Programmation unifiée multi-accélérateur OpenCL

Henry Sylvain, Alexandre Denis, Denis Barthou

Programmation unifiée multi-accélérateur OpenCL

Sylvain Henry [1], **Alexandre Denis** [2], **Denis Barthou** [1]

1. *Université de Bordeaux,*
   351, cours de la Libération,
   F-33405 Talence
   {sylvain.henry,denis.barthou}@labri.fr

2. *INRIA Bordeaux – Sud-Ouest,*
   200, avenue de la Vieille Tour,
   F-33405 Talence
   alexandre.denis@inria.fr

RÉSUMÉ. *Le standard OpenCL propose une interface de programmation basée sur un parallélisme de tâches et supportée par différents types d'unités de calcul (GPU, CPU, Cell...). L'une des caractéristiques d'OpenCL est que le placement des tâches sur les différentes unités de calcul doit être fait manuellement. Pour une machine hybride disposant par exemple de multicœur et d'accélérateur(s), l'équilibrage de charge entre les différentes unités est très difficile à obtenir à cause de cette contrainte. C'est particulièrement le cas des applications dont le grain et le nombre des tâches varient au cours de l'exécution. Il en découle par ailleurs que le passage à l'échelle d'une application OpenCL est limitée dans le contexte d'une machine hybride. Nous proposons dans cet article de remédier à cette limitation en créant une unité virtuelle et parallèle de calcul, regroupant les différentes unités de la machine. Le placement manuel d'OpenCL cible cette unité virtuelle, et la responsabilité du placement sur les unités réelles est laissée à un support exécutif. Ce support exécutif se charge d'effectuer les transferts de données et les placements des tâches sur les unités réelles. Nous montrons que cette solution permet de simplifier grandement la programmation d'applications pour architectures hybrides et cela de façon efficace.*

ABSTRACT. *The OpenCL standard defines a unified programming interface based on a task model that can target different kinds of compute units (GPU, CPU, Cell...). OpenCL gives application programmers the full control of task scheduling on the different devices that are available. Consequently, programmers are most likely to rely on a static task scheduling scheme. However this may not be efficient on heterogeneous architectures such as those containing accelerators in addition to the CPU. Load-balancing is really hard to achieve, especially if data have different granularities and if the number of tasks can't be statically known. OpenCL programs are thus hardly scalable on hybrid systems. In this paper, we propose to solve this issue by grouping every OpenCL unit into a single virtual one. By doing that, applications only have to submit*

Technique et science informatiques – n° 8-9-10/2012, 1233-1249

tasks to this virtual unit and let the runtime system manage data transfers and task scheduling on real units. We show that this approach really eases the burden typically associated with application programming on hybrid systems and that it is done in a efficient manner.

MOTS-CLÉS : *OpenCL, GPU, support exécutif, répartition de charge, portabilité.*

KEYWORDS: *OpenCL, GPU, runtime systems, load-balancing, scalability.*

DOI:10.3166/TSI.31.1233-1249

1. Introduction

Depuis quelques années, l'utilisation d'accélérateurs matériels (processeurs graphiques, Many Integrated Chips d'Intel, Cell Broadband Engine de IBM, ClearSpeed) en complément de multicœur se généralise. Ces architectures hybrides offrent en effet un très bon rapport entre les performances et le coût de fabrication. L'utilisation conjointe des accélérateurs et des multicœur permet de pallier les limitations de chacun des types d'architecture en utilisant la mieux adaptée pour effectuer chacun des calculs et ainsi obtenir de meilleures performances.

Programmer ces architectures est cependant difficile car les modèles d'exécution, l'expression du parallélisme et l'accès à la mémoire des accélérateurs diffèrent des processeurs classiques. Certains outils de compilation – HMPP (Dolbeau *et al.*, 2007), Pluto (Bondhugula *et al.*, 2008), Par4all (HPC-Project, 2012) par exemple – proposent un modèle plus ou moins unifié de programmation, s'aidant notamment de directives ou de techniques de parallélisation automatique. OpenCL est un langage standardisé de programmation parallèle apparu en 2008 et permettant l'utilisation simultanée et unifiée – avec le même modèle d'exécution – de différents types d'accélérateurs et de multicœurs. La programmation par tâches, proposée par OpenCL – les tâches étant elles-mêmes programmées sur un modèle SIMD – permet d'exprimer une large gamme de parallélismes et les compilateurs OpenCL existants génèrent des tâches efficaces sur une variété grandissante d'architectures. En revanche, la définition du grain des tâches parallèles, leur placement sur les accélérateurs et les transferts des données qu'elles utilisent restent à la charge du programmeur. Ceci a pour conséquences que les codes applicatifs se doivent de prendre en considération l'hétérogénéité des architectures pour obtenir un bon équilibrage de charge, ce qui est très difficile.

Dans cet article, nous proposons une solution permettant d'éviter le placement manuel des tâches d'OpenCL sur les unités parallèles des machines hybrides ainsi que les transferts explicites de données, tout en restant dans le modèle de programmation OpenCL. En regroupant les différentes unités réelles de calcul au sein d'une unité virtuelle et parallèle, nous montrons qu'il est possible à l'aide d'un support d'exécution d'obtenir de meilleures performances, notamment par un meilleur équilibrage de charge entre les unités.

La suite de cet article est organisée de la façon suivante. La section 2 décrit les différents modèles de programmation permettant d'exploiter des architectures hétéro-

gènes. La section 3 montre de quelle façon il est possible à travers une interface de programmation de bas niveau standard (OpenCL) de bénéficier des avantages habituellement liés à l'utilisation d'un modèle de programmation de plus haut niveau. La section 4 montre les résultats obtenus avec notre implémentation à travers trois exemples.

2. Programmation d'architectures hybrides

Dans cette section, nous présentons et analysons les différents modèles de programmation pour architectures hétérogènes.

Pour exploiter efficacement une architecture hybride, il faut être capable d'utiliser simultanément toutes les unités de calcul disponibles. Il faut donc un modèle de programmation permettant au programmeur ou au compilateur de décrire l'application sous la forme d'un graphe de tâches, de sorte qu'il soit possible de déterminer les portions de codes qui peuvent être exécutées en parallèle. Il faut ensuite placer ces tâches sur les unités de calcul, c'est-à-dire assigner à chaque tâche une des unités de calcul qui sera chargée de son exécution.

Étant donné que les accélérateurs disposent chacun de leur propre mémoire, le modèle usuel à base de mémoire partagée n'est plus directement utilisable. Des transferts de données entre les différentes mémoires doivent être programmés, souvent à l'aide de contrôleurs DMA. Ces transferts peuvent s'effectuer vers ou depuis des zones de données qui ont été allouées dans la mémoire d'un accélérateur. La gestion de la mémoire locale des accélérateurs, de son allocation, désallocation et cohérence est à la charge de l'application, du compilateur ou du support exécutif en fonction du modèle de programmation utilisé. Elle dépend étroitement de l'ordonnancement et du placement des tâches. Dans la suite de cette section, différents modèles de programmation pour architectures hybrides sont présentés.

2.1. Modèle bas niveau

Le modèle de plus bas niveau est celui qui est généralement employé dans les kits de développement fournis par les fabricants d'accélérateurs. On peut notamment citer le kit de développement CUDA de NVIDIA (NVIDIA Corporation, 2011), celui de ClearSpeed (ClearSpeed, 2010) ou, plus récemment, le standard OpenCL (Khronos OpenCL Working Group, 2010) implémenté par de nombreux fabricants depuis sa sortie. Avec ce modèle, aucune décision n'est prise automatiquement par le support exécutif. L'application doit donc elle-même gérer :

1. le placement des tâches ;
2. l'ordonnancement des tâches ;
3. la gestion de la mémoire des accélérateurs ;
4. la gestion des transferts de données.

Dans le cas d'OpenCL, le lancement asynchrone des tâches et la gestion des événements permettent de définir un ordre partiel d'exécution, laissant au support exécutif le choix de l'ordonnancement. Toutefois, le placement reste manuel.

Ce modèle laisse donc à l'application la gestion fine des accélérateurs. Il revient aux développeurs de l'application utilisant ce modèle d'effectuer eux-mêmes les choix qui leur semblent être les meilleurs. Cependant il est très difficile d'effectuer les bons choix notamment lorsque l'architecture utilisée dispose de multiples accélérateurs aux propriétés distinctes (capacité mémoire, rapidité d'exécution, bande passante des transferts vers la mémoire hôte...) et lorsque des choix *a priori* peuvent ne pas convenir pour certaines applications dont la charge évolue dynamiquement, par exemple.

Une variante du modèle bas niveau est le modèle bas niveau sans placement. Ce modèle demande aux programmeurs d'applications autant d'informations que le modèle bas niveau excepté le placement des tâches qui est effectué automatiquement. (Grewe, O'Boyle, 2011) proposent une approche statique qui détermine à la compilation et à partir de modèles prédictifs le partitionnement des données et le placement des tâches.

2.2. Modèle par déport de tâches

Le modèle consistant à déporter des tâches sur l'accélérateur (*offloading*) permet aux applications de ne spécifier que le placement des tâches. La gestion de la mémoire des accélérateurs ainsi que les transferts de données sont effectués automatiquement avant et après l'exécution de la tâche. C'est par exemple le modèle à la base de la proposition de standard OpenHMPP.

Deux implémentations de la proposition de standard OpenHMPP existent actuellement : l'une développée par CAPS Entreprise (Dolbeau *et al.*, 2007), l'autre par PathScale (PathScale, 2012). OpenHMPP repose sur un ensemble d'annotations fournies aux programmeurs pour les langages C et Fortran. L'une de ces annotations, placée au-dessus d'une déclaration de fonction, permet de décrire sur quel type d'accélérateur cette fonction sera exécutée et de préciser la façon dont ses paramètres seront accédés (lecture seule, écriture seule, lecture-écriture) ainsi que leur dimension dans le cas de données de type pointeur. De la même façon, une annotation placée au-dessus d'un appel à une fonction précédemment annotée permet de préciser des informations sur les paramètres (dimensions...) et d'indiquer si l'exécution doit être synchrone ou asynchrone.

L'un des inconvénients de ce modèle est que pour recouvrir les communications vers/depuis les accélérateurs par des calculs, il est nécessaire de donner des indications supplémentaires permettant au support exécutif d'anticiper les allocations et les transferts de données. Avec OpenHMPP, cela se traduit par d'autres annotations qui peuvent être placées dans le code de l'application et qui permettent d'allouer ou de libérer de la mémoire sur un accélérateur et d'effectuer des transferts de données. Cela revient donc à une gestion bas niveau de la mémoire des accélérateurs et des transferts

de données, de façon simplifiée toutefois car ces indications supplémentaires sont facultatives (à la fois pour le programmeur peu soucieux des performances et pour le support exécutif qui serait dans l'impossibilité d'effectuer une action anticipée).

Certains langages, tels que Chapel (Sidelnik *et al.*, 2011), reposent sur un modèle de programmation à base de tableaux dont la distribution entre les différentes mémoires peut être explicitée par l'application. Lorsqu'une architecture hétérogène est employée, l'application peut utiliser une distribution particulière qui indique au compilateur que les données devront être placées dans la mémoire d'un accélérateur. Les codes utilisant ces données seront alors exécutés par l'accélérateur qui les contient dans sa mémoire. Comme pour OpenHMPP, il est possible d'anticiper les transferts en copiant des données vers des variables placées dans la mémoire d'un accélérateur.

Le modèle PGAS (*Partitioned Global Address Space*) tel que proposé par IBM avec le langage de programmation X10 (Saraswat *et al.*, 2011) est similaire au modèle par déport de tâches lorsqu'il s'agit d'exécuter des tâches sur des nœuds distants dans un cluster (opérateur *at* dans le cas de X10). Cependant l'approche employée pour exploiter les architectures hybrides repose sur le modèle bas niveau (Cunningham *et al.*, 2011) : les allocations de données et les transferts sont effectués explicitement.

De la même façon, XcalableMP (Lee *et al.*, 2011), un *framework* à base d'annotations utilisant également un modèle PGAS, propose un nouvel ensemble d'annotations pour les accélérateurs (XcalableMP-ACC). Comme dans le cas de X10, cette extension repose sur le modèle bas niveau. L'utilisation d'annotations rend également plus difficile la portabilité sur les architectures contenant plusieurs accélérateurs. À la connaissance des auteurs, XcalableMP supporte uniquement un accélérateur par nœud.

Sequoia (Fatahalian *et al.*, 2006) dispose également d'un module permettant de placer des tâches sur un accélérateur (uniquement les GPU NVIDIA). Cependant, il nécessite quelques aménagements dans le modèle d'architecture utilisé car la hiérarchie mémoire d'un GPU ne se présente pas directement sous la forme d'un arbre. De plus, dans le cas des machines hybrides où différents accélérateurs ont des capacités mémoires différentes, la modélisation sous forme d'arbre symétrique n'est plus valable.

2.3. Modèle virtualisé

Le modèle virtualisé masque totalement les accélérateurs matériels à l'application. Cette dernière n'a besoin de fournir au compilateur ou au support exécutif qu'un graphe de tâches (*i.e.* des tâches avec leurs interdépendances).

Les tâches ne peuvent lire et écrire que des zones mémoires allouées par l'intermédiaire de l'allocateur fourni par le support exécutif. Les applications ne savent pas dans quelle(s) mémoire(s) physique(s) se situent les données à chaque instant. Cependant le modèle garantit qu'avant l'exécution d'une tâche par un accélérateur, les données qu'elle utilise auront été transférées (si nécessaire) dans sa mémoire.

Les performances obtenues avec le modèle virtualisé reposent principalement sur les stratégies employées par le support exécutif. En conséquence, les stratégies d'ordonnancement et de placement des tâches doivent absolument tenter de minimiser les quantités de données transférées tout en essayant d'exploiter au maximum le potentiel de chaque accélérateur.

Le support exécutif StarPU, ainsi que les *frameworks* (compilateur et support exécutif) StarSS et Harmony implémentent ce modèle (Augonnet *et al.*, 2010 ; Ayguadé *et al.*, 2009 ; G. F. Diamos, Yalamanchili, 2008). Ils se distinguent par la façon dont sont décrites les tâches – un même code compilé pour différentes architectures ou un code par architecture – ainsi que par la façon dont le contrôle de l'exécution est effectué – par ajout d'annotations dans le code ou par le biais d'appels aux fonctions d'une bibliothèque. Les performances obtenues avec ces systèmes peuvent être très satisfaisantes comme en témoigne la récente collaboration entre le support exécutif StarPU et la bibliothèque MAGMA (Agullo *et al.*, 2010).

Dans la même famille de solutions, Unified Parallel C (UPC) (Chen *et al.*, 2011) intègre un support des accélérateurs de type GPU. Il permet de décrire un noyau de calcul à exécuter sur GPU sous la forme d'un nœud de boucles `upc_forall` hiérarchique. Le support exécutif se charge d'effectuer les placements et les transferts de données nécessaires.

SWARM (ET International, Inc., 2011) est un *framework* permettant d'exécuter des tâches écrites avec le langage *Swarm2c* – une extension au langage C – sur des architectures multicœur grâce à un support exécutif inclus dans le *framework*. Le graphe de tâches est décrit sous la forme d'un flot de données de sorte que lorsqu'une donnée est produite par une tâche, elle est passée en argument à un *callback* qui se charge d'exécuter d'autres tâches ou de terminer l'exécution du graphe.

Enfin, le *framework* Intel ArBB est issu de la fusion entre les *frameworks* Ct et RapidMind, ce dernier étant lui-même la version commerciale de Sh. Ce *framework* permet d'effectuer des opérations de type « data-parallel » sur des vecteurs de données. Initialement, le *framework* RapidMind utilisait principalement les accélérateurs de type GPU. Depuis la fusion, seuls les architectures multicœur sont supportées. Ce *framework* repose également sur le modèle virtualisé.

2.4. Standardisation des modèles

S'il existe différents modèles et de nombreuses solutions implémentant tout ou partie de ces modèles, à l'heure actuelle aucun consensus ne se dégage dans la communauté du calcul haute performance. Bien que plusieurs projets de standardisation soient proposés (OpenHMPP, XcalableMP. . .), la seule spécification standardisée dans le domaine est celle d'OpenCL (Khronos OpenCL Working Group, 2010) qui utilise le modèle bas niveau.

3. Du modèle bas niveau au modèle virtualisé

Nous présentons maintenant notre projection du modèle bas niveau vers un modèle virtualisé.

Comme nous l'avons mentionné à la section précédente, le modèle par déport de tâches n'est pas encore standardisé, si tant est qu'il le soit un jour. De la même façon, il n'existe pas, à notre connaissance, de standard pour le modèle virtualisé. À la date d'écriture de cet article, seul le modèle bas niveau dispose d'un standard reconnu dont diverses implémentations sont mises à disposition : OpenCL.

Or, pour garantir la pérennité des développements, les développeurs d'applications dans le domaine du calcul haute performance préfèrent choisir un modèle bas niveau standardisé et dont il existe plusieurs implémentations, plutôt qu'un modèle de plus haut niveau non standardisé où chaque support exécutif dispose de sa propre interface et peut être amené à disparaître.

Prenant en considération les difficultés et les délais inhérents à la définition, la diffusion et l'adoption d'un standard pour ces modèles, nous avons choisi d'aborder le problème sous un autre angle. Nous nous sommes interrogés sur la pertinence et la faisabilité d'utiliser un modèle qui allierait à la fois un modèle de programmation bas niveau standard et un modèle d'exécution haut niveau similaire à celui utilisé pour le modèle virtualisé . Notre solution se place donc à la fois au niveau d'une implémentation OpenCL et au-dessus d'un support exécutif supportant le modèle virtualisé.

3.1. Virtualisation des accélérateurs

Pour passer du modèle bas niveau au modèle virtualisé, il faut déterminer comment chaque commande bas niveau peut être convertie en une ou plusieurs commandes supportées par le modèle virtualisé. En premier lieu, l'implémentation placera toutes les commandes sur l'accélérateur virtuel.

Le tableau 1 résume les correspondances entre les commandes bas niveau qui seront reçues par l'implémentation à travers l'interface bas niveau et celles qui seront alors envoyées au support exécutif sous-jacent en fonction des commandes qu'il supporte. Lorsque le support exécutif du modèle virtualisé ne supporte pas certaines commandes, il est nécessaire de les simuler afin de conserver la sémantique de l'application.

Tout d'abord, les allocations et suppressions de *buffers* sur les accélérateurs peuvent être directement converties en commandes équivalentes du modèle virtualisé, à la différence qu'aucun accélérateur particulier n'est indiqué. De la même façon, la soumission de tâches à exécuter se traduit directement.

Les transferts de données entre deux *buffers* sont également simples à traduire : il est possible de les simuler en utilisant une tâche qui effectue une copie entre eux. Cependant cela peut occasionner des copies inutiles si les deux *buffers* ne sont pas initialement présents sur le même accélérateur. En effet, au moins un transfert aura

alors lieu pour placer les deux *buffers* dans la mémoire d'un unique accélérateur, puis la tâche de copie sera exécutée par celui-ci. Cette tâche de copie aurait pu être évitée si le premier transfert avait été fait directement vers la bonne position dans le *buffer* destination.

Les transferts entre la mémoire hôte et un *buffer* sont légèrement plus complexes à gérer. Il faut créer un *buffer* temporaire *mappé* sur la zone de la mémoire hôte impliquée dans le transfert puis effectuer un simple transfert entre deux *buffers*. Le *buffer* temporaire doit ensuite être automatiquement supprimé une fois la copie effectuée.

Le modèle bas niveau peut fournir des commandes pour projeter un *buffer* en mémoire hôte temporairement, c'est-à-dire pour faire en sorte que l'application hôte puisse accéder au contenu d'un *buffer* de la même façon qu'elle accède aux autres données dans son propre espace d'adressage. Si cette commande n'est pas proposée par le support exécutif du modèle virtualisé , il est possible de la simuler en utilisant un transfert entre le *buffer* qui doit être projeté et la mémoire hôte. Si le *buffer* est projeté en écriture, un deuxième transfert est nécessaire pour mettre à jour les données du *buffer*.

Tableau 1. Correspondances entre le modèle bas niveau et le modèle virtualisé

Modèle bas niveau	**Modèle virtualisé**
Allocation de *buffer* sur un accélérateur	Allocation de *buffer*
Suppression de *buffer* sur un accélérateur	Suppression de *buffer*
Exécution de *kernel* sur un accélérateur	Exécution de *kernel*
Transfert de données entre deux *buffers*	Idem ou tâche de copie
Transfert de données entre la mémoire hôte et un *buffer*	Idem ou tâche de copie entre un *buffer* éphémère et un *buffer*
Projection (*mapping*) d'un *buffer* en mémoire hôte	Idem ou transfert de données entre la mémoire hôte et un *buffer* (x2 si projection en écriture)

Par cette correspondance entre les commandes d'un support exécutif bas niveau et les commandes d'un support exécutif du modèle virtualisé, il est possible d'utiliser des applications prévues pour le premier avec le deuxième. On observe alors que les gains de performances sont variables d'une application à l'autre, suivant la manière dont elle a été écrite. La section suivante donne quelques indications sur les points à prendre en compte lors de l'écriture d'une application avec le modèle bas niveau qui permettent d'optimiser la réalisation de la correspondance entre les deux modèles, et donc les performances.

3.2. Optimisation des codes

Bien qu'utilisant l'interface de programmation du modèle de bas niveau, il est possible d'écrire des codes qui prennent en compte le support exécutif sous-jacent et

permettent d'obtenir de meilleures performances. Pour cela, il faut que les applications respectent les indications suivantes.

3.2.1. Partitionnement des données

Lorsque le support exécutif virtualisé est utilisé, les applications ne savent pas combien d'accélérateurs sont effectivement présents. Contrairement à ce qu'on trouve régulièrement dans les codes écrits avec le modèle bas niveau, le partitionnement des données ne peut donc pas dépendre de ce nombre. Les données doivent cependant être partitionnées afin que le support exécutif puisse les distribuer sur les différents accélérateurs. Il est donc recommandé aux programmeurs d'applications de scinder invariablement leurs données en blocs de façon à avoir un nombre de blocs, un nombre de *kernels* et des tailles de données pertinents (de l'ordre de quelques dizaines de tâches et des tailles de données de l'ordre de quelques dizaines ou centaines de mégaoctets).

3.2.2. Initialisation des buffers

L'interface du modèle bas niveau fournit généralement deux moyens pour initialiser les données d'un *buffer* : soit en effectuant un transfert explicite de données après l'allocation du *buffer*, soit en indiquant, lors de la création du *buffer*, une zone de mémoire hôte qui contient les données avec lesquelles le *buffer* doit être initialisé [1].

Il est recommandé d'utiliser cette deuxième méthode car dans ce cas, le support exécutif n'effectue aucun transfert jusqu'à ce qu'une tâche ait besoin du *buffer* ou jusqu'à ce qu'une décision de *prefetching* de cette donnée sur un accélérateur soit prise. En quelque sorte, le transfert des données est associé à la tâche de calcul qui va utiliser le *buffer* concerné par le transfert, tandis qu'avec l'autre méthode le transfert est potentiellement exécuté vers la mémoire d'un accélérateur sur lequel aucune tâche utilisant le *buffer* ne sera placée. Néanmoins, l'inconvénient de cette méthode d'initialisation des *buffers* est que la zone mémoire utilisée pour l'initialisation est nécessairement contiguë.

3.3. Implémentation

Pour valider notre approche, nous avons besoin d'un support exécutif permettant d'utiliser le modèle « virtualisé ». Plutôt que de l'écrire nous-mêmes, nous avons choisi d'en utiliser un existant. Notre choix s'est porté sur le support exécutif StarPU (Augonnet *et al.*, 2010) avec lequel nous sommes plus familiers, mais nous aurions tout aussi bien pu utiliser le support exécutif StarSS (Ayguadé *et al.*, 2009) ou Harmony (G. Diamos, Yalamanchili, 2008). En conséquence, notre implémentation est sobrement nommée SOCL pour StarPU OpenCL.

Notre support exécutif SOCL est une bibliothèque partagée qui implémente la spécification OpenCL et qui expose donc l'API du standard. Il se base sur le support

1. Cf. le drapeau `CL_MEM_USE_HOST_PTR` de la spécification OpenCL.

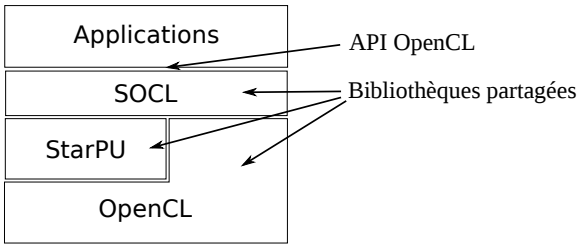


Figure 1. Illustration des interactions entre les différentes couches logicielles

exécutif StarPU et utilise les implémentations OpenCL des accélérateurs matériels disponibles. La figure 1 montre l'organisation de ces différentes couches logicielles.

StarPU est un support exécutif pour les architectures hétérogènes qui supporte les tâches OpenCL. Les dépendances entre les tâches peuvent être définies uniquement de façon explicite, notre implémentation transforme donc les dépendances implicites entre tâches OpenCL en dépendances explicites entre tâches StarPU. Le modèle mémoire de StarPU est constitué d'une mémoire virtuelle qui englobe les différentes mémoires physiques des accélérateurs et une partie de la mémoire hôte. StarPU ne fournit aucun mécanisme de copie entre la mémoire hôte et sa mémoire virtuelle.

La seule façon de faire passer des données dans la mémoire virtuelle StarPU est de les « enregistrer », c'est-à-dire d'indiquer au support exécutif qu'une zone mémoire contient une donnée. Après l'enregistrement, l'application ne doit plus utiliser cette zone mémoire car elle fait dorénavant partie de la mémoire virtuelle du support exécutif. L'opération inverse permet à l'application de récupérer le contrôle d'une zone mémoire préalablement enregistrée. L'application peut alors choisir de récupérer la zone mémoire en l'état, c'est-à-dire contenant des données potentiellement incohérentes, ou bien de forcer le support exécutif à mettre à jour la zone mémoire avec la dernière version de la donnée pouvant se trouver dans une autre mémoire. Les implémentations dans SOCL des différents modes de transferts de données prévus par le standard OpenCL reposent donc sur l'enregistrement de données proposé par StarPU et sur des tâches de copie supplémentaires.

StarPU ordonnance à la fois les exécutions de tâches et les transferts de données, en particulier en fonction de la bande passante entre les différentes mémoires qu'il a estimée lors de sa première exécution. Pour qu'une tâche soit exécutée par un accélérateur, 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* sont employées. Différentes stratégies de placement et d'ordonnancement des tâches sont proposées (Augonnet *et al.*, 2010). Elles prennent en compte différents critères tels que le nombre de *buffers* requis par la tâche qui doivent être transférés, la bande passante de l'accélérateur, ses performances de calcul...

4. Évaluation des performances

Nous avons évalué notre implémentation à travers trois exemples d'applications :

– un code de type AMR (*Adaptive Mesh Refinement*) représentatif des codes dont le graphe de tâches n'est pas connu à l'avance ;

– un code de génération d'images fractales représentatif des codes dont le graphe est connu à l'avance mais dont la durée d'exécution de celles-ci est très variable ;

– un code de multiplication matricielle représentatif des codes réguliers (le graphe des tâches est connu à l'avance et les durées d'exécution des tâches sont stables).

L'architecture sur laquelle les tests ont été effectués est composée d'un processeur Intel Xeon X5550 Nehalem (quad-core) cadencé à 2,67 GHz et disposant de 48 Go de mémoire auquel ont été associés trois accélérateurs NVIDIA Quadro FX5800 (4 Go de mémoire chacun). Les accélérateurs sont contrôlés par l'implémentation OpenCL fournie par NVIDIA dans le SDK CUDA 4, tandis que le processeur utilise le SDK OpenCL fourni par Intel en version 1.1.

4.1. Adaptive Mesh Refinement (AMR)

Le principe de l'AMR est assez simple : il s'agit d'effectuer un même calcul sur des données d'abord découpées grossièrement en blocs, puis, en fonction des résultats, de subdiviser certains des blocs pour y effectuer de nouveaux calculs généralement avec une meilleure précision. Cette méthode peut ensuite être appliquée de façon récursive sur ces nouveaux blocs.

Ce type de méthode est employé entre autre pour effectuer des simulations de phénomènes physiques, notamment grâce à la méthode des éléments finis. Dans ce cas, l'espace est divisé en blocs et les blocs les plus proches du phénomène sont les plus subdivisés. Cela permet de réduire la quantité de calculs nécessaires à l'obtention d'une solution au problème qui soit suffisamment fiable. L'un des inconvénients de cette méthode est qu'il est impossible de prévoir avant l'exécution du code le nombre de blocs de chaque taille qui devront être traités. Il est donc difficile d'effectuer une répartition de charge efficace. En particulier, les répartitions de charge manuelles de type « round-robin » ne sont pas adaptées et il est préférable d'avoir recours à un mécanisme de répartition de charge dynamique, ce qui est beaucoup plus compliqué à mettre en œuvre sans support exécutif.

Nous avons comparé les performances d'un code OpenCL de type AMR effectuant une répartition de charge manuelle de type « round-robin » en utilisant directement les implémentations fournies par les fabricants d'accélérateurs puis en utilisant notre implémentation – notre code reste un code OpenCL utilisant l'accélérateur virtuel regroupant les accélérateurs réels et suivant les recommandations de la section 3.2. Tout d'abord, nous avons mesuré la durée d'exécution nécessaire au traitement d'un

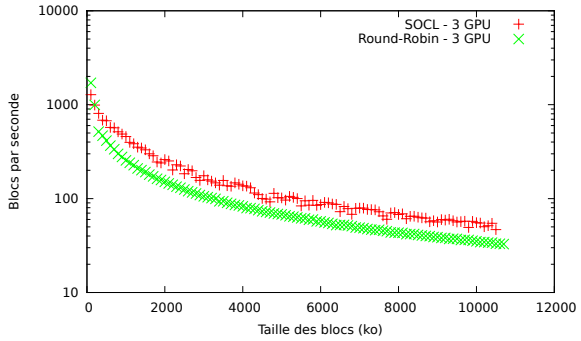


Figure 2. Nombre de blocs traités par seconde en moyenne en fonction de la dimension des blocs. Valeurs obtenues pour 25 exécutions par taille de bloc avec 30 blocs par exécution

ensemble de 30 blocs de même taille (sans subdivision), et cela pour différentes tailles de blocs. La figure 2 présente les résultats obtenus.

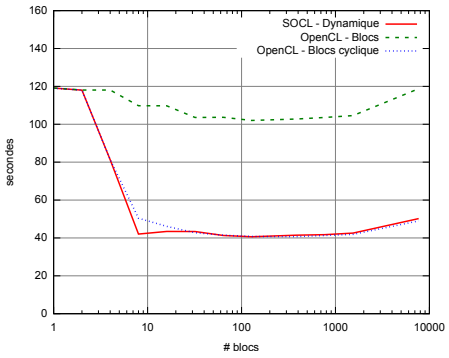
Ces résultats nous permettent d'extrapoler la durée d'exécution d'une application AMR, pour laquelle il y aurait subdivision de certains blocs, en choisissant un certain nombre de blocs de différentes dimensions. Quelques durées d'exécution extrapolées sont présentées dans le tableau 2. Les résultats obtenus avec SOCL étant meilleurs qu'avec la distribution de type « round-robin » pour quasiment toutes les tailles de blocs, nous pouvons constater que quel que soit le nombre de blocs de chaque taille, les gains de performances obtenus par notre implémentation sont importants (entre 30 et 40 % avec les nombres de blocs choisis ici). Cet exemple montre que le modèle que nous proposons est adapté aux graphes de tâches qui ne peuvent pas être connus au préalable.

Tableau 2. Estimations des durées d'exécution d'une application de type AMR

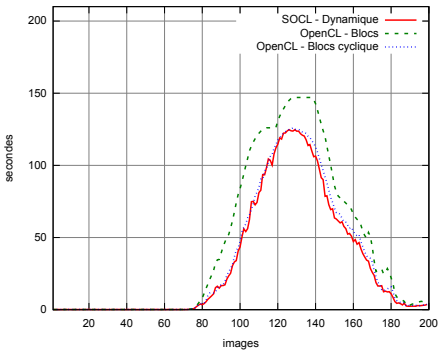
Passe 1		Passe 2		Passe 3		Durée totale d'exécution (ms)		Gain
#	taille	#	taille	#	taille	Round-Robin	SOCL	
100	5 Mo	30	2.5 Mo	10	500 ko	1769	1253	29.2 %
100	5 Mo	100	2.5 Mo	100	500 ko	2612	1603	38.6 %
100	10 Mo	60	1 Mo	30	100 ko	3100	1957	36.9 %

4.2. Génération d'images fractales

Cet exemple consiste à générer une image, composée d'environ 79 millions de pixels (32 bits), représentant une portion de l'ensemble de Mandelbrot. Le nombre d'itérations nécessaires au calcul de chaque pixel est variable, donc la durée d'exécution des tâches calculant un bloc de lignes de pixels varie également (de 0.8 millisecondes à 2.6 secondes avec notre configuration). Le graphique de la figure 3a indique



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



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

Figure 3. Génération d'images fractales de l'ensemble de Mandelbrot (3 GPU)

le temps nécessaire pour générer une image en fonction du nombre de tâches, donc du découpage de l'image en blocs de lignes. L'image fractale générée pour ce test a été choisie de façon à ce qu'il y ait un fort déséquilibre : seule la moitié supérieure de l'image contient 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 (3 GPU).

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 de meilleurs résultats que les autres distributions. En particulier, bien que faibles, les gains de performances par rapport à la distribution par blocs cyclique sont encourageants, cette dernière distribution étant, *a priori*, la meilleure décomposition manuelle possible pour ce type de problème, car considérant la nature des images, il est rare de rencontrer des cas où les pixels nécessitant le plus d'opérations soient majoritairement affectés à un même accélérateur avec cette distribution.

4.3. Multiplication matricielle

La multiplication matricielle est au cœur de nombreux algorithmes de calcul numérique. C'est un bon exemple de problème régulier car les blocs de la matrice ré-

sultat peuvent être calculés indépendamment les uns des autres et les blocs de mêmes dimensions nécessitent le même nombre d'opérations flottantes.

Le code utilisé pour ce test effectue une multiplication matricielle entre deux matrices de nombres flottants en simple précision générées de façon aléatoire. La parallélisation a été effectuée de sorte que pour calculer $C = A \times B$, chaque tâche prend la totalité de la matrice B et un bloc de lignes des matrices A et C en paramètres.

La figure 3 présente les performances obtenues en fonction de la configuration matérielle (architecture homogène avec uniquement les 3 GPU ; architecture hétérogène avec les 3 GPU et le CPU) pour une taille de bloc fixée à 64 lignes. Le code du *kernel* étant très naïf, les performances sont très en deçà de ce qu'on pourrait attendre d'un code optimisé sur cette architecture. Néanmoins, ce test montre que sur une architecture hétérogène notre support exécutif obtient des performances légèrement supérieures à une distribution par blocs cyclique, pourtant la plus adaptée à ce type de problème régulier. Cela est vraisemblablement dû au fait que si les accélérateurs sont identiques, ils partagent néanmoins un même lien PCI-Express qui peut être sujet à de la contention, influant ainsi sur les durées des transferts de données. Le support exécutif s'adapte à ces congestions contrairement à la répartition manuelle.

Tableau 3. Multiplication matricielle entre deux matrices de dimensions $16k \times 64k$ *et* $16k \times 16k$*, simple précision, blocs de 64 lignes*

	3 GPU	3 GPU + 1 CPU
Round-Robin (naïf)	440 GFlops	30.3 GFlops
SOCL	459 GFlops	546.8 GFlops

Nous avons ensuite testé la portabilité de ce code sur une architecture hétérogène. Pour cela, nous avons activé l'utilisation de l'implémentation OpenCL pour CPU fournie par Intel. Les trois GPU et le CPU sont alors utilisés simultanément. Les résultats obtenus avec la répartition manuelle sont alors très bas car la charge du CPU est beaucoup trop importante et ralentit inutilement le calcul. Pour pallier ce problème, il faudrait modifier le code de l'application pour que la répartition soit plus équilibrée entre les GPU et le CPU. Cependant cet équilibre est difficile à trouver. À l'inverse, en utilisant notre support exécutif, celui-ci s'adapte automatiquement à la relative lenteur du CPU ce qui lui permet d'obtenir des performances supérieures à celles obtenues avec les GPU seuls.

Nous avons ainsi une illustration de la portabilité des codes utilisant notre implémentation. Sous réserve que les applications créent un nombre suffisant de tâches, le support exécutif se charge de les répartir pour exploiter au mieux les capacités de l'architecture, même si l'architecture est hybride. Aucune intervention du programmeur n'est requise lors du passage d'une architecture à l'autre.

5. Conclusion et perspectives

OpenCL est l'une des solutions actuelles permettant de programmer pour des architectures hybrides composées de multicœurs généralistes et d'accélérateur(s). L'exécution concurrente de tâches sur les différentes unités (CPU ou accélérateurs) requiert actuellement de l'utilisateur un effort important de placement, pour occuper chaque unité au mieux.

Nous avons montré que la chaîne de compilation/d'exécution d'OpenCL pouvait être complétée par un support exécutif afin de gérer automatiquement ce placement sur machine hybride. Pour cela, nous avons proposé un accélérateur virtuel, utilisé par OpenCL et pouvant exécuter des tâches en parallèle. Cet accélérateur virtuel correspond aux différents accélérateurs réels et aux multicœurs généralistes. Le rôle du support d'exécution est alors de gérer les mémoires locales des accélérateurs, d'optimiser les transferts (prefetch, affinité entre tâches) tout en équilibrant la charge entre les différentes unités. Outre le placement dynamique, l'ajout d'un support exécutif permet au programmeur OpenCL de ne plus devoir spécifier les communications explicitement.

En utilisant le support exécutif StarPU (Augonnet *et al.*, 2010), nous avons montré sur deux exemples les gains apportés par une telle approche. Pour un code de maillage adaptatif (AMR, *adaptive mesh refinement*) où le nombre de tâches et leur grain changent au cours de l'application, le gain en performance sur une architecture avec 3 GPU et un multicœur est très significatif par rapport à un code où le placement manuel des tâches ne se fait que sur les 3 GPU (homogènes). Ces gains proviennent d'une part de l'utilisation conjointe (mais pas à parts égales) des multicœur et des GPU, et d'autre part de l'equilibrage de charge dynamique réalisé par le support runtime. Les bénéfices de cet équilibrage sont montrés par le second exemple, le produit matriciel, où là encore notre approche donne des gains significatifs.

Ce travail nous permet d'envisager de nouvelles améliorations avec pour objectif à la fois de simplifier la programmation d'applications tout en améliorant les performances. Il est désormais possible d'envisager de nouvelles optimisations, en particulier en travaillant sur le graphe de tâches avant qu'il soit envoyé au support exécutif sous-jacent. Vu la nature hétérogène des unités de calcul, il serait souhaitable d'adapter la granularité des tâches en fonction des capacités des accélérateurs présents. Pour cela, des extensions OpenCL facultatives pourraient être développées afin de permettre au support exécutif de faire cette adaptation.

Le logiciel SOCL est maintenant intégré à la distribution du support exécutif StarPU : https://gforge.inria.fr/projects/starpu/

Bibliographie

Agullo E., Augonnet C., Dongarra J., Faverge M. *et al. (2010).* LU Factorization for Accelerator-based Systems. *Rapport technique.*

Augonnet C., Thibault S., Namyst R., Wacrenier P.-A. (2010). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience. *Consulté sur http://hal.inria.fr/inria-00550877/en/*

Ayguadé E., Badia R. M., Igual F. D., Labarta J. et al. (2009). An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th international euro-par conference on parallel processing*, p. 851–862. Berlin, Heidelberg, Springer-Verlag. Consulté sur http://dx.doi.org/10.1007/978-3-642-03869-3_79

Bondhugula U., Hartono A., Ramanujam J., Sadayappan P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 acm sigplan conference on programming language design and implementation*, p. 101–113. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/1375581.1375595

Chen L., Liu L., Tang S., Huang L. *et al. (2011). Unified parallel c for gpu clusters: language extensions and compiler implementation. In* Proceedings of the 23rd international conference on languages and compilers for parallel computing*, p. 151–165. Berlin, Heidelberg, Springer-Verlag.*

ClearSpeed. (2010). Runtime user manual, version 3.1. *http://support.clearspeed.com/documentation/software/.*

Cunningham D., Bordawekar R., Saraswat V. (2011). GPU Programming in a High Level Language - Compiling X10 to CUDA. In ACM SIGPLAN 2011 X10 Workshop.

Diamos G., Yalamanchili S. (2008). Harmony: A Flexible Runtime for Heterogeneous Many Core Architectures. In Acm/ieee international symposium on high performance distributed computing.

Diamos G. F., Yalamanchili S. (2008). Harmony: an execution model and runtime for heterogeneous many core systems. In Proceedings of the 17th international symposium on high performance distributed computing*, p. 197–200. New York, NY, USA, ACM.*

Dolbeau R., Bihan S., Bodin F. (2007, octobre). HMPP: A hybrid Multi-core Parallel Programming Environment. In Workshop on General Purpose Processing Using GPUs.

ET International, Inc. (2011). SWift Adpative Runtime Machine(SWARM). *(http://www.etinternational.com/index.php/products/swarmbeta/)*

Fatahalian K., Knight T. J., Houston M., Erez M. et al. (2006, nov.). Sequoia: Programming the memory hierarchy. In *Sc 2006 conference, proceedings of the acm/ieee*, p. 4.

Grewe D., O'Boyle M. F. (2011). A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Cc '11: Proceedings of the 20th international conference on compiler construction.* Springer.

HPC-Project. (2012). *Par4All.* http://www.par4all.org/documentation/.

Khronos OpenCL Working Group. (2010). *The OpenCL Specification, Version 1.1.*

Lee J., Tran M. T., Odajima T., Boku T. *et al. (2011). An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters. In* HeteroPar'11 Proceedings.

NVIDIA Corporation. (2011). NVIDIA CUDA SDK 4.0. *http://developer.nvidia.com/cuda-toolkit-40.*

PathScale. (2012). ENZO. *http://www.pathscale.com/enzo/.*

Saraswat V., Bloom B., Peshansky I., Tardieu O. et al. (2011). *X10 Language Specification, version 2.2.*

Sidelnik A., Garzarán M. J., Padua D., Chamberlain B. (2011). *Using the High Productivity Language Chapel to Target GPGPU Architecture.*

Reçu le 26 septembre 2011
Accepté le 21 juin 2012

Sylvain Henry. *Sylvain Henry est doctorant au LaBRI au sein de l'équipe Runtime (Inria Bordeaux - Sud-Ouest). Il s'intéresse aux modèles et aux langages de programmation ainsi qu'à leurs supports exécutifs dans le cadre du calcul haute performance. Il a obtenu un diplôme d'ingénieur à l'ENSEIRB en 2009.*

Alexandre Denis. *Alexandre Denis est chargé de recherches à l'INRIA Bordeaux Sud-Ouest depuis 2004, membre de l'équipe INRIA Runtime. Il est ancien élève de l'École Normale Supérieure de Lyon et a obtenu un doctorat en Informatique de l'Université de Rennes 1 en 2003. Il s'intéresse aux domaines des sytèmes parallèles et du calcul haute performance. Ses travaux actuels concernent les infrastructures de communication réseau pour les grappes de calcul et les grilles.*

Denis Barthou. *Denis Barthou est professeur d'informatique à l'Institut Polytechnique de Bordeaux depuis 2009, membre du LaBRI dans l'équipe INRIA Runtime. Il a obtenu un master d'informatique théorique à l'Ecole Normale Supérieure de Lyon en 1993. Il a soutenu sa thèse en 1998 sur l'analyse de flot de données non-affine a l'Université de Versailles St Quentin sous la direction de P. Feautrier. Ses principaux centres d'intérêt en recherche portent sur le calcul haute performance, notamment la mise au point des performances et le logiciel MAQAO, la reconnaissance d'algorithme, et les interactions entre compilateur et runtime.*