue d'ensemble du pipeline d'exécution d'un multiprocesseur.

2.4.1 Lecture des instructions

Chaque warp dispose de son propre pointeur d'instruction, et d'un emplacement réservé dans la file d'instructions. À chaque cycle, lorsqu'un emplacement est libéré dans la file, une instruction est lue depuis le cache d'instructions de premier niveau à l'adresse désignée par le pointeur d'instruction du warp correspondant. Des tests suggèrent que les instructions sont chargées par blocs de 64 octets, ce qui représenterait 8 instructions [WPSAM10].

La taille du cache L1 d'instructions est estimée à 4 Ko et son associativité à 4 voies d'après les mêmes auteurs. Cela représente une taille modeste comparée aux 8 à 32 Ko rencontrés habituellement dans les processeurs généralistes. Qui plus est, nous avons vu dans la section 2.1 que la plupart des instructions étaient codées sur 64 bits, ce qui donne lieu à un code peu dense.

L'utilisation d'un cache aussi réduit est rendue possible par le modèle SIMT. Les warps qui sont simultanément en vie sur un SM exécutent le même programme, et leur exécution est fortement couplée par des synchronisations à grain fin. Leur progression dans le code est donc relativement homogène. Une même instruction sera exécutée sur plusieurs warps différents dans un intervalle de temps court. Ce modèle génère donc une forte localité temporelle dans la mémoire d'instructions.

Le nombre de warps concurrents a même une influence positive sur la pression sur le cache d'instructions : à débit d'exécution équivalent, maintenir plus de warps en vol augmente le taux de réutilisation des données du cache d'instructions et améliore la localité.

2.4.2 Ordonnancement des warps

Ordonnanceur Comme dans un processeur à exécution dans le désordre, un ordonnanceur se charge de sélectionner une instruction à exécuter depuis la file d'attente. Cette unité détermine l'ensemble des instructions pouvant être exécutées immédiatement sans violer de contraintes (dépendances). Parmi ces instructions, un calcul de priorité en fonction de divers critères tels que l'âge et le type de l'instruction détermine celle qui sera sélectionnée [LNOM08, MLC$^+$08].

Les calculs de dépendances sont plus simples que dans le cas d'une exécution scalaire dans le désordre. En effet, chaque warp bénéficie de ses propres registres privés. Deux instructions provenant de warps différents ne peuvent donc pas avoir de relation de dépendance par rapport aux registres. De plus, lorsque l'on considère un warp donné, ses instructions sont toujours démarrées dans l'ordre. Les mécanismes de suivi des dépendances mis en place se rapprochent de fait plus de ceux des processeurs pipelinés à exécution dans l'ordre.

Prédication Les dépendances entre instructions par rapport aux registres sont tenues à jour au moyen d'un *scoreboard*. Pour accepter un grand nombre de warps et de registres, tout en bénéficiant de l'indépendance entre warps, le scoreboard est indexé par les numéros de warps plutôt que par les numéros de registres comme dans un circuit conventionnel [CMOS08].

Nous avons noté que les instructions de calcul pouvaient être prédiquées par un registre de drapeaux et un code de condition. La prédication est utilisée sur des processeurs haute performance comme mécanisme spéculatif pour éliminer des dépendances de contrôle. L'exécution de l'instruction est démarrée systématiquement avant même que la valeur du prédicat soit connue, mais son résultat n'est pas écrit dans les registres architecturaux si le prédicat s'avère faux par la suite.

Sur une architecture SIMT, la prédication répond à des besoins différents. Le prédicat est un vecteur, et il est nécessaire pour offrir un contrôle différencié entre chaque thread d'un warp tout en conservant un mode d'exécution SIMD.

Pour déterminer à partir de quel étage du pipeline la valeur du prédicat est connue, j'ai effectué des mesures de performance. Comparons le débit d'exécution d'une instruction prédiquée négativement (pour tous les threads du warp) par rapport à une instruction non prédiquée.

TABLE 2.4 – Nombre de cycles SP par instruction par warp pour une opération MAD prédiquée sur le G80, le G92 et le GT200, suivant la valeur prise par le prédicat.

Predicat	G80	G92	GT200
Vrai	4,75	4,29	4,30
Faux	2,38	2,39	2,36

Les résultats sont présentés table 2.4. Nous observons que le débit des instructions prédiquées à faux est supérieur au débit crête des unités d'exécution (CPI de 4 pour MAD). Cela montre que ces instructions sont éliminées avant leur exécution.

Nous mesurons une latence de 15 cycles SM entre l'exécution d'une instruction `set` écrivant dans un registre de drapeaux et une instruction prédiquée par une condition sur ces drapeaux, contre 4 cycles sans cette dépendance. Cette latence correspond au temps d'exécution de la première instruction (10 cycles), et du lancement de la deuxième et lecture des drapeaux (5 cycles). Ce test confirme que le processeur n'effectue pas de spéculation sur la valeur du prédicat.

En revanche, en l'absence de dépendance sur les registres de drapeaux, le surcoût de la prédication est nul : une instruction prédiquée par une condition positive nécessitera le même temps d'exécution qu'une instruction non prédiquée.

De même, un saut conditionnel offre toujours une latence de 16 cycles que le branchement soit pris ou non. Cela montre qu'aucune prédiction de branchement statique ou dynamique n'est faite. Cette décision est certainement basée sur l'hypothèse qu'il y a toujours au moins une instruction susceptible d'être exécutée dans la file d'attente grâce au maintien de nombreux warps en vol. Dans ces conditions, mettre en œuvre des mécanismes spéculatifs pour tirer partie du parallélisme d'instructions n'est pas judicieux, car cela affecterait négativement la consommation et le débit en cas d'erreur de prédiction sans apporter d'avantage notable lorsque la prédiction est correcte.

Le pipeline d'exécution peut rencontrer des aléas, tels qu'un conflit de bancs en mémoire partagée ou dans le cache de constantes. Ceux-ci ne sont pas détectés *a priori* par l'ordonnanceur. Lorsqu'un tel aléa intervient, l'instruction fautive poursuit son exécution dans le pipeline. Les voies SIMD ayant rencontré un conflit sont masquées. Ainsi, le déroulement du pipeline n'est pas interrompu. L'instruction fautive est ensuite remise en attente dans la file d'instructions avec un masque mis à jour pour refléter le travail restant à exécuter [LO09]. Le fait qu'au moins un thread du warp soit servi à chaque nouvelle exécution de l'instruction garantit l'absence d'interbloquage. La latence mesurée d'un accès en mémoire de constantes est de $10n_c$ cycles SM, et celle d'un accès en mémoire partagée est $8n_c$ cycles, pour n_c le nombre de conflits de banc. En revanche, le débit effectif de chacune des mémoires reste de $16/n_c$ mots par cycle SM. Cette méthode a donc un coût en latence, mais se contente d'un pipeline d'exécution non bloquant, ce qui simplifie significativement le contrôle.

Prédication en SIMT Le matériel doit maintenir l'illusion d'un fonctionnement de type MIMD sur des unités SIMD. Lorsque les threads d'un warp prennent une branche différente dans le code, le SM séquentialise l'exécution de chaque branche prise. La liste des branches restant à parcourir est maintenue dans une pile. D'après les tests du projet Nouveau, jusqu'à douze entrées par warp du haut de la pile sont maintenues dans un cache dédié, tandis que le reste est conservé dans la hiérarchie mémoire [2].

Les informations de reconvergence sont également mémorisées dans la même pile. Des jetons associés aux entrées de la pile permettent de les différencier des branches [CL08]. Les 32 bits de chaque entrée du cache contiennent un masque, un pointeur vers la mémoire d'instructions et le jeton qui indique son type.

Les mécanismes d'ordonnancement des warps et des threads constituent le point central d'un processeur SIMT. Ils maintiennent l'abstraction du modèle d'exécution SIMT. La prédication non-spéculative qui est utilisée ici est un moyen peu coûteux de profiter de l'efficacité d'un mode d'exécution SIMD sur les applications régulières tout en conservant la flexibilité d'un processeur MIMD. Nous aborderons d'autres mécanismes d'ordonnancement de threads en SIMT dans la section 5.3.

2.4.3 Banc de registres

Le G80 et le GT200 disposent respectivement de 256 et 512 registres architecturaux vectoriels de 32×32 bits par SM. Ces registres représentent respectivement 512 Ko et $1,9$ Mo sur l'ensemble de la puce. Or, les mémoires à plusieurs ports de lecture et d'écriture tels que les bancs de registres classiques sont particulièrement coûteuses en surface et en consommation : leur surface évolue de manière quadratique et leur consommation de manière cubique avec le nombre de ports [RDK+00].

Pour atteindre de telles capacités tout en conservant un coût en surface et en consommation raisonnable, le nombre de ports de chaque banc de registres doit rester réduit. Le banc de registres de Tesla est décomposé en sous-bancs pour permettre les accès simultanés à plusieurs opérandes. Un arbitre se charge alors de gérer les accès concurrents, qui peuvent dès lors générer des conflits.

Wong et al. suggèrent une organisation en 64 bancs de 128 registres de 32-bit sur le GT200 [WPSAM10]. Une telle organisation permet de placer les bancs de registres plus près des unités de calcul et d'employer du *clock-gating* à grain fin sur les registres des threads inactifs. En revanche, elle nécessite de répliquer les décodeurs d'adresse.

En effet, les registres architecturaux sont vectoriels : tous les threads actifs d'un warp accèdent au même numéro de registre au même cycle. De fait, les photographies du *die* du GT200 présentent seulement 16 bancs par SM, probablement de taille 128×256 bits chacun [Was08]. Ils occupent environ $0,78\ \mathrm{mm}^2$ dans le processus de fabrication 65 nm généraliste de TSMC. Cela suggère des optimisations agressives en faveur de la surface. En effet, des estimations basées sur le modèle de CACTI 5.1 [TMAJ08] nous indiquent une efficacité en surface (fraction de la surface utilisée par les cellules SRAM) de l'ordre de 40 %. Cette densité est par exemple supérieure d'un facteur 20 à celle du banc de registres de l'Intel Itanium-2 [FGK+02].

2. http://wiki.github.com/pathscale/pscnv/nvidia_compute

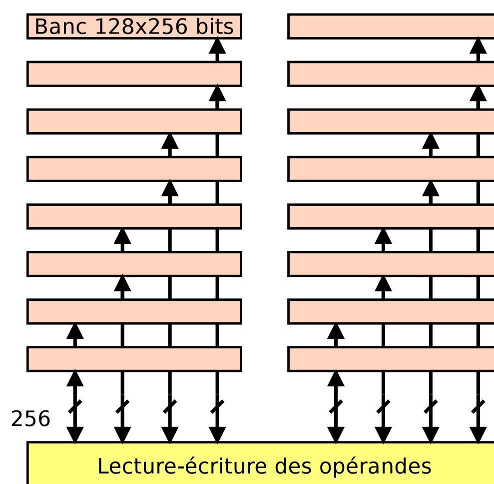


FIGURE 2.3 – Organisation probable du banc de registres du GT200.

Différents jeux de tests composés d'instructions MAD et MUL visant à saturer la bande passante du banc de registres font apparaître une limitation à deux opérandes lus par cycle SM. Ces résultats s'appliquent autant au banc de registres du GT200 qu'à celui du G92, qui dispose de 8 bancs mémoire. Le débit obtenu suggère que les bancs mémoire sont groupés par paires sur le GT200 de la manière illustrée figure 2.3, sans élargissement des chemins de données.

La stratégie adoptée pour minimiser les conflits de bancs est de profiter de la régularité de l'ordonnancement des warps (proche du tourniquet), au détriment de la latence. Ainsi, les registres d'un warp donné sont alloués préférentiellement dans le même banc, de façon à ce que des warps différents soient à l'inverse associés à des bancs différents [LSM$^+$08]. Les opérandes d'une instruction donnée devront alors impérativement être lus séquentiellement, ce que nous avons vérifié par des mesures de latence. En revanche, ces lectures peuvent être pipelinées avec les lectures d'opérandes des instructions suivantes, appartenant à d'autres warps dont les registres ont été alloués dans d'autres bancs. Ainsi, les conflits de bancs restent prévisibles et n'affectent ni l'ordonnancement ni le choix des numéros de registres.

Il semble également que chaque banc dispose d'un port de lecture et d'un port d'écriture séparés. Cette solution peut être réalisée à faible coût par multiplexage temporel [FGK$^+$02]. La figure 2.4 présente l'ordonnancement d'une série d'instructions MUL exécutées par 4 warps, que nous avons reconstitué en encadrant les MUL par des instructions MOV de lecture du compteur de cycles. Les opérandes d'entrée sont notées A et B et l'opérande de sortie R.

On note que l'écriture de la destination d'une instruction d'un warp donné peut coïncider avec la lecture de la source d'une instruction ultérieure du même warp, qui est placée dans le même banc. Les résultats obtenus suggèrent également que le chemin d'accès aux registres n'offre pas de réseau de court-circuit (*bypass*).

L'ensemble de ces décisions facilitent la tâche du compilateur en lui présentant un coût d'accès aux registres homogène. En revanche, le choix d'utiliser un banc de registres monolithique plutôt que hiérarchique comme sur les architectures AMD [AMD09b] permet difficile-

ment d'offrir une bande passante suffisante pour maintenir le débit d'exécution crête. En effet, l'organisation du banc de registres de Tesla permet de lire au plus 4 opérandes distincts tous les deux cycles à fréquence SM. Or, l'exécution simultanée d'instruction MAD et MUL sur les deux pipelines nécessiterait 5 opérandes d'entrée. Le débit crête pourra être atteint à la condition qu'il y ait partage d'opérandes ou qu'au moins un des opérandes provienne d'une autre source que le banc de registres (mémoire partagée, mémoire de constantes, immédiat dans le mot d'instructions).

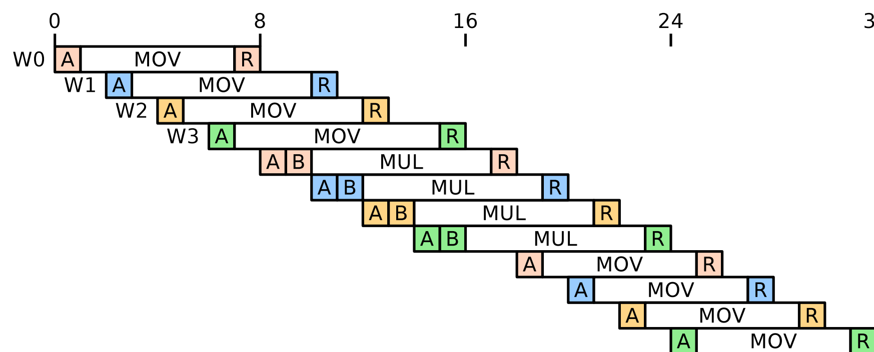


FIGURE 2.4 – Répartition des lectures des opérandes dans les bancs lors de l'exécution d'instructions MAD.

2.4.4 Unités d'exécution

Les GPU Tesla offrant la *Compute Capability* 1.3 permettent de manipuler les formats à virgule flottante Binary32 et Binary64, que nous décrirons plus en détail dans la section 3.1. Les autres GPU se limitent au format Binary32.

Huit unités SP sont chargées des calculs arithmétiques généralistes. Chaque SP est un pipeline construit autour d'un multiplieur 24×24 suivi d'un additionneur puis d'une unité d'arrondi [SO08]. Il est capable d'effectuer avec une latence et un débit constants aussi bien les instructions entières (arithmétiques, logiques, décalages) que virgule-flottante (multiplication-addition ou *MAD*, min, max, conversions) dans le format Binary32.

La latence de cette unité peut être estimée à partir de la mesure du temps d'exécution de deux instructions dépendantes. Le délai de lecture et d'écriture des registres, ainsi que le temps d'amorçage du pipeline d'exécution doit cependant être pris en compte pour ce calcul. La figure 2.6 illustre une telle estimation pour une instruction MAD. Les résultats sont récapitulés table 2.5. Nous incluons le temps d'acheminement des données depuis et vers le banc de registres dans le délai d'exécution, ce qui peut expliquer les différences constatées entre les latences du G80 et du GT200, mais aussi entre les latence des différences unités.

Une seconde unité, nommée SFU, réalise l'évaluation de fonctions élémentaires (sinus, cosinus, exponentielle et logarithme de base 2, inverse et racine carrée inverse), l'interpolation d'attributs en entrée du *fragment shader*, ainsi que des multiplications. Cette unité contient un circuit d'interpolation suivi de multiplieurs Binary32 [OS05]. Les unités d'interpolation

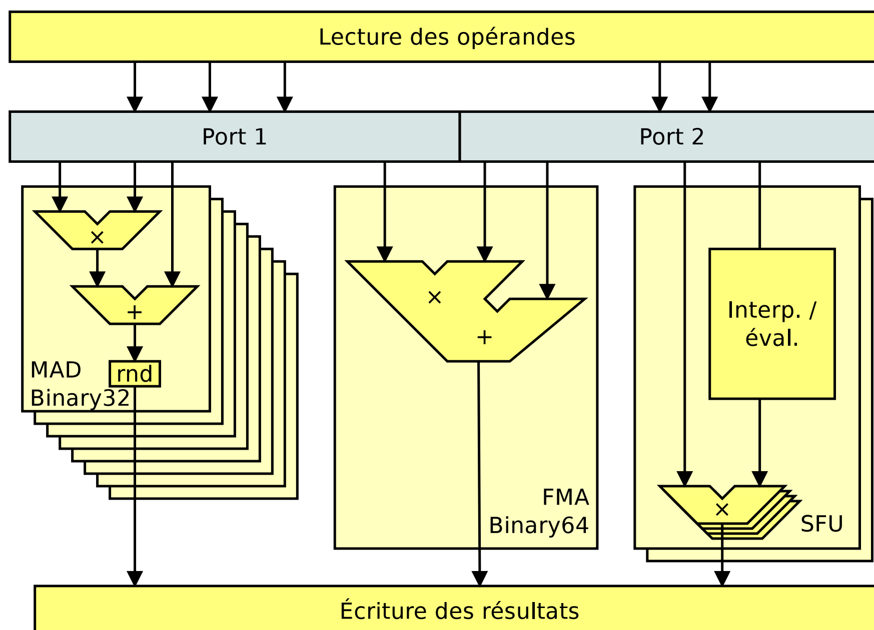


FIGURE 2.5 – Unités d'exécution d'un SM du GT200.

peuvent exécuter une instruction d'évaluation de fonction sur un warp tous les 16 cycles SP, ou une instruction de multiplication tous les 4 cycles [NVI10b].

La latence totale mesurée d'une telle évaluation de fonction est de 40 cycles SP sur le G80 et 42 cycles sur le GT200. Cela correspond à une latence d'exécution de 21 à 23 cycles, similaire à celle de la multiplication dans cette unité.

L'unité Binary64 présente sur le GT200 [OST09] peut démarrer une opération FMA (fused multiply-add) sur un warp tous les 32 cycles SP. Nous avons mesuré entre 58 et 66 cycles de latence totale pour une instruction FMA suivant le nombre d'opérandes uniques. Nous estimons donc la latence de l'unité FMA elle-même à 19 cycles SP.

Par comparaison, l'unité FMA du processeur Cell offre une latence de 6 cycles à une fréquence cible beaucoup plus agressive (3 GHz contre $1,5$ GHz pour Tesla dans une technologie comparable) [MJO+05]. Les unités virgule flottante de Tesla sont donc manifestement optimisées pour favoriser la consommation et la surface aux dépens de la latence, conformément aux objectifs de conception affichés [OS05].

2.5 Hiérarchies mémoire

L'architecture mémoire physique reflète l'organisation logique décrite dans la section 2.1. En particulier, on y retrouve des chemins d'accès distincts et autant de hiérarchies mémoire indépendantes pour la mémoire de textures, la mémoire de constantes, la mémoire d'instructions et la mémoire partagée. Cependant, pour permettre le partage de matériel, les espaces mémoire globaux et locaux partagent la majeure partie du chemin d'accès avec la mémoire de textures.

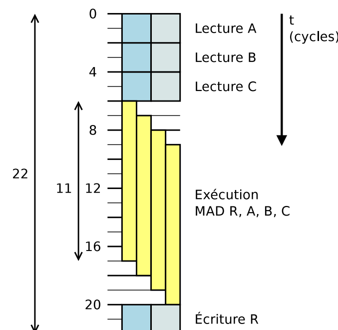


FIGURE 2.6 – Organisation du pipeline comprenant la lecture des opérandes, l'exécution et l'écriture des résultats durant l'exécution d'une instruction MUL. Les latences sont indiquées en nombre de cycles SP.

TABLE 2.5 – Latences estimées et débit des unités du G80 et GT200, en nombre de cycles SP

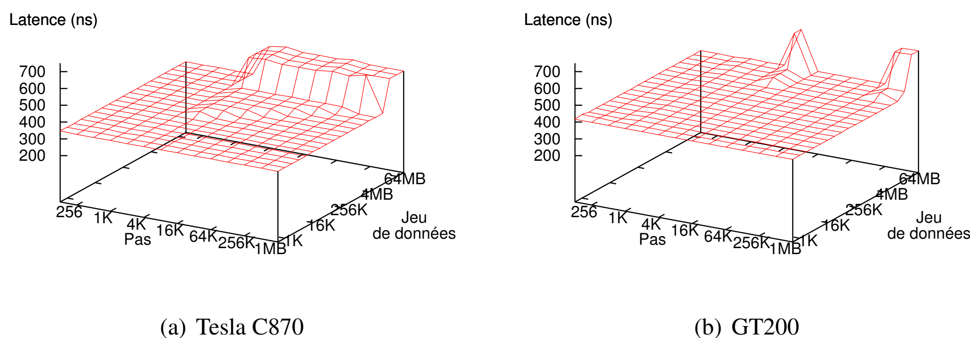
Instruction	Latence totale		Latence d'exécution (est.)		Débit
	G80	GT200	G80	GT200	(op/cycle)
MAD	18 – 22	20–24	11	13	8
SFU : Fonction	40	42	21	23	2
SFU : Mul.	26 – 28	28	19	21	8
FMA	-	58–66	-	19	1

Considérons les latences de chacune de ces mémoires. Je les ai mesuré par une séquence d'accès à *pas* constant qui parcourent itérativement une zone mémoire de taille limitée (*jeu de données*). Nous retenons le temps pris par le dernier accès effectué. Les figures 2.7, 2.8 et 2.9 retracent les résultats obtenus.

Alors que la latence d'un accès à la DRAM sur un processeur à contrôleur mémoire intégré est de l'ordre de 50 ns, les latences observées sur les GPU Tesla sont supérieures d'un ordre de grandeur et varient ici entre 300 ns et 800 ns. Ainsi, sur le Tesla C870 (figure 2.7(a)), nous observons trois plateaux situés respectivement à 345 ns, 372 ns et 530 ns en fonction de la localité des accès. Les latences associées aux lectures en mémoire de textures ne sont pas significativement plus faibles. Nous tenterons de comprendre dans cette section les mécanismes qui conduisent à de tels résultats.

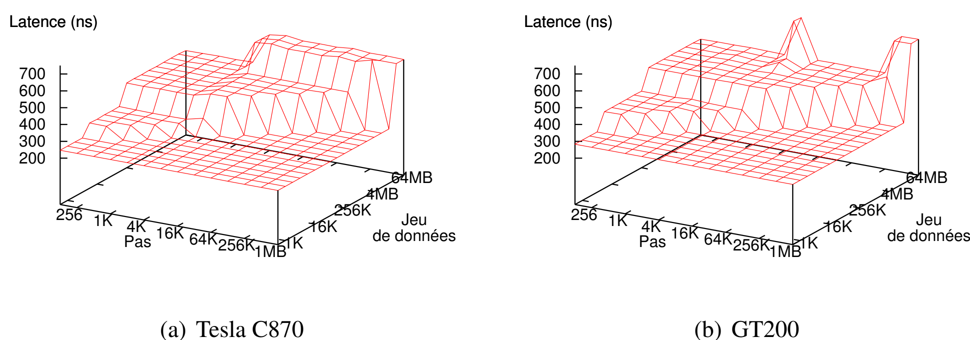
2.5.1 Mémoires internes

Chaque SM dispose :
- d'un cache de constantes de premier niveau,
- d'un cache d'instructions de premier niveau,
- d'une mémoire partagée.



(a) Tesla C870 (b) GT200

FIGURE 2.7 – Latence d'une lecture en mémoire globale.



(a) Tesla C870 (b) GT200

FIGURE 2.8 – Latence d'une lecture en mémoire de textures.

La table 2.6 récapitule les résultats connus par la documentation officielle ou des tests de la littérature [VD08, WPSAM10]. Nous avons pu vérifier ces résultats en ce qui concerne le cache de constantes.

Mémoires de constantes Les constantes et les instructions étant des données accessibles en lecture uniquement, elles peuvent être dupliquées à plusieurs emplacements parmi le sous-système mémoire sans nécessiter de protocole de maintien de la cohérence.

Qui plus est, ces données peuvent être considérées comme scalaires. Une seule instruction SIMT est nécessaire pour effectuer une opération sur un warp, par définition. Le débit effectif que le cache d'instructions doit fournir est donc réduit à une instruction par cycle. De même, les données lues en mémoire de constantes sont typiquement des scalaires « étalés » dans un registre vectoriel lors de la lecture. Il est donc possible au cache de constantes de se contenter d'un unique port de 32 bits.

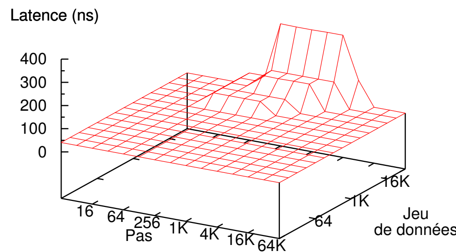


FIGURE 2.9 – Latence d'une lecture en mémoire de constantes sur GeForce 9800 GX2.

TABLE 2.6 – Caractéristiques des mémoires locales aux SM

Type	Taille	Ports	Largeur	Ligne	Associativité	Latence (cycles SM)
L1 Const.	2 Ko	1	32 bits	64 octets	4 voies	2
L1 Inst.	4 Ko	1	64 bits	256 octets	4 voies	2
Partagée	16 Ko	16	32 bits	–	–	3

Mémoire partagée En revanche, la mémoire partagée doit permettre de mémoriser temporairement des données vectorielles et d'y accéder de manière efficace. C'est également au travers de cette mémoire que sont effectuées les opérations d'échange entre threads d'un warp, plutôt qu'au moyen d'instructions dédiées (*swizzle*) comme dans d'autres architectures SIMD [Hof05].

Pour être à même de nourrir les unités de calcul sans devenir un goulot d'étranglement, la mémoire partagée doit offrir l'équivalent de 16 ports de 32 bits en lecture-écriture. Une telle mémoire n'étant pas raisonnablement réalisable, la mémoire partagée est composée de 16 bancs indépendants à l'instar des bancs de registres. Chacun de ces bancs ne possède qu'un unique port de lecture-écriture. Un arbitre se charge de simuler la présence de 16 ports. Comme celui du banc de registres, il est capable d'identifier les requêtes concurrentes à une case identique et de les fusionner, dans la limite d'une fusion par cycle. Les conflits sont traités par ré-exécution de l'instruction de la manière décrite section 2.4.2.

Notons que cette solution nécessite un *crossbar* complet de 16×16 sur 32 bits, qui est une structure coûteuse. Cela explique la latence d'accès à la mémoire partagée de 3 cycles à la fréquence SM.

En dehors du risque de conflits de bancs, on peut vérifier que les accès avec indirection ne sont pas plus coûteux en termes de latence et de débit que les accès avec adressage absolu. En effet, la valeur des registres d'adresse dédiés est vraisemblablement disponible plus tôt dans le pipeline que celle des registres généraux.

Fusion des accès Les SM accèdent à la mémoire globale par des opérations gather et scatter uniquement. Dans le cas général, le sous-système mémoire doit donc être en mesure de délivrer 16 données indépendantes. En l'absence de cache, fournir un tel débit sur des petites transactions n'est pas envisageable.

Heureusement, de nombreuses opérations gather et scatter sont en réalité des opérations de lecture et d'écriture de vecteurs, les adresses demandées par des threads d'identifiant successif étant contiguës.

Aussi, des mécanismes de fusion des accès, ou *coalescing*, permettent de consolider les lectures et écritures adjacentes des opérations gather et scatter. Ils augmentent la granularité des transactions mémoire. Leur comportement est décrit précisément dans la documentation [NVI10b]. Le mécanisme matériel qui en est responsable fonctionne sur la base de tables de requêtes en attente (PRT) [NNHM09].

Les SM des générations de GPU correspondant au Compute Model 1.0 et 1.1 détectent les opérations gather ou scatter qui suivent la forme particulière d'une lecture ou écriture de vecteur à pas unitaire. Les architectures des Compute Model 1.2 et 1.3 assouplissent ces contraintes en permettant tous les regroupements possibles entre sous-transactions [NVI10b].

2.5.2 Mémoires niveau cluster

Le TPC regroupe le cache de constantes de deuxième niveau et les unités d'accès mémoire, ce qui inclut les caches de textures de premier niveau et les unités de filtrage de textures.

Toutes les lectures de textures sont des opérations de type *gather* sur au moins 16 données distinctes. L'exécution des instructions dépendant du résultat ne peut se poursuivre que lorsque l'ensemble du vecteur lu est disponible. Cet effet diminue de manière drastique le bénéfice que le cache apporte sur la latence des accès. Pour obtenir un gain de latence, il faut que toutes les lectures composant l'opération gather soient des succès dans le cache. La probabilité de cet événement décroît rapidement avec la largeur du warp. De fait, le cache de textures est conçu en priorité comme un moyen d'augmenter le débit effectif plutôt que de diminuer la latence. Ce fait est illustré par la figure 2.8 montrant qu'un succès dans le cache L1 de textures conserve toujours une latence de 280 ns sur le GT200, soit 6 fois le temps d'un accès à la DRAM sur un système monoprocesseur actuel.

Dans l'architecture Tesla, le cache de textures intégré à chaque TPC est partagé statiquement entre chaque SM, et se comporte donc comme deux ou trois caches indépendants. Ses index et ses étiquettes référencent des adresses virtuelles. Son utilisation en lecture seule autorise la duplication de données et ne nécessite pas d'avoir recours à la traduction d'adresse pour identifier les synonymes éventuels.

L'analyse de Volkov et Demmel montre que seuls 5 Ko de cache par SM sont accessibles lorsque la mémoire de textures est adressée en mode linéaire. Nos propres tests confirment ces résultats et nous conjecturons que l'adressage linéaire correspond à un mode dégradé du cache ne permettant pas l'accès à l'intégralité des 8 Ko documentés, contrairement aux modes d'accès natifs 2D organisés en tuiles. L'associativité de cette portion du cache est de 20 voies, ce qui est largement supérieur aux associativités rencontrées habituellement dans les caches de premier niveau. Au vu des latences observées dans la section précédente, nous pouvons conjecturer que la recherche de voie est effectuée de manière séquentielle pour économiser

TABLE 2.7 – Taille des pages mesurée suivant le GPU et le pilote graphique

Pilote	G80-G92	GT200
177.13	4 Ko	16 Ko
195.17	64 Ko	256 Ko

l'énergie [ERB+95].

Les instructions de lecture et d'écriture peuvent se terminer dans le désordre par rapport aux autres instructions. Ainsi, les instructions suivant un chargement mémoire peuvent s'exécuter même si elles appartiennent au même warp, tant qu'il n'existe pas de relation de dépendance par rapport au registre cible du chargement. Nous avons ainsi pu tester que chaque warp peut opérer jusqu'à 5 opérations gather concurrentes.

L'unité d'accès mémoire est connectée aux bancs de registres de chaque SM. Contrairement à l'approche usuelle consistant à avoir un tampon de données arrivant depuis la mémoire dont le contenu est ensuite copié vers les registres, l'unité mémoire écrit ici directement dans le registre cible, évitant la nécessité d'une mémoire dédiée [LNMC08]. Ce mécanisme permet de maintenir un grand nombre de transactions mémoire en vol.

2.5.3 Traduction d'adresse

L'accès à la mémoire de textures, et par extension à la mémoire globale, nécessite au total trois étapes de traduction d'adresse successives [MGM09].

Une première étape doit calculer une adresse linéaire à partir de coordonnées de texture dans un espace à deux voire trois dimensions. Ce calcul se fait en amont de l'accès au cache de textures de premier niveau.

Un système de mémoire virtuelle est utilisé de manière à permettre à plusieurs applications de mémoriser des données privées dans la mémoire de la carte graphique tout en garantissant la fiabilité et la sécurité. La traduction des adresses virtuelles en adresses physiques est opérée au niveau du réseau d'interconnexion par des TLB.

L'analyse des latences mémoire révèle la présence de deux niveaux de TLB, suivi d'une autre structure [WPSAM10], qui pourrait être le cache de constantes de troisième niveau. Nos propres tests révèlent que, si les tailles et les valeurs d'associativité obtenues sont identiques, la taille des pages varie quant-à-elle d'une architecture à une autre, et même d'une version des pilotes à une autre sur le même GPU (table 2.7).

Enfin, les transactions mémoire doivent être acheminées vers une des partitions mémoire du GPU. Cette décision revient à calculer une adresse *brute* à partir de l'adresse physique. Le mode de traduction utilisé consiste à répartir des blocs de 256 octets en tourniquet sur les partitions disponibles.

Le nombre de partitions mémoire n'est pas nécessairement une puissance de deux, et peut même varier pour un même modèle de GPU au moment de l'incorporation sur la carte graphique. Cependant, sur tous les GPU existants, ce nombre est soit une puissance de deux, soit le produit d'un nombre premier et d'une puissance de deux. Cela rend envisageable des méca-

nismes de traduction simplifiés [SL93].

Ce découpage régulier a l'inconvénient d'être sensible aux motifs d'accès mémoire, en particulier lorsque le nombre de partitions est une puissance de deux. Le phénomène qui se caractérise par un déséquilibre entre les charges de chaque partition mémoire est nommé *partition camping* dans la littérature liée à CUDA [RM09]. Il se manifeste typiquement lors de la lecture d'une colonne d'une matrice. Pour les applications graphiques, ce problème est contourné par l'utilisation de modes d'adressages par tuiles spécifiquement conçus pour les donnés bidimensionnelles.

2.5.4 Réseau d'interconnexion et ROP

Le réseau d'interconnexion qui relie les TPC aux partitions mémoire est un circuit *ad-hoc* de type *crossbar*.

Les partitions mémoire sont chargées de gérer le trafic mémoire montant et descendant à destination ou en provenance du crossbar. Elles comportent en particulier les caches de textures de second niveau. Leur rôle est identique à celui du premier niveau : fusionner les requêtes concurrentes pour limiter la charge en bande passante et augmenter la granularité d'accès à la mémoire. Là où chaque cache L1 fusionne les accès concurrents effectués par les SM du TPC associé, les caches L2 fusionnent les accès issus des caches L1 de chaque TPC, en fonctionnant comme un cache distribué.

Les opérations atomiques, provenant aussi bien d'instructions CUDA que d'opérations de fusion de fragments en rendu graphique, sont également traitées dans cette unité.

2.5.5 Contrôleur mémoire

La lecture de brevets décrivant des contrôleurs mémoires de GPU apporte quelques éléments de réponse pour expliquer la latence mémoire observée [Eck08, HVD09]. Les contrôleurs conventionnels réordonnent les accès pour minimiser les conflits de pages DRAM et les inversions de bus entre phases de lecture et phases d'écriture. Au delà de telles politiques, le contrôleur mémoire de Tesla peut décider de volontairement retarder un accès en attente, dans l'espoir qu'une autre transaction plus avantageuse à traiter arrive. Ainsi, le risque d'alternance entre deux pages DRAM d'un même banc ou de cycles lecture/écriture répétés est réduit.

D'autre part, il peut être avantageux de maintenir une latence d'apparence uniforme du point de vue des clients, même si cela conduit à retarder les réponses aux requêtes servies rapidement pour s'aligner sur le pire cas. Ainsi, une synchronisation relative entre les warps est maintenue, ce qui est avantageux pour la localité dans les caches et autres structures exploitant la localité spatiale. Cette remise dans l'ordre des accès pour synchronisation peut être assurée par le contrôleur mémoire, le réseau d'interconnexion ou en fin de chaîne par l'ordonnanceur d'instructions lui-même [MLC+08].

Au vu des latences mémoire mesurées, nous conjecturons que le même mécanisme – ou le même type de mécanisme – est utilisé pour retarder les lectures de textures en cas de succès du cache L1.

Nous avons pu constater que l'ensemble des mémoires du GPU sont conçues en vue de maximiser le débit. Les mécanismes mis en œuvre pour atteindre cet objectif se basent sur

TABLE 2.8 – Caractéristiques des cartes graphiques testées.

Nom commercial	GPU	Fréquence cœur (MHz)	Fréquence calcul (MHz)	Fréquence mémoire (MHz)
Tesla C870	G80	575	1350	800
GeForce 9800 GX2	G92	600	1512	1000
Tesla T10P	GT200	540	1080	900

des compromis envers la latence. Ils exploitent également la coordination entre threads pour regrouper les requêtes et minimiser les conflits de bancs.

2.6 Gestion de l'énergie

L'énergie consommée et dissipée est le principal facteur limitant des architectures hautes-performances actuelles. Ainsi, les microprocesseurs tels que les Intel Core i5 et i7 opèrent désormais avec une fréquence, une tension et un nombre de cœurs actifs variables dans les limites d'une enveloppe thermique fixée, plutôt qu'à une fréquence fixe comme par le passé [Hin10]. Dans le cadre des GPU, nous disposons de peu de données concrètes sur l'impact que les choix de conception ont sur l'énergie, tant au niveau matériel qu'au niveau applicatif.

Nous avons effectué des tests sur des GPU d'architecture G80, G92 et GT200 que nous présenterons dans cette section [CDT09].

D'autres travaux ont succédé à nos tests. Suda et Ren ont mesuré la consommation d'un GPU G80 ou G92 sur des noyaux de calcul synthétiques [SR09]. Ils mettent en contraste la consommation mesurée avec le nombre de threads en vol et proposent un modèle affine pour estimer la puissance consommée. Leur étude se limite à l'exécution d'instructions d'additions flottantes, et ne considère pas la hiérarchie mémoire.

Hong et Kim ont proposé un modèle linéaire pour estimer l'énergie nécessaire à l'exécution d'un programme CUDA [HK10]. Un coût énergétique constant est associé à chaque instruction PTX. Le modèle est calibré par des mesures sur des tests synthétiques. Cependant, ce modèle ne prend pas en compte les effets non-linéaires, tels que des variations de tension et de fréquence, ou *power-gating* à gros grain. Le modèle proposé est également faiblement corrélé à l'architecture réelle du GPU considéré.

Nous cherchons ici à quantifier l'influence des choix de programmation sur la consommation du GPU, en vue d'élaborer des modèles de consommation énergétique.

2.6.1 Protocole de test

Nos tests ont été effectués sur les cartes graphiques basées sur Tesla que nous listons table 2.8. Celles-ci comprennent une carte Tesla destinée au calcul scientifique, une carte bi-GPU haut de gamme 9800 GX2 et un prototype de Tesla C1060 basé sur le GT200. La configuration logicielle est Ubuntu 8.04, CUDA 2.0 et CUDA 2.3 avec les pilotes graphiques NVIDIA 195.17.

TABLE 2.9 – Consommation électrique moyenne P_{Avg}, temps de calcul t et énergie correspondante E mesurés lors de l'exécution de transposition naïve ou optimisée, multiplication de matrices simple, et appel à la routine SGEMM des CUBLAS.

GPU		Trans. naïve			Trans. optimisée		
	À vide (W)	P_{Avg} (W)	t (ms)	E (J)	P_{Avg} (W)	t (ms)	E (J)
G80	68	103	2,40	0,247	127	0,30	0,038
G92	71	94	3,66	0,344	105	0,45	0,047
GT200	51	73	4,11	0,300	83	0,50	0,042

		MatMul			CUBLAS SGEMM		
	À vide	P_{Avg}	T	E	P_{Avg}	T	E
G80	68	132	25,0	3,30	135	11,8	1,59
G92	71	117	23,3	2,73	122	11,4	1,39
GT200	51	114	13,0	1,48	113	7,44	0,84

Pour les mesures de consommation, nous avons utilisé une pince ampèremétrique CA60 sur les câbles d'alimentation 12 V du boîtier Tesla D870, reliée à un oscilloscope Tektronix TDS 3032 mesurant également la tension d'alimentation. Cette méthodologie permet une mesure de la consommation sur des échantillons de l'ordre de 20 µs. Cela s'avère suffisant pour des mesures à l'échelle des tâches, mais non à l'échelle des instructions.

2.6.2 Tests applicatifs

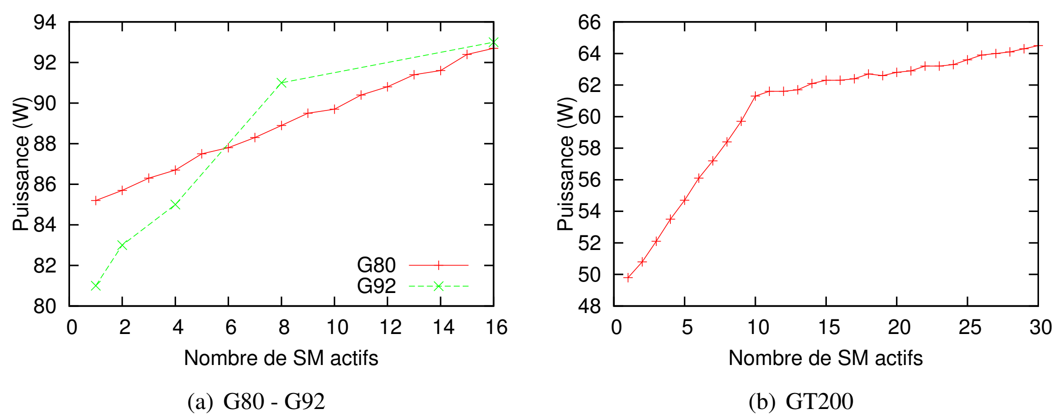
Considérons tout d'abord deux algorithmes courants, la transposition et la multiplication de matrices. Nous exécutons les exemples fournis dans le SDK CUDA sur des matrices aléatoires de 1024×1024. Les résultats en termes de temps de calcul, de consommation moyenne et d'énergie sont présentés dans la table 2.9.

Notons que le GPU GT200 nécessite plus d'énergie que le G80 pour exécuter les algorithmes de transposition de matrice considérés, bien qu'il bénéficie d'un processus de gravure plus avancé. En effet, les motifs d'accès à la mémoire de ces exemples causent des conflits de partition mémoire sur le GT200 qui est muni d'un bus 512 bits, selon le phénomène que nous avons abordé dans la section 2.5.3.

En revanche, le GT200 nécessite deux fois moins d'énergie que le G80 pour effectuer une multiplication de matrices. Cela suggère que les motifs d'accès mémoire jouent un rôle majeur, autant en termes de temps de calcul que de consommation.

2.6.3 Ordonnancement

Nous avons également mesuré la consommation des GPU considérés en faisant varier le nombre de multiprocesseurs actifs de 1 jusqu'au maximum (16 pour le G80 et le G92 et 30 pour le GT200). Chaque thread exécute une boucle déroulée d'instructions MAD. Les résultats sont présentés sur la figure 2.10. Nous observons que la puissance consommée augmente linéairement avec le nombre de multiprocesseurs jusqu'à un point correspondant respectivement à tous, la moitié et le tiers des multiprocesseurs pour le G80, le G92 et le GT200.



(a) G80 - G92 (b) GT200

FIGURE 2.10 – Variation de la puissance consommée en fonction du nombre de CTA ordonnancés.

Ces résultats confirment la différence de politique d'ordonnancement entre le G80 et les architectures ultérieures. Ils montrent également la présence d'un mécanisme de repos (*clock-gating* et/ou *power-gating*) à la granularité du TPC.

2.6.4 Instructions

J'ai construit des tests synthétiques en assembleur Tesla pour activer de manière sélective les unités de calcul. Les séquences d'instructions suivantes sont générées :

- instructions MAD pour saturer les unités SP,
- instructions prédiquées par un prédicat dont la valeur est toujours fausse,
- instructions MAD et MUL entrelacées pour saturer les unités SP et SFU,
- instructions MAD et RCP entrelacées,
- instructions RCP,
- instructions MOV entre registres.

Les résultats de performance et de consommation sont présentés sur la table 2.10. Les puissances présentées sont toujours des puissances moyennes mesurées sur l'ensemble de l'exécution du noyau. La valeur CPI représente le nombre moyen de cycles pris pour l'exécution de chaque instruction. Nous en déduisons l'énergie moyenne consommée par instruction.

Notons que la puissance maximale est généralement atteinte lors de l'exécution d'instructions MOV entre registres, plutôt que d'instructions de calcul tels que des MAD. Ce fait contre-

TABLE 2.10 – Puissance consommée totale P, nombre moyen de cycles par instruction CPI et énergie moyenne par instruction arithmétique E sur les GPU G80, G92 et GT200.

Opérations	G80			G92			GT200		
	P (W)	CPI	E (nJ /warp)	P (W)	CPI	E (nJ /warp)	P (W)	CPI	E (nJ /warp)
MAD	107	4,75	8,57	100	4,29	5,06	91	4,30	5,31
Pred	90	2,38	2,43	93	2,39	2,14	75	2,36	1,75
MAD+MUL	117	3,19	7,24	111	2,83	4,61	102	2,82	4,44
MAD+RCP	115	3,96	8,63	110	3,55	5,63	98	3,54	5,14
RCP	98	15,89	22,07	96	16,00	16,28	81	15,99	14,81
MOV	118	2,31	5,34	113	2,46	4,21	101	2,46	3,79

TABLE 2.11 – Énergie par lecture mémoire, suivant l'emplacement des données.

Espace mémoire, cache	G80 (nJ)	G92 (nJ)	GT200 (nJ)
Global	124,4	103,4	80,6
Texture, L1	60,7	28,3	25,6
Texture, L2	62,3	48,0	66,7
Texture, DRAM	102,2	110,6	83,1

intuitif s'explique par le débit d'exécution plus élevé des instructions MOV, qui permet d'exécuter davantage d'instructions par unité de temps. Or, la lecture et le décodage des instructions ainsi que les lectures et écritures dans le banc de registres sont plus coûteux en énergie que l'exécution des calculs.

2.6.5 Mémoire

Mesurons l'énergie nécessaire pour effectuer une lecture mémoire de 128 octets contigus effectuée par un warp. Nous considérons la mémoire globale et la mémoire de textures, en calculant les adresses de façon à viser chaque niveau de cache individuellement. Les résultats sur G80, G92 et GT200 sont présentés dans la table 2.11. Les valeurs sont toujours calculées à partir de l'énergie totale mesurée et du temps de transfert.

Notons que le cache de textures niveau 1 du G80 nécessite autant d'énergie par octet que le cache L2. Ceci s'explique probablement par la faible bande passante de ce cache.

Le coût en consommation d'une lecture linéaire en DRAM est comparable à l'exécution de 50 à 60 opérations en virgule flottante (flops). En termes de débit, la même lecture sera équivalente à 25 à 30 flops. Les accès mémoire sont donc particulièrement coûteux en termes de consommation, plus encore qu'en termes de débit. Autrement dit, une part plus large du

budget énergétique est consacrée aux accès mémoire qu'aux calculs.

2.7 Conclusion

À travers cette analyse de l'architecture Tesla, nous avons pu mesurer les enjeux architecturaux et micro-architecturaux soulevés par les besoins et caractéristiques de ces coprocesseurs. Elle nous apporte également des éléments pour optimiser les performances mais aussi la consommation des applications cibles.

Par le biais de multiples exemples, nous avons pu observer les aspects suivants.

Architecture haut-niveau La conception de l'architecture Tesla semble guidée par un souci de fournir un modèle de programmation simple et stable au niveau architectural. Cette simplicité se retrouve dans le modèle de programmation exposé au développeur par CUDA et OpenCL qui offre un contrôle fin sur le matériel. Le fossé sémantique à franchir pour compiler des programmes de calcul généraliste vers cette architecture est réduit. Il est donc possible d'obtenir des outils efficaces pour un faible coût de développement. Le programmeur dispose également d'opportunités pour optimiser ses programmes au travers de règles simples (*coalescing*, conflits de banc…), sans qu'il ne lui soit nécessaire de connaître le détail de l'implémentation matérielle.

En contrepartie, la conception de l'architecture est plus délicate, et des mécanismes matériels doivent être mis en place pour en maintenir l'abstraction : gestion transparente des branchements, détection dynamique des accès mémoire réguliers (*coalescing*), arbitrage des conflits… Ces mécanismes ont un coût en ressources matérielles. Il s'agit d'un des facteurs pouvant expliquer les différences de performances crêtes par mm^2 entre les architectures NVIDIA et AMD rencontrées section 1.2.3.

Absence de mécanisme spéculatif Les processeurs superscalaires emploient une grande variété de techniques basées sur la spéculation, telles que la prédiction de branchements ou le pré-chargement mémoire. Ces mécanismes permettent de réduire la latence moyenne, au détriment de la consommation et éventuellement du débit.

Nous n'avons pas observé de tels mécanismes dans l'architecture Tesla, même pour des techniques peu coûteuses telles que la prédiction de branchements statique. Cette décision semble partir du principe que l'exploitation du parallélisme de données fournit suffisamment de travail non spéculatif pour maintenir les unités de calcul occupées.

Exploitation de la cohérence Les modèles de programmation synchrones sont généralement considérés comme néfastes, car ils limitent le passage à l'échelle. Cependant, le couplage fort entre threads entrainé par une parallélisation à grain fin peut avoir des effets bénéfiques. Il contribue à la localité des données partagées entre threads concurrents. Cette localité est exploitée notamment par les caches d'instructions et les caches de textures pour augmenter leur taux de succès.

Unités arithmétiques spécialisées Les GPU tels que Tesla sont spécialisés pour les applications faisant appel au calcul virgule flottante intensif. Cela se reflète dans la conception des unités de calcul généraliste, mais aussi les unités de filtrage de textures, d'interpolation d'attributs et les ROP. Les caractéristiques de ces unités, telles que la précision des résultats intermédiaires, sont déterminées en fonction de l'application. L'espace de paramètres architecturaux à explorer est donc bien plus large que celui des processeurs équipés d'unités en virgule flottante compatibles avec la norme IEEE 754 [Obe06].

L'analyse présentée dans ce chapitre a fait l'objet de publication [CDT09], ainsi qu'un rapport publié préalablement à des accords de non-divulgation [Col10].

Chapitre 3

Exploiter les spécificités arithmétiques du GPU

Le domaine du GPGPU est au centre d'une controverse depuis sa création. D'un côté, les constructeurs de GPU, des fournisseurs de solutions logicielles aussi bien que de nombreux chercheurs vantent des facteurs d'accélération de 30 à 100 obtenus lors de portage d'applications sur GPU [GLGN+08, MWHL06, CDD08a]. En face, des groupes d'intérêts opposés dénoncent des protocoles expérimentaux biaisés, des comparaisons à des degrés d'optimisation inégaux ou entre des produits de générations et gammes différentes, ou l'absence de prise en compte des temps de transfert dans ces affirmations. Ils réduisent les facteurs d'accélération à de modestes facteurs 5 à 6 [LKC+10].

En effet, nous avons vu au chapitre 1 que le ratio entre les performances théoriques de CPU et GPU équivalents était de l'ordre d'un facteur 10. S'il est vrai que des facteurs d'accélération de 100 sont loin de refléter la réalité, il existe une classe d'applications capable d'atteindre des accélérations supérieures au facteur 10 des performances crêtes. Il s'agit des applications qui tirent parti des unités spécialisées des GPU qui n'ont pas d'équivalent sur CPU. Nous désirons donc élargir cette classe d'applications autant que possible.

Nous avons vu au cours des deux premiers chapitres que les architectures GPU mettaient l'accent sur la puissance de calcul à virgule flottante. Au delà de l'aspect quantitatif, ces unités flottantes diffèrent également de manière qualitative par rapport à leurs équivalents sur CPU généralistes. La souplesse des normes existantes – voire leur absence – ainsi que les contraintes spécifiques aux tâches graphiques ont conduit à des choix souvent différents de ceux qui se sont progressivement imposés pour les CPU dans les années 1970. Nous devons donc adapter les applications pour exploiter ces unités arithmétiques alternatives. Tout d'abord, il est nécessaire de connaître précisément le fonctionnement et les caractéristiques des unités arithmétiques dont nous disposons. Ensuite, il faut déterminer dans quelle mesure les unités spécialisées peuvent être utilisées à d'autres fins. Le cas échéant, il faudra développer des méthodes nouvelles pour les exploiter.

Après une introduction sur l'arithmétique virgule flottante section 3.1, nous établirons des tests permettant de déterminer le comportement des unités arithmétiques des GPU de la génération Direct3D 9 dans la section 3.2. Nous considérerons ensuite trois applications tirant parti

des unités spécifiques des GPU. Dans la section 3.3, nous reformulerons une application de modélisation de transferts radiatifs pour exploiter les unités spécialisées du pipeline graphique traditionnel. Nous détournerons dans la section 3.4 les unités de filtrage de textures pour évaluer des fonctions continues, et l'appliquerons aux calculs dans le système logarithmique. Dans la section 3.5, nous présenterons une bibliothèque d'arithmétique d'intervalles qui tire parti des modes d'arrondis statiques offerts par les GPU NVIDIA.

3.1 Introduction à l'arithmétique virgule flottante

Nous allons être amenés dans le reste de cette thèse à utiliser des notations et des résultats de l'arithmétique sur les nombres à virgule flottante. Nous nous basons autant que possible sur l'arithmétique définie par la norme IEEE 754-2008 [IEE08] (notée IEEE 754 dans la suite).

3.1.1 Système de représentation

Un système de représentation de nombres à virgule flottante $\mathbb{F}_p$ est caractérisé par une base b, une précision p et une plage d'exposants $(e_{\min}, e_{\max})$. Dans un tel système, un nombre à virgule flottante (ou simplement *flottant* par substantivation) $x \in \mathbb{F}_p$ est défini par un signe s, un exposant e et une mantisse m, tels que :

$$x = (-1)^s \cdot m \cdot b^e \tag{3.1}$$

avec $s \in \{0, 1\}$, $e \in \{e_{\min}, \ldots, e_{\max}\}$, et m un nombre fractionnaire à p chiffres tel que $m = m_0, m_1 \ldots m_{p-1}$, $m_i \in \{0, \ldots, b-1\}$.

Nous nous concentrerons dans la suite sur le cas de la base 2 ($b = 2$), qui est la seule actuellement en usage sur les GPU.

La norme IEEE 754 définit les nombres flottants *normalisés* qui sont tels que $m_0 \neq 0$, ce qui revient à $m_0 = 1$ en base 2. Zéro n'étant pas représentable sous forme normalisée, il nécessite un codage spécifique. Il en va de même pour les autres nombres situés entre $-2^{e_{min}}$ et $2^{e_{min}}$. Aucun de ceux-ci n'étant représentable sous forme normalisée, cela forme un « trou » centré en zéro dans la distribution des flottants. Pour y remédier, la norme définit les *dénormaux*, qui sont des flottants tels que $m_0 = 0$ et d'exposant $e = e_{\min}$.

La norme définit aussi les limites $+\infty$ et $-\infty$, et la valeur d'exception NaN (*not a number*). Plusieurs représentations des NaN sont permises, ce qui peut servir à décrire le comportement à adopter lorsqu'un tel NaN est rencontré en entrée d'une opération arithmétique.

IEEE 754 décrit plusieurs formats flottants binaires et décimaux. Nous considérerons ici uniquement les formats décrits table 3.1.

Représentation Des représentations binaires de tailles respectives 16 bits, 32 bits et 64 bits sont associées à ces formats. Le bit m_0 des nombres normalisés étant toujours égal à 1, il n'est pas inclut dans la représentation. On le nomme le bit implicite. L'exposant est représenté sous forme biaisée, c'est-à-dire en lui rajoutant une constante égale à $-e_{\min} - 1$ de façon à ce que sa représentation reste positive. L'ordre des champs (illustré figure 3.1 pour le format Binary32) est choisi de manière à ce que l'ordre sur les flottants positifs corresponde à l'ordre de leurs représentations binaires interprétées comme des entiers.

TABLE 3.1 – Formats virgule flottante IEEE 754 binaires utilisés dans ce document, et leurs caractéristiques.

Nom	Autre nom	p	$e_{\min}$	$e_{\max}$
Binary16	Demie précision	11	-14	15
Binary32	Simple précision	24	-126	127
Binary64	Double précision	53	-1022	1023

31	30 … 23	22 … 0
s	e	m

FIGURE 3.1 – Représentation binaire d'un flottant Binary32

Arrondis Lorsqu'une opération arithmétique est effectuée sur des flottants, son résultat exact n'est généralement pas représentable sous la forme d'un flottant. Il est donc nécessaire de l'arrondir en l'approximant par un flottant proche. IEEE 754 définit 4 modes d'arrondi obligatoires pour l'arithmétique flottante binaire. Ces modes sont :

- l'arrondi pair au plus près, noté $\mathcal{N}(x)$, selon lequel le résultat est le flottant le plus proche du résultat exact lorsqu'il est unique, ou le plus proche dont le bit de poids faible de la mantisse est nul sinon,
- l'arrondi vers $+\infty$, noté $\triangle(x)$, qui renvoie le flottant supérieur ou égal le plus proche,
- l'arrondi vers $-\infty$, noté $\bigtriangledown(x)$, qui renvoie le flottant inférieur ou égal le plus proche,
- l'arrondi vers zéro, noté $\mathcal{Z}(x)$, qui renvoie le flottant inférieur ou égal en valeur absolue le plus proche.

Notons que lorsque le résultat exact x est un flottant, tous les modes d'arrondi retournent ce résultat x : pour $x \in \mathbb{F}_p$, $\mathcal{N}(x) = \triangle(x) = \bigtriangledown(x) = \mathcal{Z}(x) = x$.

Les arrondis précités sont définis sans ambiguïté et sont dit *corrects*. Les arrondis corrects pouvant être coûteux à calculer, certaines implémentations se contentent d'un arrondi fidèle. L'arrondi *fidèle* d'une valeur peut retourner soit son arrondi vers $+\infty$, soit son arrondi vers $-\infty$, au choix non-uniforme de l'implémentation. Cette définition implique que si le résultat exact est un flottant, l'arrondi fidèle doit retourner ce flottant.

3.1.2 Unités de calcul

Calculs d'arrondis La norme dicte que les opérations de base sur les flottants se comportent comme si l'opération était effectuée en précision infinie avant que le résultat exact ne soit arrondi. En pratique, les calculs ne sont bien entendu pas réalisables en précision infinie.

Seuls trois bits supplémentaires sont nécessaires pour calculer l'arrondi correct dans les quatre modes d'arrondi. Ils sont appelés *guard*, *round* et *sticky*. Pour m la mantisse du résultat, ils correspondent respectivement au bit situé immédiatement après le dernier bit de la valeur

tronquée, au bit suivant et à la disjonction booléenne des bits restants :

$$g = m_p \tag{3.2}$$
$$r = m_{p+1} \tag{3.3}$$
$$s = \bigvee_{k \geq p+2} m_k. \tag{3.4}$$

La décision de sélectionner le flottant immédiatement supérieur ou le flottant immédiatement inférieur peut être déterminée par quelques équations booléennes en fonction du mode d'arrondi et de g, r et s.

UMA et FMA De nombreuses applications effectuent des multiplications immédiatement suivies d'additions. Cette séquence apparaît notamment en traitement du signal (convolution, FFT), en calcul géométrique ou en algèbre linéaire (produit scalaire, produit de matrices, évaluation polynômiale).

Un unique opérateur effectuant une multiplication suivie d'une addition est donc avantageux, car il augmente la performance par instruction exécutée.

Une première approche consiste à chaîner un multiplieur et un additionneur virgule flottante complets. Nous appellerons une telle unité *unfused multiply-add* (UMA). La surface et la latence de l'UMA restent proches du cumul de celles du multiplieur et de l'additionneur. Un compilateur peut remplacer les chaînes de multiplication-additions par des UMA sans modifier le comportement de l'application.

L'autre solution est de fusionner les deux opérations, sans effectuer d'arrondi intermédiaire. Cette unité est appelée *fused multiply-add* (FMA). L'opérateur FMA prend en entrée trois flottants $a, b, c \in \mathbb{F}_p$, et calcule le résultat $\mathcal{R}(a \times b + c)$, selon le mode d'arrondi $\mathcal{R} \in \{\mathcal{N}, \triangle, \triangledown, \mathcal{Z}\}$. L'UMA calcule $\mathcal{R}(\mathcal{R}(a \times b) + c)$ sous les mêmes hypothèses. Par rapport à un UMA, le FMA a une latence plus faible du fait de l'absence d'arrondi intermédiaire et de parallélisme entre les opérations. En revanche, il doit calculer et propager l'intégralité des $2p$ bits du produit $a \times b$.

Le FMA offre la possibilité de récupérer l'erreur d'une multiplication flottante par le calcul $\mathrm{FMA}(a, b, -\mathcal{N}(a \times b))$. Cette fonctionnalité a des applications pour le calcul de divisions et de racines carrées [Mar00], le calcul en précision multiple ou la réduction d'argument de fonctions élémentaires [BDL09]. Il est plus précis qu'une multiplication suivie d'une addition, mais peut poser des problèmes de symétrie lors de calculs tels que les produits scalaires. Remplacer des chaînes de multiplication et addition par des FMA affecte en général le comportement d'un programme.

Arrondis dynamiques et statiques La majorité des processeurs gère les modes d'arrondis par un registre de configuration interne pouvant être lu et écrit par des instructions spécifiques. Ce mode d'arrondi s'applique à toutes les opérations arithmétiques effectués entre deux instructions de changement de mode. Ce choix adopté dans les années 1970 et 1980 se justifiait par la compacité du code binaire, enjeu alors significatif. Nous nommerons ces modes d'arrondis *dynamiques*.

Certaines architectures plus récentes codent le mode d'arrondi de chaque opération dans un champ de son mot d'instruction. On parlera de modes d'arrondis *statiques*. Ainsi, le DEC Alpha permet de choisir entre trois modes statiques et un mode dynamique [HP/98]. L'Intel IA-64 permet la sélection statique entre deux modes d'arrondis dynamiques [Int06].

Drapeaux et exceptions Il peut être nécessaire de détecter des situations exceptionnelles, telles que des dépassements de capacité, perte de précision ou opération invalide.

La norme IEEE 754 définit deux mécanismes pour permettre cette détection. Les drapeaux sont un état interne au processeur contenant plusieurs bits correspondant chacun à une situation exceptionnelle : dépassement de capacité vers le bas, dépassement vers le haut, résultat inexact, opération invalide. Ces drapeaux sont rémanents : une fois levés par une opération, ils restent actifs jusqu'à leur remise à zéro manuelle. Ils permettent de déterminer si une instruction d'un bloc a provoqué une opération exceptionnelle, mais pas de connaître de quelle opération il s'agit.

Le second mécanisme est celui des exceptions, qui consiste à exécuter un code spécifié par le programmeur lorsqu'une situation exceptionnelle est rencontrée.

3.1.3 Modélisation et analyse d'erreurs

L'exigence d'arrondi correct de la norme permet de borner mathématiquement les erreurs de calcul.

Soit $\epsilon = 2^{1-p}$ *l'epsilon machine*, et $u = \frac{\epsilon}{2} = 2^{-p}$ *l'unité d'arrondi*. Pour un réel $x \geq 0$, on peut établir les encadrements suivants [Hig02] :

$$\begin{array}{rcccl} x(1-u) & \leq & \mathcal{N}(x) & \leq & x(1+u) \\ x & \leq & \triangle(x) & \leq & x(1+\epsilon) \\ x(1-\epsilon) & \leq & \triangledown(x) & \leq & x. \end{array} \tag{3.5}$$

Pour tout réel x, on a :

$$|x|(1-\epsilon) \quad \leq \quad |\mathcal{Z}(x)| \quad \leq \quad |x|. \tag{3.6}$$

Notons également que l'arrondi vers zéro est équivalent à l'un des deux autres modes d'arrondis dirigé, suivant le signe de l'argument :

$$\mathcal{Z}(x) = \begin{cases} \triangledown(x) \text{ lorsque } x \geq 0 \\ \triangle(x) \text{ lorsque } x \leq 0 \end{cases}. \tag{3.7}$$

Cela nous permet d'appliquer l'une ou l'autre des bornes dès lors que le signe de x est connu.

Nous définirons la distance entre deux flottants, ou *ulp* pour *unit in the last place*, d'après la définition de Muller [Mul05], avec clarification en zéro.

Définition 3.1 (ulp) *Si un réel x est situé entre deux flottants consécutifs a et b, sans être égal à l'un d'entre eux, alors* $\mathrm{ulp}(x) = |b - a|$. *Sinon,* $\mathrm{ulp}(x)$ *est la distance entre les deux flottants non-égaux les plus proches de x.*

Nous utiliserons également le résultat suivant, dû à Sterbenz [Ste74]. Il pose des conditions pour lesquelles le résultat d'une soustraction dans un système à virgule flottante sera calculé exactement.

Lemme 3.1 (Sterbenz) *Dans un système virgule flottante avec dénormaux, deux flottants a et b de même signe tels que :*

$$\frac{|a|}{2} \leq |b| \leq 2|a|,$$

alors $\mathcal{R}(a - b)$ et $\mathcal{R}(b - a)$ calculent le résultat exact, pour $\mathcal{R} \in \{\mathcal{N}, \triangle, \bigtriangledown, \mathcal{Z}\}$.

3.2 Test des unités de calcul

Nous avons constaté tout au long du chapitre 2 qu'il était nécessaire d'écrire des tests synthétiques pour comprendre l'architecture des GPU. Il en va de même pour l'étude du comportement des unités arithmétiques, en particulier en ce qui concerne les générations antérieures aux GPU compatibles Direct3D 10. Nous nous pencherons dans cette section sur les architectures NVIDIA G70 et ATI R520, toutes deux commercialisées à partir de 2005. Ces GPU manipulent des formats compatibles avec ceux de la norme IEEE 754, mais ne respectent pas les règles d'arrondi correct abordées section 3.1.

Les algorithmes numériques tels que l'arithmétique double-simple (adaptée de l'arithmétique double-double [Pri92]) se basent sur des propriétés de l'arithmétique flottante IEEE 754. Ces algorithmes peuvent nécessiter des adaptations lorsqu'ils fonctionnent sur des architectures qui ne respectent pas la norme. Il est donc crucial de comprendre quelles sont les différences par rapport à l'arithmétique virgule flottante normalisée afin d'écrire du code numérique correct.

3.2.1 Historique

L'environnement Microsoft Direct3D 10 définit des règles que les GPU compatibles doivent respecter. Il s'agit d'une version affaiblie de la norme IEEE 754, sans dénormaux, une exigence d'arrondi avec une borne d'erreur d'1 ulp pour l'addition et la multiplication, et de 2 ulp pour la division et la racine carrée [Bly06]. OpenCL 1.0 [Mun09] et Direct3D 10.1 [Micc] renforcent ces pré-requis en imposant également l'arrondi correct au plus près pour l'addition et la multiplication.

Cependant, les GPU antérieurs à ces spécifications n'offrent aucune garantie sur la façon dont sont effectués les calculs, et en particulier sur les règles d'arrondis. A contrario, les GPU qui respectent une norme peuvent suivre des règles plus strictes que celles qui sont édictées par l'organisme de normalisation.

Cette situation s'apparente à celle qui régnait avant la généralisation de la première norme IEEE 754-1985 sur les processeurs généralistes [IEE85]. À cette époque, plusieurs jeux de tests ont été développés pour remédier à cet état de fait. Ces tests ont divers objectifs : ceux de Schryer [Sch81] visent par exemple à détecter des bogues dans le matériel, tandis que les tests de Gentleman [GM74] permettent de récupérer des informations sur les caractéristiques des unités flottantes. Après la normalisation, des tests comme Paranoia de Kahan et Karpinski [Kar85] vérifient la conformité d'une implémentation avec la norme.

Un sous-ensemble de Paranoia et des tests de Schryer a été adapté pour s'exécuter sur GPU [HL04]. Cependant, ces tests sont directement adaptés du programme Paranoia de 1985, et ne prennent pas en compte les particularités des GPU, notamment la présence d'unités de calcul hétérogènes. Par exemple, on trouve des multiplieurs situés à divers niveaux dans le pipeline graphique, et rien n'indique qu'ils soient identiques. Les unités testées ne sont pas indiquées, bien que l'on puisse supposer que le code soit exécuté par les fragment shaders.

3.2.2 Architecture

Pour élaborer des tests pertinents, il est nécessaire de comprendre l'architecture interne des GPU considérés.

Les deux unités programmables de cette génération de GPU sont les vertex shaders et fragment shaders [1]. Ces deux processeurs sont relativement similaires. Ils comprennent chacun plusieurs cœurs d'exécution, dont le nombre varie suivant le modèle et le type de shader. Comme les GPU grand-public de cette génération traitent typiquement plus de fragments que de sommets, l'équilibre entre les deux types de shaders est ajusté en conséquence. Par exemple, le NVIDIA GeForce 7800 GTX (G70) possède 8 cœurs de vertex shaders pour 24 cœurs de fragment shaders. La figure 3.2 présente les shaders de ce G70 [MM05].

Chaque cœur d'exécution dispose de trois types d'unités de calcul :

- une ou plusieurs unités *MAD* réalisant des multiplications et additions sur des vecteurs de quatre flottants Binary32,
- une unité de filtrage de textures,
- une unité scalaire d'évaluation de fonctions élémentaires SFU.

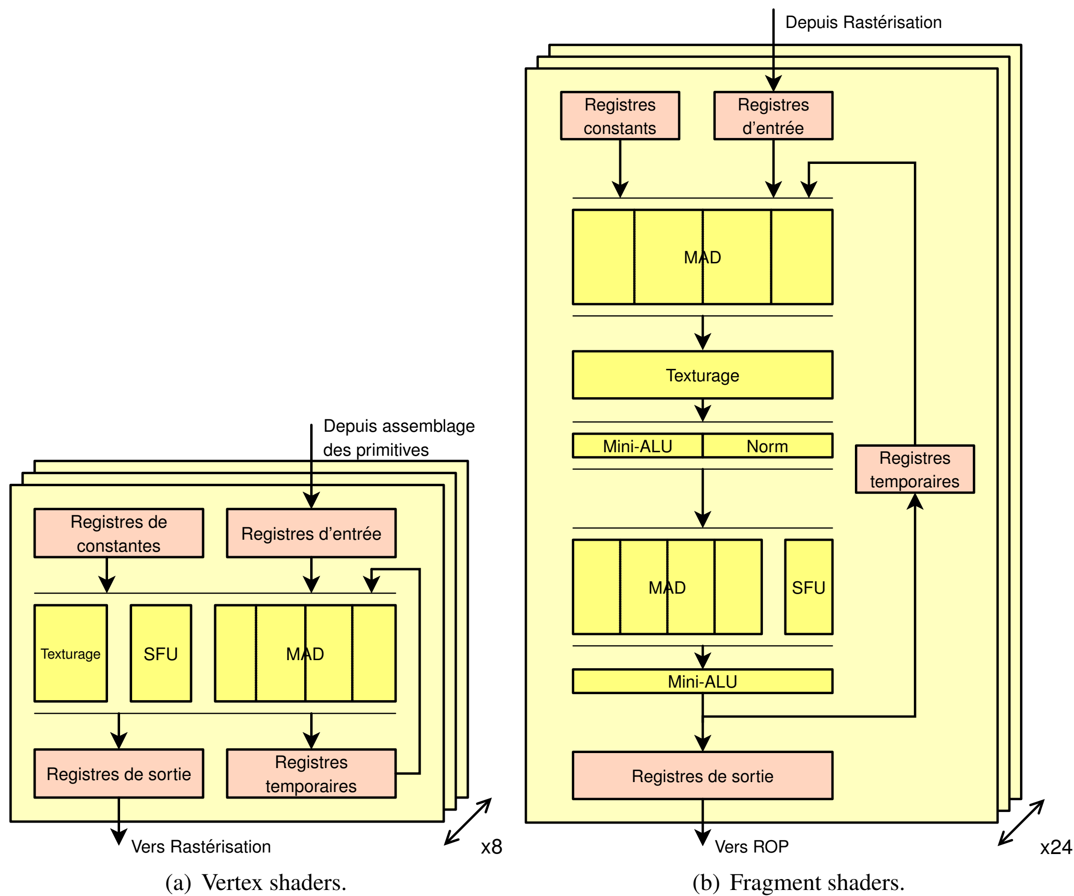
Les autres unités sont présentes pour des raisons de compatibilité et ne sont pas utilisées en GPGPU.

3.2.3 Caractéristiques des unités arithmétiques

Nous définissons et mettons en œuvre dans cette section des algorithmes permettant de mieux comprendre le fonctionnement de l'addition, de la multiplication et de l'écriture des nombres à virgule flottante dans les registres et en mémoire. Bien que ces algorithmes puissent être adaptés à d'autres formats de données, nous considérons ici le format Binary32. La table 3.2 liste les tests développés et utilisés ainsi que les résultats obtenus en fonction de la cible matérielle.

Dans cette table, $\oplus$, $\ominus$, $\otimes$ représentent respectivement l'addition, la soustraction et la multiplication disponible en matériel dans le GPU. $M = 2^{127}(2 - 2^{-23})$ est le plus grand flottant fini représentable dans le format Binary32. $U[a, b[$ sont des nombres flottants uniformément distribués sur l'intervalle $[a, b[$. PS ou VS indique si l'on considère les fragment shaders ou les vertex shaders. « Toutes » correspond à toutes les combinaisons. Les tests aléatoires sont effectués sur 2^{23} valeurs et les autres tests sont exhaustifs.

1. *Shader* désigne traditionnellement le programme qui sera exécuté sur les sommets ou les pixels. Par métonymie, nous employons ici le terme de shader pour les unités matérielles qui exécutent ces programmes.



(a) Vertex shaders. (b) Fragment shaders.

FIGURE 3.2 – Architecture des unités de vertex shaders et fragment shaders du GPU NVIDIA G70

Lectures et écritures mémoire Pour savoir si des conversions interviennent lors des lectures de textures, nous envoyons des dénormaux, des NaN et des infinis du CPU vers le GPU pour ensuite les récupérer sur le CPU. Nous avons aussi effectué des opérations sur GPU générant ces valeurs spéciales et récupéré le résultat sur CPU.

TABLE 3.2: Tests arithmétiques et résultats.

	Opérations	Unités	Observations
1	$(M \oplus M) \ominus M$	Toutes	$\longrightarrow \infty$
2	$MAD(x, y, -x \otimes y)$	Toutes	$x \sim U[1,2[\wedge y \sim U[1,2[\longrightarrow 0$
3			$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$
4		R520-PS	$i = 24 \quad \longrightarrow 1.5 - 2^{-23}$
5			$25 \leq i \quad \longrightarrow 1.5$

TABLE 3.2: Tests arithmétiques et résultats.

	Opérations	Unités	Observations
6			$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$
7	$1.5 \ominus 2^{-i}$	G70-PS	$24 \leq i \leq 25 \quad \longrightarrow 1.5 - 2^{-23}$
8			$26 \leq i \quad \longrightarrow 1.5$
9		R520-VS	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$
10			$24 \leq i \quad \longrightarrow 1.5 - 2^{-23}$
11			$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$
12	$1.5 \ominus 2^{-i}$	G70-VS	$i = 24 \quad \longrightarrow 1.5 - 2^{-23}$
13			$25 \leq i \quad \longrightarrow 1.5$
14			$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$
15	$(1 \oplus 0.5) \ominus 2^{-i}$	Toutes-PS	$24 \leq i \leq 25 \quad \longrightarrow 1.5 - 2^{-23}$
16			$26 \leq i \quad \longrightarrow 1.5$
17			$1 \leq i \leq 23 \quad \longrightarrow -2^{-i}$
18		R520-PS	$i = 24 \quad \longrightarrow -2^{-23}$
19			$25 \leq i \quad \longrightarrow 0$
20	$(1.5 \ominus 2^{-i}) \ominus 1.5$		$1 \leq i \leq 23 \quad \longrightarrow -2^{-i}$
21		G70-PS	$24 \leq i \leq 25 \quad \longrightarrow -2^{-23}$
22			$26 \leq i \quad \longrightarrow 0$
23	$x \otimes y + (\pm x) \otimes (\mp y)$	Toutes	$x \sim U[1, 2[\wedge y \sim U[1, 2[\longrightarrow 0$
24	$x \otimes y - (-x) \otimes (-y)$	Toutes	$x \sim U[1, 2[\wedge y \sim U[1, 2[\longrightarrow 0$
25	$x \otimes y - ((2 \cdot x) \otimes y)/2$	Toutes	$x \sim U[1, 2[\wedge y \sim U[1, 2[\longrightarrow 0$
26		R520-PS	$i \leq (2^{11} - 1) \cdot 2^{12} \longrightarrow$ correct
27	$(1 + 2^{-23}) \otimes (1 + 2^{-23} i)$	G70-PS	$i \leq 23 \cdot 2^{17} \quad \longrightarrow$ correct
28		R520-VS	$i \leq 2^{23} \quad \longrightarrow$ correct
29		G70-VS	$i \leq 2^{19} \quad \longrightarrow$ correct
30		R520-PS	$x \in [1, 2[\wedge y \in [1, 2/x[$ $\longrightarrow \{-1.00031 \text{ ulp} \cdots 0.00215 \text{ ulp}\}$
31			$x \in [1, 2[\wedge y \in [2/x, 2[$ $\longrightarrow \{-1.00013 \text{ ulp} \cdots 0.00085 \text{ ulp}\}$
32		G70-PS	$x \in [1, 2[\wedge y \in [1, 2/x[$ $\longrightarrow \{-0.51099 \text{ ulp} \cdots 0.64063 \text{ ulp}\}$
33	$x \otimes y - x \times y$		$x \in [1, 2[\wedge y \in [2/x, 2[$ $\longrightarrow \{-0.76504 \text{ ulp} \cdots 0.32031 \text{ ulp}\}$
34		R520-VS	$x \in [1, 2[\wedge y \in [1, 2/x[$ $\longrightarrow \{-1 \text{ ulp} \cdots 0\}$
35			$x \in [1, 2[\wedge y \in [2/x, 2[$ $\longrightarrow \{-1 \text{ ulp} \cdots 0\}$
36			$x \in [1, 2[\wedge y \in [1, 2/x[$

TABLE 3.2: Tests arithmétiques et résultats.

	Opérations	Unités	Observations
37		G70-VS	$\longrightarrow \{-0.82449$ ulp $\cdots 0.93750$ ulp$\}$ $x \in [1,2[\wedge y \in [2/x, 2[$ $\longrightarrow \{-0.91484$ ulp $\cdots 0.46875$ ulp$\}$

Lors d'un transfert sans aucune opération, les dénormaux sont remplacés par des zéros autant sur G70 que sur R520. Les NaN ne sont pas modifiés sur G70 mais le R520 normalise tous les NaN rencontrés en NaN canoniques, c'est-à-dire le codage par défaut sur cette plateforme.

Sur certaines architectures, les registres internes mémorisent les nombres avec une précision supérieure à celle des données en mémoire [Int10a] et avec une dynamique plus grande pour l'exposant [Int06]. La conversion vers le format final est réalisée lors de l'écriture en mémoire. Nous avons testé si certaines unités des GPU se comportent de manière similaire. Pour tester l'exposant maximum, nous utilisons le vecteur de test 1 de la table 3.2. Nous avons vérifié le nombre de bits de la mantisse des registres internes par les tests 17–22. Le test 2 permet de déterminer si les différents opérateurs MAD utilisent une précision interne supérieure à celle du format de travail.

L'exécution de ces tests montre qu'aucun des registres temporaires des vertex shaders et fragment shaders des deux GPU n'utilise de plage d'exposant étendue pour éviter les débordements ou d'une taille de mantisse supérieure pour augmenter la précision des calculs. Par ailleurs, aucun des MAD ne conserve le produit sur une précision étendue au-delà des 24 bits de précision des flottants Binary32.

Addition Nous avons cherché à clarifier le comportement de la soustraction des fragment shaders et vertex shaders. Nous déterminons le comportement du premier additionneur en calculant

$$1,5 - 2^{-i} \quad \text{(tests 3-13, table 3.2)}$$

pour i variant de 1 à 64. Le tableau suivant représente l'interprétation des résultats pour le fragment shader du G70 obtenus en fonction du paramètre i.

i	$1,5$	$1,5 - 2^{-i}$
1	$1.100 \cdots 00$	$1.000 \cdots 00$
2	$1.100 \cdots 00$	$1.010 \cdots 00$
3	$1.100 \cdots 00$	$1.011 \cdots 00$
$\vdots$	$\vdots$	$\vdots$
25	$1.100 \cdots 00$	$1.011 \cdots 11$
26	$\mathbf{1.100 \cdots 00}$	$\mathbf{1.100 \cdots 00}$
$\vdots$	$\vdots$	$\vdots$
64	$1.100 \cdots 00$	$1.100 \cdots 00$

Nous avons pu établir les conclusions suivantes concernant le premier additionneur :
- l'additionneur des fragment shaders du R520 et ceux des vertex shaders du G70 disposent de 1 bit supplémentaire (addition sur 24+1 bits),
- l'additionneur des fragment shaders du G70 dispose de 2 bits supplémentaires (addition sur 24+2 bits),
- l'additionneur des vertex shaders du R520 dispose des bits *guard*, *round* et *sticky* lui permettant de calculer un arrondi correct vers zéro.

De façon similaire nous avons testé le second additionneur des fragment shaders en calculant les deux soustractions suivantes

$$(1 \oplus 0.5) \ominus 2^{-i}) \quad \text{(tests 14-16, table 3.2)}$$

pour i variant de 1 à 64. Les résultats nous montrent que les deux additionneurs cascadés des fragment shaders du G70 se comportent de façon similaire.

L'utilisation d'un bit supplémentaire dans l'additionneur permet de ne pas commettre d'erreur d'arrondi lorsque l'on calcule la différence de deux nombres proches mais dont les exposants diffèrent d'une unité. En revanche, la présence d'un ou de plusieurs bits supplémentaires dans une arithmétique tronquée peut faire que la propriété suivante n'est plus vérifiée [PV93]

$$e_x - e_y > p \implies \mathcal{N}(x+y) = x$$

où e_x et e_y sont les exposants respectifs de x et y. Cette propriété garantit que la différence

$$x + y - \mathcal{N}(x+y)$$

est toujours représentable en machine sauf dépassement de capacité. Cette caractéristique impose à certains algorithmes numériques de tenir compte de ce paramètre comme c'est la cas avec les algorithmes de calcul à précision multiple [Pri92, DD06].

Nous avons lancé les mêmes additions que précédemment en remplaçant le format Binary32 par Binary16. Les résultats obtenus sur G70 montrent que les additions sont effectuées à la précision maximale pour ensuite être arrondies dans le format de destination souhaité (11 bits). Ce comportement est cohérent avec la description de Binary16 dans la norme comme un format d'échange plutôt que de calcul [IEE08].

Multiplication La multiplication est tronquée à la fois sur R520 et sur G70. Nous avons testé la façon dont cette troncature était effectuée à l'intérieur des fragment shaders et des vertex shaders. Pour cela nous avons souhaité déterminer si tous les produits partiels étaient générés ou si pour des raisons d'économie de surface, les bits de poids faibles de certains produits partiels étaient ignorés [SS93]. Si tel est le cas, une constante est ajoutée à la somme des produits partiels pour corriger le biais statistique introduit par la troncature d'une quantité toujours positive.

Pour estimer la valeur de la constante de biais, nous multiplions deux vecteurs : un premier vecteur composé uniquement de la valeur $1 + 2^{23}$ et un deuxième vecteur composé des valeurs $1 + 2^{-23}i$ avec $i \leq 23$ (tests 26–29). Ce test génère une partie tronquée qui augmente au fur et à mesure que i augmente. La constante de biais est mesurée en déterminant la valeur de i pour

laquelle un arrondi vers le haut apparaît. Une différence apparaît lorsque i est égal à 23.2^{17} pour les fragment shaders et 2^{19} pour les vertex shaders du G70. Sur le R520 cette différence apparaît lorsque i est égal à $(2^{11} - 1).2^{12}$ pour les fragment shaders et 2^{23} pour les vertex shaders.

Nous avons aussi mesuré l'erreur entre le résultat exact et le résultat produit par les multiplieurs présents dans les GPU (tests 30–37). L'ensemble de ces tests nous permet de déduire les points suivants :

- le multiplieur des vertex shaders du R520 assure l'arrondi correct vers zéro,
- les autres multiplieurs sont tronqués,
- les multiplieurs du G70 approchent le comportement d'un arrondi au plus près, en équilibrant l'erreur d'arrondi,
- les multiplieurs des fragment shaders du R520 approximent un arrondi vers zéro.

Des tests supplémentaires effectués au voisinage des puissances de 2 confirment que le multiplieur du G70 fournit un arrondi fidèle. En revanche, celui du multiplieur du R520 peut avoir une erreur par valeur inférieure de plus d'un ulp.

Conclusion Nous avons présenté un ensemble de tests permettant de découvrir les caractéristiques des unités arithmétiques des processeurs ne suivant pas la norme IEEE-754 tels que les GPU de la génération Direct3D 9. Par contraste aux travaux des années 1970, ces tests prennent en compte les caractéristiques de ces architectures, telles que les unités réunissant multiplication et addition ou les multiplieurs tronqués. Ils sont conçus pour l'architecture hétérogène des GPU de cette génération, et permettent d'isoler chaque unité et son comportement.

Ces informations permettent ainsi de mieux comprendre les différences de comportement entre une exécution sur CPU et une autre sur GPU en ce qui concerne les algorithmes virgule flottante, ainsi que les différences de comportement pour une même opération effectuée dans des unités de shaders différentes.

3.2.4 Évolution des fonctionnalités arithmétiques

D'autres générations de GPU ont succédé à celle du R520 et G70, chacune renforçant la conformité envers la norme IEEE 754. La table 3.3 présente un aperçu des fonctionnalités propres à chaque architecture, et comparées à celles de CPU conventionnels. Nous avons sélectionné les jeux d'instructions x87 et SSE2 d'Intel [Int10a], PowerPC d'IBM [IBM09], IA-64 d'Intel [Int06], ainsi que le jeu d'instructions du coprocesseur SPU du Cell [GHF+06].

Nous pouvons classer les GPU de 2005 à 2010 en trois générations successives, qui coïncident avec les révisions de l'API Microsoft Direct3D.

1. La génération Direct3D 9 de 2005 comprend les architectures considérées dans la section précédente. Elle offre un unique mode d'arrondi qui diffère de l'arrondi correct au plus près, considère les dénormaux comme nuls mais offre un support minimal pour les valeurs spéciales ($\pm\infty$, NaN). L'arithmétique Binary32 du processeur Cell SPU peut être assimilé à cette catégorie. Il offre un opérateur FMA correctement arrondi vers zéro, mais ne permet pas la manipulation de valeurs spéciales.

TABLE 3.3 – Caractéristiques de l'arithmétique flottante binaire des principales architectures CPU haute performance et GPU. W désigne la taille du mot flottant. Flags et Exn indiquent respectivement la gestion des drapeaux et des exceptions IEEE-754. µc dénote une fonctionnalité offerte par le micro-code plutôt que câblée en matériel. D et S désignent respectivement des modes d'arrondis dynamiques et statiques. Le chiffre 4 représente les quatre modes d'arrondis $\mathcal{N}, \triangle, \triangledown, \mathcal{Z}$.

Architecture	W (bits)	FMA/UMA	Arrondis	Dénorm.	$\pm\infty$, NaN	Flags	Exn
x87	80	✗	4 D	µc	✓	✓	✓
SSE2	32/64	✗	4 D	µc	✓	✓	✓
PowerPC	64	FMA	4 D	µc	✓	✓	✓
IA-64	82	FMA	4 S+D	µc	✓	✓	✓
NV40	32	UMA	✗	✗	✓	✗	✗
R500	32	UMA	✗	✗	✓	✗	✗
Cell SPU	32	FMA	$\mathcal{Z}$	✗	✗	✓	✗
	64	FMA	4 D	Sortie	✓	✓	✓
GT200	32	UMA	$\mathcal{N}$,$\mathcal{Z}$ S	✗	✓	✗	✗
	64	FMA	4 S	✓	✓	✗	✗
RV770	32	UMA	$\mathcal{N}$	✗	✓	✗	✗
	64	UMA	$\mathcal{N}$	✗	✓	✗	✗
Larrabee 1	32/64	FMA	$\mathcal{N}$	✗	✓	✓	✗
GF100	32	FMA	4 S	✓	✓	✗	✗
	64	FMA	4 S	✓	✓	✗	✗
Evergreen	32	FMA, UMA	4 D	✓	✓	✓	✗
	64	FMA, UMA	4 D	✓	✓	✓	✗

2. La génération Direct3D 10 et 10.1 de 2008 offre l'arrondi correct au plus près mais ne gère pas les dénormaux. Elle comprend les architectures NVIDIA Tesla et AMD R600–R700. La description publique de l'architecture du projet Larrabee-1 d'Intel semble le placer dans cette catégorie [SCS+08].
3. La génération Direct3D 11 de 2010 généralise le FMA, les quatre modes d'arrondis et les dénormaux traités en matériel. Elle inclut les familles NVIDIA Fermi et AMD Evergreen.

Ainsi, en l'espace de cinq ans, les unités arithmétiques des GPU ont subi une évolution accélérée, passant d'une implémentation de l'arithmétique flottante comparable à celle des architectures des années 1970 à des capacités qui excèdent celles de la plupart des processeurs généralistes actuels.

Cela s'explique par le fait que le coût des unités de calcul en termes de surface et surtout d'énergie est relativement faible par rapport au coût du contrôle et des mouvements de données.

Le coût en latence des fonctionnalités telles que la gestion matérielle des dénormaux est rendu tolérable par l'architecture orientée débit du GPU. Il est donc possible d'intégrer des unités arithmétiques sophistiquées sans que le coût global ne soit significativement impacté. De plus, les cycles de développement et de renouvellement des architectures GPU sont sensiblement plus courts que ceux des processeurs généralistes, ce qui permet d'introduire régulièrement de nouvelles fonctionnalités. Enfin, la chaîne de compilation à base de JIT permet de s'affranchir de la compatibilité binaire, favorisant les innovations.

3.3 Détourner les unités graphiques

Nous avons vu dans la section précédente que les GPU offraient des opportunités intéressantes pour mettre en œuvre des algorithmes numériques. Nous nous efforcerons de tirer partie de ces fonctionnalités dans la suite de ce chapitre.

Les algorithmes classiques en GPGPU se basent quasi-exclusivement sur l'utilisation des fragment shaders et sur l'échantillonnage de textures sans filtrage, et exploitent occasionnellement les unité ROP. Nous chercherons à compléter ces travaux en tirant parti du potentiel des autres unités spécialisées. Nous utiliserons notamment la rastérisation, l'interpolation d'attributs, le test de profondeur et le *blending*.

Considérons à nouveau les GPU de la génération Direct3D 9, en prenant toujours comme exemple le R520 (ATI Radeon X1800 XL) et le G70 (NVIDIA GeForce 7800 GTX). Ces GPU étant antérieurs aux environnement de programmation pour le GPGPU tels que CUDA et CAL, ils ne peuvent être programmées que par l'intermédiaire des API graphiques. Les calculs doivent donc être décrits sous forme d'opérations sur des sommets et des fragments dans le cadre du pipeline de rendu que nous avons abordé section 1.1.1. Nous utilisons ici OpenGL 2.0 ainsi que certaines extensions des constructeurs.

3.3.1 Application : transferts radiatifs

Nous considérons à titre d'application la modélisation du fonctionnement d'un récepteur solaire de type four solaire, destiné à la production d'énergie électrique. La partie de la simulation physique que nous cherchons à porter sur GPU est l'interaction de rayons infra-rouges avec des gaz sous pression. Ces interactions sont estimées par une méthode de Monte-Carlo consistant à lancer des rayons dans des directions aléatoires. Le chemin parcouru par ces rayons sera alors suivi dans un espace discrétisé (modélisation par volumes finis). Dans chaque volume élémentaire de gaz, nous calculons l'énergie absorbée sur chaque longueur d'onde.

En effet, les gaz ont des caractéristiques d'absorption très dépendantes de la longueur d'onde de la lumière reçue. Le spectre d'absorption d'un gaz, exprimant la fraction d'énergie absorbée par unité de distance en fonction de la longueur d'onde se présente comme un ensemble de raies étroites. Nous considérons ici le calcul des échanges radiatifs dans un volume élémentaire donné.

Correction d'intensité et calcul de l'étalement La première étape du calcul consiste à calculer un facteur de correction de l'intensité pour chaque raie. Ce facteur est dépendant à la fois

de la température et de paramètres liés à la raie considérée. Ce calcul doit donc être réalisé à chaque fois que la température change. Comme les raies sont ici indépendantes les unes des autres, on dispose d'un large parallélisme de données qui peut être exploité directement par le GPU.

Les formules à évaluer sont tirées de [R+98] et sont reproduites dans les équations (3.8), (3.9) et suivantes avec les mêmes notations, et en posant $c_2 = \frac{h}{k}$:

$$S_{\eta\eta'}(T) = S_{\eta\eta'}(T_{\text{ref}}) \cdot \frac{Q(T_{\text{ref}})}{Q(T)} \cdot \frac{e^{-\frac{c_2 E_\eta}{T}}}{e^{-\frac{c_2 E_\eta}{T_{\text{ref}}}}} \cdot \frac{\left(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T}}\right)}{\left(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T_{\text{ref}}}}\right)}, \tag{3.8}$$

$$\gamma(T, p, p_s) = \left(\frac{T_{\text{ref}}}{T}\right)^n (\gamma_{\text{air}}(p_{\text{ref}}, T_{\text{ref}})(p - p_s) + \gamma_{\text{self}}(p_{\text{ref}}, T_{\text{ref}}) p_s). \tag{3.9}$$

Calcul de l'énergie absorbée Une fois que la somme des contributions des raies en chaque point du spectre est effectuée, nous calculons l'énergie absorbée par le volume de gaz considéré sur chaque longueur d'onde. L'énergie du rayon entrant I_{in} est décomposée par longueur d'onde, formant un spectre.

Pour chaque échantillon, nous calculons la contribution à la profondeur optique en multipliant l'énergie par la fonction d'étalement de la raie (ici un profil de Lorentz), et la densité u de molécules absorbantes par longueur de chemin unitaire :

$$\tau(\nu) = u S_{\eta\eta'}(T) \frac{1}{\pi} \frac{\gamma(T, p)}{\gamma(T, p)^2 + (\nu - \nu_{\eta\eta'})^2}. \tag{3.10}$$

Nous calculons ensuite le spectre de l'intensité sortante I_{out} d'un rayon traversant une section de longueur l du volume élémentaire de gaz considéré par la loi de Beer-Lamber et la loi de Planck :

$$I_{\text{out}}(\nu) = I_{\text{in}}(\nu) \cdot e^{-\tau(\nu) \cdot l} + \frac{2h\nu^3}{c^2} \cdot \frac{1}{\left(e^{\frac{c_2 \nu}{T}} - 1\right)} \cdot \left(1 - e^{-\tau(\nu) \cdot l}\right). \tag{3.11}$$

Finalement, nous pouvons calculer l'énergie totale absorbée par le volume de gaz depuis le rayon lancé :

$$E = \int (I_{\text{out}}(\nu) - I_{\text{in}}(\nu)) d\nu. \tag{3.12}$$

Cette intégrale est approximée par une somme des échantillons.

Pour avoir des estimations plus précises pour décider de l'implémentation à utiliser, nous avons établi des statistiques sur la base de données HITEMP [RWG+95] dans les conditions visées. Celles-ci sont présentées dans la table 3.4. Nous considérons une valeur limite en dessous de laquelle les contributions des raies sont ignorées.

La densité de raies spectrales est suffisante pour rendre l'utilisation d'une structure pleine efficace. On peut donc représenter un spectre échantillonné sous la forme d'un tableau (ou d'une texture), permettant les accès aléatoires.

Cette application est massivement parallèle, régulière et intensive en calculs. La précision et la dynamique qu'offre le format Binary32 sont suffisantes pour les calculs. Elle se prête donc bien à une implémentation sur les GPU de cette génération.

TABLE 3.4 – Statistiques sur les calculs nécessaires pour différentes valeurs de la limite sur $k_{\eta\eta'}(\nu, T, p)$

Limite	Raies		Échantillons		Superp. max		Largeur max	
(cm^{-1})	H_2O	CO_2	H_2O	CO_2	H_2O	CO_2	H_2O	CO_2
10^{-13}	353	4 051	7 324	366 156	2	5	19	145
10^{-14}	2 060	19 819	126 552	2 435 108	3	15	113	461
10^{-15}	7 117	75 620	927 628	12 617 948	4	22	395	1 457
10^{-16}	20 883	222 660	4 678 780	54 189 736	4	28	1 259	4 609
10^{-17}	58 039	525 544	20 700 852	204 068 100	6	54	3 987	14 573
10^{-18}	155 565	686 791	81 948 032	684 901 816	10	122	12 609	46 085

3.3.2 Détourner le pipeline graphique

Les équations (3.8) et (3.9) sont calculées par les vertex shaders. Nous transmettons à ces derniers deux attributs vectoriels : $(\nu_{\eta\eta'}, E_i'', S_{\eta\eta'}(T_{\mathrm{ref}}), \nu)$ et (x, y, γ, n), pour chaque raie. La variable γ est précalculée par $\gamma = \gamma(T_{\mathrm{ref}}, p_{\mathrm{ref}}, p_{s,\mathrm{ref}})/p_{\mathrm{ref}}$. Les autres quantités utilisées dans les équations (3.8) et (3.9) ne dépendent pas de la raie considérée. Elles sont transmises aux shaders par l'intermédiaire de la mémoire de constantes. La figure 3.3 illustre l'enchaînement des étapes de calcul au travers du pipeline graphique.

Les valeurs calculées par le vertex shader sont les coordonnées et les attributs de chaque sommet. Nous représentons chaque raie par ses extrémités sous forme de sommets.

La taille de chaque dimension des textures de cette génération de GPU est limitée à 8192, ce qui est insuffisant pour représenter un spectre complet dans une texture unidimensionnelle. Nous replions l'espace des nombres d'onde ν dans un espace bi-dimensionnel pour permettre la mémorisation du spectre dans une texture 2D.

L'étape suivante consiste à calculer l'étalement de chaque raie, fonction de la pression.

Rastérisation Nous pouvons alors voir l'opération d'échantillonnage des raies de manière graphique. Les raies sont *dessinées* sous forme de lignes sur la texture de destination. Ainsi, nous utilisons chaque étage du pipeline graphique. Les vertex shaders effectuent la correction de l'intensité des raies. L'unité de rastérisation se charge ensuite de faire l'échantillonnage de la raie, en créant un fragment par échantillon. Cette étape permet de déterminer les échantillons pour lesquels la contribution de la raie considérée est significative.

Les paramètres sont interpolés linéairement par l'unité d'interpolation d'attributs à partir des valeurs des extrémités. Cela nous permet de calculer les valeurs des nombres d'onde intermédiaires ν. Les autres attributs sont identiques à chaque extrémité du segment. Ils sont donc transmis tels quels à tous les fragments qui composent le segment.

Comme les raies peuvent franchir les bords de la texture et se poursuivre sur la ligne suivante, il est nécessaire de les scinder. Cette opération est réalisée en pré-traitement sur le CPU, en considérant le pire cas (étalement maximal). Le calcul du facteur de correction va devoir

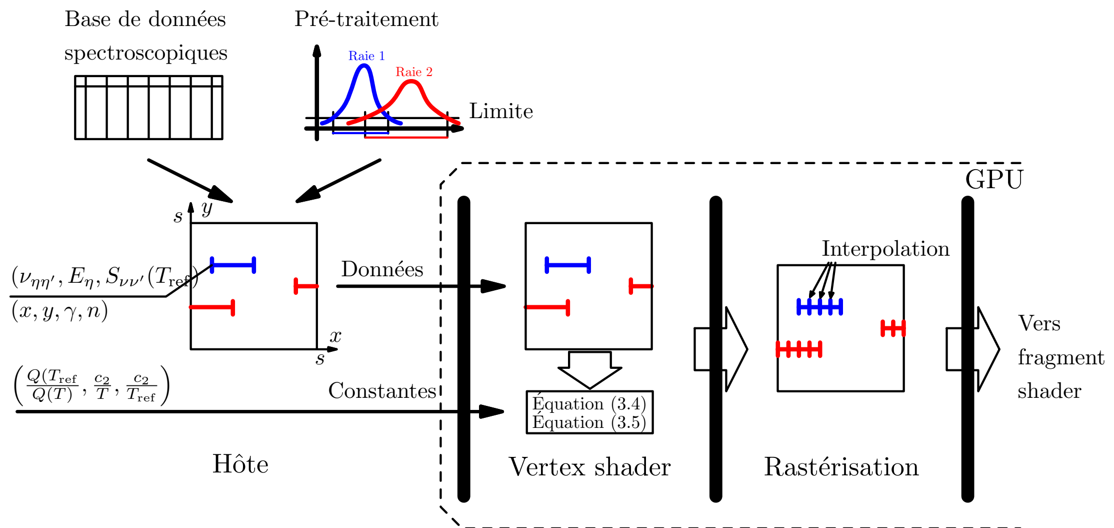


FIGURE 3.3 – Premières étapes du traitement sur GPU : correction de température et de pression.

être fait pour chaque sommet, et non pour chaque raie. La même opération devra donc être effectuée plusieurs fois. Cela n'a cependant pas d'impact perceptible sur la performance, étant donné que les statistiques de la table 3.4 nous montrent que le nombre de raies est très faible devant le nombre d'échantillons. Nos GPU utilisant des unités indépendantes pour l'exécution des vertex et fragment shaders, le calcul supplémentaire n'est pas un facteur limitant.

Calcul sur les échantillons Une fois le calcul de l'étalement effectué et les raies discrétisées, le fragment shader nous permet de travailler sur chaque échantillon. Nous l'utilisons pour calculer le profil d'étalement des raies.

Il faut ensuite considérer que plusieurs raies peuvent se superposer en un point donné. Les contributions respectives de chaque raie devront être accumulées pour calculer le facteur d'absorption de l'échantillon. Nous avons vu au cours de la section 1.1.1 que l'architecture de flot de données des shaders des GPU de cette génération ne permet pas d'accéder à la fois en lecture et en écriture à une zone mémoire donnée. L'opération d'accumulation n'est donc pas possible dans les fragment shaders.

Cependant, l'unité ROP est capable de réaliser des opérations de lecture-écriture à une même adresse, notamment au travers du mécanisme de *blending*.

Un problème pratique se pose néanmoins : les unités de blending de nos GPU G70 et R520 ne permettent pas d'effectuer ces opérations dans le format Binary32, mais sont limitées au format Binary16 ou aux formats virgule fixe. Ces représentations n'offrent pas une dynamique suffisante pour notre application, sans même entrer dans des considérations de précision.

La génération suivante de GPU (NVIDIA G80 et ATI R600) permet le blending dans le format Binary32. Pour les GPU antérieurs, nous utilisons une solution logicielle basée sur un tri réalisé en pré-traitement, un double tampon *ping-pong* [Pha05] et le test d'occlusion des ROP.

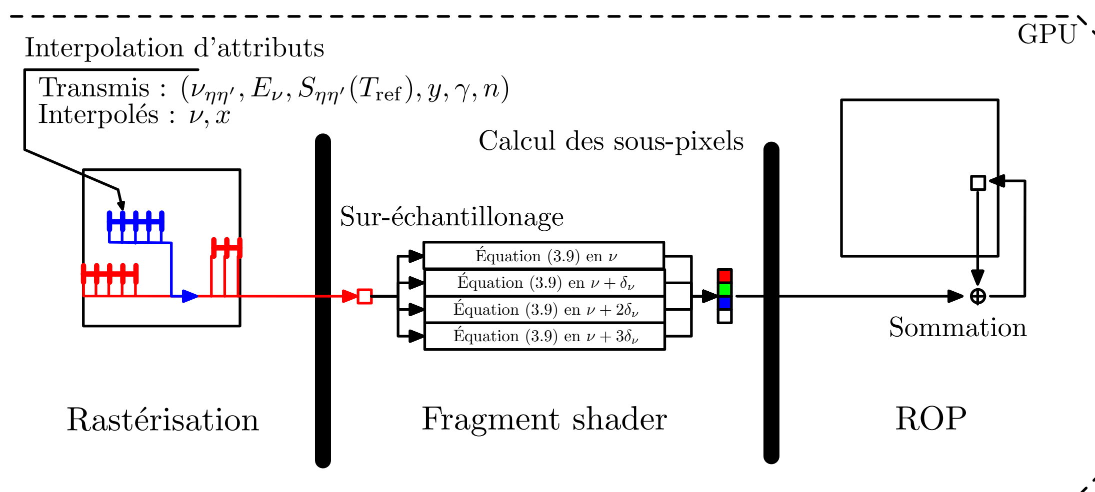


FIGURE 3.4 – Suite du traitement sur GPU : calcul de l'énergie absorbée sur chaque longueur d'onde.

Calcul de l'énergie absorbée Nous représentons le spectre du rayon entrant dans le volume sous forme échantillonnée dans une texture. Cette étape étant entièrement parallèle, elle est effectuée dans les fragment shaders, chaque fragment représentant 4 échantillons des spectres. Ainsi, nous tirons parti du parallélisme des vecteurs SIMD explicites de cette génération.

La sommation finale est calculée par une opération de réduction [Pha05]. À chaque étape de réduction, le fragment shader est utilisé pour calculer les sommes partielles des valeurs de blocs de 4 pixels.

On remarque que l'ordre des opérations n'est pas le même suivant que l'on utilise une méthode de réduction ou une accumulation récursive traditionnelle. L'addition flottante n'étant pas associative, le résultat final pourra être différent suivant la méthode utilisée. La méthode par réduction (sommation deux-à-deux) utilisée par le GPU aura une borne sur l'erreur directe inférieure à celle de la somme récursive pratiquée par le CPU [Hig02].

3.3.3 Résultats

Les résultats de cette étape sur le GPU NVIDIA G70 sont présentés sur la figure 3.5. Ils comprennent la performance du code de référence sur un CPU Pentium 4 630 de même génération. Le code est compilé pour utiliser les jeux d'instructions SSE et SSE2.

Le facteur deux entre les performances obtenues en utilisant des textures 16 bits et 32 bits révèle que cette étape est limitée par la bande passante mémoire sur le GPU. À l'inverse, changer le format des données n'a pas d'influence significative sur la performance du CPU. Le temps de traitement sur CPU est donc limité par les calculs eux-même. Une des raisons de cette différence est la présence de divisions dans le calcul. En effet, les GPU possèdent des unités d'évaluation de la fonction inverse câblées en matériel et entièrement pipelinées. À l'inverse, le CPU calcule les divisions par une méthode itérative non pipelinée, nécessitant 40 cycles d'horloge sur le Pentium 4 Prescott considéré ici.

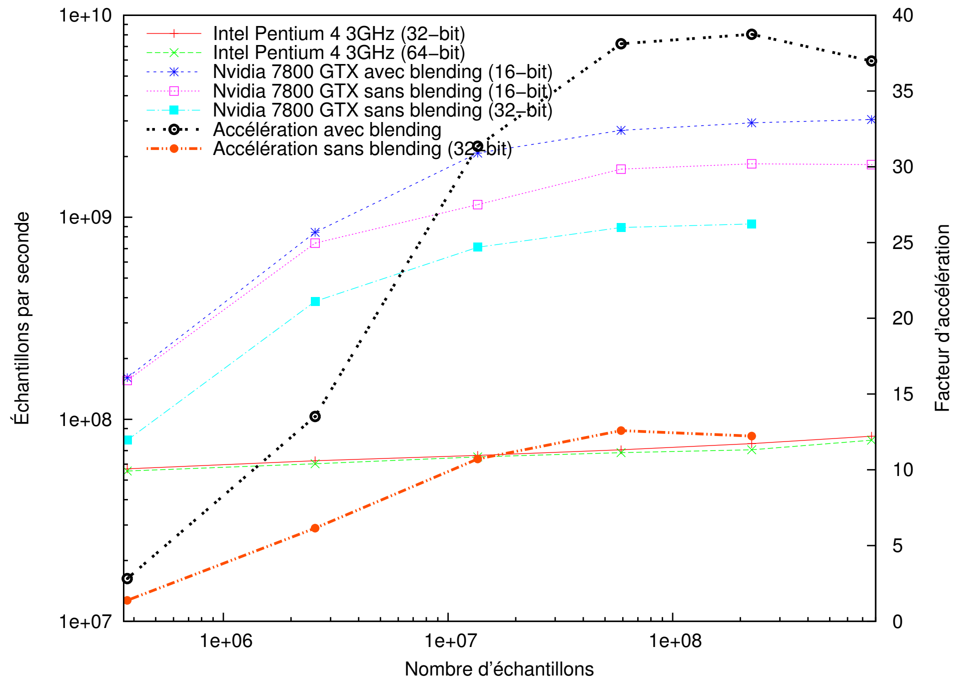


FIGURE 3.5 – Performance de la première étape – équations (3.8) à (3.10).

L'émulation de l'opération de blending dans les fragment shaders double approximativement la bande passante requise. Cela explique le facteur deux obtenu entre les deux méthodes de fusion pour des jeux de données suffisants (plus de $2 \cdot 10^6$ échantillons). Considérant ces résultats entièrement dépendants de la bande passante, nous pouvons supposer qu'un GPU ayant des unités de blending pouvant travailler en 32 bits fonctionnerait environ deux fois plus lentement qu'en mode 16 bits. Cela représenterait un facteur d'accélération de 20 à 30 par rapport à la version tournant sur le CPU.

Notre seconde configuration de test à base d'ATI R520 présente des résultats similaires lors de l'utilisation de textures 16 bits. Les ralentissements majeurs voire l'interruption de l'exécution observés pour les plus grandes tailles de problème sont dus à un manque de mémoire graphique, mais aussi de mémoire principale.

Calcul de l'énergie absorbée Une comparaison des performances de notre implémentation sur les fragment shaders de la carte NVIDIA GeForce 7800 GTX et le processeur Pentium 4 de référence est représentée figure 3.6. Le nombre d'échantillons calculés par seconde est présenté en fonction du nombre d'échantillons après fusion. Ces résultats ont été obtenus en faisant varier la taille de la texture utilisée.

Cette deuxième étape présente un facteur d'accélération proche de 330 pour $5 \cdot 10^6$ échantillons. La présence d'exponentielles dans le calcul de l'équation (3.8) permet de bénéficier des unités de fonctions spéciales câblées des GPU. Le calcul est ainsi effectué à un débit proche de celui de la mémoire.

Chaque étape de l'opération de réduction ne sollicitant que très peu les unités de calcul,

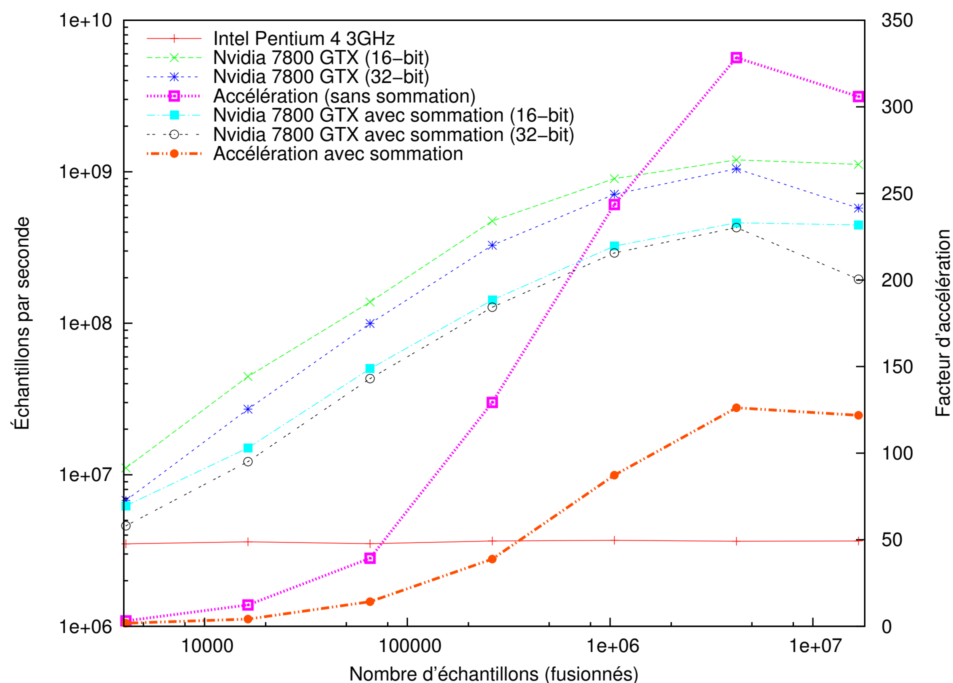


FIGURE 3.6 – Performance de la seconde étape – équation (3.11) et accumulation.

les performances de l'intégration sont entièrement limitées par la bande passante mémoire. À l'inverse, le surcoût dû à l'opération de sommation sur le CPU est négligeable, étant donné que cela nécessite uniquement l'utilisation d'un registre supplémentaire et d'une opération d'addition à chaque itération.

Conclusion Nous avons montré qu'il était possible de mettre à contribution l'ensemble du pipeline graphique sur les générations de GPU possédant des vertex shaders et fragment shaders séparés à l'aide d'un mécanisme d'allocation de tâches manuel. Au delà des unités de fragment shaders utilisées en GPGPU, nous avons exploité les unités spécialisées suivantes :

- vertex shader,
- rastérisation,
- interpolation d'attributs,
- test d'occlusion,
- blending.

Nous n'avons pas utilisé le filtrage de textures, celui-ci n'étant pas disponible en précision Binary32 sur les GPU considérés.

Ces résultats montrent qu'il existe des applications non liées au rendu graphique capables de tirer partie des unités spécialisées.

Plus spécifiquement, les gains que nous avons obtenus s'expliquent principalement par la présence des unités matérielles d'évaluation de fonctions spéciales sur les GPU. Ce choix, qui a été abandonné depuis les années 1980 sur les processeurs généralistes conformément à la philosophie RISC, est aujourd'hui remis en cause par les processeurs spécialisés tels que les

GPU. En effet, au delà du gain de performance qu'elles offrent, de telles unités spécialisées sont bien plus efficaces d'un point de vue énergétique qu'une implémentation logicielle. L'énergie devenant une ressource plus critique que la surface, il devient maintenant raisonnable de dédier de la surface de silicium à des unités spécialisées, malgré un faible taux d'utilisation.

Nous avons montré que les unités d'évaluation de fonctions dédiées au rendu graphique pouvaient également être utilisées par des applications de modélisation physique. Cela pose la question du compromis à adopter entre leur efficacité et leur flexibilité. Nous tenterons d'explorer d'autres compromis dans les sections 5.2.2 et 5.6.

3.4 Évaluation de fonction par filtrage

Nous avons noté dans la section précédente que les unités matérielles d'évaluation de fonctions élémentaires permettaient d'obtenir des facteurs d'accélération entre CPU et GPU qui dépassent le ratio de leurs performances crête. Cependant, ces unités se restreignent aux fonctions couramment utilisées dans les applications de rendu graphique : inverse, racine carrée inverse, logarithme et exponentielle, sinus et cosinus. Nous considérons dans cette section d'autres fonctions élémentaires ou composites, qui ne disposent pas de support matériel.

3.4.1 Détourner l'unité de textures

Nous cherchons à approximer une fonction $f(x)$ sur un intervalle donné. Nous supposons que l'argument x est un flottant Binary32 dans l'intervalle $[0, 1]$.

Dans le cadre du rendu graphique sur GPU, les approximations par splines et B-splines sont courantes, et tirent avantage du filtrage de textures. Sigg et al. décrivent comment effectuer du filtrage de textures d'ordre 3 en se basant sur le filtrage bilinéaire [Pha05]. Une autre technique consiste à approximer la fonction par un ou plusieurs polynômes évalués dans les unités arithmétiques. L'avantage de cette méthode est sa précision, chaque opération étant effectuée en arithmétique virgule flottante. Cependant, ces deux méthodes occupent les unités généralistes qui pourraient être utilisées à d'autres fins.

Approximation polynomiale d'ordre 2 Nous proposons d'effectuer l'interpolation polynomiale par l'intermédiaire des unités de filtrage de textures. Notre méthode est applicable lorsque la précision de la représentation (nombre de bits) en entrée est limitée mais qu'une précision importante est nécessaire en sortie.

Nous segmentons le domaine $[0, 1]$ de $f(x)$ en k intervalles régulièrement espacés, et approximons f par un polynôme de degré 2 dans chaque intervalle. Soit P_w le polynôme utilisé pour approximer $f(x)$ sur $[w/k, (w + 1)/k]$:

$$P_w(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2. \tag{3.13}$$

L'idée de cette méthode est d'ajuster le contenu de la texture T et les coordonnées de lecture dans la texture de manière à ce que que les calculs du filtrage bilinéaire réalisent l'évaluation du polynôme P_w.

Nous avons abordé dans la section 1.2.8 la formule de filtrage bilinéaire aux coordonnées $(i+\alpha, j+\beta)$, $i, j \in \mathbb{Z}$ en fonction des texels adjacents :

$$\begin{array}{rcll} V & = & (1-\alpha)\cdot(1-\beta) & T[i,j] \\ & & +(1-\alpha)\cdot\beta & T[i+1,j] \\ & & +\alpha\cdot(1-\beta) & T[i,j+1] \\ & & +\alpha\cdot\beta & T[i+1,j+1] \end{array} \tag{3.14}$$

Si nous accédons à la texture à des coordonnées calculées de telle sorte que $\alpha = \beta = \gamma$, $j = 0$ et que i soit multiple de 2, alors (3.14) est équivalente à :

$$\begin{array}{rcl} V & = & (1-\gamma)^2\cdot T[i,0] + (\gamma-\gamma^2)\cdot T[i+1,0] \\ & & +(\gamma-\gamma^2)\cdot T[i,1] + \gamma^2\cdot T[i+1,1]. \end{array} \tag{3.15}$$

Observons que l'expression obtenue est un polynôme de degré 2 en γ. Nous pouvons identifier les coefficients de l'équation de filtrage avec ceux de P_w. En effet, si l'on place dans la texture les valeurs $T[i,0] = a_0$, $T[i+1,0] + T[i,1] = 2\cdot a_0 + a_1$ et $T[i+1,1] = a_0 + a_1 + a_2$, nous obtenons le polynôme P_w :

$$\begin{array}{rcl} V & = & (1-\gamma)^2\cdot a_0 + (\gamma-\gamma^2)\cdot(2\cdot a_0 + a_1) \\ & & +\gamma^2\cdot(a_0 + a_1 + a_2) \\ & = & a_0 + a_1\cdot\gamma + a_2\cdot\gamma^2. \end{array} \tag{3.16}$$

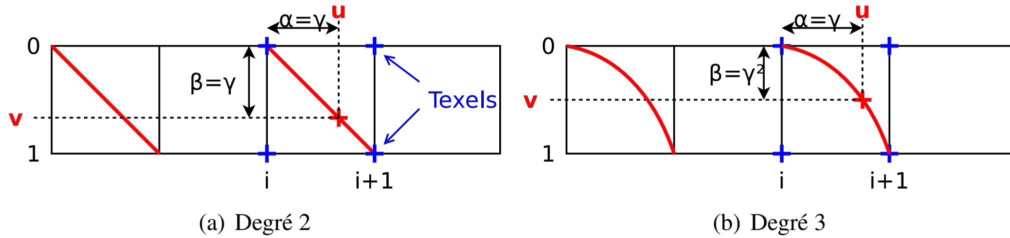
Les texels $T[i,0]$, $T[i+1,0]$, $T[i,1]$ et $T[i+1,1]$ sont utilisés pour les coefficients de chaque polynôme P_w. Pour maintenir les coefficients de k approximations polynomiales sur k intervalles successifs en évitant les recouvrements entre deux segments, les $4k$ coefficients doivent être placés dans une texture de taille $2k \times 2$.

Approximation d'ordre 3 Une technique similaire est applicable à des polynômes de degré 3. Nous calculons cette fois les coordonnées de texture de manière à ce que $\alpha = \gamma$ et $\beta = \gamma^2$. Les coefficients à placer dans la texture pour évaluer le polynôme $P_w(x) = a_0 + a_1\cdot x + a_2\cdot x^2 + a_3\cdot x^3$ sont alors :

$$\begin{array}{rcl} T[i,0] & = & a_0 \\ T[i+1,0] & = & a_0 + a_1 \\ T[i,1] & = & a_0 + a_2 \\ T[i+1,1] & = & a_0 + a_1 + a_2 + a_3. \end{array} \tag{3.17}$$

La figure 3.7 illustre l'interprétation géométique de chacune de ces techniques d'évaluation.

Évaluations multiples Bien que l'évaluation des polynômes soit réalisée par l'unité de filtrage de textures, les calculs d'adresses nécessitent toujours la contribution des unités arithmétiques. Il est toutefois possible de partager ces calculs d'adresses pour évaluer jusqu'à quatre fonctions différentes au même point. En effet, les unités de filtrage acceptent des texels contenant jusqu'à quatre composantes. Nous verrons dans la section suivante qu'appliquer le filtrage à des vecteurs ne cause pas de surcoût notable par rapport à un filtrage sur des scalaires.



(a) Degré 2

(b) Degré 3

FIGURE 3.7 – Interprétation géométrique de l'évaluation de fonction par filtrage.

Précision Les unités de filtrage de textures se basent sur des chemins de données virgule fixe en interne plutôt que d'effectuer les calculs intermédiaires en arithmétique virgule flottante IEEE. Par exemple sur Tesla, la partie fractionnaire γ est convertie dans un format virgule fixe avec 8 bits de partie fractionnaire [NVI10b]. Cette conversion restreint l'usage de ces méthodes à des précisions d'entrée inférieures à $log_2(k) + 8$. Au-delà, l'erreur de quantisation domine le calcul.

3.4.2 Application : système logarithmique

Le système logarithmique (LNS) est un système de représentation des nombres en machine. Il représente un nombre X par son logarithme en base b, $x = \log_b |X|$. Cela revient à maintenir uniquement un exposant, là ou le système virgule flottante utilisera un exposant et une mantisse. Pour obtenir une précision comparable à un système virgule flottante équivalent, nous permettons à l'exposant LNS d'avoir une partie fractionnaire. Comme en virgule flottante, la base b utilisée sera généralement 2.

L'exposant est typiquement représenté dans un format virgule fixe. Ainsi, la partie entière de l'exposant LNS est identique à l'exposant du format à virgule flottante équivalent. Les représentations de deux systèmes coïncident sur les puissances de deux, mais la distribution des nombres diffère entre ces points : elle est uniforme en virgule flottante et exponentielle en LNS.

De la même manière qu'en virgule flottante, un bit de signe additionnel est nécessaire pour représenter des nombres négatifs. Zéro nécessite également d'être traité comme un cas particulier dans la représentation et dans les calculs.

L'évaluation des produits, quotients, racines et puissances est particulièrement aisée en LNS. En effet, ceux-ci correspondent respectivement à des sommes, différences, décalages vers la droite et décalages vers la gauche sur les exposants. Les sommes et les différences sont plus délicates. Elles nécessitent d'évaluer respectivement les fonctions $s_b(x) = \log_b(1 + b^x)$ et $d_b(x) = \log_b |1 - b^x|$. En effet, pour $X = b^x$ et $Y = b^y$ deux nombres LNS positifs :

$$log_b(X + Y) = x + s_b(y - x) \tag{3.18}$$

$$log_b(X - Y) = x + d_b(y - x) \text{ pour} X \geq Y. \tag{3.19}$$

Le LNS est adapté aux applications embarquées qui réclament une dynamique importante tout en tolérant une précision réduite. Les implémentations matérielles du LNS fournissent des alternatives à faible consommation et faible latence aux unités en virgule flottante pour des

applications telles que la FFT, les modèles de Markov cachés [YKO+09] ou le problème des n corps [MT98]. Des implémentations sur FPGA sont disponibles sous forme de bibliothèques paramétrables [CdDD06, VCA07].

La performance d'une implémentation matérielle ou logicielle du LNS est entièrement dépendante de sa capacité à évaluer efficacement les fonctions s_b et d_b. Le LNS constitue donc une application aux travaux sur l'évaluation de fonctions continues.

Nous proposons d'appliquer notre méthode d'évaluation de fonctions pour évaluer les fonctions LNS s_b et d_b.

Contrairement à l'approche usuelle, nous adopterons une représentation en virgule flottante pour représenter l'exposant des nombres LNS. Ce système hybride FPLNS a la caractéristique inhabituelle de permettre une dynamique de représentation astronomique, aux dépens de la précision des nombres extrêmes. Cette caractéristique peut s'avérer avantageuse pour les calculs de probabilité de type vraisemblance maximale ou les réseaux bayésiens [GG03].

3.4.3 Résultats et validation

Nous testons les méthodes d'évaluation décrites ci-dessus en générant des polynômes minimax (minimisant la norme infinie sur un intervalle donné) à coefficients Binary32 avec l'outil Sollya [BC07]. Nous comparons ces méthodes d'ordre 2 et 3 avec :

- une interpolation linéaire utilisant l'unité de filtrage,
- un calcul basé sur les unités matérielles d'évaluation de fonctions exponentielle et logarithme (notées SFU),
- une approximation polynomiale dans l'intervalle complet, évalué par le schéma de Horner, également générée avec Sollya.

La figure 3.8 présente l'erreur au pire cas de la fonction s_2 pour chaque méthode étudiée, en fonction du nombre d'intervalles k. Nous considérons une précision en entrée w telle que $k = 2^{w-8}$. La partie fractionnaire, que nous avons appelé γ dans la section 3.4.1, est représentable exactement sur 8 bits pour satisfaire les contraintes de précision de l'unité de filtrage de Tesla.

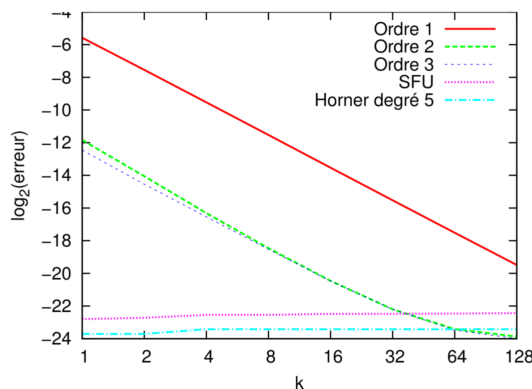


FIGURE 3.8 – Comparaison de l'erreur au pire cas des méthodes proposées sur la fonction s_2, avec $k = 2^{w-8}$.

Nous remarquons qu'il est nécessaire de conserver au minimum 64 segments sur cette fonction pour être en mesure de fournir 23 bits corrects en sortie. Notons que la précision en entrée est alors limitée à 14 bits. Cela correspond à une texture de dimensions 2×128, qui tient dans le cache de textures. Le passage à l'ordre 3 n'apporte pas de gain significatif en précision sur cette fonction régulière.

Évaluons l'efficacité de chacune de ces méthodes, appliquées à l'évaluation de la fonction s_2, ou à l'évaluation simultanée de s_2 et d_2 en un même point. Nous considérons également l'utilisation combinée de deux méthodes, dans le but de maximiser le débit. Les résultats obtenus sur un GPU GT200 sont présentés table 3.5. Ils sont exprimés en nombre de cycles par warp par SM à la fréquence SP.

TABLE 3.5 – Comparaison des performances des différentes implémentations proposées, sur GT200 en cycles d'horloge SP.

Méthode	s_2	s_2 et d_2
Ordre 1	42,5	42,5
Ordre 2	49,0	49,0
Ordre 3	49,5	49,5
SFU	32,0	33,0
Horner degré 5	35,0	44,0
Ordre 2 + SFU	46,0	52,5
$2\times$ SFU	40,0	56,0

Lorsque nous utilisons les unités d'évaluation matérielles pour calculer à la fois s_2 et d_2 comme $\log_2(1 \pm 2^x)$, les optimisations du compilateur permettent à l'expression commune 2^x de n'être calculée qu'une seule fois. L'intérêt des interpolations d'ordre 2 et 3 est limité lorsqu'elles sont utilisées seules ou pour évaluer une seule fonction. Cependant, elle permettent d'augmenter le débit de calcul pour calculer s_2 et d_2 simultanément en conjonction avec les unités SFU.

Conclusion Nous avons proposé un moyen d'exploiter les unités spécialisées de filtrage de textures des GPU pour évaluer des fonctions. Cela permet en particulier d'augmenter le débit de calcul pour les fonctions de base du LNS. D'autres applications qui nécessitent des approximations de fonctions continues avec une faible précision d'entrée peuvent également être accélérées.

Le principal obstacle que nous avons rencontré est la précision limitée de l'échantillonnage des coefficients à l'entrée de l'unité de filtrage. L'élargissement des chemins de données de l'unité de filtrage est une fausse solution, car elle diminuerait le bénéfice d'une unité dédiée. Nous proposons plutôt de rendre accessible l'unité d'interpolation d'attributs dans l'environnement CUDA. Cette unité ne souffre pas des mêmes limitations en précision que l'unité de filtrage de textures. Un accès direct permettrait l'évaluation précise de fonctions arbitraires, à la manière dont sont évaluées les fonctions de base en matériel [OS05].

3.5 Arithmétique d'intervalles sur GPU

L'aperçu des capacités arithmétiques des GPU que nous avons présenté dans la section 3.2.4 met en évidence la présence de modes d'arrondis statiques sur les GPU NVIDIA. L'application principale des modes d'arrondis dirigés est l'arithmétique d'intervalles. Nous verrons que la possibilité de mélanger des opérations flottantes utilisant des modes d'arrondis différents permet une implémentation efficace de l'arithmétique d'intervalles.

Nous présenterons dans cette section une bibliothèque d'opérateurs de l'arithmétique d'intervalles ciblant les architectures NVIDIA Tesla et Fermi.

L'arithmétique d'intervalles trouve des applications en recherche opérationnelle, en analyse numérique et dans d'autres domaines pour estimer et réduire les erreurs d'arrondis [Han79] et caractériser la tolérance par rapport aux paramètres [KK96]. Nous considérerons en particulier une application au rendu d'images de synthèse.

Application Le lancer de rayon sur surfaces implicites permet de visualiser des objets décrits par des équations implicites sans avoir à passer par une représentation intermédiaire par maillage. L'approche classique présente cependant des problèmes de fiabilité numérique [CHMS00, KB89, Mit90]. Ces problèmes se posent lorsque des détails de surfaces implicites ne sont pas reproduits correctement et disparaissent.

En effet, le test d'intersection entre la fonction implicite et un rayon consiste à résoudre une équation, c'est-à-dire à chercher les racines d'une fonction. Lorsqu'une solution existe mais que l'algorithme utilisé échoue à la trouver, des aberrations visuelles peuvent se produire. Une solution est d'effectuer la recherche de racines par la méthode de Newton en arithmétique d'intervalles afin d'obtenir un résultat garanti.

La méthode de Newton par intervalles permet de résoudre des systèmes non linéaires [HG83, Rev01]. Cette méthode itérative consiste à raffiner l'ensemble des solutions potentielles en excluant les zones situées hors d'un cône englobant la fonction considérée. L'utilisation de l'arithmétique d'intervalles permet de garantir que toutes les solutions sont situées dans les intervalles renvoyés.

Cette approche est cependant coûteuse en temps de calcul [FSSV06]. L'application étant massivement parallèle, une implémentation sur GPU est attrayante.

Les implémentations actuelles de l'arithmétique d'intervalle sur les architectures SIMD s'appuient sur des modes d'arrondis qui ne sont pas disponibles sur tous les GPU [Lam08], ou ignorent purement et simplement les problèmes d'arrondis, retournant potentiellement des résultats incorrects [KHH$^+$07]. De fait, il est nécessaire de développer une implémentation d'opérateurs d'arithmétique d'intervalles sur GPU.

Nous définirons l'arithmétique d'intervalles dans la section 3.5.1. Nous établirons différentes techniques pour approximer les modes d'arrondis non disponibles en matériel section 3.5.2, puis pour les calculer correctement section 3.5.3.

3.5.1 Arithmétique d'intervalles

Définitions L'arithmétique d'intervalles permet de calculer des bornes fiables en prenant en compte les incertitudes sur les données. Elle consiste à redéfinir les opérations usuelles sur des

intervalles de valeurs.

La représentation infinum-supremum est la plus fréquemment utilisée pour représenter les intervalles. Dans ce système, les intervalles sont caractérisés par leur borne inférieure et leur borne supérieure. Nous utiliserons la notation usuelle pour représenter les intervalles bornés :

$$[a_1, a_2] = \{x : a_1 \leq x \leq a_2\} \quad \text{pour des bornes} \quad a_1 \leq a_2.$$

Nous définissons l'ensemble des intervalles bornés $I(\mathbb{R})$ sur les réels :

$$I(\mathbb{R}) = \{[a_1, a_2] : (a_1, a_2) \in \mathbb{R}^2, a_1 \leq a_2\},$$

sur lequel nous pouvons redéfinir les opérateurs usuels $+, -, \times, /$ de $\mathbb{R}$.

Soit $A = [a_1, a_2]$ et $B = [b_1, b_2]$. Définissons :

$$\begin{array}{rcl} A+B &=& [a_1+b_1, a_2+b_2] \\ A-B &=& [a_1-b_2, a_2-b_1] \\ A\times B &=& [min(a_1b_1, a_2b_2, a_1b_2, a_2b_1), max(a_1b_1, a_2b_2, a_1b_2, a_2b_1)] \\ A/B &=& [min(a_1/b_1, a_2/b_2, a_1/b_2, a_2/b_1), \\ && max(a_1/b_1, a_2/b_2, a_1/b_2, a_2/b_1)] \text{ lorsque } 0 \notin B \end{array} \quad (3.20)$$

Les implémentations en machine de l'arithmétique d'intervalles sont typiquement basées sur des nombres à virgule flottante. Elles s'appuient sur les modes d'arrondis dirigés décrits section 3.1, et en particulier les arrondis vers $+\infty$ et $-\infty$. Les arrondis dirigés permettent de borner les erreurs d'arrondis de l'arithmétique flottante.

Une implémentation en virgule flottante des opérateurs de l'arithmétique d'intervalle $I(\mathbb{F}_p)$ deviendra alors, toujours avec $A = [a_1, a_2]$ et $B = [b_1, b_2]$:

$$\begin{array}{rcl} A+B &=& [\triangledown(a_1+b_1), \triangle(a_2+b_2)] \\ A-B &=& [\triangledown(a_1-b_2), \triangle(a_2-b_1)] \\ A\times B &=& [min(\triangledown(a_1b_1), \triangledown(a_2b_2), \triangledown(a_1b_2), \triangledown(a_2b_1)), \\ && max(\triangle(a_1b_1), \triangle(a_2b_2), \triangle(a_1b_2), \triangle(a_2b_1))] \\ A/B &=& [min(\triangledown(a_1/b_1), \triangledown(a_2/b_2), \triangledown(a_1/b_2), \triangledown(a_2/b_1)), \\ && max(\triangle(a_1/b_1), \triangle(a_2/b_2), \triangle(a_1/b_2), \triangle(a_2/b_1))] \text{ lorsque } 0 \notin B \end{array} \quad (3.21)$$

Nous pouvons accepter les valeurs spéciales $+\infty$ et $-\infty$ comme bornes pour représenter des intervalles non bornés, et NaN pour représenter les intervalles vides ou illégaux.

La règle de base de l'arithmétique d'intervalles est la propriété d'*inclusion*.

Propriété 3.1 (inclusion) *Pour f une fonction et F son extension sur les intervalles, et A un intervalle, l'inclusion $f(A) \subset F(A)$ est vérifiée.*

Cette propriété fait que l'ensemble de solutions que renvoie le calcul d'une expression en arithmétique d'intervalles doit toujours inclure au moins toutes les solutions réelles possibles.

L'effet de dépendance est un facteur qui limite l'efficacité de l'arithmétique d'intervalles. Le problème intervient lorsque la même variable apparaît plusieurs fois dans une expression ou un calcul. Lorsque le calcul est effectué en arithmétique d'intervalles, l'information que les deux valeurs sont égales est perdue. Par exemple, pour $x = [-2, 3]$, $x \times x = [-6, 9]$, alors que $x^2 = [0, 9]$.

Implémentations existantes Il existe plusieurs bibliothèques d'arithmétique d'intervalles ayant leurs spécificités propres [KDDH94, Lam08], comme MPFI pour les intervalles multiprécision [RR05]. L'utilisation de la surcharge d'opérateurs permet de simplifier l'interface exposée au programmeur. La suite de bibliothèques C++ Boost inclut ainsi une bibliothèque d'arithmétique d'intervalles basées sur les templates et la surcharge d'opérateurs [BMP06]. Deux obstacles contraignent cependant le développement et l'efficacité des bibliothèques d'arithmétique d'intervalle.

L'obstacle logiciel est l'absence de support natif des modes d'arrondis dans les langages de programmation usuels. La gestion des modes d'arrondis est typiquement reléguée aux fonctions de la bibliothèque standard, voire à des environnements spécifiques au compilateur, au système d'exploitation ou à l'architecture. En dehors des problèmes de portabilité, cela limite la visibilité du compilateur sur le code, et en conséquence les optimisations qui lui sont accessibles.

L'obstacle matériel est le surcoût significatif qu'un changement de mode d'arrondi implique sur la plupart des architectures actuelles. Un processeur superscalaire va typiquement pipeliner les instructions arithmétiques en spéculant que le registre de configuration de l'unité flottante ne change pas. Lorsque ce registre est modifié, le pipeline doit être vidé, et les instructions redémarrées. Ainsi, nous avons observé des ralentissements pouvant atteindre un facteur 20 sur Intel Core 2 Duo entre des calculs effectués dans le même mode d'arrondi et des calculs alternant entre plusieurs modes d'arrondis.

Pour contourner ce second obstacle, il est possible de reformuler les calculs des bornes des équations (3.21) pour que tous les calculs soient effectués dans le même mode d'arrondi [BMP06]. En effet, on peut remarquer que $\triangle(a+b) = -\bigtriangledown((-a)-b)$ et $\triangle(a \cdot b) = -\bigtriangledown((-a) \cdot b)$. Cependant, les calculs effectués en arithmétique flottante classique doivent toujours être évalués en arrondi au plus près. Il est donc encore nécessaire de basculer le mode d'arrondi en entrant et en sortant des routines d'arithmétique d'intervalles. Cet inconvénient est aggravé par l'impossibilité pour le compilateur de réordonner les instructions pour minimiser les changements de mode.

Nous avons vu section 3.2.4 que les architectures Tesla et Fermi permettaient de sélectionner le mode d'arrondi de chaque opération de manière statique. Le mode d'arrondi faisant partie de l'instruction plutôt que de l'état architectural du processeur, les opérations dans des modes d'arrondis différents peuvent êtres pipelinées sans surcoût, à l'instar des architectures Alpha ou IA-64.

3.5.2 Simuler les modes d'arrondis manquants

L'architecture Tesla offre des opérations Binary32 en mode d'arrondi au plus près et vers zéro, mais ne permet pas les arrondis vers $\pm\infty$. Nous devons donc contourner cette limitation.

Des solutions basées sur l'arrondi au plus près ont été conçues pour le cas où les modes d'arrondis dirigés ne sont pas disponibles ou trop coûteux [KDDH94, RZBM09]. Cependant, elles induisent une surestimation importante de l'erreur lors des calculs sur des intervalles étroits.

Nous nous baserons ici sur le mode d'arrondi vers zéro, qui est disponible sur tous les GPU Tesla.

Arrondis dirigés approchés Nous avons posé section 3.1 que suivant le signe de la valeur à arrondir, l'arrondi vers zéro était équivalent soit à l'arrondi vers $-\infty$, soit à l'arrondi vers $+\infty$. L'autre direction peut être approximée en ajoutant un ulp à la valeur arrondie vers le bas, ou en soustrayant un ulp à la valeur arrondie vers le haut. Cela revient à considérer le flottant immédiatement supérieur en valeur absolue.

Définition 3.2 *La valeur incrémentée $\sigma(x)$ d'un flottant x est le flottant de même signe immédiatement supérieur en valeur absolue.*

Notons que lorsque le résultat est exactement représentable dans le format flottant, une surestimation de l'erreur d'arrondi est commise. Cette surestimation peu fréquente est généralement tolérable, car elle n'affecte pas la propriété d'inclusion de l'arithmétique d'intervalles. Nous répondrons à cette considération dans la section 3.5.3.

Il est possible d'utiliser les propriétés du format de représentation des flottants IEEE 754 pour ajouter ou supprimer un ulp à un flottant. Une technique assez courante consiste à incrémenter la représentation binaire de la valeur flottante. En effet, les flottants consécutifs ont des représentations binaires consécutives. Cette solution est attrayante, mais les valeurs spéciales ($\pm\infty$, NaN) doivent faire l'objet d'un traitement spécifique. Une solution alternative que nous préférerons dans le cas des GPU consiste à multiplier le résultat par $1 + \epsilon$ en arrondi vers zéro. Zéro doit toujours être considéré séparément, du fait de l'absence de gestion des dénormaux sur les GPU visés.

Nous calculons l'arrondi vers zéro ainsi que la valeur incrémentée de cet arrondi, puis sélectionnons celui qui correspond au mode souhaité ($+\infty$ ou $-\infty$). Comme $\bigtriangledown(x) \leq \bigtriangleup(x)$, nous pouvons utiliser les opérations $\min$ et $\max$ pour effectuer cette sélection.

$$\bigtriangledown(x) \geq \tilde{\bigtriangledown}(x) = \min(\mathcal{Z}(x), \sigma(\mathcal{Z}(x))) \tag{3.22}$$

$$\bigtriangleup(x) \leq \tilde{\bigtriangleup}(x) = \max(\mathcal{Z}(x), \sigma(\mathcal{Z}(x))) \tag{3.23}$$

Afin d'alléger les notations, nous assimilerons les arrondis approchés $\tilde{\bigtriangledown}(x)$ et $\tilde{\bigtriangleup}(x)$ aux arrondis corrects $\bigtriangledown(x)$ et $\bigtriangleup(x)$, bien qu'ils puissent différer d'un ulp lorsque x est exactement représentable sous la forme d'un flottant.

Addition Nous appliquons directement la technique de compensation des arrondis à l'addition d'intervalle, présentée algorithme 3.1.

Algorithme 3.1 Calcul de $[r_{inf}, r_{sup}] = [x_{inf}, x_{sup}] + [y_{inf}, y_{sup}]$

$a \leftarrow \mathcal{Z}(x_{inf} + y_{inf})$
$b \leftarrow \mathcal{Z}(x_{sup} + y_{sup})$
$r_{inf} \leftarrow \min(a, \sigma(a))$
$r_{sup} \leftarrow \max(b, \sigma(b))$

Multiplication L'implémentation utilisée dans la bibliothèque Boost.Interval consiste en 4 niveaux de blocs conditionnels imbriqués pour distinguer les 13 différents cas possibles suivant le signe de chaque intervalle d'entrée (strictement positif, strictement négatif, intersectant

zéro, nul). Cette approche est raisonnablement efficace en pratique, car les prédicteurs de branchements des processeurs actuels permettent de s'adapter dynamiquement à des régularités dans les données manipulées.

Sur GPU, les instructions de saut ont un coût comparable à des multiplications, même en l'absence de divergence. Il est souhaitable de s'en affranchir, comme nous l'avons fait pour l'addition.

Nous repartons de la formule présentée section 3.5.1, qui ne nécessite pas de branchements.

$$[a,b] \times [c,d] = [\min(\triangledown(ac), \triangledown(ad), \triangledown(bc), \triangledown(bd)), \max(\triangle(ac), \triangle(ad), \triangle(bc), \triangle(bd))] \tag{3.24}$$

L'approche naïve consiste à émuler chaque arrondi dirigé avec un arrondi vers zéro et une opération d'incrémentation σ. Pour chaque produit $x \cdot y$, nous devons calculer les expressions suivantes pour obtenir les approximations de l'arrondi inférieur l et de l'arrondi supérieur u.

$$\begin{array}{rcl} \lambda & \leftarrow & \mathcal{Z}(x \cdot y) \\ \mu & \leftarrow & \sigma(\lambda) \\ l & \leftarrow & \min(\lambda, \mu) \\ u & \leftarrow & \max(\lambda, \mu) \end{array} \tag{3.25}$$

Cette méthode requiert un total de 4 multiplications, 4 incrémentations σ et 14 opérations $\min$ et $\max$.

Étudions maintenant le signe de la borne inférieure et de la borne supérieure du résultat en fonction des signes respectifs de a, b, c et d. Les cas possibles sont listés dans la table 3.6. Les nombres positifs sont dénotés par « $+$ » et les nombres négatifs par « $-$ ».

TABLE 3.6 – Signes des bornes de $[a, b] \times [c, d]$.

a	b	c	d	Borne inférieure		Borne supérieure	
$+$	$+$	$+$	$+$	$\triangledown(ac)$	$+$	$\triangle(bd)$	$+$
$-$	$+$	$+$	$+$	$\triangledown(ad)$	$-$	$\triangle(bd)$	$+$
$-$	$-$	$+$	$+$	$\triangledown(ad)$	$-$	$\triangle(bc)$	$-$
$+$	$+$	$-$	$+$	$\triangledown(bc)$	$-$	$\triangle(bd)$	$+$
$-$	$+$	$-$	$+$	$\min(\triangledown(bc), \triangledown(ad))$	$-$	$\max(\triangle(ac), \triangle(bd))$	$+$
$-$	$-$	$-$	$+$	$\triangledown(ad)$	$-$	$\triangle(ac)$	$+$
$+$	$+$	$-$	$-$	$\triangledown(bc)$	$-$	$\triangle(ad)$	$-$
$-$	$+$	$-$	$-$	$\triangledown(bc)$	$-$	$\triangle(ac)$	$+$
$-$	$-$	$-$	$-$	$\triangledown(bd)$	$+$	$\triangle(ac)$	$+$

Remarquons que les produits ac et bd sont toujours positifs tandis que ad et bc sont toujours négatifs dès lors qu'ils apparaissent dans le résultat. Ceci reste vrai quelle que soit la direction de l'arrondi. Cette information simplifie le calcul des arrondis vers le haut et vers le bas de ac, bd, ad, bc, en éliminant l'étape de sélection par les opérations $\min$ et $\max$.

En effet, nous en déduisons les égalités suivantes :

$$\begin{array}{ll} \bigtriangledown(ac) = \mathcal{Z}(ac) & \bigtriangleup(ac) = \sigma\mathcal{Z}(ac) \\ \bigtriangledown(bd) = \mathcal{Z}(bd) & \bigtriangleup(bd) = \sigma\mathcal{Z}(bd) \\ \bigtriangledown(ad) = \sigma\mathcal{Z}(ad) & \bigtriangleup(ad) = \mathcal{Z}(ad) \\ \bigtriangledown(bc) = \sigma\mathcal{Z}(bc) & \bigtriangleup(bc) = \mathcal{Z}(bc). \end{array} \tag{3.26}$$

Nous pouvons réinjecter ces égalités dans la formule (3.24) :

$$\begin{array}{rcl} \max(\bigtriangleup(ac), \bigtriangleup(bd)) & = & \max(\sigma\mathcal{Z}(ac), \sigma\mathcal{Z}(bd)) \\ \min(\bigtriangledown(ad), \bigtriangledown(bc)) & = & \min(\sigma\mathcal{Z}(ad), \sigma\mathcal{Z}(bc)). \end{array} \tag{3.27}$$

Nous pouvons réduire davantage le nombre d'opérations en remarquant que la fonction σ est monotone sur les valeurs positives. L'ordre est donc préservé. Il est alors possible de différer l'opération σ après le calcul des $\min$ et $\max$. Cela conduit à ces simplifications :

$$\begin{array}{rcl} \max(\bigtriangleup(ac), \bigtriangleup(bd)) & = & \sigma(\max(\mathcal{Z}(ac), \mathcal{Z}(bd))) \\ \min(\bigtriangledown(ad), \bigtriangledown(bc)) & = & \sigma(\min(\mathcal{Z}(ad), \mathcal{Z}(bc))). \end{array} \tag{3.28}$$

L'algorithme de multiplication 3.2 ne nécessite alors plus que 4 multiplications, 2 incrémentations σ et 6 $\min$ et $\max$, à contraster avec les 22 opérations initiales.

Algorithme 3.2 Calcul de $[r_{inf}, r_{sup}] = [x_{inf}, x_{sup}] \times [y_{inf}, y_{sup}]$

$\lambda_{ac} \leftarrow \mathcal{Z}(x_{inf} \times y_{inf})$
$\lambda_{bd} \leftarrow \mathcal{Z}(x_{sup} \times y_{sup})$
$\lambda_{ad} \leftarrow \mathcal{Z}(x_{inf} \times y_{sup})$
$\lambda_{bc} \leftarrow \mathcal{Z}(x_{sup} \times y_{inf})$
$\underline{l_1} \leftarrow \min(\lambda_{ac}, \lambda_{bd})$
$\overline{l_2} \leftarrow \min(\lambda_{ad}, \lambda_{bc})$
$\underline{l_2} \leftarrow \sigma(\overline{l_2})$
$\underline{u_1} \leftarrow \max(\lambda_{ac}, \lambda_{bd})$
$\overline{u_2} \leftarrow \max(\lambda_{ad}, \lambda_{bc})$
$\overline{u_1} \leftarrow \sigma(\underline{u_1})$
$r_{inf} \leftarrow \min(\underline{l_1}, \underline{l_2})$
$r_{sup} \leftarrow \max(\overline{u_1}, \overline{u_2})$

Élévation à une puissance entière constante Les calculs d'élévation au carré, cube et autres petites puissances ont une grande importance en arithmétique d'intervalles. En effet, elles ne peuvent pas être simplement réalisées par des multiplications d'intervalles, sous peine d'un élargissement de l'intervalle résultant du à l'effet de dépendance.

De manière similaire à la multiplication d'intervalles, nous étudions dans la table 3.7 les différents signes possibles du résultat en fonction des signes des opérandes et de la parité de l'exposant. Cela nous conduit également à une réduction du nombre d'instructions.

Pour le calcul de l'exponentiation proprement dite, nous nous basons sur la méthode d'exponentiation par décomposition binaire de l'exposant décrite par Knuth [Knu97].

TABLE 3.7 – Signes des bornes inférieures et supérieures de $[a, b]^n$.

a	b	n pair Borne inf.		n pair Borne sup.		n impair Borne inf.		n impair Borne sup.	
$+$	$+$	$\underline{a^n}$	$+$	$\overline{b^n}$	$+$	$\underline{a^n}$	$+$	$\overline{b^n}$	$+$
$-$	$+$	0	$+$	$\max(\overline{a^n}, \overline{b^n})$	$+$	$\underline{a^n}$	$-$	$\overline{b^n}$	$+$
$-$	$-$	$\underline{b^n}$	$+$	$\overline{a^n}$	$-$	$\underline{a^n}$	$-$	$\overline{b^n}$	$-$

Plutôt que de compenser individuellement chaque multiplication de l'exponentiation pour calculer l'arrondi extérieur à zéro, nous calculons la puissance entièrement en arrondi vers zéro, puis multiplions le résultat par une constante correspondant à une borne sur l'ensemble des erreurs commises.

Propriété 3.2 *Soit l'algorithme d'exponentiation en arrondi vers zéro $\underline{a^n}$ défini par induction :*

$$\begin{aligned} \underline{a^1} &= a \\ \underline{a^{2k}} &= \mathcal{Z}(\underline{a^k} \cdot \underline{a^k}) \\ \underline{a^{2k+1}} &= \mathcal{Z}(\mathcal{Z}(\underline{a^k} \cdot \underline{a^k}) \cdot a) \end{aligned}$$

Alors pour $n < 1/\epsilon - 2$, son erreur relative peut être bornée par :

$$\left| \frac{a^n - \underline{a^n}}{a^n} \right| \leq 1 + n\epsilon.$$

Démonstration. Supposons $a \geq 0$ sans perte de généralité. Établissons par récurrence une première borne $\underline{a^n} \leq a^n \cdot (1 + \epsilon)^{n-1}$.

- $\underline{a^1}$ satisfait trivialement la condition,
- $\underline{a^{2k}} = \mathcal{Z}(\underline{a^k} \cdot \underline{a^k}) \leq \underline{a^k} \cdot \underline{a^k} \cdot (1 + \epsilon) \leq a^{2k} \cdot (1 + \epsilon)^{2k-1}$,
- $\underline{a^{2k+1}} = \mathcal{Z}(\mathcal{Z}(\underline{a^k} \cdot \underline{a^k}) \cdot a) \leq \underline{a^k} \cdot \underline{a^k} \cdot a \cdot (1 + \epsilon)^2 \leq a^{2k+1} \cdot (1 + \epsilon)^{2k}$.

Pour $n < 1/\epsilon - 2$, nous pouvons élargir la borne par l'expression suivante [Hig02] :

$$\underline{a^n}/a^n \leq (1 + \epsilon)^{n-1} \leq 1 + \frac{(n-1)\epsilon}{1 - (n-1)\epsilon} \leq 1 + n\epsilon.$$

□

Pour les valeurs raisonnables de n (jusqu'à 8 million en Binary32), l'arrondi extérieur peut alors être borné par :

$$\overline{a^n} = \underline{a^n}(1 + n\epsilon). \tag{3.29}$$

La condition $|\underline{a^n}| \leq |a^n| \leq |\overline{a^n}|$ est alors respectée. De même, la détection des dépassements de capacité peut être repoussée à la fin du calcul.

Division et racine carrée Les GPU de la génération Direct3D 10 n'offrent pas de ressources matérielles permettant de calculer efficacement la division et la racine carrée avec arrondi correct. Ils approximent la division et la racine carrée par les formules suivantes :

$$x/y \approx \mathcal{N}(x \times \mathrm{rcp}(y))) \tag{3.30}$$
$$\sqrt{x} \approx \mathrm{rcp}(\mathrm{rsq}(x)), \tag{3.31}$$

où les fonctions rcp et rsq désignent respectivement l'inverse machine et la racine carrée inverse machine.

Notons que le choix d'utiliser l'équation (3.31) plutôt que $\sqrt{x} \approx \mathcal{N}(x \times \mathrm{rsq}(x))$ s'explique par l'absence de support spécifique pour les cas $x = 0$ et $x = \pm\infty$ dans les jeux d'instructions de ces GPU. Approximer l'inverse est plus efficace que de traiter séparément les cas spéciaux de la multiplication, au prix d'une perte de précision.

La documentation de NVIDIA CUDA fournit à titre informatif des bornes sur l'erreur entre le résultat renvoyé par rcp et rsq et l'arrondi correct au plus près [NVI10b].

$$|\mathcal{N}(x^{-1}) - \mathrm{rcp}(x)| \leq 1\,\mathrm{ulp}(\mathcal{N}(x^{-1})) \tag{3.32}$$
$$|\mathcal{N}(\sqrt{x}^{-1}) - \mathrm{rsq}(x)| \leq 2\,\mathrm{ulp}(\mathcal{N}(\sqrt{x}^{-1})) \tag{3.33}$$
$$|\mathcal{N}(x/y) - \mathcal{N}(x \times \mathrm{rcp}(y))| \leq 2\,\mathrm{ulp}(\mathcal{N}(x/y)) \tag{3.34}$$
$$|\mathcal{N}(\sqrt{x}) - \mathrm{rcp}(\mathrm{rsq}(x))| \leq 3\,\mathrm{ulp}(\mathcal{N}(\sqrt{x}) \tag{3.35}$$

En ajoutant 0,5 ulp de différence entre l'arrondi au plus près et la valeur exacte, nous pouvons encadrer l'erreur par une borne suffisante, mais pessimiste. La description détaillée des unités arithmétiques de Tesla nous fournit des bornes plus précises [OS05]. Nous pouvons dans tous les cas encadrer la valeur retournée par rcp et rsq, et effectuer l'opération de multiplication en arithmétique d'intervalles.

3.5.3 Arrondis dirigés corrects

Les algorithmes utilisés pour simuler les modes d'arrondis dirigés que nous avons abordé jusqu'ici n'offrent pas l'arrondi correct dans tous les cas. Lorsque le résultat est exactement représentable sous la forme d'un nombre flottant, l'erreur d'arrondi sera surestimée. Lorsque cette erreur n'est pas tolérable, il nous faut être en mesure de détecter si le résultat de l'opération est exact et d'ajuster le résultat en conséquence.

Les processeurs compatibles IEEE-754 lèvent un drapeau *inexact* lorsqu'une valeur subit un arrondi. Cependant, peu de GPU actuels respectent cette section de la norme (section 3.2.4).

Addition Pour nous permettre de déterminer si une addition est exacte, nous établirons le résultat suivant.

Propriété 3.3 *Soient $a, b, r, s \in \mathbb{F}_p$ tels que :*

$$r \leftarrow \mathcal{Z}(a + b)$$
$$s \leftarrow \mathcal{Z}(r - a)$$

Alors l'équivalence suivante est vérifiée :

$$s = b \iff r = a + b$$

Ainsi, les deux opérations en arrondi vers zéro décrites ci-dessus nous permettent de détecter si la somme de deux flottants a et b est exactement représentable sous la forme d'un flottant.

Remarquons la ressemblance de cet algorithme avec l'algorithme Fast2Sum de Knuth, qui calcule l'erreur d'une addition flottante [Knu97]. Néanmoins, deux conditions préalables sont nécessaires à l'utilisation de Fast2Sum. Il faut que $|a| > |b|$, et que les opérations soient effectuées en arrondi au plus près. Dans notre cas, aucune de ces deux conditions n'est respectée. À l'inverse, notre algorithme ne fonctionnerait pas en arrondi au plus près : un contre-exemple est $a = 2^{-100}$ et $b = 1$, qui entraine $r = 1$ et $s = 1 = b$, alors que $r \neq a + b$.

Démonstration. Une étude de cas nous montre que les erreurs d'arrondis pouvant être commises lors du calcul de r et s se produisent systématiquement dans la même direction et ne se compensent pas. De fait, lorsqu'une erreur d'arrondi se produit, elle aura nécessairement un impact sur s.

Découpons l'intervalle de b et considérons chaque sous-intervalle, pour $a \geq 0$:

1. $0 \leq a \leq b$.
   - Supposons $s = b$. Comme $a + b \geq 0$, on peut borner $r = \mathcal{Z}(a + b) \leq a + b$, soit $r - a \leq b$.
     D'autre part, comme $r - a \geq 0$, on a également $b = s = \mathcal{Z}(r - a) \leq r - a$. Nous déduisons de l'encadrement $b \leq r - a \leq b$ que $r - a = b$, soit $r = a + b$.
   - Supposons $r = a + b$. Réécrivons $s = \mathcal{Z}(r - a) = \mathcal{Z}(b)$. Comme b est un flottant, l'arrondi n'a pas lieu et $\mathcal{Z}(b) = b$, d'où $s = b$.
2. $b \leq -a \leq 0$. Par un raisonnement analogue à celui du cas 1, mais en utilisant l'autre borne de l'arrondi vers zéro appliquée aux négatifs, on obtient le résultat désiré.
3. $-\frac{1}{2}a \leq b \leq a$, $a \geq 0$.
   Par addition de a dans l'inéquation, $0 \leq \frac{1}{2}a \leq a + b \leq 2a$. En base deux et en l'absence de dépassement de capacité inférieure, $\frac{1}{2}a$ est un flottant. Nous pouvons alors borner $0 \leq \frac{1}{2}a \leq \mathcal{Z}(a + b) \leq a + b \leq 2a$.
   Par application du lemme de Sterbenz, la soustraction $s = \mathcal{Z}(r - a)$ est exacte, et $s = r - a$. On en déduit l'équivalence $s = b \iff r = a + b$.
4. $-a \leq b \leq -\frac{1}{2}a$, $a \geq 0$.
   L'opération $a + b$ provoque ici une cancellation. Nous pouvons appliquer le lemme de Sterbenz pour obtenir $r = a + b$. Le calcul de r est toujours exact sous ces hypothèses. Comme b est un flottant, le calcul de s est également exact : $s = \mathcal{Z}(r - a) = \mathcal{Z}(b) = b$.

Nous n'avons traité jusqu'ici que les cas où $a \geq 0$. Par symétrie de l'arrondi vers zéro, la même étude de cas s'applique aux valeurs de a négatives, à une inversion générale de signes près. □

Ainsi, nous sommes capable de détecter si une addition est exacte ou non en utilisant une soustraction et une comparaison. Cela nous permet de déterminer s'il est nécessaire d'appliquer

une correction pour l'arrondi vers l'infini. Le calcul complet de l'addition d'intervalles est présenté sous la forme de l'algorithme 3.3.

Notons qu'une analyse plus fine dans la preuve nous permettrait de généraliser ce résultat même en cas de dépassements de capacité inférieurs dans une arithmétique avec dénormaux. Dans l'environnement que nous considérons, les dénormaux ne sont pas gérés et les calculs qui pourraient conduire à des dénormaux renvoient zéro. Cela rend en particulier le lemme de Sterbenz inapplicable en cas de dépassement de capacité.

Algorithme 3.3 Calcul de $[r_{inf}, r_{sup}] = [x_{inf}, x_{sup}] + [y_{inf}, y_{sup}]$ avec arrondi correct

$a \leftarrow \mathcal{Z}(x_{inf} + y_{inf})$
$b \leftarrow \mathcal{Z}(x_{sup} + y_{sup})$
$s_a \leftarrow \mathcal{Z}(a - x_{inf})$
$s_b \leftarrow \mathcal{Z}(b - x_{sup})$

$$r_{inf} \leftarrow \begin{cases} a & \text{si } a \geq 0 \vee s_a = y_{inf} \\ \sigma(a) & \text{sinon} \end{cases}$$

$$r_{sup} \leftarrow \begin{cases} b & \text{si } b \geq 0 \vee s_b = y_{sup} \\ \sigma(b) & \text{sinon} \end{cases}$$

Multiplication Détecter si une multiplication est exacte nécessite plus d'opérations que dans le cas de l'addition. En l'absence d'une opération FMA, le moyen usuel de calculer l'erreur d'arrondi de la multiplication est d'employer l'algorithme de Dekker, qui nécessite 17 opérations flottantes [Dek71].

Nous allons considérer deux méthodes alternatives : l'emploi de la double précision Binary64 et celle de la multiplication entière.

Le résultat exact de la multiplication de deux flottants Binary32 est représentable en Binary64. Il est donc possible d'utiliser les opérations double précision du GT200 pour calculer le résultat exact. Ce GPU n'offre pas d'instruction de conversion de Binary64 vers Binary32 avec arrondi dirigé. Cependant, nous pouvons toujours vérifier si le résultat est exactement représentable en comparant le résultat en double précision avec le résultat obtenu en simple puis converti en double. En l'absence de dépassement de capacité, il est aussi possible de tester les 29 bits de poids faible de la représentation binaire du résultat en double précision. Lorsqu'ils sont nuls, le résultat est représentable avec une précision de 24 bits. Le calcul complet coûte l'équivalent de 14 opérations sur le GT200.

Le jeu d'instructions Tesla fournit une instruction calculant une multiplication entière sur 24×24 bits. Elle opère sur les bits de poids faibles d'opérandes sur 32 bits ; les bits de poids fort sont ignorés. Nous pouvons utiliser cette instruction pour calculer les bits de poids faibles du produit des deux mantisses, et ainsi calculer l'équivalent du *sticky bit*. Ce calcul nécessite deux opérations OU pour rendre le bit de poids fort de la mantisse explicite, puis un ET pour sélectionner les 24 bits de poids faible. Si l'intégralité de ces 24 bits est nulle, alors le résultat flottant de la multiplication est exact. Si au moins un des 23 bits de poids faible est non-nul, alors le résultat est inexact.

La réponse pour le cas restant $(100\ldots0)$ dépend d'un éventuel décalage de la mantisse

Listing 3.1 – Test de l'exactitude d'une multiplication flottante par multiplication entière en CUDA

```
float rtz = __fmul_rz(x, y);
unsigned int xi = __float_as_int(x);
unsigned int yi = __float_as_int(y);
unsigned int x_mant = xi | 0x00800000;
unsigned int y_mant = yi | 0x00800000;
unsigned int lo = __umul24(x_mant, y_mant);
if(lo & 0x007fffff== 0) {
  // Cas general, inexact
  return next_float(rtz);
}
else
{
  unsigned int shift = ((xi ^ yi ^ __float_as_int(rtz)) & 0x00800000)
    | 0x007fffff;
  if((lo & shift) == 0) {
    return rtz;    // Operation exacte
  }
  else {
    return next_float(rtz);
  }
}
```

causé par la normalisation du résultat de la multiplication flottante. Cette normalisation peut être détectée en testant si $e_r = e_a + e_b$. Notons que le décalage maximal étant de 1, un test de parité sur les bits de poids faibles respectifs des exposants ($e_{a_0} \oplus e_{b_0} \oplus e_{r_0}$) est suffisant.

Comme le cas où seul le 24e bit est actif n'advient qu'avec une probabilité très faible, nous effectuons un saut conditionnel pour éviter le test de parité dans le cas le plus courant. Cela nécessite 7 opérations dans le cas courant, et 11 dans le pire cas.

3.5.4 Résultats

Nous comparons notre bibliothèque avec deux implémentations de l'arithmétique d'intervalles sur CPU, Boost.Interval [BMP06] et RealLib [Lam08].

La bibliothèque d'arithmétique d'intervalles de Boost exploite la technique d'inversion de signes. Pour permettre à l'utilisateur de mélanger les appels aux opérateurs d'arithmétique d'intervalles avec de l'arithmétique virgule flottante en arrondi au plus près, Boost restaure par défaut le mode d'arrondi courant après chaque opération (mode protégé). Il est possible de désactiver manuellement ce comportement sur les sections de code qui n'effectuent pas d'autre opérations que de l'arithmétique d'intervalles (mode non protégé).

L'implémentation d'arithmétique d'intervalles de Lambov intégrée dans RealLib se base sur le jeu d'instructions Intel SSE2. Le mode d'arrondi associé aux instructions SSE est ajusté au début du programme et reste placé en arrondi vers l'infini pendant l'ensemble de l'exécution. Les autres calculs ne peuvent pas utiliser les unités SSE, ce qui implique de compiler spécifiquement l'ensemble du programme – et les bibliothèques auxquelles il fait appel – pour

Format	Core i5 M540			G92		GT200		GF100
	Boost P	Boost U	RealLib	Apx	Cor.	Apx	Cor.	
Binary32	0,033	0,182	✗	9,12	3,31	(14,8)	(5,39)	23,3
Binary64	0,021	0,144	0,155	✗			2,02	4,70

TABLE 3.8 – Performances d'une multiplication et addition d'intervalles, en milliards d'itérations par seconde. Les bibliothèques considérées sont Boost en mode protégé (P) et non protégé (U), RealLib et notre bibliothèque avec arrondis approchés (Apx) ou corrects (Cor.). Les nombres entre parenthèses sont des estimations.

le jeu d'instructions x87.

La table 3.8 présente une comparaison des performances de ces bibliothèques avec notre implémentation sur GPU par un micro-test. Nous effectuons de manière répétitive une opération $a \times b + c$ en arithmétique d'intervalles sur des données locales. Les cartes graphiques utilisées pour ce test recouvrent trois générations, avec une GeForce 9800 GX2 (G92), un Tesla C1060 (GT200) et une GeForce GTX 480 (GF100). Le CPU est un Intel Core i5 M540. RealLib opère sur le format Binary64 uniquement. À l'inverse, le G92 ne gère pas ce format. La performance indiquée pour le Tesla C1060 en Binary32 est une estimation calculée à partir des résultats d'un autre GPU GT200.

Les résultats obtenus confirment l'effondrement des performances que provoque l'alternance des modes d'arrondi sur CPU. Le calcul des arrondis approchés par la technique d'incrémentation présentée section 3.5.2 est peu coûteux et garantit la propriété d'inclusion. L'obtention de l'arrondi correct a un coût supplémentaire significatif sur les architectures sur lesquelles il est nécessaire de l'émuler (G92 et GT200 en Binary32). Le GT200 et le GF100 gèrent les modes d'arrondis dirigés statiques, respectivement en Binary64 et dans les deux formats. Ils ne souffrent ni de l'impact de l'émulation des arrondis comme sur les GPU antérieurs, ni de celui des basculements de modes comme sur les CPU.

Conclusion

Nous avons pu constater l'ampleur de l'évolution que les unités virgule flottantes généralistes des GPU ont subi entre 2006 et 2010. Notre analyse des GPU de la génération Direct3D 9 fait état de nombreux points de non-conformité par rapport à la norme IEEE 754. À l'inverse, les unités arithmétiques de la génération Direct3D 11 offrent désormais davantage de fonctionnalités que celles des processeurs généralistes courants.

Au-delà des unités généralistes, nous avons exploité les unités spécialisées dans le rendu graphique pour une application de simulation de transferts radiatifs. De la même manière, nous avons su détourner les unités de filtrage de textures pour évaluer des polynômes par morceaux. Ainsi, nous avons montré que la spécialisation des unités n'empêchait pas leur utilisation pour d'autre tâches.

Nous avons également tiré parti des modes d'arrondis statiques des GPU NVIDIA pour implémenter efficacement l'arithmétique d'intervalles. Nous avons proposé des techniques lo-

gicielles permettant d'émuler les modes d'arrondis dirigés sur les GPU qui n'en offrent pas de support matériel.

Nous avons conjecturé dans la section 1.3.2 que les unités spécialisées seraient amenées à se généraliser. Les résultats obtenus dans ce chapitre tendent à étayer cette hypothèse.

La problématique qui se dégage est de parvenir à atteindre le bon compromis entre flexibilité et efficacité. Nous avons pu détourner des unités spécialisées de leur destination première, au prix d'une efficacité réduite en terme de performance, énergie ou précision. Des modifications matérielles plus ou moins importantes des unités permettraient de réduire ce surcoût. Inversement, en ciblant un champ d'applications trop large, on risque de perdre le bénéfice d'efficacité qu'apportent les unités spécialisées. Il faudra donc parvenir à concevoir des unités de calcul efficaces tout en évitant de contraindre les applications possibles par des limitations arbitraires.

Cet effort pourra avoir un impact sur les architectures généralistes. Par exemple, les architectures de type DSP ont inspiré la conception de l'opérateur FMA en 1990 [MHR90], qui se généralise aujourd'hui aux processeurs x86 [Int09a]. Il reste à déterminer quels seront les futurs opérateurs ou fonctionnalités qui seront issus du développement des GPU actuels.

Les tests et applications présentés ici ont fait l'objet de publications [CDD07, CDD08a, CDD08b, CFD08, ACD10].

Chapitre 4

Barra, simulateur d'architecture CUDA

Nous avons étudié au chapitre 2 une architecture GPU existante, et présenté dans le chapitre 3 des manières d'exploiter les opérateurs arithmétiques de plusieurs générations de GPU. Ces études ont fait apparaître la nécessité d'adapter l'architecture matérielle aux applications. Nous voulons proposer des modifications architecturales aux GPU, ce que nous ferons au cours du chapitre 5. Pour être à même d'explorer et valider ces modifications, nous devons auparavant modéliser précisément le fonctionnement d'une architecture GPU.

Nous proposons dans ce chapitre un simulateur de l'architecture Tesla qui exécute des programmes binaires CUDA non modifiés. Nous montrerons qu'il est possible de simuler les architectures parallèles à grain fin de manière efficace en tirant parti du parallélisme, du faible partage et de la régularité induits par le modèle de programmation des GPU.

Après un aperçu de l'état de l'art section 4.1, nous présenterons les couches d'émulation du pilote CUDA section 4.2. Nous aborderons ensuite plus en détails le cœur du simulateur dans la section 4.3, puis analyserons des méthodes d'accélération et de parallélisation de la simulation dans la section 4.4. Enfin, nous utiliserons Barra pour caractériser des applications CUDA section 4.5, puis validerons la précision de la simulation au travers de ces programmes dans la section 4.6.

4.1 Simulateurs d'architectures parallèles

Simulateurs généralistes L'apparition de simulateurs de processeurs pour les architectures superscalaires dans les années 1990 a été le point de départ de nombreux travaux de recherche académiques et industriels en architecture des processeurs. Les simulateurs peuvent opérer à différentes granularités, suivant la précision souhaitée. Les simulateurs cycle à cycle simulent des modèles avec une précision au niveau cycle comparable au matériel simulé. De tels simulateurs nécessitent des volumes de communications importants entre les modules du simulateur lors de chaque cycle simulé. Les simulateurs au niveau transaction sont basés sur des modules fonctionnels et se contentent de simuler les communications entre ces modules. Ceux-ci sont généralement moins précis que les simulateurs au niveau cycle mais sont nettement plus ra-

pides. Enfin, les simulateurs au niveau fonctionnel reproduisent uniquement le comportement d'un processeur sur un code donné. De tels simulateurs peuvent intégrer des modèles de temps ou de consommation afin d'obtenir des estimations rapides de la performance d'un code ou d'une architecture.

Le simulateur niveau cycle SimpleScalar [ALE02] a été à l'origine de nombreux travaux accompagnant le succès des processeurs superscalaires à la fin des années 1990. Cependant, l'organisation monolithique de ce simulateur le rend difficile à adapter et modifier. Des alternatives à SimpleScalar ont été proposées pour la simulation de multi-cœurs [MSB+05] ou de systèmes complets [RBDH97, MCE+02, BDH+06].

Simulateurs de GPU En ce qui concerne les GPU, il existe des environnements de simulation modélisant le pipeline graphique, tels que le simulateur niveau cycle Attila [MGR+05] ou le simulateur niveau transaction Qsilver [SLS04]. Les questions architecturales étudiées par ces simulateurs sont sensiblement différentes de celles que posent les architectures parallèles pour le calcul généraliste telles que les GPU actuels.

L'arrivé de CUDA a stimulé le développement de simulateurs de GPU mettant l'accent sur le calcul généraliste. Nous avons abordé dans la section 1.1.4 l'émulateur associé au projet Ocelot, qui permet d'exécuter le code CUDA sur CPU au niveau PTX en offrant des possibilités d'instrumentation [DKYC10]. Le modèle d'exécution est de type SIMD : l'ensemble des threads d'un CTA partagent le même compteur de programme. De manière équivalente, la taille des warps coïncide avec celle des CTA.

Un autre émulateur est présenté dans [Exo09]. Il se base sur une représentation binaire du PTX. Les CTA sont distribués sur plusieurs threads hôte. Le modèle d'exécution est le MIMD : chaque thread maintient son propre compteur de programme. L'utilisation d'un cache d'instructions pré-décodées permet de limiter le surcoût lié au décodage des instructions. Ocelot et ce projet visent à fournir un modèle d'exécution indépendant de l'architecture, à l'inverse d'un simulateur.

GPGPU-Sim [BYF+09] est un simulateur d'architecture SIMT niveau cycle basé sur SimpleScalar et des modules additionnels. Il simule une architecture originale de type GPU utilisant le langage PTX comme jeu d'instructions. Les threads sont groupés par warps et exécutés en mode SIMT. Il est cependant difficile de déterminer à quel point l'architecture considérée est représentative des GPU actuels ou futurs. Le choix d'utiliser un langage de haut niveau en lieu et en place du jeu d'instructions peut également avoir un impact négatif sur la précision de la simulation.

Jusqu'à sa version 3.0, l'environnement CUDA intègre un mode émulation qui compile un programme CUDA pour une exécution complète sur CPU. Ce mode donne accès aux fonctions de bibliothèques et appels systèmes tels que *printf* au sein de noyaux de calcul, ce qui facilite le débogage de programmes. En revanche, le comportement et les résultats entre le mode émulation et l'exécution sur GPU peuvent différer significativement, notamment en termes de comportement de l'arithmétique flottante et entière, de granularité et d'ordre d'exécution des threads et des synchronisations.

Cet outil a été abandonné par NVIDIA au profit des débogueurs cuda-gdb [NVI09a] et Parallel Nsight [NVI], qui permettent d'exécuter pas-à-pas le code CUDA directement sur GPU. Ceux-ci reproduisent fidèlement le comportement des applications. NVIDIA fournit également

l'outil de profilage de code CUDAProf, qui offre l'accès à des compteurs de performance sur le GPU.

Objectifs Les caractéristiques souhaitables d'un simulateur sont les suivantes :
- être instrumentable : pouvoir extraire aisément des statistiques d'exécution,
- être facile à modifier en vue de tester des changements dans l'architecture,
- être paramétrable,
- être basé sur un modèle d'architecture réaliste et représentatif de l'état de l'art,
- reproduire de manière précise le comportement de l'architecture.

Or, aucun des travaux de la littérature n'offre l'ensemble de ces caractéristiques. Nous serons donc amené à développer un simulateur de GPU appelé Barra. Nous utiliserons comme base de travail l'architecture Tesla étudiée au chapitre 2.

4.2 Environnement de simulation

Nous concevons ici un simulateur de *coprocesseur*, ce qui réclame une architecture quelque peu différente de celle d'un simulateur de processeur. L'approche classique suivie par les simulateurs de CPU est de lier la durée de vie du processus de simulation avec l'exécution du programme à simuler. Le simulateur est alors un exécutable autonome qui charge puis exécute un unique programme ou système d'exploitation jusqu'à la fin de l'exécution.

Ici, le programme simulé consiste en deux parties. La première partie est constituée du code compilé pour le processeur hôte (x86). Elle fait appel à la bibliothèque CUDA pour exécuter des noyaux de calcul compilés pour l'architecture Tesla, et orchestre les mouvements de données et la gestion des ressources. L'autre partie est composée des noyaux de calculs exécutés sur GPU.

L'organisation du simulateur reflète ce découpage. Il comprend un pilote qui s'interface avec la partie hôte du code simulé, ainsi qu'un ensemble de modules chargés de la simulation des noyaux de calcul CUDA. Les deux sections suivantes couvrent plus en détail chacune de ces parties.

4.2.1 Pilote

La majorité des applications GPGPU actuelles n'effectue pas d'interactions CPU-GPU complexes, et ce fait se reflète dans les programmes de tests. Aussi, nous considérons le code GPU en isolation. La partie hôte du programme n'est pas simulée, mais est exécuté sur le processeur hôte. Lorsqu'elle fait appel à la bibliothèque CUDA Driver, l'appel est intercepté par Barra.

La pile logicielle CUDA offre trois couches distinctes : le Runtime distribué avec CUDA, la partie en mode utilisateur du pilote (CUDA Driver), et le pilote lui-même. La plupart des fonctions de l'API Runtime et de l'API Driver sont documentées.

Le simulateur Barra prend la place du GPU sans nécessiter de modification des programmes CUDA (figure 4.1). Un pilote constitué d'une bibliothèque partagée exportant les mêmes symboles que la bibliothèque propriétaire *libcuda.so* capture les appels destinés au GPU et les

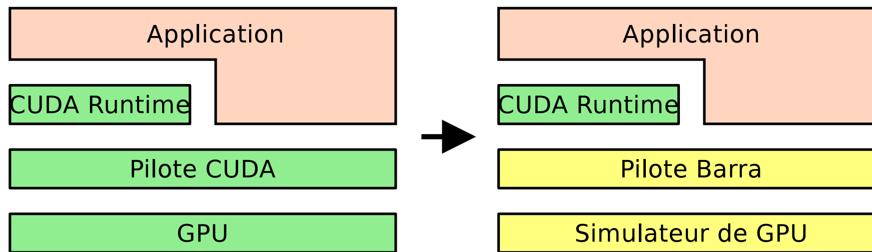


FIGURE 4.1 – Positionnement de Barra parmi les couches logicielles de CUDA.

redirige vers le simulateur. Ainsi l'utilisateur peut choisir entre une exécution sur GPU ou une simulation avec Barra en positionnant la variable d'environnement définissant l'emplacement des bibliothèques partagées (`LD_LIBRARY_PATH`) sur le répertoire de son choix.

L'ensemble des projets liés à CUDA décrits dans la section précédente se basent sur l'API CUDA Runtime. Par contraste, nous avons choisi de nous placer au niveau de la couche API Driver. D'une part, certaines applications font appel directement à la couche Driver et ne fonctionneront pas dans les autres environnements. D'autre part, ce choix permet de profiter de la couche CUDA Runtime existante et garantit une compatibilité accrue avec les applications.

4.2.2 Modules

Le simulateur Barra s'intègre dans l'environnement de simulation Unisim [ACG+07]. Un des points clefs de cet environnement est de présenter une organisation modulaire. Les simulateurs sont décrits par des composants interconnectés par des interfaces précisément spécifiées. Les composants peuvent opérer au niveau fonctionnel, au niveau transaction ou au niveau cycle.

De fait, Barra est composé de plusieurs modules séparés. Leur organisation est illustrée figure 4.2. Le pilote se charge de recevoir les appels de fonctions de l'API Driver CUDA. Les appels synchrones tels que les allocations en mémoire GPU sont traités immédiatement. Les appels asynchrones, tels que les copies mémoire et les lancements de noyaux, sont placés dans une file de commandes. Celle-ci fait l'interface entre le pilote, c'est-à-dire la partie logicielle, et un processeur de commandes faisant partie du matériel simulé.

Un gestionnaire de mémoire se charge d'allouer et libérer des zones dans la mémoire GPU simulée. Il s'interface uniquement avec le pilote et le processeur de commandes : en effet, le modèle de programmation ne permet pas d'allocation dynamique depuis le code des noyaux de calcul.

Les commandes de lancement de noyau sont relayées à un ordonnanceur, qui se charge de créer les CTA à la demande et de les répartir sur les SM simulés.

4.3 Simulation fonctionnelle

Nous avons vu que le modèle d'exécution du GPU était le SIMT, par opposition au modèle séquentiel généralement observé par les outils de simulation fonctionnelle.

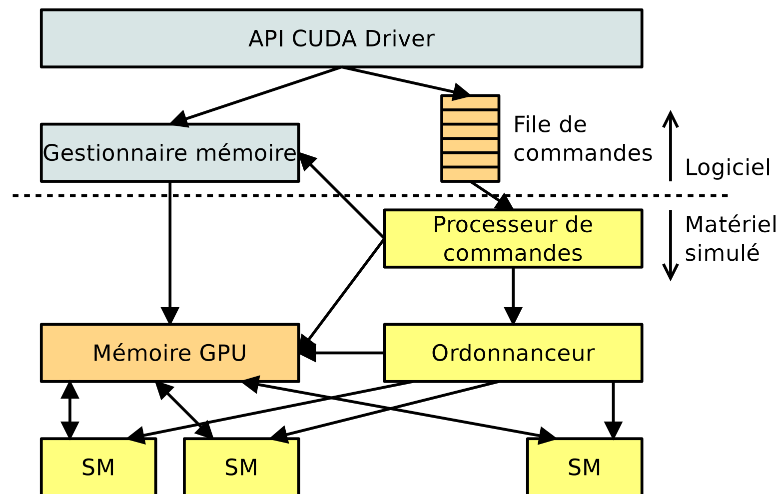


FIGURE 4.2 – Modules du simulateur.

Cette différence de modèle a des implications sur la façon d'envisager la simulation. En effet, si nous choisissons de simuler chaque thread CUDA par un contexte indépendant à la manière du mode émulation de CUDA, le surcoût des basculements de contexte ainsi que la perte de localité ralentiront la simulation.

Nous choisissons donc un mode d'exécution similaire au mode SIMT utilisé en matériel, en exécutant les instructions à la granularité du warp. D'une part, cela permet une simulation plus fidèle du matériel en reproduisant des comportements liés à l'ordonnancement des warps et au contrôle de flot SIMT. D'autre part, ce mode permet d'accélérer la simulation en tirant parti de la régularité et de la localité qu'apporte le modèle d'exécution SIMT. Nous verrons en effet que les mécanismes qui permettent une implémentation matérielle efficace profitent également à une simulation par logiciel.

Exécution des instructions Même si le modèle mémoire CUDA est composé de plusieurs mémoires séparées logiquement (constantes, locale, globale, partagée), et que l'architecture matérielle Tesla contient des espaces mémoires séparés physiquement (DRAM et mémoire partagée), nous unifions les différents types de mémoires en les répartissant dans un unique espace d'adressage physique. Cela permet de simplifier et d'unifier les mécanismes d'accès. Ce choix se justifie d'autant plus avec l'unification de la mémoire locale, la mémoire globale et la mémoire partagée apparue avec la génération suivante, Fermi.

Les instructions sont exécutées dans Barra en suivant un modèle décrit dans la figure 4.3.

1. L'ordonnanceur sélectionne le prochain warp pour exécution et récupère le compteur de programme (PC) correspondant.
2. L'instruction présente en mémoire à cette adresse est extraite et décodée.
3. Les opérandes sont lus depuis le banc de registres, la mémoire partagée ou la mémoire de constantes.

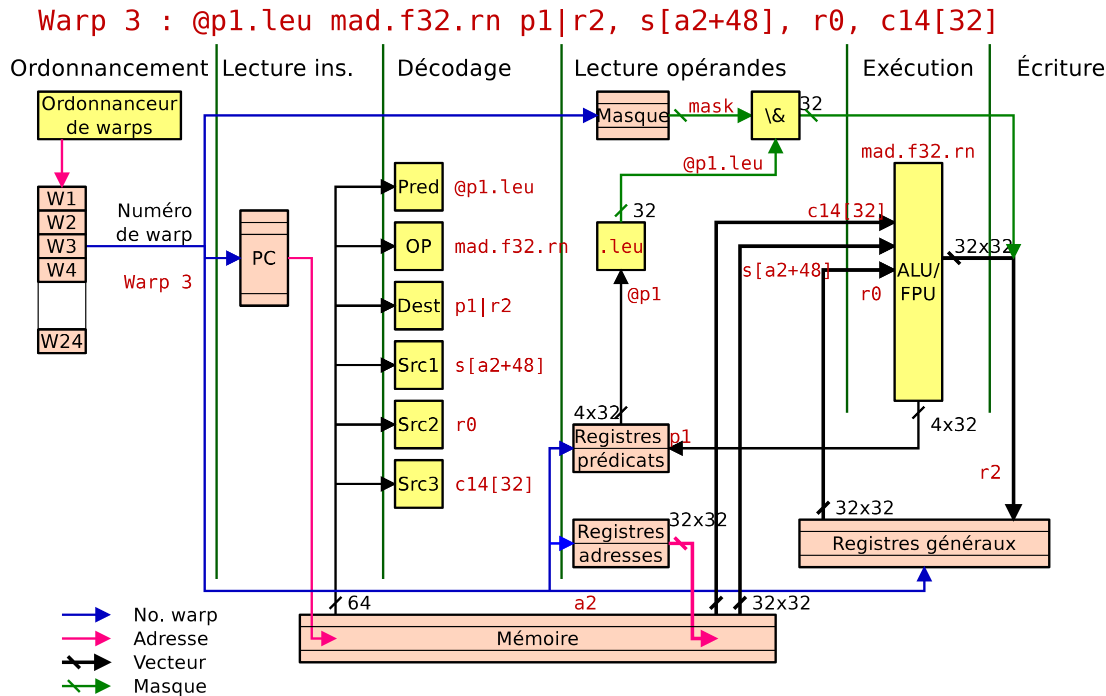


FIGURE 4.3 – Vue du pipeline fonctionnel. Exemple donné pour l'exécution d'une instruction MAD.

4. L'instruction est exécutée et le résultat est écrit dans le banc de registres.
5. Les instructions arithmétiques ont la possibilité de mettre à jour un registre de drapeaux contenant les informations de retenue, débordement, zéro ou valeur négative.

L'arithmétique virgule flottante est plus complexe à reproduire. Barra propose deux modes d'émulation de l'arithmétique flottante. Le premier mode repose sur l'utilisation directe des unités du CPU pour traiter les flottants. Ce mode présente l'avantage d'être rapide et de bénéficier de la gestion matérielle des modes d'arrondis du CPU. Le second environnement repose sur la bibliothèque d'émulation de l'arithmétique flottante Simfloat++ intégrée dans UNISIM.

Ces deux modes permettent de reproduire le comportement d'unités matérielles non conventionnelles (MAD tronqués, dénormaux traités comme des zéros) et délivrent des résultats corrects au bit près par rapport au GPU pour les opérateurs arithmétiques usuels. De plus, disposer de deux implémentations indépendantes permet de croiser les vérifications.

L'unité d'évaluation de fonctions SFU intègre des opérateurs dédiés ainsi que des données tabulées. Une simulation exacte de ce type d'unité nécessiterait soit une tabulation de tous les résultats possibles obtenus par des tests exhaustifs conduisant à une consommation de stockage et de mémoire de l'ordre du gigaoctet, soit une simulation au bit près des chemins de données et des tables de l'unité matérielle nécessitant un effort de rétro-ingénierie substantiel. Par conséquent, la simulation des unités SFU dans Barra repose sur une réduction d'argument identique à la version GPU suivie d'un appel à la fonction correspondante de la bibliothèque

mathématique du langage C. Cette implémentation implique un comportement différent pour les fonctions concernées.

Gestion des registres Les registres généraux sont partagés entre les threads s'exécutant sur un multiprocesseur donné, ce qui permet d'ajuster le compromis entre localité et parallélisme. Nous maintenons un état pour chaque warp actif du multiprocesseur. Cet état comprend le compteur de programme, les registres d'adresse et de prédication, les piles de masque et d'adresse, la fenêtre des registres assignés au warp, ainsi que la fenêtre sur la mémoire partagée du CTA.

Le multiprocesseur des GPU de type Tesla dispose d'un banc de registres multi-bancs, tel que décrit section 2.4.3. La politique de partitionnement du banc de registres n'influe pas sur le comportement du simulateur fonctionnel. De fait, l'algorithme utilisé pour la simulation repose sur une allocation séquentielle de blocs de registres dans un banc de registres unifié.

Pour tirer parti de la localité apportée par le modèle SIMT, nous allouons les registres des threads d'un même warp de manière adjacente. Ainsi, les instructions SIMD appliquées aux warps manipulent des données vectorielles contiguës.

Ordonnancement des warps La sélection du warp dont une instruction sera exécutée est faite par un ordonnancement en tourniquet.

Chaque warp dispose d'un drapeau qui définit si celui-ci est prêt à être exécuté. Au début de l'exécution, chaque warp a son drapeau positionné sur *actif* et celui des autres warps est positionné sur *inactif*. À chaque étape de la simulation, le prochain warp marqué comme actif est sélectionné et son instruction courante est exécutée.

Lorsqu'une instruction de synchronisation est rencontrée, le warp courant est positionné dans l'état *attente*. Si tous les warps sont dans l'état *attente* ou *inactif*, la barrière de synchronisation est atteinte par tous les warps et les warps en attente sont replacés dans l'état *actif*.

Un marqueur spécifique intégré dans le mot d'instruction indique la fin de kernel. Lorsqu'il est rencontré, le warp courant est positionné dans l'état *inactif* de façon à ce que celui-ci soit ignoré lors des ordonnancements futurs.

Flot de contrôle Nous nous basons sur le système de prédication de l'architecture Tesla à base de pile et jetons décrit section 2.4.2.

4.4 Parallélisation de la simulation

Les noyaux de calcul CUDA présentent par conception un fort potentiel de parallélisme de données et de régularité. Ce potentiel, qui est exploité par l'architecture du GPU, peut également profiter à une implémentation logicielle.

Traitement des instructions La section 2.1 nous a montré que le jeu d'instructions de Tesla était régulier et orthogonal. Nous tirons parti de cette orthogonalité en décodant de manière

indépendante plusieurs sous-ensembles du mot d'instruction. Cela limite l'explosion combinatoire due à la spécialisation des fonctions d'exécution des instructions. Chaque sous-partie est mémorisée dans un cache d'instructions pré-décodées séparé.

Simuler un many-core avec un multi-core Le modèle de programmation de CUDA est conçu pour réduire le couplage entre les CTA. L'ordre d'exécution des CTA n'est pas spécifié, les communications entre CTA sont restreintes et le modèle de consistance mémoire est relâché. Cela permet une implémentation matérielle efficace et capable de passer à l'échelle.

Nous simulons chaque SM dans un thread hôte différent. Cela n'affecte pas le comportement de la simulation fonctionnelle en dehors de certaines utilisations des instructions atomiques. Les CTA sont répartis de manière statique entre les SM à la manière de l'ordonnancement effectué sur l'architecture Tesla.

Les simulateurs de processeurs généralistes doivent être capable de traiter correctement les allocations dynamiques de mémoire et le code auto-modifiant dans les programmes simulés. Cela les oblige à maintenir des structures dynamiques de type cache qui peuvent grossir en fonction de la demande. Le partage de telles structures dans un environnement multi-thread requiert des mécanismes de verrouillage, qui peuvent s'avérer complexes à mettre en œuvre et à valider et avoir un impact négatif sur le passage à l'échelle.

Heureusement, CUDA impose que les allocations de mémoire soient statiques et explicites : le code hôte et le pilote se chargent d'allouer toutes les zones de mémoire avant le lancement du noyau. Aucune allocation dynamique supplémentaire n'est possible au cours du calcul. Le code auto-modifiant n'est également pas permis par le modèle.

Nous pouvons donc pré-allouer toutes les zones de données et d'instructions avant le démarrage de la simulation du noyau. La consistance est assurée naturellement pendant le déroulement de la simulation.

Simuler le SIMT par du SIMD L'exécution en mode SIMT permet d'amortir le coût de la lecture et du décodage d'une instruction sur plusieurs calculs. Ce bénéfice est tout aussi significatif dans le cadre d'une simulation au niveau instruction. En effet, le décodage des instructions est le principal goulet d'étranglement dans les simulateurs fonctionnels. Le coût de traitement de l'instruction est ici amorti sur plusieurs calculs. Cela augmente l'intensité arithmétique du code et introduit du parallélisme d'instructions.

Nous exécutons les instructions arithmétiques en virgule-flottante de base (add, mul, mad, min, max, inverse, racine carrée inverse) à l'aide d'instructions SIMD SSE lorsqu'elles sont disponibles [Int10a]. La vectorisation est facilitée par l'organisation vectorielle du banc de registres simulé. Les modes Denormal-Are-Zero (DAZ) et Flush-To-Zero (FTZ) sont activés pour refléter le comportement arithmétique de ces opérations. Nous nous assurons que le comportement des instructions Tesla est respecté. Les règles de propagation des NaN sont les mêmes que celles du processeur de l'hôte.

Ainsi, le modèle de programmation suivi par CUDA bénéficie autant aux implémentations logicielles que matérielles.

4.5 Validation et applications

L'industrie des microprocesseurs haute performance se base sur des applications de test, ou *benchmarks* pour quantifier la performance des processeurs et compilateurs [HPAD07]. Ces applications sont relativement stables, les plus répandues étant les SPEC [SPEa]. D'autres suites de tests existent dans le cadre des systèmes embarqués, ou les applications multi-média.

Les benchmarks dans le domaine des architectures parallèles à grain fin sont moins développés. Les premières applications CUDA largement répandues sont les exemples fournis avec le kit de développement (SDK) CUDA. Bien qu'elles n'aient pas été conçues pour être utilisées comme benchmarks à l'origine, leur ubiquité et leur simplicité de déploiement en a fait une norme *de facto*.

Nous considérons aussi l'implémentation de la routine SGEMM de Volkov, un calcul de produit de matrices denses [VD08]. Nous l'exécutons sur des matrices de taille 256×256. Le noyau 3DFD est un calcul de différences finies en 3D [Mic09]. Nous considérons une taille de CTA de 64×8, sur un volume de $256 \times 256 \times 100$. Ces deux programmes représentent des noyaux de calcul ayant bénéficié d'un effort d'optimisation particulier pour l'architecture Tesla.

Nous serons amenés à considérer également les suites de tests Rodinia [CBM$^+$09] et UIUC Parboil [UIU10].

L'ensemble de ces benchmarks ont fait l'objet d'analyses sur leur comportement, de manière indépendante de l'architecture [KDY09].

Nombre d'instructions Considérons quelques statistiques sur les exemples du SDK CUDA dans la table 4.1. Le nombre d'instructions exécutées (dynamiques) est obtenu par une simulation sous Barra. Le nombre d'instructions statiques provient du code exécutable. Notons qu'il n'existe pas de corrélation sensible entre la taille du code PTX et celle du code assembleur.

4.6 Caractérisation

Nous quantifierons maintenant la vitesse d'exécution et la précision de la simulation au travers des exemples du SDK CUDA.

4.6.1 Vitesse de simulation

Nous comparons les temps d'exécution des applications de test

- exécutés nativement sur GPU,
- dans le mode émulation intégré dans CUDA,
- dans une session de déboguage sous CUDA-gdb,
- dans l'émulateur d'Ocelot 0.4.46,
- lors d'une simulation mono-thread dans Barra,
- lors d'une simulation multi-thread.

Les temps d'exécution normalisés par le temps d'exécution natif sur GPU ainsi que leur moyenne géométrique sont reportés sur la figure 4.4.

Notre plate-forme de test est constituée d'un processeur Intel Core 2 Duo E8400 sur chipset Intel X48 et d'une carte graphique NVIDIA GeForce 9800 GX2. La configuration logicielle est Ubuntu 8.10 x64 avec gcc 4.3 et CUDA 2.2. Les tests *MonteCarlo* et *binomialOptions* n'ayant pas terminé leur exécution dans le débogueur sous 24 heures, ils ne sont pas inclus dans les résultats, ni pris en compte dans les moyennes.

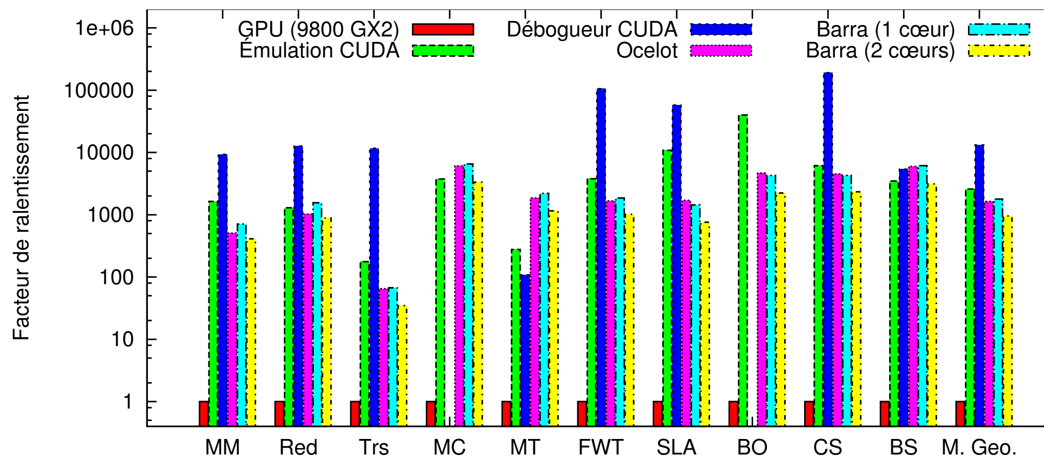


FIGURE 4.4 – Facteurs de ralentissements du mode d'émulation de CUDA, d'exécution dans le débogueur CUDA et simulation fonctionnelle dans Barra, par comparaison à une exécution native sur GPU.

Observons que la simulation mono-thread sous Barra a des performances compétitives avec le mode d'émulation de CUDA, tout en étant significativement plus précis. La raison probable est que le temps d'émulation de CUDA est dominé par le surcoût de la création et synchronisation des milliers de threads au niveau utilisateur.

Le débogueur CUDA souffre d'un surcoût encore supérieur, probablement du fait de transferts mémoire et de synchronisations après l'exécution de chaque instruction. Notons que ce surcoût devrait être nettement moins présent sur l'architecture Fermi, grâce à un support matériel renforcé des fonctions de déboguage [NVI09b].

Nous testons le passage à l'échelle fort (*strong scalability*) en simulant les mêmes jeux de tests sur une station de travail basée sur un quadri-cœur Intel Xeon E5410, avec un nombre de threads variant entre 1 et 4. La moyenne géométrique du facteur d'accélération est de 1,9 en passant de 1 à 2 cœurs et de 3,5 de 1 à 4 cœurs. Cette scalabilité est rendue possible par le modèle de programmation de CUDA qui réduit les dépendances et les synchronisations entre les cœurs. À l'inverse, le mode émulation de CUDA se base sur des threads niveau utilisateur, et ne retire pas d'avantage des architectures multi-cœurs.

La simulation avec Barra a une vitesse comparable à celle de l'émulateur d'Ocelot. Elle est généralement plus rapide que le mode émulation de CUDA, tout en offrant une précision accrue et des possibilités de modifications et d'instrumentation. En prenant en compte le modèle d'exécution SIMT dès la conception du simulateur, les coûts de décodage et de contrôle sont amortis entre plusieurs calculs. L'utilisation complémentaire de techniques connues telles qu'un cache d'instructions pré-décodées réduit davantage le surcoût de simulation. Ces tech-

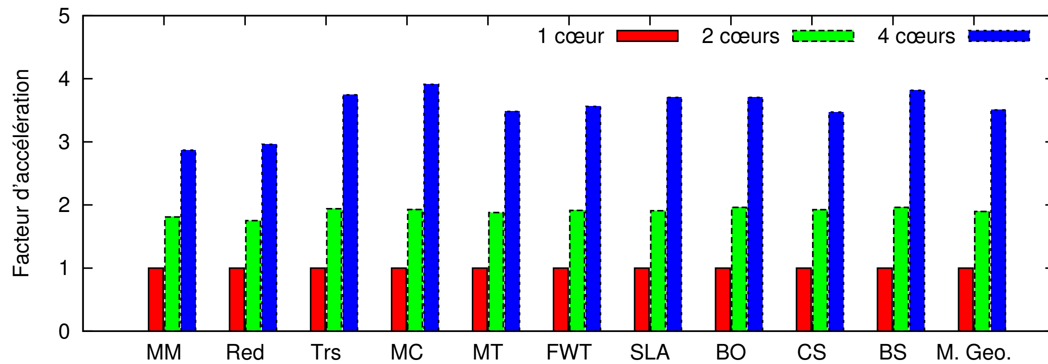


FIGURE 4.5 – Facteur d'accélération suivant le nombre de threads employés pour la simulation.

niques permettent de bénéficier de la régularité, de la localité et du parallélisme des applications parallèles.

Cet avantage est particulièrement significatif pour la simulation de noyaux de calcul sur GPU, où le ratio d'instructions dynamiques par instruction statique (ou nombre moyen d'exécutions de chaque instruction) est élevé, comme le montrent les statistiques de la table 4.1.

4.6.2 Précision

Nous avons collecté des statistiques sur le type des instructions, leurs opérandes, la divergence des branchements et les accès mémoire pour chaque instruction statique. Nous n'avons pas observé de variation entre les statistiques obtenues entre la simulation mono-thread et la simulation parallèle.

Comparons ces statistiques avec les compteurs de performance matériels accessible par l'outil de profilage CUDAProf. Ces compteurs fonctionnent à la granularité du noyau de calcul : il n'est pas possible d'isoler la contribution de chaque instruction sans modification du code original. Certains compteurs sont accessibles sur un SM uniquement, et d'autre sur un TPC uniquement. Cette information partielle est ensuite extrapolée au noyau entier par CUDAprof. La signification, l'unité et le facteur d'échelle de chaque compteur ne sont pas documentés. Cependant, nous avons pu interpréter la plupart de ces compteurs et le mettre en rapport avec les valeurs obtenues par simulation. Étudions les différences relatives observées pour le nombre d'instructions, de sauts, de divergences de sauts et de transactions mémoire sur la figure 4.6.

Les nombres d'instructions observés sont cohérents, à l'exception de l'application *scanLargeArray*. Une analyse des compteurs révèle que le noyau `prescan<true,false>` est lancé plusieurs fois sur un seul CTA. L'outil de profilage semble sélectionner un TPC différent à instrumenter pour limiter l'effet de déséquilibre que cela engendre. Cependant, le nombre d'appels (202) n'est pas multiple du nombre de TPC (8) et un certain déséquilibre demeure.

Nous n'avons pas pu déterminer la signification exacte du compteur de branchements. Nous remarquons qu'il est systématiquement supérieur ou égal au nombre de toutes les instructions de contrôle rencontrées par le simulateur.

L'application *transpose*, et dans une moindre mesure *matrixMul*, présentent des différences

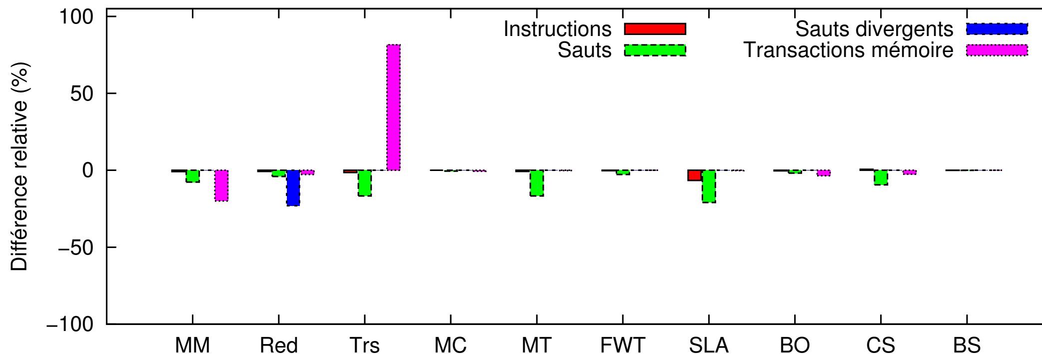


FIGURE 4.6 – Différences relatives observées entre les statistiques produites par Barra et les compteurs de performance matériels.

au niveau du nombre d'instructions d'accès mémoire reportées. Le test *transpose* est connu pour être affecté par le phénomène de *partition camping* décrit section 2.5.3. Nous avons considéré l'exemple *transposeNew* du SDK, qui réalise exactement la même action tout en évitant les conflits de partitions. Les résultats que nous obtenons alors sont cohérents entre la simulation et les compteurs de performance. Cela tend à confirmer que les différences observées sur *transpose* et *matrixMul* sont dues au phénomène de *partition camping*.

Ainsi, les quelques différences rencontrées entre les résultats de l'outil de profilage de CUDA et les statistiques issues de la simulation dans Barra peuvent toutes s'expliquer par une imprécision de la part de CUDAProf et non de Barra.

Conclusion

Nous avons présenté un simulateur fonctionnel d'architecture parallèle à grain fin. Conçu d'après l'architecture réelle NVIDIA Tesla, il offre un modèle représentatif d'une architecture GPU actuelle. L'efficacité de la simulation est maintenue grâce à l'exploitation de la régularité et de la localité offertes par le modèle de programmation SIMT. Les opérations sur les nombres entiers et à virgule flottante sont simulées avec une précision au bit près. La simulation produit des traces et des statistiques d'exécution pour chaque instruction. Ces données sont au moins aussi précises tout en étant plus détaillées que les résultats obtenus par profilage sur un GPU réel.

La simulation fonctionnelle pourrait être accélérée davantage par l'utilisation de traduction binaire. En effet, le modèle de programmation n'autorise pas le code auto-modifiant : il n'est donc pas nécessaire de mettre en place des mécanismes de compilation JIT complexes pour traiter ces cas.

La séparation des composants en modules communicants permet d'envisager la construction d'un modèle niveau transactions [SD08] sur la base du simulateur fonctionnel. La régularité et le parallélisme des applications GPU pourraient également être exploités par des simulateurs au niveau transaction (par découplage temporel [SD08] ou simulation parallèle à évè-

nements discrets (PDES) [Fuj90]) ou au niveau cycle (par vectorisation de modules [PBG09]).

Dans sa forme actuelle, Barra nous fournit une base pour caractériser les programmes parallèles et analyser l'adéquation entre les applications et l'architecture, ainsi que pour proposer et valider des modifications architecturales.

Barra a été présenté dans [CDP09, CDDP10]. e

TABLE 4.1 – Noyaux des applications de test du SDK CUDA, avec leur nombre d'instructions PTX statiques, nombre d'instructions assembleur statiques et dynamiques (respectivement St. ASM et Dyn. ASM).

Programme	Noyau	St. PTX	St. ASM	Dyn. ASM
3dfd (3DFD)	fwd_3D_16x16_orderX	199	162	15712256
binomialOptions (BO)	binomialOptionsKernel	153	114	401131008
BlackScholes (BS)	BlackScholesGPU	134	99	5201694720
convolutionSeparable (CS)	convolutionRowGPU	67	52	38486016
	convolutionColGPU	99	100	38338560
dwtHaar1D (DWT)	dwtHaar1D	92	87	10204
fastWalshTransform (FWT)	fwtBatch1Kernel	110	107	57606144
	fwtBatch2Kernel	47	46	54263808
	modulateKernel	26	24	2635776
histogram256 (HG)	histogram256Kernel	147	112	1957633952
	mergeHistogram256...	60	36	1343488
matrixMul (MM)	matrixMul	83	114	66880
MersenneTwister (MT)	RandomGPU	159	223	31526528
	BoxMuller	86	68	16879360
MonteCarlo (MC)	MonteCarloOneBlock...	122	132	27427328
quasiRandomGenerator (QRG)	inverseCNDKernel	211	147	3383048
reduction (Red)	reduce5_sm10	62	40	4000
	reduce6_sm10	75	59	20781760
scanLargeArray (SLA)	prescan<false,false>	107	94	14544
	prescan<true,false>	114	102	423560064
	prescan<true,true>	122	108	257651
	uniformAdd	28	27	42696639
transpose (Tr)	transpose_naive	29	28	1835008
	transpose	52	42	2752512

Chapitre 5

Tirer parti de la régularité parallèle

Nous avons vu au cours des chapitres 1 et 2 que les GPU disposaient de mécanismes leur permettant d'exploiter la régularité des applications parallèles pour les exécuter efficacement.

En effet, le comportement des threads de programmes parallèles est dicté par des algorithmes de plus haut niveau : leurs actions sont coordonnées et corrélées. De la redondance peut également apparaître lorsqu'un algorithme parallèle abstrait est exprimé dans un modèle de programmation plus restrictif. Ce type de régularité est manifeste dans les algorithmes d'algèbre linéaire dense, dont les motifs d'accès mémoire et de contrôle sont très réguliers [VD08].

D'autre part, les données issues de processus continus exhibent de la régularité : les données adjacentes dans l'espace ou dans le temps sont fortement corrélées. Par exemple, les données vidéo peuvent être compressées d'un facteur 100 par rapport à un codage naïf sans altération perceptible. Ce fait est bien connu de la communauté du rendu graphique, et il est exploité pour la compression de textures et de zones de rendu en mémoire.

La capture de ces formes de régularité permet d'augmenter la performance des architectures parallèles sans pour autant nécessiter de mécanismes matériels coûteux ou complexes. Les processeurs SIMD en sont un exemple. Ces gains concernent non seulement la puissance de calcul à surface de silicium équivalente, mais aussi l'efficacité en énergie [Hin10].

Nous viserons dans ce chapitre à formaliser la notion de régularité, ainsi qu'à explorer d'autres techniques permettant d'en tirer parti.

Nous introduirons dans la section 5.1 les notions de régularité séquentielle et parallèle et étudierons leurs mise en œuvre dans les architectures actuelles. Nous nous pencherons plus en détail sur la régularité séquentielle dans la section 5.2, et aborderons ses limites dans le cadre des architectures parallèles.

Nous présenterons section 5.3 un mécanisme entièrement dynamique permettant d'exploiter la régularité de contrôle, c'est-à-dire la capacité à maintenir et restaurer la synchronisation des threads au sein d'un warp.

Nous définirons les concepts de vecteurs uniformes et affines dans la section 5.4, et montrerons qu'ils permettent de détecter de nombreuses formes de régularité parallèle : branchements uniformes, accès mémoire à pas constant, calcul redondants et registres scalaires. Nous quantifierons la fréquence d'occurrence de ces vecteurs. Nous considérerons des moyens de les identifier de manière statique à la compilation, ou de manière dynamique au moment de

l'exécution.

La section 5.5 considérera des moyens d'adapter l'architecture et la micro-architecture pour tirer avantage des vecteurs uniformes et affines.

Nous considérerons également d'autres mécanismes exploitant la régularité parallèle, tels qu'un partage de tables entre threads d'un même warp qui exploite la régularité des données virgule flottante. La section 5.6 présentera une telle application à l'unité d'évaluation de fonctions.

5.1 Introduction

Les GPU actuels se reposent quasi-exclusivement sur le parallélisme de données pour maintenir un débit d'exécution élevé. Ce mode de parallélisme est explicite et impose peu de contraintes de synchronisation. Il est ainsi facile à exploiter et facilite le passage à l'échelle.

En revanche, nous avons vu au chapitre 1 qu'il existe une contrepartie : en exécutant des flux de données totalement indépendants, on perd par ailleurs en termes de localité. La localité des données est donc un enjeu crucial pour les architectures parallèles.

D'autre part, le fait de distribuer un problème parallèle sur un ensemble de threads de manière rigide conduit à de la redondance dans les calculs et dans la mémoire. Par exemple, Volkov a analysé l'utilisation des registres du noyau SGEMM des CUBLAS 1.1 [Vol10a]. Parmi les 15 registres alloués à chaque thread,

- 2 contiennent des valeurs répliquées exactement entre tous les threads,
- 7 sont des pointeurs ou des index redondants, dont la valeur ne varie que d'une constante entre un thread et son voisin,
- 4 sont utilisés de manière temporaire pour des calculs intermédiaires,
- 2 seulement contiennent des données utiles dont la sauvegarde est nécessaire.

Ainsi, dans cet exemple, seuls 13 % des registres nécessitent réellement d'être conservés par chaque thread pendant toute la durée de son exécution. Les autres sont potentiellement redondants. Nous avons vu dans la section 1.3 que les architectures AMD pouvaient partager les registres temporaires entre threads, permettant de s'attaquer aux 4 registres temporaires. Nous nous intéresserons dans la section 5.4 à réduire l'empreinte des 9 autres registres redondants.

Pour englober l'ensemble des effets de localité et de corrélation, nous introduirons dans ce chapitre la notion de *régularité*, que nous séparerons en *régularité séquentielle* et *régularité parallèle*.

Nous décrirons plus en détail chacune de ces formes de régularité dans les deux sections suivantes, et considérerons les causes de régularité, puis les applications possibles.

5.1.1 Régularité séquentielle

Considérons figure 5.1 un corps de boucle rédigé dans le modèle séquentiel. La régularité s'exprime dans le temps, entre itérations successives : on peut parler de régularité lorsque les valeurs manipulées par une instruction statique donnée varient peu d'une *itération* à une autre.

Ainsi, la localité temporelle et la localité spatiale correspondent à de la régularité séquentielle sur les adresses. Celle-ci est exploitée par les caches. Les phénomènes de corrélation

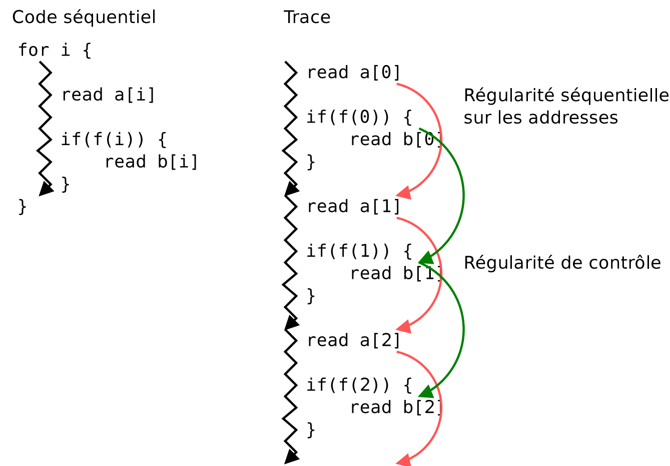


FIGURE 5.1 – Exemple de régularité séquentielle dans un modèle séquentiel. Il existe ici une corrélation entre les valeurs manipulées par les exécutions successives de chaque instruction statique.

des branchements dont tirent profit les prédicteurs de sauts peuvent être vus comme des manifestations de la régularité séquentielle sur les conditions de saut. Enfin, la localité de valeurs [LWS96] correspond à de la régularité séquentielle sur les données.

Nous verrons que les architectures parallèles se prêtent difficilement à l'exploitation des formes traditionnelles de régularité telles que la localité temporelle et spatiale, ou la régularité des branchements. En revanche, nous présenterons une autre forme de régularité plus adaptée au modèle de programmation SPMD : la régularité parallèle.

5.1.2 Régularité parallèle

Le pendant SPMD de l'exemple de régularité séquentielle de la figure 5.1 est présenté sur la figure 5.2. Dans ce dernier cas, le code a une sémantique parallèle : la régularité s'exprime alors dans l'espace. Les valeurs manipulées par une instruction statique donnée varient peu d'un *thread* à un autre. Lorsque le code est exécuté sur un processeur SIMD, cette régularité peut être vue comme une corrélation des données entre les composantes d'un vecteur.

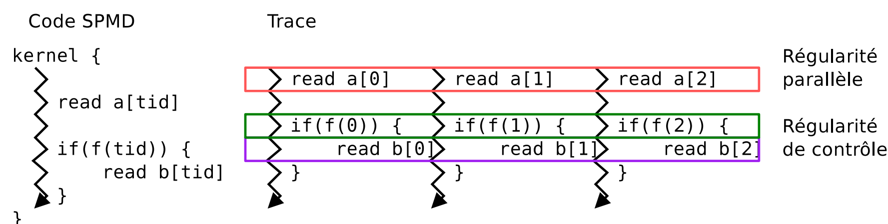


FIGURE 5.2 – Régularité parallèle dans le modèle SPMD. La corrélation se manifeste entre le instructions concurrentes de threads distincts.

La régularité parallèle est aussi appelée cohérence dans la communauté du calcul gra-

TABLE 5.1 – Correspondance des mécanismes exploitant chacune des deux formes de régularité

Effet	Régularité séquentielle	Régularité parallèle
Régularité d'instructions	Cache d'instructions, cache de trace...	Unités SIMD, exécution en pipeline
Régularité de contrôle	Prédicteur de sauts	Vectorisation dynamique (SIMT)
Localité mémoire	Cache conventionnel	Cache de textures
Régularité des accès mémoire	Tampons de combinaison d'écritures, préchargement	Fusion des accès (*coalescing*)
Localité de valeurs	Cache de valeurs	Compression de textures et de zones de rendu

phique, par analogie avec la cohérence des ondes lumineuses [Gle09]. Par opposition, on appellera *divergence* les situations où le comportement des threads voisins diffère.

On préférera les termes *séquentiel* et *parallèle* aux adjectifs *temporel* et *spatial* pour qualifier la régularité. En effet, nous nous référons ici à l'ordre d'exécution logique dicté par le modèle de programmation plutôt que l'ordre d'exécution réel.

Les GPU actuels disposent d'ores et déjà de mécanismes exploitant la régularité parallèle. Ainsi, il est possible d'établir un parallèle entre les optimisations des processeurs généralistes et celles des GPU (table 5.1).

Ces mécanismes ont en commun le fait d'être optimisés pour les situations de régularité, tout en restant fonctionnels mais offrant des performances progressivement dégradées à mesure que la régularité disparait.

5.1.3 Sources de régularité

Les raisons conduisant à la présence de régularité parallèle sont voisines de celles qui causent la régularité séquentielle, telles qu'identifiées par Lipasti, Wilkerson et Shen [LWS96]. Nous pouvons classer ces raisons parmi deux catégories.

D'une part, la régularité s'explique par des raisons algorithmiques : les programmes parallèles contiennent typiquement des sections de code intrinsèquement séquentiel, ou non parallélisé. Lorsque de tels programmes sont écrits sous forme SPMD, ces sections se retrouvent exécutées par tous les threads. Ces threads effectuent alors les mêmes calculs sur des données identiques. Lorsqu'une machine SIMT exécute le programme, elle manipule des vecteurs uniformes. De façon similaire, la manière naturelle d'opérer sur des données consécutives et indépendantes est de les distribuer à tour de rôle entre les threads (organisation SoA). Les threads voisins travaillent alors sur des données contiguës, ce qui conduit à l'apparition de régularité parallèle sur les adresses, voire sur les données.

D'autre part, on peut invoquer des raisons architecturales : le contrôle de flot irrégulier dégradant les performances sur les architectures SIMD, les programmeurs sont incités à privilégier la régularité. De fait, les conditions de saut et les calculs qui leur sont associés sont

souvent uniformes. De même, les accès mémoires devant respecter une certaine régularité pour être efficaces, les lectures et écritures de vecteurs sont typiquement à pas unitaire. Ainsi, ce phénomène tend à s'auto-entretenir.

De plus, la souplesse de programmation dans les langages parallèles tels que C pour CUDA tend à progresser. Les versions de CUDA suivant l'arrivée de l'architecture Fermi introduisent des facilités telles que les caches, la pile d'appels, la récursivité, les appels virtuels et les conventions d'appels permettant la liaison dynamique de modules [NVI09b]. Nous pouvons conjecturer que le surcoût non lié aux calculs eux-même sera en augmentation. De telles opérations redondantes représentent une source de régularité.

5.1.4 Applications

Nous nous intéresserons dans ce chapitre à des mécanismes permettant d'identifier différentes formes de régularité parallèles. Plusieurs applications peuvent bénéficier de la détection de la régularité. D'une part, cette analyse peut profiter à la compilation de code SPMD vers des plate-formes conventionnelles, de type processeurs généralistes. D'autre part, même dans le contexte des GPU, des modifications matérielles minimales permettraient de minimiser les calculs et les mouvements de données redondants, conduisant à des réductions de l'énergie consommée. Enfin, la possibilité d'extraire facilement des calculs scalaires à la compilation peut avoir des retombées directes sur la conception en amont des futures architectures GPU.

Exécution de code SPMD sur processeurs conventionnels La grande majorité des architectures conventionnelles offrent un mélange de ressources d'exécution scalaires et SIMD. Tous les microprocesseurs haute performance actuels, voire des processeurs embarqués sont équipés d'extensions SIMD à vecteurs courts. Par exemple, l'architecture Intel x86 a connu plusieurs générations d'extensions successives : SIMD 64 bits avec MMX et AMD 3DNow ! [OFW99], 128 bits avec SSE [Int10a], et des extensions 256 bits et 512 bits ont été annoncées avec AVX [Int09a] et LRBni [SCS+08].

Malgré leur omniprésence, ces extensions SIMD ont connu jusqu'ici un succès limité en dehors de bibliothèques optimisées à la main. Elles nécessitent soit d'utiliser des fonctions spécifiques au compilateur, offrant un modèle de programmation complexe et non portable, soit d'utiliser des outils de vectorisation automatique depuis du code séquentiel, qui échouent encore en présence de dépendances et de structures de contrôle complexes.

Par contraste, le modèle de programmation SPMD est simple à appréhender et se prête bien à toute une classe d'applications parallèles, ce qui est illustré par son succès dans le cadre de la programmation des GPU. Nous nous intéresserons donc à la compilation de code SPMD tel que CUDA ou OpenCL vers des CPU avec extensions SIMD.

De tels compilateurs existent pour permettre l'exécution de shaders graphiques sur des architectures de type CPU SIMD. LLVMPipe du projet Gallium [Rus10] génère notamment du code SIMD à partir de shaders GLSL. Le pipeline de rendu logiciel de Larrabee est prévu pour fonctionner de manière analogue [SCS+08]. Des projets similaires existent pour compiler du code généraliste CUDA [DKYC10, SGM+10] ou OpenCL [AMD09a], mais ne génèrent pas de code SIMD à l'heure actuelle.

Le modèle d'exécution SIMT permet aux GPU de gérer efficacement les situations de divergence de contrôle et de divergence mémoire. À l'inverse, les processeurs généralistes disposent de jeux d'instructions SIMD relativement pauvres, mais peuvent exécuter en retour du code scalaire de manière extrêmement efficace.

Pour ce type de jeu d'instructions SIMD, c'est au compilateur que revient la tâche de détecter les situations de régularité parallèle. Cela concerne en particulier les branchements uniformes, les accès mémoire à pas unitaires, et les instructions et registres scalaires. Nous nommerons *scalarisation* l'opération qui consiste à identifier des opérations scalaires dans un code SPMD, par opposition à la vectorisation qui consiste à extraire des opérations vectorielles d'un code scalaire.

Vers des GPU plus efficaces La présence même des mécanismes existant dans les GPU qui exploitent la régularité parallèle (table 5.1). montre que ce concept est bien connu de l'industrie. L'exécution du code SIMT en pipeline profite également de la régularité de données pour diminuer la consommation. Lorsque des données similaires se suivent dans le pipeline, l'activité des transistors est en effet réduite [Gle09].

Nous aborderons deux approches permettant de généraliser ces mécanismes dans les sections 5.4 et 5.6.

D'une part, une grande partie des situations de régularité évoquées précédemment pourraient être identifiées de manière statique dès la compilation, plutôt que dynamiquement lors de l'exécution comme aujourd'hui. Le cœur d'exécution pourrait donc être simplifié, permettant des réductions de surface et de consommation.

D'autre part, l'approche dynamique peut également être étendue en ajoutant de nouveaux mécanismes de détection de régularité de données. Les informations obtenues permettront d'améliorer les performances.

5.2 Régularité séquentielle

Nous analyserons dans cette section l'application de la régularité séquentielle aux architectures parallèles.

5.2.1 Caches conventionnels sur GPU

L'architecture Fermi de NVIDIA offre une hiérarchie de caches traditionnelle. Un premier niveau local est composé de 16 Ko ou 48 Ko par SM [NVI09b]. Un second niveau est distribué à hauteur de 128 Ko par partition mémoire.

Un des rôles principaux du cache L1 est de mémoriser la pile d'appels, qui contient les variables locales et les arguments de fonctions qui ne sont pas maintenus dans les registres. Or, dans le modèle SIMT, chaque thread doit disposer de sa propre pile d'appels.

Sur Fermi, le nombre maximal de threads pouvant coexister sur un SM est 1536. Lorsque chaque thread opère sur des données disjointes, chacun dispose alors d'une portion de 10 ou 32 octets du cache L1 suivant la configuration choisie. Ainsi, dans le pire des cas, chaque thread

pourra mémoriser uniquement un pointeur sur 64 bits dans la portion de cache L1 qui lui est allouée.

Il apparaît clairement qu'une utilisation non-coordonnée de la mémoire, telle qu'une sauvegardes de registres dans la pile, pourra très difficilement offrir suffisamment de localité temporelle pour assurer un taux de succès raisonnable dans le cache L1. Le développeur sera forcé de faire des compromis entre localité et parallélisme.

5.2.2 Caches de valeurs

Application : évaluateur généralisé Comme nous l'avons envisagé dans le chapitre 3, il pourrait être avantageux de réutiliser le matériel de l'unité d'interpolation et d'évaluation de fonctions pour approximer des fonctions arbitraires. Nous proposons de remplacer ici les tables en ROM par un chemin d'accès à la mémoire par l'intermédiaire des caches, permettant au programmeur d'application ou de bibliothèque d'y placer ses propres coefficients.

Cette approche est utilisée notamment pour l'évaluation de fonctions à l'aide de petites tables sur CPU. Nous tenterons d'évaluer la pertinence d'une telle approche sur GPU, en se basant sur un cache L1 du type de celui de l'architecture Fermi.

Les évaluations de fonctions élémentaires sont suffisamment courantes dans les applications de rendu graphique pour justifier une mise en œuvre matérielle sur tous les GPU existants. Par conséquent, nous considérons également ici des applications graphiques. Nous récupérons les données de ces applications en interceptant les évaluations de fonctions $\cos$, $\sin$, $\exp_2$, $\log_2$, rcp et rsqrt dans le pipeline de rendu logiciel de la bibliothèque OpenGL Mesa. Les applications graphiques que nous avons retenues sont celles de la suite de benchmarks SPEC Viewperf 8.1 [SPEb]. Nous considérons aussi l'application de calcul de trajectoires AIAA du programme Spacetrack [VCHK06], ainsi que l'application de transferts radiatifs GPU4RE décrite dans la section 3.3.

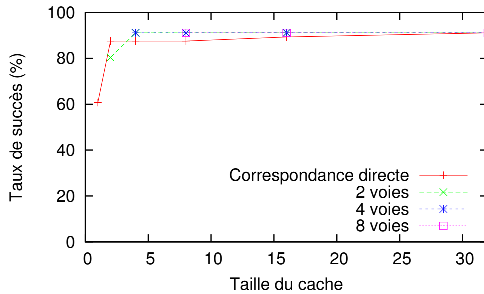
L'objectif est d'évaluer le potentiel de régularité séquentielle des évaluations de fonctions dans ces applications. Nous simulons des caches auxquels les applications accèdent dans un ordre d'exécution séquentiel.

Résultats Nous considérons des tables de coefficients de taille 64, ce qui correspond à l'implémentation matérielle des fonctions élémentaires sur Tesla [OS05]. Les figures 5.3, 5.4 et 5.5 présentent les taux de réussite des caches présents dans les SM sur les applications considérées, en fonction de la taille (de 1 à 32 entrées) et de l'associativité du cache (de 1 à 8 voies).

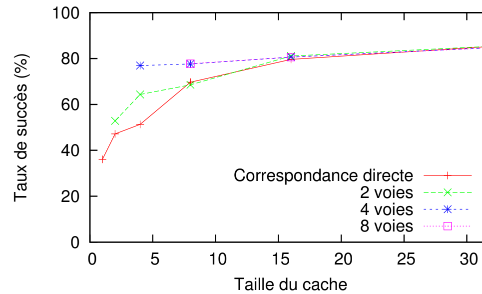
Différents appels non-corrélés se retrouvent en compétition dans le cache, menant à des conflits fréquents. Les tailles et associativités de cache donnant des taux de succès importants ont un coût relativement élevé.

5.2.3 Implications

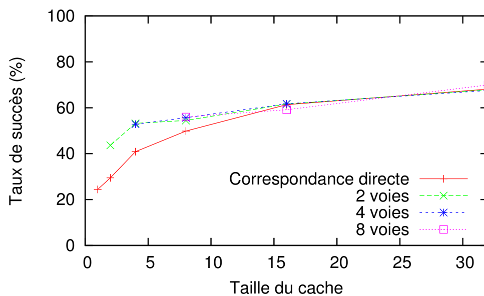
Si les caches et prédicteurs de branchements sont aujourd'hui universellement répandus, les propositions de mécanismes exploitant directement la localité de valeurs au niveau des opérandes des instructions ont connu un succès limité [Ric93, LWS96, SS97]. La principale difficulté vient du fait que les éléments redondants à identifier sont séparés dans le temps,



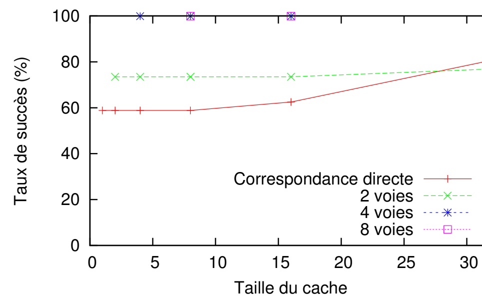
(a) Viewperf Ensight.



(b) Viewperf Lightscape.



(c) Viewperf Maya.



(d) Viewperf SolidWorks.

FIGURE 5.3 – Simulation d'un cache sur les appels à la fonction cos dans les SPEC Viewperf.

nécessitant de maintenir des caches de valeurs dont le coût est difficilement justifiable alors que les mémoires deviennent de plus en plus coûteuses relativement aux unités de calcul.

Dans le cadre des architectures parallèles, ce problème est aggravé par l'augmentation de la taille des jeux de données due à l'exploitation du parallélisme de données.

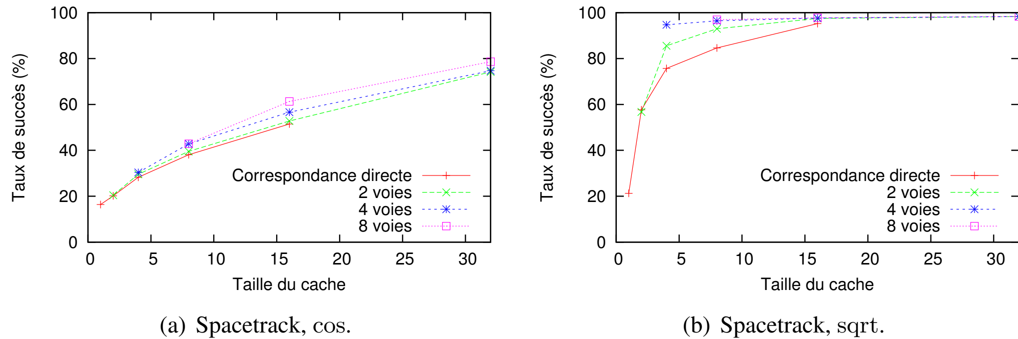
Ces constatations nous amènent à nous concentrer sur la régularité parallèle dans la suite de ce chapitre.

5.3 Régularité de contrôle

Le cas le plus flagrant d'exploitation de la régularité parallèle sur les GPU est le mode de fonctionnement SIMT, tirant parti de la régularité d'instructions. Cette dernière est directement liée à la régularité de contrôle. Nous présenterons dans cette section un moyen d'extraire de la régularité de contrôle de manière entièrement dynamique et à faible coût.

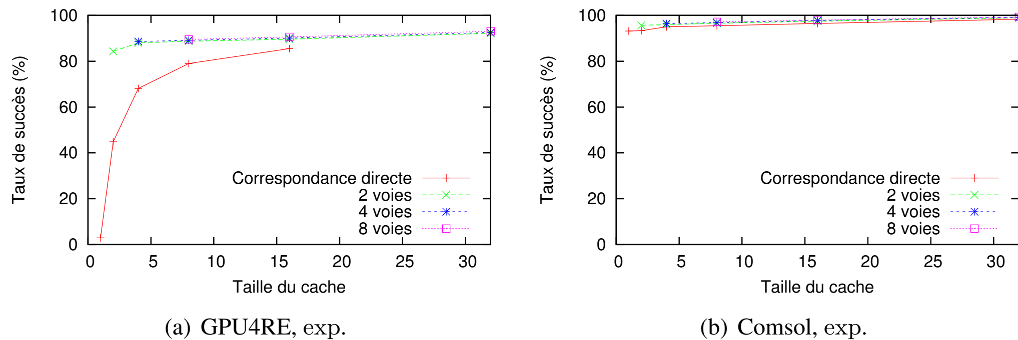
5.3.1 Problématique

Nous dirons que le contrôle est régulier, ou *non divergeant*, lorsque les chemins suivis par les threads d'un warp donné sont corrélés. Lorsque le chemin suivi est identique, les threads peuvent partager le même pointeur d'instructions et être exécutés par des unités SIMD. En cas de divergence, il est nécessaire de recourir à un mode d'exécution moins efficace.



(a) Spacetrack, cos. (b) Spacetrack, sqrt.

FIGURE 5.4 – Simulation d'un cache sur les appels aux fonctions cos et sqrt dans Spacetrack.



(a) GPU4RE, exp. (b) Comsol, exp.

FIGURE 5.5 – Simulation d'un cache sur les appels à la fonction exp dans GPU4RE et Comsol.

Deux pistes peuvent être suivies pour limiter cette perte de performances. La première approche consiste à assouplir les contraintes de l'exécution SIMD pour tirer parti d'une régularité de contrôle partielle. C'est ce que proposent notamment les travaux de Fung sur la formation dynamique de warps [Fun08]. Le second angle d'attaque est de travailler sur la détection des points de reconvergence du flot de contrôle, aussi nommés points *d'indépendance de contrôle*.

Les points de reconvergence sont typiquement choisis à la compilation et insérés dans le code machine. Ainsi, Fung propose d'utiliser à cet effet les post-dominateurs immédiats [1] dans le graphe de contrôle de flot [Fun08]. Cependant, il existe des cas où cette solution n'est pas optimale, voire où les points de reconvergence optimaux ne peuvent pas être identifiés de manière statique. Qui plus est, des mécanismes traitant conjointement la divergence mémoire et la divergence de contrôle ont été proposés [MTS10]. Ceux-ci créent des points de divergence additionnels difficiles à prévoir pour le compilateur.

Cette problématique rejoint celle de la détection de points d'indépendance de contrôle étudiée dans le contexte des processeurs superscalaires. Ainsi, des méthodes dynamiques offrant une grande précision ont été proposées dans ce cadre [CTW04]. Cependant, leur mise en œuvre

1. Un bloc de base B est un *post-dominateur* d'un bloc A (ou *post-domine* A) si tous les chemins de A jusqu'à la fin du programme passent par B. Le post-dominateur immédiat est le premier post-dominateur dans l'ordre d'exécution du programme.

en matériel a un coût qui est difficilement compatible avec les architectures parallèles actuelles.

Des techniques plus légères à mettre en œuvre sont donc nécessaires. Nous viserons ici à développer une heuristique simple qui nécessite un support matériel supplémentaire minimal par rapport aux techniques basées sur les points de reconvergence statiques des GPU actuels. À l'inverse des techniques actuellement utilisées, elle ne nécessite pas d'inclure des informations explicites de reconvergence dans le jeu d'instructions. Nous qualifierons donc cette méthode dynamique d'*implicite*, par opposition à la technique *explicite* statique utilisée par exemple dans l'architecture Tesla.

Le bénéfice d'un mécanisme entièrement dynamique est double. D'une part, la détection des points de reconvergence devient un détail de la micro-architecture, favorisant la portabilité des binaires d'une génération à une autre. D'autre part, les informations statiques d'indépendance de contrôle forment aujourd'hui la dernière barrière qui sépare encore un jeu d'instructions séquentiel d'un jeu d'instructions SIMT. Cette barrière est levée si nous sommes capables de détecter l'indépendance de contrôle à l'exécution. Ainsi, il devient possible d'exécuter en mode SIMT n'importe quel binaire existant, et ainsi de lier dynamiquement un programme parallèle avec du code de bibliothèque existant, et de le vectoriser à l'exécution.

5.3.2 Reconvergence implicite

Nous nous basons sur deux piles de masques implantées par compteurs et deux piles d'adresses, l'une servant aux structures conditionnelles (notée I) et l'autre aux boucles (notée L).

La supposition que nous faisons est que les points de reconvergence se trouvent au point le plus « bas » du code qu'ils dominent, c'est-à-dire à l'adresse la plus grande. Collins, Tullsen et Wang ont mesuré que cette supposition s'avérait correcte dans 94 % des branchements conditionnels des SPECint [CTW04]. Les noyaux de calcul CUDA actuels offrant un flot de contrôle nettement plus régulier, nous n'avons rencontré aucun contre-exemple dans les applications parallèles présentées section 4.5.

La stratégie suivie consiste alors à toujours tenter d'exécuter les instructions d'adresse inférieure lorsqu'il s'agit de décider quelle branche exécuter, de façon à ne pas dépasser un point de convergence éventuel.

Structures conditionnelles Lorsqu'un branchement vers l'avant divergeant est rencontré, le masque courant et l'adresse de destination sont sauvegardées dans la pile I, le masque courant est mis à jour et l'exécution continue dans la branche non-prise. Le haut de la pile d'adresses I contient alors la cible du saut, soit l'autre branche en attente d'être exécutée (fig. 5.6).

Après l'exécution de chaque instruction, le compteur d'instruction suivante (NPC) est comparé avec le haut de la pile d'adresses I.

En cas d'égalité, le point de convergence est atteint (`endif`), l'exécution peut alors continuer avec le masque sauvegardé, après avoir dépilé le masque et l'adresse. Plusieurs points de convergence peuvent être présents sur la même instruction. Dans ce cas, il faut répéter la comparaison et dépiler autant de fois que nécessaire, ce qui peut nécessiter la ré-exécution de l'instruction par un mécanisme analogue à celui que nous avons décrit section 2.4.2.

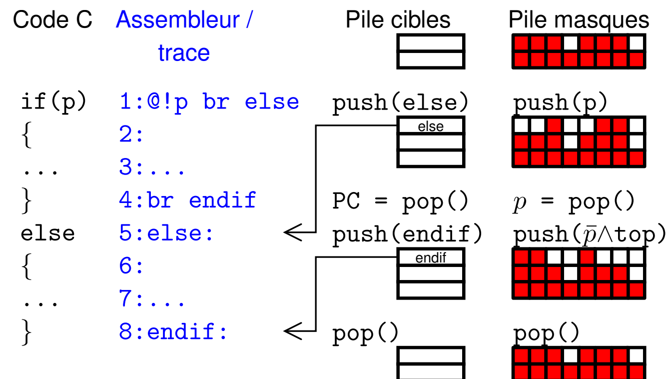


FIGURE 5.6 – Exemple de branchement avec reconvergence implicite.

Il se peut également que l'instruction courante soit un saut vers une adresse se trouvant au-delà de la cible du saut précédent, par exemple avant un bloc `else` (ligne 4 de la figure 5.6). Dans ce cas, NPC est strictement supérieur au sommet de la pile I. Le contrôle est alors transféré à l'autre branche en attente, en échangeant NPC avec le haut de la pile I et en inversant le masque courant.

Boucles Les boucles sont identifiées par un branchement arrière, conditionnel ou inconditionnel. Lorsqu'un tel branchement est rencontré et avant d'effectuer le saut, on compare le sommet de la pile L avec l'adresse de l'instruction suivante (NPC=PC+1).

En cas de différence et si le saut est pris par au moins un thread, on empile NPC et le masque courant sur la pile L, de façon à retenir l'adresse de fin de la boucle.

En cas d'égalité et si aucun thread n'effectue le saut (sortie de boucle), le masque est restauré d'après le masque présent dans la pile et l'entrée correspondante est dépilée de la pile L.

Autres structures de contrôle Les deux techniques présentées permettent de gérer efficacement les structures de contrôle strictement imbriquées. Elles sont également suffisantes pour exécuter du flot de contrôle plus irrégulier, tel que les instructions `break` et `continue` du langage C. L'ordre d'exécution suivi pourra cependant être sous-optimal pour ce type de flots.

5.3.3 Validation

Coût en matériel Considérons une instruction de saut direct, suivant le pipeline présenté figure 5.7. Les adresses de la prochaine instruction dans le cas où le branchement est pris et celui où il n'est pas pris sont toutes deux connues dès l'étape de décodage de l'instruction. Il est possible de les comparer chacune avec les adresses du haut des piles I et L. Le résultat de la comparaison déterminera la valeur que prendra NPC dans chaque cas : pris ou non-pris. Ce calcul des deux NPC peut s'effectuer en parallèle du calcul du prédicat et du masque. Le reste du pipeline reste identique : la valeur du prédicat servira à sélectionner la valeur de NPC

parmi les deux alternatives. L'état des piles peut également être mis à jour en dehors du chemin critique. La technique proposée n'a donc pas d'impact notable sur la latence des instructions de saut, ni sur les instructions de calcul.

FIGURE 5.7 – Premiers étages du pipeline d'exécution du SM Tesla, modifiés pour gérer la reconvergence implicite.

Le matériel supplémentaire consiste en quatre comparateurs. Les piles de masques existent déjà dans l'architecture Tesla. Cette méthode permet également de maintenir les piles de masque de manière compacte au moyen de compteurs d'activité [KP93], contrairement à la solution retenue par l'architecture Tesla.

Évaluation Nous cherchons maintenant à quantifier l'efficacité de la méthode implicite proposée. En raison de disparités de comportements pouvant exister entre plusieurs noyaux de calcul d'un même programme CUDA, nous avons choisi d'instrumenter séparément chaque noyau. Le même code binaire est exécuté avec l'algorithme de reconvergence explicite et notre algorithme implicite. Dans ce dernier cas, les instructions de reconvergence (`join`) sont ignorées et ne sont pas inclues dans le décompte des instructions exécutées. Cependant, les instructions `nop` qui peuvent avoir été insérées pour satisfaire les contraintes du système de branchement explicite sont exécutées dans les deux cas.

La figure 5.8 présente pour chaque noyau le nombre moyen de threads actifs par warp au cours de l'exécution, suivant l'algorithme de branchement utilisé. Une valeur de 32 indique un taux d'occupation de 100%.

Nous observons que l'algorithme de branchement implicite reste toujours au moins aussi efficace que l'algorithme explicite. Cela reste vrai dans une moindre mesure lorsque l'exécution des instructions `join` est prise en compte même en mode implicite. En effet, l'algorithme de base de Tesla nécessite d'exécuter deux fois les instructions se trouvant aux points de convergence.

Nous observons également une différence significative lors de l'exécution du noyau *inverseCND* de *quasiRandomGenerator*. Elle est due à la présence d'une instruction `return` à l'intérieur d'une imbrication de structures conditionnelles. En effet, l'algorithme explicite

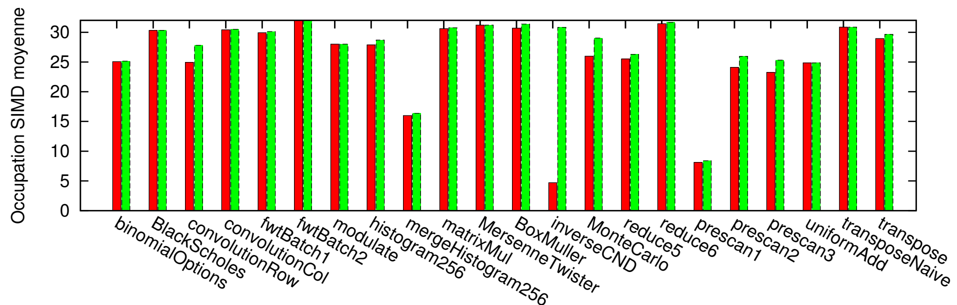


FIGURE 5.8 – Occupation SIMD en fonction de l'algorithme de branchement utilisé.

tel qu'il est décrit dans le brevet de NVIDIA ne permet pas de désactiver immédiatement les threads exécutant cette instruction.

Ainsi, il est possible de détecter de manière entièrement dynamique les points d'indépendance de contrôle. Tout jeu d'instructions scalaire peut alors être exécuté en mode SIMT sans modification. Par rapport au mécanisme statique actuellement utilisé, le surcoût en surface, latence et consommation est minimal.

5.4 Vecteurs uniformes et affines

Nous nous intéresserons dans cette partie à la détection statique et dynamique de la régularité parallèle qui apparaît dans les adresses et conditions de saut. Cette détection a des conséquences directes sur l'identification de la régularité mémoire et de la régularité de contrôle, respectivement.

Le cas le plus extrême de régularité parallèle se présente lorsque tous les threads d'un warp manipulent la même valeur lors de l'exécution d'une instruction SIMT. Un autre cas de régularité se présente lorsque des threads d'identifiant successifs manipulent des valeurs successives, ou variant suivant un pas constant.

Nous montrerons dans cette section que ces deux motifs se reproduisent fréquemment lors de l'exécution de programmes CUDA, et qu'il est possible de les identifier de manière statique ou dynamique pour un faible coût matériel.

Définitions Pour mettre en évidence ce type de régularité, nous introduisons quelques définitions. Le modèle d'exécution considéré ici est le SIMT, où les threads sont regroupés en warps partageant le même séquenceur d'instructions. Une instruction exécutée sur un warp manipule des *vecteurs*, chaque composante du vecteur appartenant à un thread différent. Il semble naturel de considérer des vecteurs dont la largeur coïncide avec la taille d'un warp. Cependant, nous serons amenés par la suite à considérer des largeurs différentes.

Définition 5.1 (Vecteur uniforme) *Un* vecteur uniforme V *est défini comme ayant toutes ses composantes initialisées à la même valeur* $V_i = x$.

Définition 5.2 (Vecteur affine) *Un vecteur V est dit* affine *lorsque chacune de ses composantes V_i, interprétée comme un entier non signé, est telle que $V_i = x + iy$, pour x et y entiers.*

Nous nommerons x la *base* et y le *pas* du vecteur affine.

Définition 5.3 (Vecteur générique) *Un vecteur qui ne répond pas à la définition d'affine est dit* générique.

Nous pouvons noter que les vecteurs uniformes sont des cas particuliers de vecteurs affines à pas nul ($y = 0$). La définition de générique exclut également les vecteurs uniformes.

Dans le cas où une partie des threads sont inactifs, les composantes correspondantes des vecteurs ne sont pas prises en compte. Lorsque les entiers machine sont représentés en complément à deux, comme c'est le cas sur toutes les architectures actuelles y compris les GPU, la définition de vecteur affine peut s'appliquer sans changement aux entiers signés.

5.4.1 Intérêt

Nous nous intéressons ici à la fréquence d'apparition de ce type de vecteurs particuliers dans les opérandes des instructions. Nous utilisons Barra pour analyser les opérandes source et destination de chaque instruction exécutée. Ces vecteurs opérandes sont classés parmi les catégories uniforme, affine, ou générique. Les applications étudiées ici sont celle du SDK CUDA.

La proportion des opérandes uniformes ou affines obtenus en entrée et en sortie est représentée sur la figure 5.9. Les opérandes sources représentent les données transférées depuis le banc de registres vers les unités d'exécution. Les opérandes destinations représentent non seulement les écritures dans le banc de registres, mais également les calculs eux-même. En effet, l'ensemble des instructions de l'architecture G80 initiale se comportent de cette façon. Les seules exceptions seraient les instructions de réduction (somme horizontale, instruction de vote…) qui acceptent des sources non uniformes et renvoient un résultat uniforme. Il pourra être pertinent de séparer le concept d'exécution uniforme de celui d'écriture uniforme dans le banc de registres sur d'autres architectures, bien que ce ne soit pas nécessaire pour les exemples considérés ici.

Nous observons que 27 % (respectivement 44 %) des vecteurs lus depuis le banc de registres sont uniformes (affines) et 15 % (28 %) des vecteurs écrits sont uniformes (affines).

Certaines applications telles que Transpose, MatrixMul et 3DFD utilisent des CTA de taille 16×16. La majorité des vecteurs uniformes et affines qu'elles manipulent sont donc de taille 16 plutôt que 32. Nous considérons donc figure 5.10 les résultats obtenus en fixant la largeur des vecteurs à 16.

Les résultats de ces applications sont nettement améliorés. Nous observons notamment que 97 % des lectures de registres de Transpose sont des vecteurs affines. Cela n'est pas surprenant dans la mesure où cette application n'effectue que des calculs d'adresses accompagnés de lectures et écritures de vecteurs, mais aucun calcul vectoriel proprement dit.

Les proportions de vecteurs affines et uniformes que nous observons sont suffisamment importantes pour justifier des optimisations au niveau du compilateur, de la micro-architecture voire de l'architecture.

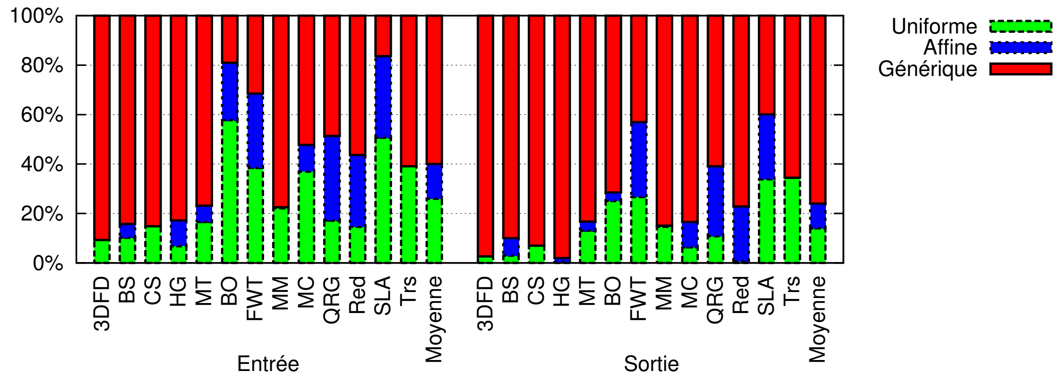


FIGURE 5.9 – Proportion d'opérandes uniformes et affines depuis et vers le banc de registres, pour des vecteurs de largeur 32.

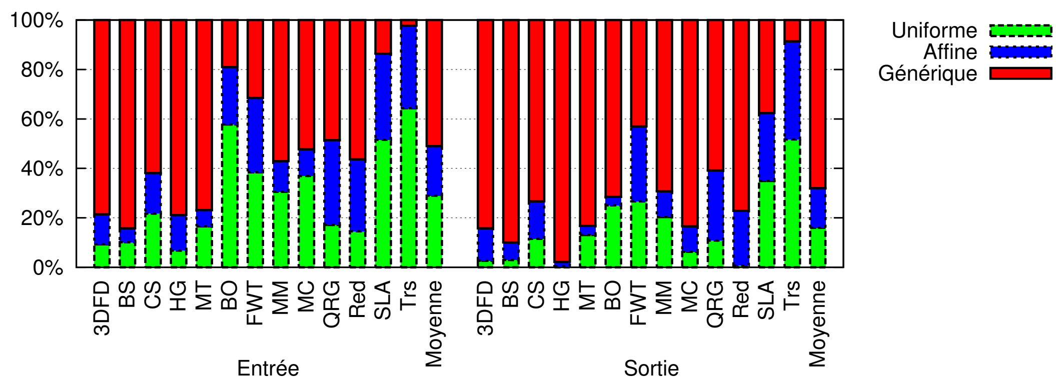


FIGURE 5.10 – Proportion d'opérandes uniformes et affines depuis et vers le banc de registres, pour des vecteurs de largeur 16.

5.4.2 Scalarisation

Notre objectif est d'être capable de détecter à faible coût les vecteurs uniformes et affines. L'approche naïve consistant à analyser chaque vecteur au cours de l'exécution pour en déterminer le type aurait un coût certainement plus élevé que les bénéfices qu'elle apporte ; cependant, c'est celle-ci qui est utilisée pour détecter si les sous-transactions mémoire d'un gather ou scatter peuvent être fusionnées dans l'architecture Tesla. En effet, comme nous l'avons vu au chapitre 2, le coût en latence, débit et consommation des accès mémoire est suffisant pour justifier un tel calcul. Pour étendre ce type d'analyse aux instructions arithmétiques, une technique moins coûteuse est nécessaire.

Une première solution, la scalarisation statique, consiste à modifier le jeu d'instructions en ajoutant des instructions scalaires et des registres architecturaux scalaires. Une seconde approche est de conserver l'architecture existante, pour concentrer les modifications sur la

micro-architecture : nous parlerons de scalarisation dynamique. Nous envisagerons tour à tour chacune de ces approches.

Scalarisation statique Nous considérons ici une étape de détection des vecteurs uniformes et affines intégrée à un compilateur d'applications CUDA vers une architecture de type SIMD. Nous opérons sur une représentation intermédiaire incluant un graphe de flot de données, partitionnant les noyaux en blocs de base [AL+06]. Les registres sont alloués avec assignation statique unique (SSA).

À chaque registre de la forme SSA du noyau, nous associons une étiquette

$$T \in \{\bot, C(v,a), U(a), A(s,a), G(a)\}$$

telle que C est associée aux valeurs constantes, U aux vecteurs uniformes, A aux vecteurs affines, G aux vecteurs génériques (non-affine), ou $\bot$ lorsque l'état est encore inconnu.

D'autres données sont conservées avec l'étiquette :

- $v \in \mathbb{Z}$ est la valeur de la constante,
- $a \in \{\bot, 0, 1, \ldots, a_{\max}\}$ est *l'alignment* de la valeur, défini comme le nombre minimal de zéros terminaux de la représentation binaire de chaque composante du vecteur, ou $\bot$ pour le vecteur nul,
- $s \in \mathbb{Z} \cup \{\top\}$ est le pas du vecteur affine, ou $\top$ s'il est inconnu.

Nous propageons ensuite ces méta-données de registre en registre dans chaque bloc de base suivant des règles semblables à celles présentées table 5.2. Pour propager les étiquettes entre les blocs de base, nous nous basons sur une analyse de flot de données vers l'avant [AL+06].

TABLE 5.2 – Exemples de règles de propagation des étiquettes au travers des instructions, avec $s + \top = \top$, $a \times \bot = a$, $min(a, \bot) = a\ldots$

$$\frac{x : A(s,a) \quad y : A(s',a')}{z = x + y \Rightarrow z : A(s+s', min(a,a'))} \qquad \frac{x : A(s,a) \quad y : U(a')}{z = x \times y \Rightarrow z : A(\top, a \cdot a')}$$

$$\frac{x : A(s,a) \quad y : A(s,a')}{z = x - y \Rightarrow z : U(min(a,a'))} \qquad \frac{x :: A(s,a) \quad y : C(v,a')}{z = x \times y \Rightarrow z : A(s \cdot v, a \cdot a')}$$

Lorsque la première dimension du bloc est multiple de la taille des vecteurs considérés w, nous associons l'annotation $A(1, \log_2(w))$ à la composante x de l'identifiant du thread, et $U(0)$ aux autres dimensions de l'identifiant. Les étiquettes des variables du noyau sont déterminées à partir de celles des constantes et de l'identifiant du thread.

Scalarisation dynamique Nous proposons d'inclure des annotations complétant le banc de registres, et conservés dans une mémoire interne au SM. Ces annotations associent à chaque registre vectoriel un état parmi *Uniforme*, *Affine* et *Générique*, de manière similaire aux étiquettes de l'approche statique. Les données supplémentaires qui sont maintenues par l'analyse statique ne sont plus nécessaires ici, car elles sont présentes dans les registres eux-mêmes et aisément accessibles. L'état indique que le registre associé est du type indiqué avec certitude : un vecteur affine peut être annoté par *affine* ou *générique*, mais en aucun cas par *uniforme*.

Listing 5.1 – Exemple de fausse dépendence envers le masque courant.

```
__global__ void kernel(int * g) {
    int i = 0, j;
    do
    {
        j = i;
    }
    while(i++ < threadIdx.x);
    g[threadIdx.x] = j;
}
```

Les registres contenant l'identifiant du thread sont initialisés de la même manière que dans le cas de l'analyse statique. Lorsqu'une instruction retournant toujours un résultat uniforme est exécutée (chargement de constante immédiate, chargement avec une adresse immédiate ou relative par rapport à un registre uniforme, réductions. . .), une annotation *uniforme* est associée au registre destination.

Les instructions arithmétiques propagent quant-à-elles les annotations depuis les registres sources vers les registres destinations suivant un ensemble de règles analogues à celles suivies dans le cas statique (table 5.2).

5.4.3 Gestion du flot de contrôle

Ce type d'analyse, qu'elle soit statique ou dynamique, présente des difficultés supplémentaires par rapport à une analyse sur du code séquentiel. En effet, nous devons considérer les cas où les threads composant un warp empruntent des chemins d'exécution différents.

Nous avons vu section 5.3 la divergence de branchements est gérée par prédication sur les architectures SIMT actuelles. Du point de vue de l'architecture, une instruction prédiquée écrira uniquement dans certaines voies de son registre destination, laissant les autres dans leur état précédent. Dans ce cas, même si le vecteur résultant de l'opération est uniforme ou affine, cette propriété ne peut pas être garantie pour le registre de destination. Du point de vue du compilateur, cette situation peut être vue comme une dépendance implicite entre le masque courant et la destination de chaque instruction. Lorsque le masque n'est pas uniforme, la destination ne sera généralement pas uniforme.

Dans le cas de l'analyse dynamique, il suffit de tester la valeur du masque courant avant l'exécution de chaque instruction. Lorsque celui-ci n'est pas uniforme, on considérera que la destination de l'instruction est générique quelle que soit la forme des opérandes sources.

L'analyse statique doit quant à elle considérer qu'il existe une dépendance de contrôle entre le numéro de voie et l'instruction dès lors que l'instruction est soumise à une divergence potentielle. Elle effectue donc une estimation pessimiste.

Inversement, l'analyse statique dispose d'informations qui ne sont pas accessibles lors de l'exécution. En particulier, elle a accès aux durées de vie des variables. Cette information permet de s'abstraire de certaines dépendances.

Considérons à titre d'exemple le code CUDA du listing 5.1. Lors de la première itération

de la boucle, le compteur *i* est uniforme entre tous les threads, et il est incrémenté systématiquement par tous les threads. En revanche, la condition d'arrêt de la boucle n'est pas uniforme. Certains threads continueront d'exécuter la boucle en incrémentant le compteur, tandis que d'autres seront mis dans un état inactif, les bits de masque qui leur sont associés étant alors désactivés. Le registre vectoriel représentant *i* ne sera plus uniforme.

Une analyse plus fine révèle cependant deux propriétés :

- d'un part, la variable *i* n'est plus en vie après la fin de la boucle,
- d'autre part, un thread qui a été désactivé pendant l'exécution de la boucle ne sera plus réactivé avant que tous les threads du warp n'atteignent la fin de la boucle.

En conséquence, la valeur des composantes du vecteur *i* correspondant aux threads inactifs n'a pas d'importance, car elle ne sera jamais relue dans le futur. Le vecteur *i* peut donc être remplacé par un vecteur uniforme dans ce cas.

À l'inverse, la variable *j* reste en vie après la fin de la boucle, et doit toujours être considérée comme un vecteur générique. Il est à noter que la représentation SSA conventionnelle utilisée en compilation ne permet pas de modéliser correctement ce type de dépendance, lié à une exécution par prédication.

Considérons une variable SSA x définie dans un bloc de base A. Lorsque nous déterminons que A peut être exécuté dans un état divergeant, nous examinons chaque bloc B contenant au moins une utilisation de x. Si B est un post-dominateur de A, nous considérons qu'il existe une dépendance de contrôle et nous marquons la variable x comme générique. Dans les autres cas, l'annotation de x est déterminée par les règles abordées précédemment sans prendre en compte de dépendance de contrôle.

En effet, nous supposons que les points de reconvergence sont insérés aux post-dominateurs immédiats [Fun08] ou à un point postérieur dans l'ordre d'exécution du programme, mais en aucun cas plus tôt. Cela signifie que si le bloc B ne post-domine pas A, les threads actifs lors de l'exécution de B seront un sous-ensemble (non strict) des threads actifs en A. Autrement dit, tous les threads exécutant B auront aussi exécuté A auparavant, et aucune dépendance de contrôle ne s'applique.

5.4.4 Problèmes et solutions

Des difficultés techniques peuvent affecter l'efficacité de chacune des méthodes proposées.

Dépassement de capacité. Un dépassement de capacité arithmétique peut survenir dans une composante d'un vecteur affine, même si la base et le pas sont tous deux représentables. Les dépassements de capacité n'ont pas de conséquences directes dans le système de numération en complément à deux, mais le programme peut effectuer des conversions entre différent formats signés ou non signés de taille différente. Par exemple, une valeur sur 16 bits ayant débordé peut être étendue en une valeur sur 32 bits. Elle donnera lieu à un vecteur non-affine.

Des dépassements de capacités ne devant pas se produire dans les calculs d'adresse de programmes bien formés, nous pouvons nous attendre à ce que ce phénomène reste exceptionnel. De fait, nous n'avons rencontré ce cas dans aucun des programmes testés. Cependant, il nous faut considérer cette éventualité pour garantir le respect de la sémantique du programme.

Listing 5.2 – Exemple de divergence absolue.

```
if(threadIdx.x % 2) {   // Condition divergeante
 x1 = 42;
 if(threadIdx.y == 3) { // Condition uniforme
   x2 = 17;    // Bloc A
 }
 x3 = phi(x1,x2)   // Bloc B
 // Utilisation de x3;
}
// Aucune utilisation de x
```

Dans le cas dynamique, ce problème peut être évité en testant les dépassements lorsque des opérations affines sont effectuées, et en ré-exécutant l'instruction ayant provoqué le débordement après avoir dégradé les opérandes d'entrée en vecteurs génériques.

Le cas statique est plus délicat. Dans l'idéal, le cas le plus courant ne doit pas être ralenti par du code visant à contourner un cas pathologique. Pour éviter de générer du code qui ne sera jamais exécuté, le compilateur doit être en mesure de garantir l'absence de débordement dans les calculs d'indices. Il faudra donc effectuer des analyses statiques bornant les intervalles de valeurs possible, à partir d'informations de haut niveau sur les zones d'allocation. Dans notre prototype d'analyse, nous considérons qu'aucun débordement ne se produit.

Dimensions des CTA Pour que la détection statique des vecteurs affines soit utilisable, il est nécessaire que la dimension la plus interne du CTA soit multiple de la taille de vecteur considérée. Cependant, cette valeur n'est connue qu'au moment du lancement du noyau. Bien que des dimensions irrégulières soient fortement découragées [NVI10a], le cas de CTA de largeur 16 est relativement commun dans les applications CUDA optimisées pour Tesla. En effet, si la granularité de divergence de contrôle (largeur du warp) est de 32, la granularité de divergence mémoire (largeur d'une transaction gather/scatter) est de 16. Cela peut poser problème dans le cas statique où la dimension des CTA n'est pas connue à la compilation.

La solution est d'utiliser une compilation juste-à-temps (JIT), telle que celle qui est employée dans Ocelot. Ainsi, un test au moment du lancement du noyau déterminera si l'optimisation est applicable ou non.

Divergence relative L'analyse des dépendances de contrôle par recherche des post-dominateurs présentée dans la section 5.4.3 n'est pas optimale. Considérons par exemple le listing 5.2 en pseudo-code sous forme SSA. Il contient deux affectations à une variable x, qui est décomposée en trois registres SSA x_1, x_2 et x_3. Le registre x_2 sera considéré comme générique par la méthode que nous venons de décrire. En effet, le bloc de base A peut être exécuté dans un état divergeant et B post-domine A.

Il est possible d'identifier correctement x_2 comme uniforme si l'on considère la divergence *relative* plutôt que la divergence *absolue*. En effet, le bloc A n'est pas divergeant relativement à B.

L'analyse de la divergence relative nécessiterait de connaître le niveau d'imbrication de chaque bloc de base au sein des structures de contrôles. Ces informations ne sont pas pré-

sentes dans le code PTX, mais pourraient être obtenues moyennant des transformations sur le code [ZHD04].

5.4.5 Résultats

Évaluation Nous évaluons les méthodes de détection statiques et dynamiques en nous basant sur deux environnements distincts.

Nous effectuons l'analyse statique au sein de l'environnement de compilation Ocelot. Cette analyse est faite au niveau du langage intermédiaire PTX, après allocation de registres SSA. Ocelot est bien adapté à la réalisation d'un prototype de compilateur. PTX constitue un langage intermédiaire suffisamment haut-niveau pour retenir des informations de typage, et suffisamment bas-niveau pour permettre une re-compilation JIT non-intrusive lors de l'exécution.

L'analyse dynamique se doit de disposer d'un modèle d'exécution réaliste. Nous nous basons donc sur Barra pour la réaliser. Les registres du simulateur sont complétés par des étiquettes.

Les résultats de chacune de ces approches sont comparés avec un résultat optimal, obtenu par simulation. Dans le cas dynamique, cela consiste à tester dans Barra pour chaque instruction si les opérandes sources et destinations sont uniformes ou affines. On peut alors en déduire un ratio entre les vecteurs affines identifiés et les vecteurs affines susceptibles d'être identifiés qui quantifiera la précision de la détection.

Dans le cas statique, nous modifions l'émulateur d'Ocelot pour classer chaque opérande du code PTX en fonction de son état au cours de l'exécution entière du programme. Cela revient à exécuter notre analyse statique sur la trace d'exécution. Cette approche fournit une borne supérieure sur la quantité de vecteurs détectables de manière statique.

Les applications que nous analysons sont les noyaux SGEMM et 3DFD, les tests de Rodinia et Parboil et les exemples du SDK CUDA, que nous avons abordés dans la section 4.5. La largeur des vecteurs considérés est de 16.

Instructions et opérandes La figure 5.11 présente la classifications des opérandes de destination des instructions statiques et dynamiques, établie par l'analyse statique.

La proportion d'instructions scalaires parmi les instructions statiques est supérieure à leur proportion parmi les instructions dynamiques. En effet, les calculs scalaires sont souvent intégrés au code d'initialisation, qui est placé en dehors de la boucle interne par le programmeur ou le compilateur. Cet effet est le plus significatif dans du code hautement optimisé tel que SGEMM et 3DFD. Les instructions écrivant des données uniformes représentent alors moins de 8 % des instructions exécutées. Les auteurs du code de SGEMM ont pris en compte le surcoût associé à la réplication de pointeurs et autres données scalaire entre les threads, ce qui les a conduit à lancer moins de threads, mais à effectuer plus de travail par thread afin d'amortir ce surcoût [VD08, Vol10b].

Cependant, les autres tests présentent nettement moins de variations entre les résultats statiques et dynamiques, indiquant que de nombreuses instructions scalaires restent présentes dans la boucle interne.

Le taux d'opérandes scalaires statiques est présenté figure 5.12. Le grand nombre d'opérandes uniformes dans SGEMM s'explique par l'algorithme utilisé pour le produit de matrices,

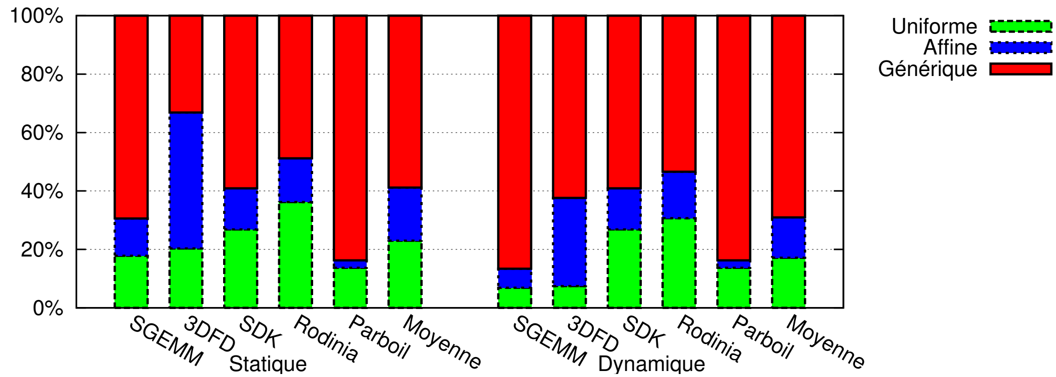


FIGURE 5.11 – Classification des instructions statiques et dynamiques d'après les résultats de l'analyse statique.

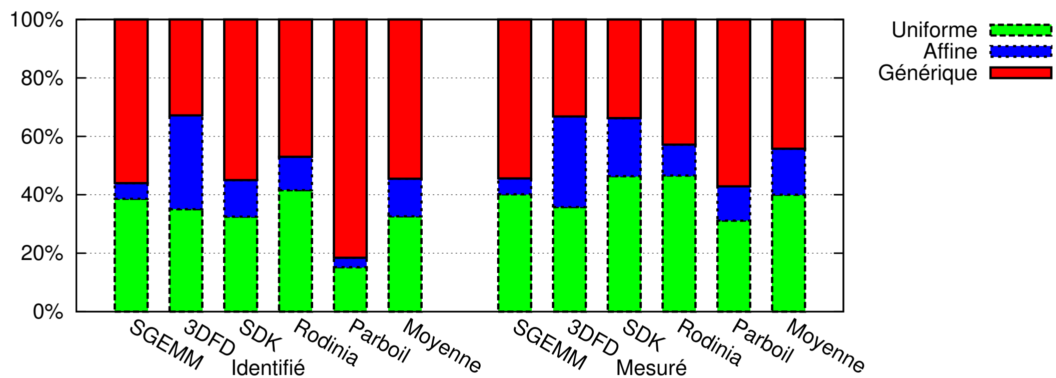


FIGURE 5.12 – Classification des opérandes sources des instructions statiques.

qui conduit à effectuer des multiplications entre scalaires et vecteurs [VD08]. Cet exemple montre que certains algorithmes peuvent tirer parti de registres scalaires pour mémoriser non seulement des adresses et des compteurs, mais aussi des données.

Approche dynamique Les figures 5.13 et 5.14 représentent les proportions respectives d'opérandes uniformes et affines détectées par la technique dynamique proposée. On observe qu'en moyenne, 19 % des entrées et 11 % des sorties peuvent êtres identifiées comme uniformes. Ces taux atteignent respectivement 34 % et 22 % lorsque l'on considère également les données affines.

Lectures & écritures Nous analyse permet de classer les lectures et écritures parmi les catégories suivantes :

– uniforme, lorsque tous les threads du warp accèdent à la même adresse,
– à pas unitaire et aligné, lorsque les accès des threads du warp sont contigus et commencent à une adresse multiple de la largeur du vecteur chargé, ce qui correspond aux

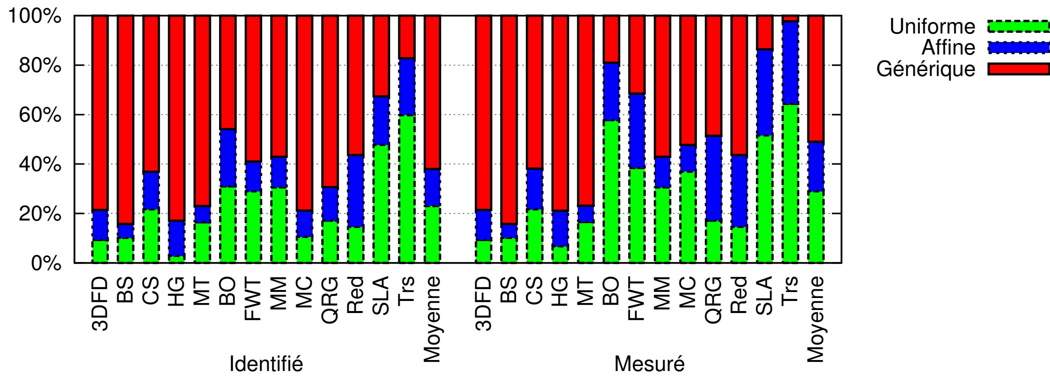


FIGURE 5.13 – Classification des opérandes d'entrée d'après l'analyse dynamique.

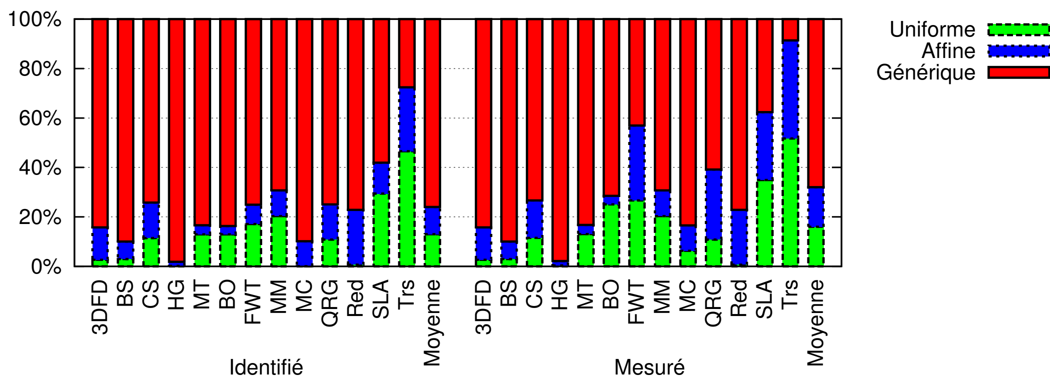


FIGURE 5.14 – Classification des opérandes de sortie d'après l'analyse dynamique.

restrictions du *coalescing* du G80 [NVI10b],
- à pas unitaire non aligné, lorsque les accès sont contigus mais que l'adresse de début n'est pas nécessairement alignée,
- à pas non-unitaire, lorsque les adresses des transactions des threads voisins sont séparés par un pas constant, sans que les zones concernées ne soient contiguës,
- gather et scatter, qui représentent le cas général qui ne correspond à aucune des situations précédentes.

Les figures 5.15 et 5.16 présentent le classement des accès en mémoire globale identifiés par l'analyse statique suivant ces catégories.

Notons la présence d'écritures mémoires (scatter) à des adresses uniformes. Après analyse, il ne s'agit pas de conditions de courses : ces écritures sont prédiquées de manière à n'être exécutées que par un seul thread, par exemple à la dernière étape d'une opération de réduction. Il s'agit ici d'une opération scalaire décrite explicitement par le programmeur.

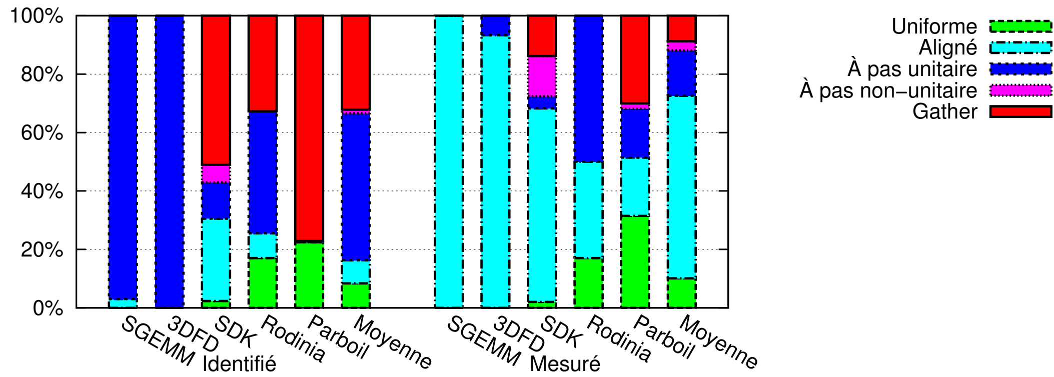


FIGURE 5.15 – Classification des lectures en mémoire globale d'après l'analyse statique.

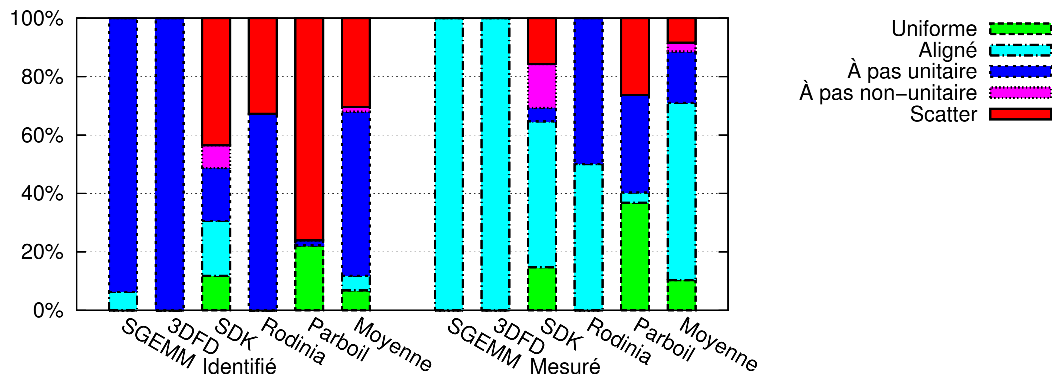


FIGURE 5.16 – Classification des écritures en mémoire globale d'après l'analyse statique.

Branchements uniformes Les proportions d'instructions de saut uniformes identifiées et observées sont présentées sur la figure 5.17.

De nombreuses applications CUDA font appel à des comparaisons entre deux vecteurs affines de pas identique. Dans ce cas, le résultat de la comparaison est un vecteur uniforme. Il est probable que ces opérations apparaissent à la suite d'optimisations de réduction de force effectuées par le compilateur haut-niveau. L'ajout de la prise en compte de ce cas dans l'analyse fait passer le taux de détection statique des instructions affines à 41 %, contre 16 % précédemment. Cela fait apparaître la nécessité de propager les informations liées au pas des variables affines au cours de l'analyse, même dans le cas où les informations sur les pas des accès mémoire ne sont pas nécessaires.

Le taux de succès de la détection des sauts uniforme reste faible dans les programmes du SDK et de Parboil. Une analyse des codes concernés montre que la plupart des cas concernés pourraient être résolus en considérant la divergence relative, selon la méthode présentée dans la section 5.4.4.

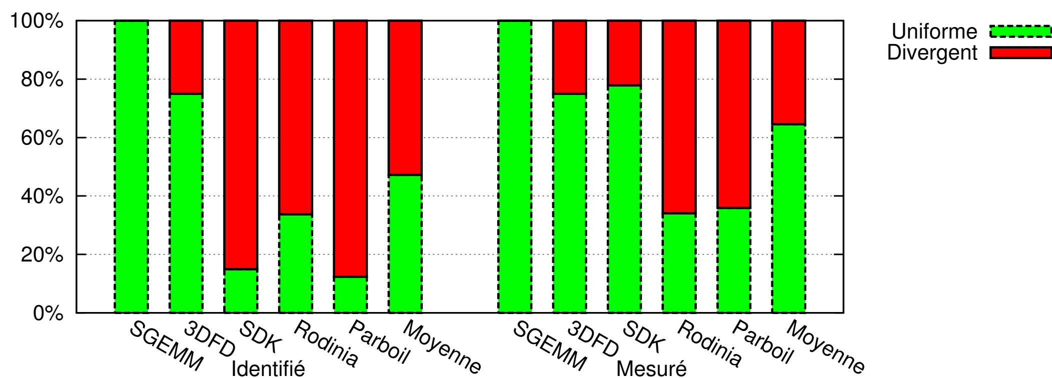


FIGURE 5.17 – Classification des sauts.

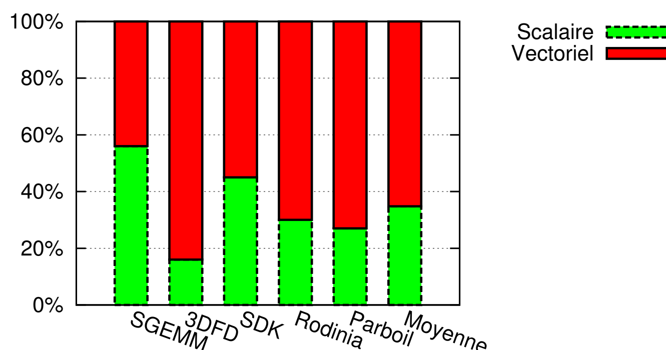


FIGURE 5.18 – Types des registres après allocation sur le code PTX.

Registres Pour évaluer la réduction sur la pression des registres qu'apporterait l'introduction de registres architecturaux scalaires, nous utilisons l'algorithme d'allocation de registres au niveau PTX intégré dans Ocelot. Une première allocation est effectuée en considérant tous les registres, et une seconde en ignorant les registres qui ont été déterminés comme scalaires. Cette approche se justifie dans un contexte où seuls les registres vectoriels sont une ressource critique, ce que nous supposons être le cas sur les CPU avec extensions SIMD. Les résultats sont présentés sur la figure 5.18. Ainsi, l'application de la scalarisation statique permettrait une réduction du nombre de registres vectoriels de 15 à 50 %. Nous avons vu précédemment que le banc de registres vectoriels constituait un facteur limitant le passage à l'échelle des GPU. Il est donc important de réduire la pression sur les registres vectoriels.

L'allocation présentée ici étant effectuée au niveau du code PTX, les résultats ne correspondent en général pas au nombre de registres dans le code assembleur étant donné les différences au niveau du jeu d'instructions, des optimisations du compilateur et des algorithmes d'allocation de registres.

5.5 Conséquences sur l'architecture et la micro-architecture

Nous avons montré dans la section précédente qu'il était possible d'identifier les vecteurs uniformes et affines avec une précision acceptable, aussi bien de manière statique au cours de la compilation que de manière dynamique durant l'exécution. Chacune des alternatives aura des conséquences sur la manière de concevoir l'architecture des GPU.

5.5.1 Approche statique

Dans le cas où les vecteurs affines sont détectés de manière statique, le jeu d'instructions doit intégrer des registres et des opérations scalaires. Le modèle d'exécution le plus adapté dans ce contexte est une architecture séquentielle conventionnelle offrant des instructions SIMD, et qui manipule des vecteurs explicites et des scalaires. Larrabee est un exemple typique d'une telle architecture.

Avantages Les avantages de cette approche par rapport à la détection dynamique sont les suivants.

- Une allocation de registres optimisée.
  Le compilateur est capable de séparer les registres vectoriels des registres scalaires, et de les allouer dans des bancs de registres différents. Ainsi, la pression sur le banc de registres vectoriels est réduite. Obtenir les mêmes effets de manière dynamique nécessiterait des mécanismes de renommage complexes, qui annuleraient vraisemblablement les gains obtenus en termes d'énergie.
- L'application de la propagation de constantes sur les pas des vecteurs affines.
  Les résultats montrent que les pas utilisés sont typiquement des constantes connues dès la compilation. Ainsi, il devient inutile de les calculer lors de l'exécution, ni même de les mémoriser dans des registres.
- Moins de ressources matérielles dédiées au contrôle.
  Les architectures SIMT disposent de matériel détectant dynamiquement les accès mémoire à pas unitaires et uniformes, et les branchements uniformes. Si ces situations sont détectables à la compilation avec une précision suffisante, il est possible de se contenter d'une architecture SIMD conventionnelle. Nous pouvons supposer qu'un raisonnement analogue a guidé les choix effectués pour l'architecture Larrabee.
- Une représentation plus compacte en mémoire.
  Lorsqu'un registre doit être sauvegardé dans la pile, le compilateur est capable de déterminer si la place à réserver est celle d'un vecteur ou celle d'un scalaire. Cela promet une réduction drastique de l'empreinte sur le cache que nous avons abordée section 5.2.1.

5.5.2 Approche dynamique

L'approche opposée consiste à conserver un modèle SIMT pur, disposant d'un jeu d'instructions SPMD. Les architectures Tesla et Fermi illustrent ce modèle.

Avantages Par comparaison à l'approche statique décrite précédemment, la détection dynamique des vecteurs uniformes et affines apporteraient les avantages suivants.

- La capture de comportements dynamiques.
  L'analyse statique détecte uniquement les registres dont l'état reste uniforme ou affine tout au long de l'exécution du programme, sauf transformations profondes de code.
- Une précision accrue.
  Les dépendances envers le masque rendent l'analyse statique moins efficace dès lors que certains branchements ne sont pas identifiés comme uniformes. L'analyse dynamique dispose en revanche en permanence du masque d'exécution réel.
- Le maintien de la compatibilité.
  Les jeux d'instructions SIMT actuels peuvent être conservés tels quels. Cette approche évite aussi d'avoir à introduire des détails liés à la micro-architecture (notamment la gestion des branchements divergeants par prédication) dans le jeu d'instructions, alors qu'ils pourraient ne plus s'appliquer dans le futur.
- Un passage à l'échelle facilité face à la complexité logicielle.
  L'analyse statique nécessite de propager une connaissance globale sur chaque variable du programme. Cette information doit être propagée au travers des appels de fonctions et des frontières entre modules. Nous avons vu également qu'il était nécessaire de recourir à des mécanismes de compilation JIT pour exploiter des informations connues uniquement à l'exécution. L'approche dynamique permet de s'affranchir de toutes ces difficultés.

Coût en matériel. Quatre bits par registre sont nécessaire pour mémoriser les annotations, soit 2 Kbit de mémoire par SM du GT200. Par comparaison aux 512 Kbit du banc de registres correspondant, le surcoût en surface reste négligeable. L'annotation de la destination de chaque instruction peut être calculée par quelques opérations booléennes d'après les annotations de l'entrée.

En termes de latence, un niveau d'indirection est ajouté lors de la lecture des registres opérandes. Cependant, la lecture des étiquettes peut être effectuée en parallèle de la lecture du registre de drapeaux, dans l'étape *Contrôle* du pipeline décrit figure 5.19. Comme la lecture des étiquettes est effectuée un cycle avant la lecture des registres, l'écriture doit être avancée d'autant pour ne pas allonger le pipeline. Cela ne représente pas un problème étant donné que la latence du calcul du l'étiquette d'un résultat (traversée de quelques portes logiques) est toujours significativement plus faible que la latence d'exécution de l'instruction associée.

La possibilité de répartir une donnée scalaire entre toutes les voies d'un vecteur SIMD est d'ores et déjà présente dans l'architecture Tesla, pour les lectures en mémoire constante ou partagée, ainsi que pour la diffusion des constantes immédiates.

Bénéfices Connaître le type d'un registre (uniforme, affine, ou générique) à l'avance apporte plusieurs avantages.

D'une part, lorsque l'opérande source d'une instruction est identifié comme uniforme ou affine, il n'est pas nécessaire de lire l'ensemble du registre vectoriel. Il est possible de lire uniquement la première voie du registre, et de distribuer son contenu à toutes les unités de calcul SIMD. Nous rangeons la première voie des registres vectoriels dans un banc mémoire

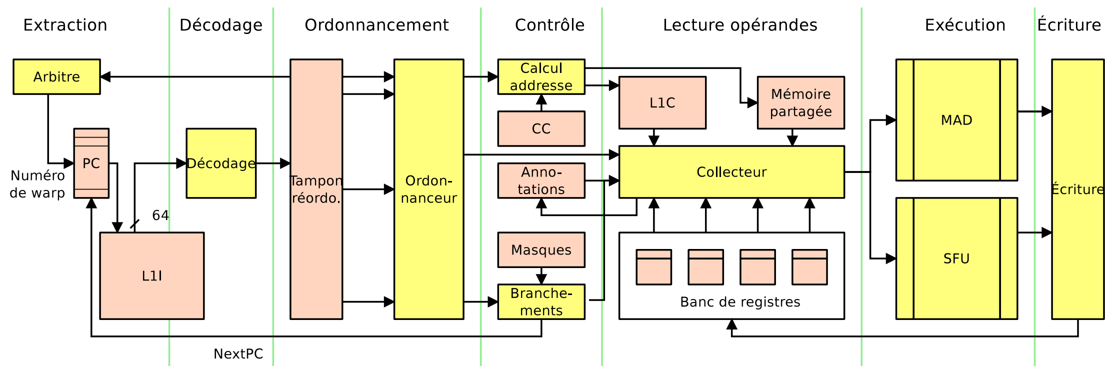


FIGURE 5.19 – Pipeline d'un SM Tesla modifié pour inclure la détection dynamique des registres uniformes et affines.

différent des autres voies. Ainsi, les bancs mémoire qui ne sont pas accédés peuvent bénéficier du *clock gating* pour permettre une réduction de la consommation. Cette situation représente 38 % des lectures d'opérandes dans les résultats de la section 5.4.5.

D'autre part, lorsque l'opération elle-même est uniforme ou affine, nous pouvons restreindre son exécution à un ou deux calculs scalaires. Les autres unités SIMD restent inactives, pour une réduction de consommation supplémentaire. Cette optimisation concerne 24 % des instructions exécutées.

Lorsqu'une instruction SIMT nécessite plusieurs cycles pour s'exécuter, comme c'est le cas sur l'ensemble des GPU actuels, il est aussi possible de gagner en débit. Lorsqu'une instruction est détectée comme uniforme ou affine, elle pourra être exécutée avec un débit d'une instruction par cycle, laissant les unités de calcul libres pour exécuter immédiatement d'autres instructions.

Enfin, les instructions de lecture et écriture mémoire peuvent profiter des annotations. Quand l'adresse est identifiée comme un vecteur affine, l'unité mémoire sait qu'elle a affaire à une lecture à pas constant. Par contraste, les architectures actuelles doivent analyser l'ensemble du vecteur d'adresses pour regrouper les requêtes qui peuvent l'être (section 2.5.1).

Amélioration possible Les changements proposés jusqu'ici s'appliquent uniquement au niveau micro-architectural. Le jeu d'instructions peut être conservé tel quel. C'est ce modèle que nous avons considéré pour obtenir les résultats de la section 5.4.5.

Cependant, il pourrait être utile de faire apparaître au niveau du langage machine des informations sur la durée de vie des registres. Par exemple, nous pouvons inclure dans chaque mot d'instruction des annotations signalant les opérandes source qui sont utilisés pour la dernière fois par cette instruction [Gle09]. Cela permettrait de résoudre le problème des écritures partielles décrit section 5.4.3, et garantirait à l'analyse dynamique d'être au moins aussi efficace que l'analyse statique.

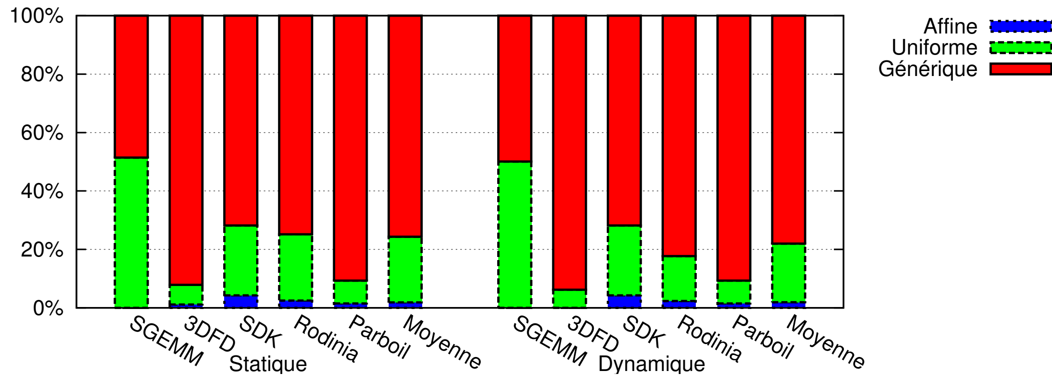


FIGURE 5.20 – Type des sources des instructions dont la destination est générique.

5.5.3 Conséquences sur les unités de calcul

Conversion d'affine vers générique. Alors que la conversion d'un vecteur uniforme en vecteur générique peut être prise en charge par le mécanisme de broadcast, la conversion d'un vecteur affine en vecteur générique réclame un traitement plus complexe. Il faut en effet calculer une multiplication-addition vectorielle supplémentaire. Les instructions nécessitant de telles conversions sont les instructions qui prennent en paramètre des vecteurs affines et renvoient un vecteur générique. Il s'agit par exemple des instructions prenant également des vecteurs génériques en entrée, ou des instructions prédiquées par un masque non-uniforme.

Si nous restreignons les valeurs du pas à des puissances de deux, la conversion d'affine vers générique peut être réalisé efficacement en matériel. Cependant, si ces conversions s'avèrent suffisamment rares, il sera tout aussi avantageux de les effectuer en utilisant les unités arithmétiques SIMD, puis de ré-exécuter l'instruction.

Intéressons-nous aux opérandes sources des instructions classées comme génériques par l'analyse statique. Nous cherchons à avoir une idée de la fréquence d'occurrence des conversions d'affine vers générique. La figure 5.20 présente les nombres d'occurrences relatifs de chaque type, au cours d'une exécution complète. Les instructions de lecture et écriture mémoire ne sont pas considérées, car elles bénéficient au contraire d'une adresse affine représenté sous forme compacte.

Notons que le nombre de conversions d'uniforme vers générique (broadcast) est significatif, avec une moyenne de 12 %. Ce n'est pas un problème étant donné le coût raisonnable d'un mécanisme de broadcast. La proportion de sources affines dans les instructions génériques est quant à lui inférieur à 1 %. Une implémentation entièrement logicielle est donc parfaitement envisageable.

Si l'on décide cependant d'ajouter du matériel spécifique, quels sont les cas qu'il devra traiter ? Analysons sur la figure 5.21 de quoi est composé le 1 % de sources affines que nous avons observé précédemment. Dans la majorité (81 %) des cas, le pas est une puissance de deux connue dès la compilation et la base est un multiple du pas. Une simple juxtaposition des bits de poids forts de la base et de la valeur décalée du numéro de voie permet de gérer ces cas. Remplacer la juxtaposition par une addition permet de traiter également les cas où le pas est

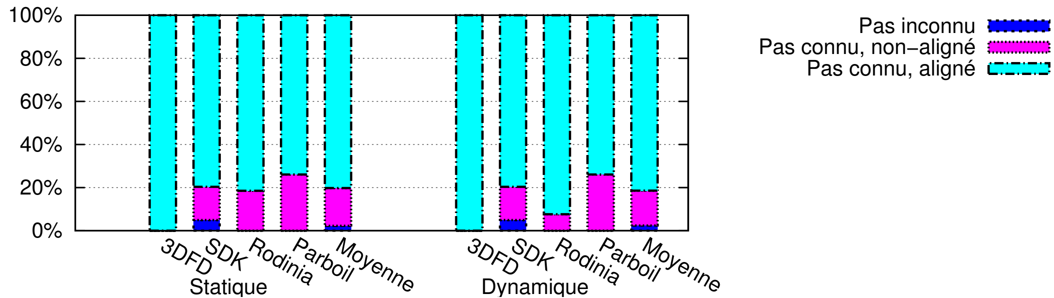


FIGURE 5.21 – Type des vecteurs impliqués dans des conversions d'affine vers générique.

connu mais n'est pas multiple de la base. Ces deux situations combinées représentent 98 % des occurences de conversions d'affine vers générique (qui elles-mêmes interviennent pour moins de 1 % des instructions). Le dernier cas est suffisamment rare pour qu'une implémentation logicielle par l'émission de quelques instructions supplémentaires se justifie naturellement.

En résumé, les conversions d'affine vers générique peuvent être réalisées par l'exécution d'instructions supplémentaires sans surcoût notable. Alternativement, la solution matérielle la plus simple permet de traiter la vaste majorité des cas se posant en pratique.

Unités 64 bits De nombreux facteurs tendent à rendre inévitable l'adoption d'un espace mémoire sur 64 bits par les GPU. On peut citer notamment l'arrivée de cartes GPU disposant de plus de 4 Go de mémoire, la possibilité d'accéder directement à la mémoire du système hôte depuis le GPU, l'unification des espaces d'adressage et l'utilisation du GPU en environnement multi-tâche. Ainsi, l'architecture Fermi dispose d'un espace d'adressage virtuel de 40 bits, et représente les pointeurs sur 64 bits. Cependant, les registres, chemins de données et unités de calcul entier restent sur 32 bits, comme sur Tesla. En effet, il serait très coûteux d'élargir l'ensemble des chemins de données des unités SIMD. Les calculs d'adresses sont donc réalisés par des instructions 32 bits sur des registres 32 bits, ce qui a un coût en performance et en occupation mémoire.

Nous avons vu que les adresses constituaient une source de vecteurs affines. Si la proportion de calculs sur 64 bits qui sont affines ou uniformes s'avère suffisante, nous pourrons envisager un banc de registres scalaires et une ALU scalaire sur 64 bits, en conjonction avec les unités SIMD actuelles sur 32 bits.

Quantifions ceci sur la figure 5.22, qui représente le classement des instructions sur 64 bits par l'analyse statique. En moyenne, 62 % des calculs 64 bits s'effectuent sur des vecteurs affines (incluant les vecteurs uniformes), ce qui est nettement supérieur à la proportion globale, tous types confondus (fig. 5.11). Sur des applications régulières comme SGEMM et 3DFD, l'ensemble des calculs d'adresse peuvent ainsi être « scalarisés ». Notons que l'application PNS de la suite Parboil utilise intensivement l'arithmétique entière 64 bits sur des données plutôt que des pointeurs, conduisant à une proportion de vecteurs affines nettement plus faible. Le langage PTX ne nous permet pas de distinguer les pointeurs des entiers sur 64 bits.

Un chemin de données scalaire 64 bits permettrait donc d'alléger significativement la charge sur les unités SIMD et bancs de registres vectoriels, et ceci à un coût raisonnable.

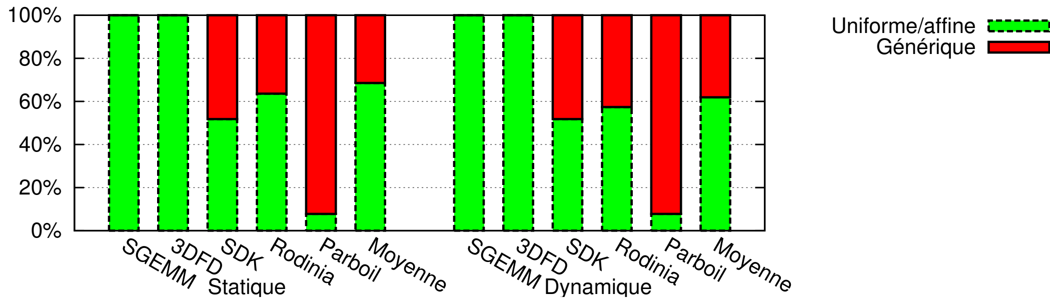


FIGURE 5.22 – Proportion de vecteurs affines identifiés parmi les opérandes destination de type entier 64 bits.

En résumé, la détection des vecteurs uniformes et affines de manière dynamique permet des réductions de consommation à un coût raisonnable, qui peut être contrebalancé par ailleurs par une réduction de la complexité de l'unité mémoire. Elle ne nécessite aucune modification du jeu d'instructions.

L'approche statique permet de décharger les unités SIMD des calculs scalaires, et simplifier l'architecture en identifiant la régularité mémoire et la régularité de contrôle dès la compilation.

5.6 Partage de tables pour l'évaluation de fonctions

Revenons maintenant sur l'unité généralisée d'évaluation de fonctions. Notre objectif est toujours de rendre plus souple ces unités en leur permettant d'évaluer des fonctions arbitraires en fonction des besoins du programmeur. Nous nous efforçons de maintenir les performances en termes de débit moyen rapporté à la surface et l'énergie consommée.

Nous avons observé dans la section 5.2 que l'exploitation de régularité séquentielle offrait des résultats mitigés. En effet, nous avons considéré uniquement la corrélation entre les accès successifs d'un même thread, et non entre threads différents. Plaçons-nous maintenant dans un cadre où les fonctions sont évaluées en parallèle par les threads d'un même warp. Nous exploiterons cette fois la régularité parallèle pour permettre le partage de ports vers la mémoire.

Chaque SM dispose déjà d'un cache de constantes qu'il serait avantageux de réutiliser pour mémoriser les coefficients. Malheureusement, cette mémoire n'est accessible que par un unique port de 32 bits (Tesla) ou 64 bits (Fermi), alors que les unités d'évaluation réclameraient 4 ports de 52 bits.

C'est ici que la régularité parallèle intervient. Les tables de coefficients étant adressées par les bits de poids fort de l'argument réduit, l'unité d'évaluation accède aux mêmes entrées dans la table lorsque les nombres en entrée sont relativement proches. De fait, si les données présentent de la régularité parallèle, il sera possible de partager un unique port entre toutes les voies SIMD, en contrôlant les accès par un arbitre. Ce système est similaire au mode de fonctionnement actuel du cache de constantes, et ne réclame pas de mécanisme matériel supplémentaire.

Plus précisément, dans l'architecture Tesla, les deux unités SFU exécutent une opération sur un vecteur de 16 composantes en 8 cycles à fréquence SP, ce qui correspond à 4 cycles à la

TABLE 5.3 – Conditions d'invocation des fonctions de base

Application	Fonction	Appel (`module:ligne`)	Appels SIMD	Adr. / requête
3DSMax	`inv_sqrtf`	`math/m_norm_tmp.h:71`	818866	1,00
	`inv_sqrtf`	`tnl/t_vb_lighttmp.h:386`	846128	1,18
	`sqrtf`	`tnl/t_vb_lighttmp.h:315`	1637732	1,95
Catia	`inv_sqrtf`	`main/light.c:1111`	1208	1,00
	`inv_sqrtf`	`main/light.c:1116`	1208	1,00
	`inv_sqrtf`	`math/m_norm_tmp.h:71`	2383	1,00
Maya	`sqrtf`	`main/light.c:1172`	1001	1,41
Pro/ ENGINEER	`exp2`	`main/light.c:961`	6120	14,09
	`inv_sqrtf`	`main/light.c:1111`	1644	1,00
	`inv_sqrtf`	`main/light.c:1116`	1644	1,00
	`log2`	`main/light.c:961`	6120	11,83
	`sqrtf`	`main/light.c:1172`	3451	1,03
TCVis	`inv_sqrtf`	`main/light.c:1111`	32194	1,00
	`inv_sqrtf`	`main/light.c:1116`	32194	3,00
	`inv_sqrtf`	`main/light.c:1170`	3151	1,00
Unigraphics	`sqrtf`	`swrast/s_aalinetemp.h:137`	741140	12,89
GPU4RE	`exp`	`soft_comp.cpp:170`	6229	5,11
	`exp`	`soft_comp.cpp:171`	6229	1,00
Comsol	`exp`		516652	1,87

fréquence SM. Il sera donc possible d'effectuer 4 accès au cache de constantes pendant cette période. Même si le cache n'est doté que d'un port de 32 bits, il est possible de récupérer 2 jeux de coefficients sans que le débit ne soit affecté.

Régularité parallèle Reprenons les mêmes applications que celles que nous avons étudiées dans la section 5.2.

La table 5.3 quantifie la régularité parallèle des évaluations de fonctions. La première colonne de résultats présente le nombre d'évaluations de fonctions sur des vecteurs à 16 composantes. La dernière colonne indique le nombre d'adresses distinctes rencontrées au cours de l'évaluation. Il peut varier de 1 (partage maximal) à 16 (aucun partage). Ces résultats sont donnés en fonction du module et numéro de ligne de l'appel. Nous ne considérons que les fonctions appelées au moins 1 000 fois au cours d'une exécution.

Une analyse plus précise du code montre que :

– Dans TCVis, l'opération effectuée consiste à calculer l'éclairement d'un sommet en

fonction de trois sources de lumière. Alors que la bibliothèque Mesa calcule successivement les contributions de chaque source pour chaque sommet, un GPU traiterait 16 sommets simultanément. Notre protocole de test ne nous permet pas de reconstituer cet ordre d'évaluation, conduisant au résultat de 3 accès concurrents en moyenne au lieu du résultat de 1 attendu.

– Pour les lignes du tableau où le nombre d'accès est exactement égal à 1, le code calcule toujours la même valeur. Ce calcul redondant est dû au fait que le programme appelle des routines OpenGL conçues pour un modèle d'éclairement général dont le logiciel considère un cas particulier.
– L'appel à la fonction pow dans `main/light.c:961` dans Pro/ENGINEER, est décomposé en $\log_2$ et $\exp_2$ et crée très peu de localité. En fait, l'application remplit une table pour la fonction pow destinée à accélérer les calculs d'éclairement. Les opérateurs matériels d'évaluation des fonctions $\log_2$ et $\exp_2$ du GPU rendent ces calculs moins coûteux que des accès à la mémoire. On peut donc supposer que cette méthode ne serait pas utilisée lors d'une exécution sur GPU.

Sur l'ensemble des programmes testés, nous observons qu'une requête ne concerne en moyenne que 3,3 adresses différentes pour des tables composées de 64 entrées. Si nous éliminons les cas que nous venons d'aborder, cette moyenne retombe à 1,9. Ainsi, l'utilisation du cache de constantes en remplacement des ROM est envisageable, et n'aurait pas d'influence sur la performance de la plupart des applications étudiées.

En cas de conflit ou d'échec dans le cache, l'instruction fautive sera exécutée à nouveau par le mécanisme de ré-exécution existant. Élargir le port du cache de constantes à 64 bits ou considérer une granularité de vecteur plus large réduirait le risque de conflit.

Ainsi, nous éliminons le coût en surface des tables en ROM. D'autre part, les tables deviennent modifiables, permettant l'évaluation efficace de fonctions arbitraires. L'impact sur le débit et la consommation reste limité grâce à l'exploitation de la régularité parallèle sur les données.

Les propositions architecturales présentées ici ont fait l'objet de publications [CDDO08, CDDP09, CDZ09].

Conclusion

Nous avons vu au cours des chapitres 1 et 2 que les GPU exploitaient non seulement le parallélisme de données présent dans les applications, leurs déplacements de données limités, mais aussi leur régularité. De nombreux mécanismes statiques et dynamiques permettent de tirer parti de ces trois caractéristiques pour maximiser la performance par millimètre carré et la performance par watt de ces architectures. Ils reposent en particulier sur du multithreading massif à grain fin avec agrégation des threads pour simplifier le contrôle, et visent à maximiser le débit au prix d'une hausse des latences.

Nous avons suivi l'évolution accélérée des unités de calcul en virgule flottante dans le chapitre 3. Cette étude portait autant sur les unités génériques que sur les unités spécialisées pour le rendu graphique. Nous avons montré que ces dernières pouvaient être détournées de leur usage premier pour contribuer à des calculs généralistes, éventuellement aux prix d'une efficacité moindre. En particulier, les unités matérielles pipelinées d'évaluation des fonctions élémentaires et des divisions et racines carrées apportent un gain de performances significatif sur des applications de simulation physique.

Ces résultats font apparaître la nécessité de parvenir à un bon compromis entre la généricité et la spécialisation des unités de calcul lors de leur conception. En effet, les changements technologiques (impact de la consommation) et applicatifs (marché du jeu vidéo et du multimédia) qui sont intervenus au cours des dernières années réclament de revoir l'architecture des unités de calcul, voire les opérateurs de base.

Tirer pleinement parti de telles unités spécialisées nécessite d'analyser au préalable les impératifs de précision de chaque application. En effet, une unité optimisée spécifiquement en fonction de la précision nécessaire apporte des gains substantiel par rapport à la reproduction naïve en matériel d'un algorithme opérant en virgule flottante IEEE 754 [Obe06, dDKP09].

Nous avons présenté un simulateur de l'architecture GPU Tesla au chapitre 4. L'efficacité de la simulation est assurée par les mêmes facteurs que ceux qui permettent le passage à l'échelle des architectures GPU : le parallélisme de données et la régularité. Ce simulateur constitue un outil pouvant être utilisé par la communauté pour étudier des applications sur GPU, explorer l'espace micro-architectural des architectures parallèles à grain fin et expérimenter de nouvelles fonctionnalités.

Nous nous sommes ensuite intéressés plus particulièrement à la régularité, en comparant les notions de régularité séquentielle et de régularité parallèle dans le chapitre 5. Nous avons présenté plusieurs techniques permettant d'exploiter des cas particuliers de régularité parallèle. Nous avons mis au point un mécanisme de détection des points d'indépendance de contrôle entièrement dynamique et peu coûteux. Il permet d'exécuter en mode SIMD des programmes

binaires existants, ce qui constitue un pas supplémentaire vers la convergence entre CPU et GPU.

Nous avons montré que les vecteurs affines présentaient un enjeu majeur pour permettre l'identification de situations de régularité parallèle, telles que la régularité mémoire, la régularité de contrôle et la régularité de données. Ces travaux peuvent être appliqués dès aujourd'hui pour compiler des noyaux SPMD pour les jeux d'instructions de CPU avec extensions SIMD. Ils sont complémentaires avec les techniques de fusion de threads qui ciblent les architectures multi-cœurs [DKYC10, SGM$^+$10].

Au-delà de ces applications, les résultats obtenus suggèrent d'étendre les mécanismes SIMT d'agrégation de threads des GPU pour factoriser les calculs redondants et les registres dupliqués entre threads. Ainsi, nous pouvons améliorer l'efficacité des architectures parallèles à grain fin sans compromettre la simplicité du modèle de programmation.

La scalarisation peut être réalisée de manière statique ou dynamique, et reste compatible avec les optimisations de compilation et les mécanismes d'exécution existants. Son emploi est également complémentaire avec celui d'approches logicielles pour réduire la redondance [Vol10b].

Directions futures

Nous avons étudié dans cette thèse plusieurs formes de régularité parallèle : régularité de contrôle et d'instructions, régularité sur les adresses, et régularité sur les données virgule flottantes. Nous avons également abordé différents moyens d'exploiter ces régularités. Il est possible d'aller plus loin en généralisant ces mécanismes.

Cache de données affines Les annotations uniformes et affines que nous proposons dans la section 5.4 s'appliquent uniquement aux registres. Lorsque les variables sont sauvegardées en mémoire puis restaurées, ces annotations sont perdues. Or, l'évolution des architectures GPU semble aller dans le sens d'un déplacement progressif des variables locales depuis le banc de registres vers le cache L1. Le cache est utilisé pour contenir les piles d'appels des threads, qui sont nécessaires pour permettre la liaison dynamique entre modules binaires et la récursivité. Cependant, nous avons vu section 5.2.1 que les limites de capacité sont rapidement atteintes lorsque l'on sauvegarde des registres vectoriels dans le cache.

Comme un nombre significatif de tels registres sont uniformes ou affines, il serait avantageux d'étendre le mécanisme d'annotation au cache L1. Contrairement aux registres dont l'allocation est fixée à la compilation, un cache offre la possibilité d'une allocation flexible entre les données vectorielles et scalaires. Ainsi, on peut concevoir un cache scalaire en complément du cache vectoriel, à la manière des caches augmentés par du contenu nul [DPS09].

Pour les zones de mémoire locales telles que la pile d'appel, les données des différents threads d'un warp résidant à la même adresse sont rangées de manière quasiment contiguë. En particulier, les registres vectoriels sauvegardés dans la pile coïncident exactement avec les lignes du cache. Si le registre sauvegardé est uniforme ou affine, la ligne de cache associée peut être conservée séparément du cache usuel, de manière bien plus compacte.

Compression intra-vecteur La solution présentée ci-dessus peut être vue comme une technique de compression très spécialisée des lignes de cache. Nous pouvons la généraliser. Les vecteurs uniformes et affines sont principalement restreints aux adresses mémoire et aux variables de contrôle. Cependant, les résultats obtenus sur les données virgule flottante dans la section 5.6 suggèrent que les données manipulées par les calculs présentent également de la régularité parallèle. Celle-ci peut être exploitée par un mécanisme de compression de la mémoire. La compression vise à réduire l'empreinte sur les caches, mais aussi et surtout à limiter la quantité de transferts vers et depuis la mémoire externe, qui sont particulièrement coûteux en énergie.

De tels systèmes de compression au vol ont déjà été étudiés pour des architectures séquentielles et multi-cœurs [Ala06]. Néanmoins, ces propositions se heurtent à deux obstacles. D'une part, le mécanisme de compression induit une augmentation de la latence mémoire, qui est difficilement acceptable sur les architectures séquentielles. D'autre part, les données rangées ensemble dans une ligne de cache sont peu corrélées, ce qui oblige le système de compression à considérer chaque valeur indépendamment du contexte, ou au contraire travailler à une granularité élevée.

Dans le cadre des architectures parallèles de type GPU, ces deux problèmes sont nettement moins présents. L'architecture est conçue dès l'origine pour être en mesure de tolérer des latences mémoires élevées. L'architecture, le modèle de programmation, les bibliothèques et l'ensemble des environnements de programmation pour GPU favorisent également les structures de données de type structure de tableaux (SoA), qui groupent ensemble les données de même nature.

Nous pouvons donc envisager des mécanismes matériels de compression des données transitant par les caches et les bus mémoire dans le cadre des architectures parallèles à grain fin.

Matériel reconfigurable Nous avons constaté l'intérêt des unités spécialisées au cours du chapitre 3. Elle permettent de s'affranchir de la majeure partie du surcoût de contrôle auxquelles sont soumises les architectures programmables : lecture et décodage des instructions, mémorisation des données temporaires.

Cependant, il est difficile voire impossible de prévoir à l'avance tous les cas d'utilisation possibles de ces unités. Pour permettre de la souplesse dans leur utilisation, les unités dédiés se doivent d'être configurables. Nous avons considéré plusieurs pistes pour améliorer la généricité des unités spécialisés au cours du chapitre 3.

Poursuivre dans cette voie ferait converger ces unités vers des architectures reconfigurables, à l'image des FPGA [Mar99]. Plusieurs degrés de granularité sont possible. Les FPGA permettent traditionnellement une configuration au niveau du bit, mais tendent de plus en plus à embarquer des composants plus gros tels que des mémoires, des multiplieurs ou blocs DSP, voire des microprocesseurs complets.

La possibilité de reconfiguration présente un coût significatif en termes de surface. Cependant, en termes de consommation dynamique, le surcoût est raisonnable par comparaison aux processeurs généralistes. À mesure que le fossé entre la densité des semiconducteurs et la consommation énergétique s'accroît, les architectures reconfigurables deviennent plus intéressantes.

L'enjeu majeur de ces architectures porte sur les modèles de programmation et les outils de compilation. En effet, les environnements de développements pour FPGA sont traditionellement basés sur des outils de synthèse pour le matériel, qui ne sont pas familiers aux programmeurs.

Des langages de plus haut niveau promettant de rendre la programmation plus aisée commencent néanmoins à émerger [Men, Alt06]. Cependant, ces technologies restent propriétaires et cantonnées à des marchés de niche. Le calcul sur GPU doit son succès en grande partie à la simplicité de son modèle de programmation SPMD, issu des langages de *shaders* graphiques. Pour que les architectures reconfigurables puissent se démocratiser, il sera nécessaire de trouver un équivalent au modèle SPMD qui soit en mesure d'offrir les mêmes avantages.

Ainsi, les problématiques que nous avons abordées au cours de cette thèse recouvrent de nombreux domaines. Nous avons en particulier considéré des questions liées à la programmation parallèle, la programmation graphique, l'arithmétique des ordinateurs, l'architecture des processeurs et la compilation. En effet, seule une approche conjointe de l'ensemble des composants logiciels et matériels permettra de mettre en place des environnements d'exécution parallèles capable de faire face aux enjeux actuels et futurs.

Bibliographie

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research : A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[Abr09] Michael Abrash. Rasterization on Larrabee : A first look at the Larrabee New Instructions (LRBni) in action, March 2009.

[ACD10] Mark Arnold, Caroline Collange, and David Defour. Implementing LNS using filtering units of GPUs. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2010.

[ACG+07] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM : An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2) :45–48, 2007.

[AL+06] Alfred V. Aho, Monica S. Lam, , Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[Ala06] Alaa R. Alameldeen. *Using compression to improve chip multiprocessor performance*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 2006.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar : An infrastructure for computer system modeling. *Computer*, 35(2) :59–67, 2002.

[Alt06] Altera. Automated generation of hardware accelerators with direct memory access from ANSI/ISO Standard C functions. White Paper, May 2006.

[AMD09a] AMD. *ATI Stream Computing – Technical Overview*, March 2009.

[AMD09b] AMD. *Evergreen Family Instruction Set Architecture : Instructions and Microcode*, December 2009.

[AMD09c] AMD. *R700-Family Instruction Set Architecture*, March 2009.

[AMD09d] AMD. *Radeon R6xx/R7xx Acceleration*, 2009.

[ARB] ARB_fragment_program. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`.

[Asa98] Krste Asanović. *Vector microprocessors*. PhD thesis, University of California, Berkeley, 1998.

[Bad10] Badaboom and the new NVIDIA GF100 cards, 2010. `http://www.badaboomit.com/node/508`.

[BC07] Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial L-∞ approximations. In *18th IEEE Symposium on Computer Arithmetic ARITH '07*, pages 169–176, June 2007.

[BDH+06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator : Modeling networked systems. *IEEE Micro*, 26(4) :52–60, 2006.

[BDL09] Sylvie Boldo, Marc Daumas, and Ren-Cang Li. Formally verified argument reduction with a fused multiply-add. *IEEE Transactions on Computers*, 58 :1139–1145, 2009.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs : stream computing on graphics hardware. In *SIGGRAPH '04 : ACM SIGGRAPH 2004*, pages 777–786, 2004.

[Bly06] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3) :724–734, 2006.

[BMP06] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the Boost interval arithmetic library. *Theor. Comput. Sci.*, 351(1) :111–118, 2006.

[BYF+09] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, Boston, April 2009.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia : A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0 :44–54, 2009.

[CDD07] Caroline Collange, Marc Daumas, and David Defour. Graphic processors to speed-up simulations for the design of high performance solar receptors. In *IEEE 18th International Conference Application-specific Systems, Architectures and Processors*, pages 377–382, Montréal Canada, 2007. IEEE.

[CDD08a] Caroline Collange, Marc Daumas, and David Defour. Line-by-line spectroscopic simulations on graphics processing units. *Computer Physics Communications*, 178 :135–143, 2008.

[CDD08b] Caroline Collange, Marc Daumas, and David Defour. État de l'intégration de la virgule flottante dans les processeurs graphiques. *Technique et science informatiques*, 27/6 :719–733, 2008.

[CdDD06] Caroline Collange, Florent de Dinechin, and Jérémie Detrey. Floating point or LNS : Choosing the right arithmetic on an application basis. In *EuroMicro Digital System Design DSD*, pages 197–203, 2006.

[CDDO08] Caroline Collange, Marc Daumas, David Defour, and Regis Olivès. Fonctions élémentaires sur GPU exploitant la localité de valeurs. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, February 2008.

[CDDP09] Caroline Collange, Marc Daumas, David Defour, and David Parello. Étude comparée et simulation d'algorithmes de branchements pour le GPGPU. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.

[CDDP10] Caroline Collange, Marc Daumas, David Defour, and David Parello. Barra : a parallel functional simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.

[CDP09] Caroline Collange, David Defour, and David Parello. Barra, a modular functio-nal GPU simulator for GPGPU. Technical Report hal-00359342, HAL-CCSD, January 2009.

[CDT09] Caroline Collange, David Defour, and Arnaud Tisserand. Power consumption of GPUs from a software perspective. In *ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 922–931. Springer, 2009.

[CDZ09] C. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume LNCS 6043, pages 46–55, 2009.

[CFD08] C. Collange, J. Flòrez, and D. Defour. A GPU interval library based on Boost.Interval. In *Real Numbers and Computers*, pages 61–72, jul 2008.

[CHMS00] Ole Capriani, Lars Hvidegaard, Mikkel Mortensen, and Thomas Schneider. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 1(6) :9–21, 2000.

[CL08] Brett W. Coon and John Erik Lindholm. System and method for managing divergent threads in a SIMD architecture. US Patent 7353369, April 2008.

[CMOS08] Brett W. Coon, Peter C. Mills, Stuart F. Oberman, and Ming Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators. US Patent 7434032, October 2008.

[Coe95] Tim Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal*, 20(4) :129–135, 148, 1995.

[Col10] Caroline Collange. Analyse de l'architecture GPU Tesla. Technical Report hal-00443875, HAL-CCSD, Jan 2010.

[Cor97] Henk Corporaal. *Microprocessor Architectures : From VLIW to TTA*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[CTW04] Jamison D. Collins, Dean M. Tullsen, and Hong Wang. Control flow optimization via dynamic reconvergence prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2004.

[Cud] NVIDIA CUDA Zone. `http://www.nvidia.com/cuda`.

[DD06] Guillaume Da Graça and David Defour. Implementation of float-float operators on graphics hardware. Technical report, HAL-CCSD, 2006.

[dDKP09] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. Generating high-performance custom floating-point pipelines. In *Field Programmable Logic and Applications*. IEEE, August 2009.

[Dek71] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3) :224–242, 1971.

[DKYC10] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot : A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[DPS09] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *ICS'09 : Proceedings of the 23rd International Conference on Supercomputing*, pages 46–55, 2009.

[Eck08] Roger E. Eckert. Page stream sorter for DRAM systems. US Patent 7376803, May 2008.

[Env] Envytools. `https://github.com/pathscale/envytools`.

[ERB+95] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, J. Benschneider, Debra Bernstein, Ruben W. Castelino, M. Cooper, Daniel E. Dever, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Rasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7 :119–135, 1995.

[Exo09] Albert Claret Exojo. Design and implementation of a PTX emulation library. Master's thesis, Universitat Politècnica de Catalunya, 2009.

[FGK+02] Eric S. Fetzer, Mark Gibson, Anthony Klein, Naomi Calick, Chengyu Zhu, Eric Busta, and Baker Mohammad. A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11) :1433–1440, Nov 2002.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21 :948+, 1972.

[For09] Tom Forsyth. SIMD programming on Larrabee : A second look at the Larrabee New Instructions (LRBni) in action, March 2009.

[FSSV06] Jorge Flórez, Mateu Sbert, Miguel A. Sainz, and Josep Vehí. Improving the interval ray tracing of implicit surfaces. *Lecture Notes in Computer Science*, 4035 :655–664, 2006.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10) :30–53, 1990.

[Fun08] Wilson Wai Lun Fung. Dynamic warp formation : exploiting thread scheduling for efficient MIMD control flow on SIMD graphics hardware. Master's thesis, University of British Columbia, 2008.

[GG03] Stéphane Guindon and Olivier Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic biology*, 52(5) :696–704, 2003.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2) :10–24, 2006.

[Gle09] Andy Glew. Coherent vector lane threading. *Berkeley ParLab Seminar*, 2009.

[GLGN+08] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4) :13–27, 2008.

[GM74] W. Morven Gentleman and Scott B. Marovitch. More on algorithms that reveal properties of floating point arithmetic units. *Communications of the ACM*, 17(5), 1974.

[Gö10] Dominik Göddeke. *Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters*. PhD thesis, Technische Universität Dortmund, 2010.

[Han79] Eldon R. Hansen. Global optimization using interval analysis : The one-dimensional case. *Journal of Optimization Theory and Applications*, 29 :331–344, 1979.

[HG83] E. R. Hansen and R. I. Greenberg. An interval Newton method. *Applied Mathematics and Computation*, 12(2-3) :89 – 98, 1983.

[Hig02] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. Second edition.

[Hin10] Glenn Hinton. Key Nehalem choices. Computer Systems Colloquium (EE380), Stanford University Department of Electrical Engineering, February 2010.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3) :280–289, 2010.

[HL04] Karl Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, page C8, August 2004.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *International Symposium on High-Performance Computer Architecture*, pages 258–262, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[HP/98] HP/Compaq. *Alpha Architecture Handbook*, 1998.

[HPAD07] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau. *Computer architecture : a quantitative approach*. Morgan Kaufmann, 2007.

[HVD09] Brian D. Hutsell and James M. Van Dyke. Efficiency based arbiter. US Patent 7603503, October 2009.

[IBM09] IBM. *Power ISA*, 2009.

[IEE85] IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[IEE08] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, 2008.

[Int06] Intel. *Intel Itanium Architecture : Software Developer's Manual*, 2006.

[Int09a] Intel. *Intel Advanced Vector Extensions Programming Reference*, 2009.

[Int09b] Intel. *Intel G45 Express Chipset Graphics Controller PRM, Volume Four : Subsystem and Cores*, February 2009.

[Int10a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manuals Volume 1 : Basic Architecture*, 2010.

[Int10b] Intel. Intel unveils new product plans for high-performance computing. Intel news release, May 2010.

[Kan08] David Kanter. NVIDIA's GT200 : Inside a parallel processor. Technical report, Real World Technologies, 2008.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara : A 32-Way multithreaded Sparc processor. *IEEE Micro*, 25 :21–29, 2005.

[Kar85] Richard Karpinski. PARANOIA : a floating-point benchmark. *Byte*, 10(2) :223–235, 1985.

[KB89] Devendra Kalra and Alan H. Barr. Guaranteed ray intersection with implicit surfaces. *Computer Graphics (Siggraph proceedings)*, 23 :297–206, 1989.

[KBH+04] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The Vector-Thread architecture. *IEEE MICRO*, 24(6) :84–90, 2004.

[KDDH94] Ralph Baker Kearfott, M. Dawande, K. S. Du, and C. Y. Hu. Algorithm 737 : INTLIB : a portable Fortran 77 interval standard function. *ACM Transactions on Mathematical Software*, 20(4) :447–459, 1994.

[KDR+02] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *ICCD'2002 : IEEE International Conference on Computer Design*, pages 282–288, 2002.

[KDY09] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of GPGPU kernels. Technical Report GIT-CERCS-09-06, Georgia Institute of Technology, 2009.

[Ker92] Ronan Keryell. *POMP : d'un Petit Ordinateur Massivement Parallèle SIMD à Base de Processeurs RISC — Concepts, Étude et Réalisation*. PhD thesis, Laboratoire d'informatique de l'École normale supérieure — Université Paris XI, 1992.

[KHH+07] Aaron Knoll, Younis Hijazi, Charles D. Hansen, Ingo Wald, and Hans Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[KK96] R. Baker Kearfott and Vladik Kreinovich. *Applications of Interval Computations*. Kluwer Academic Publishers, 1996.

[Knu97] Donald E. Knuth. *The Art of Computer Programming : Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[KP93] Ronan Keryell and Nicolas Paris. Activity counter : New optimization for the dynamic scheduling of SIMD control flow. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 184–187, 1993.

[LA04] Chris Lattner and Vikram Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *CGO '04 : Proceedings of the international symposium on Code generation and optimization*, page 75, 2004.

[Lam08] Branimir Lambov. Interval arithmetic using SSE-2. In *Reliable Implementation of Real Number Algorithms : Theory and Practice*, volume 5045 of *Lecture Notes in Computer Science*, pages 102–113. Springer Berlin / Heidelberg, 2008.

[Leu02] Rainer Leupers. Compiler design issues for embedded processors. *IEEE Design & Test*, 19(4) :51–58, 2002.

[Lit61] John D. C. Little. A proof for the queuing formula : $L = \lambda W$. *Operations Research*, 9(3) :383–387, 1961.

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth : an evaluation of throughput computing on CPU and GPU. In *ISCA '10 : Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, 2010.

[LNMC08] John Erik Lindholm, John R. Nickolls, Simon S. Moy, and Brett W. Coon. Register based queuing for texture requests. US Patent 7456835, November 2008.

[LNOM08] John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla : A unified graphics and computing architecture. *IEEE Micro*, 28(2) :39–55, 2008.

[LO09] John Erik Lindholm and Stuart F. Oberman. Execution of parallel groups of threads with per-instruction serialization. US Patent 7634637, December 2009.

[LP84] Adam Levinthal and Thomas Porter. Chap - a SIMD graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, 1984.

[LSM+08] John Erik Lindholm, Ming Y. Siu, Simon S. Moy, Samuel Liu, and John R. Nickolls. Simulating multiported memories using lower port count memories. US Patent 7339592 B2, March 2008.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. *SIGOPS Oper. Syst. Rev.*, 30(5) :138–147, 1996.

[Mar99] Pierre Marchal. Field-programmable gate arrays. *Commun. ACM*, 42 :57–59, April 1999.

[Mar00] Peter Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN : 0130183482.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics : A full system simulation platform. *Computer*, 35(2) :50–58, 2002.

[Men] Mentor Graphics Catapult C. `http://www.mentor.com/esl/catapult`.

[Mer10] Rick Merritt. China taps Nvidia for world's second biggest computer. *EE Times (www.eetimes.com)*, May 2010.

[MGM09] John S. Montrym, David B. Glasco, and Steven E. Molnar. Apparatus, system, and method for using page table entries in a graphics system to provide storage format information for address translation. US Patent 7545382, June 2009.

[MGR+05] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. Shader performance analysis on a modern GPU architecture. In *MICRO 38 : Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.

[MHR90] Robert K. Montoye, E. Hokenek, and Steve L. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM J. Res. Dev.*, 34(1) :59–70, 1990.

[Mica] Microsoft DirectX. `http://msdn.microsoft.com/directx/`.

[Micb] Microsoft HLSL. `http://msdn.microsoft.com/en-us/library/bb509561.aspx`.

[Micc] Direct3D 10.1 Features. `http://msdn.microsoft.com/en-us/library/bb694530.aspx`.

[Mic09] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2 : Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009.

[Mit90] Don Mitchell. Robust ray intersection with interval arithmetic. *Proceedings on Graphics interface '90*, pages 68–74, 1990.

[MJO+05] Silvia M. Mueller, Christian Jacobi, Hwa-Joon Oh, Kevin D. Tran, Scott R. Cottier, Brad W. Michael, Hiroo Nishikawa, Yonetaro Totsuka, Tatsuya Namatame, Naoka Yano, Takashi Machida, and Sang H. Dhong. The vector floating-point unit in a synergistic processor element of a CELL processor. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 59–67, June 2005.

[MLC+08] Peter C. Mills, John Erik Lindholm, Brett W. Coon, Gary M. Tarolli, and John Matthew Burgess. Scheduling instructions from multi-thread instruction buffer based on phase boundary qualifying rule for phases of math and data access operations with better caching. US Patent 7366878, April 2008.

[MM05] John Montrym and Henry Moreton. The GeForce 6800. *IEEE Micro*, 25(2) :41–51, 2005.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33 :2005, 2005.

[MT98] Junichiro Makino and Makoto Taiji. *Scientific Simulations with Special-Purpose Computers : The GRAPE Systems*. John Wiley & Son Ltd., 1998.

[MTS10] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3) :235–246, 2010.

[Mul05] Jean-Michel Muller. On the definition of ulp(x). Technical Report RR-5504, INRIA, February 2005.

[Mun09] Aaftab Munshi. The OpenCL specification. *Khronos OpenCL Working Group*, 2009.

[MWHL06] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of GPUs using the RapidMind development platform. In *SC'06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, 2006.

[NNB09] Bryon S. Nordquist, John R. Nickolls, and Luis I. Bacayo. Parallel data processing systems and methods using cooperative thread arrays and SIMD instruction issue. US Patent 7584342, September 2009.

[NNHM09] Lars Nyland, John R. Nickolls, Gentaro Hirota, and Tanmoy Mandal. Systems and methods for coalescing memory accesses of parallel threads. US Patent Application 2009/0240895, September 2009.

[Nou] Nouveau. `http://nouveau.freedesktop.org`.

[NVI] Nsight. `http://developer.nvidia.com/nsight`.

[NVI09a] NVIDIA. *CUDA-GDB : The NVIDIA CUDA Debugger, Version 2.2*, 2009.

[NVI09b] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture : Fermi*, 2009.

[NVI10a] NVIDIA. *NVIDIA CUDA Best Practices Guide*, 2010.

[NVI10b] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.1*, 2010.

[NVI10c] NVIDIA. *NVIDIA Tegra Multi-Processor Architecture Whitepaper*, 2010.

[NVI10d] NVIDIA. *PTX : Parallel Thread Execution ISA Version 2.1*, 2010.

[Obe06] Stuart Oberman. GPUs : Applications of computer arithmetic in 3D graphics. *7th Conference on Real Numbers and Computers (RNC 7)*, 2006.

[OFW99] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow ! technology : architecture and implementations. *IEEE Micro*, 19(2) :37–48, 1999.

[OGL] OpenGL. `http://www.opengl.org/`.

[OLG^{+}07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, March 2007.

[OS05] Stuart F. Oberman and Michael Siu. A high-performance area-efficient multifunction interpolator. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 272–279, July 2005.

[OST09] Stuart Oberman, Ming Y. Siu, and David C. Tannenbaum. Fused multiply-add functional unit. US Patent Application 2009/0150654, June 2009.

[Pat04] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10) :71–75, 2004.

[PBG09] David Parello, Mourad Bouache, and Bernard Goossens. Improving cycle-level modular simulation by vectorization. In *Rapid Simulation and Performance Evaluation : Methods and Tools (RAPIDO'09)*, pages 63–68, 2009.

[Pfi08] Greg Pfister. Larrabee vs. Nvidia, MIMD vs. SIMD. *The Perils of Parallel*, September 2008.

[Pha05] Matt Pharr, editor. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[Pri92] Douglas M. Priest. *On properties of floating point arithmetics : numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, Berkeley, California, 1992.

[Psc] PSCNV. `http://github.com/pathscale/pscnv`.

[PSG06] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *SIGGRAPH '06 : ACM SIGGRAPH 2006 Sketches*, page 184, 2006.

[PV93] Michèle Pichat and Jean Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Editions Technip, 1993.

[R^{+}98] L.S. Rothman et al. The HITRAN molecular spectroscopic database and HAWKS (hitran atmospheric workstation) : 1996 edition. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 60(5) :665–710, 1998.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1) :78–103, 1997.

[RDK^{+}00] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 375–386, 2000.

[Rev01] Nathalie Revol. *L'arithmétique des ordinateurs*, volume 13 of *Réseaux et systèmes répartis*, chapter Arithmétique par intervalles, pages 387–426. Hermes, 2001.

[Ric93] Stephen E. Richardson. Exploiting trivial and redundant computation. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[RM09] Greg Ruetsch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. NVIDIA CUDA SDK Application Note, 2009.

[RR05] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4) :275–290, 2005.

[Rub08] Norman Rubin. Issues and challenges in compiling for graphics processors, April 2008.

[Rus10] Zack Rusin. The software renderer, 2010. `http://zrusin.blogspot.com/2010/03/software-renderer.html`.

[RWG+95] L. S. Rothman, R. B. Wattson, R. Gamache, J. W. Schroeder, and A. McCann. HITRAN HAWKS and HITEMP : high-temperature molecular database. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, volume 2471, pages 105–111, June 1995.

[RZBM09] Siegfried Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49(2) :419–431, 2009.

[Sch81] N. L. Schryer. A test of computer's floating-point arithmetic unit. Technical report 89, AT&T Bell Laboratories, 1981.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3) :1–15, 2008.

[SD08] Gunar Schirner and Rainer Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *Trans. on Embedded Computing Sys.*, 8(1) :1–29, 2008.

[SGM+10] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO '10 : Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, 2010.

[SL93] André Seznec and Jacques Lenfant. Odd memory systems may be quite interesting. *SIGARCH Comput. Archit. News*, 21(2) :341–350, 1993.

[SLS04] Jeremy W. Sheaffer, David Luebke, and Kevin Skadron. A flexible simulation framework for graphics architectures. In *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 85–94, New York, NY, USA, 2004. ACM.

[SO08] Ming Y. Siu and Stuart F. Oberman. Multipurpose functional unit with multiply-add and format conversion pipeline. US Patent 7428566, September 2008.

[SPEa] SPEC Benchmarks. `http://www.spec.org`.

[SPEb] SPEC Viewperf 8.1. `http://www.spec.org/gwpg/gpc.static/vp81info.html`.

[SR09] Reiji Suda and Da Qi Ren. Accurate measurements and precise modeling of power dissipation of CUDA kernels toward power optimized high performance CPU-GPU computing. In *PDCAT '09 : Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 432–438, 2009.

[SS93] Michael J. Schulte and Earl E. Swartzlander. Truncated multiplication with correction constant. In *Proceedings of the 6th IEEE Workshop on VLSI Signal Processing*, pages 388–396. IEEE Computer Society Press, 1993.

[SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *ISCA '97 : Proceedings of the 24th annual international symposium on Computer architecture*, pages 194–205, 1997.

[Ste74] Pat H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[Swe09] Tim Sweeney. The end of the GPU roadmap, August 2009.

[SWND07] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 2.1*. Addison-Wesley, 6th edition, 2007.

[TMAJ08] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Technical report, HP Labs, 2008.

[Tri10] Damien Triolet. Nvidia GeForce GF100 : la révolution géométrique ? *Hardware.fr*, January 2010.

[UIU10] UIUC Parboil Benchmarks, 2010. `http://impact.crhc.illinois.edu/parboil.php`.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, 1990.

[VCA07] Panagiotis Vouzis, Caroline Collange, and Mark G. Arnold. LNS subtraction using novel cotransformation and/or interpolation. In *IEEE 18th Internatio-nal Conference on Application-specific Systems, Architectures and Processors*, pages 107–114, 2007.

[VCHK06] David A. Vallado, Paul Crawford, Richard Hujsak, and T.S. Kelso. Revisiting spacetrack report #3. In *Proceedings of the AIAA/AAS Astrodynamics Specialist Conference, Keystone, CO*, August 2006.

[VD08] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC'08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[vdL] Wladimir J. van der Laan. Decuda. `http://wiki.github.com/laanwj/decuda`.

[Vol10a] Vasily Volkov. Programming inverse memory hierarchy : case of stencils on GPUs. *GPU Workshop for Scientific Computing, International Conference on Parallel Computational Fluid Dynamics (ParCFD)*, May 2010.

[Vol10b] Vasily Volkov. Use registers and multiple outputs per thread on GPU. *6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*, June 2010.

[Was08] Scott Wasson. Nvidia's GeForce GTX 280 graphics processor. Technical report, The Tech Report, 2008.

[WPSAM10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[YKO+09] Steve Young, Dan Kershaw, Julian Odell, Dave Ollason, Valtcho Valtchev, and Phil Woodland. *The HTK Book (for HTK Version 3.4.1)*. Cambridge University Engineering Dept., 2009.

[ZHD04] Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4) :231–245, 2004.

Résumé

Les processeurs graphiques (GPU) actuels offrent une importante puissance de calcul disponible à faible coût. Ce fait a conduit à détourner leur emploi pour réaliser du calcul non graphique, donnant naissance au domaine du calcul généraliste sur processeur graphique (GPGPU).

Cette thèse considère d'une part des techniques logicielles pour tirer parti de l'ensemble des opérateurs arithmétiques spécifiques aux GPU dans le cadre du calcul scientifique, et d'autre part des adaptations matérielles aux GPU afin d'exécuter plus efficacement les applications généralistes.

En particulier, nous identifions la régularité parallèle comme une opportunité d'optimisation des architectures parallèles, et exposons son potentiel par la simulation d'une architecture GPU existante. Nous considérons ensuite deux alternatives permettant d'exploiter cette régularité. D'une part, nous mettons au point un mécanisme matériel dynamique afin d'améliorer l'efficacité énergétique des unités de calcul. D'autre part, nous présentons une analyse statique opérée à la compilation permettant de simplifier le matériel dédié au contrôle dans les GPU.

Mots-clef Architecture des ordinateurs, processeurs graphiques, arithmétique des ordinateurs, architectures parallèles

Abstract

Design challenges of GPGPU architectures : specialized arithmetic units and exploitation of regularity

Current Graphics Processing Units (GPUs) are high-performance, low-cost parallel processors. This makes them attractive even for non-graphics calculations, opening the field of General-Purpose computation on GPUs (GPGPU).

In this thesis, we develop software techniques to take advantage of the fixed-function units that GPU provide for scientific computations, and consider hardware modifications to execute general-purpose applications more efficiently.

More specifically, we identify parallel regularity as an opportunity for improving the efficiency of parallel architectures. We expose its potential through the simulation of an actual GPU architecture. Subsequently, we consider two alternatives to take advantage of regularity. First, we design a dynamic hardware scheme which improves the energy efficiency of datapaths and register files. Then, we present a static compiler analysis to reduce the complexity associated with instruction control on GPUs.

Keywords Computer architecture, graphics processing units, computer arithmetic, parallel architectures