HAL
open science

Analyse de l'architecture GPU Tesla

Caroline Collange

To cite this version:

Caroline Collange. Analyse de l'architecture GPU Tesla. [Rapport de recherche] ELIAUS. 2010. hal-00443875

HAL Id: hal-00443875

https://hal.science/hal-00443875

Submitted on 4 Jan 2010

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

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

Analyse de l'architecture GPU Tesla

Caroline Collange
DALI, ELIAUS, Université de Perpignan

4 janvier 2010

Résumé

Les processeurs graphiques forment une architecture émergente encore peu comprise. Nous présentons ici une description du fonctionnement de l'architecture Tesla de NVIDIA et de ses caractéristiques propres. Les mécanismes d'ordonnancement de tâches à gros grain et à grain fin, d'exécution et d'accès mémoires sont considérés. Il apparaît que les objectifs de conception de cette architecture sont l'exploitation du parallélisme de données, l'optimisation du débit aux dépends de la latence et de la localité, l'absence de mécanisme spéculatif, mais aussi la conservation d'un modèle de programmation accessible y compris au niveau architectural.

1 Introduction

Les processeurs graphiques (GPU) forment une nouvelle architecture parallèle ayant suivi une évolution extrêmement rapide en l'espace de quelques années. À l'inverse des architectures parallèles des années 1990, et à l'instar des microprocesseurs haute-performances grand-public, ils forment un marché de volume, et leur conception est associée à des enjeux économiques importants. Leur intégration en cours au sein des processeurs généraliste augure une poursuite additionnelle de leur démocratisation. Au delà de leur destination première, l'essor du calcul généraliste sur processeur graphique ou GPGPU[1] à fait prendre conscience autant aux constructeurs qu'aux développeurs et aux utilisateurs de la pertinence d'accélérateurs dédiés au calcul parallèle au delà de la sphère du rendu graphique.

Malgré cette combinaison de facteurs, la compréhension des spécificités de ce type d'architecture tout comme les outils pouvant faciliter leur conception sont encore peu répandus parmi la communauté de l'architecture des ordinateurs. En effet, si de nombreuses similitudes avec des travaux antérieurs portant sur des supercalculateurs peuvent être observées, les constantes évolutions technologiques, scientifiques voire économiques intervenues depuis ont déplacé certain des enjeux intervenant dans la conception d'une architecture parallèle dédiée au calcul.

Or, des travaux de modélisation de performances ou d'exploration micro-architecturale ne peuvent s'envisager sans une bonne compréhension des architectures existantes et des pratiques suivies dans l'industrie. Nous nous proposons dans cet article d'analyser en détail un GPU vu comme un processeur parallèle de calcul généraliste, dans le but d'en dégager les enjeux intervenant dans sa conception.

[1] `http://gpgpu.org`

2 Tesla

Nous nous pencherons dans cet article sur l'architecture des GPU NVIDIA débutée avec le G80 (GeForce 8800 GTX) en 2006, et dont le dernier dérivé remonte à fin 2009. La base architecturale commune de ces GPU est dénommée *Tesla* [12]. Cette appellation correspond à un nom de code associé à l'architecture, et n'a pas de relation directe avec le nom commercial Tesla désignant la gamme destinée au calcul scientifique du même constructeur.

En tant que partie intégrante de la plate-forme CUDA, cette architecture a eu une influence majeure sur le développement du GPGPU, au point de devenir un standard *de facto*. En novembre 2009, NVIDIA recensait ainsi près de 700 travaux académiques et industriels basés sur CUDA et donc Tesla[2].

Cela en fait l'architecture privilégiée à analyser, modéliser et étendre. Néanmoins, la documentation officielle se contente de présenter un modèle simplifié de l'architecture, sans décrire son fonctionnement. Un article des concepteurs fournit également un aperçu plus détaillé [12]. S'il est défendable que ces ressources soient suffisantes pour développer des applications efficaces ciblant cette architecture, il n'en va pas de même pour les travaux visant à analyser et modéliser ce type d'architecture.

Des travaux expérimentaux présents dans la littérature tentent de reconstituer les caractéristiques de certains GPU de l'architecture Tesla au moyen de micro-tests. Volkov et Demmel [23] étudient les latences et les débits des transferts mémoire et des unités de calcul pour optimiser des algorithmes d'algèbre linéaire, sur plusieurs GPU du G80 au GT200. Papadopoulou, Sadooghi-Alvandi et Wong [20] font une étude détaillée des hiérarchies mémoires sur le GT200.

Ces travaux offrent une compréhension qualitative des structures architecturales internes au GPU. Cependant, l'utilisation de l'horloge dite *SP* comme seule référence ne permet pas d'estimer les temps de traversée des composants dans les autres domaines d'horloge. Une autre difficulté des mesures de performances sur les GPU est la constante évolution des pilotes qui remettent régulièrement en cause les résultats obtenus. Au-delà des mesures, il est donc nécessaire de construire un modèle paramétrable de l'architecture.

Nous tenterons ici de compléter ces travaux en offrant une vision d'ensemble, combinant un aperçu des données connues avec nos propres résultats expérimentaux et hypothèses. Nous commencerons par une description au niveau architectural, puis examinerons point par point la micro-architecture.

3 Jeu d'instruction

Au niveau architectural, les unités de calcul de tous les GPU Tesla reconnaissent le même jeu d'instruction de base, mais des extensions sont introduites ou retirées en fonction des révisions de l'architecture ou de la gamme de produit.

Ces révisions de l'architecture sont désignées sous le nom de *Compute Model* par NVIDIA et sont représentées sous forme d'un numéro de version. Le premier chiffre indique l'architecture, et le second le numéro de révision de l'architecture. Chaque révision englobe les fonctionnalités apportées par l'ensemble des révisions antérieures. La table 1 liste les révisions de l'architecture et leurs principales fonctionnalités.

Il n'existe pas à ce jour de documentation officielle du jeu d'instruction Tesla. Cependant, la majorité du jeu d'instruction a été retrouvée par rétro-ingénierie, notam-

[2] `http://www.nvidia.com/cuda`

TAB. 1 – Révisions de l'architecture.

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

ment grâce au projet *decuda* dés 2007[3], ou plus récemment à *nv50dis* associé au projet *Nouveau*[4].

Ce jeu d'instruction partage de nombreuses similitudes avec les langages intermédiaires issus de la compilations des shaders de Direct3D (assembleur HLSL[5]), OpenGL (ARB Fragment/Vertex/...Program[6]) ou CUDA (PTX [17]). En particulier, il abstrait presque complètement la largeur SIMD architecturale. Les instructions opèrent toutes sur des vecteurs, y compris les instructions de lecture/écriture mémoire, qui deviennent alors des instructions gather/scatter, et les instructions de comparaison et de branchement, qui reproduisent de manière transparente une exécution proche d'un modèle MIMD. Il n'offre aucune instruction de calcul scalaire, ni de registre scalaire, ni même d'opération de lecture ou écriture mémoire ; le jeu d'instruction est entièrement vectoriel. Ironiquement, c'est cette généralisation vectorielle qui permet à cette architecture d'être qualifiée de *scalaire* par son constructeur [12]. Du point de vue d'une voie SIMD, les calculs effectués sont indépendants des autres voies du vecteur SIMD, et peuvent se décrire comme si le mode d'exécution était scalaire, ou MIMD.

Ce modèle étant très proche de celui qui est exposé au programmeur par les langages de développement sur GPU, le processus de compilation reste aisé et efficace.

Pour conserver ce modèle de programmation et abstraire la largeur vectorielle, Tesla n'offre pas d'instruction de type swizzle ou autres modes de communication directe entre voies SIMD, à l'exception d'une instruction de réduction de booléens (vote) introduite avec la révision 1.2. Ce type d'opération doit donc être effectué en passant par la mémoire au moyen d'instructions gather et scatter. L'aspect multithreading simultané est également masqué, chaque thread possédant son propre ensemble de registres architecturaux privés.

Plusieurs types de registres architecturaux sont accessibles :
- registres généraux (GPR),
- registres de drapeaux,
- registres d'adresse.

Les registres de drapeaux permettent de stocker le résultat d'une comparaison ou la classification d'un résultat (zéro, négatif, retenue, débordement). Ces registres sont vectoriels, et peuvent contenir une valeur différente pour chaque voie SIMD. Associés à un code de condition, ils peuvent être utilisés par les instructions de branchements, mais aussi par la plupart des autres instructions, qui peuvent être prédiquées de manière généralisée par un registre de drapeaux et un code de condition.

Au niveau architectural, l'espace mémoire de Tesla est hétérogène, divisé en es-

[3] http://wiki.github.com/laanwj/decuda
[4] http://0x04.net/cgit/index.cgi/nv50dis
[5] http://msdn.microsoft.com/en-us/library/ee418149(VS.85).aspx
[6] http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt

TAB. 2 – Caractéristiques des cartes graphiques testées.

Nom commercial	GPU	Fréquence cœur (MHz)	Fréquence calcul (MHz)	Fréquence mémoire (MHz)
Tesla C870	G80	575	1350	800
GeForce 8500 GT	G86	459	918	400
GeForce 9800 GX2	G92	600	1512	1000
Prototype T10P	GT200	540	1080	900

paces distincts adressés explicitement. Chaque espace est accessible au moyen d'instructions distinctes. La décision de placement des données ne peut donc être faite qu'au moment de la compilation. En particulier, le compilateur doit être capable d'inférer statiquement l'espace mémoire désigné par chaque pointeur.

Ces espaces logiques comprennent :
- la mémoire de constantes,
- la mémoire de texture,
- la mémoire globale,
- la mémoire locale,
- la mémoire partagée.

Les mémoires de textures et de constantes sont héritées des langages de shaders pour la programmation graphique, qui y font référence explicitement. Effectuer la distinction entre ces espaces mémoire au niveau du jeu d'instruction est une décision naturelle permettant une mise en œuvre efficace de l'architecture et une traduction simplifiée pour le compilateur. En revanche, la séparation entre les mémoires partagée, globale et locale destinées principalement au calcul généraliste nécessite un effort supplémentaire de la part du compilateur (en C pour CUDA) ou du programmeur (en OpenCL).

En contrepartie, cette distinction permet de séparer nettement les zones mémoires en lecture seule (constantes, texture), locale à un thread (locale), à latence courte et déterministe (partagée). La latence de chaque type de mémoire pouvant être estimée précisément de manière statique, le compilateur bénéficie de cette information supplémentaire pour effectuer le choix et l'ordonnancement des instructions. Ainsi, on observe que le compilateur CUDA déplace les lectures en mémoire globale au plus tôt, quitte à nécessiter plus de registres, tandis qu'à l'inverse il fusionne les lectures en mémoire partagée avec des instructions de calcul, voire duplique ces lectures pour réduire l'utilisation des registres.

4 Protocole de test

Les tests ont été effectués sur les cartes graphiques basées sur Tesla décrites table 2. Celles-ci comprennent une carte Tesla destinée au calcul scientifique, une carte GeForce 8500 GT d'entrée de gamme, une carte bi-GPU haut de gamme 9800 GX2 et un prototype de Tesla C1060 basé sur le GT200. La configuration logicielle est Ubuntu 8.04, CUDA 2.0 et CUDA 2.3 avec les pilotes graphiques NVIDIA 177.13 et 195.17.

Pour les mesures de consommation, nous avons utilisé une pince ampèremétrique CA60 sur les câbles d'alimentation 12 Vdu boîtier Tesla D870, reliée à un oscilloscope Tektronix TDS 3032 mesurant également la tension d'alimentation. Cette méthodologie permet une mesure de la consommation sur des échantillons de l'ordre de 20 µs. Cela

TAB. 3 – Répartitions des domaines d'horloge de l'architecture Tesla. L'exemple choisi est la GeForce 9800 GX2.

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

s'avère suffisant pour des mesures au niveau tâche, mais non au niveau instruction [3].

Pour réaliser les tests sur l'ordonnancement des instructions, nous avons utilisé des instructions de barrière de synchronisation (`BAR`) pour isoler le code testé, et des instructions de lecture du registre d'horloge (`clock`) pour mesurer au cycle près l'exécution de ces instructions.

```
BAR
MOV R7, clock
MAD R3, R1, R2, R0
MOV R3, clock
BAR
```

FIG. 1 – Exemple de code assembleur utilisé pour les mesures de temps.

5 Organisation générale

La partie calcul est composée d'une hiérarchie de cœurs [12, 10]. Au niveau le plus élevé, le GPU contient jusqu'à 10 TPC (texture / processor cluster) connectés aux autres composants par réseau d'interconnexion en croix. Chaque TPC contient une unité d'accès à la mémoire, disposant de son propre cache de premier niveau et de ses unités de calcul d'adresse et filtrage de texture, et plusieurs cœurs de calcul ou SM (streaming multiprocessors) ou encore multiprocesseurs. Chacun de ces SM est un processeur autonome SIMD multi-thread.

Domaines d'horloge Le GPU est divisé entre plusieurs domaines d'horloges, présentés table 3, en fonctions des compromis entre latence, débit, surface et consommation souhaités pour chaque composant matériel. Les unités de calcul (SP) fonctionnent à la fréquence la plus haute. Le reste des SM est animé par une horloge de fréquence divisée par deux. Les unités non dédiées au calcul généraliste tel que le réseau d'interconnexion et l'ordonnanceur global fonctionnent à une fréquence indépendante, généralement inférieure. Enfin, les contrôleurs mémoires suivent l'horloge de la mémoire externe, en général de type GDDR3.

Souplesse et passage à l'échelle Les différentes micro-architectures de l'architecture Tesla ont suivi des évolutions progressives. La stratégie généralement suivie consiste à introduire une nouvelle révision majeure dans le segment haut-de-gamme, pour ensuite réutiliser sa micro-architecture pour les segments milieu et entrée de gamme en réduisant progressivement le nombre de TPC et de partitions mémoire. Des révisions

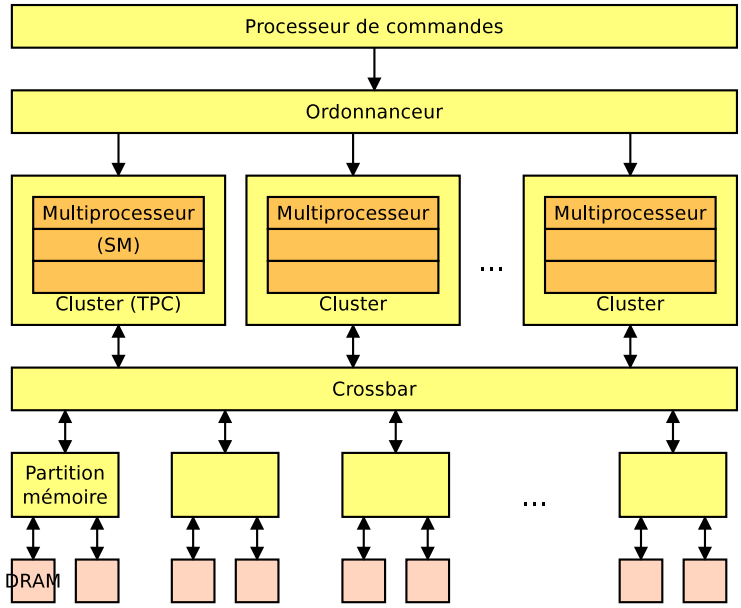


FIG. 2 – Vue générale de l'architecture Tesla.

mineures peuvent êtres introduites à cette occasion, mais également des pertes de fonctionnalités (double-précision sur les dérivés du GT200). La table 4 liste l'ensemble des GPUs dérivés de l'architecture Tesla. Au delà des différences dues au choix du GPU, il est possible de produire d'autres modèles de cartes graphiques en désactivant certains TPC et partitions mémoire et en variant les différentes fréquences pour améliorer les rendements de production ou réduire les autres coûts de fabrication de la carte graphique (circuit imprimé, mémoire).

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

6 Interface

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

6.1 Processeur de commandes

Pour fonctionner de manière asynchrone par rapport aux CPU, le GPU est contrôlé au travers d'une file de commandes selon un schéma producteur-consommateur classique. La file est typiquement placée en mémoire centrale. Le pilote graphique ou GPGPU se charge d'ajouter des commandes dans la file, tandis que le processeur de commandes du GPU retire les commandes à l'autre extrémité de la file.

TAB. 4 – Réalisations de l'architecture Tesla.

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

Les commandes peuvent être des commandes de configuration, des copies mémoires, des lancements de kernels. . .Pour permettre à plusieurs contextes d'exécution de coexister, et pour exploiter du parallélisme en recouvrant copies mémoire et exécution sur le GPU, plusieurs files de commandes virtuelles peuvent être utilisées. Le basculement du GPU d'un contexte d'exécution à l'autre se fait par exécution de microcode[7].

7 Répartir le travail

Un ordonnanceur est une unité du GPU chargée de répartir les blocs (groupes de threads garantis de s'exécuter sur le même multiprocesseur) à exécuter sur les multiprocesseurs. Une telle unité est décrite dans [16]. Avant d'envoyer un signal de début d'exécution aux multiprocesseurs, il procède à l'initialisation des registres et des mémoires partagées.

En particulier, il doit initialiser :
- les arguments du kernel,
- les coordonnées du bloc,
- la taille des blocs et de la grille,
- les coordonnéees du thread à l'intérieur du bloc.

Les arguments et les coordonnées du bloc sont communs à tous les threads d'un bloc. Ils peuvent donc être placées en mémoire partagée. Ainsi, les 16 premiers octets de la mémoire partagée sont initialisés par 8 entiers 16 bits contenant les informations de dimension. Les arguments sont stockés aux adresses immédiatement supérieures. Il est à noter que certaines de ces données sont constantes, et en tant que telles, pourraient être passées par la mémoire de constantes. Le choix qui a été fait s'explique probablement par le coût d'une initialisation de la mémoire de constantes.

L'identifiant de chaque thread est quant-à-lui écrit dans le registre architectural R0 sous la forme d'un champ de bits, suivant le format indiqué figure 3.

Des tests effectués en lançant des blocs de durées d'exécution variables nous montrent que la politique d'ordonnancement des blocs est de type *round-robin*, et que l'ordonnanceur global attend la terminaison de tous les blocs en cours d'exécution sur le GPU

[7] `http://nouveau.freedesktop.org/wiki/CtxInit`

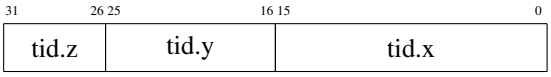
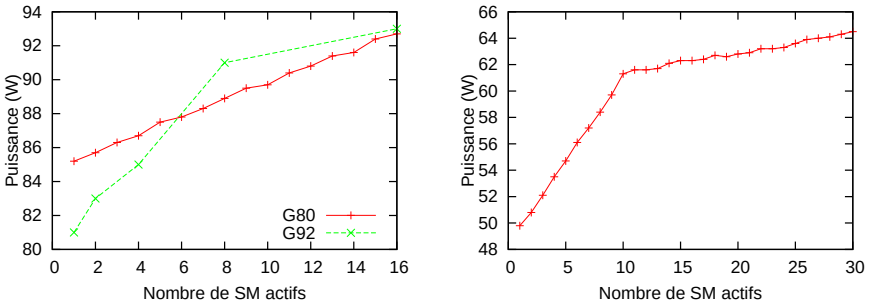


FIG. 3 – Contenu du registre R0 au lancement d'un kernel.

avant d'ordonnancer le groupe de blocs suivant. Il n'y a pas de fonctionnement de type pipeline. Cependant, lorsqu'un TPC n'a plus de travail à exécuter, son compteur d'horloge cesse de s'incrémenter, ce qui laisse supposer la présence de *clock gating* à gros grain pour minimiser la consommation d'énergie.

Nous avons également mesuré la consommation des GPU considérés en faisant varier le nombre de multiprocesseurs actifs de 1 jusqu'au maximum (16 pour le G80 et le G92 et 30 pour le GT200). Les résultats sont présentés sur la figure 4. On observe que la puissance consommée augmente linéairement avec le nombre de multiprocesseurs jusqu'à un point correspondant respectivement à tous, la moitié et le tiers des multiprocesseurs pour le G80, le G92 et le GT200. Cela implique que la stratégie d'ordonnancement des blocs à exécuter est différente entre le G80 et les G92 et GT200. Celle des GPU les plus modernes est de réaliser l'ordonnancement en largeur, en distribuant les blocs sur des TPC différents, favorisant la bande passante disponible et l'équilibrage du réseau d'interconnexion. À l'inverse, celle du G80 est un ordonnancement en profondeur.



(a) G80 - G92

(b) GT200

FIG. 4 – Variation de la puissance consommée en fonction du nombre de blocs ordonnancés.

8 Front-end

La capacité à masquer les latences des diverses opérations tout en conservant un débit soutenu élevé repose sur l'exploitation par les multiprocesseurs de multithreading simultané (SMT) massif. Ainsi, le G80 et le GT200 maintiennent respectivement 24 *warps* (fil d'exécution d'instructions SIMD) et 32 warps en vol simultanément.

À la différence des supercalculateurs vectoriels des années 1990 où une même instruction est exécutée sur des vecteurs significativement plus longs que la largeur des unités de calcul, cette approche permet de désynchroniser l'exécution des sous-vecteurs (warps), offrant un fonctionnement de type MIMD au sein d'un cœur de calcul.

Contrairement à l'approche suivie par les architectures graphiques d'AMD [1], Tesla n'utilise pas de multithreading à basculement sur évènement. Une fois encore,

il s'agit probablement d'une volonté de conserver un modèle simple au niveau architectural.

Des processeurs superscalaires conventionnels gèrent le SMT, mais se limitent à deux voire quatre threads. Au delà, le gain de performance additionnel justifie difficilement le coût en matériel et la complexité de la conception, et du multithreading à basculement sur évènement est préféré. Pour atteindre efficacement un tel niveau de parallélisme, l'architecture des multiprocesseurs est conçue dés l'origine pour le SMT. En contrepartie, cette architecture est inefficace pour les charges de type monothread. Ainsi, lorsqu'un seul warp est en cours d'exécution, le multiprocesseur n'est capable d'exécuter qu'une instruction tous les 8 cycles SP, contre un CPI de 2 lorsque le multiprocesseur est saturé. De plus, Tesla se limite à l'exécution de code SPMD, qui offre plus de régularité et de localité que l'exécution simultanée de threads arbitraires. Plutôt que d'ajouter de la complexité au système, le SMT apporte ici à l'inverse de la régularité et du parallélisme permettant de simplifier l'architecture.

La figure 5 présente une vue qualitative du pipeline d'exécution d'un SM. Les étages présentés sont des étapes logiques ; chacun peut nécessiter en réalité plusieurs cycles.

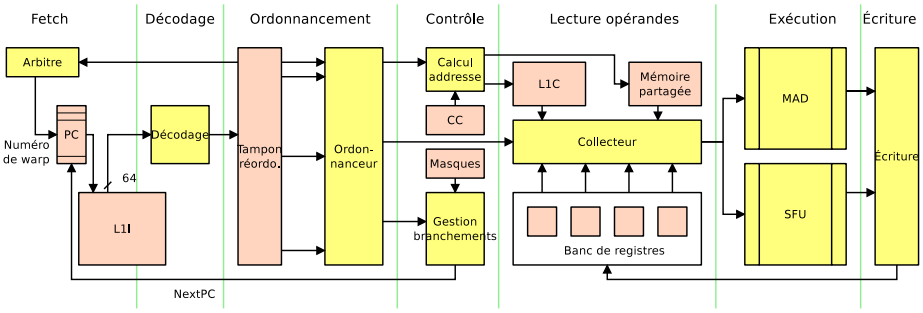


FIG. 5 – Vue d'ensemble du pipeline d'exécution d'un multiprocesseur.

8.1 Fetch

Chaque warp dispose de son propre pointeur d'instruction, et d'un emplacement réservé dans la file de réordonnancement des instructions. À chaque cycle, lorsqu'un emplacement est libre dans la file de réordonnancement, une instruction est lue depuis le cache d'instructions de premier niveau à l'adresse désignée par le pointeur d'instruction du warp correspondant. Des tests suggèrent que les instructions sont chargées par blocs de 64 octets, ce qui représenterait 8 instructions [20]. Nous n'avons pas pu confirmer ni infirmer de manière définitive ces observations.

La taille du cache L1 d'instructions est estimée à 2 Ko et son associativité à 4 voies [20]. L'utilisation d'un cache aussi réduit est possible grâce au modèle SPMD et à la faible granularité des communications entre warps assurant une synchronisation relative et donc une localité temporelle dans la mémoire d'instructions. En effet, une même instruction sera exécutée sur plusieurs warps différents dans un intervalle de temps court. Le nombre de warps concurrents a même une influence positive sur la pression sur le cache d'instructions : à débit d'exécution équivalent, plus de warps diminueront le turnaround [?] du cache d'instructions et amélioreront la localité. Cet

effet peut être encouragé par une politique d'ordonnancement favorisant la localité dans le cache d'instruction [14].

8.2 Ordonnancement

Comme dans un processeur à exécution dans le désordre, un ordonnanceur se charge de sélectionner une instruction dans la file de réordonnancement pour exécution. Cette unité détermine l'ensemble des instructions pouvant être exécutées immédiatement sans violer de contraintes (dépendances). Parmi ces instructions, un calcul de priorité en fonction de divers critères tels que l'âge et le type de l'instruction détermine celle qui sera sélectionnée [12, 14].

Comparé à une exécution scalaire dans le désordre, les conditions sont plus simples : chaque warp bénéficiant de ses propres registres privés, aucune relation de dépendance par rapport aux registres ne peut exister entre deux instructions provenant de warps différents. De plus, lorsque l'on considère un warp donné, ses instructions sont toujours démarrées dans l'ordre. Les mécanismes de suivi des dépendances mis en place se rapprochent de fait plus de ceux des processeurs pipelinés à exécution dans l'ordre.

Les dépendances entre instructions par rapport aux registres sont tenues à jour au moyen d'un *scoreboard*. Pour accepter un grand nombre de warps et de registres, tout en bénéficiant de l'indépendance entre warps, le scoreboard est indexé par les numéros de warps plutôt que par les numéros de registres comme dans un circuit conventionnel [5].

Les instructions de calcul peuvent être prédiquées par un registre de drapeaux et un code de condition. La prédication est utilisée sur des processeurs haute-performances comme mécanisme spéculatif pour éliminer des dépendances de contrôle. L'instruction est exécutée systématiquement avant-même que la valeur du prédicat soit connue, mais son résultat n'est pas écrit dans les registres architecturaux si le prédicat se révèle faux par la suite.

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

Pour déterminer à partir de quel étage du pipeline la valeur du prédicat est connue, nous avons effectué des mesures de performance et de consommation en comparant le débit et l'énergie consommée par une instruction prédiquée négativement par rapport à une instruction non prédiquée. Les résultats sont présentés table 5. D'une part, on observe que le débit des instructions prédiquées à faux est supérieur au débit des unités d'exécution (CPI de 4 pour MAD). Cela montre que ces instructions sont éliminées avant leur exécution. D'autre part, le gain en énergie qui en résulte est significatif.

Également, nous mesurons une latence de 15 cycles SM entre l'exécution d'une instruction `set` écrivant dans un registre de drapeaux et une instruction prédiquée par une condition sur ces drapeaux, contre 4 cycles sans cette dépendance. Cette latence correspond au temps d'exécution de la première instruction (10 cycles), et du dispatch de la deuxième et lecture des drapeaux (5 cycles). En revanche, en l'absence de dépendance sur les registres de drapeaux, le surcoût de la prédication est nul : une instruction prédiquée par une condition positive nécessitera le même temps d'exécution qu'une instruction non prédiquée.

De même, un saut conditionnel offre toujours une latence de 16 cycles que le branchement soit pris ou non. Cela montre qu'aucune prédiction de branchement statique ou dynamique n'est faite. Cette décision est certainement basée sur l'hypothèse qu'il y a toujours au moins une instruction susceptible d'être exécutée dans la file d'attente

TAB. 5 – Puissance mesurée P, nombre de cycles par instruction et énergie requise par warp pour une opération MAD prédiquée sur le G80, le G92 et le GT200, suivant la valeur prise par le prédicat.

GPU	Pred.	P (W)	CPI	E (nJ/warp)
G80	Vrai	107	4.75	8.57
	Faux	90	2.38	2.43
G92	Vrai	100	4.29	5.06
	Faux	93	2.39	2.14
GT200	Vrai	91	4.3	5.31
	Faux	75	2.36	1.75

grâce au SMT massif, indépendamment des dépendances de données et de contrôle présentes dans chaque warp. Dans ces conditions, mettre en œuvre des mécanismes spéculatifs pour tirer partie du parallélisme d'instruction n'est pas judicieux, car cela affecterait négativement le débit en cas d'erreur de prédiction sans apporter d'avantage notable en cas de prédiction juste.

Lorsqu'un aléa non prévu par l'ordonnanceur tel qu'un conflit de banc en mémoire partagée ou cache de constantes intervient, l'instruction fautive continue à s'exécuter partiellement. Seules les voies SIMD ayant rencontré un conflit sont masquées. Ainsi, le déroulement du pipeline n'est pas interrompu. L'instruction fautive est ensuite remise en attente dans la file d'instructions avec un masque mis à jour pour refléter le travail restant à exécuter. La latence mesurée d'un accès en mémoire de constantes est en effet de $10n_c$ cyles SM, et celle d'un accès en mémoire partagée est $8n_c$ cycles, pour n_c le nombre de conflits de banc. En revanche, le débit effectif des deux mémoire reste de $16/n_c$ mots par cycle SM. Ces évènements peuvent être détectés et comptés par le compteur de performance *warp serialization* du profiler NVIDIA.

8.3 Banc de registres

Le G80 et le GT200 disposent respectivement de 256 et 512 registres architecturaux vectoriels de 32×32 bits par SM, ce qui représente 512 Ko et 1,9 Mo de registres sur l'ensemble de la puce. Pour atteindre de telles capacités en conservant un coût en surface et en consommation raisonnable, le nombre de ports de chaque banc de registres se doit d'être réduit. Pour permettre tout de même les accès simultanés à plusieurs opérandes, un banc de registres est décomposé en sous-bancs. Un arbitre se charge alors de gérer les accès concurrents, qui peuvent dés-lors générer des conflits.

Papadopoulou et al. suggèrent une organisation en 64 bancs de 128 registres de 32-bit sur le GT200 [20]. Les registres architecturaux étant vectoriels, il est cependant peu intéressant d'utiliser une largeur de port aussi réduite. De fait, les photographies du *die* du GT200 présentent seulement 16 bancs par SM, probablement de taille 128×256 bits chacun [24]. Ils occupent environ 0,78 mm^2 dans le processus de fabrication 65 nm généraliste de TSMC. Cela suggère une optimisation agressive visant à limiter la surface. En effet, des estimations basées sur le modèle de CACTI [22] nous indiquent une efficacité en surface (fraction de la surface utilisée par les cellules SRAM) de l'ordre de 40 %, ce qui est par exemple supérieur d'un facteur 20 à celle du banc de registres

de l'Intel Itanium-2 [7].

La stratégie suivie pour minimiser les conflits de bancs est de profiter de la régularité de l'ordonnancement des warps (proche du round-robin), au dépends de la latence. Ainsi, les registres d'un warp donné sont alloués prioritairement dans le même banc, de façon à ce que des warps différents soient à l'inverse associés à des bancs différents [11]. Les opérandes d'une instruction donnée devront alors impérativement être lus séquentiellement, ce que nous vérifions par des mesure de latence. En revanche, ces lectures peuvent être pipelinées avec les lectures d'opérandes des instructions suivantes, appartenant à d'autres warps dont les registres ont été alloués dans d'autres bancs. Ainsi, les conflits de bancs restent prévisibles et n'affectent pas l'ordonnancement ni le choix des numéros de registres.

Il semble également que chaque banc dispose d'un port de lecture et d'un port d'écriture séparés. Par exemple, la figure 6 présente un ordonnancement possible d'une série d'instructions MUL à 2 opérandes d'entrées exécutés par plusieurs warps. Encore une fois, ces décisions facilitent la tâche du compilateur en présentant un coût d'accès aux registres homogène.

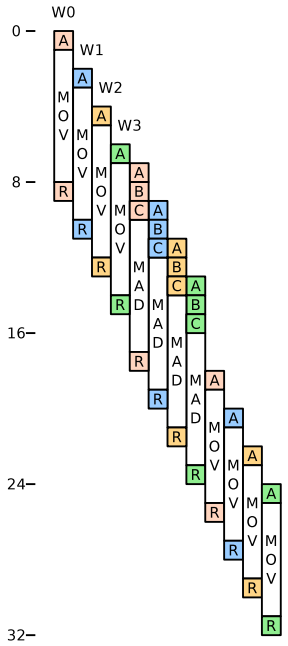


FIG. 6 – Répartition des lectures des opérandes dans les bancs lors de l'exécution d'instructions MAD.

9 Unités d'exécution

Huit unités dites MAD sont chargées des calculs arithmétiques généralistes. Il s'agit d'un pipeline construit autour d'un multiplieur 24×24 suivi d'un additionneur puis d'une unité d'arrondi [21]. Il est capable d'effectuer avec une latence et un débit constants aussi bien les instructions entières (arithmétique, logique, décalages) que

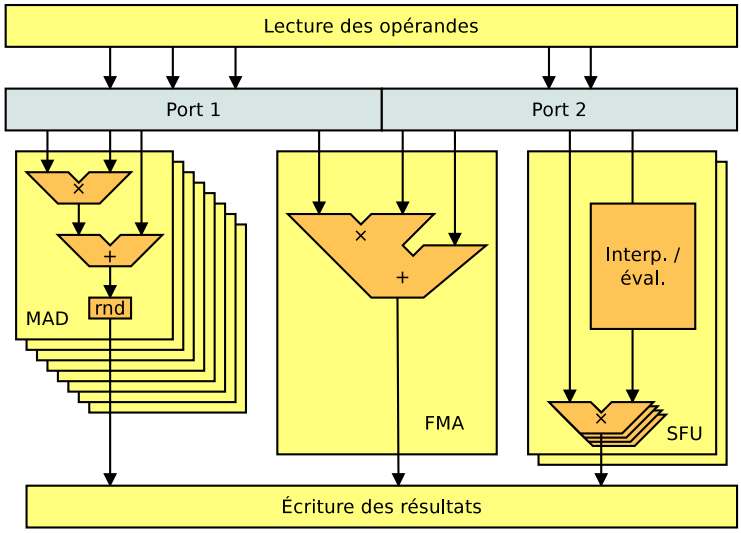


FIG. 7 – Unités d'exécution d'un multiprocesseur.

virgule-flottante (multiplication-addition, min, max, conversions) en simple précision.

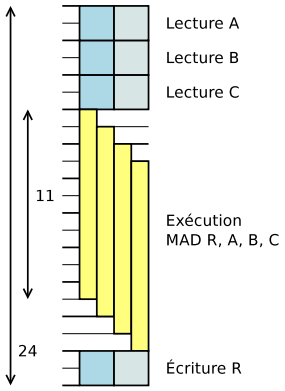


FIG. 8 – Organisation du pipeline comprenant la lecture des opérandes, l'éxécution et l'écriture des résultats durant l'exécution d'une instruction MAD. Les latences sont indiquées en cycles SP.

Il est à noter que le résultat de la multiplication virgule-flottante simple précision est arrondi vers zero en un nombre flottant simple précision, ce qui conduit l'instruction MAD à avoir un comportement différent de celui d'un FMA, mais aussi d'une multiplication suivie d'une addition lorsque le mode d'arrondi n'est pas vers zéro. Cela s'explique par le choix de n'inclure qu'une seule unité d'arrondi dans le pipeline. Ainsi, les instructions de multiplication et d'addition individuelles peuvent bénéficier de cette unité d'arrondi, mais pas le MAD combiné.

La latence de cette unité peut être estimée à partir de la mesure du temps d'exécution de deux instructions dépendantes. Le délai de lecture et d'écriture des registres, ainsi

TAB. 6 – Latences estimées et débit des unités du G80 et GT200, en cycles SP

Instruction	Latence totale		Latence exécution (estimée)		Débit
	G80	GT200	G80	GT200	(op/cycle)
MAD/MUL1	18 – 22	20–24	11	13	8
SF	40	42	21	23	2
MUL2	26 – 28	28	19	21	8
FMA	-	58–66	-	19	1

que le temps d'amorçage du pipeline d'exécution doit cependant être pris en compte pour ce calcul. La figure 8 illustre une telle estimation pour une instruction MAD. Cette estimation présuppose l'absence de réseau de bypass. Les résultats sont récapitulés table 6.

Une seconde unité, nommé SFU, réalise l'évaluation des fonctions élémentaires, l'interpolation d'attributs (coordonnées de texture, couleurs...) en entrée du *pixel shader*, ainsi que des multiplications. Cette unité contient un circuit d'interpolation suivi de multiplieurs virgule-flottante [19]. Les unités d'interpolation peuvent exécuter une instruction d'évaluation de fonction sur un warp tous les 16 cycles, ou une instruction de multiplication tous les 4 cycles.

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

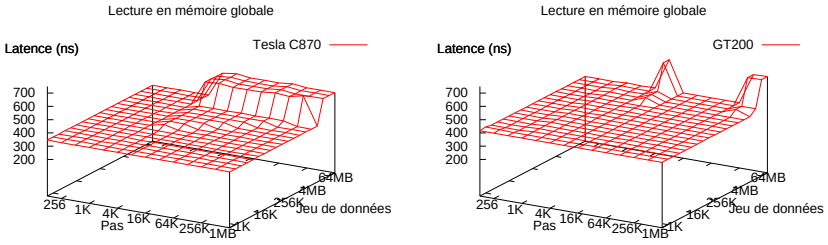
L'unité double précision [18] peut démarrer un FMA sur un warp tous les 32 cycles. La latence totale d'une instruction FMA est de 58 à 66 cycles suivant le nombre d'opérandes uniques. La latence de l'unité FMA elle-même est donc estimée à 19 cycles.

10 Hiérarchies mémoires

L'architecture mémoire reflète l'organisation logique décrite dans la section 3. En particulier, on y retrouve des chemins d'accès distincts et autant de hiérarchies mémoires pour la mémoire de textures, la mémoire de constantes, la mémoire d'instructions et la mémoire partagée. Cependant, pour des raisons de réutilisation de matériel, les espaces mémoires globaux et locaux partagent la majeure partie du chemin d'accès avec la mémoire de textures.

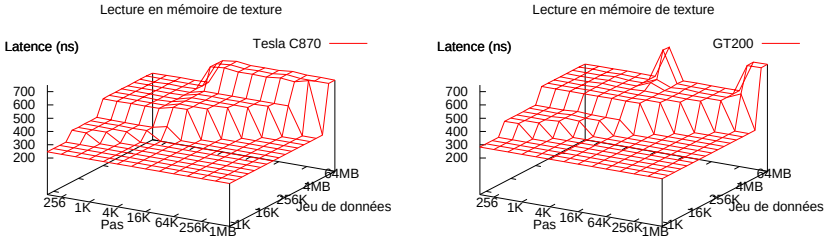
Considérons les latences de chacune de ces mémoires, mesurées par une séquence d'accès à pas constant traversant de manière répétitive une zone mémoire de taille limitée, en conservant le temps pris par le dernier accès effectué. Les figures 9, 10 et 11 retracent les résultats obtenus.

Alors que la latence d'un accès en mémoire centrale d'un processeur moderne est de l'ordre de 40 ns, les latences observées sur les GPU Tesla sont un ordre de grandeur supérieures et varient ici entre 300 et 800 ns. Ainsi, sur le Tesla C870 (figure 9 a), nous observons trois plateaux situés respectivement à 345 ns, 372 ns et 530 ns en fonction de la localité des accès. Les lectures en mémoire de texture ne sont pas significativement plus faible. Nous tenterons d'expliquer dans cette section les mécanismes qui peuvent conduire à de tels résultats.



(a) Tesla C870 (b) GT200

FIG. 9 – Latence d'une lecture en mémoire globale.



(a) Tesla C870 (b) GT200

FIG. 10 – Latence d'une lecture en mémoire de texture.

10.1 Mémoires internes

Chaque SM dispose :

- d'un cache de constantes de premier niveau,
- d'un cache d'instruction de premier niveau,
- d'une mémoire partagée.

La table 7 récapitule les résultats connus par la documentation officielle ou des tests de la littérature [23, 20]. Nous avons pu vérifier ces résultats pour le cache de constantes.

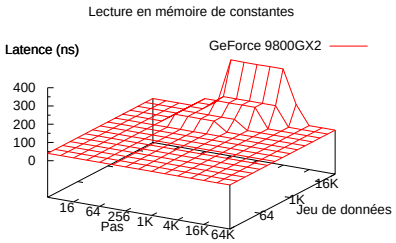


FIG. 11 – Latence d'une lecture en mémoire de constantes.

TAB. 7 – Caractéristiques des mémoires locales aux SM

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

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

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

En revanche, la mémoire partagée doit permettre de stocker temporairement et accéder rapidement à des données vectorielles. C'est également au travers de cette mémoire que sont effectuées les opérations d'échanges entre voies SIMD (*swizzle*), plutôt qu'au moyen d'instructions dédiées comme dans d'autres architectures SIMD [8].

Pour être à même de nourrir les unités de calcul sans devenir un goulot d'étranglement, la mémoire partagée doit offrir l'équivalent de 16 ports de 32 bits en lecture-écriture. Une telle mémoire n'étant pas raisonnablement réalisable, la mémoire partagée est composée de 16 bancs indépendants à l'instar des bancs de registres. Chacun de ces bancs ne possède qu'un unique port de lecture-écriture. Un arbitre se charge de simuler la présence de 16 ports. Comme celui du banc de registres, il est capable d'identifier les requêtes concurrentes à une cellule identique et de les fusionner, dans la limite d'une fusion par cycle. Les conflits sont traités de la manière décrite section 8.2.

En dehors du risque de conflits de bancs, on peut vérifier que les accès avec indirection ne sont pas plus coûteux que les accès avec adressage absolu, grâce à l'utilisation de registres d'adresse dédiés dont la valeur est disponible plus tôt dans le pipeline que celle des registres généraux.

10.2 Mémoires niveau cluster

Le TPC regroupe le cache de constantes de deuxième niveau et les unités d'accès mémoire, incluant les caches de texture de premier niveau et les unités de filtrage de texture.

La mémoire de texture sert typiquement à effectuer des lectures à un emplacement calculé à partir de coordonnées bidimensionnelles, suivi d'un filtrage éventuel. Ce type d'accès offre une forte localité spaciale. Un cache de texture agit comme un filtre pour fusionner les lectures similaires et adjacentes. Les accès redondants sont éliminés, et la granularité des accès augmentée, ce qui permet une meilleure exploitation de la bande passante mémoire.

Toutes les lectures de texture sont des opérations de type *gather* sur 16 données distinctes voire plus, et l'exécution des instructions SIMD dépendant du résultat ne peut se poursuivre tant que l'ensemble des résultats ne sont pas disponibles. Ce fait

diminue drastiquement le bénéfice qu'un cache peut avoir sur la latence des accès. Pour obtenir un gain de latence, il faut que toutes les lectures composant l'opération gather soient des succès dans le cache, évènement dont la probabilité est faible. De fait, le cache de texture est conçu en priorité comme un moyen d'augmenter le débit effectif plutôt que de diminuer la latence. Ce fait est illustré de façon flagrant par la figure 10 montrant qu'un succès dans le cache L1 de texture conserve toujours une latence de 280 ns sur le GT200, soit 7 fois le temps d'un accès à la DRAM sur un système monoprocesseur actuel.

Dans l'architecture Tesla, le cache de texture intégré à chaque TPC est partagée statiquement entre chaque SM, et se comporte donc comme deux ou trois caches indépendants. Ses index et ses *tags* référencent des adresses virtuelles. L'analyse de Volkov et Demmel montre que seuls 5 ko de cache par SM sont accessible lorsque la mémoire de texture est addressée en mode linéaire. Nos propres tests confirment ces résultats et nous conjecturons que l'adressage linéaire correspond à un mode dégradé du cache ne permettant pas l'accès à l'intégralité des 8 ko documentés, contrairement aux modes d'accès natifs « pavés ».

La gestion de la cohérence mémoire se fait de manière manuelle, par invalidation des caches. La taille réduite des caches et la faible localité temporelle rend cette approche raisonnable. Dans la pratique, la mémoire de texture est considérée comme accessible en lecture seule pendant tout le temps d'exécution d'un calcul sur le GPU.

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

L'unité d'accès mémoire est connectée aux bancs de registres de chaque SM. Contrairement à l'approche usuelle consistant à avoir un tampon de données arrivant depuis la mémoire dont le contenu est en suite copié vers les registres, l'unité mémoire écrit ici directement dans le registre cible, évitant la nécessité d'une mémoire dédiée [13]. Ce mécanisme permet de maintenir un grand nombre de transactions mémoires en vol.

10.3 Traduction d'adresse

L'accès à la mémoire de texture, et par extension à la mémoire globale, nécessite au total trois étapes de traduction d'adresse successives [15].

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

De manière à permettre à plusieurs applications de stocker des données privées en mémoire graphique tout en garantissant la fiabilité et la sécurité, un système de mémoire virtuelle doit être utilisé. La traduction des adresses virtuelles en adresses physiques est opérée au niveau du réseau d'interconnexion par des TLB.

L'analyse des latences mémoires révèle la présence de deux niveaux de TLB, suivi d'une autre structure [20], qui pourrait être le cache de constantes de troisième niveau. Nos propres tests révèle que, si les tailles et les valeurs d'associativité obtenues sont identiques, la taille des pages varie quant-à-elle d'une architecture à une autre, et même d'une version des pilotes à une autre sur le même GPU (table 8).

Enfin, les transactions mémoires doivent être routées vers une des partitions mémoires du GPU. Cette décision revient à calculer une adresse *brute* à partir de l'adresse physique. Il est à noter que le nombre de partition mémoires n'est pas nécessairement un

TAB. 8 – Taille des pages mesurée suivant le GPU et le pilote graphique

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

multiple de deux, et peut même varier pour un même modèle de GPU au moment de l'incorporation sur la carte graphique. Dans la pratique, le mode de traduction utilisé consiste à répartir des blocs de 256 octets en round-robin sur les partitions disponibles, ce qui revient à effectuer une division euclidienne de l'adresse par le nombre de partitions, même si le système réellement employé en matériel est potentiellement plus souple.

10.4 Réseau d'interconnexion et back-end

Le réseau d'interconnexion qui relie les TPC aux partitions mémoire est manifestement un circuit *ad-hoc* de type *crossbar* conçu pour chaque GPU plutôt qu'un circuit générique extensible de type *ring bus*.

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

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

10.5 Contrôleur mémoire

La lecture de brevets décrivant des contrôleurs mémoires de GPU apporte quelques éléments de réponse pour expliquer la latence mémoire observée [6, 9]. Au-delà du réordonnancement des accès pour minimiser les chargement/déchargement de page et les inversion de bus lecture/écriture tel qu'il est opéré par des contrôleurs conventionnels, ce type de contrôleur peut décider de volontairement retarder un accès en attente, dans l'espoir qu'une autre transaction plus avantageuse à traiter arrive. Ainsi, le risque de *ping-pong* entre deux paged DRAM ou des alternances lecture/écriture est évité.

D'autre part, il peut être avantageux de maintenir une latence d'apparence uniforme du point de vue des unités mémoire, même si cela conduit à retarder les réponses aux requêtes servies rapidement pour s'aligner sur le pire cas. Ainsi, une synchronisation relative entre les warps est maintenue, ce qui est avantageux pour la localité dans les caches et autres structures exploitant la localité spatiale. Cette remise dans l'ordre des accès pour synchronisation peut être assurée par le contrôleur mémoire, le réseau d'interconnexion ou en fin de chaîne par l'ordonnanceur d'instruction lui-même [14].

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

11 Quel modèle de programmation ?

Comme il a été observé au travers de multiples exemples, la conception de l'architecture Tesla semble guidée par un souci de fournir un modèle de programmation simple et stable directement au niveau architectural. À l'inverse, le modèle de programmation exposé au développeur par CUDA et OpenCL est relativement bas-niveau. Ainsi, le fossé sémantique à franchir pour compiler des programmes de calcul généraliste vers cette architecture est relativement faible. Il est donc possible d'obtenir des outils efficaces pour un coût de développement réduit. Le développeur dispose également d'opportunités pour optimiser ses programmes au travers de règles simples (*coalescing*, conflits de banc...), sans nécessité de connaître le détail de l'architecture.

En contrepartie, la conception de l'architecture est plus délicate, et des mécanismes matériels doivent être mis en place pour maintenir l'abstraction : gestion transparente des branchements, détection dynamique des accès mémoire réguliers (coalescing), arbitrage des conflits...ont un coût en ressources matérielles.

Cette opposition est analogue à l'antagonisme classique entre processeurs superscalaires à exécution dans le désordre reposant sur des optimisations dynamiques effectuées en matériel, et architectures VLIW se basant sur des optimisations statiques opérées par le compilateur. Ce parallèle montre qu'il est difficile de trancher sur la pertinence de chaque approche du point de vue de l'efficacité en ce qui concerne les GPU. En effet, si le choix d'une architecture bas-niveau permet de simplifier la micro-architecture, l'alternative offre de son côté des opportunités d'optimisations dynamiques effectuées en matériel qu'il ne serait pas possible ou difficile d'opérer de manière statique à la compilation.

Les objectifs de conception du jeu d'instruction de Tesla ne semblent pas différer significativement de ceux des jeux d'instructions CISC des années 1970 et 1980 : combler le fossé sémantique entre assembleur et langages haut-niveau. À partir de là, les alternatives possibles seraient de repartir d'une architecture de type RISC plus proche du matériel, ou bien de développer la micro-architecture pour s'abstraire du jeu d'instruction à l'instar des processeurs x86.

Persister dans le choix d'une architecture haut-niveau conduirait certainement à adopter de plus en plus d'optimisations dynamiques spécifiques aux architectures parallèles, telles qu'une gestion des branchements sans annotations [2] ou une détection des opérations scalaires et à pas constant [4].

Références

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

[2] Caroline Collange, Marc Daumas, David Defour, and David Parello. Étude com-parée et simulation d'algorithmes de branchements pour le GPGPU. In *SYMPo-sium en Architectures nouvelles de machines (SYMPA)*, 2009.

[3] Caroline 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.

[4] Caroline 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)*, 2009.

[5] Brett W. Coon, Peter C. Mills, Stuart F. Oberman, and Ming Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators, October 2008. US7434032.

[6] Roger E. Eckert. Page stream sorter for DRAM systems, May 2008. US7376803.

[7] E.S. Fetzer, M. Gibson, A. Klein, N. Calick, Chengyu Zhu, E. Busta, and B. Mohammad. A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor. *Solid-State Circuits, IEEE Journal of*, 37(11) :1433–1440, Nov 2002.

[8] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *International Symposium on High-Performance Computer Architecture*, pages 258–262, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[9] Brian D. Hutsell and James M. Van Dyke. Efficiency based arbiter, October 2009. US7603503.

[10] David Kanter. NVIDIA's GT200 : Inside a parallel processor. Technical report, Real World Technologies, 2008.

[11] 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.

[12] 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.

[13] John Erik Lindholm, John R. Nickolls, Simon S. Moy, and Brett W. Coon. Register based queuing for texture requests, November 2008. US7456835.

[14] Peter C. Mills, John Erik Lindholm, Brett W. Coon, Gary M. Tarolli, and John Matthew Burgess. Scheduling instructions from multi-thread instruction buffer based on phase boundary qualifying rule for phases of math and data access operations with better caching, April 2008. US 7366878.

[15] John S. Montrym, David B. Glasco, and Steven E. Molnar. Apparatus, system, and method for using page table entries in a graphics system to provide storage format information for address translation, June 2009. US7545382.

[16] Bryon S. Nordquist, John R. Nickolls, and Luis I. Bacayo. Parallel data processing systems and methods using cooperative thread arrays and simd instruction issue, September 2009. US 7584342.

[17] NVIDIA. *PTX : Parallel Thread Execution, ISA Version 1.4*, 2008.

[18] Stuart Oberman, Ming Y. Siu, and David C. Tannenbaum. Fused multiply-add functional unit, June 2009. US Application 20090150654.

[19] 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.

[20] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the GT200 GPU. Technical report, Computer Group, ECE, University of Toronto, 2009.

[21] Ming Y. Siu and Stuart F. Oberman. Multipurpose functional unit with multiply-add and format conversion pipeline, September 2008. US7428566.

[22] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Technical report, HP Labs, 2008.

[23] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[24] Scott Wasson. Nvidia's GeForce GTX 280 graphics processor. Technical report, The Tech Report, 2008.