RenPar'20 / **SympA'14** / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT

Caroline Collange

ENS de Lyon, Université de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL),
École Normale Supérieure de Lyon,
46 allée d'Italie,
69364 Lyon Cedex 07, France

Résumé
Les architectures parallèles qui suivent le modèle SIMT telles que les GPU tirent parti de la régularité des applications en exécutant plusieurs threads concurrents sur des unités SIMD de manière synchronisée. Cela nécessite de séparer les threads lorsqu'il prennent des chemins différents dans le graphe de contrôle de flot, et d'être à même de les re-synchroniser dès que possible de manière à maximiser l'occupation des unités SIMD. Nous proposons dans cet article une technique pour traiter la divergence de contrôle en SIMT qui opère en espace constant et gère les sauts indirects et la récursivité. Nous décrivons une réalisation possible qui s'appuie sur le matériel existant de l'unité de gestion de la divergence mémoire, assurant un coût matériel très réduit. En termes de performance, cette solution est au moins aussi efficace que les techniques existantes.

Mots-clés : Reconvergence de flot de contrôle, SIMD, SIMT, GPU

1. Introduction

Les processeurs graphiques (GPU) se sont progressivement imposés comme des alternatives crédibles aux processeurs généralistes haute performance pour de nombreuses applications faisant apparaître du parallélisme de données. Ce champ d'applications dépasse de loin le domaine du rendu graphique [11]. Ainsi, la première place au classement des supercalculateurs Top500 d'octobre 2010 revenait à une machine à base de GPU.
L'ensemble des GPU actuels opèrent selon le modèle d'exécution SIMT (*single instruction, multiple threads*). Du point de vue du programmeur et du compilateur, ce modèle est similaire au SPMD (*single program, multiple data*). Le programmeur écrit un unique programme ou *noyau* dont un grand nombre d'instances (ou *threads*) seront exécutées en parallèle.
Lors de l'exécution sur GPU, des mécanismes matériels transparents regroupent des threads en convois nommés *warps*, pour exécuter leurs instructions sur des unités SIMD. À la différence des architectures disposant de jeux d'instructions SIMD à vecteurs explicites, la vectorisation s'effectue lors de l'exécution plutôt qu'à la compilation [19].
Bien que ce modèle permette d'atteindre de bonnes performances en restant simple à programmer par rapport à un jeu d'instructions SIMD classique, la gestion dynamique de la divergence de contrôle des threads a un coût en surface, consommation et complexité du matériel.
D'autre part, les techniques actuellement utilisées reposent sur un choix d'ordonnancement statique opéré par le compilateur. Ce choix *a priori* peut s'avérer sous-optimal lors de l'exécution, sans que la micro-architecture ne puisse revenir sur ces décisions. De plus, cette solution exclut l'emploi des jeux d'instructions les plus répandus, qui n'incluent pas d'information de reconvergence, dans les architectures SIMT. Cela constitue un frein à une adoption plus large du modèle SIMT, notamment dans les processeurs généralistes.
Par ailleurs, un certain nombre de solutions ont été proposées dans les années 1980 et 1990, mais semblent être peu connues aujourd'hui, comme en témoignent les mécanismes relativement primitifs employés dans les GPU actuels.

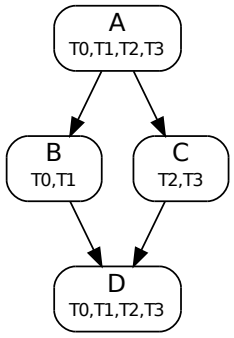
Dans cet article, nous repoussons chacune de ces limitations. D'une part, nous dressons un état de l'art des techniques existantes de gestion de la divergence dans le cadre des architectures SIMT, vectorielles et SIMD. Nous proposons d'autre part un mécanisme dynamique de maintien de la synchronisation qui est réalisé entièrement au niveau micro-architectural et peut opérer sur des jeux d'instructions scalaires conventionnels. Enfin, nous suggérons de réaliser ce mécanisme en réutilisant le matériel de l'unité d'accès mémoire qui est présente dans les processeurs SIMT, permettant une mise en œuvre économe en matériel.
Nous présentons tout d'abord l'aperçu des travaux existants relatifs à la gestion de la divergence de contrôle en SIMT dans la section 2. Nous décrivons ensuite la technique que nous proposons dans la section 3. La section 4 décrit sa mise en œuvre en matériel. Enfin, nous discuterons de la correction de la méthode et quantifierons son efficacité par simulation dans la section 5, puis évoquerons plusieurs perspectives.

2. Travaux existants

Nous présentons dans cette section différentes techniques visant à maintenir et à restaurer la synchronisation entre les threads d'un warp. La majorité de ces méthodes ont été élaborées dans le cadre des processeurs SIMD et vectoriels des années 1980 et 1990. Nous nous permettrons quelques anachronismes en les décrivant au moyen du vocabulaire actuel des GPU.
Nous distinguons les méthodes *explicites*, dont le fonctionnement est exposé au niveau architectural et qui réclament un travail spécifique de la part du compilateur, et les méthodes *implicites*, qui se restreignent au niveau micro-architectural et peuvent opérer sur des jeux d'instructions scalaires classiques.
Les mécanismes de maintien de la convergence répondent à deux problématiques distinctes :

— définir un ordre de parcours du graphe de flot de contrôle : quelle est la branche à exécuter en priorité en cas de divergence,
— détecter la reconvergence.

L'ordre de parcours a une influence significative sur l'efficacité : un ordre non optimal conduira à exécuter plusieurs fois les mêmes blocs de base.



(a) Graphe de contrôle de flot

BB	Threads actifs
A	0, 1, 2, 3
B	0, 1
C	2, 3
D	0, 1, 2, 3

(b) Ordre de parcours optimal

BB	Threads actifs
A	0, 1, 2, 3
B	0, 1
D	0, 1
C	2, 3
D	2, 3

(c) Ordre de parcours sous-optimal

Figure 1 – Exemple de deux parcours SIMT possibles du graphe de contrôle de flot d'un bloc `if-then-else` parcouru par 4 threads. Les blocs de base sont annotés avec les numéros des threads devant les parcourir.

Par exemple, la figure 1 présente l'exemple d'un bloc conditionnel exécuté par 4 threads, nommés T0 à T3. Les threads T0 et T1 prennent la branche contenant le bloc B, tandis que les threads T2 et T3 prennent l'autre branche. Si le bloc D est exécuté avant le bloc C alors que seuls les threads T0 et T1 sont actifs, alors il devra être exécuté une seconde fois pour le compte des threads T2 et T3.
L'unité de gestion du contrôle doit également pouvoir identifier les points de reconvergence du flot de contrôle pour regrouper les threads concernés. Les mécanismes qui seront présentés par la suite sont

résumés d'après ces deux critères dans la table 1.

TABLE 1 – Comparaison synthétique des mécanismes guidant l'ordre de parcours et la reconvergence dans les techniques de gestion des branchements présentés dans cet article. Pour chaque technique, il est précisé si elle autorise la récursivité et les sauts indirects. « ISA » fait référence à des annotations explicites dans le jeu d'instructions. « Sync » indique que les pointeurs de pile des threads restent synchronisés, et s'oppose à « div ».

Technique	Ordre de parcours	Reconvergence	Récursivité	Indirect
Lorie-Strong [17]	Codage prio.	ISA + codage prio.	Non	Non
Pixar Chap [15]	ISA	ISA + pile masques	Non	Non
AMD [1]	ISA	ISA + pile masques	Sync	Non
POMP [14]	ISA	ISA + compteurs	Sync	Non
Intel GMA [12]	ISA	ISA + compteurs	Non	Non
NVIDIA [8]	ISA + pile adresses	ISA + pile masques	Sync	Oui
Intel Sandy Bridge [13]	ISA	ISA + PC	Non	Non
Takahashi [22]	PC + bit activité	PC	Non	Non
SympA'13 [4]	PC + pile adresses	pile masques	Non	Non
Cet article	PC	PC	Div	Oui

Pour les techniques qui permettent les appels de fonctions, la pile d'appel peut être réalisée de deux manières. Les différences entre ces deux solutions se manifestent lors des situations de récursivité.

Les architectures parallèles traditionnelles partagent un pointeur de pile unique par warp. Cela assure que les accès des différents threads à la pile restent toujours synchronisés. En contrepartie, cela restreint l'exécution en mode SIMT aux threads qui partagent le même pointeur de pile en plus du même pointeur d'instructions. Il n'est donc pas possible d'exécuter ensemble des threads se trouvant à des niveaux de récursivité différents.

L'autre solution consiste à maintenir un pointeur de pile indépendant par thread. En cas de récursivité, la divergence de contrôle est réduite. Elle est remplacée par de la divergence mémoire : en effet, lorsque les pointeurs de pile sont désynchronisés, les accès à la pile deviennent irréguliers.

2.1. Explicite avec pile

Pixar Chap

Un travail précurseur notable est le processeur du Pixar Image Computer, une machine dédiée au traitement d'images développée au début des années 1980 [15]. Cette machine est construite sur la base de processeurs SIMD nommés Chap, qui offrent directement dans leur jeu d'instructions des instructions de contrôle reflétant les structures usuelles des langages de programmation impératifs : `if-then`, `else`, `fi`, `while-do`, `done`.

Ces instructions permettent de décrire un flot de contrôle indépendant pour chaque voie SIMD. 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 GPU actuels d'AMD poursuivent une approche similaire [1].

POMP

Une amélioration proposée par Keryell et Paris dans le cadre du projet de machine parallèle POMP consiste à remarquer que les masques conservés dans la pile forment en réalité des histogrammes, car un thread inactif à une profondeur p restera nécessairement inactif à la profondeur $p + 1$. Ainsi, seule la profondeur d'imbrication de la dernière activité de chaque thread nécessite d'être conservée. La quantité de données à conserver passe ainsi de $O(n)$ à $O(\log(n))$ pour n la profondeur d'imbrication maximale [14]. Cette technique de compteurs d'activité est utilisée par les GPU intégrés d'Intel dans les générations antérieures à Sandy Bridge [12].

NVIDIA Tesla

L'objectif affiché de l'architecture GPU NVIDIA Tesla est d'être capable d'exécuter du code généraliste [16]. Son jeu d'instructions emploie des instructions de saut conditionnel à la manière des processeurs scalaires, plutôt que des instructions de contrôle structuré comme les autres GPU. Il est complété par des annotations indiquant les points de divergence et de reconvergence.
La gestion de la divergence est assurée par une technique à base de pile, comme dans Chap et les GPU AMD.
Néanmoins, Tesla inclut moins d'informations dans le code que les autres jeux d'instructions. En particulier, il est nécessaire de retenir les adresses des points de reconvergence dans la pile en plus des masques, car la correspondance entre les points de divergence et les points de reconvergence n'est pas explicite dans le code machine. Cela exclut de fait la possibilité de représenter la pile au moyen de compteurs d'activité.
On notera qu'il est possible de transformer un graphe de flot de contrôle arbitraire en une imbrication de conditionnelles et de boucles lors de la compilation, quitte à devoir répliquer du code lorsque le graphe n'est pas réductible [24]. Il n'est donc pas strictement nécessaire de recourir à un mécanisme tel que celui employé par NVIDIA pour exécuter du code arbitraire. Cependant, le mécanisme utilisé par Tesla peut être étendu au sauts indirects, comme le propose l'architecture Fermi [19]. De plus, cette approche apporte une souplesse possible dans l'ordre de parcours des instructions que les autres techniques ne permettent pas. Elle va dans le sens d'un déplacement des décisions d'ordonnancement depuis l'architecture vers la micro-architecture.

2.2. Implicite avec pile

Nous avons proposé à SympA'13 une solution à base de pile capable de gérer la divergence et la reconvergence sans aucune intervention du compilateur [4]. Elle permet également d'exécuter en mode SIMT du code écrit dans des jeux d'instructions scalaires classiques sans recompilation.
L'emploi d'une pile présente cependant plusieurs inconvénients. La difficulté principale consiste à réaliser de manière efficace une pile en matériel. Les architectures actuelles telles que Tesla emploient un cache spécialisé dans le processeur qui contient quelques entrées du haut de la pile pour chaque warp. Les autres entrées sont conservées dans les niveaux inférieurs de la hiérarchie mémoire. Ce cache a un coût en surface et en consommation, mais surtout nécessite un port d'accès à la mémoire supplémentaire. Par exemple, l'architecture NVIDIA Tesla emploie un cache de 3 blocs de 4 entrées par warp, chaque entrée contenant 64 bits, ce qui représente au total 3 Ko par processeur [2].
Par ailleurs, les méthodes à base de piles sont fragiles face à du logiciel mal écrit. La pile peut déborder ou se trouver dans un état incohérent si les instructions de divergence et de reconvergence ne sont pas correctement équilibrées. Ces cas exceptionnels doivent être détectés et traités par le matériel et le système pour terminer le programme et restaurer la pile dans un état « propre ». La pile ajoute également un état qui doit être sauvegardé et restauré lors des changements de contexte.
Enfin, ces méthodes sont basées sur une machine à états complexe qu'il est difficile de valider pour garantir que la sémantique des programmes exécutés est respectée et que l'état de la pile reste cohérent.
Pour l'ensemble de ces raisons, nous allons nous intéresser à des algorithmes dépourvus de pile, et dont l'état associé reste en espace constant.

2.3. Explicite sans pile

Sandy Bridge

Les GPU intégrés dans les processeurs Intel Sandy Bridge emploient une technique différente de leurs prédécesseurs. Plutôt que de maintenir des piles ou des compteurs, le GPU conserve un compteur de programme (PC) par thread [13]. Ainsi, chaque thread dispose de sa propre copie de l'ensemble des registres architecturaux et du PC, ce qui suffit pour caractériser son état.
Le jeu d'instructions reste similaire à celui des générations précédentes, et contient des instructions explicites pour décrire les conditionnelles et les boucles. Celles-ci mettent à jour les PC individuels de chacun des threads. Il suffit alors au processeur de comparer le PC de chaque thread avec le PC global pour connaître les threads actifs. La reconvergence est observée lorsque les PC de différents threads coïncident.

Lorie-Strong

La solution des PC multiples d'Intel se rapproche d'une réalisation ancienne décrite dans un brevet d'IBM par Lorie et Strong [17]. Dans cette proposition, le compilateur effectue un tri topologique sur le graphe de contrôle de flot et numérote l'ensemble des blocs de base suivant l'ordre induit. Cette numérotation contrôle à la fois l'ordre de parcours du graphe de flot de contrôle et la détection de la reconvergence.
Le processeur maintient à la fois un PC et un numéro de bloc par thread. Lorsqu'une situation de divergence intervient, c'est le bloc d'indice inférieur qui sera exécuté en priorité. Lorsqu'un point de reconvergence éventuel est atteint, les numéros de blocs de chaque thread sont comparés avec le numéro du prochain bloc que le processeur se prépare à exécuter. Lorsqu'il y a égalité, les threads correspondants deviennent actifs.
Dans ces dernières solutions, le mécanisme de contrôle de la divergence est exposé au niveau architectural. L'ordre de parcours du graphe de flot de contrôle est fixé de manière statique lors de la compilation.

2.4. Implicite avec bit d'activité

Une technique proposée par Takahashi en 1997 permet d'exécuter du code en mode SIMT sans nécessiter d'annotations dans le jeu d'instructions [22]. Chaque thread dispose de son propre compteur de programme. Une unité de contrôle parcourt le graphe de contrôle de flot. Lorsqu'un branchement est rencontré, la priorité est donnée à la branche dont le point d'entrée est d'adresse inférieure tant qu'au moins un thread est actif. Lorsque qu'aucun thread n'est actif, l'autre branche est suivie.
La supposition qui est faite semble être que les points de reconvergence se trouvent au point le plus « bas » du code qu'ils dominent, c'est-à-dire à l'adresse la plus grande. La stratégie suivie consiste alors à toujours tenter d'exécuter les instructions d'adresse inférieure lorsqu'il s'agit de décider quelle branche exécuter, de façon à ne pas dépasser un point de convergence éventuel.
L'inconvénient de cette méthode est que les signaux d'activité des threads sont renvoyés avec un cycle de retard à l'unité de contrôle. Au moment où l'unité de contrôle reçoit cette valeur, les informations associées au dernier branchement rencontré ne sont plus disponibles. Le processeur doit donc continuer à parcourir le programme alors que tous les threads sont inactifs. Les blocs *else* sont donc toujours exécutés même en cas de convergence, et les boucles effectuent toujours un tour supplémentaire. Par ailleurs, le fait que le processeur puisse exécuter de manière spéculative des branches qui ne sont suivies par aucun thread laisse la possibilité qu'il rencontre des instructions invalides ou déborde de la zone de code. Contrairement au cas des prédicteurs de branchements des processeurs superscalaires, aucun mécanisme n'est prévu pour retourner à un état précédent non spéculatif.
Pour contourner ces problèmes, les auteurs proposent que le compilateur duplique les instructions de saut concernées. Cependant, cela remet en cause en grande partie l'intérêt de ne pas dépendre du compilateur.

3. Reconvergence implicite sans pile

Nous allons maintenant proposer une technique qui cumule les avantages de la solution des PC multiples des GPU Intel (en espace constant) et de notre proposition précédente (non exposée au niveau architectural). Contrairement à la proposition de Takahashi, elle ne nécessite pas d'exécuter des blocs de base plus souvent que nécessaire.
Malgré son apparente simplicité, cette méthode ne semble pas avoir été étudiée spécifiquement par le passé. Fung considère plusieurs politiques d'ordonnancement dans ses travaux sur la formation dynamique de warps, dont certaines réalisables en espace constant, mais n'envisage pas leur utilisation dans le cadre de la formation « statique » de warps [9, 10].
De même Meng, Tarjan et Skadron comparent une reconvergence opportuniste lorsque les PC de plusieurs threads coïncident avec la reconvergence aux post-dominateurs dans le cadre de leur technique de subdivision dynamique de warps [18].
Ces deux propositions réclament une restructuration complète du pipeline d'exécution et sont gourmands en matériel. Par exemple, Fung estime le surcoût en surface de la formation dynamique de warps à 1,6353 mm^2 en 90 nm [9]. À titre de référence, un processeur SIMT complet de GPU NVIDIA G80 dans

Listing 1 – Exemple de cas où la politique min(PC) se révèle inefficace.

```
void kernel() {
    ...
    if(c) {  // Condition divergente
        f();
    }
    ...
}

void f() {
    ...
}
```

le même processus de fabrication occupe 6,37 mm^2 d'après nos mesures sur une photographie de *die*.
Dans cet article, nous envisageons une architecture plus modeste, qui vise une consommation et une surface réduite. Nous maintenons un PC distinct par thread comme dans la solution d'Intel, et calculons le PC commun à partir de la valeur des PC individuels.

Reconvergence

Nous identifions les situations de reconvergence en comparant les valeurs des PC individuels avec le PC commun sélectionné. À l'inverse de la technique employée dans les GPU Intel, nous effectuons la comparaison en continu, à chaque cycle. Il n'est donc pas nécessaire de signaler explicitement les points de reconvergence éventuels dans le code machine.
Les instructions de branchement sont gérées de manière répartie : chaque thread calcule et maintient à jour son propre PC, à la manière d'un processeur MIMD. La divergence s'opère naturellement lorsque le PC local d'un thread est affecté d'une valeur différente de celle des autres threads.

Ordre de parcours

Un arbitre se charge de déterminer le PC commun, c'est-à-dire l'adresse à laquelle l'instruction suivante sera chargée, à partir des valeurs des PC individuels des threads. Comme dans la solution proposée à SympA'13 et dans une certaine mesure comme dans le brevet de Lorie et Strong, nous choisissons le PC de valeur minimale. Ce choix correspond à la politique DPC dans les travaux de Fung sur la formation dynamique de warps [9].
Cela revient à supposer que les points de reconvergence se trouvent à des adresses plus élevées que le code qui les précède. Collins, Tullsen et Wang ont mesuré que cette supposition s'avérait correcte dans 94 % des branchements conditionnels des SPECint [7]. Les noyaux de calcul CUDA actuels offrant un flot de contrôle nettement plus régulier, nous n'avons rencontré aucun contre-exemple dans les applications parallèles présentées section 5.2, à une exception majeure près. Une illustration de cas posant problème est présenté listing 1.
Lors d'un appel à une fonction `f` à l'intérieur d'un bloc conditionnel, les PC des threads exécutant le bloc pointeront vers `f`. Lorsque le code de cette fonction se trouve à une adresse supérieure à celle du site d'appel, la politique min(PC_i) peut se révéler inefficace, car elle exécutera d'abord le code se trouvant après la fin du bloc conditionnel. La reconvergence avec les threads exécutant `f` ne sera possible qu'à la fin du programme ou l'appel de `f` par tous les autres threads.
Les premières versions du compilateur CUDA se contentant de réaliser *l'inlining* de toutes les fonctions appelées, ce problème ne se posait pas auparavant.
Un première solution consiste à exiger du compilateur qu'il place systématiquement les définitions de fonctions avant leurs appels lors de l'édition des liens. Excepté dans les cas de récursivité croisée, cette solution est toujours applicable. Cependant, il n'est pas toujours praticable de recompiler le code.
Une solution plus générale que nous proposons consiste à prendre en compte la valeur du pointeur de pile (SP_i) de chaque thread `i` lors du choix de la branche à exécuter. Pour donner la priorité au niveau

d'imbrication d'appels de fonctions le plus profond, nous retenons le thread dont le SP_i est minimal [1]. En cas d'ex-aequo, nous choisissons le PC_i minimal comme précédemment.
La table 2 présente l'ordre de parcours du code de la figure 1, dans lequel des instructions de saut ont été insérées. Les valeurs des PC individuels actuels et des PC futurs (NPC) sont indiqués pour chaque cycle. Les threads actifs sont représentés par un PC souligné.

TABLE 2 – Trace d'exécution d'une structure `if-then-else`

PC	Instruction	PC_i 0 1 2 3	NPC_i 0 1 2 3
1	A	$\underline{1}\ \underline{1}\ \underline{1}\ \underline{1}$	2 2 2 2
2	if(!c) br 5	$\underline{2}\ \underline{2}\ \underline{2}\ \underline{2}$	3 3 5 5
3	B	$\underline{3}\ \underline{3}$ 5 5	4 4 5 5
4	br 6	$\underline{4}\ \underline{4}$ 5 5	6 6 5 5
5	C	6 6 $\underline{5}\ \underline{5}$	6 6 6 6
6	D	$\underline{6}\ \underline{6}\ \underline{6}\ \underline{6}$	7 7 7 7

4. Réalisation en matériel

Nous considérons un processeur SIMT muni d'un unique chemin de lecture, décodage et ordonnancement des instructions, et de plusieurs unités de calcul parallèles nommées traditionnellement *processing elements* (PE). Les exemples d'architectures présentées ici à des fins d'illustration comportent 4 PE, mais les techniques que nous proposons visent aussi bien des architectures disposant de 16 à 64 PE comme les processeurs des GPU actuels.
Comme dans toutes les architectures SIMT existantes, la latence du pipeline est masquée au moyen de multithreading matériel [2]. La latence n'est donc pas un facteur critique dans ce type d'architecture.

4.1. Arbitrage

La figure 2 illustre la solution proposée. Dans sa variante la plus simple, chaque PE_i communique les valeurs de son PC_i et SP_i à un arbitre. Cet arbitre consiste en un arbre de réduction qui calcule le minimum $SP : PC = \min(SP_i : PC_i)$, où : représente l'opérateur de concaténation. La valeur calculée représente le compteur de programme commun PC, qui indique l'emplacement de la prochaine instruction à exécuter. L'instruction désignée par PC est alors lue, décodée, et transmise à l'ensemble des PE_i.
Une optimisation possible consiste à incrémenter localement le PC commun à chaque cycle. Ainsi, l'arbitrage n'est nécessaire que lorsqu'une instruction pouvant influer sur le contrôle de flot (typiquement un saut) est exécutée. Le reste du temps, la valeur du PC commun suivra la valeur du PC minimal.
En effet, on peut montrer que si le PC_i d'un sous-ensemble non vide des threads actifs est incrémenté et que le PC_j de tous les autres threads n'est pas modifié, alors le nouveau PC commun est la valeur incrémentée de l'ancien PC commun.

4.2. Similarité avec le chemin d'accès mémoire

Considérons l'unité d'accès au cache de données de premier niveau dans une architecture SIMT telle que NVIDIA Fermi [19], dont une représentation simplifiée est présentée figure 3.
Le cache L1 de données dispose d'un port unique de la largeur d'une ligne de cache, qui est de 128 bits dans notre exemple jouet. Lors d'une instruction de lecture mémoire, chaque PE_i dispose d'une adresse A_i à laquelle il veut accéder. Les bits de poids forts de cette adresse, qui indiquent dans quelle ligne de cache se trouve la donnée, sont envoyés à un arbitre.

1. Nous supposons ici que la pile suit l'organisation traditionnelle en étant ordonnée par adresses décroissantes.
2. Par souci de clarté, nous décrivons notre technique avec un unique thread actif par PE, mais tous les mécanismes présentés dans cette section sont directement généralisables à plusieurs threads par PE, c'est-à-dire plusieurs warps par processeur.

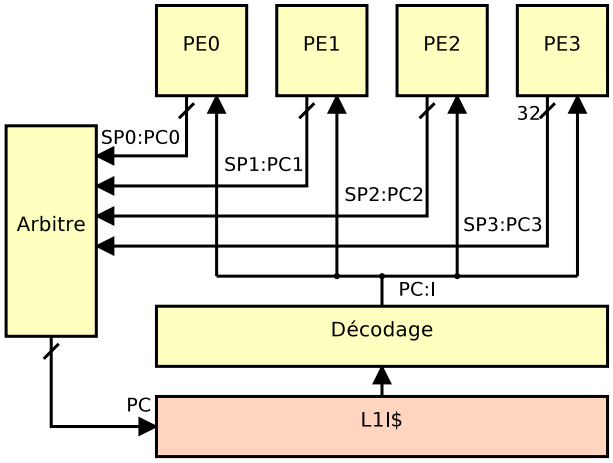


Figure 2 – Chemin des compteurs de programme et des instructions dans notre première proposition de réalisation. « I » désigne l'instruction décodée.

Cet arbitre se charge de sélectionner une adresse parmi celles qu'il reçoit en entrée, par exemple au moyen d'un codeur de priorité choisissant le premier thread actif, et l'envoie au cache L1 de données. La ligne de cache correspondante est lue, puis envoyée au travers d'un réseau d'interconnexion de type commutateur (*crossbar*). Ce dernier route les données vers le ou les PE qui l'ont réclamé, en fonction des poids faibles de l'adresse. Les bits d'activité des PE vers l'arbitre et les bits de sélection revenant de l'arbitre vers les PE ne sont pas représentés afin de ne pas alourdir la figure.
Il est possible que plusieurs adresses pointent sur la même ligne de cache. Dans le cas idéal, tous les PE accèdent à la même ligne de cache et peuvent se partager le port du cache L1. Dans le cas contraire, les accès aux différentes lignes de cache sont sérialisées [20].

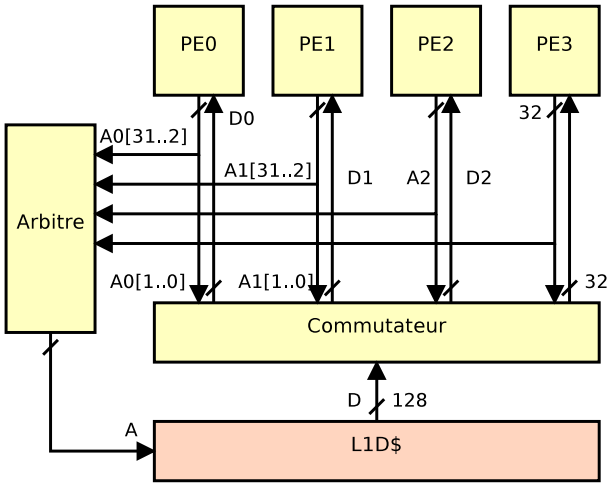


Figure 3 – Diagramme de l'unité de lecture mémoire d'une architecture SIMT.

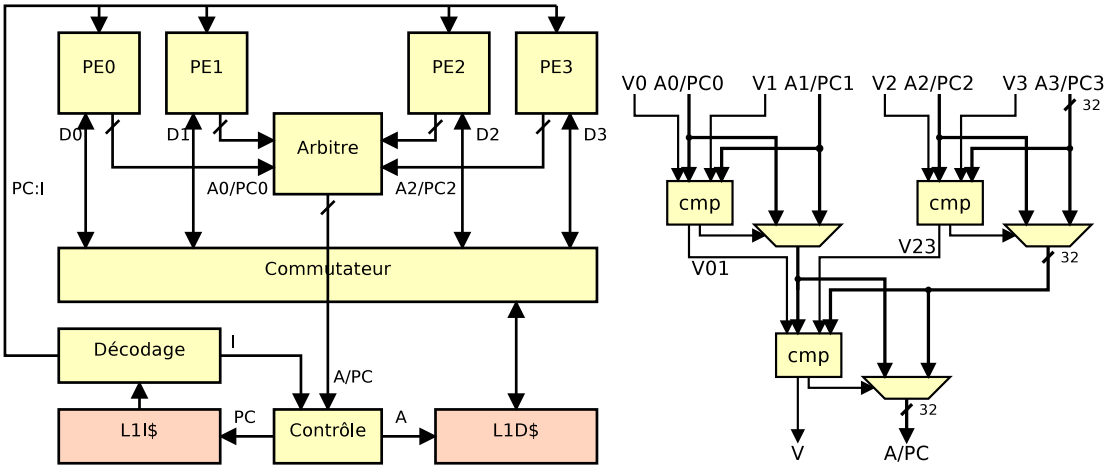
Notons que notre proposition permet de gérer naturellement les conflits et les échecs dans le cache de données. Lorsqu'un thread i est servi, il incrémente son PC_i pour signaler qu'il est prêt à exécuter l'instruction suivante. Les threads ayant rencontré un conflit conserveront la même valeur de PC_i, ce qui les conduira à tenter à nouveau l'opération mémoire au tour d'ordonnancement suivant. Les threads qui rencontrent un échec dans le cache se signalent comme inactifs jusqu'à réception de leur données.

Une fois ces données arrivées, un arbitrage de PC a lieu pour donner l'opportunité à ces threads de « rattraper » les autres. Ce mécanisme permet aux threads ayant réussi leur accès au cache de prendre de l'avance sur ceux qui sont bloqués sur un échec, offrant gratuitement une fonctionnalité analogue à la subdivision dynamique de warps [18].

4.3. Architecture unifiée

Observons la similarité entre l'unité d'accès mémoire et l'unité de suivi du flot de contrôle que nous venons d'aborder. Dans les deux cas, un arbitre effectue une opération de réduction depuis des données envoyées par chaque PE. Nous proposons dans cette section une architecture unifiée permettant d'amortir le coût de la gestion du flot de contrôle en partageant du matériel existant.

Le coût en énergie et en latence d'un arbre de réduction ou de sélection entre les PE est dominé par la traversée des fils, plutôt que par les calculs eux-même. Nous pouvons donc généraliser le circuit d'arbitrage de l'unité mémoire en lui permettant d'effectuer également des calculs de minimums pour un coût additionnel marginal.



(a) Architecture globale (b) Arbitre multi-fonctions

Figure 4 – Architecture unifiée de gestion de divergence de contrôle et de divergence mémoire proposée.

L'architecture proposée est présentée figure 4a. Lors des instructions de saut, l'arbitre reçoit les PC et SP individuels de chaque PE et calcule le minimum, puis met à jour le PC commun maintenu par l'unité de contrôle. Pour les autres instructions, l'unité de contrôle incrémente sa copie locale du PC commun.

Lors des instructions de lecture et d'écriture mémoire, l'arbitre sert à déterminer l'adresse à laquelle accéder dans le cache de données.

La figure 4b présent un diagramme de l'arbitre unifié. Pour chaque PE, il prend en entrée les adresses dans le code ou dans les données (A_i/PC_i), ainsi qu'un bit de validité V_i. En mode arbitrage de PC, le bit de validité V_i indique que le thread i est actif. En mode arbitrage mémoire, il indique que le thread i demande l'accès à la case mémoire d'adresse A_i. Une opération de réduction OU est effectuée sur les bits de validité individuels V_i pour former le bit de validité V du résultat de l'arbitrage. Ainsi, les threads inactifs sont simplement ignorés.

5. Validation

5.1. Correction

La politique de parcours du graphe de contrôle de flot correspond uniquement à une heuristique d'optimisation. Elle n'a pas d'incidence sur la correction de l'exécution : hors boucles d'attente active, dans le pire cas, les instructions de chaque thread seront exécutées séquentiellement sur une seule voie SIMD et la performance sera dégradée au niveau de celle d'un processeur scalaire.

La politique qui détermine l'ordre de parcours du graphe de contrôle de flot est la même que dans la technique proposée à SympA'13. Cependant, la reconvergence est possible à d'autres points que ceux que prévoient la solution à base de pile.

5.2. Validation expérimentale

Nous avons modélisé l'architecture proposée dans le simulateur de GPU Barra [5]. Nous considérons des warps de 32 threads comme dans les architectures NVIDIA actuelles. Les applications de test utilisées sont celles du kit de développement CUDA [21], ainsi que le noyau de calcul FFT du jeu d'applications Parboil de l'UIUC [23]. Les noyaux de calcul de chaque application sont listés table 3 avec leurs nombres d'instructions statiques et dynamiques.

La figure 5 compare le nombre moyen de threads actifs par warps avec la technique proposée, avec la technique de NVIDIA et avec la technique implicite.

TABLE 3 – Noyaux des applications de test du SDK CUDA, avec leur nombre d'instructions PTX statiques, nombre d'instructions assembleur statiques et dynamiques.

Programme	Noyau	Instructions	
		Statiques	Dynamiques
FFT	Parboil_FFT	431	8771584
fastWalshTransform	fwtBatch1Kernel	108	57212928
	fwtBatch2Kernel	46	54263808
	modulateKernel	24	84246528
BlackScholes	BlackScholesGPU	99	5201694720
transpose	transpose_naive	29	1835008
	transpose	42	2752512
quasiRandomGenerator	inverseCNDKernel	147	3383048
sortingNetworks	bitonicSortShared	100	631696
	bitonicMergeGlobal	43	53760
	bitonicMergeShared	71	212736
	bitonicSortShared1	101	573456
binomialOptions	binomialOptions	109	277266944
convolutionSeparable	convolutionRows	214	39260160
	convolutionColumns	234	42946560
3dfd	stencil_3D_16x16_order8	162	3174912
MonteCarlo	MonteCarloOneBlock…	129	27394560
reduction	reduce5_sm10	40	3900
	reduce6_sm10	59	318513600
dwtHaar1D (DWT)	dwtHaar1D	87	9366
matrixMul (MM)	matrixMul	114	1785600

La simulation du noyau FFT échoue avec la technique implicite. En effet, ce noyau contient des appels de fonctions, alors que l'algorithme implicite ne gère pas les instructions `call` et `return`.

Rappelons que des gains de performances ne sont pas le but premier de la méthode que nous proposons. Néanmoins, on note une amélioration du taux d'utilisation des unités SIMD atteignant 7 % dans plusieurs applications par rapport à la technique de base de NVIDIA, et jusqu'à 4 % par rapport à la solution proposée à SympA'13.

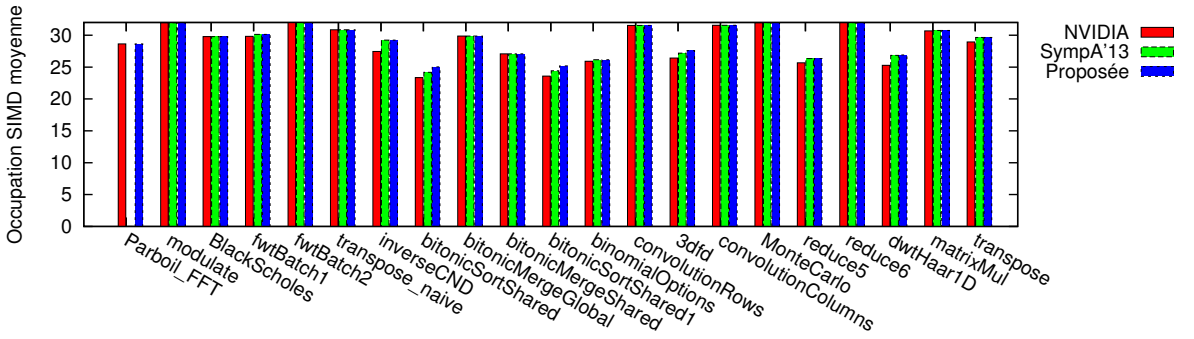


Figure 5 – Nombre moyen de threads actifs par warp pour la technique de reconvergence de Tesla et Fermi [8], celle proposée à Sympa'13 [4] et celle que nous proposons dans cet article.

Ces légères différences s'expliquent par la nécessité d'exécuter plusieurs fois la même instruction lors de la reconvergence dans les méthodes à base de pile. La technique de SympA'13 permet de réduire le nombre de ré-exécutions ; celle que nous proposons élimine totalement ce surcoût.
Dans tous les cas, notre méthode est toujours au moins aussi efficace que celle qui est employée dans les architectures Tesla et Fermi et que la solution proposée précédemment.

6. Conclusions et perspectives

Nous avons présenté une méthode de détection de la divergence et de la reconvergence dans le modèle SIMT qui ne réclame pas de support de la part du compilateur et qui n'impacte pas le jeu d'instructions. Cette méthode peut se réaliser pour un coût en matériel très réduit, grâce à la réutilisation des chemins de données existants de l'unité de gestion de la divergence mémoire.
La politique de parcours du graphe de contrôle de flot que nous avons considérée dans cet article est celle du SP : PC minimal, qui donne de bons résultats en pratique pour un faible coût matériel. Cependant, il pourrait être avantageux de considérer d'autres politiques d'ordonnancement. En particulier, on peut envisager d'employer une politique spéculative, pour ne pas payer le coût en latence de l'arbitrage. Dans ce cas, l'unité de contrôle devient un prédicteur de saut, comme dans les processeurs superscalaires. Notons qu'en cas de divergence lors d'une instruction de saut conditionnel, la prédiction faite sera toujours correcte, et aura uniquement une influence sur l'ordre de parcours des branches. De même, l'unité de comparaison du PC commun avec le PC individuel en fin de pipeline devint l'unité *commit*, qui se charge de vérifier que la spéculation était correcte avant d'écrire le résultat de l'exécution dans le banc de registres.
Une autre analogie possible consiste à remarquer que l'unité de contrôle joue le même rôle que l'unité scalaire dans les processeurs vectoriels et SIMD classiques. De manière similaire à ce qui est fait dans ces architectures, on peut envisager d'adjoindre à l'unité scalaire une unité arithmétique et des registres scalaires pour factoriser les calculs similaires effectués par plusieurs threads, qui représentent une quantité significative des calculs et des registres [3, 6].
Les techniques de gestion de la divergence et de la reconvergence forment la base indispensable sur laquelle peuvent s'appuyer des politiques d'ordonnancement de threads de plus haut niveau tels que la formation dynamique de warps [10] et la subdivision dynamique de warps [18].

Bibliographie

1. AMD. *Evergreen Family Instruction Set Architecture : Instructions and Microcode*, December 2009.
2. Caroline Collange. Analyse de l'architecture GPU Tesla. Technical Report hal-00443875, HAL-CCSD, Jan 2010.
3. Caroline Collange, Marc Daumas, David Defour, et Regis Olivès. Fonctions élémentaires sur GPU exploitant la localité de valeurs. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2008.

4. Caroline Collange, Marc Daumas, David Defour, et David Parello. Étude comparée et simulation d'algorithmes de branchements pour le GPGPU. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.
5. Caroline Collange, Marc Daumas, David Defour, et David Parello. Barra : a parallel functional simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.
6. Caroline Collange, David Defour, et Yao Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume LNCS 6043, pages 46–55, 2009.
7. Jamison D. Collins, Dean M. Tullsen, et Hong Wang. Control flow optimization via dynamic reconvergence prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2004.
8. Brett W. Coon et John Erik Lindholm. System and method for managing divergent threads in a SIMD architecture. US Patent 7353369, April 2008.
9. Wilson Wai Lun Fung. Dynamic warp formation : exploiting thread scheduling for efficient MIMD control flow on SIMD graphics hardware. Master's thesis, University of British Columbia, 2008.
10. Wilson Wai Lun Fung, Ivan Sham, George Yuan, et 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.
11. Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, et Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4) :13–27, 2008.
12. Intel. *Intel G45 Express Chipset Graphics Controller PRM, Volume Four : Subsystem and Cores*, February 2009.
13. Intel. *Intel HD Graphics OpenSource PRM Volume 4 Part 2 : Subsystem and Cores – Message Gateway, URB, Video Motion, and ISA*, July 2010.
14. Ronan Keryell et Nicolas Paris. Activity counter : New optimization for the dynamic scheduling of SIMD control flow. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 184–187, 1993.
15. Adam Levinthal et Thomas Porter. Chap - a SIMD graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, 1984.
16. John Erik Lindholm, John Nickolls, Stuart Oberman, et John Montrym. NVIDIA Tesla : A unified graphics and computing architecture. *IEEE Micro*, 28(2) :39–55, 2008.
17. Raymond A. Lorie et Hovey R. Strong. Method for conditional branch execution in SIMD vector processors. US Patent 4435758, March 1984.
18. Jiayuan Meng, David Tarjan, et Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3) :235–246, 2010.
19. John Nickolls et William J. Dally. The GPU computing era. *IEEE Micro*, 30 :56–69, March 2010.
20. NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.2*, 2010.
21. NVIDIA CUDA SDK, 2010.
22. Yoshizo Takahashi. A mechanism for SIMD execution of SPMD programs. In *Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, HPC-ASIA '97, pages 529–534, 1997.
23. UIUC Parboil Benchmarks, 2010. `http://impact.crhc.illinois.edu/parboil.php`.
24. Fubo Zhang et Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4) :231–245, 2004.