Programmation multi-accélérateurs unifiée en OpenCL

Henry Sylvain

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

Programmation multi-accélérateurs unifiée en OpenCL

Sylvain HENRY - sylvain.henry@labri.fr

Équipe RUNTIME - INRIA Bordeaux Sud-Ouest - LaBRI - Université de Bordeaux
351, cours de la Libération, 33405 Talence, France

Résumé
Le standard OpenCL propose une interface de programmation adaptable à différents types d'accélérateurs (GPU, CPU, CELL…). Pour chaque architecture, il revient aux applications d'effectuer explicitement les partitionnements et les transferts de données ainsi que les placements des tâches sur les accélérateurs disponibles, ce qui est très difficile. Néanmoins, nous montrons que le standard OpenCL peut également être utilisé avec une implémentation qui masque les différents accélérateurs aux applications et ne lui en présente qu'un seul (virtuel). Les transferts de données et les placements des tâches sont alors réalisés par l'implémentation. Nous montrons que ce modèle de programmation permet d'exploiter efficacement et de façon unifiée des architectures hétérogènes.

Mots-clés : OpenCL, GPU, Support exécutif, Équilibrage de charge, Portabilité

1. Introduction

Depuis quelques années, les architectures des super-calculateurs deviennent de plus en plus hétérogènes. Outre ceux composés de nœuds ayant eux-mêmes une architecture hétérogène (par exemple RoadRunner composé de processeurs CELL), il est devenu commun de trouver des accélérateurs en soutien des processeurs classiques. L'exemple le plus connu reste celui des cartes graphiques utilisées à des fins non nécessairement liées au graphisme (GPGPU).
Jusque récemment, programmer ces accélérateurs requérait l'usage de technologies propres à chaque marque. En particulier, Nvidia proposait un SDK pour ses cartes (CUDA), AMD en proposait un pour les siennes (ATI Stream) et IBM un autre pour le CELL. La situation s'améliore cependant en 2008, lorsque Apple soumet au groupement d'industriels *Khronos* une proposition d'interface commune pour les accélérateurs : OpenCL. Ce dernier devient alors un standard et différentes implémentations sont successivement mises à disposition par les différents fabricants [3, 1, 2].

Le standard OpenCL se divise en deux parties. La première définit un langage, proche du C, qui peut être compilé par chaque implémentation pour être exécuté sur différents accélérateurs. Les codes écrits avec ce langage sont appelés *kernels*. La seconde partie décrit une API pouvant être utilisée par des programmes classiques pour allouer de la mémoire sur les accélérateurs ; pour transférer des données entre la mémoire d'un accélérateur et la mémoire hôte ; pour compiler des *kernels* ; pour exécuter ces *kernels* sur les différents accélérateurs.
OpenCL homogénéise les différents *frameworks* pour les différents accélérateurs. Néanmoins la portabilité des codes et des performances n'est pas assurée. Ainsi, tant du côté hôte que du côté des accélérateurs, les codes doivent être écrits en fonction de l'architecture cible. Le nombre d'accélérateurs OpenCL disponibles dans une machine et leurs caractéristiques peuvent influencer profondément à la fois les codes des différents *kernels* (prise en compte du *coalescing*, support de la double précision, nombre de registres disponibles…) et les codes des programmes hôtes (partitionnement des données, équilibrage de charge, *prefetching*…).
Dans cet article nous proposons une solution au problème de portabilité des codes du côté du programme hôte. Notre solution expose une interface de programmation unifiée permettant d'exploiter efficacement et simplement un nombre quelconque d'accélérateurs OpenCL. De plus, elle reprend les

Travaux financés en partie par le projet ANR ProHMPT

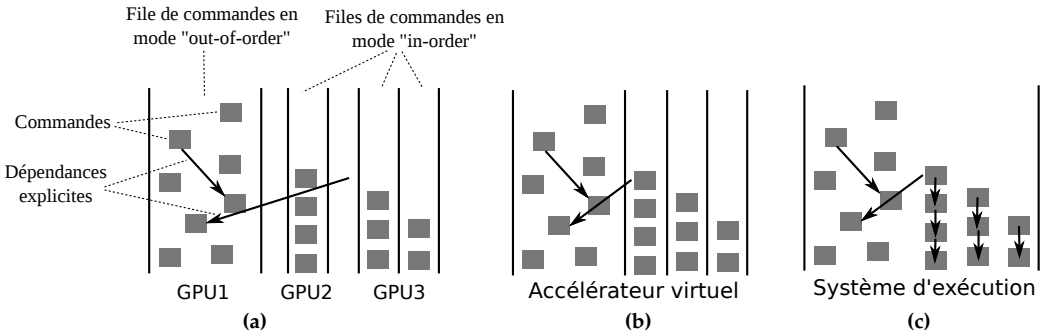


FIGURE 1 – Les tâches et leurs dépendances. (a) Files de commandes OpenCL. Une ou plusieurs files de commandes par accélérateur. (b) Files de commandes avec l'accélérateur virtuel. (c) Le support exécutif ordonnance un graphe de tâches avec des dépendances explicites.

concepts et l'API de la spécification OpenCL, ce qui la rend facilement abordable. À la place du modèle habituel avec plusieurs accélérateurs disposant chacun d'une mémoire propre et ne pouvant exécuter qu'un faible nombre de *kernels* simultanément, le modèle proposé met à disposition un seul accélérateur (virtuel) pouvant en exécuter un grand nombre en parallèle. Ce modèle de programmation, associé à un support exécutif efficace, donne de très bons résultats.

Cet article se décompose comme suit. La section 2.1 décrit le modèle de programmation exposé par notre solution. La section 2.2 détaille le support exécutif mis en œuvre à partir du modèle de programmation proposé. Enfin, la section 3 présente les performances obtenues par notre solution avec différents types de *kernels* et différents types d'architectures.

2. Modèle de programmation multi-accélérateurs unifié avec OpenCL

Le modèle OpenCL est basé autour d'un graphe de commandes asynchrones. À chaque accélérateur peut être associé une ou plusieurs files de commandes qui seront exécutées par lui. Ces commandes peuvent être de différents types : exécution de *kernel*, transfert de données, projection de données en mémoire hôte ou synchronisation. La Figure 1a montre un exemple de configuration de files de commandes pour 3 accélérateurs.
L'application doit elle-même créer une ou plusieurs files de commandes pour chaque accélérateur et choisir explicitement l'accélérateur qui exécutera une commande en la soumettant dans une de ses files. Généralement, l'application scinde les données qu'elle doit traiter en plusieurs parties qu'elle distribue ensuite sur les différents accélérateurs. Elle doit pour cela prendre en considération les dimensions des différentes mémoires car celles ci varient grandement d'un accélérateur à l'autre. Enfin, elle demande aux accélérateurs d'effectuer des traitements sur les données dont ils disposent en leur soumettant des programmes (*kernels*) à exécuter.
Les choix de partitionnement et de distribution des données effectués par l'application sont intimement liés à l'architecture sur laquelle elle s'exécute. Les paramètres à prendre en compte pour espérer une utilisation efficace des accélérateurs sont nombreux. Parmi ceux-ci, on peut citer : la capacité mémoire des accélérateurs, les débits entre les accélérateurs et la mémoire de l'hôte, les capacités de calculs et les fonctionnalités supportées par les accélérateurs…On conviendra aisément qu'une exploitation totale du potentiel des accélérateurs est loin d'être acquise. Afin d'éviter ces écueils aux programmeurs d'applications, qui ne sont pas obligatoirement experts en parallélisme et qui ont déjà les problèmes liés à leur domaine à résoudre, nous proposons une abstraction qui délègue ces difficultés à un support exécutif dédié.

2.1. Unification du modèle

Le modèle que nous proposons a pour objectif d'abstraire et d'automatiser les décisions de distribution des données et d'exécution des *kernels*. Les programmeurs d'applications n'ont ainsi plus à se préoccuper des caractéristiques des différents accélérateurs composant la plateforme d'exécution. Pour que cela

soit possible, les commandes ne doivent plus dépendre de l'architecture et donc ne doivent plus être soumises à un accélérateur en particulier.
Notre avons décidé de baser notre solution sur le standard OpenCL, la rendant ainsi plus facilement abordable, en détournant seulement légèrement le modèle utilisé. Nous proposons une implémentation de la spécification OpenCL qui ne présente qu'une seule plateforme ne contenant qu'un seul accélérateur, virtuel. Cet accélérateur virtuel masque les accélérateurs réels et se charge de la distribution des données ainsi que du placement et de l'ordonnancement des *kernels*.
La Figure 1a montre le fonctionnement classique d'OpenCL avec lequel l'application crée différentes files de commandes pour chacun des accélérateurs, tandis que la Figure 1b illustre le masquage des accélérateurs réels par un accélérateur virtuel. Cet accélérateur virtuel peut exécuter plusieurs *kernels* simultanément et sa mémoire est non-uniforme car répartie sur plusieurs accélérateurs physiques (et en partie sur la mémoire hôte). Nous utilisons donc un support exécutif qui a la charge de l'exécution des commandes sur les accélérateurs réels de sorte que la sémantique de l'application soit respectée et que les performances soient les meilleures possibles.

Ce changement de modèle soulève les problématiques suivantes :

Partitionnement des données

Le programme ne sachant pas combien d'accélérateurs sont effectivement présents, le partitionnement des données ne peut plus dépendre de ce nombre. Toutefois les données doivent être partitionnées afin que le support exécutif puisse les distribuer sur les potentiels multiples accélérateurs. On demande donc aux programmes de scinder leurs données en blocs de façon à avoir un nombre de blocs et de *kernels* raisonnable (de l'ordre de quelques dizaines ou centaines).

Compatibilité des *kernels*

Les *kernels* n'étant pas nécessairement portables, le support exécutif doit déterminer sur quels accélérateurs il peut exécuter chaque *kernel* soumis. Cela peut être réalisé par analyse de code. Les critères sont, entre autres, l'utilisation de la double précision ou de fonctionnalités introduites avec OpenCL 1.1 (vecteurs de 3 éléments, etc.). Le support exécutif doit ensuite veiller à n'ordonnancer les *kernels* que sur des accélérateurs compatibles.

Allocation de *buffer*

Lorsque des *buffers* sont alloués par l'application dans la mémoire (virtuelle) de l'accélérateur virtuel, le support exécutif peut choisir dans quelle mémoire physique les allouer. Il peut choisir de le faire dans la mémoire de l'hôte et/ou dans la mémoire de certains accélérateurs. Plusieurs *buffers* physiques alloués dans différentes mémoires peuvent alors être associés à un seul *buffer* en mémoire virtuelle. Les données dupliquées peuvent ainsi être utilisées simultanément par plusieurs *kernels* qui y accèdent en lecture seule. Dans tous les cas, le support exécutif se doit de garantir la cohérence des données dupliquées avant qu'un *kernel* ne les utilise.

Transferts de données

En OpenCL, les transferts de données peuvent se faire entre la mémoire hôte et un *buffer* ou entre deux *buffers*. Dans le cas de *buffers* virtuels qui peuvent être représentés physiquement par plusieurs *buffers* dans différentes mémoires, le support exécutif doit choisir quels *buffers* réels utiliser pour effectuer les copies. Dans le cas d'une lecture (copie d'un *buffer* vers la mémoire hôte), le support exécutif doit choisir un *buffer* source de sorte que la durée de la copie soit la plus faible possible. Dans le cas d'une copie de *buffer* à *buffer*, le support exécutif doit cette fois-ci déterminer à la fois le *buffer* source et le *buffer* destination à utiliser.
Le cas le plus difficile est celui de l'écriture (copie de la mémoire hôte vers un *buffer*). Le support exécutif doit sélectionner un *buffer* de destination. Pour cela, il peut retarder la prise de décision afin d'attendre le placement des tâches qui vont utiliser le *buffer* et effectuer le transfert directement vers le bon accélérateur. Il peut également préférer effectuer le transfert sans attendre en prenant le risque que le choix du *buffer* destination se révèle non optimal.

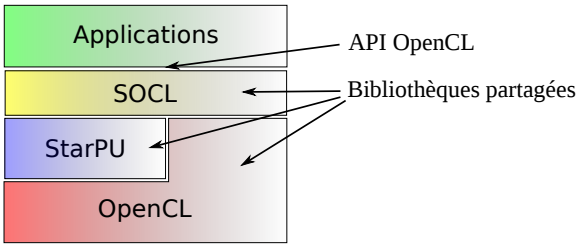


FIGURE 2 – Notre système exécutif SOCL est une bibliothèque partagée qui implémente la spécification OpenCL et qui expose donc l'API standard. Il se base sur le système exécutif StarPU et utilise les implémentations OpenCL des accélérateurs physiques disponibles

Le cas le plus simple pour le support exécutif est lorsque le programme demande le transfert implicite de données lors de la création d'un *buffer* . Dans ce cas, les données restent en mémoire hôte jusqu'à ce qu'un *kernel* en ait besoin ou jusqu'à ce qu'une décision de *prefetching* de cette donnée sur un accélérateur soit prise.

2.2. Système d'exécution

Pour réaliser notre implémentation de la spécification OpenCL avec le modèle présenté, nous avions besoin d'un support exécutif capable d'effectuer les opérations décrites à la section précédente. L'étude de l'état de l'art nous a conduit à utiliser un support exécutif existant assez proche de ce dont nous avions besoin. Notre avons utilisé le support exécutif StarPU [4] pour réaliser notre implémentation, qui se nomme en conséquence *SOCL* (pour *StarPU OpenCL*). Comme toute implémentation de la spécification OpenCL, elle se présente sous la forme d'une bibliothèque partagée implémentant l'API OpenCL (voir la Figure 2).

2.2.1. Le support exécutif StarPU

StarPU est un support exécutif pour les architectures hétérogènes. Il supporte des *kernels* écrits en CUDA, en OpenCL ou en code natif. Les différents *kernels* ayant un même « prototype » (même nombre et même types de paramètres) et accomplissant la même fonction sont regroupés au sein d'un même *codelet*. Notre implémentation n'utilise que des *kernels* OpenCL, ceux fournis par l'application. Pour chaque *kernel*, elle crée donc un *codelet* StarPU contenant uniquement le *kernel* OpenCL.
Une tâche StarPU est un *codelet* dont les paramètres ont été définis (OpenCL regroupe la notion de *codelet* et de tâche sous la même notion de *kernel*). Il est possible de définir des dépendances entre tâches StarPU puis de les soumettre pour qu'elles soient exécutées. Les dépendances étant définies explicitement, notre implémentation transforme donc les dépendances implicites d'OpenCL en dépendances explicites StarPU. La figure 1c montre un exemple de transformation à partir du cas de la figure 1b.

Modèle mémoire

Le modèle mémoire de StarPU est proche de celui dont on souhaite disposer : c'est une mémoire virtuelle englobant les différentes mémoires physiques des accélérateurs et de la mémoire hôte. Cependant StarPU ne fournit aucun mécanisme de copie entre la mémoire hôte et sa mémoire virtuelle. La seule façon de faire passer des données est de les « enregistrer » puis de les « désenregistrer ». L'équivalent OpenCL de l'enregistrement de données est la création de *buffers* avec transfert implicite (cf fin de la section 2.1). Notre implémentation utilise donc ce mécanisme pour implémenter les autres types de transferts de données autorisés par OpenCL. Pour faire simple, il s'agit de créer un *buffer* StarPU temporaire sur les données de la mémoire hôte impliquées dans le transfert, puis de soumettre une tâche qui va effectuer la copie et qui prend en paramètres le *buffer* source/destination et le *buffer* temporaire qu'elle « désenregistre » après la copie.

L'indicateur `CL_MEM_USE_HOST_PTR` ainsi qu'un pointeur vers la zone mémoire contenant les données peuvent être spécifiés lors de la création d'un *buffer* pour demander un transfert implicite des données.

Ordonnancement

StarPU ordonnance à la fois les exécutions de tâches et les transferts de données. Pour qu'une tâche soit ordonnancée sur un accélérateur par StarPU, les données dont elle a besoin doivent déjà être présentes dans sa mémoire. Les données sont automatiquement transférées par le support exécutif et des stratégies de *prefetching* peuvent donc être employées. De plus, lors de la première exécution d'une application utilisant StarPU, celui-ci détermine la bande passante entre les différents mémoires, ce qui lui permet de prendre des décisions plus pertinentes par la suite.
Le support exécutif doit déterminer l'accélérateur qui exécutera une tâche en fonction des *buffers* requis par la tâche manquants dans chaque mémoire, de la bande passante de l'accélérateur, de ses performances de calcul... Différentes stratégies sont donc possibles et sont proposées par StarPU [5]. L'utilisateur peut changer la stratégie d'ordonnancement utilisée et régler ses différents paramètres lors de l'exécution de son application. Pour cela, il lui suffit de modifier la valeur de certaines variables d'environnement.

Compatibilité des *kernels*

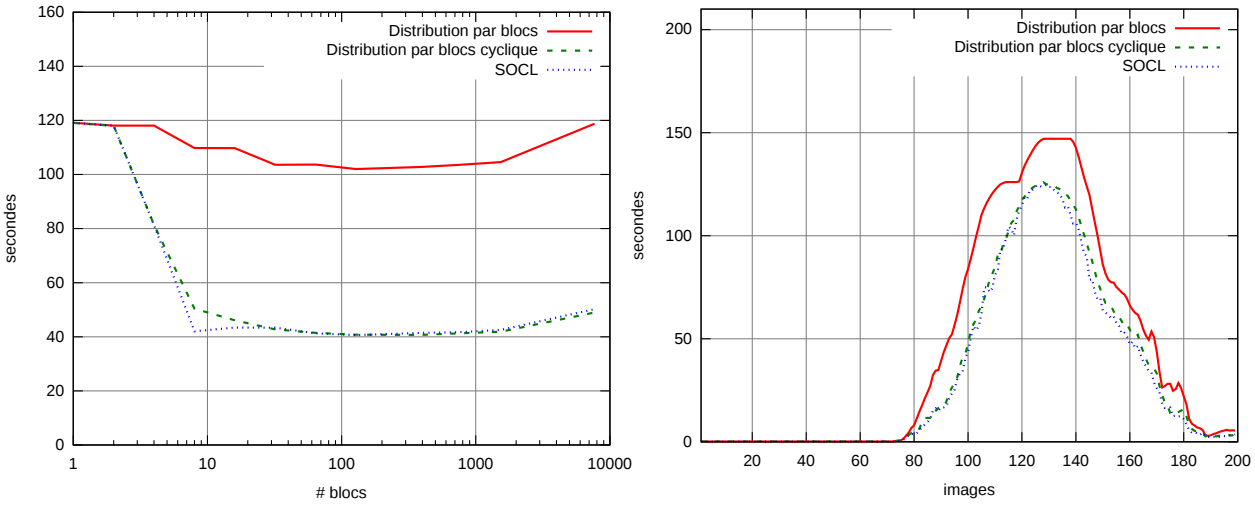
À l'heure actuelle, StarPU suppose que tous les *kernels* OpenCL qui lui sont soumis peuvent être exécutés sur tous les accélérateurs OpenCL dont il dispose. Notre implémentation souffre donc également de cette limitation. Celle-ci pourra être levée lorsque StarPU intégrera un mécanisme permettant d'exclure certains accélérateurs des choix possibles pour l'ordonnancement d'un *kernel*. En se basant sur l'échec de compilation d'un *kernel* pour certains accélérateurs ou sur une analyse du code, notre implémentation pourra ainsi autoriser l'exécution uniquement sur des accélérateurs compatibles.

3. Évaluation

Pour évaluer notre implémentation, nous avons voulu tester les cas suivants : cas où les tâches ont des durées très variables et qui ne peuvent pas être déterminées statiquement ; cas où l'architecture est hétérogène. Le premier cas a été évalué avec un code de génération d'images fractales à partir de l'ensemble de Mandelbrot. Chaque pixel est déterminé avec une vitesse de convergence qui peut varier grandement. Le deuxième cas a été évalué avec un code de multiplication matricielle à la fois sur CPU et sur GPU.
Les différents résultats ont été obtenus avec 3 GPU NVIDIA Quadro FX5800 associés à un processeur Nehalem X5550 cadencé à 2.67GHz avec 48Go de mémoire.

3.1. Génération d'images fractales

Pour cet exemple, l'image à générer est composée d'environ 79 millions de pixels (32 bits). Or le nombre d'itérations nécessaires au calcul de chaque pixel est variable. Chaque *kernel* se voyant attribuer un bloc de lignes de pixels différent à calculer, les durées d'exécution des *kernels* varient grandement (de 0.8ms à 2.6s). Le graphique de la figure 3a indique le temps nécessaire pour générer une image en fonction du nombre de blocs de lignes. L'image fractale générée pour ce test a été choisie de façon à ce que seule la moitié supérieure de l'image contienne des pixels nécessitant beaucoup de calcul. On compare notre implémentation avec une distribution par blocs et avec une distribution par blocs cyclique, utilisant directement l'implémentation OpenCL de NVIDIA.
On constate que les performances obtenues par notre implémentation sont bien supérieures à celles obtenues avec la distribution par blocs et comparables à celles obtenues avec la distribution par blocs cyclique. Pour généraliser ces résultats à un plus grand nombre de cas, nous avons comparé de la même façon la génération de 200 images différentes obtenues par déplacement et zoom dans l'ensemble de Mandelbrot avec un nombre de blocs fixé. Les résultats obtenus sont indiqués dans le graphique de la Figure 3b.
On remarque que notre support exécutif obtient les meilleurs résultats. Ses performances restent proches de celles de la distribution par blocs cyclique, ce qui est assez positif, cette dernière étant *à priori* la meilleure décomposition statique possible pour ce type de problème. En effet, considérant la nature des images générées, il est improbable de rencontrer des cas où les pixels nécessitant le plus d'opérations sont majoritairement affectés à un même accélérateur avec cette distribution. Cette dernière effectue



(a) Temps d'exécution en fonction du nombres de blocs pour une image fixée

(b) Temps d'exécution pour différentes images avec un nombre de blocs fixé à 16

FIGURE 3 – Génération d'images fractales de l'ensemble de Mandelbrot

donc un lissage de charge très correct bien que légèrement moins bon que celui effectué par notre support exécutif.

3.2. Multiplication matricielle

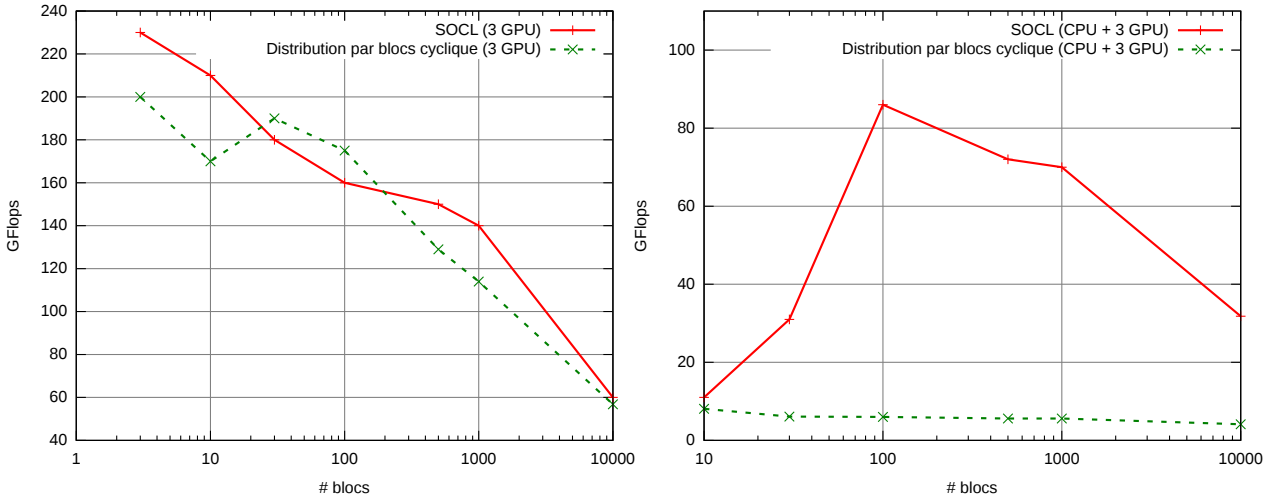
Le code utilisé pour ce test effectue une multiplication matricielle entre deux matrices de flottants en simple précision générées de façon aléatoire. La parallélisation a été effectuée de sorte que pour calculer $C = A \times B$, chaque *kernel* prend la totalité de la matrice B et un bloc de lignes des matrices A et C en paramètres. La Figure 4a présente les performances obtenues avec 3 GPU en fonction du découpage plus ou moins fin des matrices A et C. Le code du *kernel* étant très naïf, les performances sont très en deça de ce qu'on pourrait attendre d'un code optimisé sur cette architecture. Néanmoins, ce test montre que notre support exécutif obtient des performances comparables à une distribution par blocs cyclique, adaptée à ce type de problème.

On constate qu'un faible nombre de blocs peut fortement pénaliser les performances. Par exemple avec 10 blocs, l'un des trois GPU va exécuter quatre *kernels* tandis que les deux autres vont en exécuter trois. Avec un mauvais placement, comme cela semble être le cas avec la distribution par blocs cyclique sur cette configuration, les performances sont dégradées. Notre support exécutif limite ce phénomène en choisissant le premier des trois GPU qui a terminé l'exécution de trois *kernels* pour exécuter le dernier.

Nous avons ensuite voulu tester la portabilité de ce code sur une architecture hétérogène. Pour cela, nous avons activé l'utilisation de l'implémentation OpenCL pour CPU fournie par AMD. Les trois GPU et le CPU sont alors utilisés simultanément. La Figure 4b présente les résultats que nous avons obtenus avec notre support exécutif et ceux obtenus avec une distribution par blocs cyclique en utilisant directement les implémentations OpenCL de NVIDIA et de AMD.

On constate que la présence du CPU fait globalement s'effondrer les performances mais qu'elles sont néanmoins bien meilleures avec notre support exécutif. En analysant le journal des ordonnancements effectués par ce dernier, on observe que les exécutions de *kernels* sur GPU sont toutes terminées lorsque le CPU termine l'exécution de son premier *kernel*, et donc que notre support exécutif n'a exécuté qu'un seul *kernel* sur le CPU. La distribution statique exécute quant à elle un *kernel* sur quatre sur le CPU, ce qui explique l'écart de performances.

En employant une stratégie d'ordonnancement de notre support exécutif basée sur l'historique des exécutions précédentes et en exécutant plusieurs fois l'application, nous avons pu constater que le support exécutif ne place plus ce *kernel* sur le CPU du tout. On retrouve alors les performances de la Figure 4a. Le changement de stratégie d'ordonnancement s'effectue sans toucher au code, il suffit de paramétrer



(a) Architecture homogène composée de 3 GPU **(b)** Architecture hétérogène composée de 3 GPU et 1 CPU

FIGURE 4 – Multiplication matricielle entre deux matrices de dimensions $10^3 \times 10^5$ et $10^3 \times 10^3$ en simple précision

quelques variables d'environnement. Cette façon de faire permet de bien décorréler l'écriture du code métier de l'application et le *tuning* des performances pour une architecture donnée. Ces deux actions pouvant alors aisément être effectuées par des personnes différentes, chacune experte dans son domaine.

4. Travaux similaires

L'ordonnancement de tâches sur architectures hétérogènes a fait l'objet de nombreux travaux. Grewe et O'Boyle [9] proposent une approche statique basée sur OpenCL qui détermine à partir de modèles prédictifs le partitionnement des données et le placement des tâches. Leurs travaux se basent sur une analyse statique lors de la compilation. Contrairement à nos travaux, ils n'ont donc pas de surcout lié à la prise de décision à l'exécution. Cependant si des erreurs sont commises lors de la prise de décision, ils ne peuvent pas s'adapter dynamiquement. En particulier, il est très difficile de modéliser les contentions dues au partage de la bande passante entre plusieurs accélérateurs ainsi que la durée d'exécution de certains *kernels*.

Le *framework* Harmony [7] permet d'ordonnancer des tâches sur des architectures hétérogènes. Couplé au compilateur Ocelot [8], il prend en entrée du code PTX (CUDA) et le compile pour diverses architectures. L'ordonnancement des tâches s'effectue en fonction de leurs durées d'exécution précédentes. Ce type d'ordonnancement peut également être utilisé avec le support exécutif StarPU que nous utilisons comme nous l'avons montré.

StarSS [6] dispose également d'un support exécutif pour architectures hétérogènes. Actuellement, il prend en entrée un codé annoté à l'aide d'un ensemble de *pragmas* à la manière d'OpenMP. Ces annotations permettent de déclarer les tâches et les données qu'elles utilisent. Il pourrait néanmoins être envisagé d'utiliser notre modèle de programmation basé sur OpenCL avec le support exécutif StarSS.

5. Conclusion

Pour exploiter les accélérateurs OpenCL, les applications doivent habituellement effectuer elles-mêmes le partitionnement et la distribution des données ainsi que le placement des *kernels* sur les accélérateurs disponibles. C'est une tâche complexe, qu'il est nécessaire de refaire pour chaque application, et dont les bons résultats en matière de portabilité des performances sont difficiles à obtenir.

Nous proposons un modèle de programmation qui simplifie et unifie l'expression du parallélisme tout

en permettant d'obtenir de bonnes performances de façon portable. Nous avons réalisé un support exécutif permettant d'utiliser ce modèle en nous basant sur un support exécutif existant, StarPU. Contrairement à ce dernier, notre support exécutif expose ses différentes fonctionnalités au travers de l'API du standard OpenCL, si bien qu'il est très facile à prendre en main. Contrairement aux implémentations OpenCL usuelles, il masque les accélérateurs réels aux applications et se charge de l'ordonnancement des transferts de données et des exécutions de *kernels*. De plus, il propose différentes stratégies d'ordonnancement et est capable de s'adapter à des architectures variées.
Nous avons évalué notre solution dans le cas de *kernels* effectuant un nombre d'opérations variable sur une architecture homogène et dans le cas de *kernels* effectuant un nombre d'opérations fixe sur une architecture hétérogène. Dans les deux cas, nous avons constaté que les performances obtenues avec notre support exécutif étaient très satisfaisantes et souvent bien meilleures que celles obtenues avec des distributions statiques (distribution par blocs et distribution par blocs cyclique). Nous avons également pu constater, lors de l'écriture de ces tests, que notre modèle est beaucoup plus simple à utiliser que le modèle OpenCL usuel, en particulier lors de l'utilisation simultanée de différentes plateformes OpenCL, puisque il s'écrit aussi simplement que si un seul accélérateur était utilisé.
Ce travail nous permet maintenant d'envisager de nouvelles optimisations avec pour objectifs à la fois de simplifier encore la programmation d'applications tout en améliorant les performances. En particulier, nous pourrions automatiser les partitionnements de données et adapter la granularité du découpage en fonction des accélérateurs disponibles. Il serait également envisageable de travailler sur les codes s'exécutant sur les accélérateurs (*kernels*) afin de les optimiser pour les architectures cibles, même celles pour lesquelles le code n'a pas été manuellement optimisé par le programmeur. Cela pourrait être fait à partir du langage OpenCL C ou à partir d'un langage de plus haut niveau d'abstraction. Notre support exécutif supporte d'ores et déjà différentes stratégies d'ordonnancement et de *prefetching* des données sur les accélérateurs. La sélection automatique et le changement dynamique de stratégie pourrait être réalisés, soulageant le programmeur d'un choix dont les conséquences sont parfois difficiles à évaluer.

Notre support exécutif SOCL est librement accessible à l'adresse suivante :
`http://socl.gforge.inria.fr/`.

Bibliographie

1. AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream). `http://developer.amd.com/gpu/amdappsdk`.
2. IBM OpenCL Development Kit for Linux on Power. `http://www.alphaworks.ibm.com/tech/opencl`.
3. NVIDIA OpenCL SDK. `http://www.nvidia.com/opencl`.
4. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par '09 : Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.
5. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 2010.
6. Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.
7. G. Diamos and S. Yalamanchili. Harmony : A Flexible Runtime for Heterogeneous Many Core Architectures. In *ACM/IEEE International Symposium on High Performance Distributed Computing*, 2008.
8. Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili. GPU Application Developpement, Debugging, and Performance Tuning with GPU Ocelot. In *GPU Computing GEMS, vol. 1*, 2011.
9. Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC '11 : Proceedings of the 20th International Conference on Compiler Construction*. Springer, 2011.