UPVD
Université de Perpignan Via Domitia

École doctorale Energie Environnement ED305

Contributions au calcul sur GPU :
considérations arithmétiques et architecturales

HABILITATION À DIRIGER LES RECHERCHES

présentée et soutenue publiquement le 31 10 2014

pour l'obtention d'une

Habilitation de l'Université de Perpignan

(mention informatique)

par

David Defour

Composition du jury

Président :	Pierre Boulet,
Rapporteurs :	Daniel Etiemble, Jean-luc Lamotte, Stephane Vialle,
Examinateurs :	Bernard Goosssens, Fréderic Petrot,
Invité :	Jean-Michel Muller.

DALI -LIRMM — UMR 5506

Sommaire

Introduction

Le calcul est omniprésent dans notre quotidien. Il est aujourd'hui un bien de grande consommation, au même titre qu'une voiture ou un téléphone portable et offre les mêmes propriétés "commerciales" : il est standardisé, performant, fiable et économe.

Le fait de standardiser un outil comme le calcul présente de multiples avantages, tant du point de vue de l'utilisateur que du fabricant. L'utilisateur sait à quoi s'attendre car le standard a été conçu pour répondre aux attentes du plus grand nombre. De leur côté, les fabricants peuvent alors le produire à grande échelle en l'optimisant pour ce standard, réduisant ainsi les coûts de production. L'inconvénient est que le standard ne répond par forcément à toutes les attentes. Il y a alors deux réponses possibles : soit utiliser les outils génériques pour émuler ces besoins spécifiques, quitte à perdre en performance, soit faire évoluer le standard en fonction des besoins mais aussi des possibilités offertes par la technologie.

Le travail présenté dans ce document s'inscrit dans ce double objectif d'utiliser le matériel nécessaire au calcul, d'en analyser les différentes problématiques et de proposer des évolutions de celui-ci. Il décrit le travail entrepris à l'Université de Perpignan au sein de l'équipe DALI avec l'aide de mes deux doctorants Sylvain Collange puis Manuel Marin. Je soulignerai que l'année de mon arrivée à Perpignan en 2003 a été marquée par plusieurs changements. Ceux d'ordre personnel, puisque j'ai pu profiter de nouvelles collaborations pour traiter le calcul sous un angle plus global. Et enfin ceux relatif aux microprocesseurs, puisque cette période a été marquée par l'abandon de la course au gigahertz par les constructeurs.

Cet abandon s'est accompagné par une augmentation du nombre de cœurs de calcul par puce. Les architectures généralistes ne fournissant plus l'augmentation de performance escomptée, les chercheurs se sont tournés vers les coprocesseurs graphiques pour l'accélération de code scientifique. Dans le même temps, les environnements de développement pour ce type d'architecture se simplifiaient et les capacités de calcul flottant s'amélioraient. L'objet du premier chapitre de ce mémoire est de replacer nos contributions dans ce contexte historique en rappelant le fonctionnement de ces coprocesseurs, leurs évolutions et leurs capacités de calcul en virgule flottante.

L'arrivée des GPU et de leurs nombreuses unités de calculs permet d'envisager l'utilisation d'arithmétiques que nous qualifions de non conventionnelles et réputées coûteuses en temps et en ressources de calcul. Nous montrerons au deuxième chapitre comment la connaissance du comportements des diverses unités de calcul, de leur arrangement et de leurs propriétés peut aider à proposer des implantations logicielles efficaces de ces arithmétiques non standards. Nous décrirons nos contributions à travers des implantations optimisées sur GPU d'arithmétique en multiprécision, par intervalle, floue et logarithmique.

Dans le troisième chapitre, nous prendrons la liberté de nous soustraire des contraintes architecturales des processeurs existants pour envisager le calcul dans un cadre architectural plus large. Nous y envisagerons la problématique de la simulation fonctionnelle, de l'accès aux registres, des branchements, et des unités de calcul spécialisées.

Les GPU sont représentatifs de cette tendance de fond à avoir un nombre grandissant de

cœurs de calcul. Même si ceux-ci sont homogènes, gérer efficacement un grand nombre d'unités de calcul pose des problèmes aussi bien au niveau matériel qu'algorithmique. Il devient en effet indispensable de pouvoir en contrôler la consommation électrique, de gérer les inévitables erreurs matérielles ainsi que leur comportement qui se doit d'être le plus régulier possible. Nous avons réalisé des mesures relatives à ces éléments qui nous ont permis de formuler un certain nombre de propositions couvertes dans le quatrième chapitre.

1

Calcul flottant sur GPU

Les processeurs graphiques, ou GPU, que nous considérons sont avant tout destinés à l'accélération de traitement graphique. Ces processeurs sont progressivement passés de pipelines de traitement graphique paramétrables à des unités de calcul programmables. Grâce aux nombreuses et diverses unités flottantes, ils disposent aujourd'hui d'une importante puissance de calcul. Cette puissance s'est rapidement révélée très intéressante pour l'accélération d'applications généralistes à l'aide de GPU que l'on appelle aussi traitement GPGPU. La démocratisation du GPGPU est le fait de l'amélioration du support de la norme flottante IEEE-754 et l'arrivée des environnements de développement dédiés tels que CUDA et OpenCL. Aussi dans ce chapitre, nous commencerons avec une présentation du fonctionnement des différentes unités de calcul, suivie des liens entre la norme IEEE-754 et les processeurs graphiques.

Liste des publications relatives à ce chapitre : [DDD06, CDD08a, MD13].

1.1 Fonctionnement d'un GPU

1.1.1 Modèle de programmation

Différents environnements de programmation permettent d'accéder aux ressources de calcul des processeurs graphiques. Historiquement, il était nécessaire de passer par un langage de programmation graphique comme les langages de *shaders* d'OpenGL et DirectX. Les programmes étaient alors compilés par le pilote graphique avant d'être exécutés par le processeur graphique. Des environnements alternatifs sont apparus avec la progression de l'utilisation des processeurs graphiques pour le calcul généraliste (GPGPU), comme CTM proposé par AMD/ATI, ou CUDA proposé par Nvidia. CUDA s'est rapidement imposé comme un standard dans le monde du GPGPU. L'inconvénient est qu'il ne pouvait fonctionner que sur des cartes Nvidia. Un standard plus ouvert vers les autres constructeurs est alors apparu : OpenCL. Il existe une analogie entre les deux langages et le découpage du travail résumé dans le tableau 1.1. Nous utiliserons dans la suite de ce document la terminologie utilisée par CUDA.

Les fonctions exécutées sur le GPU sont appelées *kernels*. Du point de vue du programmeur, la hiérarchie des threads est divisée en trois niveaux : *threads*, *blocks* et *grids*. Le même code d'un kernel est exécuté par plusieurs threads fonctionnant en parallèle sur des données différentes. Les *threads* sont regroupés en paquets de *block_size* afin de former ce que l'on appelle des *blocks*. Les blocs sont eux aussi regroupés en paquets de *grid_size* afin de former ce que l'on appelle une *grid*. Les threads d'un bloc ainsi que les blocs d'une grille sont identifiés de façon unique par leurs coordonnées respectives dans les blocs et la grille.

CUDA	OpenCL
Thread	Work-item
Bloc de threads	Work-group
Dimension de grille	Local-work-size et global-work-size
Mémoire globale	Mémoire globale
Mémoire de constante	Mémoire de constante
Mémoire partagée (Shared)	Mémoire locale
Mémoire locale	Mémoire privée

TABLE 1.1 – Relation entre terminologie CUDA et OpenCL.

La compilation d'un programme sur GPU se décompose en trois étapes. Premièrement, à l'aide de directives, le programme est divisé en un programme hôte et un programme GPU. La partie hôte est ensuite compilée par la chaîne de compilation C/C++ traditionnelle. La partie GPU est elle, compilée à l'aide d'un back-end dédié. Le résultat est un binaire destiné à être exécuté sur le GPU. Les programmes hôte et GPU sont ensuite liés à l'aide des bibliothèques fournies par le constructeur.

Lors de l'exécution, le CPU contrôle via le driver les différentes étapes : création d'un contexte associé à un GPU, allocation de la mémoire sur le GPU, instanciation des transferts de données et de code, et lancement du kernel. Les dimensions de la grille et des blocs sont spécifiées au lancement.

1.1.2 Un peu d'histoire : le pipeline graphique

Les GPU traitent principalement des objets géométriques et des pixels. Les images sont créées en appliquant des transformations géométriques aux sommets et en découpant les objets en fragments ou pixels. Les calculs sont réalisés par différents étages de ce que l'on appelle le pipeline graphique, comme présenté à la figure 1.1. Ce pipeline est formalisé par les environnements de développement OpenGL et DirectX. Il était accéléré en matériel en utilisant un pipeline paramétrable qui a progressivement évolué vers un pipeline programmable. Jusqu'en 2006, deux types d'unités étaient programmables : les *pixel shader* et les *vertex shader*.

La machine hôte envoie des sommets pour positionner dans l'espace des objets géométriques primitifs (polygones, lignes, points). Ces objets primitifs subissent des transformations (rotation, translation, illumination,...) avant d'être assemblés pour créer un objet plus complexe. Ces opérations sont réalisées dans l'unité de traitement des sommets (*vertex shader*).

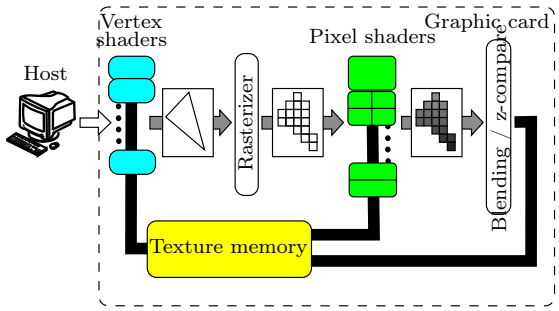


FIGURE 1.1 – Vue d'ensemble du pipeline graphique

Quand un objet a atteint sa position, sa forme et son éclairage final, il est découpé en fragments ou pixels. Une interpolation est effectuée pour obtenir les propriétés de chaque pixel. Les pixels sont ensuite traités par l'unité de traitement des pixels (*pixel shader*) qui réalise des tâches d'affichage comme par exemple l'application d'une texture ou le calcul de la couleur d'un pixel.

1.1.3 GPU modernes

En 2007, les environnements de développement ont évolué afin de permettre un accès aux ressources de calcul sans passer par les environnements de développement graphique. L'environnement CUDA en est le plus représentatif. Cette évolution s'est accompagnée au niveau matériel d'une fusion des vertex shader et des pixel shader pour donner naissance aux *compute unit*.

Au cours de nos travaux, nous avons suivi ces évolutions et considéré principalement des GPU produits par Nvidia. En terminologie CUDA, les GPU sont constitués de cœurs CUDA, appelés Streaming Processors (SP), organisés de façon hiérarchique. Ces cœurs CUDA sont regroupés en multiprocesseur (MP). Le nombre de cœurs CUDA par multiprocesseur varie entre 8 et 64 en fonction de ce que l'on appelle *CUDA capability*, correspondant à la génération architecturale. Le tableau 1.2 liste les caractéristiques des principaux processeurs que nous avons considérés. De façon similaire, les multiprocesseurs sont regroupés afin de former les *Graphics Processing Cluster* (GPC). La figure 1.2 décrit la relation entre l'architecture TESLA, première architecture compatible CUDA, et l'environnement de développement CUDA.

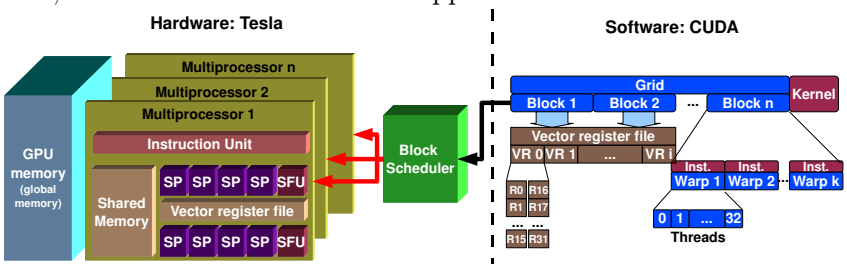


FIGURE 1.2 – Description de l'architecture TESLA.

Le support architectural d'exécution repose sur des unités SIMD, ce qui impose de regrouper l'exécution des threads en *warps* de 32 threads. Chaque instruction est prise en charge par un multiprocesseur pour s'exécuter sur un warp. Les multiprocesseurs maintiennent un scoreboard pour chaque warp afin de gérer le lancement des warps en fonction de leur type, des instructions et d'une politique d'ordonnancement "équitable". Cette technique est au cœur de l'efficacité des GPU en permettant de recouvrir les longues latences d'accès à la mémoire en entrelaçant l'exécution d'instructions de plusieurs warps éventuellement de blocs différents.

Au niveau mémoire, les GPU disposent d'une hiérarchie assez proche des processeurs généralistes avec des bancs de registres, une mémoire partagée, des caches de données L1 et L2, de la mémoire de constante et de texture.

1.1.4 Unités de calcul

Chaque multiprocesseur (SM) contient la logique nécessaire pour charger, décoder et exécuter des instructions qui travaillent sur des vecteurs de 32 éléments. Il y a entre 256 et 2048 registres vectoriels composés de 32 mots sur 32 bits. En plus de ce banc de registres, chaque multiprocesseur contient une mémoire partagée et des caches séparés pour les données constantes et les instructions.

Nom Commercial	Arch.	CUDA Capability	#MP/ GPC	#MP	CUDA Core / MP	Warp Scheduler / MP	GPU Clock (Mhz)	Memory Clock (Mhz)
C870	G80	1.0	2	16	8	1	1350	800
9800 GX	G92	1.1	2	16	8	1	1500	1000
GTX 480	GF100	2.0	4	15	32	2	1401	1848
GTX 560	GF114	2.1	4	7	48	2	1620	2004
GTX 680	K10	3.0	3	8	192	4	1059	3004

TABLE 1.2 – Description de GPU Nvidia de différentes générations.

Le calcul est pris en charge par différents types d'unité de calcul organisés hiérarchiquement, comme présenté sur la figure 1.3 pour le cas du GT200. On retrouve un nombre variable de streaming processors (SP), capables de réaliser les opérations de type MAD ou FMA en simple ou double précision en fonction des générations. L'approximation de fonctions comme la division, la racine carrée ou les fonctions trigonométriques est accélérée en matériel à l'aide des unités de calcul des fonctions de base (SFU).

En plus de ces unités de calcul, les GPU disposent d'unités spécialisées dans le traitement graphique comme les unités ROP (*Raster Output*) et les unités de texturage/filtrage.

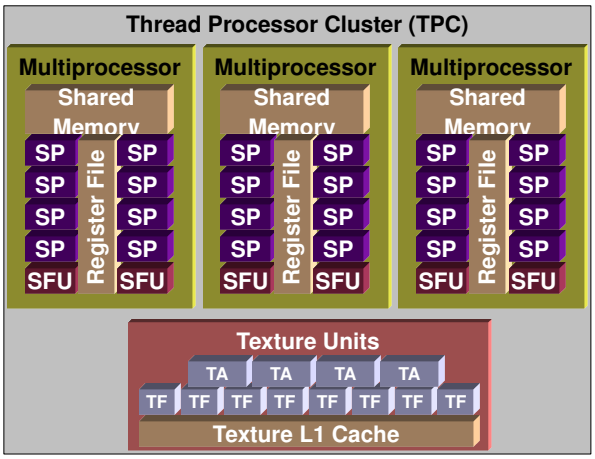


FIGURE 1.3 – Diagramme d'un Thread Processor Cluster d'une NVIDIA GT200.

Les unités d'évaluation des fonctions de base

Depuis l'apparition des *shaders* programmables, les processeurs graphiques disposent de plusieurs unités capables d'évaluer les fonctions de base les plus utilisées en graphisme. Parmi ces fonctions, on retrouve l'évaluation en matériel en simple précision de l'inverse, de l'inverse de la racine carrée, du cosinus, du sinus, du logarithme et de l'exponentiel en base 2.

Oberman propose dans [107] de fusionner l'unité d'interpolation et l'unité d'évaluation des fonctions de base pour réduire l'empreinte matérielle tout en conservant une latence raisonnable. Cette unité d'interpolation est capable d'évaluer la fonction $U(x, y) = A \times x + B \times y + C$. Pour évaluer les fonctions de base, il propose d'utiliser l'interpolateur précédé d'un peu de matériel pour évaluer des approximations polynomiales de degré 2. Les coefficients des polynômes, stockés dans une table, dépendent de la fonction évaluée et des arguments. Les 22,75 Kb de cette table

représentent 18% de la surface totale de l'ensemble interpolation et évaluation des fonctions de base, ou encore 91% des circuits à ajouter pour faire en sorte que l'unité d'interpolation soit capable d'évaluer ces fonctions de base.

Une unité similaire est décrite dans un brevet d'ATI [45]. Elle repose sur un circuit *ad hoc* d'évaluation de polynômes et sur une table à 32 entrées par fonction. Chaque entrée contient un polynôme d'approximation de degré 3 dont les coefficients sont codés sur 72 bits au total.

Les unités de texturage

Les unités de texturage, appelées aussi unités de filtrage, sont destinées à l'accélération de certains types d'accès mémoire rencontrés pour des opérations graphiques, bénéficiant de la localité spatiale en 1,2 ou 3 dimensions. Ces unités sont situées dans un autre domaine d'horloge que le reste du processeur et comportent des unités de calcul d'adresses de texture (TA) ainsi que des unités de filtrage de texture (TF). La localité spatiale est exploitée à l'aide d'un cache de texture en lecture seule. Les unités de texturage accèdent à des texels, et calculent la valeur finale en appliquant une fonction de filtrage (linéaire, bilinéaire, anisotropique).

Filtrage bilinéaire Une texture peut être configurée pour retourner des données flottantes. Il est alors possible de choisir un type de filtrage (linéaire ou au plus près). Le filtrage linéaire est une interpolation en faible précision entre des texels voisins. Cette interpolation peut être réalisée en 1, 2 ou 3 dimensions.

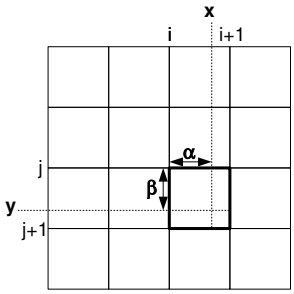


FIGURE 1.4 – Filtrage bilinéaire d'un texture bidimensionnelle.

Soit T une texture en 2 dimensions de $N \times M$ texels, chargée en utilisant des coordonnées x et y normalisées ($x \in [0, N]$ and $y \in [0, M]$). L'unité de filtrage renverra alors une valeur V telle que :

$$\begin{aligned} V \quad = \quad & (1-\alpha)\cdot(1-\beta)\cdot T[i,j] \\ & +(1-\alpha)\cdot\beta\cdot T[i+1,j] \\ & +\alpha\cdot(1-\beta)\cdot T[i,j+1] \\ & +\alpha\cdot\beta\cdot T[i+1,j+1] \end{aligned} \tag{1.1}$$

avec

$$i = floor(x - 0.5) \text{ et } \alpha = frac(x - 0.5)$$

$$j = floor(y - 0.5) \text{ et } \beta = frac(y - 0.5)$$

Il est à remarquer que sur les GPU Nvidia, α et β sont généralement manipulés en virgule fixe avec une partie fractionnaire sur 8 bits. Les opérateurs de calcul impliqués pour le filtrage sont taillés au plus juste, ce qui limite la précision du dernier bit du résultat produit.

1.2 Arithmétique IEEE-754

Les processeurs graphiques des générations 7800GTX et ATI RX1800XL disposaient déjà de la puissance et de la flexibilité de calcul pour supporter des opérations sur 16, 24 ou 32 bits. Malheureusement, à cette période, les deux principaux constructeurs (ATI et Nvidia) gardaient confidentielles les caractéristiques et la description de l'implantation des différents opérateurs arithmétiques embarqués dans les différentes unités de traitement. Or ces informations étaient essentielles pour estimer et contrôler les erreurs d'arrondi, ou pour mettre en œuvre des techniques de réduction ou de compensation afin par exemple de travailler en précision double, quadruple ou arbitrairement étendue.

1.2.1 Tests de l'arithmétique à virgule flottante

Les premiers logiciels réalisés pour tester l'arithmétique à virgule flottante sur CPU avaient pour unique but de découvrir les caractéristiques des fonctionnalités implantées [39,59,133]. Suite à l'adoption massive de la norme IEEE-754 en 1985, d'autres sont apparus pour vérifier la bonne implantation de cette norme par une série de tests bien choisis [80,110,149], ou simplement pour tester des fonctions élémentaires, spéciales ou complexes [40,41]. Certains logiciels tels UCBTest [1] testent à la fois la conformité des fonctionnalités normalisées, la qualité des autres fonctionnalités usuelles et le bon fonctionnement de la chaîne de compilation.

Comme nous l'avions évoqué précédemment, les premiers GPU ne respectaient pas la norme IEEE-754, contrairement à la majorité des CPU. Ainsi il a été nécessaire de vérifier le comportement arithmétique des opérateurs flottants afin de faciliter le portage des programmes pour CPU sur GPU comme par exemple ceux permettant d'augmenter la précision des opérateurs arithmétiques [DD06].

Des tests ciblant une meilleure compréhension interne des GPU avaient déjà été développés et regroupés dans une bibliothèque OpenGL appelée GPUbench [28]. L'objectif principal était de caractériser certains paramètres internes des pixel shaders tels que le débit, le nombre d'instructions pouvant être exécutées, le chargement de données à partir d'une texture, mais ils ne permettaient pas de vérifier le comportement de l'arithmétique flottante sur les GPU.

Nous avions alors défini et implanté dans [DDD06, CDD08a] des algorithmes pour mieux comprendre le fonctionnement de l'addition, de la multiplication et le stockage des nombres à virgule flottante dans les registres et en mémoire. Bien que les algorithmes proposés puissent être adaptés à d'autres formats de données, nous avions ciblé le format simple précision sur 32 bits. Les tests avaient été écrits en OpenGL et utilisaient les *Pbuffer* pour le stockage des textures. L'avantage des Pbuffer était qu'ils étaient acceptés par les 2 principaux constructeurs de cartes graphiques de l'époque : ATI et Nvidia. L'inconvénient était de nécessiter une syntaxe propre à chaque système (SGIX_pbuffer sous Linux et WGL_pbuffer sous Windows) ce qui rendait le portage d'applications plus complexe.

Ainsi en testant les additionneurs et multiplieurs des vertex et pixels shader des cartes Nvidia 7800 GTX et ATI RX1800XL, nous avions montré, entre autre, que :

- l'addition et la multiplication étaient tronquées sur les deux GPU,
- la soustraction bénéficiait d'un bit de garde sur Nvidia mais pas sur ATI,
- la multiplication implantait un arrondi fidèle sur les deux GPU,
- la division reposait sur une multiplication du numérateur par une approximation de l'inverse du dénominateur [25],
- les registres temporaires stockaient les nombres flottants uniquement sur 32 bits,

1. Ce programme est disponible sur le dépôt Netlib `http://www.netlib.org`.

— les multiplieurs utilisaient un biais pour compenser la troncature,
— les additionneurs avaient 2 bits de garde, expliquant les incertitudes lors des calculs d'erreur d'arrondi.

Ces tests nous ont permis de démontrer que le portage d'applications sur GPU devait être réalisé avec les plus grandes précautions car le comportement des opérateurs arithmétiques flottants différaient d'un constructeur à un autre mais aussi à l'intérieur même d'une puce.

Ces informations essentielles nous ont permis de mieux comprendre les différences relatives à l'arithmétique flottante entre une exécution sur CPU et une autre sur GPU. Elles ont constitué une première étape pour la construction d'algorithmes numériques plus précis et plus rapides sur GPU (voir section 2.1.1).

1.2.2 Format et performance des calculs

L'arithmétique à virgule flottante offre un spectre de nombres représentables plus étendu que l'arithmétique entière ou à virgule fixe sans que le programmeur ait besoin de gérer manuellement décalages et recadrages. L'implantation de l'arithmétique à virgule flottante [46] est normalisée par un standard international IEC-ISO 60559 [4] et par deux standards américains IEEE-ANSI 754 [141, 142] et 854 [42], révisés en 2008 [140].

Ce standard définit en particulier la représentation des données en virgule flottante (nombres normalisés, dénormalisés, infinis, NAN), les modes d'arrondis, le comportement d'un ensemble d'opérations sur ces formats et la gestion des exceptions. Le format de représentation binaire, résumé dans tableau 1.3, est aujourd'hui le plus utilisé et, logiquement, est celui que l'on trouve dans les GPU.

Nom	Nom d'usage	Mantisse	Exposant min	Exposant max
binary16	half / __float16	10+1	-14	+15
binary32	float / simple précision	23+1	-126	+127
binary64	double / double précision	52+1	-1022	+1023
binary128	quad / __float128	112+1	-16382	+16383

TABLE 1.3 – Format de représentation binaire des nombres flottants

Historiquement, les GPU n'étaient pas destinés au calcul haute performance (HPC) et de ce fait s'avéraient assez mauvais lorsqu'il s'agissait de double précision. Le support de la double précision n'est apparu chez Nvidia qu'à partir des GT200, avec un surcoût d'un facteur 8 par rapport à la simple précision. Depuis, Fermi et Kepler ont permis de réduire cette différence à un facteur compris entre 2 et 3.

Ainsi, la majorité des GPU actuels implantent en matériel le format binary32 (float) ainsi que le binary64 (double). Mais depuis peu, on trouve aussi du support matériel pour le format binary16 (half) dont on retrouve trace dans le langage Cg de Nvidia apparu en 2002. Ce format ayant été conçu pour le stockage et le transfert des données, il est préférable de le réserver pour cet usage. Enfin, le format binary128 n'est à ce jour qu'assez peu utilisé et n'est disponible que par émulation logicielle.

Les dernières versions de CUDA supportent nativement les types simple et double précision et leurs dérivés vectoriels (float2,4, double2,4), ainsi que les types entiers de 8 à 64 bits (char, short, int, long, long int). Mais ce support est limité pour les types binary16 et les opérations sur 8 et 16 bits (les opérations sur des char et short ont un coût identique aux entiers). En

effet, dans ces formats, il est uniquement possible de charger et sauvegarder les données depuis et vers la mémoire, ce qui permet de réaliser des économies en bande passante. Une fois en registre, il suffit de les convertir vers des types supérieurs par exemple à l'aide des unités de texturage pour effectuer des opérations. Dans [MD13], nous avons montré comment utiliser les différents formats de représentation des nombres flottants (half, simple, double, double-double, quad-double) et mesuré leur impact sur la performance.

S'il est correctement utilisé, le format half permet de réduire les transferts de données entre CPU et GPU, entre mémoire globale et registres mais aussi dans les transferts impliquant la mémoire partagée. Ce phénomène est illustré dans la figure 1.5 pour le problème de la sommation. Pourquoi ? Parce qu'au delà de simplement réduire les transferts, elle permet d'atteindre des facteurs d'accélération supra-linéaires grâce aux effets de cache. Ces effets reposent sur le fait que plus de données peuvent tenir en cache L1, L2, ou en shared memory, ce qui diminue d'autant le nombre d'itérations ou de tuiles à considérer dans les algorithmes numériques.

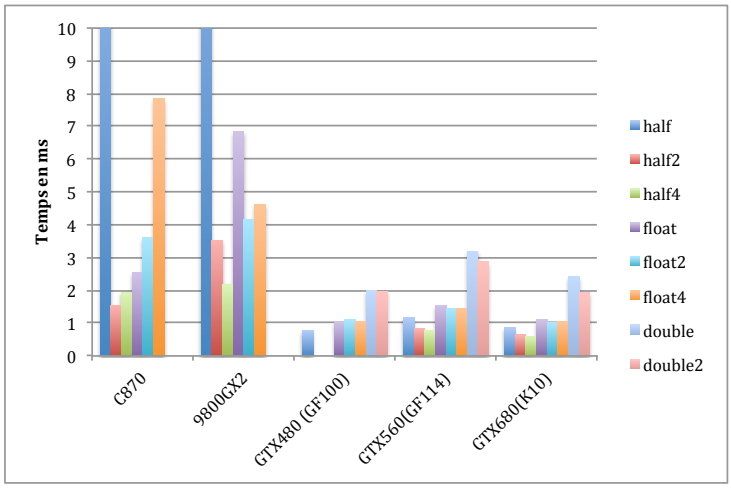


FIGURE 1.5 – Temps d'évaluation d'une somme de 2^{25} flottants selon l'architecture, le format (half, float, double) et l'alignement mémoire.

Ce phénomène n'apparait pas pour le problème de la réduction (la mémoire partagée n'est pas utilisée pour accélérer les accès mémoire). En revanche, pour des problèmes comme la multiplication matrice-vecteur creux et les solveurs PDE ou QCD, il peut se révéler intéressant. Ainsi, dans l'expérience de Clark et al. [38], l'implantation d'un solveur QCD basé sur des half s'est revélée 6.7 fois plus rapide que la version en double et 2.1 fois plus rapide que la version simple.

2

Arithmétiques non conventionnelles

Lorsque les propriétés de l'arithmétique offerte par le standard IEEE-754 ne sont pas suffisantes, il faut faire appel à ce que nous appellerons *arithmétique non conventionnelle*. Cela peut couvrir des problématiques de précision, de dynamique de représentation, ou d'incertitude dans les données manipulées.

Les processeurs graphiques offrent en grand nombre des unités de calcul flottant avec des caractéristiques propres qu'il est possible d'utiliser pour réaliser des opérations en arithmétique non conventionnelle. Ainsi dans une première partie, nous discuterons de la problématique des opérations en grande précision dans un contexte de support incomplet de la norme IEEE-754 correspondant aux premiers GPU. Ensuite nous présenterons comment réaliser efficacement sur GPU des opérations sur des données intégrant un ou plusieurs degrés d'incertitude avec les opérations d'arithmétique par intervalle et d'arithmétique floue. Enfin nous terminerons ce chapitre par la description d'une implantation d'arithmétique logarithmique sur GPU reposant sur des évaluations polynomiales accélérées à l'aide des unités d'interpolation.

Liste des publications relatives à ce chapitre : [CDGI14, DM14, DM13a, CDD12, CFD08, ACD10, DD06, DDD06, MDM14].

2.1 Arithmétique multiprécision

Le support de la norme IEEE-754 n'a que peu d'intérêt pour les jeux vidéo, mais est indispensable pour le calcul scientifique. Aussi, à mesure que les processeurs graphiques ont été utilisés pour le HPC, les fabricants ont progressivement amélioré le support de la norme. Pendant longtemps, les précisions de calcul disponibles dans les pipelines graphiques étaient limitées : par exemple les canaux de couleurs pouvaient se contenter de 8 bits de précision. Aussi le support (partiel) des flottants simple précision n'est apparu qu'avec l'introduction des shaders 3.0. Ces premières architectures avec lesquelles nous avons travaillé, Nvidia GeForce 6 et ATI X1k, ne disposaient que d'un support partiel de la norme flottante pour la simple précision et ne proposait pas la double précision.

2.1.1 Formats de représentations

Les applications telles que les simulations longues, celles de la théorie des nombres, ou d'algorithmes itératifs comme le calcul d'ombre ou d'illumination [143], nécessitent plus de précision que celle offerte sur le GPU. Une solution était de transférer les opérations nécessitant plus de précision sur le CPU, comme dans le cas du solver d'éléments finis décrit dans [60] où les calculs

en double précision étaient réalisés sur CPU au prix de coûteux transferts CPU-GPU. La solution pour contourner ce problème est d'utiliser de l'arithmétique multiprécision directement sur GPU. Cette arithmétique consiste à représenter un nombre par plusieurs mots en mémoire. Le nombre de mots mémoires, ou *limb*, peut être choisi statiquement à la compilation ou déterminé dynamiquement à l'exécution.

On retrouve dans la littérature de nombreux travaux autour des algorithmes multiprécision adaptés pour les CPU. Ces bibliothèques émulent des opérateurs arithmétiques avec une précision supérieure à celle fournie par le matériel en utilisant soit les unités entières, soit les unités flottantes, en fonction de leur représentation interne.

Bibliothèques basées sur les entiers. Toutes les bibliothèques de cette catégorie utilisent un tableau de mots machine entiers (32 ou 64 bits) pour représenter un nombre en multiprécision. C'est le cas de GMP [52] qui est à la base de plusieurs autres bibliothèques comme MPFR [2]. Ces bibliothèques ont l'avantage d'offrir une gestion dynamique de la précision pour chaque nombre et chaque opération, même si d'autres bibliothèques [DdD03, 73] définissent la précision au moment de la compilation pour obtenir de meilleures performances.

Bibliothèques basées sur les flottants. Les processeurs modernes ont des unités flottantes très performantes. Aussi certaines bibliothèques, comme MPFUN [16], exploitent ces opérateurs en utilisant une représentation en virgule flottante des *limbs*.

D'autres bibliothèques représentent un nombre en plus grande précision en utilisant une somme non évaluée de plusieurs nombres flottants. C'est le cas de la bibliothèque double-double de Briggs [24], quad-double de Bailey [70] et des expansions de Priest [117]. Ces formats utilisent des algorithmes simples basés sur des briques de base de type *Error Free Transformation, (EFT)* pour réaliser les opérations arithmétiques. Toutefois, ils sont limités à de faibles précisions (2 à 3 nombres à virgule flottante) car la complexité des algorithmes augmente quadratiquement avec la précision.

2.1.2 Implantation sur GPU

La référence en terme de bibliothèque multiprécision est GMP, et MPFR pour la version flottante. Ces bibliothèques exhibent un parallélisme de données important et peuvent donc apparaître comme de bons candidats pour une accélération sur GPU. Cependant, leur intérêt est de pouvoir s'adapter dynamiquement à la précision nécessaire, ce qui implique une gestion dynamique de la mémoire et des traitements associés. Ce type de représentation est donc inadapté pour une utilisation sur GPU. On lui préfèrera une solution impliquant une gestion statique de la représentation.

Dans [DDD06,DD06], nous avons donc adapté le format *float-float* pour une utilisation sur les GPU de 2006 correspondant à une précision de 48-bits ciblant les architectures Nvidia. Ce format s'inspire du format double-double décrit dans [24] et utilise une représentation interne basée sur la somme non-évaluée de deux nombres flottants simple précision. Ce format présente deux avantages sur ce type d'architecture : Le premier est de ne pas nécessiter de gestion dynamique de la mémoire puisque les nombres sont stockés sous la forme de deux flottants. Le deuxième est de ne nécessiter que des registres flottants et de ne pas avoir de coûteux aller-retours avec la hiérarchie mémoire.

Nous avons vu en section 1.1.2 que le pipeline graphique des architectures de 2006 était composé de plusieurs unités de calcul avec des caractéristiques assez différentes de celles disponibles sur un CPU. En effet, les tests et comparaisons sont à éviter et les propriétés arithmétiques

diffèrent légèrement. Nous avons donc repris les algorithmes de [138] et prouvé qu'ils étaient toujours valides dans le contexte de l'arithmétique des architectures Nvidia de 2006, à savoir des architectures avec un bit de garde pour l'addition/soustraction et la troncature comme mode d'arrondi.

Nous avons réalisé une implantation de l'addition et de la multiplication dans le langage de haut niveau Brook [29]. Ce langage permettait de s'affranchir des nombreuses contraintes liées au driver et au système. Il générait au choix une version OpenGL ou DirectX mais réalisait des optimisations interdites par la norme IEEE-754, comme par exemple le remplacement de la séquence de calcul de l'erreur $r = ((a \oplus b) \ominus a)$ par $r = b$, avec a et b flottant. Les implantations étant limitées au format float, des tests exhaustifs avaient été réalisés pour vérifier la présence d'erreurs dans les briques de base des algorithmes mis en œuvre. Nous avions alors détecté des erreurs de comportement dans des cas très particuliers, comme par exemple pour l'algorithme $Add12$ qui réalise la somme non-évaluée de deux flottants a et b avec $a = 1.0$ et $b = -(2^{24} - 2^{48})$ en entrée. Sans test exhaustif, nous n'aurions pas pu mettre en évidence une erreur de ce type et cela nous montre combien un support de la norme IEEE-754 et une connaissance approfondie du comportement des opérateurs est indispensable pour pouvoir construire des programmes numériquement fiables.

Nous avons comparé les implantations entre une carte Nvidia 7800GTX avec 256Mo de mémoire et fonctionnant à 430 MHz et un Pentium IV HT à 3.2 Ghz. Les résultats ont montré que les opérations float-float sur GPU ne représentaient qu'un surcoût de 2 par rapport à des opérations en simple précision alors que ce ratio variait entre 5 et 13 sur CPU.

Au delà de la faisabilité d'une bibliothèque multiprécision sur GPU, nous avons montré à travers ces résultats que le parallélisme de données pouvait être utilisé avantageusement sur ce type d'architecture pour compenser un parallélisme d'instruction (ILP) moindre. En effet, les CPU exploitent efficacement l'ILP disponible des différents algorithmes de ce type [84], mais ne sont pas capables de recouvrir les latences en utilisant les instructions entre les itérations. Dans la continuité de ces résultats, [151] a montré qu'il fallait à la fois exhiber de l'ILP et du parallélisme de données pour exploiter efficacement les GPU.

La multiprécision exhibe naturellement du parallélisme de données. Nos travaux ont principalement porté sur l'utilisation de briques de base utilisant des error-free transformation (EFT) car ceux-ci ne nécessitent pas de gestion dynamique de la mémoire. Avec la relaxation du modèle mémoire des nouveaux GPU et l'arrivée de la gestion dynamique du parallélisme, il est possible d'envisager au cas par cas des algorithmes multiprécision offrant plus de liberté dans le choix du format de représentation interne des données et dans l'exploitation du parallélisme. Ce n'est cependant pas chose facile, comme nous pouvons le constater avec notre adaptation sur GPU de l'accumulateur long de Kulisch [CDGI14].

2.2 Arithmétique par intervalle

La question de la fiabilité et de la confiance dans les calculs produits par les machines ressurgit régulièrement. Par le passé, l'incapacité à produire un résultat précis était le fait d'une mauvaise gestion des exceptions (division par zéro), de cancellation catastrophique, de problème mal conditionné ou mal posé.

Avec l'arrivée de processeurs graphiques dont les performances se comptent en teraflops par puce, que l'on retrouve dans des machines exaflopiques, la majorité des systèmes fonctionnant sur ce type de machine peuvent être considérés comme mal conditionnés.

Une solution est d'utiliser une partie de la puissance de calcul de ces systèmes pour évaluer

une borne certifiée pour le résultat en lieu et place d'une estimation sans garantie fournie par l'arithmétique en virgule flottante. L'arithmétique d'intervalle (IA) permet cela en intégrant une incertitude de premier ordre dans les données. L'intervalle correspond alors à l'ensemble des valeurs possibles pour une variable. Il permet aussi de certifier l'inclusion du résultat en intégrant les erreurs réalisées lors des calculs, comme les erreurs d'arrondi. Ainsi une variable n'est plus représentée par un scalaire, mais par un intervalle bornant les valeurs possibles.

2.2.1 Formats de représentations

Un intervalle peut être représenté de 3 façons différentes, chacune ayant ses avantages et inconvénients :

1. En utilisant une des deux bornes combinées au diamètre de l'intervalle.
2. En utilisant une borne inférieure et une borne supérieure.
3. En utilisant le centre et le rayon de l'intervalle.

La plupart des bibliothèques existantes, Boost [26], MPFI [121] et GAOL [62], utilise une représentation borne inférieure/borne supérieure. La représentation centre-rayon est parfois utilisée afin de bénéficier de bibliothèques BLAS existantes et optimisées pour ce format. Décrivons ces deux alternatives.

Représentation Inf-Sup Dans ce format, un intervalle est représenté par deux flottants notés, l et u représentant respectivement la borne inférieure et la borne supérieure de l'intervalle. L'intervalle $[l, u]$ est l'ensemble des éléments compris entre l et u :

$$[l, u] = \{x \in \mathbb{R} : l \leq x \leq u\}$$

Les opérations sur les intervalles sont alors définies de la manière suivante [101] :

$$\begin{array}{rcl} [a,b] + [c,d] & = & [\bigtriangledown(a+b), \bigtriangleup(c+d)], \\ [a,b] - [c,d] & = & [\bigtriangledown(a-d), \bigtriangleup(b-c)], \\ [a,b] \cdot [c,d] & = & [\min(\bigtriangledown(a\cdot c), \bigtriangledown(b\cdot d), \bigtriangledown(a\cdot d), \bigtriangledown(b\cdot c)), \\ & & \max(\bigtriangleup(a\cdot c), \bigtriangleup(b\cdot d), \bigtriangleup(a\cdot d), \bigtriangleup(b\cdot c))] \end{array}$$

où $\bigtriangledown$ est l'arrondi vers $-\infty$ et $\bigtriangleup$ est l'arrondi vers $+\infty$. Le changement de mode d'arrondi garantit que l'intervalle résultat inclut toutes les solutions possibles. Pour de nombreuses architectures, dont les CPU, le mode d'arrondi correspond à un état du processeur, ce qui nécessite de vider le pipeline entre chaque changement de mode d'arrondi. Ces changements ont donc un surcoût non négligeable dans les opérations d'arithmétique d'intervalle.

Représentation centre-rayon Dans ce format, un intervalle est représenté par un centre m et un rayon ρ [126]. L'intervalle $\langle m, \rho\rangle$ est l'ensemble des éléments se trouvant à distance au plus ρ de m, c'est à dire :

$$\langle m, \rho\rangle = \{x \in \mathbb{R} : |x - m| \leq \rho\}$$

Les opérations sur les intervalles sont alors définies de la manière suivante :

$$\begin{array}{rcll} \langle a,\alpha\rangle + \langle b,\beta\rangle & = & \langle \square(a+b), & \bigtriangleup(\epsilon' |\square(a+b)| + \alpha + \beta)\rangle \\ \langle a,\alpha\rangle - \langle b,\beta\rangle & = & \langle \square(a-b), & \bigtriangleup(\epsilon' |\square(a-b)| + \alpha + \beta)\rangle \\ \langle a,\alpha\rangle \cdot \langle b,\beta\rangle & = & \langle \square(a\cdot b), & \bigtriangleup(\eta + \epsilon' |\square(a\cdot b)| + \\ & & & (|a| + \alpha)\beta + \alpha|b|)\rangle \end{array}$$

où $\bigtriangledown$ est l'arrondi vers $-\infty$, $\bigtriangleup$ est l'arrondi vers $+\infty$ et $\square$ est l'arrondi au plus près. Ici $\epsilon' = \frac{1}{2}\epsilon$ avec ϵ l'erreur d'arrondi, et η est le plus petit nombre flottant (dénormalisé) représentable. En double précision, $\epsilon = 2^{-53}$ et $\eta = 2^{-1074}$.

2.2.2 Implantation sur GPU

Nous avons décrit dans [CDD12, CFD08] les différentes implantations d'arithmétique d'intervalle que nous avons réalisées. Ces développements ont commencé en 2006, période où l'environnement de développement et les opérations arithmétiques n'étaient pas stabilisés. Regroupons ces développements selon deux catégories : ceux concernant les premières générations de GPU et ceux relatifs aux dernières générations de processeurs Nvidia.

Première génération

Cette version utilisait le pipeline graphique et reposait sur le langage Cg [54], fournissant une interface d'exécution unique pour les GPU ATI/AMD et Nvidia. Rapidement, CUDA est arrivé et s'est imposé comme le langage de développement en GPGPU. Nous avons donc cessé de maintenir la version Cg.

La version CUDA était basée sur la bibliothèque Boost, nous permettant de bénéficier de la souplesse de programmation C++ comme les templates. Malgré tout, nous avons dû contourner de nombreux problèmes pour cette implantation :

arrondi. Les premières générations de GPU Nvidia ne supportaient pas de mode d'arrondi vers $+/-\infty$. En revanche, il était possible d'effectuer des arrondis au plus près et/ou vers zéro pour l'addition et la multiplication (Section 1.2). Une solution a été d'utiliser le mode d'arrondi vers zéro et d'ajouter/soustraire un ulp en fonction du signe du résultat. On peut constater que la multiplication nécessite de nombreux calculs identiques mais avec des modes d'arrondi différents. En étudiant les résultats en fonction du signe des entrées, nous avons ramené le nombre d'opérations min/max de 14 à 6 [CFD08].

Branchements. L'implantation de l'arithmétique d'intervalle sur GPU est nécessairement différente de celle sur CPU, dans la mesure où les branchements divergents sont coûteux. Aussi, lorsqu'il y a une forte probabilité de rencontrer un branchement divergent, nous avons utilisé des alternatives nécessitant moins de branchements.

Troncature dans les multiplications. Comme nous l'avons évoqué en section 1.2, les premiers GPU ne se conformaient pas au standard IEEE-754. Par exemple, ils fusionnaient quasi automatiquement les opérations de multiplication et d'addition en une opération MAD. Si le code résultant est plus concis et rapide, il est aussi moins précis qu'une multiplication suivie d'une addition. Ce problème a été résolu pour les générations de GPU qui ont succédé aux cartes Nvidia GeForce 8.

Nous noterons cependant que, sur les Geforce 8, le résultat de la multiplication du MAD était tronqué avant d'être intégré dans l'addition qui elle, était arrondi selon le mode choisi. Si ce comportement est problématique pour de nombreux algorithmes numériques, ce n'était pas le cas de notre implantation basée sur un mode d'arrondi par troncature (arrondi vers zéro).

Dernières générations

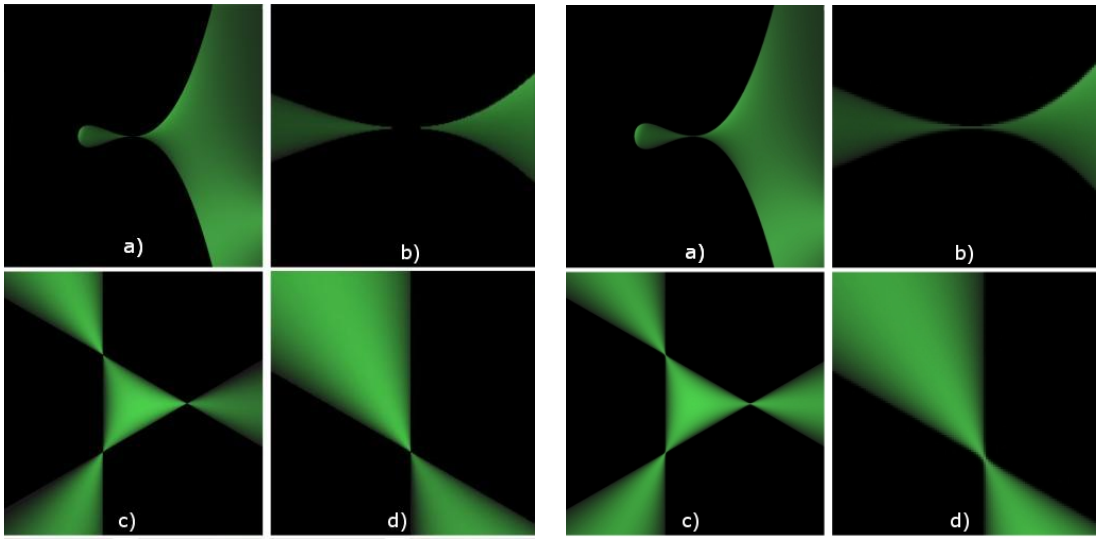
Les dernières générations de GPU Nvidia offrent l'arrondi correct pour l'addition et la multiplication dans les deux modes d'arrondi dirigés. A partir des GT200 (Compute capability 1.3), tous les modes d'arrondi définis par le standard IEEE-754 sont supportés pour la double précision. Ce support a été étendu à la simple précision à partir des architectures Fermi (compute capability 2.0). Contrairement à la majorité des CPU qui stockent le mode d'arrondi sous la forme d'un état global du processeur, les GPU Nvidia intègrent le mode d'arrondi dans l'instruction. On parle ainsi de gestion statique du mode d'arrondi, conformément à la norme IEEE-754 :2008.

En CUDA, ces modes d'arrondis sont accessibles à travers un attribut au niveau du langage bas niveau (PTX) ou par le biais de fonctions intrinsics. Le surcoût du changement des modes d'arrondis est ainsi complètement éliminé rendant les implantations d'IA sur GPU simples et efficaces.

2.2.3 Application : lancé de rayon certifié

L'arithmétique d'intervalle peut se révéler très utile dans de nombreux cas, dont celui du lancé de rayon sur une surface implicite. Dans ce cas, lorsque les rayons s'approchent de singularités décrites par les équations régissant les surfaces implicites, les rayons peuvent passer à côté de la surface et ne rien afficher.

Nous avons testé avec J. Flórez dans [CFD08] la bibliothèque d'IA sur GPU pour la résolution du problème de lancé de rayon fiable décrit en détails dans [57]. Les tests ont porté sur les équations régissant la goutte d'eau (figure 2.1a) et tritrumpet (figure 2.1c). Sur ces figures, on observe que les fines liaisons entre les formes sont correctement rendues à l'aide de l'arithmétique par intervalle. La version sur GPU était entre 100 et 300 fois plus rapide que la version CPU principalement grâce à notre implantation d'IA sur GPU.



Sans arithmétique d'intervalle — Avec arithmétique d'intervalle

FIGURE 2.1 – Comparaison des surfaces rendues par la technique du lancé de rayon sur des surfaces implicites avec et sans arithmétique d'intervalle.

2.3 Arithmétique floue

Nous avons vu dans la section précédente que l'arithmétique d'intervalle permet de gérer un unique degré d'incertitude. Ce niveau d'incertitude peut être insuffisant dans des domaines tels que la finance [157], le transport [23], la gestion de chaînes de traitement [34] ou les réseaux électriques [100, 127]. Dans ces systèmes, la base de connaissance peut être ambiguë et imprécise. L'information y est généralement représentée sous la forme de phrases ambigües telle que *au moins x*, *probablement entre 0.4 et 0.7*... Cette forme d'incertitude se modélise à l'aide de l'arithmétique floue, qui permet de gérer ce type de quantificateur [159]. Elle est basée sur une représentation par α-cut qui fait appel aux opérations de l'arithmétique par intervalle [50]. Cependant, le coût de cette arithmétique est fonction de la précision du modèle (nombre d'α-cut) ce qui la rend coûteuse en calculs et en mémoire.

2.3.1 Définitions

A chaque nombre flou $\tilde{p}$ est associé une fonction d'appartenance, $\mu_{\tilde{p}}$, qui assigne à chaque valeur de l'espace des réels un degré d'appartenance. Considérons les nombres flous pour lesquels il existe une unique valeur $m \in \mathbb{R}$ pour laquelle $\mu_{\tilde{p}}(m) = 1$, correspondant au plus haut degré d'appartenance, et dont la fonction d'appartenance est monotone autour de cette valeur. ($\mu_{\tilde{p}}(x)$ est croissant pour $x < m$ et décroissant pour $x \geq m$). Alors un nombre flou est défini par :

$$\tilde{p} = \{(x, \mu_{\tilde{p}}(x)) \mid x \in \mathbb{R}\}$$

Un α-cut de $\tilde{p}$ correspond à l'ensemble des éléments dont le degré d'appartenance est supérieur à un α_i donné :

$$p_{\alpha_i} = \{x \in \mathbb{R} \colon \mu_{\tilde{p}}(x) \geq \alpha_i\}$$

Les opérations d'arithmétique floue sont réalisées dans ce format en réalisant les opérations sur les α-cuts de même niveau. Soit $\tilde{p}$ et $\tilde{q}$ des nombres flous avec k niveaux d'incertitude, p_{α_i} et q_{α_i}, pour $i \in \{0, \ldots, k-1\}$, leurs k α-cuts respectifs, et $\oplus \in \{+, -, \times\}$, alors :

$$\tilde{r} = \tilde{p} \oplus \tilde{q}$$

est évalué à l'aide de k opérations d'arithmétique par intervalle :

$$r_{\alpha_i} = p_{\alpha_i} \oplus q_{\alpha_i}$$

pour $i \in \{0, \ldots, k-1\}$.

2.3.2 Optimisation de la représentation et des calculs

Une des caractéristiques importantes des nombres flous est la forme de la fonction d'appartenance. En fonction de l'application, les nombres peuvent avoir une fonction d'appartenance symétrique, de sorte que tous les α-cuts soient centrés autour du même point [33, 36]. La figure 2.2 montre un exemple classique de fonction d'appartenance suivant une loi gaussienne centrée autour de m.

Dans [DM14], nous avons introduit le concept de nombres flous symétriques, sur lesquelles nos opérations s'appliquent.

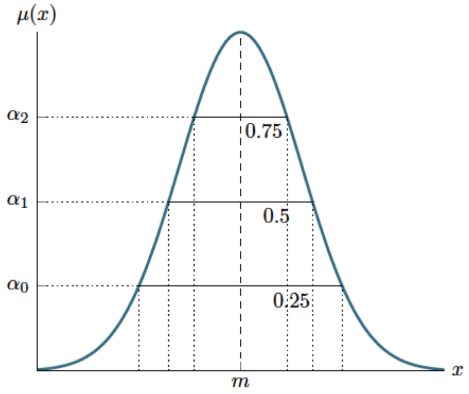


FIGURE 2.2 – Nombre flou symétrique et les α-cuts correspondant centrés autour du même point.

Definition (Nombre flou symétrique)**.** *Soit $\tilde{p}$ un nombre flou, $\mu_{\tilde{p}}$ sa fonction d'appartenance et $m \in \mathbb{R}$ tel que $\mu_{\tilde{p}}(m) = 1$. Si $\mu_{\tilde{p}}$ est symétrique autour de m, c'est-à-dire :*

$$\mu_{\tilde{p}}(m - x) = \mu_{\tilde{p}}(m + x), \forall x \in \mathbb{R},$$

alors $\tilde{p}$ est un nombre flou symétrique.

Si ces nombres ne présentent que peu d'intérêt pour une implantation d'arithmétique par intervalle basée inf-sup, ce n'est pas le cas pour une représentation centre-rayon. Dans ce cas, lorsque les nombres flous manipulés sont symétriques, il est alors possible de factoriser le centre entre tous les intervalles représentant les différents α-cut. Nous avons démontré que la propriété de symétrie est préservée par l'addition et la multiplication. Elle ne l'est en revanche pas pour la division.

Nous avons implanté une bibliothèque d'arithmétique floue en C++ et CUDA [DM13a], dont les opérations et types sont accessibles aussi bien à partir du CPU que du GPU. Si x est le nombre d'α-cuts, ce format de représentation réduit l'empreinte mémoire d'un nombre flou de $2x$ flottants pour une représentation classique à seulement $(1+x)$ nombres flottants. De manière similaire, on divise le nombre d'opérations élémentaires pour la multiplication par un facteur 3. Lors de nos tests, nous avons montré que ces optimisations de format combinées à l'utilisation des modes d'arrondi statiques des GPU permettent de réduire grandement le coût de l'arithmétique floue.

2.4 Arithmétique logarithmique

Lorsque la dynamique de représentation offerte par la virgule flottante est insuffisante comme c'est le cas dans le calcul de probabilité dans les réseaux bayésien ou en phylogénie, il est nécessaire d'utiliser un système de représentation adapté à la représentation de très grand ou très petite valeurs. Le système de représentation logarithmique des nombres (LNS) permet de représenter un nombre X par son logarithme en base b, $x = \log_b |X|$, plutôt que d'utiliser une représentation à base de mantisse et d'exposant. x peut être représenté en virgule fixe pour des raisons de rapidité d'exécution des opérations, ou en virgule flottante afin d'augmenter significativement l'espace de représentation des nombres.

La multiplication, division, racine carré et mise à la puissance sont réalisées trivialement dans le système LNS. A l'inverse l'addition et la soustraction nécessitent de calculer les fonctions

$s_b(x) = \log_b(1+b^x)$ et $d_b(x) = \log_b|1-b^x|$. Aussi, l'évaluation de ces deux fonctions $s_b(x)$ et $d_b(x)$ est indispensable pour calculer en LNS mais se révèle très coûteuse, surtout lorsque le format de représentation utilise des nombres flottants. Par exemple, la somme de deux nombres en système LNS repose sur l'identité $Z + Y = Y(1 + Z/Y)$, et évalue la somme comme $\log(Z + Y) = y + s_b(z - y)$.

2.4.1 Implantation sur GPU

Une implantation triviale du système LNS sur GPU serait d'évaluer les fonctions s_b et d_b à l'aide des nombreuses unités de calcul flottant présentes dans les GPU. Cependant, ceux-ci disposent d'autres unités comme celles d'interpolation de texture (Section 1.1.4). Ces unités restent généralement inexploitées lors de calculs généralistes.

2.4.2 Evaluation polynomiale par filtrage bilinéaire

Nous avons proposé dans [ACD10] une méthode pour réaliser une évaluation polynomiale à l'aide de ces unités de texturage appliquée aux fonctions $s_b(x)$ et $d_b(x)$ décrit en section 1.1.4.

La fonction cible $f(x)$ est approchée par une évaluation polynomiale de degré 2 par morceau sur l'intervalle $[0, 1]$ à l'aide de k polynômes. Soit P_w le polynôme utilisé pour l'approximation de $f(x)$ sur l'intervalle $[w.2^{-k}, (w + 1).2^{-k}]$:

$$P_w(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2. \tag{2.1}$$

L'originalité de cette approche est de proposer une méthode pour définir correctement les paramètres et adresses de texturage T à utiliser de sorte que l'équation du filtrage bilinéaire (eq. 1.1, p. 5), soit similaire à celle des polynômes P_w.

Ainsi, si nous effectuons un chargement de texture aux coordonnées :
$(x + 0.5, (x - floor(x)) + 0.5)$ et l'équation (1.1, p. 5) du filtrage devient équivalente à :

$$\begin{aligned} V &= (1-\gamma)^2 \cdot T[i,0] + (\gamma - \gamma^2) \cdot T[i+1,0] \\ &\quad +(\gamma - \gamma^2) \cdot T[i,1] + \gamma^2 \cdot T[i+1,1] \end{aligned} \tag{2.2}$$

avec $\gamma = frac(x)$, $i = floor(x)$

Si la texture T stocke les valeurs $T[i,0] = a_0$, $T[i+1,0] + T[i,1] = 2 \cdot a_0 + a_1$ et $T[i+1,1] = a_0 + a_1 + a_2$, alors nous avons :

$$\begin{aligned} V &= (1-\gamma)^2.a_0 + (\gamma - \gamma^2).(2.a_0 + a_1) \\ &\quad +\gamma^2.a_0 + a_1 + a_2 \\ &= a_0 + a_1.\gamma + a_2.\gamma^2 \end{aligned} \tag{2.3}$$

Ainsi, $T[i,0]$, $T[i+1,0]$, $T[i,1]$ et $T[i+1,1]$ sont utilisés pour conserver les coefficients d'un polynome. Ceux-ci seront stockés dans une texture de taille $2k \times 2$ pour k approximations polynomiales de la fonction $f(x)$ sur k intervalles.

Nous noterons que ces unités sont conçues pour travailler sur des textures à 4 composantes. Aussi le coût d'évaluation d'une fonction est sensiblement identique à celui de 4 fonctions différentes. Sans rentrer dans les détails, l'implantation proposée nécessite des alignements dans le calcul des adresses. Ces alignements ne sont pas standard et ne peuvent être pris en charge par les unités de calcul d'adresses propres aux unités de texturage. Ils doivent être effectués à l'aide des unités de calcul flottant, diminuant sensiblement l'efficacité de la méthode proposée. Cette solution s'adresse à l'évaluation de fonctions pour de petites précisions car les unités de

texturage travaillent sur des mots en virgule fixe. En effet, la principale difficulté de la méthode réside dans le sous-échantillonnage des mots en entrée de l'unité de filtrage.

Ce travail démontre l'intérêt d'avoir accès au paramétrage des unités d'interpolation comme celles décrites dans [107] afin de pouvoir évaluer en matériel n'importe quelle fonction. Il est cependant clair que les détails d'une telle unité paramétrable basée sur une évaluation polynomiale rapide reste à définir.

3

Considérations architecturales

Dans ce chapitre, nous aborderons la problématique du calcul au niveau architectural, c'est-à-dire celle de l'intégration des unités de calcul dans le processeur. Pour cela, nous commencerons par des discussions autour de la simulation fonctionnelle et montrerons comment elle peut aider à proposer des solutions microarchitecturales originales, que ce soit au niveau des registres (section 3.2) ou de la gestion des branchements (section 3.3). Nous terminerons ce chapitre par des propositions de nouvelles unités d'évaluation (section 3.4) afin d'accélérer le calcul.

Liste des publications relatives à ce chapitre : [CDP09b, CDDP10, CDDO08, CDZ09, GD06a, GD06b, Def05a, GD05a, GD05b, DG04].

3.1 Simulation architecturalle

3.1.1 La simulation multicœur

L'apparition de simulateurs de processeurs pour les architectures superscalaires dans les années 1990 fut le point de départ de nombreux travaux de recherche académique et industrielle en architecture des processeurs. La granularité de simulation dépend de 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 de gros volumes de communication entre les modules du simulateur pour 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 rapides. Les simulateurs les plus rapides sont les simulateurs au niveau fonctionnel, qui reproduisent simplement le comportement d'un processeur sur un code donné. De tels simulateurs peuvent intégrer des modèles de consommation ou de performance afin d'obtenir des estimations rapides d'un code ou d'une architecture.

Le simulateur au niveau cycle SimpleScalar [15] est à l'origine d'un grand nombre de travaux qui ont accompagné le succès des processeurs superscalaires à la fin des années 1990. Ce simulateur était connu pour son manque d'organisation, la difficulté d'en modifier le comportement et plus particulièrement pour réaliser des simulations d'architectures multicœurs. Afin de pallier à ces problèmes, des alternatives ont été proposées dans le domaine de la simulation d'architectures multicœurs [97] ou de systèmes complets [22, 95, 123].

Le développement de simulateurs bénéficie aujourd'hui de nombreux environnements dédiés et modulaires [13, 14, 112]. Ces environnements sont qualifiés de modulaires car ils permettent la construction de simulateurs à partir de blocs logiciels correspondant à des blocs matériels. Ces environnements diffèrent sur le degré de *modularité*, la richesse des *outils* proposés et leurs

performances. Certains imposent un protocole de communication pour distribuer la logique de contrôle dans les modules comme dans LSE [14], MicroLib [112] et UNISIM [13].

La simulation de processeurs plus spécifiques aux GPU et à leurs pipelines graphiques, peut s'appuyer sur des simulateurs comme Attila [102] pour la simulation au niveau cycle ou Qsilver [137] pour la simulation au niveau transaction. Plus récemment, des simulateurs de processeurs graphiques pour le GPGPU basé sur le langage intermédiaire NVIDIA PTX ont été proposés avec gpgpu-sim [17] et Ocelot [49].

3.1.2 Barra, un simulateur de GPU Nvidia

Il existe plusieurs possibilités pour reproduire le comportement d'un programme CUDA. Dans les premières version de CUDA, il était possible d'émuler sur CPU l'exécution d'un programme CUDA sans avoir nécessairement de processeur graphique compatible CUDA. Dans cet environnement, les threads GPU étaient tout simplement remplacés par des threads POSIX. Ce mode d'exécution permettait aussi de déboguer les programmes, mais différait en plusieurs points de l'exécution sur GPU (comportement flottant, ordonnancement, et organisation mémoire).

Les derniers GPU offrent désormais un mode de débogage qui permet d'observer les états du GPU entre l'exécution de chaque instruction. Si ce mode offre une exécution fonctionnelle précise pour une architecture donnée, il n'est pas possible d'en altérer l'exécution pour instrumentation ou permettre l'évaluation de nouvelles fonctionnalités.

Nous avons donc proposé un simulateur fonctionnel du jeu d'instructions NVIDIA Tesla avec le simulateur Barra [CDP09b, CDDP10]. Le comportement est reproduit avec une précision au niveau bit, à l'exception des fonctions exp, log, sin, cos, rcp, rsq.

Ce simulateur se compose de deux parties : le driver et le simulateur. Le driver est une bibliothèque partagée avec le même nom et qui propose les mêmes symboles que la bibliothèque *libcuda.so* de sorte que les appels du driver NVIDIA puissent être dynamiquement redirigés vers le simulateur. Cela inclut toutes les fonctions principales de chargement et d'exécution d'un programme CUDA ainsi que de transfert mémoire. Le simulateur prend en entrée un programme binaire généré avec le compilateur nvcc et simule l'exécution du kernel afin de produire des statistiques d'exécution pour chaque instruction.

Lors des tests de temps de simulation, nous avons remarqué qu'une exécution sur un GPU de type 9800GX2 était entre 150 et 10 000 fois plus rapide qu'une simulation avec Barra sur un Core2 Duo E8400, avec une moyenne géométrique de 1 050 sur les programmes du SDK CUDA. Cette vitesse d'exécution était du même ordre de grandeur que celle du mode émulation proposé par le compilateur CUDA (moyenne de 1 150), basé sur une exécution parallèle par *pthreads* de code natif.

Le simulateur Barra exécute chaque instruction assembleur selon un pipeline décrit en détail dans [CDP09b]. Premièrement, un ordonnanceur sélectionne un warp prêt à être exécuté avec une politique d'ordonnancement de type round-robin, ainsi que le compteur de programme associé. L'instruction correspondante est ensuite chargée et décodée. Les opérandes sont ensuites lus à partir du banc de registres, de la mémoire partagée ou du cache de constantes. Les instructions sont ensuites exécutées et les résultats réécrits en registre.

Gestion des registres. Les registres généraux sont partagés entre les instances des threads s'exécutant sur un multiprocesseur donné, ce qui permet d'exploiter du parallélisme de données au prix de plus de registres. Barra maintient différents états pour chaque warp actif du multiprocesseur. Ces états correspondent au compteur de programme, aux registres d'adresse et de prédication, aux piles de masques et d'adresses, à la fenêtre des registres assignés au warp, ainsi

que la fenêtre sur la mémoire partagée. Dans le cadre d'une simulation purement fonctionnelle, les warps sont ordonnancés selon une politique round-robin.

Le multiprocesseur des GPU de type TESLA dispose d'un banc de registres multi-bancs. Ce banc est partitionné entre les warps en utilisant un schéma complexe décrit dans [89] afin de minimiser les conflits de bancs et de permettre un regroupement plus efficace. Cette politique d'allocation des registres n'a pas d'influence sur le comportement du simulateur fonctionnel à l'exception du nombre exact de warps pouvant partager le banc de registres. Afin d'améliorer l'efficacité du simulateur et de simplifier le fonctionnement du banc, l'algorithme utilisé dans Barra repose sur une allocation séquentielle de blocs de registres dans un banc de registres unifié.

Ordonnancement des warps. Chaque warp a son propre drapeau qui définit si celui-ci est prêt à être exécuté. Au début de l'exécution d'un bloc de thread, chaque warp a son drapeau positionné sur *actif* et celui des autres warps est positionné sur *inactif*. À chaque étape de la simulation, un warp marqué comme actif est sélectionné en utilisant un algorithme round-robin pour l'exécution de l'une de ses instructions.

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. Lorsque tous les warps des blocs à exécuter ont atteint l'état *inactif*, un jeu de nouveaux blocs est ordonnancé sur le multiprocesseur. Nos tests ont montré que les GPU NVIDIA se comportent également de cette manière, sans recouvrement entre l'exécution des jeux de blocs successifs.

3.2 Le banc de registres

Le banc de registres pose un réel problème conceptuel aussi bien pour des architectures superscalaires que pour des architectures vectorielles. Ce problème vient de la surface et de la consommation du banc de registres. La surface occupée par le banc de registres est fonction du nombre de registres, de la largeur de mot et surtout du carré du nombre de ports. La consommation évolue, elle, de façon cubique par rapport au nombre de ports. En effet, l'ensemble des ports se présente sous la forme d'une matrice dont les lignes sont les commandes d'accès, les colonnes forment les données et la diagonale contient les transistors. Ajouter un port étend la matrice d'une ligne et d'une colonne.

Le banc de registres n'est pas la seule structure critique. Les stations de réservation, qui indiquent pour chaque instruction si ses sources sont disponibles, contiennent des comparateurs chargés de repérer les destinations des résultats calculés par les unités fonctionnelles [108]. Le nombre de tels comparateurs varie en fonction de deux fois le nombre de résultats produits par cycle multiplié par le nombre de stations. Là encore, l'augmentation est quadratique par rapport au nombre d'opérations.

Pour diminuer le nombre de comparateurs, on peut remarquer que peu d'instructions attendent deux sources. [51] propose de ne laisser qu'un seul comparateur par instruction. On divise ainsi par deux le nombre total de comparateurs.

Une autre possibilité est de hiérarchiser les stations en séparant par exemple les instructions en fonction de leur latence [86,99]. Cela permet de ne laisser qu'une partie réduite des instructions

dans les stations de premier niveau. Le nombre de comparateurs devient ainsi indépendant du nombre total de stations.

Une autre voie pour simplifier le mécanisme de réveil des instructions [118, 130, 153] est de remplacer les comparateurs de stations par des vecteurs de successeurs. Chaque instruction désigne ses successeurs par un vecteur booléen. Pour une fenêtre de n instructions, l'ensemble des vecteurs comprend n^2 bits. [118] propose une méthode pour réduire le nombre de vecteurs. A chaque terminaison d'instruction, on propage l'arrivée de son résultat à l'ensemble des successeurs grâce au vecteur.

3.2.1 Optimisation de la latence

Pour les architectures superscalaires de type CPU, un des objectifs est d'augmenter la taille du banc de registres tout en conservant un temps d'accès aux registres faible. Or, un processeur doté de p opérateurs doit, à chaque cycle, lire $2p$ sources et écrire p résultats dans l'un des 2^n registres de renommage. Chaque registre doit ainsi avoir $2p$ ports de lecture et p ports d'écriture. A titre d'exemple, le banc de registres d'un Pentium 4 [71] (128 registres de 32 bits avec 18 ports d'accès) est accédé en deux cycles.

Par conséquent, si l'on souhaite doubler le nombre de ports d'un banc de registres, par exemple pour doubler le degré superscalaire du processeur tout en conservant un temps d'accès aux registres constant, il faut diviser par quatre le nombre de registres du banc. Une solution proposée pour augmenter la taille du banc de registres sans dégrader le cycle est de le hiérarchiser [30, 44, 144, 158]. Un premier niveau sert de cache, le banc de registres formant un second niveau. Néanmoins, il faut noter que le premier niveau hiérarchique ne peut bénéficier d'aucune réduction sur le nombre de ports.

Un certain nombre de travaux ont été consacrés à la réduction du nombre de ports : en partitionnant le banc [78, 160], en limitant le nombre de lectures et d'écritures [61, 82] ou en combinant ces deux méthodes [109, 136]. Dans tous les cas, le nombre de ports est fonction du nombre d'opérations lancées par cycle. Plus il augmente et plus le nombre de ports nécessaires est élevé.

Nous avons proposé dans [GD06b], un mécanisme permettant de répartir les requêtes de lecture et d'écriture visant le banc de registres de renommage. La solution repose sur un partitionnement du banc de registres en autant de bancs que d'instructions lancées en parallèle. Cependant ce partitionnement déplace le problème vers la répartition des requêtes de lecture et d'écriture. Pour le résoudre, nous avons proposé une politique d'ordonnancement des registres de renommage permettant de contenir le nombre de ports nécessaire tout en conservant de bonnes performances. Cela passe, entre autre, par l'assignation d'une destination par banc et le placement des sources dans la même partition que les destinations dont elles dépendent.

3.2.2 Optimisation du débit

Le modèle d'exécution vectorielle repose sur la gestion concurrente d'un grand nombre de threads, ce qui nécessite des débits importants ainsi qu'un grand nombre de registres architecturaux. Ce sont eux qui, entre autre, déterminent le nombre d'instances d'un programme qui vont pouvoir s'exécuter simultanément en fonction du nombre de registres disponibles par rapport au nombre de registres nécessaires à l'exécution du programme. Par exemple le GT200 a 512 registres de 32×32bits par multiprocesseur. Comme dans le cas superscalaire, le banc de registres vectoriels doit être de grande taille, tout en conservant un coût en surface et en consommation raisonnable. Afin d'y parvenir, les bancs de registres sont structurés en plusieurs bancs (64 bancs

de 128 registres de 32 bits pour le GT200) pouvant être accédés de façon hiérarchique. Dans ce modèle, le nombre de conflits de bancs est minimisé en utilisant la régularité de l'ordonnancement des warps. Ainsi sur ce type d'architecture, les registres d'un warp donné sont placés dans le même banc ce qui nécessite de sérialiser les accès aux registres. Ce choix augmente la latence d'accès aux registres pour un warp donné, mais en simplifie l'allocation et introduit une gestion statique des conflits de bancs.

Une autre piste permettant de réduire la taille et la consommation d'un banc de registres vectoriels est de s'inspirer des machines CRAY qui dissocient les traitements scalaires des traitements vectoriels. Si les derniers GPU d'AMD le font, ce n'est pas le cas des GPU Nvidia. Dans ce cas, ce sont alors les unités vectorielles qui travaillent avec ces données. Les unités vectorielles doivent alors exécuter la même instruction sur les mêmes données conduisant à une redondance aussi bien au niveau du stockage que des opérations. Ces opérations superflues ont un coût aussi bien en transfert de données qu'en énergie consommée. A l'aide du simulateur Barra, nous avons analysé dans [CDZ09] les données transitant par les registres vectoriels dans le cas d'une architecture Nvidia et nous avons observé qu'en moyenne 44% d'entres elles étaient identiques ou fortement corrélées. Ces valeurs remarquables peuvent faire l'objet d'un traitement à part, permettant de réduire significativement la taille et la consommation du banc de registres. Cependant, dans le contexte des architectures vectorielles, il reste à démontrer que le gain obtenu est suffisant pour justifier une augmentation de la complexité matérielle liée à la gestion des cas scalaires.

3.3 Les branchements

Les instructions de contrôle de flot ont un rôle important dans un modèle de programmation généraliste. Elles permettent au programmeur de construire des séquences de code à exécution irrégulière comme des boucles ou des séquences d'instructions dont l'exécution est conditionnée (*si-alors-sinon*). L'implantation des instructions de contrôle de flot se fait généralement par l'utilisation d'instructions de branchements conditionnels mises à disposition par le matériel. Celles-ci sont déterminantes pour la performance des applications.

3.3.1 Gestion dans le cas superscalaire

Les processeurs généralistes de type *superscalaire* accordent une part importante, en terme de transistors, à la mise en œuvre de ces instructions via la prédiction de branchement. L'effet des mauvaises prédictions peut être diminué en utilisant par exemple un Branch Target Buffer multi-niveau situé en dehors du chemin de chargement des instructions comme dans [119] ou sur du prefetching pour masquer les latences en cas de miss [35, 115, 120, 148].

La gestion des branchements est en effet cruciale dans les architectures superscalaires car les branchements cassent la nature séquentielle du chargement des instructions. Des solutions impliquant les compilateurs [31, 55], ou la microarchitecture [11, 65, 96] ont été proposées pour en réduire l'effet. Parmi elles, le cache de traces [111, 124] est une des solutions que l'on retrouve par exemple dans le Pentium 4.

Le rôle du cache de traces est d'alimenter le pipeline d'exécution avec suffisamment d'instructions en éliminant les incertitudes induites par les instructions de contrôle de flot. Le cache de traces y parvient en stockant la séquence d'instructions exécutée dynamiquement, ce qui inclut les branchements pris. Les séquences d'instructions reposent sur la notion de bloc de base (BB) défini dans ce contexte comme :

1. un BB a au plus une instruction de contrôle.
2. si un BB a une instruction de contrôle, alors c'est la dernière.
3. un BB a au plus n instructions consécutives, avec n la taille d'un bloc dans le cache de traces.

Le cache de traces présente cependant un certain nombre de défauts. Le premier est lié à la troisième règle définissant un BB, qui garantit qu'un BB doit toujours être de taille inférieure à la taille d'une ligne du cache de traces. Une conséquence est que deux BB ne peuvent pas se trouver simultanément dans le cache si ceux-ci partagent le même préfixe. C'est un problème dans le cas des branchements équilibrés. Le deuxième problème est que des BB complets sont chargés, même si le prédicteur ne souhaite pas utiliser la deuxième partie du BB induisant une consommation de bande passante inutile. Le troisième problème est que, pour des raisons de performance, le cache de traces ne stocke qu'une ligne de traces par ligne de cache introduisant des trous en fin de ligne. Enfin le cache de traces n'est pas capable de créer les lignes de traces qui traversent les frontières de saut indirect.

En partant du constat qu'un banc de registres est plus efficace qu'un cache L1, on peut envisager de placer un banc de registres dédié aux instructions entre le cache d'instructions et la partie décodage du pipeline. C'est ce que nous avons utilisé avec B. Goossens dans [GD05a] pour reconstruire les BB dynamiquement sur le modèle du cache de traces. Cette solution offre deux avantages. Premièrement, elle permet d'extraire et de fusionner les BB à la volée, éliminant ainsi la redondance des traces. Deuxièmement, elle permet de paralléliser les accès au cache avec le chargement des instructions.

3.3.2 Gestion dans le cas vectoriel

La problématique des branchements dans le cas vectoriel est légèrement différente. Dans le modèle Single Instruction Multiple Threads (SIMT), les branchements permettent à plusieurs *threads* au sein d'un même warp partageant le même flot d'instructions de s'exécuter suivant différents chemins. Ce problème est connu sous le nom de *divergence des branchements*. Cette divergence impacte négativement les performances d'un programme. Il est possible d'en diminuer le coût en réorganisant dynamiquement les threads entre les warps afin de conserver des comportement réguliers [58, 79].

Cette divergence repose sur une désynchronisation des threads. Elle permet une exécution séquentielle de chaque branche du code en sélectionnant les threads actifs par prédication. Les implantations matérielles pour les GPU sont à rapprocher des implantations plus anciennes utilisées dans les architectures SIMD [93].

Nous avons montré dans [CDDP09] que l'algorithme utilisé sur les GPU Nvidia est très proche de celui décrit dans le brevet [43]. Pour chaque *warp*, un multiprocesseur maintient un masque indiquant les threads actifs, un compteur de programme actif ainsi qu'une pile permettant de mémoriser les informations relatives aux désynchronisations successives.

Les GPU Intel GMA [74] offrent directement des instructions de contrôle de flot explicites reflétant les structures usuelles des langages de programmation : `if`, `else`, `endif`, `do`, `while`, `break`, `continue`. Leur sémantique est décrite en terme d'opérations sur le masque courant et deux piles de masques, servant respectivement aux blocs conditionnels (`if-then-else`) et aux boucles. Les adresses de saut étant toutes présentes explicitement dans les instructions de contrôle de flot, il n'est pas nécessaire de maintenir de pile d'adresses à jour comme dans l'implantation de Nvidia. En contrepartie, il n'existe pas de gestion matérielle des appels de fonctions (`call` et `return`).

Les programmes exécutés par les GPU R600 et R700 d'AMD [10] sont composés de trois flots d'instructions distincts : un flot d'instructions arithmétiques, un flot d'instructions d'échantillonnages de textures et un flot d'instructions de contrôle qui se charge d'orchestrer l'exécution de l'ensemble. De manière similaire à l'implantation d'Intel, des instructions spécifiques correspondent directement aux structures de contrôle du langage C et des différents langages de shaders.

Avec ces solutions, le matériel n'est pas capable de déterminer les points de reconvergence et impose leurs insertions dans le code par le compilateur. Nous avons proposé dans [CDDP09] une méthode qui permet de s'abstraire de ces annotations et d'exécuter par exemple directement du code Single Program Multiple Data (SPMD) compilé par un compilateur traditionnel. Pour cela, nous utilisons deux piles de masques implantées par des compteurs et deux piles d'adresses, l'une servant aux structures conditionnelles et l'autre aux structures itératives. Ces piles d'adresses sont utilisées pour gérer la divergence ainsi que pour déterminer les points de reconvergence et ne nécessitent donc pas d'annotation du code. Elles gèrent efficacement les structures de contrôle strictement imbriquées, ainsi que les instructions de contrôle de flot plus irrégulier (ex : `break` et `continue`). Il s'avère que lors des tests, ce type d'algorithme est toujours au moins aussi efficace que l'algorithme de reconvergence explicite, tout en ayant un coût matériel raisonnable.

3.4 Opérateurs spécialisés

Les architectures hautes performances actuelles reposent sur des techniques logicielles ou matérielles pour exploiter le parallélisme d'instructions des applications. Avec l'augmentation du budget transistors, il est possible d'envisager de nouvelles solutions améliorant la performance des applications limitées par le matériel. Deux alternatives existent du point de vue calcul. Soit il est possible de multiplier les unités de calcul, soit d'en proposer de nouvelles adaptées aux applications. Les GPU combinent les deux approches.

3.4.1 Mécanisme de fusion pour la réalisation d'opérateurs ternaires

Proposer de nouvelles unités répond donc à une logique destinée à améliorer les performances d'un programme. Cependant, ces unités se doivent d'être les plus générales possible. Nous avons donc exploré la piste des opérations ternaires consistant à fusionner 2 opérations flottantes en une seule. Cette solution présente plusieurs avantages :

- Elle réduit les erreurs d'arrondis dans le cas d'opérations flottantes.
- Elle diminue la latence globale d'un programme en réduisant la longueur du graphe de dépendance.
- Elle diminue la pression sur le banc de registres, en nécessitant moins de registres temporaires.
- Elle réduit la taille du code généré.

Le principal inconvénient de cette solution est de nécessiter des opérateurs matériels dont une partie restera sous-utilisée dans le cas où l'une des deux opérations n'est pas utilisée.

L'exemple du Fused Multiply and Add ($y = a+(b*c)$) permettant avec une seule opération de réaliser 2 opérations de base avec un seul arrondi final est représentatif de cette problématique. Il n'est pas le seul puisque des techniques similaires avaient déjà été proposées par Philips et Vassiliadis dans [114, 147] pour de l'arithmétique à virgule fixe. Cette étude avait été étendue aux opérations composées de 3 instructions dans [131].

Deux opérations peuvent être fusionnées s'il existe une dépendance de données vraie entre deux instructions. Cette dépendance de type producteur/consommateur se retrouve par exemple

dans la séquence suivante.

1. F1 = F2 * F3
2. F4 = F1 + F5

Si sur cet exemple l'addition et la multiplication prennent chacune 5 cycles, alors le résultat final ne serait disponible qu'après 10 cycles contre 5 pour un opérateur fusionné. Le bénéfice est donc immédiat et ce type de fusion est valable pour n'importe quel type d'instruction (arithmétique, mémoire, etc ...).

Pour être fusionnable, les opérations doivent respecter un certain nombre de contraintes. Premièrement, elles doivent appartenir à un même bloc de base. Un bloc de base est constitué par des instructions consécutives sans instruction venant casser le flot (ex : saut, branchement). Deuxièmement, le résultat intermédiaire doit être réutilisé par au plus une opération.

Nous avons considéré la fusion des opérations flottantes de type addition, multiplication et division. Pour cela nous avons développé une plateforme de test afin de mesurer le gain potentiel de la fusion d'opérations. Cette plateforme partait du code assembleur Alpha [5] pour produire le graphe de flot d'instructions (DFG). Le travail était réalisé en trois phases. Une première phase collectait l'information des programmes à l'aide de l'outil Pixstat distribué avec Atom [53]. Le code assembleur était annoté d'informations statistiques sur le temps pris par chaque bloc. La deuxième phase consistait en la production du DFG à partir de la trace d'exécution. Enfin, le parcours du DFG combiné aux informations statistiques nous permettait d'analyser le gain potentiel lié à la fusion.

Les mesures réalisées sur un ensemble de 13 programmes tirés des SPEC2000fp et présentées dans la table 3.1 donnent le nombre d'opérations fusionnables. Les nombres en gras représentent les opérateurs fusionnés ayant le plus d'intérêt. Les opérateurs fusionnés sont représentés avec la syntaxe suivante : le premier opérateur correspond à la dernière opération effectuée, et le deuxième à la première opération (ex : le FMA est représenté par $+*$).

Name	++	+*	+/	*+	**	*/	/+	/*	//
168.wupwise	38	**74**	0	42	27	0	7	4	0
171.swim	180	**184**	0	162	48	0	10	7	0
172.mgrid	**249**	98	0	91	11	0	2	0	0
173.applu	**1826**	1783	86	1797	782	7	69	109	0
177.mesa	116	**274**	0	164	71	0	10	10	0
178.galgel	358	**856**	26	415	443	21	57	36	10
179.art	73	98	0	**100**	34	0	8	3	0
183.equake	52	**119**	3	75	74	8	10	8	1
188.ammp	260	**418**	2	291	205	5	12	19	0
189.lucas	80	**402**	0	87	54	2	2	2	0
191.fma3d	215	**461**	12	266	144	11	24	32	0
301.apsi	514	**1015**	175	881	673	83	192	302	14
Total	3961	**5782**	304	4371	2566	137	403	532	25

TABLE 3.1 – Potentiel de fusion des opérations flottantes dans les programmes SPEC2000fp.

Nous avons observé que, sans surprise, le FMA est l'opérateur fusionné le plus fréquent. Mais de façon plus surprenante, l'opérateur que nous avons appelé ADD2 (++) revêt un intérêt fort, puisque nous avons mesuré que 58% des instructions d'additions flottantes pouvaient être remplacées par l'opérateur ADD2. Ce taux était supérieur à celui du FMA. Nous avons, à partir

de cette étude, réalisé une première tentative d'implantation de l'opérateur ADD2 dans le rapport de recherche [DG04]. D'autres implantations de cet opérateur ont par la suite été proposées en FPGA dans [47,150]. Le premier résultat est donc de confirmer l'intérêt d'un opérateur ADD2 du point de vue applicatif. Comme pour le FMA, les opérateurs fusionnés changent le résultat final en introduisant une seule erreur d'arrondi. Il faut donc considérer la problématique de l'erreur dans sa globalité avant de fusionner. Sur ce point, C. Lauter a démontré l'intérêt numérique de cet opérateur pour l'évaluation de fonctions élémentaires dans sa thèse [85]. Nous noterons que cette problématique revêt aujourd'hui un caractère tout particulier dans le contexte de la reproductibilité numérique, puisqu'il permettrait de réduire les erreurs liées aux accumulations.

3.4.2 Unités de calculs spécialisées avec cache partagé

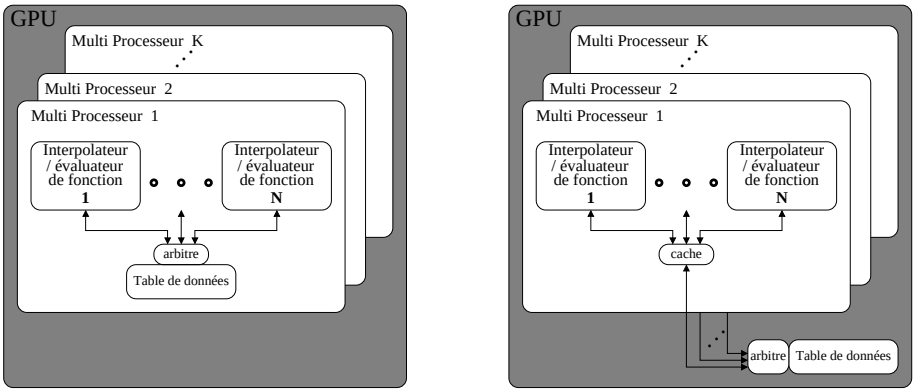
Les GPU disposent d'unités d'évaluation des fonctions de base qui ont en charge l'évaluation des fonctions inverse, de la racine carrée inverse, sinus, cosinus, exponentielle et logarithme en base 2. Leur évaluation en matériel repose sur l'utilisation combinée ou non d'un cache de données et d'une approximation polynomiale.

Le cache de données tel qu'il est utilisé dans les unités d'évaluation des fonctions de base permet d'accélérer les calculs en mémorisant des résultats déjà calculés. On parle de mémoïzation. Ce principe est exploité dans la *Tree Machine* [116] et il aboutit par exemple à un mécanisme de détection de calculs complexes redondants associé à un cache de résultats [122]. Lipasti *et al.* décrivent un mécanisme de cache de valeurs et de chargement du résultat prédit [90]. Une étude sur les caractéristiques du prédicteur utilisé pour le chargement de résultat prédit est donnée dans [145]. La mémoïzation peut diminuer la latence de l'évaluation en matériel de la division, de l'inverse et de la racine carrée [106]. On peut aussi utiliser le cache uniquement sur les premières étapes de la division en l'adressant avec les bits de poids fort du diviseur [20]. On obtient alors un compromis entre la taille du cache, son taux de réussite et la latence de la division. Si l'on s'autorise des résultats moins précis, on peut mettre en œuvre un « cache flou » qui renvoie un résultat stocké même si l'argument ne correspond pas exactement à l'entrée du cache [12]. Enfin, on peut étendre le jeu d'instructions des processeurs avec des instructions permettant la mémoïzation efficace de fonctions logicielles sans effet de bord comme la racine carrée et les fonctions élémentaires usuelles (exponentielle, logarithme, puissance, fonctions trigonométriques ...) [37].

Les processeurs graphiques actuels comptent plusieurs dizaines d'unités d'évaluation de fonctions de base. Pour accélérer les calculs, chacune de ces unités intègre une table adressée par les bits de poids fort de l'argument. Ces tables sont identiques entre chaque unité. Il est donc possible de partager ces tables entre toutes les unités. Cependant chaque unité peut accéder à n'importe quelle adresse de cette table. Cette contrainte impose d'avoir une mémoire avec autant de ports que d'unités, ce qui n'est pas envisageable.

Nous avons proposé dans [CDDO08] d'exploiter la localité de valeurs quand les valeurs passées en argument sont très proches sur un groupe de données traitées en SIMD. Cette forme de localité de valeurs permet d'envisager de nouvelles architectures où les tables utilisées pour l'approximation de ces fonctions sont mutualisées sans perte de performance entre les unités d'évaluation des fonctions de base. Les deux solutions proposées sont présentées dans la figure 3.1. La première solution utilise une table unique par bloc SIMD. La seconde solution propose un accès hiérarchique avec un cache par bloc SIMD et une table unique pour tout le processeur graphique.

Pour quantifier l'intérêt des deux architectures proposées, nous avons instrumenté les appels aux fonctions racine carrée, racine carrée inverse, exponentielles, logarithmes, sinus et cosinus



Solution 1 : Table partagée intra−multiprocesseur

Solution 2 : Table partagée inter−multiprocesseur

FIGURE 3.1 – Proposition de partage des tables entre les unités d'évaluation des fonctions élémentaires.

pour générer des traces d'exécution de programmes tirés des benchmarks SPECviewperf 8.1 [3], Spacetrack [146], GPU4RE [CDD08b], Comsol [1].

Sur l'ensemble des programmes testés, nous avons observé qu'en moyenne une requête ne concerne que 3.3 adresses différentes pour des tables composées de 64 entrées et partagées entre 16 voies SIMD. Ce type de table est équivalente à celle disponible dans les architectures NVidia GeForce 8. Cette moyenne tombe à 1.9 si l'on exclut les applications avec peu de localité car écrites pour des architectures sans unité d'évaluation de fonctions. (Ces programmes remplissent des tables d'évaluation de fonctions). Ces tests ont démontré qu'il était possible de se contenter d'une seule mémoire double port par bloc SIMD pour mémoriser la table nécessaire à l'évaluation des fonctions de base au lieu d'une mémoire par unité d'évaluation.

Nous avons poussé le raisonnement plus loin en essayant de mutualiser la table entre les blocs SIMD et un cache partagé par bloc SIMD pour différentes configurations de cache en nombre de lignes et associativité. Comme nous pouvions nous y attendre sur ce type d'architecture, différents appels non-corrélés se retrouvent en compétition dans le cache, menant à des conflits fréquents. Ces appels non-corrélés correspondent à des instructions différentes ou à des instructions traitant des données avec une très faible localité spatiale et donc une faible localité de valeurs dans un contexte de données graphiques (des pixels voisins ont souvent les même attributs). Ce phénomène rend les associativités les plus faibles peu efficaces. Les taux de succès des caches ne permettent donc pas d'envisager un partage hiérarchique efficace de la table entre les blocs SIMD.

4

Comportement du calcul

Dans ce chapitre, nous décrivons nos contributions à l'amélioration du calcul sous différents aspects : performance, consommation, fiabilisation et prédictabilité. Pour cela, nous discuterons de l'intérêt de la régularité dans les calculs en prennant l'exemple de l'algorithme du parcours d'arbre, dans la section 4.1. Nous évoquerons dans la section 4.2 le problème de la redondance dans les données présentes dans les registres vectoriels. Nous aborderons la question de la consomation énergétique et de l'impact de l'ordonnancement sur la consommation dans la section 4.3. Nous discuterons ensuite de la problématique du calcul fiable sur une architecture non-fiable en section 4.4. Enfin nous terminerons par une analyse de la prédictabilité et du déterminisme dans les processeurs graphiques en section 4.5.

Liste des publications relatives à ce chapitre : [Def14,DP13b,DP13a,DM13b,DM12,CDT09b].

4.1 Régularité dans les calculs

Il est apparu très tôt que les transistors dédiés aux calculs dans les GPU permettaient d'accélérer des applications régulières comme celles décrites dans [134]. Les applications irrégulières comme les opérations sur les graphes, comme le list ranking [152] ou connected components [68], ont été envisagées plus tardivement.

4.1.1 Etude d'une application irrégulière : le parcours d'arbre

Nous avons considéré dans [DM13b, DM12] une implantation des opérateurs *Treefix* utilisés dans les itérations de l'algorithme BFS de simulation de réseaux électriques [139] ou pour l'évaluation du score de parcimonie dans les arbres phylogénétiques [56,128]. Leiserson and Maggs [87] ont montré comment ils pouvaient être utilisés dans des opérations de traitement sur les graphes à l'aide des deux opérateurs *Rootfix* et *Leaffix* (figure 4.1). Rootfix renvoie pour chaque sommet de l'arbre le résultat d'une opération appliquée à chacun de ses ancêtres, Leaffix renvoie pour chaque sommet le résultat d'une opération appliquée à tous ses descendants. L'implantation de ces deux opérations repose sur un parcours de haut en bas ou de bas en haut de l'arbre.

Si l'opération binaire à appliquer est l'addition, alors nous pouvons définir les opérations *+Rootfix* et *+Leaffix*. On obtient ainsi comme résultat l'arbre dont les sommets correspondent respectivement à la somme de tous les ancêtres ou de tous les descendants de l'arbre pris en entrée. Dans le cas particulier où les sommets ont un poids de 1, +Rootfix renvoie pour chaque sommet sa profondeur et +Leaffix renvoie pour chaque sommet, le nombre de sommets dans le sous arbre dont il est à l'origine.

(a) +Rootfix (b) +Leaffix

FIGURE 4.1 – Exemple d'opération +Rootfix et +Leaffix.

Il existe deux façons de résoudre ce type de problème. Une solution basée sur l'équilibrage de charge et une autre basée sur l'approche vectorielle. Différentes implantations basées sur un équilibrage de charge sur GPU ont été proposées [48, 67, 72, 94, 98]. Les versions [48, 67, 72] examinent à chaque itération les nœuds dont les prédécesseurs ont été traités à l'itération précédente en parcourant l'ensemble des nœuds non encore traités du graphe. Cette solution peut, dans le cas d'une liste chaînée, nécessiter un nombre quadratique de tests par rapport au nombre de nœuds. [94,98] optimisent le nombre de tests à l'aide d'une frontière générée à chaque itération comportant les éléments à visiter lors de la prochaine itération. Si ces méthodes se révèlent efficaces du point de vue du nombre d'opérations, elles exhibent toutes des schémas de calcul irréguliers, complexes à mettre à œuvre et dont les performances sont très dépendantes de la topologie des graphes. Ces travaux diffèrent dans les stratégies utilisées pour parvenir à un équilibrage de charge adapté au fonctionnement du GPU.

4.1.2 Régularité des calculs vs réduction du nombre d'opérations

Nous avons voulu mesurer le bénéfice apporté par l'utilisation d'une structure basée sur la représentation eulérienne d'un arbre qui est plus favorable à un traitement sur une architecture vectorielle. Cette représentation permet alors d'utiliser les opérations parallèles classiques comme l'algorithme *scan*. Les opérations Rootfix et Leaffix sur un arbre à n sommets nécessitent alors $O(\log n)$ appels successifs à l'algorithme scan, et ce indépendamment de la topologie de l'arbre. Malheureusement ces méthodes utilisent des structures vectorielles nécessitant deux fois plus de données temporaires ainsi que deux fois plus de calculs que les solutions existantes basées sur l'équilibrage de charge.

En implantant sur GPU ces méthodes et en réalisant des tests sur des arbres extraits de vrais problèmes [9], nous avons pu montrer dans [DM13b] que les versions vectorielles, en plus de leur simplicité, donnaient des performances prédictibles car indépendantes de la topologie, ce qui n'était pas le cas des versions basées sur de l'équilibrage de charge. Nous avons aussi montré que ce type de solution était entre 2 et 5 fois plus performante que les solutions basées sur de l'équilibrage de charge, et ce malgré l'augmentation du volume de données et de la quantité de calculs intermédiaires. Ce facteur pouvait même monter jusqu'à 5000 dans des cas dégénérés comme les listes chaînées.

Cependant, des problèmes subsistent. En effet, si les opérations Treefix ne posent pas de problème de précision lorsque les données manipulées sont des entiers, ce n'est pas le cas avec des données flottantes. Dans le cas flottant, il faut nuancer ce gain en terme de performance et de simplicité par la perte potentielle de précision. La version vectorielle modifie l'ordre et le nombre des opérations. Ainsi, pour l'algorithme *+Rootfix* flottant, l'addition flottante n'étant pas associative, nous avons mesuré que la version vectorielle perdait en moyenne 2 bits de plus que la version basée sur de l'équilibrage de charge.

4.2 Régularité dans les données

Les architectures proposées par Nvidia sont des architectures vectorielles de type SIMD. Au niveau matériel, toutes les instructions, y compris les instructions mémoire et de contrôle, travaillent sur des vecteurs, et tous les registres architecturaux sont eux aussi des vecteurs. Bien que cela offre en théorie un environnement de programmation efficace en permettant de s'abstraire de la longueur des vecteurs SIMD et de permettre des implantations qui puissent facilement passer à l'échelle, cela peut devenir une source d'inefficacité importante lorsque les opérations à réaliser sont scalaires. Ce problème est similaire au problème de la localité de valeurs [18], c'est-à-dire, lorsqu'il existe une relation entre les valeurs pour des variables manipulées dans l'espace et le temps. Pour ce qui est des architectures vectorielles, cette corrélation se retrouve principalement au sein d'un même vecteur.

4.2.1 Données uniformes et affines

Un vecteur V est dit *uniforme* si toutes ses composantes sont égales ($V_i = x,\ \forall i$). Il existe deux sources possibles pouvant conduire à des valeurs uniformes. La première source est la diffusion d'une valeur lue depuis la mémoire (globale ou constante). La deuxième source de valeurs uniformes se trouve dans les conditions de contrôle. Par exemple, une boucle *for* avec une borne uniforme, utilisera un compteur lui aussi uniforme. Si les GPU modernes autorisent les conditions non-uniformes au sein d'un même vecteur, celles-ci conduisent généralement à des dégradations importantes de performance au sein d'un warp [105]. Ainsi les implantations optimisées ont des comportements uniformes entre les différentes voies de calcul d'un même warp.

De façon assez similaire, la bande passante se trouve fortement améliorée lorsque les accès mémoire respectent des motifs d'accès bien définis. C'est le cas des accès contigus ou sans conflits pour la mémoire partagée. Par exemple, l'architecture TESLA utilise les unités de calcul vectorielles pour calculer les adresses mémoires. Aussi les programmes se conformant aux recommandations faites par Nvidia pour ce qui est des accès mémoires opèreront sur des adresses consécutives, ce qui est une chose fréquente en programmation vectorielle. Ce dernier cas correspond à des motifs *affine*, car les données présentes dans le registre vectoriel V sont décrites par une fonction affine dépendant de l'indice de la donnée dans le vecteur : $V_i = x + iy$.

Il est possible de quantifier la proportion de données affines ou uniformes transitant par les registres vectoriels en observant le contenu de ceux-ci. Cette opération requiert un simulateur d'architecture comme Barra, présenté en section 3.1.2. Les résultats obtenus sur des exemples du SDK de CUDA sont donnés dans la figure 4.2. Ce graphique présente la proportion de vecteurs uniformes ou affines qui sont lus ou écrits dans le banc de registres. Les valeurs sont présentées sous la forme de pourcentage par rapport au nombre total de lectures ou écritures. On constate ainsi qu'en moyenne 27 % (44 %) des lectures du registre vectoriel sont uniformes (affines) et que 15 % (28 %) des écritures sont uniformes (affines). Cette proportion est plus que suffisante pour justifier de mécanismes d'optimisations.

4.2.2 Détection dynamique de valeurs remarquables dans les registres

En partant de ce constat, il est possible d'envisager des solutions combinant approche statique et dynamique capable de marquer les vecteurs en fonction du type de données qu'il contient (valeur uniforme, affine ou générique). Cette technique permet de réduire l'activité mémoire et des bus entre les unités fonctionnelles et les registres, dans les mêmes proportions que la quantité de données redondantes capturées.

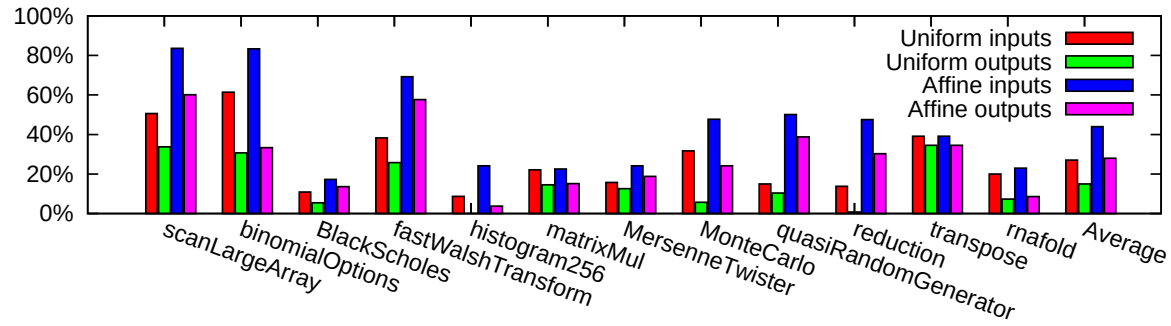


FIGURE 4.2 – Proportion d'opérandes uniformes et affines dans les registres. La moyenne est de 27 % d'entrées uniformes, 15 % de sorties uniformes, 44 % d'entrées affines et 28 % de sorties affines.

Cette détection peut se faire en plusieurs endroits, à proximité de la mémoire, du banc de registres ou des unités fonctionnelles. La détection à proximité des unités fonctionnelles permet de spécialiser du matériel pour le calcul scalaire. Ce peut être le fait d'unités dédiées comme dans les architectures Cray, ou d'unités vectorielles existantes en n'utilisant que le nombre de lignes de calcul nécessaires et en désactivant les autres par exemple à l'aide de clock-gating à grain fin comme dans le cas des unités SPU et SFU du Cell d'IBM [103]. Les architectures TESLA travaillant à une granularité de 32 éléments par vecteur, le gain en consommation est supérieur à celui d'une architecture SIMD conventionnelle travaillant à une granularité de 4 ou 8 éléments par vecteur.

Nous avons observé que les données scalaires et uniformes étaient principalement générées par des diffusions, ou des copies de registres contenant l'identifiant de threads. Dans ces cas, la détection peut être réalisée aussi bien statiquement à la compilation, que dynamiquement par du matériel dédié. La détection statique nécessite elle aussi des modifications du matériel. Premièrement car le compilateur devra marquer tous les registres et instructions manipulant des données redondantes au sein des mots machines. Deuxièmement, le matériel devra automatiquement ordonnancer les instructions lors du décodage en fonction des tags. La version dynamique ne nécessite pas de modification du compilateur, ni du jeu d'instructions. Cette solution à l'inverse requiert au minimum un registre vectoriel acceptant des tags.

Nous avons proposé et décrit dans [CDZ09], une solution dynamique basée sur un banc de registres marqué avec le type de registre qu'il contient (uniforme, affine ou générique) et testé cette solution avec le simulateur Barra.

Le surcoût estimé de cette solution était inférieur à 0.5 % du banc de registres, sans impliquer de surcoût en temps, et ce malgré le niveau d'indirection supplémentaire induit par les tags. Avec cette solution, les vecteurs affines ne nécessitent plus qu'une seule donnée et les vecteurs uniformes 2. Cette réduction s'accompagne également d'une réduction de l'utilisation des bus et ports d'accès, et par conséquent de la consommation. Les mesures ont montré que nous étions capable de réduire l'activité du bus entre le banc de registres et les unités fonctionnelles de 34 % pour les lectures, et de 22 % pour ce qui est de l'activité des unités fonctionnelles.

Des problèmes peuvent limiter l'efficacité de la méthode proposée comme les écritures partielles, l'utilisation de demi-registres et les opérations de conversion entre les vecteurs affines et uniformes.

Les valeurs redondantes sont présentes en grandes quantités dans les registres vectoriels et nous avons proposé une méthode simple et efficace pour les éliminer. Cette méthode principalement axée sur l'aspect dynamique est perfectible. En effet, les 32 données qui composent un vecteur ne sont qu'une sous partie d'un vecteur plus grand dans le modèle de programmation

SIMT utilisé par les GPU modernes. Ainsi en combinant cette approche à une analyse statique, on pourrait espérer avoir des gains jusqu'à cinq fois plus important. Cependant, manipuler des vecteurs d'une aussi grande taille n'est pas sans poser de sérieux problèmes pour la gestion de la divergence.

4.3 Analyse de comportement par mesure de la consommation

Tant que les GPU n'étaient destinés qu'à de l'accélération graphique, leur consommation électrique était un problème secondaire. Même si la consommation électrique pondérée par le nombre de GFLOP est largement en faveur de ce type de processeur comparé à des CPU traditionnels (4 GFLOP-SP/W pour la GTX280 et 0.8 GFLOP-SP/W pour un core i7 960), ces processeurs n'en demeurent pas moins de grands consommateurs. De nombreux défis doivent être relevés afin de pouvoir démocratiser leur utilisation dans les centres de calcul.

La réduction de la consommation peut être le fait de plusieurs facteurs comme les aspects technologiques, architecturaux et microarchitecturaux, ou bien algorithmiques. Concernant ce dernier point, il est nécessaire d'établir des relations entre les relevés de consommation d'une carte graphique, le temps pris et le type d'unités sollicités à un instant donné. Plusieurs outils existent pour réaliser ce type de mesure, comme Wattch [27], SimplePower [156] ou SoftExplorer [135]. Cependant ils nécessitent soit une description architecturale du processeur, soit ne fournissent qu'une estimation grossière de la consommation.

4.3.1 Mesure de consommation

Nous avons mesuré dans [CDT09b, DP13a] comment et où la consommation était réalisée au sein de différentes cartes GPU pouvant consommer jusqu'à 300 W. Les mesures ont été réalisées en utilisant un boitier PCIe Tesla externe D870 et avec un oscilloscope muni d'une pince ampèremétrique. La fréquence d'échantillonnage était de 50 Khz, fréquence permettant d'observer la consommation à l'échelle de la tâche mais pas à celle de l'instruction.

		G80			G92			GT200		
Opération	# inst.	P (W)	CPI	E (nJ/warp)	P (W)	CPI	E (nJ/warp)	P (W)	CPI	E (nJ/warp)
MAD	32	107	4.75	8.57	100	4.29	5.06	91	4.3	5.31
Pred	32	90	2.38	2.43	93	2.39	2.14	75	2.36	1.75
MAD+MUL	64	117	3.19	7.24	111	2.83	4.61	102	2.82	4.44
MAD+RCP	40	115	3.96	8.63	110	3.55	5.63	98	3.54	5.14
RCP	8	98	15.89	22.07	96	16	16.28	81	15.99	14.81
MOV	32	118	2.31	5.34	113	2.46	4.21	101	2.46	3.79

TABLE 4.1 – Nombre d'instructions, mesure de la puissance P, nombre de cycles par instruction et énergie nécessaire par warp pour une opération sur G80, G92 et GT200.

Nous avons, par exemple, mesuré la puissance consommée, le nombre de cycles par instruction et l'énergie nécessaire pour exécuter une instruction sur un warp pour différentes opérations. Les résultats sont repris dans le tableau 4.1. Afin de comparer le coût énergétique du calcul par rapport au transfert de données, nous avons aussi mesuré l'énergie consommée pour différents types d'accès mémoire. Le tableau 4.2 reprend certains de ces résutlats.

Accès mémoire	G80	G92	GT200
Coalesced stream read	124.4	103.4	80.6
L1 texture	60.7	28.3	25.6
L2 texture	62.3	48.0	66.7
Texture miss	102.2	110.6	83.1

TABLE 4.2 – Energie consommée par requête en fonction du type de mémoire sollicité. Les résultats pour l'énergie sont donnés en nJ par requête mémoire de 128 octets.

4.3.2 Analyse des résultats

Largeur des bus. Nous avons remarqué que, comparativement au Tesla C870, même si la GT200 embarquait de la GDDR3 basse consommation et un procédé de gravure plus fin, elle avait besoin de plus d'énergie pour réaliser une transposition de matrice. L'explication réside dans la largeur du bus. En effet, l'efficacité d'une implantation dépend grandement des motifs d'accès à la mémoire. Ces accès seront plus ou moins performants en fonction du nombre de bancs et de la taille du bus. Aussi dans certains cas, par exemple une matrice de taille 1024, le bus de 384 bits du C870 génère moins de conflits de bancs que le bus de 512 bits de la GT200. Nous noterons tout de même que la GT200 a besoin de deux fois moins d'énergie que le C870 pour réaliser une multiplication matricielle. Ainsi, nous avons pu constater que les accès mémoire jouaient un rôle important sur l'efficacité des algorithmes mais aussi sur la consommation.

Multiprocesseurs. Les GPU intègrent plusieurs multiprocesseurs avec leur propre front-end : chargement, décodage, lancement, unité d'exécution, ... Nous avons ainsi pu constater que la consommation augmentait linéairement avec le nombre de multiprocesseurs jusqu'à la moitié et le tiers des multiprocesseurs disponibles pour respectivement les architectures G80 et GT200. En analysant la consommation de différentes cartes pour un nombre variable de multiprocesseurs utilisés, nous avons pu mettre en avant différentes politiques d'ordonnancement des blocs sur les multiprocesseurs. Par exemple l'ordonnancement des G92 et GT200 consiste à distribuer les blocs entre les TPC jusqu'à les saturer. A l'inverse, le G80 distribue les blocs sur les multiprocesseurs d'un TPC jusqu'à le saturer avant de passer au suivant. Ces deux stratégies s'opposent du point de vue des objectifs. La première solution permet d'optimiser l'utilisation de la bande passante, alors que la deuxième solution permet d'économiser la consommation électrique.

Unités d'exécution. Nous avons également mesuré la consommation de différentes unités au sein d'un GPU (MAD, MAD et MUL, MAD et unité d'évaluation de fonctions complexes et unité double précision) dans différentes situations. Les résultats sont données dans la Table 4.1. Nous avons ainsi pu mettre en avant que les instructions prédiquées à faux sont chargées, décodées mais pas exécutées. Ce comportement est différent de celui que l'on observe sur des architectures de type ARM ou IA-64, ou la prédication est utilisée pour éviter les trous dans les pipelines. Le mécanisme de prédication au sein des GPU sert au contrôle des lignes d'exécution SIMD, alors que les trous dans le pipeline sont éliminés à l'aide du multithreading.

Hiérarchie mémoire. Nous avons aussi mesuré comment la hiérarchie mémoire impacte la consommation des GPU à l'aide de benchmarks basés sur des accès stridés à la mémoire. Nous avons ainsi pu retrouver d'importantes propriétés liées à l'utilisation de ces GPU. Nous avons

ainsi pu mesurer que la latence d'accès à la mémoire varie entre 300 et 800 ns en fonction de la localité des adresses. Par ces mesures, il est aussi possible d'inférer le fonctionnement de la hiérarchie mémoire vis à vis du TLB, de la PTE et du fonctionnement DRAM de la mémoire.

Mais plus important encore, ces mesures permettent de faire des comparaisons entre le calcul (tableau 4.1) et les accès mémoires (tableau4.2). On constate ainsi qu'un accès mémoire vectoriel, en terme de coût énergétique, correspond à l'exécution de 7 à 15 MAD en fonction de la place de la donnée dans la hiérarchie mémoire.

4.4 Fiabilisation des calculs dans les architectures multicoeurs

Les processeurs font des fautes au niveau matériel [75–77]. Les coprocesseurs massivement parallèles tel que les GPU ou le Xeon Phi (MIC) souffrent du même problème. De nombreux facteurs tel que le faible voltage, les fréquences de fonctionnement, la finesse de gravure, le nombre et la densité des transistors peuvent augmenter significativement le taux d'apparition de fautes [63, 64, 69]. En utilisant des cœurs plus simples et une fréquence moins élevée, les architectures de type GPU ont une densité de transistors beaucoup plus élevée que les CPU. Par exemple, pour une surface de 550 mm2 et 7.1 milliards de transistors, l'architecture Kepler a une densité moyenne de 12,1 milliards de transistors par mm2 alors que l'architecture SandyBrige E avec une surface de 435 mm2 et 2.27 milliards de transistors, a une densité de 5 milliards de transistors par mm2. Il y a donc un ratio de 2.5 entre les CPU et les GPU.

Les fautes matérielles peuvent être regroupées en 3 grandes catégories [63] :

— **Erreurs permanentes :** Elles sont dues à des défauts de fabrication, ou un défaut permanent lié au vieillissement du circuit.

— **Erreurs transitoires :** Ce sont des erreurs ponctuelles et non-reproductibles, résultant de facteurs externes comme les radiations ou les interférences.

— **Erreurs intermittentes :** Ce sont des erreurs temporaires mais reproductibles dans le même contexte. Elles résultent d'un vieillissement du circuit, ou d'une variation localisée de la qualité de fabrication.

De nombreux travaux se sont intéressés aux erreurs susceptibles d'apparaître dans les GPU [66]. Pour limiter l'impact de ce type d'erreur dans la hiérarchie mémoire, Nvidia a par exemple sur les architectures à partir de Fermi, introduit de la correction d'erreur ECC jusqu'au niveau registre. Cependant, cette solution ne fournit pas une protection totale contre les erreurs et les taux d'erreurs restent relativement important pour les grands systèmes [132]. Afin d'améliorer la tolérance aux pannes des GPU, il est nécessaire de considérer tous les types d'erreurs.

Afin d'analyser les erreurs intermittentes et permanentes de différents GPU, nous avons conçu et implanté une plateforme de micro-benchmarking en OpenCL [81]. Cette plateforme est destinée à permettre à l'utilisateur de détecter et localiser les erreurs intermittentes et permanentes des GPU. En utilisant la localisation, il est possible de modifier le matériel, le driver, ou le code pour réordonnancer le travail uniquement sur les unités fiables et éliminer les unités fautives.

4.4.1 Détection d'erreur

Afin de valider nos hypothèses sur le caractère localisé d'une erreur et de sa non-contamination à l'ensemble du circuit, nous avons réalisé un banc de tests destiné à accélérer le vieillissement des puces GPU lors de l'exécution de nos benchmarks. Dans l'industrie, les protocoles de test des circuit intégrés et la classification des résultats sont définis par la *Joint Electron Device Engineering Council*, JEDEC. Les protocoles de test peuvent être accélérés en changeant des paramètres physiques comme le voltage ou la température [75–77].

Nous avons donc modifié le système de refroidissement des architectures cibles et isolé thermiquement les cartes graphiques. Par un contrôle précis de la température de fonctionnement, nous avons été en mesure d'accélérer leur vieillissement. Nous avons fait fonctionner deux cartes TESLA C870 en condition de stress thermique jusqu'à l'apparition d'erreurs et constaté qu'aucune des deux cartes ne présentait d'erreur au niveau du banc de registres. Presque toutes les erreurs sont apparues au niveau des opérations MAD. Les erreurs étaient de 2 sortes :

— **Erreur vectorielle :** Ce type d'erreur s'est manifesté par l'apparition de résultat faux sur l'intégralité d'un demi-warp. Cela signifie que l'erreur est apparue soit au niveau du lancement de l'instruction ou au niveau registre.
— **Erreur scalaire :** Ce type d'erreur conduit à l'apparition d'un résultat faux impliquant un seul élément de calcul d'un multiprocesseur. Ce type d'erreur apparait plusieurs fois dans un temps très court tout en impliquant le même élément de calcul (erreur en mode burst).

4.4.2 Calculer en présence d'erreurs

Ce protocole de tests permet d'établir une cartographie des unités fautives. Nous avons pu confirmer que ces fautes demeurent silencieuses, dans la mesure où seul le résultat est entaché d'erreur et n'impacte pas le reste des calculs réalisés sur le GPU. On constate donc que l'erreur est confinée à une zone bien définie impliquant soit un SP, soit un vecteur (un SM entier), et n'apparait que de façon sporadique. Nous avons proposé dans [DP13b] une solution logicielle permettant de mettre en quarantaine l'élément fautif. Cette solution se rapproche de ce qui est déjà réalisé en matériel par les constructeurs qui, lors des tests ont la possibilité de déconnecter des multiprocesseurs complets afin de vendre ces processeurs dans une version dégradée avec moins d'unités de calcul.

La solution proposée repose sur l'altération logicielle de l'ordonnancement des blocs et threads dans les architectures TESLA afin de ne pas utiliser les résultats produits par les unités défectueuses. Pour celà, nous utilisons la cartographie des unités défectueuses afin d'englober les calculs par des tests qui n'exécutent le travail que si ceux-ci sont réalisés par des unités fiables. Ce mécanisme implique de lancer plus de threads et de blocs pour compenser le fait que les unités matérielles fautives seront évitées.

4.5 Prédictibilité dans les GPU

L'exploitation efficace des architectures multicœurs modernes passe par une structuration hiérachique du calcul mais aussi par la concurrence d'exécution. Or la concurrence d'exécution impacte le déterminisme et la reproductibilité des logiciels, rendant leur développement plus complexe. Dans ce modèle, il est en effet difficile de s'assurer qu'une tâche complexe soit correcte pour toutes les combinaisons possibles d'interaction entre ces tâches. Ces interactions utilisent des hypothèses sur l'ordre d'exécution ou l'utilisation de mécanisme de synchronisation comme les verrous, les fonctions atomiques ou les barrières.

Si la prédictibilité est une notion à l'origine de nombreux mécanismes en architecture des ordinateurs (prédicteur de branchement, d'adresse, de valeur, ...), elle est souvent manipulée sous sa forme binaire, à savoir "est-on capable de prédire ou pas". Mais il peut s'avérer pertinent de considérer cette prédictibilité sous une version probabiliste, notamment pour ce qui est de la prédictibilité relative au comportement d'une architecture. Ces informations sont essentielles pour caractériser le comportement des processeurs pour la construction de simulateurs [CDDP10],

estimer le WCET [91], ou à l'inverse être utile pour mesurer la pertinence de générateurs de nombres aléatoires basés sur le non-déterminisme dans l'accès à une ressource partagée [32].

4.5.1 Sources d'indéterminisme dans les GPU

Le fait même que les GPU soient des coprocesseurs les isole de certaines sources d'indéterminisme comme les interruptions ou les changements de contexte que l'on retrouve dans le cas des processeurs généralistes. Cependant, les futures générations pourraient être plus sensibles à ces sources, si leurs architectures venaient à se rapprocher des CPU.

Les GPU ont plusieurs domaines d'horloge, chacun fonctionnant à une fréquence optimale. Les circuits de synchronisation entre ces domaines peuvent introduire de l'indéterminisme lié aux délais induits par les changements de phase lorsque des messages sont échangés [129]. Le poids de cette source d'indéterminisme devrait augmenter dans les générations futures, dans la mesure où les techniques de gestion dynamique de fréquence et de voltage (DVFS) devraient se généraliser à la granularité du multiprocesseur.

Une autre source d'indéterminisme est liée au temps d'accès à la mémoire hors puce car celui-ci est fonction de l'endroit physique (partition mémoire) où la donnée est placée [155]. Sur les GPU modernes, la probabilité que l'allocation et le placement mémoire soient identiques entre deux appels consécutifs est très faible. De plus, les variations dans les cellules DRAM invitent à l'utilisation de techniques de rafraichissement dynamique introduisant des temps d'accès aux cellules DRAM variables [92].

Avec la miniaturisation des transistors, l'apparition de fautes matérielles devient de plus en plus courante. Aussi, ces fautes sont dans certains cas corrigées par des mécanismes introduisant des délais variables dans les latences d'accès mémoires [154].

Enfin l'utilisation d'ordonnanceurs non initialisés introduit de l'indéterminisme en modifiant l'ordre d'exécution et potentiellement les assignations entre logiciel et matériel. Ceci regroupe les ordonnanceurs de warps dans chaque unité de calcul, les ordonnanceurs de blocs et les arbitres de bus dans les réseaux d'interconnexions. Même si ces éléments sont initialisés lors de leur mise sous tension, ils ne le sont pas entre chaque lancement de kernel. Aussi l'état de ces unités est donc dépendant des lancements des kernels précédents, et peuvent donc être considérés comme non prédictibles.

Ordonnanceurs. Au lancement d'un kernel, l'ordonnanceur de blocs distribue les blocs entre les multiprocesseurs disponibles jusqu'à saturation des ressources (registres, mémoire partagée, ou nombre de blocs admissibles par multiprocesseur). L'algorithme d'assignation entre les blocs (niveau logiciel) et les multiprocesseurs est de type round-robin et décrit dans [104]. Dès que l'exécution d'un bloc se termine, les ressources occupées par le bloc sont libérées et l'ordonnanceur peut en assigner un nouveau. Chaque multiprocesseur gère donc en parallèle les warps de plusieurs blocs. Le lancement des instructions est réalisé dans l'ordre par le ou les ordonnanceurs de warps. Cette unité a accès à la file des instructions en attente et en sélectionne une prête à être exécutée. La sélection se base sur la disponibilité des ressources matérielles, l'âge, le type d'instruction [88, 113] et l'architecture.

Il existe des variations entre ces ordonnanceurs qui dépendent de la génération de GPU Nvidia considérée, ou version de *compute capability* [105]. Cette classification va de 1.x à 3.x, pour les dernières architectures disponibles en 2013.

4.5.2 Mesure de la prédictibilité

La prédictibilité, dans un modèle d'exécution parallèle, correspond à la possibilité de prédire le comportement ou les résultats d'une exécution. Or ces éléments ne sont pas quantitatifs mais plutôt qualitatifs. En effet, il est difficile de donner une mesure de distance d'un résultat ou comportement par rapport à une prédiction ; celle-ci est soit juste, soit fausse. Cependant, si l'on s'autorise à réaliser des observations sur plusieurs exécutions, il est alors possible de collecter des informations sur l'espace des résultats possibles associés à des probabilités d'occurrences.

Nous avons donc introduit une mesure de la prédictibiité dans [Def14] en nous concentrant sur les ordonnanceurs de blocs et de threads des GPU Nvidia et leur contribution à l'indéterminisme observable. Nous avons fait le choix d'utiliser des vecteurs d'exécution contenant l'information d'ordonnancement afin d'analyser dans de futurs travaux les corrélations entre les exécutions. A l'aide du mode statistique et de la mesure de dispersion des ordonnancements observables, nous avons montré les différences entre les architectures mais aussi entre les ordonnancements de blocs et de threads. Nous avons ainsi pu définir les configurations exhibant la plus grande prédictibilité.

Nous avons remarqué que certaines architectures exhibent un mode statistique important tout en ayant une mesure de dispersion élevée, comme c'est le cas pour l'ordonnanceur de warps de la GTX480. Aussi, pour ce type d'architecture, il pourrait être utile d'augmenter artificiellement la probabilité d'apparition de vecteurs considérés comme peu probables afin d'accélérer les tests de couvertures, ou améliorer les générateurs de nombres pseudo-aléatoires. Pour y parvenir, on peut envisager une altération logicielle du mécanisme d'ordonnancement semblable à ce que nous avions proposé pour éviter l'ordonnnancement sur les unités défectueuses [DP13b].

A l'inverse, si l'on souhaite limiter l'indéterminisme pour rejouer des jeux de tests, faciliter le calcul du WCET ou la compréhension d'erreurs de type Heisenbug, il faut trouver des moyens de limiter les comportements possibles pour permettre l'observation d'un comportement donné et prédictible. Il reste donc à montrer comment certains facteurs tels que la fréquence de fonctionnement, l'initialisation du coprocesseur ou la synchronisation impactent la prédictibilité, et à en évaluer le coût.

5

Conclusion et perspectives

Réalisations

Les processeurs graphiques sont désormais considérés comme de puissants accélérateurs de calcul, comme en témoigne leur intégration dans les supercalculateurs. Nous nous sommes intéressés dans ce manuscrit à la problématique du calcul et en particulier du calcul flottant, au sein de ces processeurs dans leur dimension architecturale et algorithmique. Nos travaux sur ce sujet ont accompagné l'évolution de ces architectures qui sont progressivement passées d'un support incomplet et non documenté du standard IEEE-754 à un support plus proche de ce que l'on est en droit d'attendre d'un accélérateur.

Nous avons constaté que ces processeurs ont intégré, et même parfois devancé, la révision en 2008 de la norme IEEE-754 qui a standardisé de nouveaux formats afin de s'adapter aux contraintes de bande passante et/ou de précision. Nous avons montré qu'il est désormais possible d'envisager pour un coût raisonnable, l'utilisation d'arithmétiques non conventionnelles réputées consommatrices de ressources et de temps de calcul.

L'amélioration de la puissance de calcul ou de la précision passe par l'analyse du comportement aussi bien au niveau matériel qu'applicatif. Nous avons pour cela proposé un simulateur fonctionnel de processeurs graphiques. Avec l'aide de ce type d'outil, nous nous sommes intéressé à la problématique de l'accès aux données dans les registres mais aussi à celle de la gestion des branchements. Nous avons montré comment accélérer les calculs au niveau matériel, en diminuant la latence de séquence d'instructions répétitives comme c'est le cas avec les opérateurs ternaires ou avec les unités d'évaluation de fonctions spécialisées.

Si les GPU sont de puissants accélérateurs, la débauche d'unités de calcul conduit nécessairement à de la redondance. Cette redondance se retrouve aussi bien dans la structuration du calcul que dans les données manipulées. Nous avons montré que si elle impacte la consommation, elle permet en revanche de fiabiliser les calculs lorsque le processeur devient défaillant. De plus, nous avons constaté que ces architectures, qui reposent sur la concurrence d'exécution, ont des comportements de moins en moins prédictibles. Or cette absence de prédictibilité est néfaste dans le cas des algorithmes numériques car elle impacte négativement la reproductibilité numérique des résultats.

Perspectives

Les travaux présentés dans ce document peuvent se poursuivre dans plusieurs directions. Nous en détaillons cinq.

Virtualisation du calcul flottant

Si les numériciens et les architectes mesurent l'importance du coût des transferts de données, ils ne disposent aujourd'hui que de peu d'outils pour sélectionner dynamiquement le bon format flottant. Le choix est laissé au développeur qui n'a pas forcément les compétences et les connaissances pour faire le bon choix. De plus, ce choix est réalisé statiquement à la création du programme et ne peut plus faire l'objet de modifications ultérieures. C'est une des raisons pour lesquels, de nombreux programmes surdimensionnent les formats conduisant à un gaspillage de la mémoire et de la bande passante.

Une solution serait de considérer le calcul non plus comme un élément figé dans le temps et l'espace, mais de lui donner une dimension dynamique pouvant évoluer en fonction de critères comme les ressources disponibles, la précision souhaitée ou d'un budget consommation. Ainsi une nouvelle dimension pourrait être apportée à l'arithmétique flottante sous l'influence des nouvelles contraintes introduites par les architectures multicœurs.

De récents travaux ont déjà été entrepris en ce sens, que ce soit avec PRECIMONIOUS [125] ou d'autres [21, 83] qui mesurent l'impact en terme de temps et/ou de précision des différents formats flottants pour un jeu de donnée fixé. Afin de démocratiser l'utilisation de ces formats, une alternative que nous proposons est d'envisager une abstraction du format flottant qui s'adapterait dynamiquement aux besoins, mais aussi aux conditions dans lesquelles le calcul est réalisé. Ce travail fait l'objet d'une thèse (Abdelkrim Chaouch, début en septembre 2014) et d'une collaboration avec Nikita Astafiev, Intel Moscou.

Opérateurs spécialisés

Une des spécificités des GPU est d'embarquer des opérateurs spécialisés pour les tâches répétitives liées au traitement graphique (interpolation, $1/x$, $1/\sqrt{(x)}$,...). L'implémentation des unités d'évaluation de fonctions spéciales présentes dans les GPU est le résultat d'une étude exhaustives [107] pour obtenir le meilleur compromis entre précision et performance. Cette spécialisation des unités permet d'accélérer le traitement de ces fonctions, mais elle se fait au détriment de la souplesse d'utilisation. Il est en effet très difficile d'utiliser ces unités pour d'autres fonctions que celles pour lesquelles elles ont été définies.

Nous avons montré en section 2.4, qu'il était toutefois possible de les détourner pour l'évaluation de fonctions pouvant être approchées par un polynôme de faible degré. Les limitations de ces unités rendaient ces détournements peu efficaces. Néanmoins, ces travaux ont démontré l'intérêt d'avoir des unités paramétrables permettant de réaliser rapidement des approximations en matériel à base de données tabulées et d'approximations polynomiales.

Une piste de recherche serait de proposer de telles unités paramétrables aussi efficaces que les unités existantes, tout en élargissant le spectre de fonctions pouvant être évaluées.

Applications aux domaines des énergies renouvelables

Nos travaux ont pour une large part été inspirés par des problématiques liées à l'utilisation des accélérateurs graphiques pour la simulation d'applications du domaine des énergies renouvelables. Que ce soit avec la simulation de récepteurs solaires [CDD08b,CDD07b], ou de réseaux électriques [DM12] pour lesquels les GPU permettent aux physiciens de réaliser des simulations à moindre coût sur des stations de travail.

Dans le cas de la simulation de réseaux électriques, nous avons proposé des opérateurs d'arithmétique floue pour gérer efficacement l'incertitude dans les modèles [MDM14], ou des algorithmes

de parcours d'arbre adaptés au traitement sur GPU [DM13b]. Ces premiers travaux ont démontré le potentiel des GPU, pour les outils classiques basés sur des itérations de Newton-Raphson, ou la résolution de systèmes linéaires. Ces outils, s'ils sont parfaitement maitrisés depuis de nombreuses années, présentent un caractère intrinsèquement séquentiel dans la résolution des systèmes. Or cette séquencialité ne correspond pas à la réalité physique d'un réseau électrique ou les composants interagissent en parallèle avec leurs voisins directs jusqu'à stabilisation du circuit.

En collaboration avec Federico Milano (University College of Dublin), nous explorons sur GPU le potentiel des itérations asynchrones [19] qui permet de reproduire ce comportement physique au niveau algorithmique. Ce travail fait l'objet d'une thèse (Manuel Marin, début en septembre 2012).

Fiabilisation des calculs

Les processeurs modernes sont de plus en plus sujet aux défaillances matérielles. Ces défaillances sont dues à différents facteurs liés à la tension, la fréquence ou le processus de gravure des transistors. Ces erreurs peuvent être permanentes, transitoires ou intermittentes. Leurs nombres et leurs fréquences d'apparitions augmentent avec le vieillissement des puces. Il est donc nécessaire de composer avec le fait qu'une architecture à l'origine fiable, devienne non-fiable avec le temps.

Les GPU ont une densité de transistors supérieure aux processeurs classiques et sont à ce titre sujet à ce type de problème. Cependant ils offrent plusieurs milliers de cœurs de calcul organisés hiérarchiquement. Aussi, nous avons démontré par des tests choisis dans [DP13b], en collaboration avec Eric Petit (Université de Versailles Saint-Quentin), qu'il est possible d'identifier les tuiles et le niveau hiérarchique défaillant.

Nous souhaitons donc poursuivre ce travail pour proposer des solutions logiciels (middleware) permettant de contenir le calcul sur les tuiles valides et éviter les tuiles problématiques. Pour cela, nous envisageons d'exploiter les relations entre le modèle de programmation des GPU et le modèle architectural de ces processeurs.

Reproductiblité numérique

L'augmentation du nombre de cœur rend le matériel et les algorithmes de plus en plus imprédictibles. Le comportement des processeurs est dépendant de nombreux paramètres externes (température, fréquence de fonctionnement, type et âge du processeur, ...) et interne (états initiaux, charge). De même les algorithmes tendent à exploiter aux mieux ces ressources de calculs ce qui impacte, entre autre, la reproductibilité numérique. Sur cette question, l'objectif peut être d'améliorer la reproductibilité pour simplifier le debbogage, ou à l'inverse d'augmenter l'entropie pour, par exemple, accélerer du test de couverture. Nous souhaitons poursuivre les travaux présentés en section 4.5 sur ce sujet au niveau architectural et algorithmique.

Au niveau architectural, nous envisageons d'explorer l'impact sur la reproductibilité de modifications de la fréquence de fonctionnement, de la température, de la charge du processeur ou des ordonnanceurs.

Au niveau algorithmique, nous envisageons deux pistes permettant d'améliorer la reproductibilité numérique. La première consiste à contraindre l'ordonnancement afin d'éliminer les variations dans les schémas d'évaluation. La deuxième consiste à éliminer les erreurs numériques sources d'indéterminisme. Ce dernier point est l'objet d'un travail en cours avec Stef Graillat, Roman Iakimchuck (LIP6, Paris 6), et Sylvain Collange (IRISA, Rennes). Il porte sur la réalisation

de briques de base d'algèbre linéaire (BLAS) reproductibles et optimisées pour les architectures manycœurs (GPU, Intel MIC), inspiré de l'accumulateur long de Kulisch [CDGI14] .

Bibliographie personnelle

[ACD10] M. G. Arnold, S. Collange, and D. Defour. Implementing LNS using filtering units of gpus. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, number hal-00423434, January 2010.

[CDD07b] Sylvain Collange, Marc Daumas, and David Defour. Graphic processors to speed-up simulations for the design of high performance solar receptors. In *ASAP*, pages 377–382, July 2007.

[CDD08a] S. Collange, M. Daumas, and D. Defour. Etat de l'intégration de la virgule flottante dans les processeurs graphiques. *Revue des sciences et technologies de l'information*, 27/6 :719–733, 2008.

[CDD08b] Sylvain Collange, Marc Daumas, and David Defour. Line-by-line spectroscopic simulations on graphics processing units. *Computer Physics Communications*, 178 :135–143, January 2008.

[CDD12] Sylvain Collange, Marc Daumas, and David Defour. Chapter 9 - interval arithmetic in cuda. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 99 – 107. Morgan Kaufmann, Boston, 2012.

[CDDO08] Sylvain 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)*, pages 1–11, February 2008.

[CDDP09] S. Collange, M. Daumas, D. Defour, and D. Parello. Étude comparée et simulation d'algorithmes de branchements pour le gpgpu. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, September 2009.

[CDDP10] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra : A parallel functional simulator for gpgpu. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351–360, 2010.

[CDGI14] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Full-speed deterministic bit-accurate parallel floating-point summation. Technical Report hal-00949355, HAL-CCSD, 2014.

[CDP09b] Sylvain Collange, David Defour, and David Parello. Barra, a Modular Functional GPU Simulator for GPGPU. Technical Report hal-00359342, Université de Perpignan, 2009. http ://hal.archives-ouvertes.fr/hal-00359342/en/.

[CDT09b] Sylvain Collange, David Defour, and Arnaud Tisserand. Power Consuption of GPUs from a Software Perspective. In *ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 922–931. Springer, 2009.

[CDZ09] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, number hal-00396719, pages 1–10, June 2009.

[CFD08] Sylvain Collange, Jorge Flóres, and David Defour. A gpu interval library based on boost interval. In *Real Numbers and Computers*, pages 61–72, July 2008.

[DD06] Guillaume Da Gra ca and David Defour. Implementation of float-float operators on graphics hardware. In *RNC7*, pages 23–32, July 2006.

[DdD03] David Defour and Florent de Dinechin. Software carry-save : A case study for instruction-level parallelism. In *PaCT*, pages 207–214, 2003.

[DDD06] Marc Daumas, Guillaume Da Gra ca, and David Defour. Caractéristiques arthmétiques des processeurs graphiques. In *SympA*, pages 86–95, October 2006.

[Def05a] David Defour. Collapsing dependent floating point operations. In *IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, pages 1–10, Paris, France, July 2005.

[Def14] D. Defour. Prédictibilité des ordonnanceurs des gpu. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 1–10, April 2014.

[DG04] David Defour and Bernard Goossens. Implémentation de l'opérateur add2. Research Report 3, DALI Research Team, LP2A, University of Perpignan, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, December 2004.

[DM12] D. Defour and M. Marin. Real-time simulation of power networks using multi-core architecture. In *DERBI 2012*. DERBI, October 2012.

[DM13a] David Defour and Manuel Marin. Fuzzygpu : a fuzzy arithmetic library for gpu. http://code.google.com/p/fuzzy-gpu/, 2013.

[DM13b] David Defour and Manuel Marin. Regularity versus load-balancing on GPU for treefix computations. *Procedia Computer Science*, 18(0) :309 – 318, 2013. 2013 International Conference on Computational Science.

[DM14] David Defour and Manuel Marin. Fuzzygpu : a fuzzy arithmetic library for gpu. *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2014)*, 2014.

[DP13a] D. Defour and E. Petit. Températures, erreurs matérielles et gpu. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 1–10, January 2013.

[DP13b] David Defour and Eric Petit. GPUburn : A system to test and mitigate GPU hardware failures. In *ICSAMOS*, pages 263–270, 2013.

[GD05a] Bernard Goossens and David Defour. The instruction register file micro-architecture. *Future Generation Comp. Syst.*, 21(4) :767–773, 2005.

[GD05b] Bernard Goossens and David Defour. Ordonnancement dynamique distribué. In *SympA*, pages 1–10, 2005.

[GD06a] Bernard Goossens and David Defour. The instruction register file micro-architecture. *Future Generation Computer Systems*, 21(5) :767–773, 2006.

[GD06b] Bernard Goossens and David Defour. Ordonnancement distribué d'instructions. *Technique et Science Informatiques*, 25(7) :827–844, 2006.

[MD13] Manuel Marin and David Defour. Cuda et les formats de représentation des nombres flottants. *HPC Magazine*, (4) :52–57, June 2013.

[MDM14] Manuel Marín, David Defour, and Federico Milano. Power flow analysis under uncertainty using symmetric fuzzy arithmetic. In *IEEE PES General Meeting*, July 2014.

Bibliographie générale

[1] Comsol Multiphysics modeling environment. `http://www.comsol.fr`.

[2] MPFR, the Multiprecision Precision Floating-Point Reliable library. http ://www.mpfr.org/.

[3] SPECviewperf 8.1 : Standard Performance Evaluation Corporation. `http://www.spec.org/gwpg/gpc.static/vp81info.html`.

[4] Binary floating-point arithmetic for microprocessor systems. ISO / IEC Standard 60559, International Electrotechnical Commission, 1989.

[5] Alpha architecture handbook, october 1998.

[6] *16th IEEE International On-Line Testing Symposium (IOLTS 2010), 5-7 July, 2010, Corfu, Greece*. IEEE, 2010.

[7] *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, www.ispass.org, 28-30 March 2010, White Plains, NY, USA*. IEEE Computer Society, 2010.

[8] *29th IEEE VLSI Test Symposium, VTS 2011, May 1-5, 2011, Dana Point, California, USA*. IEEE Computer Society, 2011.

[9] The university of florida sparse matrix collection. http ://www.cise.ufl.edu/research/sparse/matrices/, 2012.

[10] Advanced Micro Device, Inc. *R700-Family Instruction Set Architecture*, March 2009.

[11] Pritpal S. Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Multipath execution : Opportunities and limits. In *International Conference on Supercomputing*, pages 101–108, 1998.

[12] Carlos Álvarez, Jesús Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7) :922–927, 2005.

[13] 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.

[14] David I. August, Sharad Malik, Li-Shiuan Peh, Vijay Pai, Manish Vachharajani, and Paul Willmann. Achieving structural and composable modeling of complex systems. *Int. J. Parallel Program.*, 33(2) :81–101, 2005.

[15] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar : An infrastructure for computer system modeling. *Computer*, 35(2) :59–67, 2002.

[16] D. H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4) :379–387, 1995.

[17] 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.

[18] Saisanthosh Balakrishnan and Gurindar S. Sohi. Exploiting value locality in physical register files. In *MICRO 36 : Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 265, Washington, DC, USA, 2003. IEEE Computer Society.

[19] D. El Baz. *Contribution à l'algorithmique parallèle, Le concept d'asynchronisme : étude théorique, mise en œuvre et application.* PhD thesis, Institut National Polytechnique de Toulouse, October 1998.

[20] Edward Benowitz, Miloš Ercegovac, and Farzan Fallah. Reducing the latency of division operations with partial caching. *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*, 2 :1598–1602 vol.2, 3-6 Nov. 2002.

[21] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM.

[22] 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.

[23] S. Bonvicini, P. Leonelli, and G. Spadoni. Risk analysis of hazardous materials transportation : evaluating uncertainty by means of fuzzy logic. *Journal of Hazardous Materials*, 62(1) :59 – 74, 1998.

[24] K. Briggs. The doubledouble library, 1998. http ://members.lycos.co.uk/keithmbriggs/doubledouble.html.

[25] Nicolas Brisebarre, Jean-Michel Muller, and Saurabh Kumar Raina. Accelerating correctly rounded floating point division when the divisor is known in advance. 53(8) :1069–1072, 2004.

[26] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the boost interval arithmetic library. *Theor. Comput. Sci.*, 351(1) :111–118, 2006.

[27] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch : a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2) :83–94, May 2000.

[28] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. GPUbench : evaluating gpu performance for numerical and scientifc application. In *Proceedings of the ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, Los Angeles, California, 2004.

[29] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs : Stream computing on graphics hardware. In *Proceedings of SIGGRAPH 2004*, pages 777 – 786, August 2004.

[30] J.A. Butts and G.S. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004*, pages 302–313, 2004.

[31] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, October 1994.

[32] J.J.M. Chan, B. Sharma, Jiaqing Lv, G. Thomas, R. Thulasiram, and P. Thulasiraman. True random number generator using gpus and histogram equalization techniques. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 161–170, 2011.

[33] Chir-Ho Chang and Ying-Chiang Wu. The genetic algorithm based tuning method for symmetric membership functions of fuzzy logic control systems. In *Industrial Automation and Control : Emerging Technologies, 1995., International IEEE/IAS Conference on*, pages 421–428, 1995.

[34] Chen-Tung Chen, Ching-Torng Lin, and Sue-Fn Huang. A fuzzy approach for supplier evaluation and selection in supply chain management. *International Journal of Production Economics*, 102(2) :289 – 301, 2006.

[35] I-Cheng K. Chen, Chih-Chieh Lee, and Trevor N. Mudge. Instruction prefetching using branch prediction information. In *ICCD*, pages 593–601, 1997.

[36] Min-You Chen and D.A. Linkens. Rule-base self-generation and simplification for data-driven fuzzy models. In *Fuzzy Systems, 2001. The 10th IEEE International Conference on*, volume 1, pages 424–427, 2001.

[37] Daniel Citron and Dror G. Feitelson. Hardware memoization of mathematical and trigonometric functions, 2000.

[38] Michael A. Clark, Ronald Babich, Kipton Barros, Richard C. Brower, and Claudio Rebbi. Solving lattice qcd systems of equations using mixed precision solvers on gpus. pages 1517–1528, 2010.

[39] William J. Cody. MACHAR : a subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4) :303–311, 1988.

[40] William J. Cody. Algorithm 714 : CELEFUNT : a portable test package for complex elementary functions. *ACM Transactions on Mathematical Software*, 19(1) :1–21, 1993.

[41] William J. Cody. Algorithm 715 : SPECFUN – a portable Fortran package of special function routines and test drivers. *ACM Transactions on Mathematical Software*, 19(1) :22–30, 1993.

[42] William J. Cody, Richard Karpinski, et al. A proposed radix and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4) :86–100, 1984.

[43] Brett W. Coon and John Erik Lindholm. System and method for managing divergent threads in a SIMD architecture. US Patent US 7353369 B1, April 2008. NVIDIA Corporation.

[44] J.L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000*, pages 316–325, 2000.

[45] ATI Technologies Inc Daniel B. Clifton. Method and system for approximating sine and cosine functions. US Patent 6 976 043, US Patent Office, 2001.

[46] Marc Daumas. pages 227–244. 2005.

[47] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. Generating high-performance custom floating-point pipelines. In Martin Danek, Jiri Kadlec, and Brent E. Nelson, editors,

19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic, pages 59–64. IEEE, 2009.

[48] Yangdong (Steve) Deng, Bo David Wang, and Shuai Mu. Taming irregular eda applications on gpus. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, New York, NY, USA, 2009. ACM.

[49] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology, 2009.

[50] Didier Dubois and Henri Prade. Operations on fuzzy numbers. *International Journal of Systems Science*, 9(6) :613–626, 1978.

[51] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, pages 37–46, 2002.

[52] Torbjörn Granlund et al. GNU multiple precision arithmetic library. http ://s-wox.com/gmp/.

[53] A. Eustace and A. Srivastava. Atom : A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 303–314, 1995.

[54] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial : The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[55] J. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7) :478–490, July 1981.

[56] Walter M. Fitch. Toward defining the course of evolution : Minimum change for a specific tree topology. *Syst Biol*, 20 :406–416, 1971.

[57] Jorge Flórez, Mateu Sbert, MiguelA. Sainz, and Josep Vehí. Efficient ray tracing using interval analysis. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 1351–1360. Springer Berlin Heidelberg, 2008.

[58] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07 : Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[59] 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.

[60] Dominik Goddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision fem simulations with GPUs. In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, September 2005.

[61] R. Gonzalez, A. Cristal, M. Pericàs, A. Veidenbaum, and M. Valero. Scalable Distributed Register File. In *Workshop on Complexity-effective Design held in conjunction with the 31st International Symposium on Computer Architecture, 2004*, 2004.

[62] Frédéric Goualard. Gaol 3.1. 1 : Not just another interval arithmetic library. *Laboratoire d'Informatique de Nantes-Atlantique*, 4, 2006.

[63] Julien Guilhemsang. *Test en ligne pour la détection des fautes intermittentes dans les architectures multiprocesseurs embarquées*. These, Université de Nice Sophia-Antipolis, April 2011.

[64] Julien Guilhemsang, Olivier Héron, Nicolas Ventroux, Olivier Goncalves, and Alain Giulieri. Impact of the application activity on intermittent faults in embedded systems. In *VTS* [8], pages 191–196.

[65] Eric Hao, Po-Yung Chang, Marius Evers, and Yale N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming*, 26(4) :449–478, 1998.

[66] Imran S. Haque and Vijay S. Pande. Hard data on soft errors : A large-scale assessment of real-world error rates in gpgpu. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 691–696, Washington, DC, USA, 2010. IEEE Computer Society.

[67] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[68] K.A. Hawick, A. Leist, and D.P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, 36(12) :655 – 678, 2010.

[69] Olivier Héron, Julien Guilhemsang, Nicolas Ventroux, and Alain Giulieri. Analysis of online self-testing policies for real-time embedded multiprocessors in dsm technologies. In *IOLTS* [6], pages 49–55.

[70] Y. Hida, X. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, Jun 2001.

[71] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. In *Intel technology journal, Q1, 2001*, 2001.

[72] M. Hussein, Amitabh Varshney, and Larry S. Davis. On implementing graph cuts on cuda. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007/// 2007.

[73] IBM accurate portable mathematical library.

[74] Intel. *Intel® G45 Express Chipset Graphics Controller PRM, Volume Four : Subsystem and Cores*, February 2009. www.intellinuxgraphics.org.

[75] JEDEC. Foundry process qualification guidelines. *JEDEC technical report*, 2004.

[76] JEDEC. Accelerated moisture resistance-unbiased autoclave. *JEDEC technical report*, 2008.

[77] JEDEC. Stress-test-driven qualification of integrated circuits. *JEDEC technical report*, 2011.

[78] K. Kailas, M. Franklin, and K. Ebcioglu. A register file architecture and compilation scheme for clustered ILP processors. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002*, pages 500–511, 2002.

[79] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33 : Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170, New York, NY, USA, 2000. ACM.

[80] Richard Karpinski. PARANOIA : a floating-point benchmark. *Byte*, 10(2) :223–235, 1985.

[81] KHRONOS. Opencl official website. *http ://www.khronos.org/opencl/*, 2012.

[82] N. S. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the 17th Annual International Conference on Supercomputing, 2003*, pages 172–182, 2003.

[83] Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. *Parallel Comput.*, 39(3) :146–155, March 2013.

[84] Philippe Langlois and Nicolas Louvet. More instruction level parallelism explains the actual efficiency of compensated algorithms. Technical Report hal-00165020, HAL-CCSD, 2007.

[85] Christoph Quirin Lauter. *Arrondi correct de fonctions mathématiques : fonctions univariées et bivariées, certification et automatisation.* PhD thesis, 2008. 2008ENSL0482.

[86] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, pages 59–70, 2002.

[87] Charles Leiserson and Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3 :53–77, 1988.

[88] Erik Lindholm, John Nickolls, Stuart F. Oberman, and John Montrym. NVIDIA tesla : A unified graphics and computing architecture. *IEEE Micro*, 28(2) :39–55, 2008.

[89] Erik Lindholm, Ming Y. Siu, Simon S. Moy, Samuel Liu, and John R. Nickolls. Simulating multiported memories using lower port count memories. US Patent US 7339592 B2, March 2008. NVIDIA Corporation.

[90] 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.

[91] Björn Lisper. Towards parallel programming models for predictability. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, volume 23 of *OASICS*, pages 48–58. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[92] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR : Retention-aware intelligent DRAM refresh. In *ISCA*, pages 1–12. IEEE, 2012.

[93] Raymond A. Lorie and Hovey R. Strong. Method for conditional branch execution in simd vector processors. US Patent 4,435,758, March 1984. IBM.

[94] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[95] 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.

[96] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[97] 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.

[98] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8) :117–128, February 2012.

[99] P. Michaud and A. Seznec. Data flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture, 2001*, pages 27–36, 2001.

[100] V Miranda and JT Saraiva. Fuzzy modelling of power system optimal load flow. In *Power Industry Computer Application Conference, 1991. Conference Proceedings*, pages 386–392. IEEE, 1991.

[101] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to interval analysis*. Society for Industrial and Applied Mathematics, pub-SIAM :adr, 2009.

[102] 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.

[103] S.M. Mueller, C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S.H. Dhong. The vector floating-point unit in a synergistic processor element of a cell processor. pages 59–67, June 2005.

[104] 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*, 2009.

[105] NVIDIA. *NVIDIA CUDA C Programming Guide 5.0*. 2013.

[106] Stuart F. Oberman and Michael J. Flynn. On division and reciprocal caches. Technical Report CSL-TR-95-666, 1995.

[107] Stuart F. Oberman and Michael Siu. A high-performance area-efficient multifunction interpolator. In Koren and Kornerup, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (Cap Cod, USA)*, pages 272–279, Los Alamitos, CA, July 2005. IEEE Computer Society Press.

[108] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997*, pages 206–218, 1997.

[109] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th International Symposium on Microarchitecture, 2002*, pages 171–182, 2002.

[110] Michael Parks. Number theoretic test generation for directed rounding. pages 241–248, 1999.

[111] Alex Peleg and Uri Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. US Patent 5,381,533, US Patent & Trademark Office, Washington DC, patent filed 1992, granted 1995.

[112] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. Microlib : A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37 : Proceedings of*

the 37th annual IEEE/ACM International Symposium on Microarchitecture, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society.

[113] John Erik Lindholm Peter C. Mills, 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*, 2008.

[114] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing alu's. *IEEE Transactions on computer*, 43(3) :257–268, 1994.

[115] Jim Pierce and Trevor N. Mudge. Wrong-path instruction prefetching. In *MICRO*, pages 165–175, 1996.

[116] Samuel Pollock Harbison III. *A computer architecture for the dynamic optimization of high-level language programs.* PhD thesis, Pittsburgh, PA, USA, 1980.

[117] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. pages 132–144, 1991.

[118] M. Ramirez, A. Cristal, A. Veidenbaum, L. Villa, and M. Valero. Direct instruction wakeup for out-of-order processors. In *Proceedings of the International Workshop on Innovative Architecture (IWIA'04), 2004*, pages 2–9, 2004.

[119] G. Reinman, B. Calder, and T. Austin. A scalable front-end architecture for fast instruction delivery. In Doug DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, volume 27, 2 of *Computer Architecture News*, pages 234–245, New York, N.Y., May 1–5 1999. ACM Press.

[120] Glenn Reinman, Brad Calder, and Todd M. Austin. Fetch directed instruction prefetching. In *MICRO*, pages 16–27, 1999.

[121] N Revol and F Rouillier. The mpfi library, 2001.

[122] 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.

[123] 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.

[124] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache : A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[125] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious : Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27 :1–27 :12, New York, NY, USA, 2013. ACM.

[126] Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3) :534–554, 1999.

[127] A.Y. Saber and G.K. Venayagamoorthy. Resource scheduling under uncertainty in a smart grid with renewables and plug-in vehicles. *Systems Journal, IEEE*, 6(1) :103–109, 2012.

[128] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(35–42), 1975.

[129] Smruti R. Sarangi, Brian Greskamp, and Josep Torrellas. CADRE : Cycle-accurate deterministic replay for hardware debugging. In *DSN*, pages 301–312. IEEE Computer Society, 2006.

[130] T. Sato, Y. Nakamura, and I. Arita. Revisiting direct tag search algorithm on superscalar processors. In *Workshop on Complexity-effective Design held in conjunction with the 28th International Symposium on Computer Architecture, 2001*, 2001.

[131] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & colapsing. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 238–247, 1996.

[132] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild : a large-scale field study. *Commun. ACM*, 54(2) :100–107, 2011.

[133] N. L. Schryer. A test of computer's floating-point arithmetic unit. Technical report 89, AT&T Bell Laboratories, 1981.

[134] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[135] E. Senn, J. Laurent, N. Julien, and E. Martin. SoftExplorer : Estimating and optimizing the power and energy consumption of a C program for DSP applications. *EURASIP Journal on Applied Signal Processing*, pages 2641–2654, January 2005.

[136] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization : a path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th annual international symposium on Microarchitecture, 2002*, pages 383–394, 2002.

[137] J. W. Sheaffer, D. Luebke, and K. 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.

[138] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[139] D. Shirmohammadi, H.W. Hong, A. Semlyen, and G.X. Luo. A compensation-based power flow method for weakly meshed distribution and transmission networks. *Power Systems, IEEE Transactions on*, 3(2) :753 –762, may 1988.

[140] IEEE Computer Society. *IEEE Std. 754-2008 Standard for Floating-Point Arithmtic*. IEEE, 3 Park Avenue, NY 10016-5997, USA, August 2008. available at : http ://ieeexplore.ieee.org/servlet/opac ?punumber=4610933.

[141] David Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3) :51–62, 1981.

[142] David Stevenson et al. An American national standard : IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2) :9–25, 1987.

[143] R. Strzodka. Virtual 16 bit precise operations on rgba8 textures. In *Proceedings of Vision, Modeling, and Visualization*, pages 171–178, 2002.

[144] J.A. Swenson and Y.N. Patt. Hierarchical registers for scientific computers. In *Proceedings of the International Conference on Supercomputing, 1988*, pages 346–353, 1988.

[145] Carmen Turinici, Christine Rochange, and Pascal Sainrat. Prédicteurs mixtes pour l'anticipation des instructions . In *5ème Symposium sur les Architectures Nouvelles de Machines (SYMPA'5) , Rennes, 08/06/99-11/06/99*, pages 165–174. INRIA, juin 1999.

[146] 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.

[147] S. Vassiliadis, J. Phillips, and B. Blanner. Interlock collapsing alu's. *IEEE Transactions on computer*, 42(7) :825–839, 1993.

[148] Alexander V. Veidenbaum, Qingbo Zhao, and Abduhl Shameer. Non-sequential instruction cache prefetching for multiple-issue processors. *International Journal of High Speed Computing*, 10(1) :115–140, 1999.

[149] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision and range independent tool for testing floating-point arithmetic I : basic operations, square root and remainder. *ACM Transactions on Mathematical Software*, 27(1) :92–118, 2001.

[150] Amit Verma, Ajay K. Verma, Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. Synthesis of floating-point addition clusters on FPGAs using carry-save arithmetic. In *FPL*, pages 19–24. IEEE, 2010.

[151] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[152] Zheng Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[153] S. Weiss and J.E. Smith. Instruction issue logic for pipelined supercomputers. In *Proceedings of the 11th International Symposium on Computer Architecture, 1984*, pages 110–118, 1984.

[154] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, 31(2) :50–59, March/April 2011.

[155] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS* [7].

[156] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower : a cycle-accurate energy estimation tool. In *DAC '00 : Proceedings of the 37th conference on Design automation*, pages 340–345, New York, NY, USA, 2000. ACM.

[157] Yuji Yoshida, Masami Yasuda, Jun ichi Nakagami, and Masami Kurano. A new evaluation of mean value for fuzzy numbers and its application to american put option under uncertainty. *Fuzzy Sets and Systems*, 157(19) :2614 – 2626, 2006.

[158] R. Yung and N. Wilhelm. Caching processor general registers. In *Proceedings of the International Conference on Computer Design, 1995*, pages 307–312, 1995.

[159] L.A. Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy Sets and Systems*, 11(1–3) :197 – 198, 1983.

[160] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Hierarchical clustered register file organization for VLIW processors. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 77, 2003.

Résumé

L'optimisation du calcul passe par une gestion conjointe du matériel et du logiciel. Cette règle se trouve renforcée lorsque l'on aborde le domaine des architectures multicœurs où les paramètres à considérer sont plus nombreux que sur une architecture superscalaire classique. Ces architectures offrent une grande variété d'unité de calcul, de format de représentation, de hiérarchie mémoire et de mécanismes de transfert de donnée.

Dans ce mémoire, nous décrivons quelques uns de nos résultats obtenus entre 2004 et 2013 au sein de l'équipe DALI de l'Université de Perpignan relatifs à l'amélioration de l'efficacité du calcul dans sa globalité, c'est-à-dire dans la suite d'opérations décrite au niveau algorithmique et exécutées par les éléments architecturaux, en nous concentrant sur les processeurs graphiques.

Nous commençons par une description du fonctionnement de ce type d'architecture, en nous attardant sur le calcul flottant. Nous présentons ensuite des implémentations efficaces d'opérateurs arithmétiques utilisant des représentations non-conventionnelles comme l'arithmétique multiprécision, par intervalle, floue ou logarithmique. Nous continuerons avec nos contributions relatives aux éléments architecturaux associés au calcul à travers la simulation fonctionnelle, les bancs de registres, la gestion des branchements ou les opérateurs matériels spécialisés. Enfin, nous terminerons avec une analyse du comportement du calcul sur les GPU relatif à la régularité, à la consommation électrique, à la fiabilisation des calculs ainsi qu'à la prédictibilité.

Abstract

Optimization of computation goes through joint management of hardware and software. This fact is reinforced when addressing the area of multicore architectures where the parameters to be considered are more numerous than on a conventional superscalar architecture. These architectures offer a wide variety of computational unit, representation format, memory hierarchy and data transfer mechanisms.

In this manuscript, we describe some of our results obtained between 2004 and 2013 within the DALI team from the University of Perpignan related to improving computational efficiency in its globality, that is to say in the sequence of operations described at algorithmic level and executed by architectural elements, with a focus on graphics processors.

We begin this manuscript with a description of those architectures and their floating-point performance. Then, we deal with the efficient implementations of arithmetic operators relying on non-conventional representations such as multiprecision, interval, fuzzy and logarithmic arithmetic. Then, we will study some architectural elements associated with computation such as functional simulation, register bank, branch management and specialized evaluation units. Finally, we present our contributions related to the behavior' analysis of computation on GPUs related to regularity, power consumption, error resilience and predictability.