Améliorer le rendement énergétique des GPU par localité des données

Fabrice Mouhartem
avec Sylvain Collange

Résumé

Dans ce rapport est présentée une méthode matérielle permettant d'augmenter l'efficacité des processeurs graphiques dans le cadre des calculs généralistes (une accélération d'exécution de 29% en moyenne), tout en diminuant leur consommation énergétique. Cela est possible grâce à la redondance des données transitant dans les puces graphiques, ce qui est exploitable en amortissant le surcoût de leur reconnaissance par un traitement rapide (latence divisée par 8) et à moindre coût (7% de mémoire supplémentaire).

1 Introduction

Les processeurs modernes intègrent de plus en plus souvent un processeur graphique (GPU) sur la même puce que les cœurs CPU (IGP pour *Integrated Graphic Processor*). Si ce type d'architectures profite aux applications de rendu graphique, il est aussi utilisé pour le calcul généraliste sur GPU (GPGPU). Les processeurs graphiques dont les performances étaient déjà limitées par la consommation énergétique se retrouvent ainsi d'autant plus contraints par ce rapprochement. Le problème de la limitation énergétique des GPU devient donc critique.

L'architecture des GPU diffère de celle des CPU que nous connaissons du fait qu'elle suit le modèle SIMD (*Single Instruction Multiple Data*), ce qui signifie qu'une même instruction est exécutée sur plusieurs données différentes. Ici ce mécanisme est appliqué à des vecteurs de registres, et l'ordonnancement est fait de telle manière que l'instruction est exécutée au compte de plusieurs threads. Un regroupement de threads exécutant le même flot d'instruction est appelé *warp*. Ce modèle d'exécution est ainsi dénommé SIMT (*Single Instruction Multiple Thread*) par NVidia [9]. L'objectif de cette architecture est d'optimiser le débit plutôt que la latence.

Cette structure permet ainsi de réduire la demande sur le cache d'instruction, l'acquisition et le décodage d'instructions en amortissant le coût de traitement d'une instruction sur plusieurs calculs. Nous pouvons donc imaginer différentes manières de généraliser ce modèle. Les données qui transitent sur GPU suivent en effet des schémas très réguliers, il arrive souvent que des données soient affines (c'est-à-dire séparées d'un pas constant, par exemple un bloc d'adresses mémoires contigus), ou uniforme. Nous pouvons alors factoriser non seulement le traitement de l'instruction, mais aussi les calculs eux-mêmes [4]. Ces instructions peuvent être qualifiées de SDMT : *Single Data Multiple Threads*.

De la contrainte énergétique précédemment évoquée découle une consommation limitée des IGP. Cette architecture est donc actuellement utilisée en coopération avec un GPU séparé pour augmenter l'autonomie des batterie [10], tout en offrant la possibilité d'utiliser la puissance de la carte graphique séparée.

Répondre au problème énergétique ainsi soulevé peut se faire de plusieurs manières. Nous pourrions imaginer une solution logicielle, où on marquerait les instructions scalaires, et vectorielles, par exemple par le biais d'un profilage pour détecter quelles données sont affines. Mais ce système est limité par la nature dynamique des données. En effet, on ne peut pas savoir à l'avance quelles en seront la forme, par exemple dans le cadre de valeurs rentrées par l'utilisateur. Nous nous tournerons donc vers des solutions matérielles qui offrent plus de flexibilité à cet égard.

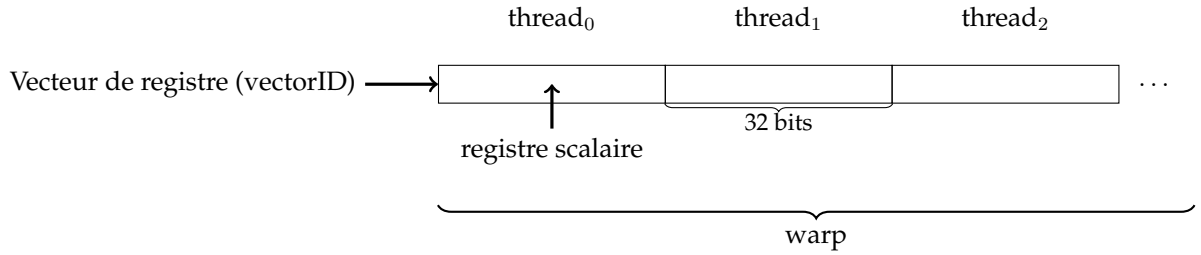


FIGURE 1 – Structure d'un vecteur de registres sous CUDA

Au cours de ce stage nous chercherons donc à améliorer les performances des architectures SIMT. Pour cela nous allons essayer de généraliser l'approche SDMT pour des données similaires entre threads, mais qui ne sont pas nécessairement identiques, un exemple étant les vecteurs affines, mais ce n'est pas la seule similarité existante. Nous allons donc dans un premier temps utiliser un simulateur afin d'étudier les redondances exploitables dans les données que nous aurons à manipuler, à l'issue de quoi nous proposerons une solution en fonction de notre analyse. Enfin nous implémenterons cette solution dans le simulateur afin de pouvoir étudier les résultats obtenus.

2 Étude de similarités sur les registres

Avant de proposer une solution, nous devons étudier la corrélation entre les données de différents threads, afin de déterminer une solution efficace à proposer. Pour cela nous utiliserons le simulateur Barra [3], qui émule la librairie CUDA et un GPU NVidia, que j'ai modifié afin de récupérer les informations qui nous intéressent. Nous allons donc commencer par présenter ce sur quoi nous allons travailler : les registres sous CUDA, puis nous verrons comment le simulateur fonctionne et la manière dont les données sont récoltées avant de présenter nos résultats.

2.1 Les registres dans CUDA

CUDA (*Computer Unified Device Architecture*) propose une plateforme permettant le calcul sur carte graphique en mettant à disposition une interface de programmation.

Le programmeur dispose dans cette interface de la possibilité de spécifier différentes granularités de parallélisme : par grille, par bloc et par *warp*. Une grille est composée de plusieurs blocs qui sont eux même composés de plusieurs warps [8].

L'unité qui nous intéresse ici est le warp, puisqu'il correspond matériellement à un vecteur de registre. Comme nous l'avons dit en introduction, un warp correspond à l'exécution d'un flot d'instructions identiques sur des données différentes. Notre but sera donc d'analyser les relations entre les registres utilisés dans un même warp.

Le vecteur de registre est identifié par son vectorID, et chacun de ses registres scalaire possède un identifiant qui peut être utilisé dans les calculs, par exemple si un thread est utilisé pour traiter une case d'une matrice, alors le threadId peut être utilisé pour en déterminer les coordonnées.

Un vecteur de registre est subdivisé en plusieurs *lanes*, qui correspondent matériellement à un registre. Sur ces lanes s'exécutent un thread.

Les vecteurs de registres suivent alors la structure décrite en figure 1, qui en indique la structure, mais aussi les différents marqueurs (ou tags) que l'on retrouve pour identifier les différentes composantes d'un registre.

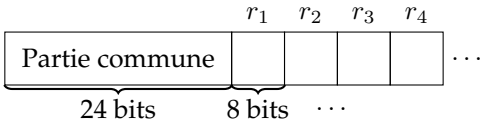


FIGURE 2 – Structure d'un vecteur de registres SDMT

2.2 Barra

Barra simule un environnement CUDA sur une architecture Fermi. Pour cela il génère une librairie dynamique qui prend la place de celle de CUDA. Ce simulateur repose sur l'environnement unisim [1], qui propose un squelette modulaire en C++ pour la construction de tels simulateurs.

Barra dispose de deux composantes : un simulateur fonctionnel (celui qui nous intéresse dans cette section), et un simulateur matériel qui sera utilisé plus tard pour pouvoir évaluer au niveau cycle les performances de notre proposition d'architecture.

Dans sa partie fonctionnelle, Barra ne simule la micro architecture même, mais le jeu d'instruction de CUDA. Nous avons donc accès à des informations sur la forme des données transitant au travers de notre simulateur, par exemple s'il s'agit d'une donnée entière ou un nombre en virgule flottante. C'est ainsi que nous pouvons obtenir des informations sur la répartition des données, information que ne peut pas nous fournir un simulateur matériel puisqu'il manipule directement les signaux au travers d'unités fonctionnelles.

Nous utiliserons comme jeu de tests pour obtenir nos données les exemples de la SDK (*Software Development Kit*) de Cuda, bien qu'ils n'y soient pas destinés à l'origine, mais en tant qu'exemples, ils représentent les bonnes pratiques à adopter pour un programmeur CUDA. Ainsi que le la batterie de tests Rodinia [2] qui exploite le parallélisme des données, ce qui permet de profiter les améliorations qu'on peut attendre de la compression des opérations sur les données.

2.3 La collecte des données

Le simulateur est composé d'un module statistique comportant différents compteurs qui capturent les données de chaque instruction statique du programme au fur et à mesure de son exécution. Ces données sont ensuite écrites dans un fichier au format CSV (*comma separated value*), qui est ensuite post-traité par un script afin d'en extraire les données qui nous intéressent.

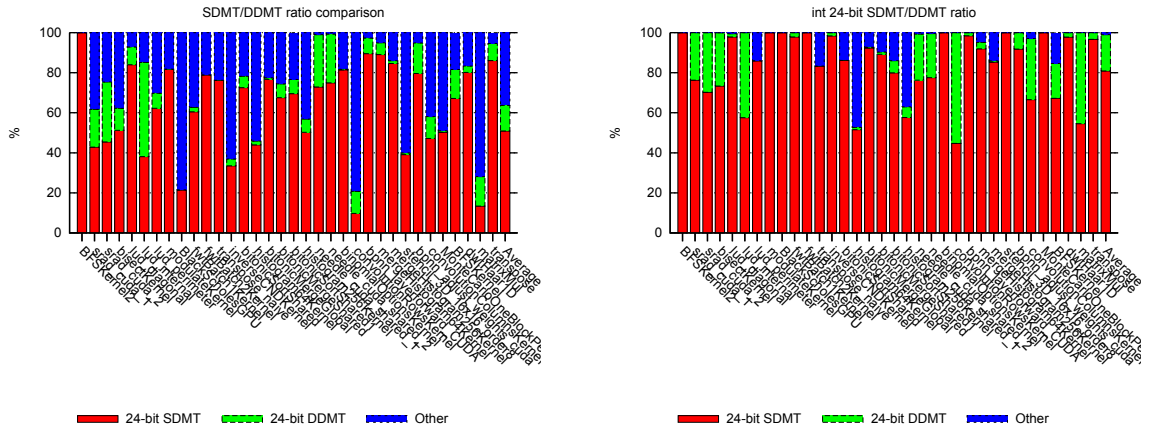
Celles-ci utiliseront souvent le concept de similarité qui se définit comme suit :

$$V \sim_n V' \iff \text{les } n \text{ bits de poids forts de } V \text{ et } V' \text{ sont égaux}$$

Comme les registres font chacun 32 bits, nous utiliserons la similarité sur 24 bits, ce choix se justifie par le fait qu'une grande majorité des données est similaire sur 24 bits. Ce qui offre 8 bits de modularité. Ainsi si l'on souhaite compresser les registres comme décrit dans la figure 2, on économise 72% de l'espace total.

Différentes valeurs nous intéressent ici :

1. Le nombre de vecteurs similaires entre deux instructions successives d'une instruction donnée pour un thread donné. Nous pourrons ainsi voir si un résultat calculé peut être factorisé dans le temps comme proposé en introduction.
2. Le nombre de vecteurs de registres dont les composantes entre threads sont similaires pour une exécution donnée d'une instruction afin de pouvoir réaliser la compression précédemment évoquée. Et ainsi assouplir le modèle SDMT en l'appliquant non pas à un vecteur de registres, mais à la partie commune des registres.



(a) Cas général

(b) Cas des entiers

FIGURE 3 – Répartition des vecteurs 24-bit DDMT/SDMT

3. Les vecteurs qui sont composés de deux classes d'équivalences pour la relation de similarité, aussi nommés DDMT, pour généraliser le modèle *Dual Data Multiple Threads*. Ici on gardera ainsi en mémoire non pas un, mais deux préfixes communs ainsi qu'un masque indiquant pour chaque registre quel est son préfixe.
4. Pour chacune de ces quantités, connaître la proportion de vecteurs affine (c'est à dire les vecteurs de la forme $b + s \times i$) et uniforme, dont les calculs peuvent être compressés en opérant sur leur base et leur pas au lieu d'agir sur le vecteur complet.

Nous remarquerons enfin que l'énergie consommée ne fait pas partie de la couche fonctionnelle. Ainsi nous ne la prendrons pas en compte ici.

2.4 Nos résultats

Nous notons dans un premier temps que les vecteurs DDMT ne représentent que 50% du nombre de vecteurs représentés au cours des tests (cf. Figure 3a). Mais ce résultat ne prend pas en compte la nature des données (nombre entier ou à virgule flottante). En effet nous voyons que pour les entiers, ils représentent 97% de l'ensemble des vecteurs.

Ces régularités dans les structures peuvent avoir plusieurs origines. Elles peuvent provenir de calculs d'adresses mémoires, qui peuvent être dans des zones proches, par exemple dans le cas de l'espace mémoire alloué à un tableau. Ou alors de threads qui agissent sur des données similaires à l'origine. Si le flot de calculs ne subit que peu de divergences, alors le résultat final sera lui aussi composé de registres proches. Enfin ces opérations peuvent être issues de calculs indexés par les identifiants (ThreadID par exemple), qui représentent souvent des valeurs proches.

Les flottants disposent en revanche de mauvaises propriétés de similarité dans leur représentation binaire. En effet les nombres à virgule flottantes simple-précision décrits par la norme IEEE 754 disposent d'un bit de signe, 8 bits d'exposants, et 23 bits de mantisse. Ce qui fait que les 8 bits de poids faibles représentent les faibles variations de la valeur par rapport à son ordre de grandeur ($2 \cdot 10^{-5}$ par rapport à l'exposant pour le 8^{e} bit de poids faible, et $1 \cdot 10^{-7}$ pour premier bit de poids faible). Ainsi les vecteurs composés de nombre à virgules flottantes sont à 24% similaire dans le temps et ne sont composés qu'à 32% de vecteurs DDMT.

Nous en déduisons alors que nos solutions profiterons aux calculs riches en opérations entières. Ce qui est souvent le cas en raisons des calculs structurels, comme déterminer l'adresse de l'élément à la 13^{e} ligne et 2^{e} colonne d'une matrice, qui n'agissent non pas sur les données (qui n'ont souvent pas de structures particulières) mais sur leur organisation en mémoire qui suit des motifs plus réguliers.

De plus une grande majorité des vecteurs DDMT sont affines (96%, comme le montre la figure 4). Cela peut s'expliquer par le fait que les opérations que nous avons décrites comme pouvant être à l'origine

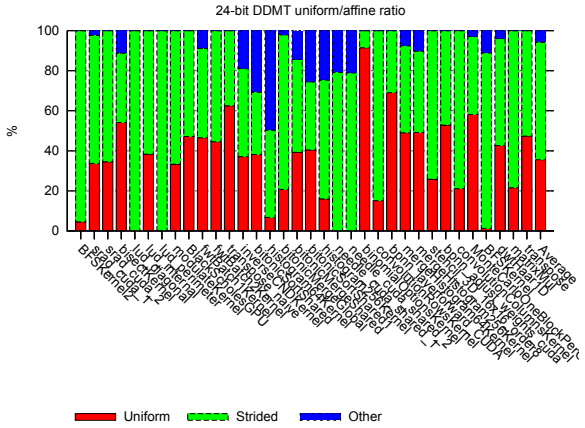


FIGURE 4 – Proportion de vecteurs DDMT affines/uniforme

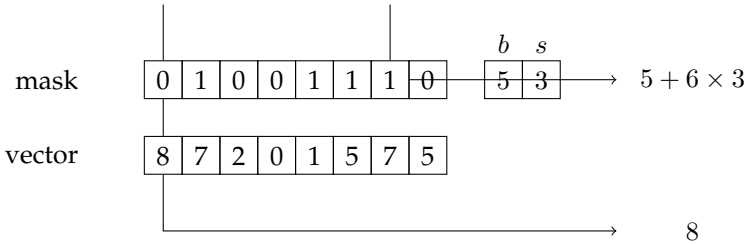


FIGURE 5 – Fonctionnement d'un vecteur de registre affine

des vecteurs similaires sont le plus souvent des opérations affines (lecture d'un tableau en mémoire par exemple, ou alors utilisation des threadIds d'un vecteur de registre). Proposer une solution qui cherchera à optimiser le traitement des vecteurs DDMT profitera donc moins qu'une solution visant à améliorer le traitement des vecteurs affines. C'est pourquoi nous proposons une solution fondée sur la reconnaissance et le traitement des vecteurs affines dans le but d'augmenter le débit et de diminuer la latence des opérations.

3 Vecteur de registres affine

La solution proposée repose sur l'ajout d'une nouvelle structure : le banc de registres affine, qui vient se greffer au banc de registre tels que nous le connaissons. Nous allons donc dans cette partie présenter la solution, puis nous effectuerons une première approximation optimiste des résultats espérés afin d'estimer l'efficacité de notre méthode de détection dynamique avant de l'implémenter dans le simulateur.

3.1 Présentation

L'idée du vecteur de registre affine est un prolongement de celle de la détection de registres affines [4]. Cette méthode rajoutait un drapeau supplémentaire au banc de registre afin de spécifier si un vecteur de registre était affine ou non. Mais les divergences (c'est-à-dire les cas où seule une partie du vecteur était utilisée pour effecteur les calculs, par exemple si on n'a besoin d'exécuter que des 8 premiers threads du warp) limitaient les performances de cette méthode en faisant perdre l'information sur l'affinité du vecteur. Pour contourner ce problème, nous allons augmenter la finesse de la détection en rajoutant les composantes suivantes :

1. Un masque affine indiquant si le registre demandé est affine ou non. (32 bits : 1 bit par composante)

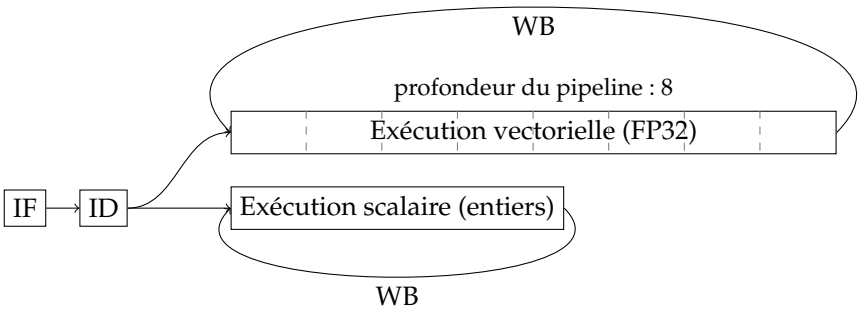


FIGURE 6 – Notre proposition de pipeline d'exécution modifié.

2. La base et le pas du vecteur affine. (32 + 8 bits, car on a vu que 97% des registres entiers étaient DDMT, ce qui signifie que la somme des pas le long du vecteur a peu de chance (3%) de dépasser 8 bits, cela est donc suffisant.)

On perd ainsi la rigidité de l'ancienne méthode qui spécifiait entièrement le vecteur, puisque désormais un vecteur de registre peut être à la fois affine et générique.

Le surcoût engendré par ce marquage est de 76 bits, soit une augmentation d'environ 7% de la taille totale des vecteurs de registre.

Nous pourrons remarquer que la structure de notre solution est similaire à celle du cache affine [5]. D'où une méthode de lecture analogue : le masque affine joue le rôle de filtre pour savoir si on lit ou non la composante générique du vecteur ou alors si on décode sa composante affine comme illustré en figure 5.

La détection dynamique se fait suivant les deux algorithmes 1 et 2 exécutés lors d'une lecture ou une écriture affine et utilisant une nouvelle opération appelée "flatten". Lors d'un appel à cet opérateur, les valeurs génériques d'un vecteur affine sont calculées et écrites dans ce même registre, et le masque affine est mis à 0, ce qui est une opération facile à calculer.

Algorithme 1 : Procédure de lecture

Entrée : $R_i = (v_i, b_i, s_i)$ où v_i est le masque affine, b_i la base et s_i le pas, avec le masque d'exécution m
si $\forall i, m \subseteq v_i$ **alors**
  effectuer une opération affine
sinon
  flatten sur R suivant le masque d'exécution
  effectuer une opération générique

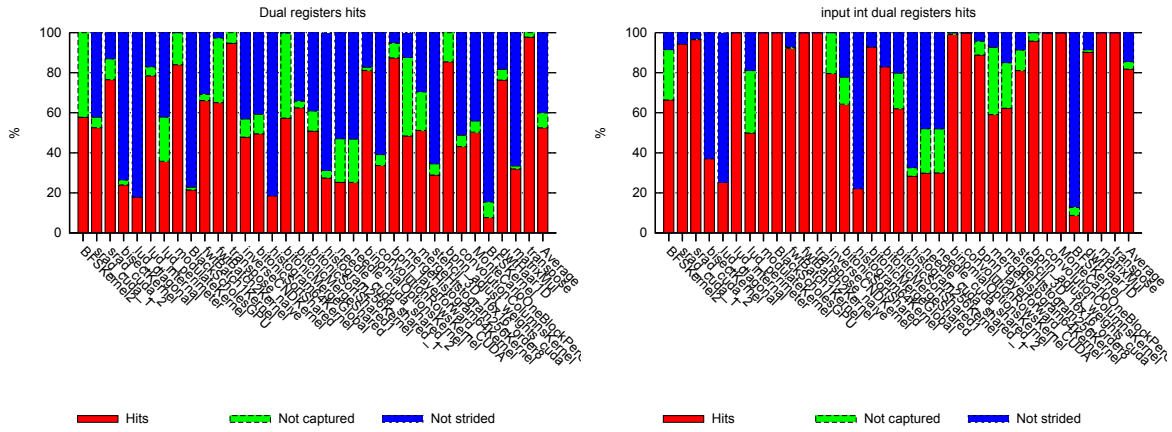
Algorithme 2 : Procédure d'écriture

Entrée : $R = (v, b, s)$ où v est le masque affine, b la base et s le pas, avec le masque d'exécution m
si $v \subseteq m$ **alors**
  effectuer une opération affine
sinon
  flatten sur R suivant le masque d'exécution
  effectuer une opération générique

Le but de cette nouvelle opération est de revenir à un vecteur générique si jamais une incompatibilité entre le masque affine et le masque d'exécution est détectée. Cela permet d'éviter un coût de détection de registres affines qui serait difficile à mettre en œuvre matériellement.

Comme le suggèrent les algorithmes, les incompatibilités de masques sont de deux types :

1. Les incompatibilités en lecture, qui apparaissent lorsque le masque d'exécution n'est pas inclus (dans le sens de l'inclusion booléenne : $a \subseteq b$ si tous les bits à 1 dans a sont à 1 dans b) dans un des masques affines des vecteurs lus. Dans ce cas l'opération ne peut être entièrement scalaire, et nous sommes obligés de perdre l'information affine des vecteurs responsables pour les traiter.
2. Les incompatibilités en écriture, dans le cas où le masque affine du vecteur d'arrivée n'est pas inclus dans le masque d'exécution. On a alors une information erronée (les bits à 1 supplémentaires dans le masque affine d'arrivée) dans le masque affine si on se contente d'écrire le résultat de notre opération affine. Nous devons donc obtenir les composantes du vecteur avant d'y écrire notre résultat.



(a) Cas général (b) Cas entier

FIGURE 7 – Quantité de registres affines capturés

Pour la mise en place matérielle, j'ai donc rajouté au simulateur une classe "Tag" qui représente ces données supplémentaires, ainsi que les règles de mise à jour dynamique des masques. J'ai ensuite rajouté des compteurs afin d'observer le nombre de registres affines ainsi capturés.

L'objectif étant d'effectuer les opérations sur les composantes affines des registres lorsque cela est possible afin de n'avoir qu'une seule opération scalaire au lieu d'une opération vectorielle qui doit agir sur tout le registre de vecteurs. Nous libérons donc les unités vectorielles pour d'autres calculs qui ne pourraient être ainsi simplifiés, tout en effectuant une opération moins coûteuse en temps et en énergie. L'exécution se déroule suivant le pipeline présenté en figure 6.

3.2 Évaluation optimiste

Nous allons chercher à obtenir une majoration du nombre d'opérations affines que cette méthode sera en mesure de reconnaître. Pour cela j'ai donc implémenté une version fonctionnelle de ces règles. Nous effectuons pour ce faire un test à chaque écriture afin de vérifier si le registre écrit est toujours affine ou s'il est devenu générique suite à l'opération afin de maintenir le masque affine à jour. Les règles sur la lecture restant identiques à celles précisées plus haut.

Les mesures sur cette version du simulateur montrent qu'une grande partie (87%, d'après la figure 7a) des vecteurs affine sont capturés, ce qui est plus efficace que les méthodes actuelles [4]. Cette augmentation de la finesse lors de la détection est donc bénéfique pour la détection de registres affines. Ce qui est encore plus significatif sur les données entières, comme nous pouvons le remarquer dans la figure 7b, où 95% des données affines sont capturées.

Notre méthode de détection dynamique des données fondée sur un système de résolution de conflits se montre donc potentiellement efficace. Nous pouvons alors passer à la simulation matérielle, toujours avec Barra, pour quantifier le gain au niveau cycle que nous pouvons espérer obtenir.

4 Unité Scalaire

Nous proposons ainsi l'ajout d'une unité chargée d'effectuer les calculs anciennement destinés aux unités vectorielles opérant sur les entiers. Nous présenterons dans un premier temps les spécificités de cette unité fonctionnelle avant de présenter les résultats obtenus.

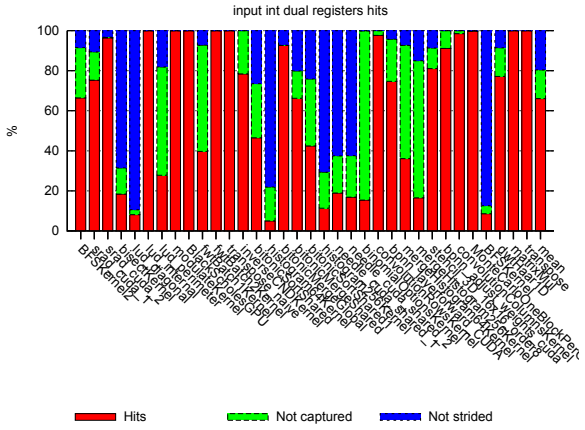


FIGURE 8 – Quantités de registres capturés dans le cas entier

4.1 Fonctionnement

Nous rajoutons donc aux unités fonctionnelles déjà présentes une unité scalaire chargée d'effectuer les calculs sur les composantes affines des vecteurs de registre. Cela libèrera donc les ressources vectorielles pour d'autres calculs, qui pourront donc y être ordonnancés dans le même temps. Cette partie est gérée par la composante matérielle de Barra qui simule l'ordonnanceur présent dans l'architecture Fermi [3].

Une autre différence se situe au niveau de la latence des opérations, en effet les unités scalaires effectuent les opérations en un cycle d'horloge, contrairement aux unités vectorielles qui sont pipelinés avec 8 étages suivant le modèle SIMD horizontal. Cela signifie qu'une instruction est transmise au rythme d'une unité par cycle, ce qui assure un débit semblable à celui des unités scalaires.

Une autre solution possible aurait été de réserver une lane des vecteurs de registre pour effectuer les opérations scalaires, afin de ne pas rajouter une unité matérielle, mais cela ne diminue pas la latence. De plus si on utilise les bus déjà présents pour brancher notre unité scalaire, le surcoût matériel est négligeable. Une variation de cette méthode aurait été d'effectuer les calculs affines par groupes ce qui permettrait d'amortir en bande passante le surcoût en latence, mais cela complique la tâche de l'ordonnanceur qui devra choisir le bon moment pour exécuter les opérations affines.

4.2 Le simulateur matériel

Barra propose aussi un simulateur matériel de l'architecture Fermi, pour cela les opérations du pipeline d'exécution sont simulées au niveau du cycle d'horloge, ce qui nous permet d'évaluer les performances de notre proposition d'architecture.

J'ai donc implémenté l'opération *flatten*, appelée suivant les règles évoquées plus haut (cf. algorithmes 1 et 2). J'ai de plus rajouté sur les instructions un drapeau signifiant "scalaire" s'il est levé ou "vectoriel" sinon. Nous pouvons ainsi obtenir des mesures simulées différentes suivant le type d'opération.

La détection se fait suivant deux critères, le premier est le type de l'opération, il faut qu'elle soit entière, puis sur la forme des données, ce qui se fait à la lecture : si jamais toutes les opérandes en entrée sont affines, alors l'opération est affine.

Il en résulte donc un pipeline modifié similaire à celui de la figure 6 : à la fin du décodage de l'instruction, on vérifie s'il n'y a pas de conflit en lecture ou en écriture suivant le masque, et si c'est le cas, on remplace l'instruction par un flatten avant de réexécuter l'instruction, qui est donc marquée comme générique si cela a été fait. Dans le cas contraire une instruction affine est exécutée. De plus si un vecteur affine est utilisé dans une instruction générique, on perd cette information en raison d'un flatten, nous reviendrons sur ce cas plus tard.

Cette implémentation permet d'obtenir les résultats donnés dans la figure 8. Nous voyons que ces derniers sont légèrement moins bons : nous réussissons à capturer 82% des données entières observées,

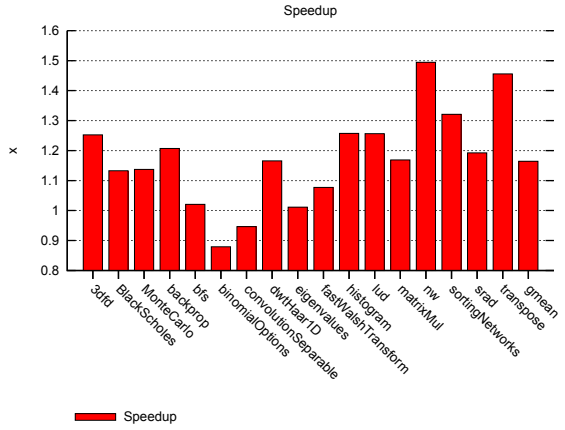


FIGURE 9 – Accélération

l'évaluation n'était donc pas trop optimiste dans ses hypothèses. Ce qui signifie que les flattens issue d'opérations génériques mettant en jeu des opérandes scalaires (ce qui est l'opération qui n'est pas prise ne compte par notre évaluation optimise) ne sont pas en nombre trop importants.

La figure 9 montre de plus une accélération moyenne de 16% par rapport à la version actuelle du simulateur. Ce qui est une amélioration significative, pour un surcout de 7% sur la taille du vecteur de registres, et l'ajout d'une unité de calculs scalaire [1], avec un maximum à 49% et un minimum à -12% pour `binomialOptions`. Les ralentissements s'expliquent par le faible taux de registres affines capturés dans ces cas (15% dans le cas de `binomialOptionsKernel`, alors que son évaluation optimiste en estimait 99%, ce qui indique que les opérations génériques agissant sur des registres affines sont la cause de ces ralentissements, nous y reviendrons dans la section 5.1), ce qui se traduit par un nombre important de flattens qui crée un surcoût qui n'est pas amorti par les calculs affines puisqu'ils ne sont pas effectués.

Nous pouvons de plus recouper deux informations que le simulateur matériel peut nous apporter et dont j'ai implémenté les compteurs : le taux d'occupation des unités fonctionnelles et leur temps respectifs d'utilisation.

Nous remarquons alors que malgré la forte occupation de ces unités, leur temps d'utilisation reste faible comparé à l'ensemble du temps passé pour le calcul, comme le montre la figure 10. En effet pour une utilisation moyenne de 80%, les unités scalaires ne sont utilisés que pendant 33% de la durée totale du calcul. Ce qui explique les accélérations que nous avons notés, et peut se traduire par une économie énergétique.

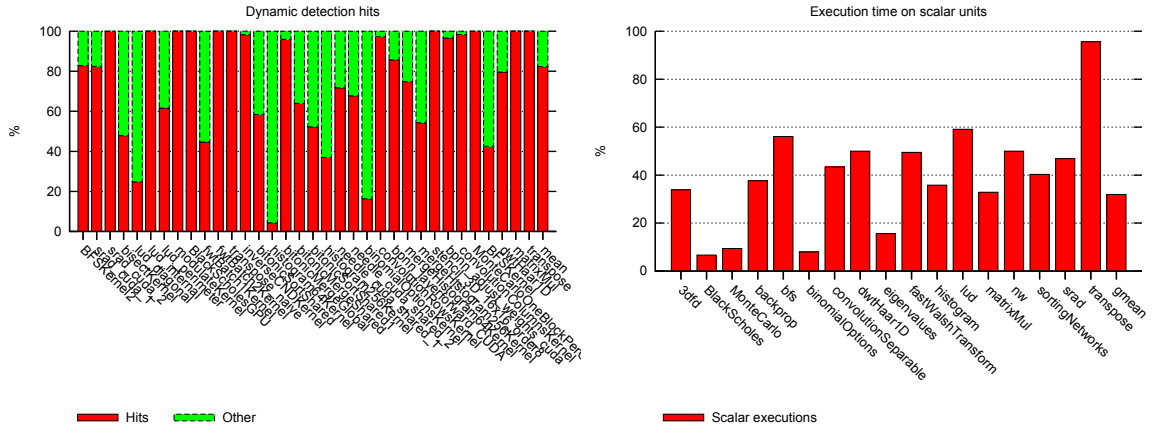
5 Améliorations de l'architecture

Nous avons donc pu observer une amélioration notable des performances du GPU sur notre jeu de tests suite à l'ajout des vecteurs de registres affines. Nous allons donc désormais travailler sur des méthodes pour améliorer la technique. Pour cela nous verrons dans un premier temps le principe du SIMT vertical [7] et comment l'utiliser pour éviter des flattens, et ensuite nous y brancherons un cache affine [5]

5.1 Contourner les flattens en lecture

Nous avons vu que les flattens issus de la présence d'opérandes affines dans une opération génériques étaient la cause des ralentissements que nous avons pu observer. De plus les résultats présentés en fi-

1. On pourrait imaginer dans le cas des IGPs faire les calculs sur le cœur CPU, mais aujourd'hui le déplacement des données se fait via le cache L3 voire de manière externe, ce qui introduit une latence importante.



(a) Nombre d'opérations effectuées sur l'unité scalaire

(b) Temps d'occupation de l'unité scalaire

FIGURE 10 – Utilisation des unités scalaires

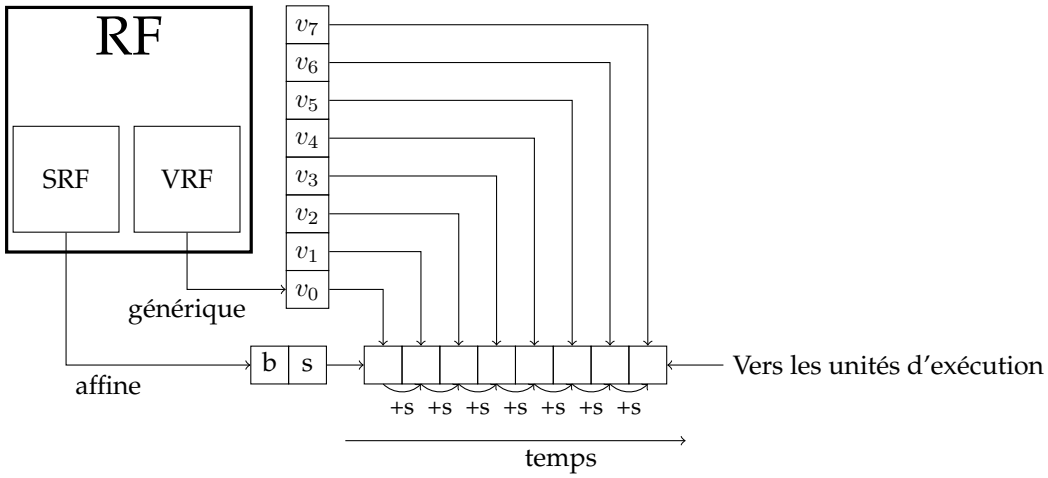
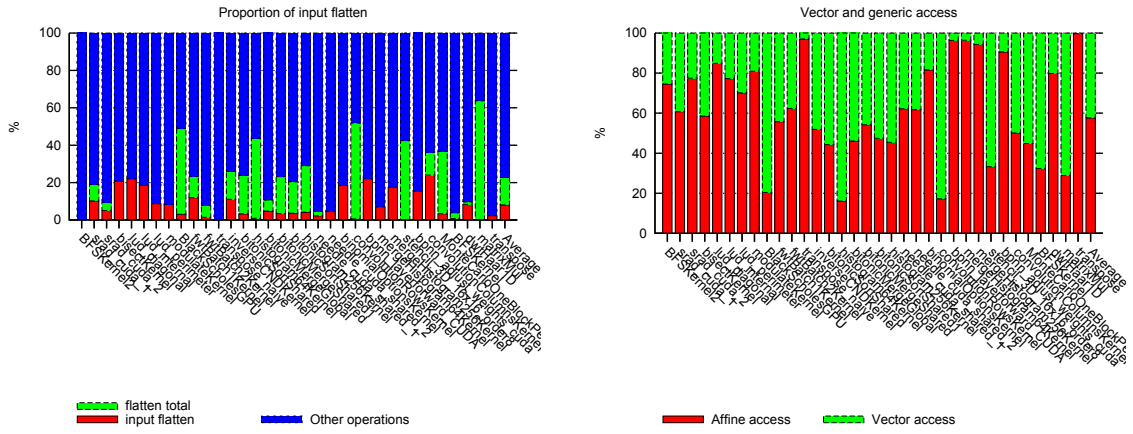


FIGURE 11 – Structure d'une architecture SIMD verticale



(a) Proportion de flatten dues aux opérations génériques, ce qui traduit le nombre de flatten que nous pouvons éviter.

(b) Taux en lecture de vecteurs purement affine, ce qui s'interprète comme le nombre de lectures de vecteurs économisables.

FIGURE 12 – Résultats expliquant le choix du SIMT vertical

gure 12a montrent qu'ils représentent en moyenne 8% des opérations, ce qui représente 35% du nombre total de flattens exécutés. Ces flattens se déroulant en lecture, il pourraient donc être évitées. C'est ce que nous allons essayer de faire.

Nous avons donc indiqué précédemment que les GPU actuels et en particulier l'architecture Fermi que nous simulons suit un déroulement SIMD horizontal, ce qui signifie que le vecteur d'instruction est envoyé dans le pipeline au rythme d'un élément par cycle d'horloge, ce qui permet d'optimiser la bande passante, la latence se retrouvant dans les unités d'exécution.

Mais nous pouvons aussi imaginer une architecture suivant un modèle vertical, comme suggéré par Krashinsky [7] dans le cadre des registres uniforme. Il propose ainsi d'amortir la lecture des données redondantes en profitant du pipeline de lecture des données. Ainsi il propose de lire la donnée redondante en boucle là où les données génériques sont lues classiquement, ce qui permet d'éviter le déplacement en mémoire d'un vecteur entier de données identiques. Il en résulte une diminution de la bande passante pour une latence identique.

Mais cette méthode a pour avantage de permettre le traitement des opérations affines à la volée, suite à la modification que nous proposons : nous lisons la base du vecteur avant de propager le pas à chaque registre lu. En effet comme l'écriture se fait registre à registre dans le pipeline, on peut profiter de cette opération pour effectuer une opération générique à partir d'un registre affine et générique sans avoir besoin d'effectuer un flatten, comme illustré en figure 11.

Ainsi nos registres deviennent à la fois affine et générique, et ce même au cours des opérations, puisqu'on ne perd pas l'information à la lecture comme c'était le cas avant.

Comme la lecture d'un vecteur complet (1024 bits) est une opération plus complexe que la lecture du masque, de la base et du pas (76 bits), on va séparer les registres affines et vectoriels. On change ainsi le procédé de lecture de registres pour le suivant :

Algorithme 3 : Déroulement d'une opération lisant dans les registres

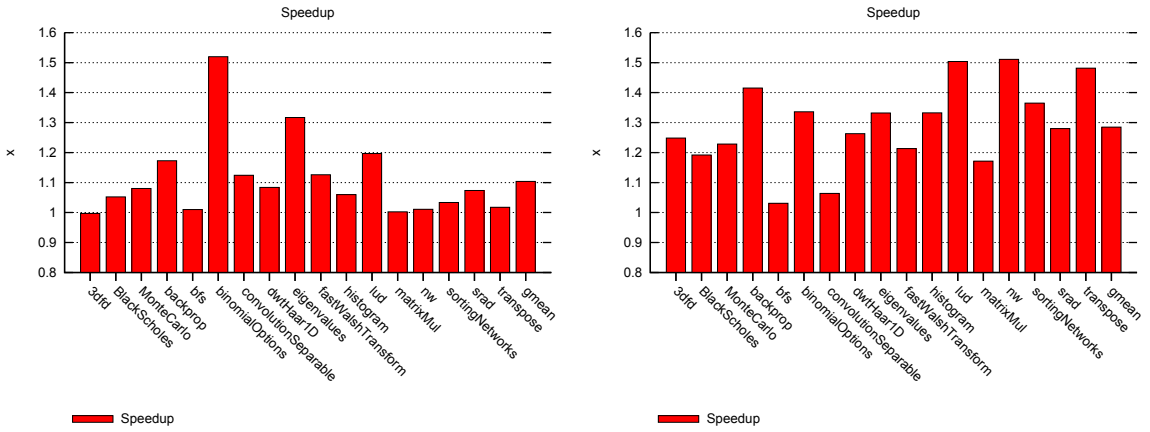
Entrée : Le masque m d'exécution
Les vecteurs en entrée R_i
Lire les masques des R_i
si *on doit lire la partie affine pour* R_i **alors**
    L'opération est générique
    Demander au registre vectoriel la partie vectorielle de R_i
    Effectuer suivant un SIMT vertical
sinon
    L'opération est affine et est exécutée

Nous économisons ainsi le coût de la lecture du registre entier dans le cas des opérandes scalaires. Le taux de registres ainsi économisés est de 57% comme indiqué dans la figure 12b. Ce qui est signe d'une possible amélioration énergétique.

Nous remarquons de plus une augmentation de la vitesse de 10% par rapport à notre première proposition d'architecture, ce qui se traduit par une accélération de 29% en moyenne par rapport à la version de base. En outre nous en déduisons que cette amélioration permet d'amortir le coût des flattens puisque dans notre jeu de tests, il n'y a plus de programmes subissant une décélération, comme le montre la figure 13. Ce qui est confirmé par le fait que `binomialOptions`, qui était précédemment le plus ralenti à cause des flattens en début de programme, devient le plus accéléré en ayant une amélioration de vitesse relative à notre architecture de 52%.

5.2 Travaux futurs : le Cache affine

Nous pourrions aussi chercher un moyen de profiter de travaux déjà existants. Je m'intéresse donc au cache affine [5] qui dispose des structures de nos vecteurs affines.



(a) Gain par rapport à notre architecture

(b) Gain par rapport à la version de base

FIGURE 13 – Accélération apportée par le SIMT vertical

En effet le cache affine est une proposition d'architecture qui pourrait se joindre au cache déjà présent afin de compresser l'espace en mémoire les vecteurs affines qui comme nous l'avons vu représentent une part non négligeable des données manipulées par les GPU dans le cadre des applications GPGPU. Cela se fait par le biais d'une mémoire qui retient les composantes affines ainsi que le masque des vecteurs à placer en mémoire. Deux unités d'encodage et de décodage sont ensuite rajoutées pour permettre l'utilisation de cette mémoire par les moyens habituels par le biais de traductions. Il a été montré que cette structure permettait d'augmenter l'efficacité en mémoire (mémoire réelle sur quantité de matériel utilisé) ainsi que de diminuer la consommation énergétique en permettant aux registres affines d'être lus en une seule fois, ce qui signifie qu'on évite l'énergie dissipée par le transfert de 32 registres.

La similarité structurelle nous permet ainsi d'éviter les opération d'encodage et de décodage en écrivant dans le cache le contenu du tag, et en lisant le tag directement de la même manière, ce qui permet d'économiser l'ajout de telles unités pour l'utilisation des vecteurs de registres ainsi stockés.

Nous pouvons donc espérer obtenir des avantages du cache affine en plus de ceux de notre structure de donnée, et ainsi profiter de la synergie des deux structures.

6 Conclusion

Nous avons donc profité de la redondances des données présente dans le modèle d'exécution SIMT afin de proposer une architecture qui offre un gain autant en performance qu'en consommation énergétique pour un faible surcoût matériel.

Au cours de ce stage, un papier traitant de l'utilisation du "flatten" suivant un modèle pessimiste (les divergences ne sont pas traitées, à partir de ce moment un flatten est effectué) a néanmoins été publié [6], et les gains en performance de notre architecture comparés à celle-ci sont négligeables (0.5% sur un test, aucun sur les autres).

Nous avons alors proposé une amélioration de notre architecture en modifiant la manière dont les données sont traitées en exploitant la structure du SIMD vertical, nous obtenons un gain de vitesse ainsi que des économies d'énergie supplémentaires.

Une autre voie de recherche possible aurait été de se pencher sur les registres SDMT et DDMT pour le traitement des données redondantes en n'exécutant le calcul que sur les parties mobiles et associer ces méthodes de compression d'opérations à un cache compressés suivant la méthode base + Δ [11].

Ce stage a donc été l'occasion de proposer une architecture et de valider son intérêt dans le cadre du problème de la limitation énergétique des GPU.

Il m'a de plus permis de découvrir le monde de la recherche informatique dans l'équipe ALF dont les domaines de recherches s'étendent de la microarchitecture à la compilation.

Références

[1] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. Unisim : An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters*, 6(2) :45–48, 2007.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia : A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

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

[4] S. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Euro-Par 2009–Parallel Processing Workshops*, pages 46–55. Springer, 2010.

[5] S. Collange, A. Kouyoumdjian, et al. Affine vector cache for memory bandwidth savings. 2011.

[6] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten. Microarchitectural mechanisms to exploit value structure in SIMT architectures. 2013.

[7] R. M. Krashinsky. Temporal SIMT execution optimization. `http://www.freepatentsonline.com/y2013/0042090.html`, February 2013.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla : A unified graphics and computing architecture. *Micro, IEEE*, 28(2) :39–55, 2008.

[9] J. Nickolls and W. J. Dally. The GPU computing era. *Micro, IEEE*, 30(2) :56–69, 2010.

[10] Nvidia. Nvidia optimus technology. `http://www.nvidia.com/object/optimus_technology.html`.

[11] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression : practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388. ACM, 2012.