Assouplir les contraintes des architectures SIMT à faible coût

Nicolas Brunie, Caroline Collange

▶ **To cite this version:**

Nicolas Brunie, Caroline Collange. Assouplir les contraintes des architectures SIMT à faible coût. [Rapport de recherche] ENS Lyon. 2011. ensl-00649186

HAL Id: ensl-00649186
https://ens-lyon.hal.science/ensl-00649186

Submitted on 7 Dec 2011

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

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

Assouplir les contraintes des architectures SIMT à faible coût

Nicolas Brunie* — **Sylvain Collange***

* *ENS de Lyon, Université de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL)*
46 allée d'Italie, 96364 Lyon Cedex 07
nicolas.brunie@ens-lyon.fr, sylvain.collange@gmail.com

RÉSUMÉ. *Les architectures parallèles qui obéissent au modèle SIMT telles que les GPU tirent parti de la régularité des applications en exécutant plusieurs threads concurrents sur des unités SIMD de manière synchrone. Lorsque les threads empruntent des chemins divergents dans le graphe de flot de contrôle, leur exécution est séquentialisée jusqu'au prochain point de convergence. La reconvergence doit être effectuée au plus tôt de manière à maximiser l'occupation des unités SIMD. Nous proposons dans cet article deux techniques permettant de traiter la divergence de contrôle en SIMT et d'identifier dynamiquement les points de reconvergence, dont une qui opère en espace constant et gère les sauts indirects et la récursivité. Nous évaluons une réalisation matérielle consistant à partager le matériel existant de l'unité de gestion de la divergence mémoire. En termes de performances, cette solution est au moins aussi efficace que les techniques de l'état de l'art employés par les GPU actuels.*

ABSTRACT. *Parallel architectures following the SIMT model such as GPUs benefit from application regularity by issuing concurrent threads running in lockstep on SIMD units. As threads take different paths across the control-flow graph, lockstep execution is partially lost, and must be regained whenever possible in order to maximize the occupancy of SIMD units. In this paper, we propose two techniques to handle SIMT control divergence and identify reconvergence points. The most advanced one operates in constant space and handles indirect jumps and recursion. We evaluate a hardware implementation which leverage the existing memory divergence management unit. In terms of performance, this solution is at least as efficient as state of the art techniques in use in current GPUs.*

MOTS-CLÉS : *Reconvergence de flot de contrôle, SIMD, SIMT, GPU*

KEYWORDS: *Control flow reconvergence, 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 (Garland *et al.*, 2008). 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 (Nickolls *et al.*, 2010).

Les threads individuels d'un warp peuvent suivre chacun des chemins différents dans le graphe de flot de contrôle. Pour maintenir le mode de fonctionnement SIMD, le processeur parcourt séquentiellement l'ensemble des chemins qui sont empruntés par au moins un thread du warp. Les autres threads du warp sont temporairement désactivés pendant l'exécution des instructions qui ne les concernent pas.

Bien que ce modèle permette d'atteindre de bonnes performances en restant simple à programmer par rapport à un jeu d'instructions SIMD classique, nous verrons que la gestion dynamique de la divergence de contrôle des threads telle qu'elle est réalisée actuellement présente 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 élémentaires considérés dans la littérature récente (Fung *et al.*, 2009).

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

deux mécanismes dynamiques de maintien de la synchronisation qui sont réalisés entièrement au niveau micro-architectural et peuvent opérer sur des jeux d'instructions scalaires conventionnels. Enfin, nous comparons deux réalisations possibles de ce second mécanisme : une unité indépendante basée sur un arbre de réduction et une unité combinée avec l'unité d'accès mémoire qui est présente dans les processeurs SIMT.

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 les techniques que nous proposons dans les sections 3 et 4. La section 5 décrit sa mise en œuvre en matériel. Enfin, nous quantifions son efficacité par simulation et évaluons le coût d'une mise en œuvre matérielle des méthodes proposées dans la section 6.

2. État de l'art du SIMT

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.

2.1. *Problématique*

Les mécanismes de maintien de la convergence répondent à deux problématiques distinctes :

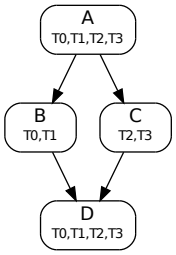
– définir un ordre de parcours du graphe de flot de contrôle : quelle est la branche à exécuter en priorité en cas de divergence,

– 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.

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 être à même d'identifier les points de reconvergence du flot de contrôle pour regrouper les threads concernés. Les



(a) Graphe de flot de contrôle

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 flot de contrôle d'un bloc `if-then-else` parcouru par 4 threads. Les blocs de base sont annotés avec les numéros des threads devant les parcourir.

mécanismes qui seront présentés par la suite sont résumés d'après ces deux critères dans la table 1.

2.2. *Explicite avec pile*

Nous considérons maintenant en détail chaque technique de contrôle de divergence listée table 1, en commençant par celles qui s'appuient sur des structures de piles.

2.2.1. *Compilateurs pour Illiac IV*

La machine SIMD Illiac IV permet un contrôle différencié en fonction des données. Bien que la machine elle-même et son jeu d'instruction ne permettent pas de décrire des structures de contrôles basées sur des conditions locales, des compilateurs de langages de plus haut niveau tels que TRANQUIL (Abel *et al.*, 1969) et Glypnir (Lawrie *et al.*, 1975) permettent une gestion logicielle de la divergence et la reconvergence. Les deux compilateurs se basent sur une pile de masques réalisée en logiciel.

2.2.2. *Pixar Chap*

En ce qui concerne les réalisations entièrement matérielles, un travail précurseur notable est le processeur du Pixar Image Computer, une machine dédiée au traitement d'images développée au début des années 1980 (Levinthal *et al.*, 1984). 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

Tableau 1 – Comparaison synthétique des mécanismes guidant l'ordre de parcours et la reconvergence dans les techniques de gestion des branchements présentées dans cet article. Pour chaque technique, il est précisé si elle permet la réalisation efficace des instructions `break` et `continue` (B), et si elle autorise la récursivité (R), les sauts arbitraires type `goto` (G), les sauts indirects (I). « ISA » fait référence à des annotations explicites dans le jeu d'instructions.

Technique	Ordre de parcours	Reconvergence	B	R	G	I
Illiac IV (Abel *et al.*, 1969; Lawrie *et al.*, 1975)	Statique	Pile masques	-	-	-	-
Pixar Chap (Levinthal *et al.*, 1984)	ISA	ISA + pile masques	-	-	-	-
AMD (AMD, 2009)	ISA	ISA + pile masques	✓	✓	-	-
POMP (Keryell *et al.*, 1993)	ISA	ISA + compteurs	✓	✓	-	-
Intel GMA (Int, 2009)	ISA	ISA + compteurs	✓	-	-	-
NVIDIA (Coon *et al.*, 2008; Coon *et al.*, 2011)	ISA + pile adresses	ISA + pile masques	✓	✓	✓	✓
IPDOM (Fung *et al.*, 2009)	ISA + pile adresses	ISA + pile masques	-	✓	✓	-
C*-Hypercube (Quinn *et al.*, 1988)	PC	ISA + PC	✓	-	-	-
Intel Sandy Bridge (Int, 2010)	ISA	ISA + PC	✓	-	-	-
Lorie-Strong (Lorie *et al.*, 1984)	Codage prio.	ISA + codage prio.	✓	-	-	-
Takahashi (Takahashi, 1997)	PC + bit activité	PC	✓	-	✓	-
RIP (section 3)	PC + pile adresses	pile masques	✓	✓	✓	-
RIA (section 4)	PC	PC	✓	✓	✓	✓

courant et deux piles de masques, servant respectivement aux blocs conditionnels (`if-then-else`) et aux boucles.

Ce modèle peut être étendu pour gérer l'équivalent des instructions `break` et `continue` du C, qui sont notamment présentes dans le langage MPL pour MasPar dès 1990 (Mas, 1992), ou plus récemment dans le langage de *shaders* graphique HLSL de Microsoft Direct3D depuis la version 9.0c en 2004. Il s'agit de l'approche poursuivie

par les GPU GeForce 6xxx et 7xxx de NVIDIA (Mes, 2011) et par l'ensemble des GPU actuels d'AMD (AMD, 2009).

2.2.3. *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 (Keryell *et al.*, 1993). Cette technique de compteurs d'activité est utilisée par les GPU intégrés d'Intel dans les générations antérieures à Sandy Bridge (Int, 2009).

2.2.4. *NVIDIA Tesla*

L'architecture GPU NVIDIA Tesla a pour objectif affiché la capacité d'exécuter du code généraliste (Lindholm *et al.*, 2008). 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. Il conserve cependant les instructions spécifiques dédiées au support des structures `break` et `continue`. Des tests que nous avons effectués ont montré que l'algorithme utilisé sur les GPU NVIDIA est très proche de celui décrit dans deux brevets (Coon *et al.*, 2008; Coon *et al.*, 2011).

La gestion de la divergence est assurée par une technique à base de pile, comme dans Chap, les GPU AMD ou la génération précédente de GPU NVIDIA. 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 (Zhang *et al.*, 2004). Il n'est donc pas strictement nécessaire de recourir à un mécanisme tel que celui employé par NVIDIA pour exécuter du code arbitraire. Cependant, la technique à base de sauts et annotations permet d'éviter la duplication statique de code en la remplaçant par de la duplication dynamique (Diamos *et al.*, 2011). Le mécanisme utilisé par Tesla peut également être étendu à certains sauts indirects, comme le propose l'architecture Fermi (Nickolls *et al.*, 2010).

2.2.5. *IPDOM*

Fung et Aamodt proposent en 2007 une variante de la technique employée par NVIDIA, en conservant uniquement les instructions de saut et de reconvergence. Les points de reconvergence sont choisis à la compilation et insérés dans le code machine. Les auteurs proposent d'utiliser à cet effet les post-dominateurs immédiats [1] dans le graphe de contrôle de flot (Fung *et al.*, 2009). Ainsi, le comportement obtenu est équivalent à celui de Chap pour des graphes de flot de contrôle structurés, tout en étant applicable à des graphes arbitraires. Cette technique ne permet cependant pas d'assurer la reconvergence au plus tôt lorsque le code contient des structures de type `break` et `continue` (Fung *et al.*, 2011), contrairement aux solutions employées par l'environnement MasPar et par l'ensemble des GPU programmables.

2.3. *Explicite sans pile*

Parallèlement au développement des techniques à base de pile des architectures SIMD, des solutions de gestion de la divergence ont été élaborées pour des architectures plus proches du modèle MIMD.

2.3.1. *C* pour Hypercube*

Un compilateur du langage SIMD C* pour des architectures MIMD (Quinn *et al.*, 1988) répond à une problématique voisine de la gestion de la divergence en SIMT. En effet, l'objectif est de garantir l'exécution synchrone d'un ensemble de processeurs exécutant chacun le même code sur des données différentes, tout en minimisant le nombre de synchronisations entre les processeurs (dits PE dans la suite).

Dans cette environnement, chaque thread dispose de son propre compteur de programme (PC). L'ordre de parcours du graphe de flot de contrôle est décidé par un processeur maître, qui maintient à jour un PC maître (MPC). En cas de divergence, le MPC est calculé pour suivre la première branche active dans l'ordre textuel du programme, soit le minimum des PC individuels.

Chaque PE exécute un ensemble de threads, également synchronisés entre eux. Seules les instructions des threads dont le PC correspond au PC commun sont exécutées, grâce à une file de priorité. Ainsi, le surcoût lié à la divergence des threads est amoindri.

Un mécanisme approchant a été proposé en 2011 dans le cadre des architectures SIMT (Fung *et al.*, 2011). Dans cette proposition, la logique de gestion de la divergence est réalisée en matériel plutôt qu'en logiciel, et repose sur des masques plutôt que des PC multiples. Le MPC est maintenu au moyen d'une pile, plutôt que recalculé.

---

1. Un bloc de base B est un *post-dominateur* d'un bloc A (ou *post-domine* A) si tous les chemins de A jusqu'à la fin du programme passent par B. Le post-dominateur immédiat est le premier post-dominateur dans l'ordre d'exécution du programme.

2.3.2. *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 PC par thread (Int, 2010). 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.

2.3.3. *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 (Lorie *et al.*, 1984). Dans cette proposition, le compilateur effectue un tri topologique sur le graphe de flot de contrôle 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.

Diamos et ses coauteurs formalisent cette approche en présentant un algorithme permettant de calculer l'ordre optimal des blocs de base et proposent une réalisation logicielle (Diamos *et al.*, 2011).

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 (Takahashi, 1997). Chaque thread dispose de son propre compteur de programme. Une unité de contrôle parcourt le graphe de flot de contrôle. 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.

Tout comme dans la technique de (Quinn *et al.*, 1988), 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 avec pile

Nous proposons dans cette section une solution à base de pile capable de gérer la divergence et la reconvergence sans aucune intervention du compilateur (Collange *et al.*, 2009a). Elle permet également d'exécuter en mode SIMT du code écrit dans des jeux d'instructions scalaires classiques sans recompilation. Nous nous référerons à cette technique sous le nom de *reconvergence implicite à base de pile* (RIP).

Elle repose sur l'utilisation d'une pile de masques réalisée par des compteurs d'activité semblables à ceux utilisés par POMP (Keryell *et al.*, 1993) et une pile d'adresses permettant de gérer les structures de contrôles strictement imbriquées. La stratégie générale consiste 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.

3.1. *Structures conditionnelles*

Lorsqu'un branchement vers l'avant divergeant est rencontré, le masque courant et l'adresse de destination sont sauvegardées dans les piles respectives. Le masque courant est mis à jour et l'exécution se poursuit dans le branche qui traverse le branchement sans le prendre. Le haut de la pile d'adresse contient alors la cible du saut, soit l'autre branche en attente d'être exécutée (fig. 2).

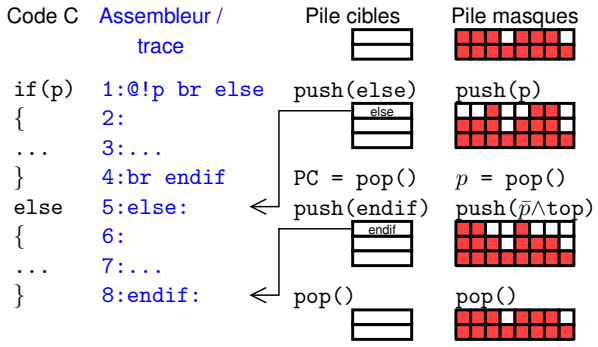


Figure 2 – Illustration du fonctionnement de la méthode RIP sur un exemple de structure conditionnelle.

Après l'exécution de chaque instruction, le compteur d'instruction suivante (NPC=PC+1) est comparé avec le haut de la pile d'adresse. En cas d'égalité, le point de convergence est atteint (`endif`). L'exécution peut alors continuer avec le masque sauvegardé, après avoir dépilé le masque et l'adresse. Plusieurs points de convergence peuvent être présents sur la même instruction. Dans ce cas, il faut répéter la comparaison et dépiler autant de fois que nécessaire, ce qui peut nécessiter la ré-exécution de l'instruction.

Il se peut également que l'instruction courante soit un saut vers une adresse se trouvant au-delà de la cible du saut précédant, par exemple avant un bloc `else` (ligne 4 de la figure 2). Dans ce cas, l'adresse NPC est strictement supérieure à l'adresse présente au sommet de la pile d'adresse. Le contrôle est alors transféré à l'autre branche en attente, en échangeant NPC avec le haut de la pile d'adresse et en inversant le masque courant.

3.2. *Boucles*

Les boucles sont identifiées par un branchement arrière, conditionnel ou inconditionnel. Lorsqu'un tel branchement est rencontré et avant d'effectuer le saut, on compare le sommet de la pile d'adresse avec l'adresse de l'instruction suivante (NPC).

En cas de différence et si le saut est pris par au moins un thread, on empile NPC et le masque courant sur la pile d'adresse, de façon à retenir l'adresse de fin de la boucle.

En cas d'égalité et si aucun thread n'effectue le saut (sortie de boucle), le masque est restauré d'après le masque présent dans la pile et l'entrée correspondante est dépilée de la pile d'adresse. Cette entrée correspond à l'entrée relative à l'entrée dans la boucle et ce grâce au fait de ne considérer que des structures strictement imbriquées.

3.3. *Appels de fonctions*

Les appels de fonctions et leur retour emploient une pile d'adresses séparée comme dans une architecture scalaire. Lorsqu'un appel de fonction pris par au moins un thread est rencontré, l'adresse de l'instruction suivante est empilée à la fois sur la pile d'appel et sur la pile d'adresses de branchements.

Nous ajoutons une contrainte pour que le mécanisme d'échange entre le NPC et le haut de la pile décrit section 3.1 n'ait lieu que lorsque le haut de la pile d'appel est différent du haut de la pile d'adresses de branchements. Ainsi, le code situé après le site d'appel n'est exécuté qu'une fois que tous les threads ont quitté la fonction au travers d'une instruction `return`.

3.4. *Autres structures de contrôle*

Les instructions de contrôle structuré telles que les instructions `break` et `continue` du langage C sont identifiées par des branchements inconditionnels vers l'avant ou vers l'arrière pris par l'ensemble des threads actifs. Leur fonctionnement est donc similaire à celui décrit pour la gestion des boucles et des structures conditionnelles.

3.5. *Limitations*

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 (Collange, 2010).

Par ailleurs, les méthodes explicites 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 considérer un algorithme dépourvu de pile, et dont l'état associé reste en espace constant.

4. Reconvergence implicite sans pile

Nous proposons maintenant la technique de *reconvergence implicite à base d'arbitrage entre PC* (RIA) 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. Les techniques à base de priorité respectivement proposées par Lorie et Strong et par Diamos réclament toutes deux l'implication du compilateur pour numéroter les blocs de base et annoter les points de reconvergence.

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 (Fung *et al.*, 2009).

Meng, Tarjan et Skadron considèrent dans le cadre de leur technique de subdivision dynamique de warps une reconvergence opportuniste lorsque les PC de plusieurs threads coïncident (Meng *et al.*, 2010). Cependant, le principal moyen de reconvergence repose toujours sur des points de synchronisation explicites et une pile de masques.

Dans la proposition RIA, nous maintenons un PC distinct par thread comme dans la solution d'Intel, et calculons le PC commun MPC à partir de la valeur des PC individuels.

4.1. *Reconvergence*

Nous identifions les situations de reconvergence en comparant les valeurs des PC individuels avec celle du MPC. 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.

4.2. *Ordre de parcours*

Un arbitre se charge de déterminer le MPC, 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 RIP 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 (Fung *et al.*, 2009).

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 (Collins *et al.*, 2004). 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 6.1, à une exception majeure près due aux appels de fonction, sur laquelle nous reviendrons dans la section 4.3.

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.

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

MPC	Instruction	PC_i	NPC_i
		0 1 2 3	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

La politique de parcours du graphe de flot de contrôle 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 flot de contrôle est la même que dans la technique RIP. Cependant, la reconvergence est possible à d'autres points que ceux que prévoient la solution à base de pile.

Listing 1 – Exemple de cas où la politique $\mathrm{MPC} = \min(\mathrm{PC}_i)$ se révèle inefficace.

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

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

4.3. *Appels de fonction*

La politique $\min(\mathrm{PC}_i)$ ne se prête pas naturellement aux appels de fonctions. 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 $\min(\mathrm{PC}_i)$ 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.

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 pratique ou même possible 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[2]. En cas d'ex-aequo, nous choisissons le PC_i minimal comme précédemment.

4.4. *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

2. Nous supposons ici que la pile suit l'organisation traditionnelle en étant ordonnée par adresses décroissantes.

qui partagent le même pointeur de pile en plus du même pointeur d'instructions. Il n'est donc pas possible d'exécuter simultanément des threads se trouvant à des niveaux de récursivité différents, même si leurs PC coïncident.

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. La technique RIA permet naturellement de gérer la récursivité désynchronisée.

5. Réalisation de RIA 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[3]. La latence n'est donc pas un facteur critique dans ce type d'architecture, comme nous le verrons dans la section 6.3.

5.1. *Arbitrage*

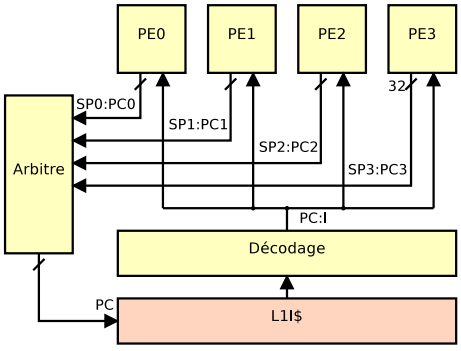


Figure 3 – Chemin des compteurs de programme et des instructions dans notre première proposition de réalisation. ≪ I ≫ désigne l'instruction décodée.

3. 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.

La figure 3 illustre la solution proposée pour la réalisation de RIA en matériel. 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 $\mathrm{SP} : \mathrm{PC} = \min(\mathrm{SP}_i : \mathrm{PC}_i)$, où : représente l'opérateur de concaténation. La valeur calculée représente le compteur de programme commun MPC, qui indique l'emplacement de la prochaine instruction à exécuter. L'instruction désignée par MPC est alors lue, décodée, et transmise à l'ensemble des PE_i.

Une optimisation possible consiste à incrémenter localement le MPC à chaque cycle. Ainsi, l'arbitrage n'est nécessaire que lorsqu'une instruction pouvant influer sur le flot de contrôle (typiquement un saut) est exécutée. Le reste du temps, la valeur du MPC 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 MPC est la valeur incrémentée de l'ancien MPC.

5.2. *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 (Nickolls *et al.*, 2010), dont une représentation simplifiée est présentée figure 4.

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.

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 les ont réclamées, 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 (NVIDIA, 2010).

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

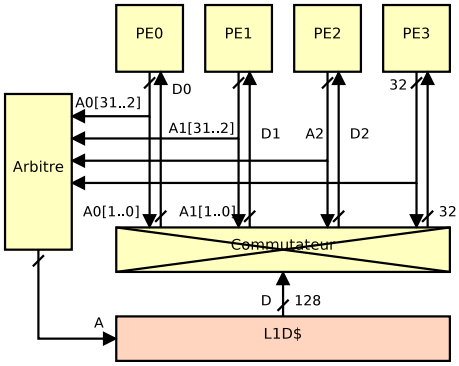


Figure 4 – Diagramme de l'unité de lecture mémoire d'une architecture SIMT.

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 donnée. 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 (Meng *et al.*, 2010).

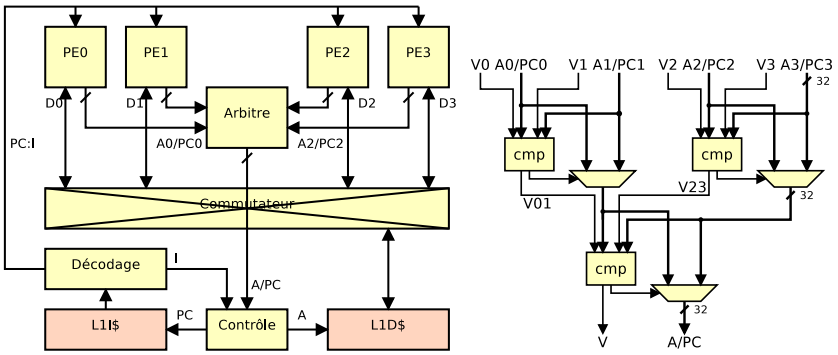
5.3. *Architecture unifiée*

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.

L'architecture proposée est présentée figure 5a. 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 MPC maintenu par l'unité de contrôle. Pour les autres instructions, l'unité de contrôle incrémente sa copie locale du MPC.

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 5b présente un diagramme de l'arbitre unifié. Pour chaque PE, il prend en entrée les adresses dans le code ou dans les données ($\mathrm{A}_i/\mathrm{PC}_i$), 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



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

Figure 5 – Architecture unifiée de gestion de divergence de contrôle et de divergence mémoire proposée.

de validité individuels V_i pour former le bit de validité V du résultat de l'arbitrage. Ainsi, les threads inactifs sont simplement ignorés.

6. Validation

Nous évaluons dans cette section chacune des techniques proposées. Nous simulons les architectures RIP et RIA sur un simulateur de GPU, et menons une étude de sensibilité sur la latence de l'unité de gestion des branchements. Nous synthétisons les deux réalisations matérielles proposées pour la politique RIA dans un processus de l'état de l'art.

6.1. *Méthodologie de simulation*

Nous avons modélisé les techniques proposées dans le simulateur de GPU Barra (Collange *et al.*, 2010). Nous considérons des warps de 32 threads comme dans les architectures NVIDIA actuelles. Nous employons un modèle de performance s'approchant du niveau cycle, qui modélise une architecture GPU inspirée par NVIDIA Tesla et Fermi. Son pipeline d'exécution est présenté figure 6.

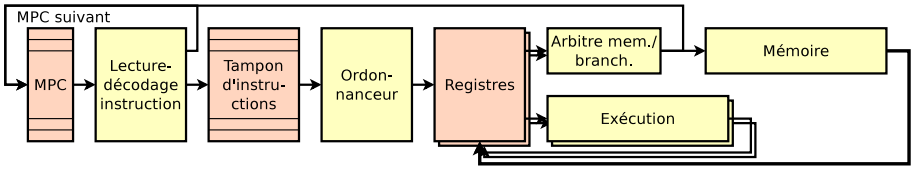


Figure 6 – Diagramme du pipeline d'exécution de l'architecture considérée pour les expériences de simulation.

Les principaux paramètres du processeur simulé sont listés table 3. Le processeur dispose de 32 unités d'exécutions SIMD, permettant un débit crête d'une instruction arithmétique sur un warp par cycle. Le modèle tient compte des dépendances de données entre instructions, des conflits de banc en mémoire partagée et de la fusion des accès mémoire (*coalescing*). La politique d'ordonnancement des warps considérée consiste à sélectionner l'instruction prête la plus agée. Nous considérons une mémoire limitée en débit et de latence fixée, suivant la méthodologie observée par (Gebhart *et al.*, 2011). Nous calibrons les paramètres du modèle d'après les résultats de micro-tests (Collange *et al.*, 2009b).

Tableau 3 – Paramètres de simulation.

Paramètre	Valeur
Nombre de PE	32
Largeur de warp	32
Nombre de warps	32
Registres vectoriels	512
Fréquence d'horloge	1 GHz
Latence des PE	8 cycles
Débit mémoire par processeur	10 Go/s
Latence mémoire	330 ns

Les applications de test utilisées sont issues de la suite de tests Rodinia (Che *et al.*, 2009) et du kit de développement CUDA (NVI, 2010), Les applications de chaque suite sont listées table 4.

Tableau 4 – Liste des applications de test utilisées.

Rodinia	CUDA SDK
backprop	binomialOptions
bfs	BlackScholes
cfd	dwtHaar1D
hotspot	fastWalshTransform
lud	histogram
nw	matrixMul
srad	MonteCarlo
	sortingNetworks
	transpose

6.2. *Taux d'activité*

La figure 7 compare le nombre moyen de threads actifs par warps entre les techniques de NVIDIA, RIP et RIA. Rappelons que des gains de performances ne sont pas

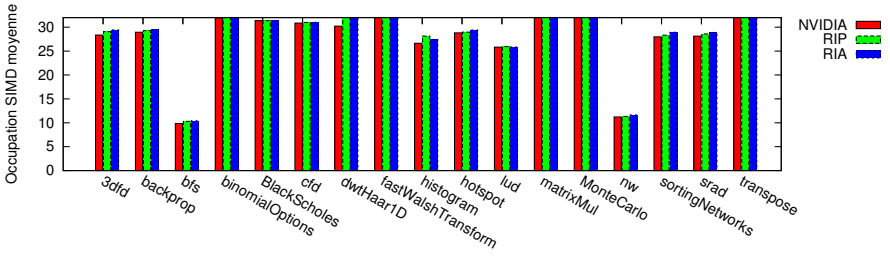


Figure 7 – Nombre moyen de threads actifs par warp pour la technique de reconvergence de Tesla et Fermi, notre méthode implicite à base de pile (RIP) et notre méthode à base d'arbitrage entre PC (RIA).

le but premier des solutions que nous proposons. Néanmoins, on note une amélioration moyenne du taux d'utilisation des unités SIMD par rapport à la technique de base de NVIDIA de 1,6 % pour la technique RIP et de 2,1 % pour la technique RIA.

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 RIP permet de réduire le nombre de ré-exécutions ; RIA élimine totalement ce surcoût. Pour toutes les applications considérées, chacune des deux méthodes proposées est toujours au moins aussi efficace en termes de taux d'utilisation que celle qui est employée dans les architectures Tesla et Fermi.

6.3. *Sensibilité à la latence des branchements*

La contrepartie de la solution que nous proposons est une augmentation de la latence des instructions de saut. En effet, le MPC de l'instruction suivante n'est disponible qu'après la traversée de l'unité d'arbitrage. Afin de quantifier l'impact de cette latence supplémentaire sur la performance globale, ainsi que pour déterminer les contraintes qui influeront sur la conception de l'unité d'arbitrage, nous menons une analyse de sensibilité sur la latence des branchements.

Nous étudions trois configurations :

1) une prédiction de branchement parfaite, où la valeur du MPC suivant est connue dès le cycle suivant l'ordonnancement d'une instruction,

2) une latence de branchement de 4 cycles, correspondant à une unité d'arbitrage optimisée pour minimiser la latence,

3) une latence de branchement de 8 cycles, correspondant à une unité optimisée pour une faible consommation.

À titre de référence, nous considérons également la technique à base de pile de NVIDIA et notre solution implicite. Les temps d'exécution dans chaque cas tiennent comptent des instructions de reconvergence qui ne sont pas nécessaires aux techniques implicites, donnant lieu à une sur-estimation du temps d'exécution pour ces configurations.

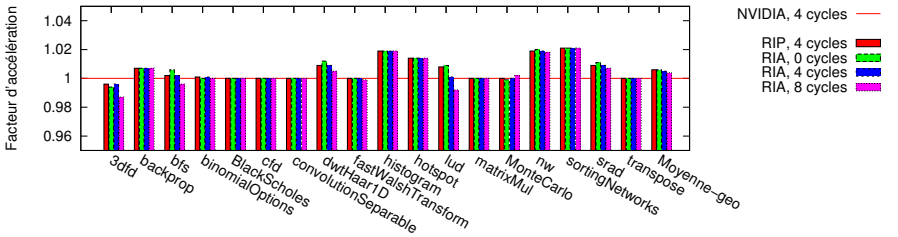


Figure 8 – Performance de la technique RIP, ainsi que de la technique RIA pour trois configurations du modèle de performance, en comparaison avec la technique employée par NVIDIA.

La figure 8 présente les performance respectives de RIP et RIA dans les trois configurations étudiées, en prenant comme point de référence la solution de NVIDIA. Nous incluons la moyenne géométrique des facteurs d'accélération obtenus pour chaque application. Nous observons que l'influence de la latence des branchements sur la performance globale est négligeable, étant inférieure à un pour mille, et se trouve compensée en moyenne par le taux d'occupation supérieur des unités observé dans la section précédente. En effet, le multithreading massif mis en œuvre par une architecture orientée débit telle qu'un GPU est conçu pour absorber des latences mémoire de l'ordre de 300 cycles. Le même mécanisme permet aisément de masquer un délai de quelques cycles lors des instructions de branchements.

6.4. *Coût matériel*

Afin d'évaluer la performance de chaque réalisation matérielle de la technique RIA, nous avons décrit au niveau matériel (RTL) et synthétisé quatre opérateurs :

1) *l'arbitre de PC* est l'arbre de comparaison décrit section 5.1,

2) *l'arbitre mémoire* est l'unité mémoire décrite section 5.2,

3) *l'arbitre combiné* correspond à l'intégration de l'unité d'arbitrage de PC et de l'unité d'arbitrage mémoire au sein d'un même composant, de manière à fournir un point de comparaison direct avec l'unité d'arbitrage combinée,

4) *l'arbitre unifié* est l'arbitre multi-fonction décrit section 5.3.

L'ensemble de ces opérateurs ont été réalisés de manière combinatoire, non pipelinés.

Tableau 5 – Résultats de synthèse des unités d'arbitrage

Arbitre	Optimisé surface		Optimisé latence	
	Surface (NAND2)	Latence (FO4)	Surface (NAND2)	Latence (FO4)
PC	6 045	164	16 067	121
Mémoire	1 308	163	3 560	75
Combiné	6 914	164	16 982	121
Unifié	7 197	164	15 681	121

Les synthèses ont été effectuées à l'aide d'un compilateur RTL de l'industrie ciblant une technologie CMOS de l'état de l'art en 2011. Le tableau 5 résume les résultats de synthèse. Les surfaces sont données en surface équivalente de portes NAND à deux entrées, et les latences en délai équivalent d'inverseurs FO4. Nous présentons les résultats pour deux compromis différents. Nous synthétisons une version optimisée pour la surface en fixant les contraintes de latence à 5 ns. Nous reportons également les résultats de la configuration de latence minimale que nous avons pu obtenir.

La latence des versions optimisées pour la surface, inférieure à 164 équivalents FO4, permet de respecter notre budget de 8 étages de pipeline à 30 FO4 par étage, tout en conservant une marge confortable en prévision du surcoût dû aux registres et au routage des données.

Les résultats des versions optimisées pour la latence montrent qu'une réduction significative du délai est possible, au détriment de la surface, qui augmenterait alors d'un facteur allant de 2,1 à 2,7. Les estimations de consommation de puissance fournies par notre outil de synthèse varient dans les mêmes proportions que ces différences de surface. En accord avec les résultats obtenus section 6.3, nous privilégions la version optimisée pour la surface aux dépens de la latence.

Parmi les unités optimisées pour la latence, notons que l'arbitre unifié présente une surface plus faible que l'arbitre des PC seul. Ce résultat contre-intuitif s'explique par des décisions différentes de l'outil de synthèse de haut niveau dans chaque cas. Il illustre le fait que les différences entre l'arbitre de PC seul et l'arbitre unifié sont négligeables devant les variations inhérentes au processus de synthèse matérielle.

Les résultats de synthèse présentés reflètent la surface et la latence obtenues avant l'étape de placement-routage, et ne prennent donc pas en compte le coût du routage des données depuis chaque PE vers l'arbitre. Le partage des ressources de routage devrait tendre à favoriser la solution de l'arbitre fusionné par rapport à des unités séparées. En effet, la séparation des unités dédouble le réseau de fils nécessaire au routage des PC et des adresses.

L'ensemble des solutions envisagées conserve une surface négligeable dans le contexte des processeurs SIMT. À titre de comparaison, Fung estime le coût en surface de l'unité de formation dynamique de warps à 1,635 mm^2 en 90 nm (Fung *et al.*, 2009). En tenant compte de la technologie employée, cela représente un équivalent de 160 000 portes NAND, soit plus d'un ordre de grandeur de plus que les unités d'arbitrage proposées. Enfin, un processeur SIMT complet de GPU NVIDIA GF100 dans une technologie 40 nm occupe 15,6 mm^2 d'après nos mesures sur une photographie de *die*, ce qui représente l'équivalent de 7 600 000 portes NAND. Le coût en surface de l'arbitrage est donc de l'ordre de 0,1 %.

7. 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 réduit, notamment par la réutilisation des chemins de données existants de l'unité de gestion de la divergence mémoire.

La politique de parcours du graphe de flot contrôle que nous avons considérée dans cet article est celle du SP : PC minimal, qui offre des résultats égaux ou supérieurs à l'état de l'art 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 masquer le coût en latence de l'arbitrage dans les conditions de faible parallélisme de threads. 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 MPC avec le PC individuel en fin de pipeline devient 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 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 (Collange *et al.*, 2009c). Il s'agit notamment de l'approche suivie par le projet d'architecture GPU AMD Graphics Core Next (Demers, 2011).

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 (Fung *et al.*, 2009) et la subdivision dynamique de warps (Meng *et al.*, 2010). Ces techniques complémentaires pourront bénéficier des avancées dans le domaine de la gestion de la reconvergence.

8. Bibliographie

Abel N. E., Budnik P. P., Kuck D. J., Muraoka Y., Northcote R. S., Wilhelmson R. B., ≪ TRANQUIL : a language for an array processing computer ≫, *Proceedings of the spring joint computer conference*, AFIPS '69 (Spring), p. 57-73, 1969.

AMD, *Evergreen Family Instruction Set Architecture : Instructions and Microcode*. December, 2009.

Che S., Boyer M., Meng J., Tarjan D., Sheaffer J. W., Lee S.-H., Skadron K., ≪ Rodinia : A benchmark suite for heterogeneous computing ≫, *IEEE Workload Characterization Symposium*, vol. 0, p. 44-54, 2009.

Collange S., Analyse de l'architecture GPU Tesla, Technical Report n°hal-00443875, HAL-CCSD, Jan, 2010.

Collange S., Daumas M., Defour D., Parello D., ≪ Étude comparée et simulation d'algorithmes de branchements pour le GPGPU ≫, *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009a.

Collange S., Daumas M., Defour D., Parello D., ≪ Barra : a Parallel Functional Simulator for GPGPU ≫, *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, p. 351-360, 2010.

Collange S., Defour D., Tisserand A., ≪ Power Consumption of GPUs from a Software Perspective ≫, *ICCS 2009*, vol. 5544 of *Lecture Notes in Computer Science*, Springer, p. 922-931, 2009b.

Collange S., Defour D., Zhang Y., ≪ Dynamic detection of uniform and affine vectors in GPGPU computations ≫, *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, vol. LNCS 6043, p. 46-55, 2009c.

Collins J. D., Tullsen D. M., Wang H., ≪ Control Flow Optimization Via Dynamic Reconvergence Prediction ≫, *IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, p. 129-140, 2004.

Coon B. W., Lindholm J. E., ≪ System and method for managing divergent threads in a SIMD architecture ≫, , US Patent 7353369, April, 2008.

Coon B. W., Nickolls J. R., Lindholm J. E., Tzvetkov S. D., ≪ Structured programming control flow in a SIMD architecture ≫, , US Patent 7877585, January, 2011.

Demers E., ≪ Evolution of AMD's Graphics Core, and Preview of Graphics Core Next ≫, , AMD Fusion Developer Summit keynote, june, 2011.

Diamos G., Ashbaugh B., Maiyuran S., Kerr A., Wu H., Yalamanchili S., ≪ SIMD Re-Convergence At Thread Frontiers ≫, *MICRO 44 : Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December, 2011.

Fung W., Aamodt T., ≪ Thread block compaction for efficient SIMT control flow ≫, *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, p. 25-36, February, 2011.

Fung W. W. L., Sham I., Yuan G., Aamodt T. M., ≪ Dynamic warp formation : Efficient MIMD control flow on SIMD graphics hardware ≫, *ACM Trans. Archit. Code Optim.*, vol. 6, p. 7 :1-7 :37, July, 2009.

Garland M., Le Grand S., Nickolls J., Anderson J., Hardwick J., Morton S., Phillips E., Zhang Y., Volkov V., ≪ Parallel Computing Experiences with CUDA ≫, *IEEE Micro*, vol. 28, n° 4, p. 13-27, 2008.

Gebhart M., Johnson D. R., Tarjan D., Keckler S. W., Dally W. J., Lindholm E., Skadron K., ≪ Energy-efficient mechanisms for managing thread context in throughput processors ≫, *Proceeding of the 38th annual international symposium on Computer architecture*, p. 235-246, 2011.

Int, *Intel G45 Express Chipset Graphics Controller PRM, Volume Four : Subsystem and Cores*. February, 2009.

Int, *Intel HD Graphics OpenSource PRM Volume 4 Part 2 : Subsystem and Cores – Message Gateway, URB, Video Motion, and ISA*. July, 2010.

Keryell R., Paris N., ≪ Activity Counter : New Optimization for the dynamic scheduling of SIMD Control Flow ≫, *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, p. 184-187, 1993.

Lawrie D. H., Layman T., Baer D., Randal J. M., ≪ Glypnir–a programming language for Illiac IV ≫, *Commun. ACM*, vol. 18, p. 157-164, March, 1975.

Levinthal A., Porter T., ≪ Chap - a SIMD graphics processor ≫, *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, p. 77-82, 1984.

Lindholm J. E., Nickolls J., Oberman S., Montrym J., ≪ NVIDIA Tesla : A Unified Graphics and Computing Architecture ≫, *IEEE Micro*, vol. 28, n° 2, p. 39-55, 2008.

Lorie R. A., Strong H. R., ≪ A SIMD Data Processing System ≫, , European Patent EP0035647, October, 1984.

Mas, *MasPar Programming Language Reference Manual*. July, 1992.

Meng J., Tarjan D., Skadron K., ≪ Dynamic warp subdivision for integrated branch and memory divergence tolerance ≫, *SIGARCH Comput. Archit. News*, vol. 38, n° 3, p. 235-246, 2010.

Mes, *Mesa Gallium3D NVFX graphics driver*. 2011, `http://cgit.freedesktop.org/mesa/mesa/tree/src/gallium/drivers/nvfx`.

Nickolls J., Dally W. J., ≪ The GPU Computing Era ≫, *IEEE Micro*, vol. 30, p. 56-69, March, 2010.

NVI, ≪ NVIDIA CUDA SDK ≫, 2010. `http://www.nvidia.com/cuda/`.

NVIDIA, *NVIDIA CUDA Programming Guide, Version 3.2*. 2010.

Quinn M. J., Hatcher P. J., Jourdenais K. C., ≪ Compiling C* programs for a hypercube multicomputer ≫, *SIGPLAN Not.*, vol. 23, p. 57-65, January, 1988.

Takahashi Y., ≪ A Mechanism for SIMD Execution of SPMD Programs ≫, *Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, HPC-ASIA '97, p. 529-534, 1997.

Zhang F., H. D'Hollander E., ≪ Using Hammock Graphs to Structure Programs ≫, *IEEE Trans. Softw. Eng.*, vol. 30, n° 4, p. 231-245, 2004.