Étude comparée et simulation d'algorithmes de branchements pour le GPGPU

Caroline Collange, Marc Daumas, David Defour, David Parello

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Étude comparée et simulation d'algorithmes de branchements pour le GPGPU

Sylvain Collange, Marc Daumas, David Defour & David Parello

ELIAUS, Université de Perpignan Via Domitia
52 avenue Paul Alduy — 66860 Perpignan — France
prenom.nom@univ-perp.fr

Résumé
L'accélération d'applications généralistes à l'aide de processeurs graphiques (GPGPU) permet d'obtenir de bonnes performances pour un faible investissement. Toutefois, la structure interne de ces processeurs est largement inconnue et les compteurs de performances ne sont que peu ou pas accessibles. Cette absence contraint fortement les développeurs dans les optimisations qu'ils peuvent mettre en place pour ces architectures. Une solution est de simuler le comportement d'un programme afin de recueillir des statistiques d'exécution. Dans cet article nous présentons d'une part un simulateur fonctionnel de processeur graphique ciblé pour le GPGPU basé sur l'environnement de simulation modulaire UNISIM prenant en entré du code binaire CUDA pour processeurs NVIDIA. D'autre part, nous exposons la problématique de la gestion des branchements par prédication pour les architectures SIMD et effectuons un état de l'art des différentes solutions adoptées par les principaux constructeurs de GPU. Enfin, nous proposons une technique matérielle originale de suivi des branchements permettant de simplifier le jeu d'instructions utilisé, et la validons au moyen du simulateur.

Mots-clés : GPU, simulation, SIMD, prédication, flot de contrôle

1. Introduction

L'apparition de langages de programmation de haut niveau pour les processeurs graphiques (GPU), tels que celui de l'environnement CUDA de NVIDIA, a renforcé l'intérêt des GPU pour l'accélération de tâches habituellement accomplies par des processeurs généralistes (CPU). Malgré ces nouveaux langages, il est difficile d'exploiter efficacement ces architectures complexes. En effet, les cartes graphiques évoluent rapidement, chaque génération apportant son lot de nouvelles fonctionnalités dédiées à l'accélération de routines graphiques ou au calcul hautes performances. Les détails architecturaux de ces architectures restent en grande partie secrets, les constructeurs étant réticents à divulguer les implémentations utilisées. Ces nouvelles caractéristiques ajoutées aux GPU sont le résultat de la simulation de différentes solutions architecturales effectuées par les fabricants pour en déterminer leur validité et leur performance.
La complexité et les performances des GPU actuels présentent d'importants défis pour tout ce qui touche à l'exploration de nouvelles solutions architecturales ou à la modélisation fine de certaines parties du processeur. Les concepteurs de processeurs généralistes (CPU) disposent d'un choix important de simulateurs fonctionnels ou de simulateurs cycle-à-cycle pour en étudier les effets. Cependant, les simulateurs de GPU disponibles librement ciblant la problématique GPGPU sont plus rares.
Nous présentons dans cet article un simulateur modulaire de GPU dans le cadre d'applications GPGPU. Ce simulateur, basé sur UNISIM [3], s'appelle *Barra* [9]. Nous avons choisi UNISIM comme plateforme de simulation afin de bénéficier de la modularité de cet environnement et des modules déjà développés (mémoire, unités fonctionnelles, générateur de décodeur d'instruction…) L'environnement proposé se découpe en deux grandes parties : le simulateur de la structure matérielle et des unités fonctionnelles du GPU, et le pilote qui gère le chargement du programme, les tâches liées à l'environnement et émule le pilote du GPU. Barra prend en charge le jeu d'instructions (ISA) des processeurs NVIDIA en raison de la dominance de CUDA dans le domaine du GPGPU. Ce choix n'est pas sans présenter de défis. En effet, NVIDIA ne documente pas son jeu d'instructions, nécessitant un effort de rétro-ingénierie parallèlement au développement du simulateur.

NVIDIA distribue plusieurs outils pour faciliter le développement ou la collecte d'informations sur l'exécution, tel qu'un mode émulation, un *profiler* ou un débogueur. L'intérêt de Barra par rapport à ces outils propriétaires sont la flexibilité, la possibilité de modifications et la vitesse de simulation. Cet outil permet de collecter des traces d'exécution et des statistiques sur les instructions exécutées.

Un aperçu des techniques et outils de simulation est donné dans la section 2. L'environnement NVIDIA CUDA est quant à lui décrit dans la section 3. Une description générale de l'environnement de simulation Barra est présentée dans la section 4. Les algorithmes de branchements et leurs validations sont respectivement décrits dans les sections 5 et 6.

2. Simulation

L'apparition de simulateurs de processeurs pour les architectures superscalaires dans les années 1990 fut le point de départ de nombreux travaux de recherches académiques et industriels en architecture des processeurs. La granularité de la 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 communications 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 [5] 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 et la difficulté d'en modifier le comportement. Afin de pallier à ces problèmes, des alternatives ont été proposées dans le domaine de la simulation d'architectures multicœurs [19] ou de systèmes complets [7, 18, 28]. Des simulateurs de GPU axés sur le fonctionnement du pipeline graphique ont aussi été proposés : Attila [20] pour la simulation au niveau cycle ou Qsilver [30] pour la simulation au niveau transaction. Les architectures et les problématiques considérés par ces simulateurs sont toutefois éloignés du GPGPU. Plus récemment, un simulateur de processeur *many-core* pour le GPGPU basé sur une version de SimpleScalar modifiée pour accepter le langage intermédiaire NVIDIA PTX a été proposé [6]. Il est cependant difficile de déterminer à quel point l'architecture considérée est représentative des GPU actuels ou futurs, ainsi que de l'influence que le choix d'utiliser un langage de haut niveau en lieu et en place de l'ISA peut avoir sur la précision de la simulation.

2.1. L'environnement de simulation modulaire UNISIM

Le développement de nouveaux simulateurs bénéficie aujourd'hui de nombreux environnements dédiés et modulaires [3, 4, 27]. 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*.

La modularité de ces environnements repose sur des interfaces communes et compatibles afin de permettre le partage et la réutilisation des modules. Certains imposent un protocole de communication pour distribuer la logique de contrôle dans les modules comme dans LSE [4], MicroLib [27] et UNISIM [3].

L'environnement UNISIM dispose d'un outil de gestion des jeux d'instructions GenISSLib. Celui-ci génère un décodeur d'instructions à partir de la description d'un jeu d'instructions. Le décodeur ainsi généré est construit autour d'un cache conservant les instructions pré-décodées. Lors du premier décodage, chaque instruction binaire est décodée et ajoutée au cache, afin que les exécutions ultérieures de la même instruction puissent accéder directement aux champs pré-décodés dans le cache d'instructions.

Avec l'augmentation de la complexité du matériel et du logiciel simulé, les performances des simulateurs deviennent critiques. Deux solutions ont été proposées pour résoudre ce problème. Toutes les deux reposent sur un compromis entre précision et vitesse de simulation. La première solution est la technique d'échantillonnage [32] adaptée pour la simulation d'un seul thread. La deuxième solution est quant à elle, plus adaptée pour la simulation de systèmes complets ou multicœurs. Elle propose de modéliser l'architecture à un plus haut niveau avec moins de détails que le niveau cycle : transaction-level modeling (TLM) [29]. À notre connaissance, UNISIM est l'unique environnement modulaire proposant à la fois une modélisation au niveau cycle et au niveau TLM basée sur le standard SystemC [12]. Enfin, de nouvelles techniques [26] ont été proposées pour améliorer la simulation au niveau

cycle d'architectures multicœures.

3. L'environnement CUDA

L'environnement CUDA (Compute Unified Device Architecture), développé par NVIDIA [24] est un environnement orienté calcul vectoriel hautes performances. Il repose sur une architecture, un langage, un compilateur, un pilote et divers outils et bibliothèques. Dans un programme CUDA typique, les données sont envoyées de la mémoire centrale vers la mémoire GPU, le CPU envoie des commandes au GPU, ensuite le GPU exécute les noyaux de calcul en ordonnançant le travail sur le matériel disponible pour enfin copier le résultat de la mémoire GPU vers la mémoire CPU.

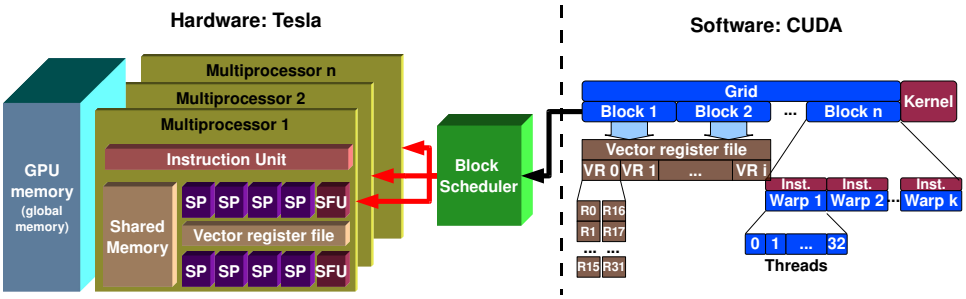

FIG. 1 – Exécution d'un programme CUDA.

L'architecture matérielle utilisée par CUDA se compose d'un processeur hôte et d'une mémoire hôte, d'une carte graphique NVIDIA prenant en charge CUDA. Les GPU prenant en charge CUDA sont actuellement tous basés sur l'architecture Tesla. Les GPU basés sur cette architectures exécutent en parallèle plusieurs milliers d'instances (threads) d'un code unique (modèle SPMD). Ce parallélisme est exploité par l'utilisation combinée de multicœur (CMP), de multithreading simultané (SMT) et de SIMD [16]. La figure 1 présente l'organisation matérielle de l'architecture Tesla.

L'organisation matérielle est liée à l'organisation du parallélisme défini au niveau du langage dans CUDA. Le langage de programmation est basé sur le langage C avec des extensions pour indiquer si une fonction est exécutée sur le CPU ou le GPU. Les fonctions exécutées sur le GPU sont aussi appelées *kernels*. CUDA donne la possibilité au développeur de définir des variables résidant dans l'espace d'adressage du GPU et le type de parallélisme pour l'exécution des kernels en terme de dimension de *grille*, *bloc*, et *thread*. CUDA repose sur une architecture intrinsèquement vectorielle. Une instruction est donc exécutée sur un groupe de threads que NVIDIA appel *warp* correspondant aussi à la largeur des registres vectoriels fixée par construction à 32. À chaque cycle une instruction d'un warp donné est exécutée par un multiprocesseur. Certaines instructions ont des latences importantes, aussi les instructions de chaque warp s'exécutent à leur propre rythme. Lorsque les kernels n'utilisent pas de mémoire partagée, le nombre de warps pris en charge par un multiprocesseur est égal à la taille du banc de registres divisé par le nombre de registres nécessaires pour exécuter le kernel. Les warps sont regroupés en blocs qui sont ordonnancés sur les multiprocesseurs disponibles pour exécution. Un multiprocesseur peut prendre en charge plusieurs blocs simultanément si les ressources matérielles le permettent en termes de registres et de mémoire partagée.

Le processus de création d'un exécutable CUDA comprend 3 étapes liées au compilateur CUDA *nvcc* [22]. En premier, le programme est divisé en une section CPU et une section GPU en suivant des directives insérées par le programmeur. La partie CPU est compilée par le compilateur de l'hôte, tandis que la partie GPU est compilée à l'aide d'un compilateur spécifique ciblant le jeu d'instruction natif Tesla[1]. Le code GPU résultant est un binaire CUDA (fichier *cubin*) composé du programme et des données pour l'exécution sur GPU. La classe des architectures NVIDIA pour lesquelles un fichier CUDA doit être compilé peut être changée par une option du compilateur nvcc.

[1] Le compilateur peut également générer du code dans un langage intermédiaire nommé PTX, qui sera compilé à la volée vers l'architecture cible par un compilateur JIT.

Les deux programmes CPU et GPU sont ensuite liés avec les bibliothèques CUDA qui contiennent les fonctions pour charger le code binaire cubin et l'envoyer sur le GPU pour exécution.
L'environnement CUDA intègre également des outils propriétaires tels qu'un mode émulation qui compile un programme CUDA pour une exécution complète sur CPU. Ce mode donne accès aux fonctions de bibliothèques et appels systèmes tels que *printf* au sein de kernels, ce qui facilite le débogage de programmes. En revanche, le comportement et les résultats entre le mode émulation et l'exécution sur GPU peuvent différer, notamment en termes de comportement de l'arithmétique flottante et entière, de granularité et ordre d'exécution des threads et de synchronisation. Pour une exécution plus précise, NVIDIA propose un débogueur pour CUDA depuis CUDA 2.0 [23].

4. Environnement de simulation Barra

Notre simulateur Barra est basé sur l'environnement UNISIM décrit dans la section 2.1. Il est distribué sous licence BSD, disponible en téléchargement[2] et est désormais intégré dans l'environnement UNISIM dans la branche `/unisim/devel/unisim_simulators/cxx/tesla`.
Il est conçu autour d'une plate-forme de simulation minimisant les modifications nécessaires pour simuler l'exécution d'un programme cubin au niveau fonctionnel comparativement à une exécution classique sur GPU NVIDIA. Toutes les instructions de l'ISA Tesla nécessaires à la simulation complète des principaux exemples du kit de développement CUDA SDK listé en section 6 sont pris en charge.

4.1. Pilote Barra

L'environnement Barra est conçu de telle sorte qu'il prenne la place du GPU sans nécessiter de modification des programmes CUDA. Un pilote constitué d'une bibliothèque partagée avec les mêmes noms que la bibliothèque propriétaire *libcuda.so* capture les appels destinés au GPU et les reroute vers le simulateur. Ainsi l'utilisateur peut choisir une exécution sur GPU ou sur le simulateur en positionnant la variable d'environnement liée aux bibliothèques partagées (`LD_LIBRARY_PATH`) sur le répertoire de son choix. Le pilote Barra proposé inclut les principales fonctions de l'API CUDA bas-niveau (Driver API) afin que tout programme CUDA puisse être chargé, décodé et exécuté sur le simulateur.
Ce pilote permet à l'utilisateur d'activer de manière sélective la génération de traces d'exécution (pour chaque instruction exécutée) et de statistiques (pour chaque instruction statique). Les utilisateurs intéressés par la collecte de données supplémentaires sur l'utilisation des unités fonctionnelles, la bande passante mémoire nécessaire pour atteindre le maximum de performance, la précision des calculs ou les goulets d'étranglement au niveau des registres peuvent facilement intervenir au sein du simulateur pour développer leurs propres compteurs de performance.
Même si le modèle mémoire CUDA est composé de plusieurs mémoires séparées (constantes, locale, globale, partagée), et que l'architecture matérielle Tesla contient des espaces mémoires séparés physiquement (DRAM et mémoire partagée), nous avons fait le choix de n'utiliser qu'un seul espace d'adressage physique pour tous les types de mémoires. Actuellement Barra ne fait pas de différences entre adresses virtuelles et adresses physiques. Dans le futur, il devrait intégrer un mécanisme de traduction des adresses virtuelles en adresses physiques afin de disposer d'une vérification précise des adresses, de pouvoir gérer les contextes CUDA multiples et de modéliser les performances des TLB.
L'utilisateur fixe le type de parallélisme en définissant les dimensions des grilles et blocs à l'aide des fonctions *cuFunctionSetBlockShape* et *cuLaunchGrid*. Ces paramètres sont ainsi passés aux kernels dans un espace d'adressage statique localisé dans la mémoire partagée, tandis que l'identificateur de threads est écrit dans le registre général (GPR) 0 avant que l'exécution ne démarre. La seule information connue du matériel (le cœur du simulateur) est un pointeur sur une fenêtre en mémoire partagée pour chaque warp. Lors de l'initialisation d'un multiprocesseur, la mémoire partagée est divisée en un nombre de fenêtres égal au nombre de blocs maximum pouvant s'exécuter en parallèle sur un multiprocesseur, et le pointeur sur la mémoire partagée appartenant à chaque bloc est défini.
Sur le GPU, les blocs de threads sont exécutés en parallèle à l'aide des multiprocesseurs disponibles en matériel suivant un ordonnancement *round-robin*. La simulation fonctionnelle procède de la même manière, en considérant qu'un seul multiprocesseur est présent de manière à forcer un ordre d'exécution séquentiel et déterministe.

[2] http://gpgpu.univ-perp.fr/index.php/Barra

4.2. Barra et le décodage de l'ISA Tesla

NVIDIA, contrairement à AMD [1], ne publie pas de documentation de son jeu d'instructions. Toutefois, grâce au travail effectué dans le projet decuda [31] et à nos propres tests, nous avons pu collecter l'information nécessaire pour l'intégration du jeu d'instruction Tesla première génération (Compute Model 1.0). Ces instructions couvrent l'ensemble des instructions utilisées dans les kernels CUDA tels que les add, mul, mad, mac, mov et conversion en flottant et entier, calcul booléen et décalages, scatter et gather mémoire, instructions de contrôle...

L'architecture Tesla inclut un jeu d'instructions 64 bits à 4 opérandes avec certaines instructions pouvant être compactées sur un mot de 32 bits. Un autre type d'encodage permet d'encoder une valeur immédiate de 32 bits dans un mot d'instruction de 64 bits. Le compilateur garantit l'alignement sur 64 bits en groupant les instructions courtes de 32 bits par paires. Plusieurs champs dans l'encodage des instructions courtes se trouvent placés au même endroit dans les instructions longues, ce qui simplifie l'étape de décodage.

Barra s'appuie sur la bibliothèque GenISSLib inclue dans UNISIM pour le décodeur d'instructions. Il est possible avec cette bibliothèque d'utiliser plusieurs sous-décodeurs pour chaque sous-opération d'un jeu d'instructions de type CISC. Cependant cette caractéristique nécessite que chaque sous-opération soit constituée de champs contigus, ce qui n'est pas le cas dans l'ISA Tesla. Nous générons donc six décodeurs RISC séparés travaillant tous sur le mot d'instruction complet de 64 bits et responsables chacun du décodage d'une partie de l'instruction.

4.3. Exécution des instructions

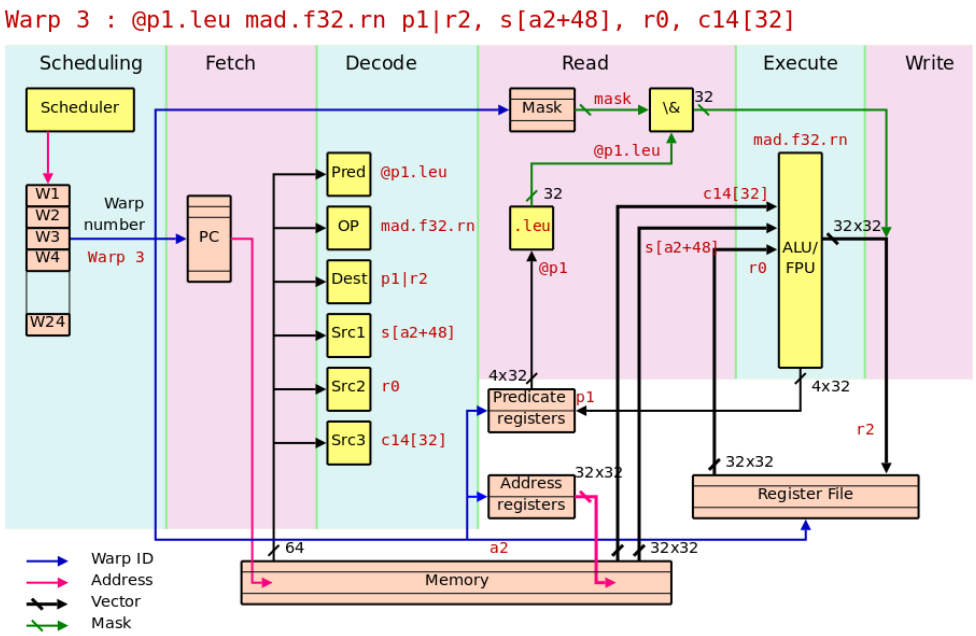

FIG. 2 – Vue du pipeline fonctionnel pour l'exécution d'une instruction MAD.

Les instructions sont exécutées dans Barra en suivant un modèle décrit dans la figure 2. Premièrement l'ordonnanceur sélectionne le prochain warp pour exécution et récupère le compteur de programme (PC) correspondant. L'instruction est alors chargée et décodée tel que décrit dans la section 4.2. L'instruction décodée est ensuite mémorisée dans un tampon dédié pour accélérer les futurs accès à la même instruction. Cette optimisation permet de réduire le temps de simulation. Les opérandes sont alors lus depuis le banc de registres, la mémoire partagée ou la mémoire de constantes. Les différentes mémoires disponibles en CUDA sont physiquement unifiées sous Barra ce qui ne nécessite qu'un seul mécanisme d'accès. Ensuite, l'instruction est exécutée et le résultat est renvoyé dans le banc de registres.

Les instructions sur les entiers peuvent mettre à jour un registre de drapeaux contenant une information de retenue,

débordement, zéro ou valeur négative comme décrit dans [21]. Des vérifications et validations par l'expérimentation montrent que ce comportement correspond à celui reproduit dans Barra. L'arithmétique entière sur 16 bits et l'indexage de demi-registres sont également pris en charge.

L'arithmétique flottante est plus complexe à reproduire et plus particulièrement dans le cas des GPU [8]. Barra propose deux modes d'émulation de l'arithmétique flottante. Le premier mode repose sur l'utilisation directe des unités du CPU pour traiter les flottants. Ce mode présente l'avantage d'être rapide et de bénéficier de la gestion matérielle des modes d'arrondis du CPU. Le second environnement repose sur la bibliothèque d'émulation de l'arithmétique flottante Simfloat++ intégrée dans UNISIM. Ces modes permettent tous deux de reproduire le comportement des unités matérielles comme les MAD tronqués ou les dénormaux arrondis à 0 et délivrent des résultat corrects au bit près par rapport au GPU pour les opérateurs arithmétiques usuels. Disposer de deux implantations indépendantes permet d'effectuer aisément des vérifications croisées.

En plus des instructions de calcul usuelles, les GPU actuels offrent des instructions d'évaluation de fonctions élémentaires (exponentielle et logarithme, sinus et cosinus, inverse et racine carrée inverse), qui repose sur l'utilisation d'une unité spécifique. Cette unité appelée *Special Function Unit* (SFU), décrite dans [25], intègre des opérateurs dédiés ainsi que des données tabulées. Une simulation exacte de ce type d'unité nécessiterait soit une tabulation de tous les résultats possibles obtenus par des tests exhaustifs conduisant à une consommation de stockage et de mémoire de l'ordre du gigaoctet, soit une simulation au bit près des chemins de données et des tables de l'unité matérielle nécessitant un effort de rétro-ingénierie conséquent. Par conséquent, la simulation des unités SFU dans Barra repose actuellement sur une réduction d'argument identique à la version GPU suivie d'un appel à la fonction correspondante de la bibliothèque mathématique C standard.

4.4. Simulation du matériel dans Barra

4.4.1. Gestion des registres

Les registres généraux (GPU) 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 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 masque et d'adresse, 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 [15] afin de minimiser les conflits de bancs. La politique de partitionnement du banc de registres n'influe pas sur le comportement du simulateur fonctionnel. De fait, l'algorithme utilisé dans Barra repose sur une allocation séquentielle de blocs de registres dans un banc de registres unifié.

4.4.2. Ordonnancement des warps

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

4.5. Performance

Des tests de temps de simulation montrent que le surcoût de la simulation sur un Core2 Duo E8400 par rapport à l'exécution sur GPU GeForce 9800 GX2 est un facteur de 300 à 3 000, avec une moyenne de 1 300 sur les programmes du SDK CUDA (table 1). Par comparaison, le mode d'émulation et le débogueur de CUDA offrent des surcoûts moyens respectifs de 7 130 et 49 000, soit prés d'un et deux ordres de grandeur de plus.

5. Gestion des branchements

5.1. Problématique

Le traitement des branchements est un des enjeux principaux dans le cadre de la vectorisation de code pour des architectures SIMD. En effet, il est parfois nécessaire d'effectuer un traitement différent pour différentes composantes d'un vecteur SIMD en fonction des données, par exemple lorsqu'un structure *if-then-else* est vectorisée.

Ainsi, les extensions SIMD des jeux d'instructions généralistes tels que Intel SSE et IBM Altivec comportent des instructions de comparaison vectorielle calculant indépendamment une condition pour chaque élément des vecteurs source. Le résultat de cette comparaison vectorielle forme un masque de valeurs booléennes, qui peut ensuite être utilisé par des instructions booléennes tels que AND, NAND et OR pour masquer sélectivement les résultats à conserver parmi les résultats de l'exécution des deux branches de la condition.

Cette solution est bien adaptée aux architectures superscalaires avec des vecteurs courts, mais présente l'inconvénient de nécessiter l'exécution des deux branches de la condition dans tous les cas, y compris lorsque cette condition est évaluée à la même valeur pour tous les éléments du vecteur. Cela rend l'utilisation de cette technique inenvisageable pour la vectorisation systématique de flot de contrôle complexe contenant des structures conditionnelles imbriquées et des boucles.

Les GPU actuels offrent des mécanismes matériels basés sur la prédication pour exécuter un flot de contrôle arbitraire en SIMD de manière transparente. Ces techniques sont à rapprocher des implémentations plus anciennes utilisées dans les architectures SIMD [17]. D'autres techniques reposant sur la formation dynamique de warps plutôt que sur la prédication ont également été proposées [14, 11].

Les deux paragraphes suivants présentent respectivement les techniques utilisées par les principaux constructeurs de processeurs graphiques NVIDIA, Intel et AMD pour gérer la divergence des branchements et une nouvelle technique que nous proposons. Nous verrons que la difficulté principale des différentes solutions proposées ne réside pas tant dans la gestion des points de divergence du flot de contrôle que dans celle des points de reconvergence.

Gestion des branchements dans les GPU NVIDIA

Des tests que nous avons effectués ont montré que l'algorithme utilisé sur les GPU NVIDIA est très proche de celui décrit dans un brevet [10].

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. Pour les branchements on trouvera deux types de jetons sur le sommet de la pile : un jeton `DIV` indiquant que l'on se trouve dans un état de divergence et un jeton `SYNC` indiquant que l'on se trouve dans un état de synchronisation.

La figure 3 illustre sur un exemple la gestion des branchements au sein d'un multiprocesseur. L'état initial de la pile indique qu'il y a déjà eu une désynchronisation ou divergence (`DIV`) pour deux threads (sommet de la pile) et qu'une synchronisation (`SYNC`) de tous les threads (masque complet) sera nécessaire. La colonne PC présente la trace du compteur de programme actif. Le masque actif est le complémentaire du masque se trouvant sur le sommet de la pile. L'instruction de branchement `br` est annotée de `sync` indiquant un point de divergence et l'instruction `nop` est annotée de `join` indiquant un point de convergence.

Lorsque le PC est égal à 1 et que l'instruction de branchement est chargée, le bit de synchronisation (`sync`) est positionné pour l'instruction correspondante. Le multiprocesseur place un jeton de synchronisation sur la pile (`SYNC`) avec le compteur de programme de l'instruction suivante ainsi que le masque actif. Un jeton de divergence (`DIV`) est placé pour cette instruction avec le PC de l'instruction suivante et un masque égal à la conjonction de la condition du branchement et du masque actif. Le compteur de programme actif prend la valeur 4 (adresse du `ELSE`) pour ensuite exécuter les instructions jusqu'à l'adresse 7. Lorsque l'instruction de `nop.join` est chargée, le sommet de la pile est lu. Le jeton `DIV` indique que l'on se trouve dans un état de divergence qui signifie que certains threads doivent encore être exécutés. Le compteur de programme et le masque du sommet du pile sont utilisés pour mettre à jour le compteur de programme et le masque actif, le multiprocesseur dépile le jeton `DIV`. Le multiprocesseur exécute les instructions du *if* correspondant à une exécution des instructions 2, 3, 6 et 7. Le branchement `br` d'adresse 3 est non conditionnel, il n'est donc pas un point de divergence. Lorsque l'instruction `nop.join` est chargée, le sommet de la pile est lu. Cette fois-ci le jeton est de type `SYNC` indiquant que la synchronisation peut avoir lieu. Le masque se trouvant sur le sommet de pile est utilisé pour remplacer le masque actif et le sommet est dépilé. La pile est revenue dans l'état initial, le programme peut poursuivre sont exécution. Notons que l'instruction `nop.join` a été exécutée deux fois.

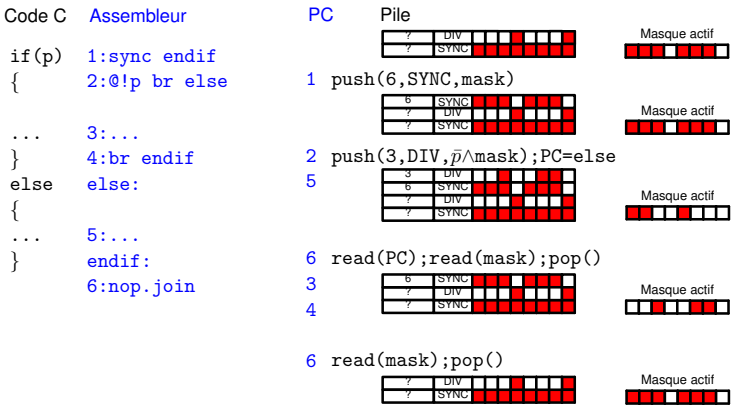

FIG. 3 – Branchement avec annotations (NVIDIA).

Intel

Les GPU Intel GMA [13] offrent directement dans leur ISA des instructions de contrôle reflétant les structures usuelles des langages de programmation : `if`, `else`, `endif`, `do`, `while`, `break`, `cont(inue)`.
Les langages de *shaders* utilisés pour programmer les GPU (Cg, HLSL, GLSL) ne permettant pas d'utiliser des instructions pouvant amener à un flot de contrôle non structuré telles que `goto`, cet ensemble est suffisant pour l'implantation des API graphiques OpenGL et Direct3D.
Leur sémantique est décrite en termes 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, 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`).

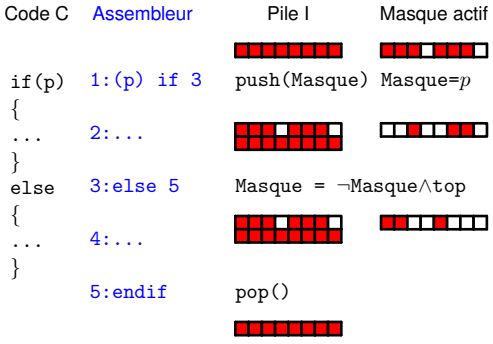

FIG. 4 – Exemple de branchement avec instructions explicites (Intel, AMD).

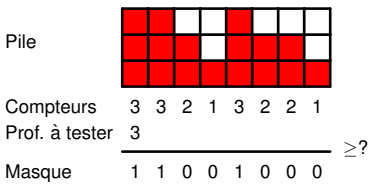

FIG. 5 – Réalisation d'une pile de masques à l'aide de compteurs.

Un point original de l'implantation d'Intel réside dans la structure de données utilisée pour stocker les piles de masques. On observe qu'un thread qui était inactif à une profondeur d'imbrication de structure de contrôle donnée ne peut pas devenir actif à une profondeur d'imbrication supérieure. En termes de masques, cela se traduit par le fait qu'un *0* présent dans un masque de la pile ne peut pas être recouvert par un *1*. Plutôt que d'allouer une entrée en mémoire pour chaque masque, il est donc possible de ne retenir que la hauteur de la colonne de *1* pour chaque voie de la pile au moyen de compteurs. Un autre compteur contient la hauteur courante de la pile, permettant de calculer le masque courant en comparant chaque compteur de voie avec la hauteur de la pile (fig. 5). Cette

structure de données n'est pas utilisable dans le cas de l'implantation de NVIDIA dont les masques ne respectent pas la contrainte imposée (exemple fig. 3).
Ces compteurs sont accessibles au programmeur ou au compilateur sous la forme de registres architecturaux, permettant leurs sauvegarde et restauration en cas de dépassement de capacité et la définition de flot de contrôle non-standard.

AMD

Les programmes exécutés par les GPU R600 et R700 d'AMD [2] sont composés de trois flots d'instructions distincts : un flot d'instructions arithmétiques, un flot d'instructions d'échantillonnage 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. Des instructions `call` et `return` sont également présentes, et utilisent une ou plusieurs entrées de la pile de masques pour stocker chaque adresse de retour. Cela exclut donc une implantation à base de compteurs.

5.2. Contrôle implicite

À notre connaissance, les méthodes utilisées sur les processeurs SIMD pour gérer les branchements de manière transparente nécessitent toutes l'annotation du code compilé pour indiquer explicitement les points de divergence et surtout les points de reconvergence. À l'inverse, la méthode que nous proposons permet de s'abstraire de ces annotations et d'exécuter directement du code SPMD compilé par un compilateur traditionnel.
Nous nous basons sur deux piles de masques implantées par compteurs et deux piles d'adresses, l'une servant aux structures conditionnelles (notée I) et l'autre aux boucles (notée L).

Structures conditionnelles

Lorsqu'un branchement vers l'avant divergeant est rencontré, le masque courant et l'adresse de destination sont sauvegardées dans les piles I, le masque courant est mis à jour et l'exécution continue dans la branche non-prise. Le haut de la pile d'adresses I contient alors la cible du saut, soit l'autre branche en attente d'être exécutée (fig. 6).
Après l'exécution de chaque instruction, le compteur d'instructions suivante (NPC) est comparé avec le haut de la pile d'adresses I.
En cas d'égalité, le point de convergence est atteint (`endif`), l'exécution peut alors continuer avec le masque sauvegardé, après avoir dépilé le masque et l'adresse. Plusieurs points de convergence peuvent être présents sur la même instruction. Dans ce cas, il faut répéter la comparaison et dépiler autant de fois que nécessaire, ce qui peut nécessiter la ré-exécution de l'instruction.
Il se peut également que l'instruction courante soit un saut vers une adresse se trouvant au-delà de la cible du saut précédant, par exemple avant un bloc `else` (ligne 4 de la figure 6). Auquel cas, NPC est strictement supérieur au sommet de la pile I. Le contrôle est alors transféré à l'autre branche en attente, en échangeant NPC avec le haut de la pile I et en inversant le masque courant.

Boucles

Les boucles sont identifiées par un branchement arrière, conditionnel ou inconditionnel. Lorsqu'un tel branchement est rencontré et avant d'effectuer le saut, on compare le sommet de la pile L avec l'adresse de l'instruction suivante (NPC=PC+1).
En cas de différence et si le saut est pris par au moins un thread, on empile NPC et le masque courant sur la pile L, de façon à retenir l'adresse de fin de la boucle.
En cas d'égalité et si aucun thread n'effectue le saut (sortie de boucle), le masque est restauré d'après le masque présent dans la pile et l'entrée correspondante est dépilée de la pile L.

Autres structures de contrôle

Les deux techniques présentées permettent de gérer efficacement les structures de contrôle strictement imbriquées. Si elles sont également suffisantes pour exécuter du contrôle de flot plus irrégulier, telles que les instructions `break` et `continue` du langage C, l'ordre d'exécution suivi dans ce cas sera sub-optimal.
Pour une exécution implicite plus efficaces de ces structures, il peut être nécessaire d'ajouter deux piles *ad-hoc* supplémentaires.

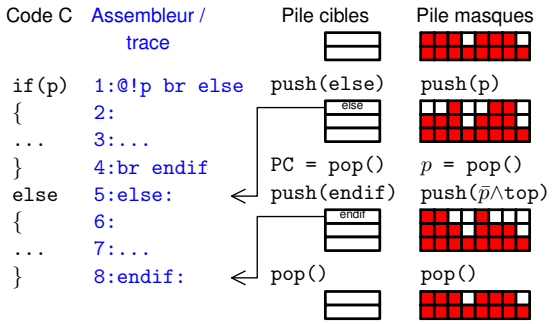

FIG. 6 – Exemple de branchement avec contrôle de flot implicite.

Programme	Kernels
binomialOptions	binomialOptions
BlackScholes	BlackScholes
convolutionSeparable	convolutionRow, convolutionCol
fastWalshTransform	fwtBatch1, fwtBatch2, modulate
histogram256	histogram256, mergeHistogram256
matrixMul	matrixMul
MersenneTwister	MersenneTwister, BoxMuller
quasiRandomGenerator	inverseCND
MonteCarlo	MonteCarlo
reduction	reduce5, reduce6
scanLargeArray	prescan1, prescan2, prescan3, uniformAdd
transpose	transposeNaive, transpose

TAB. 1 – Programmes de tests exécutés et kernels associés.

6. Validation et résultats

Nous avons utilisé les exemples du kit de développement NVIDIA CUDA SDK pour comparer les exécutions entre Barra et des exécutions réelles sur architectures Tesla. Nous avons testé avec succès les exemples du tableau 1. Les résultats obtenus sont identiques sauf pour les exemples utilisant les fonctions élémentaires conformément aux choix d'implémentation faits pour Barra.

En raison de disparités de comportements pouvant exister entre plusieurs kernels d'un même programme CUDA, nous avons choisi d'instrumenter séparément chaque kernel. Le même code binaire est exécuté avec l'algorithme de branchement NVIDIA et notre algorithme implicite. Dans ce dernier cas, les instructions `join` sont ignorées et ne sont pas inclues dans le décompte des instructions exécutées. Cependant, les instructions `nop` qui peuvent avoir été insérées pour satisfaire les contraintes du système de branchement explicite sont exécutées dans les deux cas.

La figure 7 présente le taux d'occupation moyen des unités SIMD par kernel suivant l'algorithme de branchement utilisé. Une valeur de 32 indique un taux d'occupation de 100%.

On remarque au vu des résultats que l'algorithme de branchement implicite est toujours au moins aussi efficace que l'algorithme explicite. Cela reste vrai dans une moindre mesure lorsque l'exécution des instructions `join` est prise en compte même en mode implicite. En effet, l'algorithme de NVIDIA nécessite d'exécuter deux fois les instructions se trouvant aux points de convergence.

On observe également une différence significative lors de l'exécution du kernel *inverseCND* de *quasiRandomGenerator*. Elle est due à la présence d'une instruction `return` à l'intérieur d'une imbrication de structures conditionnelles. En effet, l'algorithme décrit dans le brevet de NVIDIA ne permet pas de désactiver immédiatement les threads exécutant cette instruction.

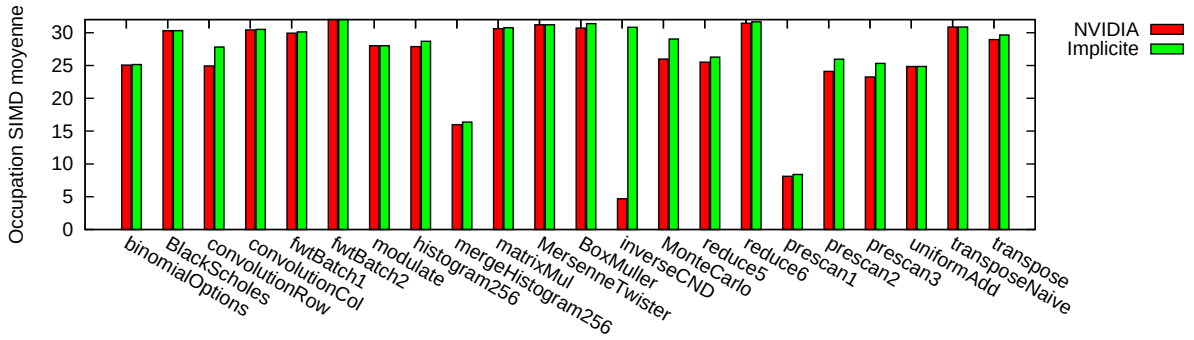

FIG. 7 – Occupation SIMD en fonction de l'algorithme de branchement utilisé.

Lors des tests nous n'avons remarqué aucun phénomène de reconvergences multiples, ce qui conforte l'intérêt de la technique de contrôle de flot implicite proposé dans la section 5.2.

7. Conclusion

Nous avons décrit dans cet article le pilote et le simulateur Barra ainsi que la problématique et différentes solutions aux problèmes des branchements sur les architectures SIMD. Nous avons montré que malgré l'absence de documentation du jeu d'instructions NVIDIA Tesla, il était possible d'émuler l'exécution complète d'un programme binaire CUDA complet au niveau fonctionnel. Le développement de Barra dans l'environnement UNISIM permet aux utilisateurs de modifier le simulateur et de réutiliser des modules ou des fonctionnalités disponibles dans UNISIM. Ce travail permet de tester l'efficacité de programmes sur différentes architectures sans avoir à réaliser les tests sur des architectures physiques. Il rend possible une meilleure compréhension des mécanismes architecturaux des GPU et des architectures multicœurs comme nous l'avons montré à travers la gestion de la divergence dans les branchements. Le support d'instructions et de fonctionnalités supplémentaires devrait être ajouté à Barra dans un futur proche.

Bibliographie

1. Advanced Micro Device, Inc. *AMD R600-Family Instruction Set Architecture*, December 2008.
2. Advanced Micro Device, Inc. *R700-Family Instruction Set Architecture*, March 2009.
3. 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.
4. 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.
5. Todd Austin, Eric Larson, and Dan Ernst. Simplescalar : An infrastructure for computer system modeling. *Computer*, 35(2) :59–67, 2002.
6. 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.
7. 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.
8. S. Collange, M. Daumas, and D. Defour. État de l'intégration de la virgule flottante dans les processeurs graphiques. *Revue des sciences et technologies de l'information*, 27/6 :719–733, 2008.
9. 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/.

10. 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.
11. 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.
12. The Open SystemC Initiative. Systemc.
13. Intel. *Intel® G45 Express Chipset Graphics Controller PRM, Volume Four : Subsystem and Cores*, February 2009. www.intellinuxgraphics.org.
14. 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.
15. 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.
16. John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla : A unified graphics and computing architecture. *IEEE Micro*, 28(2) :39–55, 2008.
17. Raymond A. Lorie and Hovey R. Strong. Method for conditional branch execution in simd vector processors. US Patent 4,435,758, March 1984. IBM.
18. 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.
19. 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.
20. 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.
21. NVIDIA. *NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x)*, February 2007. http ://developer.download.nvidia.com/opengl/ specs/g80specs.pdf.
22. NVIDIA. *The CUDA Compiler Driver NVCC, Version 2.0*, 2008.
23. NVIDIA. *CUDA-GDB : The NVIDIA CUDA Debugger, Version 2.1 Beta*, 2008.
24. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0*, 2008.
25. 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.
26. David Parello, Mourad Bouache, and Bernard Goossens. Improving cycle-level modular simulation by vectorization. *Rapid Simulation and Performance Evaluation : Methods and Tools (RAPIDO'09)*, 2009.
27. 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.
28. 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.
29. Gunar Schirner and Rainer Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *Trans. on Embedded Computing Sys.*, 8(1) :1–29, 2008.
30. 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.
31. Wladimir J. van der Laan. Decuda and cudasm, the cubin utilities package, 2009. http ://www.cs.rug.nl/~wladimir/decuda.
32. Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. Simflex : Statistical sampling of computer system simulation. *IEEE Micro*, 26(4) :18–31, 2006.