Programming Models and Runtime Systems for Heterogeneous Architectures

Sylvain Henry

Numéro d'ordre: 4899

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE DE BORDEAUX

par Sylvain HENRY

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

****

Modèles de programmation et supports exécutifs pour architectures hétérogènes

Programming Models and Runtime Systems for Heterogeneous Architectures

****

Soutenue le jeudi 14 novembre 2013

Composition du jury

Président :	M. Luc Giraud, Directeur de Recherche à Inria
Rapporteurs :	M. Jean-François Méhaut, Professeur à l'Université de Grenoble M. François Bodin, Professeur à l'Unviersité de Rennes
Examinateurs :	M. Eric Petit, Ingénieur de Recherche à l'Université de Versailles Saint Quentin
(directeur de thèse)	M. Denis Barthou, Professeur à l'Institut Polytechnique de Bordeaux
(directeur de thèse)	M. Alexandre Denis, Chargé de Recherche à Inria

Résumé en français

Le travail réalisé lors de cette thèse s'inscrit dans le cadre du calcul haute performance sur architectures hétérogènes. Pour faciliter l'écriture d'applications exploitant ces architectures et permettre la portabilité des performances, l'utilisation de supports exécutifs automatisant la gestion des certaines tâches (gestion de la mémoire distribuée, ordonnancement des noyaux de calcul) est nécessaire. Une approche bas niveau basée sur le standard OpenCL est proposée ainsi qu'une approche de plus haut niveau basée sur la programmation fonctionnelle parallèle, la seconde permettant de pallier certaines difficultés rencontrées avec la première (notamment l'adaptation de la granularité).

Mots-clés : calcul haute performance, architectures hétérogènes, supports exécutifs, OpenCL

Contexte de la recherche

L'évolution des architectures des super-calculateurs montre une augmentation ininterrompue du parallélisme (nombre de cœurs, etc.). La tendance récente est à l'utilisation d'architectures hétérogènes composant différentes architectures et dans lesquelles certains cœurs sont spécialisés dans le traitement de certains types de calculs (e.g. cartes graphiques pour le calcul massivement parallèle).

Les architectures matérielles ont influencé les modèles et les langages de programmation utilisés dans le cadre du calcul haute performance. Un consortium s'est formé pour établir la spécification OpenCL permettant de répondre au besoin de disposer d'une interface unifiée pour utiliser les différents accélérateurs. Cette spécification s'inspire notamment de l'interface CUDA développée par NVIDIA pour ses propres accélérateurs graphiques. Depuis lors, de nombreux fabricants fournissent une implémentation OpenCL pour piloter leurs accélérateurs (AMD, Intel, etc.).

L'écriture d'applications utilisant l'interface OpenCL ou les autres interfaces de bas niveau est notoirement difficile, notamment lorsqu'on cherche à écrire des codes qui puissent être portables entre différentes architectures hétérogènes. C'est pourquoi la tendance actuelle est au développement de supports exécutifs (*runtime systems*) permettant d'automatiser la gestion des ressources matérielles (mémoires, unités de calcul, etc.). Ces supports exécutifs s'utilisent par le biais de modèles de programmation de plus haut niveau, notamment par la création explicite de graphes de tâches. Cependant, aucun standard pour ces modèles de programmation n'a encore émergé.

Lors de cette thèse, en nous inspirant des supports exécutifs existants, nous nous sommes intéressés au développement de supports exécutifs plus avancés à différents points de vue : simplicité du modèle de programmation fourni à l'utilisateur, portabilité des performances notamment par le support de l'automatisation de l'adaptation de la granularité des tâches, support des interfaces de programmation reconnues (dans notre cas l'interface OpenCL), etc.

Démarche adoptée

Deux approches ont été menées en parallèle lors de cette thèse : la première consistait à partir du standard OpenCL qui propose un modèle de programmation standard de bas niveau (ges-

tion explicite des mémoires et des noyaux de calcul par l'utilisateur) et à l'étendre de façon à intégrer des mécanismes de gestion de la mémoire et d'ordonnancement des noyaux automatisés. La seconde approche consistait à ne pas nous imposer de contrainte sur le modèle de développement fourni à l'utilisateur de façon à pouvoir proposer celui qui nous semblerait le plus adapté au but poursuivi.

Extension du standard OpenCL

Nous avons développé notre propre implémentation de la spécification OpenCL ayant la particularité de ne piloter aucun accélérateur directement mais d'utiliser les autres implémentations OpenCL disponibles pour cela. Afin de fournir les mécanismes existants dans certains supports exécutifs (gestion de la mémoire et ordonnancement des noyaux de calculs automatiques) nous avons choisi de faire utiliser le support exécutif StarPU par notre implémentation, d'où son nom SOCL pour StarPU OpenCL. Tout d'abord nous avons fourni une plateforme unifiée : avec OpenCL, les implémentations fournies par les fabricants d'accélérateurs ne peuvent interagir entre elles (synchronisations, etc.) alors que par l'intermédiaire de SOCL elles peuvent. Pour cela, toutes les entités OpenCL ont été encapsulées dans SOCL et les mécanismes manquants ont pu être ajoutés.

Afin d'ajouter le support automatique de la gestion de la mémoire distribuée, nous avons fait en sorte que chaque *buffer* OpenCL alloué par l'application avec SOCL alloue une *Data* StarPU : lorsqu'un noyau de calcul utilise un de ces *buffers*, StarPU se charge de le transférer dans la mémoire appropriée. De la même façon, les noyaux de calculs (*kernels*) créés à partir de SOCL sont associés à des *codelets* StarPU puis à des tâches StarPU lorsqu'ils sont exécutés. De cette façon, SOCL bénéficie des mécanismes d'ordonnancement automatique des noyaux de calculs fournis par StarPU. Enfin, les contextes OpenCL (i.e. ensembles d'accélérateurs) sont associés aux contextes d'ordonnancement de StarPU de sorte qu'il est possible de contrôler précisément l'ordonnancement automatique fourni par SOCL à travers un mécanisme préexistant dans OpenCL.

Approche par programmation fonctionnelle parallèle

Les approches actuelles pour la programmation des architectures parallèles hétérogènes reposent majoritairement sur l'utilisation de programmes séquentiels auxquels on adjoint des mécanismes pour définir du parallélisme (typiquement par le truchement de la création d'un graphe de tâches). Nous nous sommes intéressés à une approche différente dans laquelle on utilise un modèle de programmation intrinsèquement parallèle : modèle fonctionnel pur avec évaluation en parallèle. Notre choix s'est porté sur ce modèle du fait de sa similitude avec la programmation d'architectures hétérogènes : les noyaux de calculs sont des fonctions pures (sans effets de bords autres que sur certains de leurs paramètres), les dépendances entre les noyaux de calculs décrivent le plus souvent des dépendances de type flot de données. Nous avons donc proposé un modèle fonctionnel parallèle et hétérogène dans lequel on combine des noyaux de calculs écrits avec des langages de bas niveau (OpenCL, C, Fortran, CUDA, etc.) et un langage de coordination purement fonctionnel.

Pour notre première implémentation, nommée HaskellPU, nous avons combiné le compilateur GHC avec le support exécutif StarPU. Cela nous a permis de montrer qu'il était possible

de créer un graphe de tâches à partir d'un programme fonctionnel et qu'il était possible de modifier statiquement ce graphe en utilisant les règles de réécriture de GHC. Afin de mieux contrôler l'évaluation parallèle des programmes fonctionnels et l'exécution des noyaux de calculs sur les accélérateurs, nous avons conçu une seconde implémentation nommée ViperVM totalement indépendante de GHC et de StarPU.

Résultats obtenus

Notre implémentation du standard OpenCL montre qu'il est possible de l'étendre pour améliorer la portabilité des applications sur les architectures hétérogènes. Les performances obtenues avec plusieurs applications que nous avons testées sont prometteuses. Toutefois cette approche a ses limites, notamment à cause du modèle de programmation proposé, et il nous a été difficile d'implémenter un mécanisme permettant l'adaptation automatique de la granularité des noyaux de calculs.

L'approche de haut niveau basée sur la programmation fonctionnelle hérite de tous les travaux sur la manipulation de programmes dans le formalisme de Bird-Meertens en particulier et de ce point de vue est beaucoup plus prometteuse. Notre implémentation montre que les performances obtenues sont du même ordre que celle obtenues avec les supports exécutifs existants, avec l'avantage supplémentaire de permettre des optimisations à la compilation et à l'exécution.

Ces travaux ont donné lieu à différentes publications en conférence (RenPAR'20), en *workshop* (FHPC'13) et dans un journal (TSI 31) ainsi qu'à un rapport de recherche Inria.

Résumé en anglais

This work takes part in the context of high-performance computing on heterogeneous architectures. Runtime systems are increasingly used to make programming these architectures easier and to ensure performance portability by automatically dealing with some tasks (management of the distributed memory, scheduling of the computational kernels...). We propose a low-level approach based on the OpenCL specification as well as a high-level approach based on parallel functional programming.

Keywords: high-performance computing, heterogeneous architectures, runtime systems, OpenCL

Cette thèse a été préparée au sein du laboratoire Inria Bordeaux - Sud-Ouest dans l'équipe Runtime dirigée par le professeur Raymond Namyst.

Remerciements

Comme il est d'usage de remercier toutes les personnes qui ont compté sur la période de réalisation de cette thèse, en guise de prolégomènes j'aimerais faire des excuses anticipées à l'endroit de celles et ceux que j'aurais pu oublier. Je me rattraperai dans la prochaine.

Je dois à Yves Métivier de m'avoir suggéré de considérer poursuivre en thèse après l'obtention de mon diplôme d'ingénieur. Qu'il en soit remercié car, sans son conseil, je n'aurais pas trouvé ma voie (en tout cas pas tout de suite). Je remercie Raymond Namyst de m'avoir accueilli au sein de son équipe et de m'avoir trouvé un financement (public), ce qui relevait de la gageure. Je remercie mes deux encadrants, Denis Barthou et Alexandre Denis, pour le temps qu'ils m'ont consacré et pour leur soutien *in fine* à la piste de recherche que je souhaitais explorer – utilisant la programmation fonctionnelle – qui est pourtant perçue comme en opposition avec l'objectif de haute performance poursuivi. Enfin, je souhaite remercier chaleureusement les autres membres du jury, Jean-François Méhaut, François Bodin, Luc Giraud et Eric Petit, qui ont tous pris le temps de relire le manuscrit et m'en ont fait des retours constructifs.

Lors de ces quatre années au sein du laboratoire INRIA[1], j'ai été amené à rencontrer et côtoyer de nombreuses personnes que j'aimerais ici remercier. Tout d'abord les membres permanents de l'équipe Runtime : Sylvie, Raymond, qui a toujours une histoire sympa à raconter, Pierre-André, Emmanuel, Samuel, Olivier, Nathalie, merci pour tes conseils pour ma soutenance, Marie-Christine, qui m'a incité à positiver un peu plus, Guillaume et Brice, qui devraient s'installer de façon permanente dans l'open-space des thésards à la fois pour l'animation et la température. Je tiens également à remercier les anciens doctorants de l'équipe qui m'ont passé le relais : Broq, Paulette, les "bip-bip café" se sont arrêtés avec ton départ, Jérôme, Cédric, Louis-Claude, Stéphanie, tu avais raison à propos de l'interpellation "Alors la rédaction ?".

De nombreux collègues sont devenus des amis : François, avec ses chatons, ses absences et ses nombreux liens NSFW, Bertrand, préparateur sportif, fan de mamies à vélo et jamais avare d'une contrepèterie, Paul-Antoine, notre référence en linguistique anglaise et française, pratiquant un sport de collégien sans en avoir honte (CO), Cyril, globe-trotter cinéphile, capable de faire Agen–Villeneuve-sur-Lot à pied dans la boue et de nuit et maître de la réalisation de scripts divers et variés, Andra, qui a enduré une équipe masculine de thésards à l'humour léger et qui n'est pas la dernière à proposer d'aller boire des bières, Sébastien, adepte de la crasse mitraillette et des fruits de mer et d'une discrétion de violette de façon générale. Courage à vous, si je l'ai fait, vous pouvez le faire !

La liste serait incomplète si j'oubliais Manu auprès duquel j'ai appris de nombreuses choses notamment sur l'engagement, la philo, etc. Merci pour tous les conseils et toutes les sorties également. Géraldine, qui est toujours de bon conseil (e.g. préparer sa soutenance bien en avance) et à qui je dois beaucoup, notamment qui m'a sauvé la mise pour l'organisation de mon pot de thèse et qui m'a encouragé et reboosté lorsque j'en avais besoin (sans parler de nos goûts musicaux communs :)). Aurélien, qui m'a fait découvrir Mano Solo et les joies de la randonnée dans les Encantats. Merci pour tous les conseils avisés également. Merci également à Abdou, Pascal, Mathieu, Damien et George pour toutes les restaurants, les barbecues et autres sorties. Merci à Ludo pour son soutien dans notre "croisade" contre l'obscurantisme impératif

[1] Transformé depuis 2011 en Inria "Inventeurs du monde numérique" (cf "No Logo" de Naomi Klein sur le concept du *branding*)

et pour toutes les discussions intéressantes. Merci à François du SED pour sa bonne humeur et nos discussions. Enfin, merci à Antoine R., bon courage à Zurich, Corentin, Yannick, Cyril R., pour les discussions sur la place du travail dans la société et pour l'humour toujours fin et distingué, Yohan, pour la découverte de la gastronomie réunionnaise, Nicolas, Marc, Julie, et son obsession pour les chaussures, Astrid, la parachutiste, Laëtitia, Aude et Lola, pour leur bonne humeur, Andres, pour toutes nos discussions lorsqu'on partageait le même bureau et par la suite, Julien, Pierre, Allyx, la musique dans la peau...

En dehors du laboratoire, je souhaite remercier tous ceux qui ont rendu mon séjour bordelais et, plus globalement, ces quatre années, plus agréables. Matthieu, Philippe, Benoit, Mellie, Jérémie et Pierre, compagnons d'infortune de classe prépa et qui en sont restés de véritables amis. Simon & Cécilia, Claire, Stéfanie & Michel, Cédric & Marie, Rémi & Laure, Rémi & Frédérique, Benoit & Christelle, Cyril & Sarah, amis rencontrés à l'ENSEIRB. Le groupe du Bassin : Caroline, Nelly & Nicolas, Pauline, Clémentine, Laurie, Quentin, Antoine, Eric, François, Laëtitia, sans oublier Marlène. Tous ceux avec qui j'ai fait de la musique : Romain, Mathieu, François, Benjamin, Antoine, Xavier, David R., Myriam, David J., Alexandra, Stéphanie... Tous les camarades et amis Francisco, Anne, Bertrand, Brigitte L., Brigitte D., Marie-Claude, Maïa, Pierre, Édouard, Grégoire, Jean-Claude, Yves, Yannick... Beaucoup de souvenirs impérissables (congrès, etc.). Merci également à Irina, tu me dédicaceras ton livre ?, Alexandre, tu vois maintenant qu'on avait bien raison, Jeoffrey, courage pour la fin de ta thèse, Mélanie, Charlotte, Ophélie...

N'étant pas excessivement démonstratif d'ordinaire, je tiens à profiter de l'occasion pour remercier ma famille, mon frère et ma soeur et en particulier mes parents qui ont supporté que je passe une grande partie de mon adolescence à lire de nombreux livres de programmation et à passer un temps incommensurable à mettre en pratique, souvent jusqu'aux petites heures du matin, parfois au détriment d'autres activités, et qui m'ont apporté leur soutien indéfectible durant cette thèse comme toujours auparavant. J'en profite également pour les remercier d'avoir soutenu, encouragé et enduré ma deuxième passion, légèrement plus bruyante, à savoir la pratique de la batterie...

Enfin je remercie ceux qui sont une source d'inspiration et de motivation pour moi et qui ne doivent pas se trouver souvent cités, si jamais, dans les remerciements d'une thèse en informatique : Hiromi Uehara, Dave Weckl, Chick Corea, Simon Phillips, Vinnie Colaiuta, Sting, Jean-Luc Mélenchon, Jacques Généreux...

CONTENTS

INTRODUCTION

Scientific applications require a huge amount of computing resources. In particular, some applications such as simulations of real world phenomena can increase the precision of their results or widen the domain of their simulation so that they can use as many computing resources as can be made available. Hence high-performance computer architectures are consistently improved to sustain the computing needs.

During a substantial period of time, thanks to advances in miniaturization, architectures have been enhanced by increasing their clock speeds and by adding mechanisms to alleviate the slow-downs induced by the slowest components (typically memory units). With a minor impact on application codes, newer architectures were able to provide enhanced performance. However this process has come to an end because further improvements of the same nature would come at too much a price to pay, especially regarding power consumption and heat (the "Power wall"). Instead, the trend has become to conceive architectures composed of simpler and slower processing elements but concentrating a lot of them ("many-core" architectures). As these new architectures are not well-suited to execute common applications and operating system and because they would imply performance loss of the unparalelizable sequential parts of applications (cf Amdahl's law), they are often used as auxilliary accelerators. These architectures composed of a host multi-core and some many-core accelerators are called *heterogeneous architectures* and are the subject of the work presented here. In Top500 [1] and Green500 [2] supercomputer lists of June 2013, there are 4 architectures with accelerators (INTEL Xeon Phi, NVIDIA K20x...) in both top 10 and the remaining 6 architectures are mostly IBM BlueGene/Q which have followed the same trend.

Programming heterogeneous architectures is very hard because codes executed on accelerators (that we will refer to as *computational kernels* or *kernels* in the remainder of this document) are often intrinsically difficult to write because of the complexity of the accelerator architecture. In addition, writing the code to manage the accelerators (*host code* executed by the host multi-core) is very cumbersome, especially because each accelerator has its own constraints and capabilities. Host codes that use low-level frameworks for heterogeneous architectures have to explicitly manage allocations and releases in each memory as well as transfers between host memory and accelerator memories. Moreover, computational kernels must be explicitly compiled, loaded and executed on each device without any help from the operating system.

Runtime systems have been developed as substitutes for missing operating system features such as computational kernel scheduling on heterogeneous devices and automatic distributed

memory management. However, there has been no convergence to a standard for these runtime systems and only two specifications, namely OpenCL and OpenACC, are widely recognized despite them being very low-level (OpenCL) or of limited scope (OpenACC). As a consequence, we decided to use OpenCL specification as a basis to implement a runtime system (called SOCL and presented in Chapter II) that could be considered as a set of OpenCL extensions so that its acceptance factor is likely to be higher than other runtime systems providing their own interfaces. It provides automatic distributed device memory management and automatic kernel scheduling extensions as well as a preliminary support for granularity adaptation.

The programming model used by SOCL, thus inherited from OpenCL and based on a graph of asynchronous commands, makes the implementation of some optimizations such as automatic granularity adaptation very hard. This difficulty is shared with some other runtime systems that use a programming model based on graphs of tasks such as StarPU or StarSS. In particular, most of them lack a global-view of the task graph because the latter is dynamically built, hence inter-task transformations cannot be easily implemented as runtime systems cannot predict their impact on tasks that will be submitted later on. A solution is to use a more declarative language to build the task graph to let the runtime system analyze it so that transformations can be performed more easily. Frameworks such as DaGUE [28] use a dependency analysis applied to an imperative loop nest to infer task dependencies and store the task graph in a form more easily usable by the runtime system.

In Chapter III we present a solution alleviating parallel functional programming so that we do not need a dependency analysis nor an intermediate form to store task graphs. We show that this approach has a great potential because it is very close to both state-of-the-art programming paradigms used by runtime systems for heterogeneous architectures and high-level functional programming languages. This convergence of two distinct research domains – programming languages and high-performance computing – is very inspiring. The specific characteristics of high-performance programs such as the lack of user interaction during the execution and the fact that they consist in transforming a (potentially huge) set of data into another set of data make them an ideal fit for functional programming.

In this last chapter, the work we describe is still in progress. We show that the solution we propose combines several previous works such as parallel functional programming, computational kernel generation from high-level data-parallel operators, transformation of functional programs as in the Bird-Meertens calculus, algorithmic choice... The automatic granularity adaptation method we propose uses several of these mechanisms and is based on heuristics that would need to be evaluated in real cases. Hence we need a complete implementation of a runtime system supporting our approach, which we did not have yet. Completing the implementation of this integrated environment is our next objective. One of the major design constraint we have established is to make it easily extendable to facilitate collaborations between people from different communities (scheduling, high-performance computing, programming language and type systems...) so that they could all use it as a playground to test their algorithms, as well as very easy to use for end users.

Outline and Contributions

In Chapter I, the evolution of high-performance computing architectures toward heterogeneous ones and the frameworks used to program them are presented. Generic high-level approaches that are not tied to a specific kind of architecture are also considered.

SOCL Our first contribution is an OpenCL implementation called SOCL that is described in Chapter II. It provides a unified OpenCL platform that fix many shortcomings of the OpenCL implementation multiplexer (the installable client driver) so that it is beneficial even for applications directly using the low-level OpenCL API. In addition, it offers automatic device memory management so that buffers are automatically evicted from device memory to host memory to make room in device memory if necessary. It also extends OpenCL contexts so that they become scheduling contexts into which kernels can be automatically scheduled by the runtime system. Finally, preliminary support for automatic granularity adaptation is included.

Heterogeneous Parallel Functional Programming Our second contribution, presented in Chapter III is a programming model for heterogeneous architectures. By combining the high-level parallel functional programming approach and low-level high-performance computational kernels, it paves the way to new algorithms and methods, especially to let runtime systems automatically perform granularity adaptation. ViperVM, a preliminary implementation of this programming model is also presented.

I find digital computers of the present day to be very complicated and rather poorly defined. As a result, it is usually impractical to reason logically about their behaviour. Sometimes, the only way of finding out what they will do is by experiment. Such experiments are certainly not mathematics. Unfortunately, they are not even science, because it is impossible to generalise from their results or to publish them for the benefit of other scientists.

C. A. R. Hoare

Although Fortress is originally designed as an object-oriented framework in which to build an array-style scientific programming language, [...] as we've experimented with it and tried to get the parallelism going we found ourselves pushed more and more in the direction of using immutable data structures and a functional style of programming. [...] If I'd known seven years ago what I know now, I would have started with Haskell and pushed it a tenth of the way toward Fortran instead of starting with Fortran and pushing it nine tenths of the way toward Haskell.

Guy Steele, Strange Loop 2010 Keynote

CHAPTER I

TOWARD HETEROGENEOUS ARCHITECTURES

Chapter abstract

In this chapter, we present the evolution of high-performance architectures towards heterogeneous ones that have become widespread in the last decade. We show how imperative programming approaches that are based on the Von Neumann architecture model have been adapted to deal with the new challenges introduced by each kind of architecture. Finally, we present models and languages that drop the lineage with the Von Neumann model and propose more radical approaches, considering the root of some of the issue is the use of this model. A comprehensive description of the architecture evolution is out of the scope

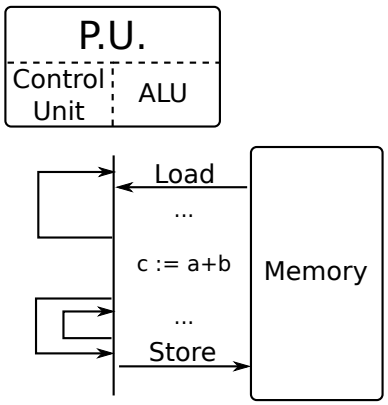


Figure I.1: Von Neumann architecture model

of this document. Here we mainly stress on the breakthroughs that have led to the current situation and the foreseen future of which heterogeneous architectures are very likely to be part of.

I.1 Introduction

Thanks to miniaturization, the number of transistors that can be stored initially on integrated circuits and later on microprocessors keeps increasing. This fact is commonly referred to as Moore's law. Indeed, in 1965 Gordon Moore has conjectured that the number of transistors would double approximately every two years and this conjecture has been empirically verified. As a consequence, architectures have evolved in order to advantage of these new transistors.

The architecture model that has prevailed and that is at the root of most programming languages still in use today (e.g. imperative languages) is referred to as the Von Neumann model. Architectures based on this model are composed of two main units as shown in Figure I.1: a memory unit and a processing unit. In this basic model, instructions are executed in sequence and can be of the following types: memory access (load or store), arithmetic operation, control instruction (i.e. branches). Actual architectures are of course more complex than the Von Neumann model. Memory hierarchies have been introduced to speed-up memory accesses. Multicore architectures contain tens of processing units and several memory units. Processing units themselves contain more execution units (SIMD units, floating-point units...), execute instructions in parallel and at much higher clock frequencies. Finally, clusters interconnect thousands of these architectures together through high-performance networks.

The complexity of these architectures has been pushed as far as possible to enhance performance without having to change programs much until the *power-wall* has been faced: energy consumption and temperature rise have become too high to continue this way. The alternative direction that has been taken instead consisted in conceiving architectures containing a very large amount of comparatively simple processing units with a very low power consumption. This way, the degree of parallelism has been increased to thousands of simultaneous instruction executions inside a single processor, hence these architectures are often referred to as *many-core* architectures. The price to pay for this architecture design shift has been the

necessity to write specific codes able to take advantage of so many threads.

Most many-core architectures are not designed to handle tasks such as user interaction or controller management (disks, etc.). Hence, many-core architectures are often used on devices used as accelerators and interconnected with a multi-core that is able to perform the aforementioned tasks. Heterogeneous architectures are these architectures composed of a multi-core alongside an arbitrary number of accelerators of different kinds. The main topic of this document is the programming of these heterogeneous architectures, not of a specific one in particular but the coordination of codes executed on accelerators and on the host as well as memory management.

I.2 Multi-Core and Clusters Programming

Programming languages, frameworks and runtime systems used for high-performance computing are often very close to the underlying architecture in the sense that they do not offer much abstraction and give full control to hardware provided mechanisms. As such programming multi-core architectures for high-performance computing is usually done by explicitly instantiating several threads that are executed independently by each processing unit. Clusters are similarly exploited by explicitly spawning at least one process per node.

Some level of abstraction is necessary to enhance code portability, productivity, etc. but the introduced abstraction mechanisms must not have a high perceptible impact on performance. As such, we can say that the rather low-level approaches presented in this section are conceived from the bottom-up (abstractions are added carefully) while in Section I.4, we present approaches conceived from the top-down (starting from a high-level of abstraction, we want to produce codes that match the underlying architecture as best as possible).

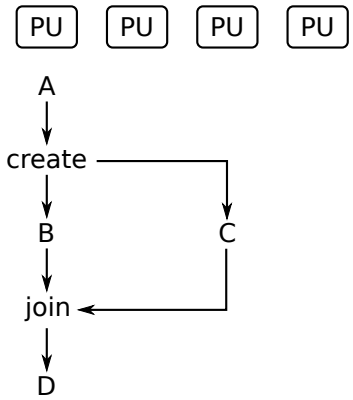
Threading APIs such as POSIX thread (PThread) API [53] are the most basic way for an application to exploit multi-core architectures. It lets applications start the execution of a function in a new thread. Both the current thread and the newly created one share global variables while local variables (typically allocated on the stack) are private to each thread. Listing I.2a shows a example of code using POSIX threads whose execution is depicted in Figure I.2b. Synchronization primitives are provided in order to let a thread wait for another one in particular (`join`), to create mutually exclusive regions that only one thread can execute at a time (`mutex`), etc.

Another thread API example is Cilk [126] that lets programmers execute C functions by different threads just by prefixing function calls with the `spawn` keyword. A work-stealing policy is then used to perform load-balancing on the different processing units [26]. A Cilk procedure, prefixed with the keyword `cilk`, can spawn other Cilk procedures and wait for their completion with the `sync` keyword.

Low-level thread APIs are quite burdensome to use, especially for programs that regularily create parallel regions (i.e. regions of code simultaneously executed by several threads). OpenMP [14] is a framework that is used to easily manage this kind of regions in C or Fortran codes. Compiler annotations are used to indicate to delimit regions that are then executed in parallel by a specified number of threads. Listing I.3a shows an example of a C code using OpenMP to create a parallel region executed by 5 threads as depicted in Figure I.3b. OpenMP provides synchronization primitives such as barriers to force all threads of a parallel region to

```c
#include <pthread.h>

void * mythread(void * arg) {
   // C
}

int main() {
   pthread_t tid;
   // A
   pthread_create(&tid, NULL, &mythread, NULL);
   // B
   pthread_join(tid, NULL);
   // D
}
```

(a) Listing



(b) Execution illustration

Figure I.2: POSIX Thread API example. The thread created by `pthread_create` executes code labelled as C in parallel with the code labelled as B executed by the main thread.

wait all of the others before continuing their execution.

As it often happens that applications have to distribute loop iterations over a pool of threads, OpenMP provide specific annotations to transform a `for` loop into a `forall` loop. The latter indicates to the compiler and runtime system that each loop iteration can be executed in parallel. In this case, the loop iterator is used to identify which loop iteration is being executed. OpenMP loop annotations can be used on loop nests producing nested parallelism: a tree of parallel threads. OpenMP implementations such as ForestGomp [32] exploit this nested parallelism to automatically enhance thread mapping on NUMA architectures.

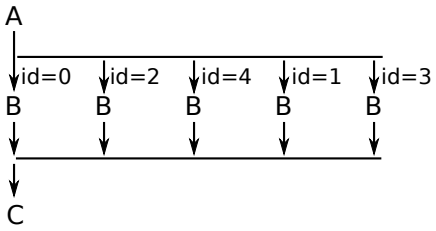
In order for applications to use more computational power and to be able to exploit bigger data sets, processors have been interconnected to form clusters. As such, an application can simultaneously uses thousands of processing units and memory units. There are numerous networking technologies such as Infiniband, Quadrics, Myrinet or Ethernet. Usually, applications do not use the low-level programming interface provided by these technologies but use higher level programming models.

Message Passing Interface (MPI) is a standardized interface that mainly allows applications to use synchronous message-passing communication. Basically, one process is executed on each node of the cluster and a unique identifier is given to each of them. Processes can exchange data by sending messages to other processes using these identifiers. The communication is said to be synchronous because messages are transmitted through the network only when both the sender and the receivers are ready. As such, messages are transferred directly to their final destination, avoiding most superfluous data copy that would be involved if the data transfer started before the receiver had indicated where the data was to be stored eventually. MPI provides synchronization primitives between processes [58].

MPI uses a mapping file to indicate where (i.e. onto which workstation) each MPI process is to be executed. MPI implementations are responsible for the correct and efficient transport

```
#include <omp.h>

int main() {
   // A
   #pragma omp parallel num_threads(5)
   {
      int id = omp_get_thread_num();
      printf("My id: %d\n", id); // B
   }
   // C
}
```

(a) Listing



(b) Execution illustration

Figure I.3: OpenMP parallel region example. Each thread of the parallel region has a different value of `id`

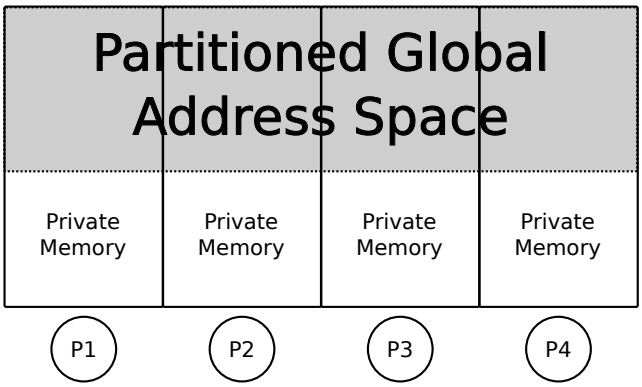
of the messages. The network topology is not exposed to applications and the implementation has to optimize the path followed by messages through the network. This is especially true for collective operations such as reductions or broadcasts which involve several nodes.

Using a synchronous message-passing model can be hard because it is easy for a programmer to introduce bugs such as a process stalled on a send operation because of a receiving process that is never ready to receive. The actor model is an alternative that allows asynchronous message-passing between processes. Basically, an actor is a process or a thread to which a message box is associated. Actors can post messages into message boxes of other actors and consult their own message boxes when they want. Charm++ uses asynchronous message passing between active objects called *chares*. Remote communication is possible by using proxy chares originally called *branch-office chares* (BOC) [84]. Erlang programming language [15] is also based on the actor model and has influenced several newer languages such as Scala that provides an actor library using a syntax borrowed from Erlang [72]. The actor model can be more amenable to implement functionalities such as fault tolerance or load-balancing. An actor can be moved from a node to another by copying its private data comprising its message box. Charm++ uses a default mapping if there is no user-provided one and then uses dynamic load-balancing strategies to move *chares* from one processing unit to another [138].

Partitioned Global Address Space (PGAS) is a shared memory model where each thread owns a portion of a virtually shared memory in addition to its private memory as shown in Figure I.4. Threads can access the following memories sorted by affinity: locale private memory, locale shared memory and remote shared memories. Many languages implement this model. On the one hand, some of them extend existing languages such as Unified Parallel C (UPC) [45] which is based on C, Co-Array Fortran (CAF) [109] and High-Performance Fortran (HPF) [57] which are based on Fortran, Titanium which is based on Java [77] and XcalableMP [106] which provides compiler annotations for C and Fortran. On the other hand, some of them are new languages dedicated to the model such as x10 [119] which is mostly influenced by Java though or ZPL and its successor Chapel [40].

Some frameworks use a *global view* instead of the local view typically used in SPMD model

Figure I.4: Partitioned Global Address Space (PGAS) memory model



where each thread does something different depending on its unique identifier. With the global view model of programming, programs are written as if a single thread was performing the computation and it is the data distribution that indicates how the parallelization is done. HPF [57] is a language based on Fortran (Fortran 90 for HPF 1.0 and Fortran 95 for HPF 2.0) that uses the global view model. It provides directives to specify data mapping on "processors" (`DISTRIBUTE` and `ALIGN`). Similarily, ZPL [39, 93] and its successor Chapel [40] are languages that use imperative data-parallel operators on arrays. Array domains (or *regions*) are first class citizens in these languages. For two arrays defined on the same domain, the runtime system only ensures that two values having the same index in the two arrays are stored in the same physical memory but the domain can be distributed among any number of cluster nodes.

Virtual address space model consists in partitioning a single address space in pages that can be physically transferred from one memory to another without modifying the virtual address space. Compared to the PGAS model, memory accesses are always performed locally but may imply preliminary page transfers. Similarly, *Single System Image* (SSI) model consists in letting a distributed operating system distribute processes and data using a virtual address space on a distributed architecture. Kerrighed, openMOSIX, OpenSSI and Plurix [59, 96] are examples of such operating systems. A mix between the virtual address space model and the PGAS model is used on NUMA architectures (cf Section I.2) that can be seen as a cluster on a chip. Basically, memory accesses in remote memories can be performed as in the PGAS model but libraries such as `libnuma` or `hwloc` [33] can be used to move a page from one NUMA node to the other.

Shared objects model can be seen as a kind of virtual address space model with different page granularities. Data (objects) are allocated in the virtual address space and an object handle is associated with each of them. Threads must use these handles to be allowed to accede to the object contents in a specific mode (read-only, write-only or read-write mode). Data may be copied into a local memory before a thread uses it and coherency is automatically maintained by a runtime system. Note that an object handle is not necessarily an address. For instance it can be an opaque object or an integer. Compared to single address space and virtual address space models, pointer arithmetic cannot be used to arbitrarily read or write memory and object handles must be used instead. Jade [115, 116] uses the shared objects model. *Local pointers* are used to temporarily get access to shared objects using a specified access mode enforced by the system. Jade targets shared memory and distributed memory architectures.

Similarly, StarPU [16, 17] also uses the shared objects model on heterogeneous architectures (cf Section I.3). Tasks are explicitly created with shared objects as parameters and task-specific access modes. Data are transferred on appropriate devices before task execution.

One of the main difficulty faced by programmers using clusters is to correctly map processes of an application to the network topology. Given a trace of the communications between processes and a description of the network topology, algorithms such as TreeMatch [83] produces an efficient mapping of processes to cluster nodes.

Because using low-level programming models for parallel computing is hard, some models and languages have introduced high-level concepts to enhance productivity. For instance, MapReduce is a model of framework which has roots in the Bird-Meertens Formalism (cf Section I.4.3) and that has been successfully applied to compute on clusters. If a computation follows a given pattern, programmers only have to give some functions to the framework and the input collection of data. The framework automatically distributes tasks on cluster nodes, performs load-balancing and may even support fault tolerance. Examples of such frameworks are Google MapReduce [50] and Hadoop [134]. Another example is Fortress [125] which has been designed to be the successor to Fortran and is mostly inspired by Java, Scala and Haskell languages. It has been conceived alongside x10 [119] and Chapel [40] for the DARPA's "High Productivity Computing System" project [98] initiated in 2002.

It is not uncommon to use several frameworks at the same time. For instance, OpenMP for intra-node parallelism using shared memory and MPI for inter-node communications using messages. In Section I.3.3, we show that by adding accelerators to some of the nodes, it is often necessary to use an additional framework such as OpenCL or CUDA or to substitute on of the framework used with another supporting heterogeneous architectures.

I.3 Heterogeneous Architectures

Programming languages, frameworks and runtime systems presented in the previous section had to be adapted to support heterogeneous architectures. In addition, newer approaches dedicated to these kinds of architectures have been developed. In this section we present several accelerators, then we show how to exploit them. We distinguish codes that are executed on the accelerators and the host code executed on a commodity multi-core and used to manage them.

I.3.1 Many-Core and Accelerators

Many-core architecture is a term coined to refer to the trend of increasing the number of cores up to tens, hundreds or even thousands of cores, in particular by simplifying the core that is duplicated. Indeed, by removing units used to speed up the basic Von Neumann architecture such as out-of-order execution management unit (Cell, ATOM, Xeon Phi…), virtual memory management unit (Cell's SPE, GPUs…), cache management units (Cell's SPE, GPUs…) and by decreasing the frequency, the number of cores that can be stored on a chip can be increased significantly. These many-core architectures often present themselves as accelerators that can be used in conjunction with a classic multi-core architecture. This heterogeneity is dealt with in the next section.

The Cell BroadBand Engine architecture (known as *Cell BE* or *Cell*) was among the first widespread heterogeneous architectures prefiguring the many-core trend. Conceived by Sony, Toshiba and IBM, it featured a modest Power4 multi-core called *Power Processing Element* (PPE) that was slower than the best architectures at the time such as the Power6. In addition, 8 cores called *Synergistic Processing Elements* (SPE) have been introduced on the chip and connected to the PPE and the memory unit through a fast ring network called *Element Interconnect Bus* (EIB) [6]. Each SPE has its own 256KB scratchpad memory called *Local Store* (LS) and an associated DMA unit to transfer data between the different LS and the main memory. The major difference of this architecture compared to previous multi-core architectures is that it requires explicit management of the SPE. Programs executed on SPE have to be compiled with a dedicated compiler and explicitly loaded at runtime by an host application executed by the PPE. Due to the limited amount of memory available on local stores, programs have to be relatively small prefiguring computational kernels used on subsequent accelerators such as GPUs. In addition, local stores do not use virtual memory units and are explicitly managed using direct addressing. DMA units have to be explicitly configured to transfer data from one memory unit to another. For the first time, a widespread architecture was not using a shared memory model (be it NUMA). Most of the forthcoming many-core architectures followed this road too.

Graphics processing units (GPU) are specialized chips that speed up graphic rendering by performing embarrassingly parallel computations such as per vertex or per pixel ones in parallel. They have been made increasingly programmable to the point that are used to perform generic scientific computing, not only graphic rendering. In the early days, only a few levels of the graphic pipeline were programmable using *shaders* through graphics API such as OpenGL's GLSL, Microsoft's HLSL and NVIDIA's Cg [101]. Now most details of their architecture are exposed through API no longer dedicated to graphics such as CUDA and OpenCL. NVIDIA GPU architectures are composed of several processing units called *Streaming Multi-Processors* (SM) and a single main memory. Each SM has its own scratchpad memory called *shared memory* as it is shared amongst every execution unit contained in the SM. Fermi and Kepler architectures introduced a L2 cache memory. Moreover they allow part of the shared memory to be configured as a L1 cache while the other part is still used as a scratchpad. Streaming multiprocessors contain bundles of execution units which have the same size. Every execution unit of a bundle executes the same instruction at the same time but on different data. Similarly, AMD GPU architectures are composed of several SIMD computation engines that differ from streaming multi-processors mostly because they contain several *thread processors* (TP) that are VLIW processors [137]. The memory hierarchy is very similar though. Using accelerators such as GPUs is hard, not only because of their intrinsic complexity but because programs have to explicitly manage data transfers between main memory and accelerator memories. Architectures embedding a CPU and a GPU on the same package have been introduced. For instance, AMD Fusion accelerated processing units (APU) such as Bulldozer or Llano use a unified north bridge to interconnect a CPU core and a GPU core.

Intel's answer to the rise of high performance computing using GPUs has been to provide accelerators based on x86 ISA. It started with the *Single-chip Cloud Computer* (SCC) prototype, a Cluster-on-Chip architecture composed of 48 cores (24 dual-core processing units). Each core executes its own instance of the Linux operating system and the main memory is split so that each core has its own private region in it. A part of the main memory is shared between all the cores but the hardware does not ensure any cache coherency. Each core has a 8kB on-die shared-memory called *Message-Passing Buffer* (MPB) that can be used to implement software

cache coherence policies. Frameworks based on message-passing such as MPI can be used to program the SCC as if it was a cluster [43]. Intel *Many Integrated Core* (MIC) [46], branded as Xeon Phi, is the successor of the Larrabee prototype and is a full-fledged accelerator. It contains up to 61 cores interconnected to 8 memory controllers through a high performance bidirectional ring network. Contrary to the SCC, cache coherence is ensured on the MIC through a distributed tag directory.

Accelerators tend to share many properties with low power consumption architectures such as those used in embedded devices to the point that there is a convergence and that the same system-on-a-chip architectures can be used both for high performance computing and embedded uses (video, networking, etc.). Tilera's TILE-Gx [3] architecture can be composed of up to 72 cores interconnected through a proprietary on-chip network called *iMesh*. Kalray's MPPA 256 [4] chip contains 256 VLIW cores – 16 clusters of 16 cores – interconnected with a network-on-chip (NoC). Platform 2012 (P2012) [103], now known as STHorm, is another example of a SoC using a NoC to interconnect cores.

I.3.2 Programming Many-Core

Existing frameworks for multi-core architectures and for clusters cannot be used directly on many-core architectures such as GPUs or Cell BE. The complex memory hierarchy dismisses most frameworks for shared-memory architectures. In addition, constraints on IOs (memory allocation, etc.) and memory sizes make models such as MPI unsuitable. An exception to this is Intel Xeon Phi that can be programmed with MPI because its architecture is closer to multi-core than to a GPU.

Many-core architecture vendors often provide proprietary frameworks: IBM's Cell SDK [81], NVidia's CUDA [110], ATI's Stream SDK [11]... In addition, some frameworks that may use more abstract models target several architectures. Among the different frameworks and models, OpenCL and OpenACC are two specifications that have been endorsed by a large panel of vendors.

OpenCL [69] inherits from NVidia's CUDA and thus is especially suitable for GPU computing as the programming model matches closely the architecture model. CUDA and OpenCL let programmers write computing kernels by using an SPMD programming model similar to the one used in OpenMP. Independent groups of threads (work-groups of work-items in OpenCL) are executed in parallel. Threads of a group have access to a shared memory and can be synchronized by explicitly using barriers in the kernel code. No synchronization can occur between threads of different groups but all threads of every group have access to the same global memory[1]. Threads perform explicit transfers between global memory and their shared memory. Listing I.1 shows a simple matrix addition kernel written using OpenCL C language. Thread identifiers are obtained with the `get_global_id` built-in function. Shared memory is not used in this example.

OpenACC [71] specification defines a set of compiler annotations (*pragmas*) for C and Fortran programs similar to those of OpenMP. Instead of explicitly writing a kernel by using a SPMD model, these annotations let programmers indicate which loop nests should be converted into equivalent kernels. Hints can be given by the programmers to enhance the gener-

[1]Some OpenCL implementations allows synchronizations between groups by providing atomic operations in global memory (see OpenCL base and extended atomics extensions).

Listing I.1: OpenCL Matrix Addition kernel

```
__kernel void floatMatrixAdd(const uint width, const uint height,
   __global float * A, const uint strideA, const uint offA,
   __global float * B, const uint strideB, const uint offB,
   __global float * C, const uint strideC, const uint offC) {

  int gx = get_global_id(0);
  int gy = get_global_id(1);

  if (gx < width && gy < height) {
    C[offC + gy*strideC + gx] = A[offA + gy*strideA + gx] +
                                B[offB + gy*strideB + gx];
  }

}
```

ation of the kernels. OpenHMPP [36] is a superset of the OpenACC pragmas that influenced the OpenACC specification. Additional pragmas are used to target architectures containing several accelerators. Similarly to OpenACC and OpenHMPP, Par4All [12] is a compiler for sequential C and Fortran programs that uses polyhedral analysis on loop nests and source-to-source transformations to target other frameworks for multi-core and many-core architectures such as OpenMP, OpenCL and CUDA.

Brook [34] is an extension to C language dedicated to streaming processors which have been adapted to use GPUs as streaming co-processors. AMD's Stream SDK [11] is based on Brook. Streams of data (i.e. collections) can be defined and kernels can operate on them element-wise or reductions can be applied. Listing I.2 shows a matrix-vector multiplication $y = Ax$ written with Brook. Streams are created by using the syntax `<d0,d1...>` after a variable declaration to indicate the dimensions of the stream. x is automatically duplicated to match the size of A in order to perform `mul` kernel element-wise with results stored in T. Then T is reduced by using the `sum` kernel in the second dimension so that the result can be stored in y.

I.3.3 Programming Heterogeneous Architectures (Host Code)

Different programming models can be used to coordinate the execution of the computational kernels on the accelerators. The difficulty lies in the huge variety of heterogeneous architectures that can be encountered and that have to be supported by an application. Indeed it is common to have an architecture with several accelerators, potentially from different vendors connected to a host multi-core architecture. Each accelerator can contain a variable amount of memory, capabilities (e.g. available execution units, degree of parallelism...) may vary between the different devices and transfer bandwidths and latencies may be different between host memory and accelerator memories. A major advantage of using heterogeneous architectures is that a device with a small degree of parallelism and executed at a high frequency can be used in conjunction with massively parallel devices. The former executes the operating system

Listing I.2: Matrix-Vector Multiplication $y = Ax$ written with Brook

```
kernel void mul (float a<>, float b<>, out float c<>) {
  c = a * b;
}

reduce void sum (float a<>, reduce float r<>) {
   r += a;
}

float A<50,50>;
float x<1,50>;
float T<50,50>;
float y<50,1>;

mul(A,x,T);
sum(T,y);
```

and intrinsically sequential parts of applications while the latter are used to offload computationally intensive kernels. However this has to be handled either directly by the application in its host program or by the framework that manages the accelerators.

Using an heterogeneous architecture means having to handle different programs for different kinds of processing units. It has a major impact on the compilation chain typically used for homogeneous architectures: several programs or kernels may have to be written, several compilers may have to be used and compilations may have to be performed at execution time to ensure portability. For instance, OpenCL specification defines an API to handle kernel compilation at execution time for OpenCL compliant devices [69]. Additionally, an intermediate representation for OpenCL kernels is being specified [70] in order to make runtime compilations more efficient as some optimizations could be performed before execution time. Program loading on homogeneous architectures are transparently handled by the operating system but it is no longer the case with some heterogeneous architectures, especially those where devices have very few kilobytes of memory such as the Cell. Programs may have to be explicitly loaded into device memory and are subject to out-of-memory errors. Program loading techniques such as *code overlay* may have to be used.

The first step for a typical application to exploit an heterogeneous architecture is to discover the available accelerators and the way they are connected to the host. hwloc [33] is a generic tool that can be used by applications to gather information about an architecture: memory hierarchy, NUMA sockets, physical and logical cores, connected devices... For instance, Figure I.5 is obtained on a NUMA architecture with three NVIDIA GPUs. Each NUMA socket contains 6 cores (i.e. processing units) that use SMT so that there are two logical processing units for a physical one. Two GPUs are connected to the first socket and the other is connected to the other. Main memory is split in two parts of 24GB each. GPU memory capacity is not shown on this representation. Other frameworks for heterogeneous architectures such as OpenCL can retrieve the list of available accelerators.

DMA transfers between memory units can be affected by the topology of the interconnec-

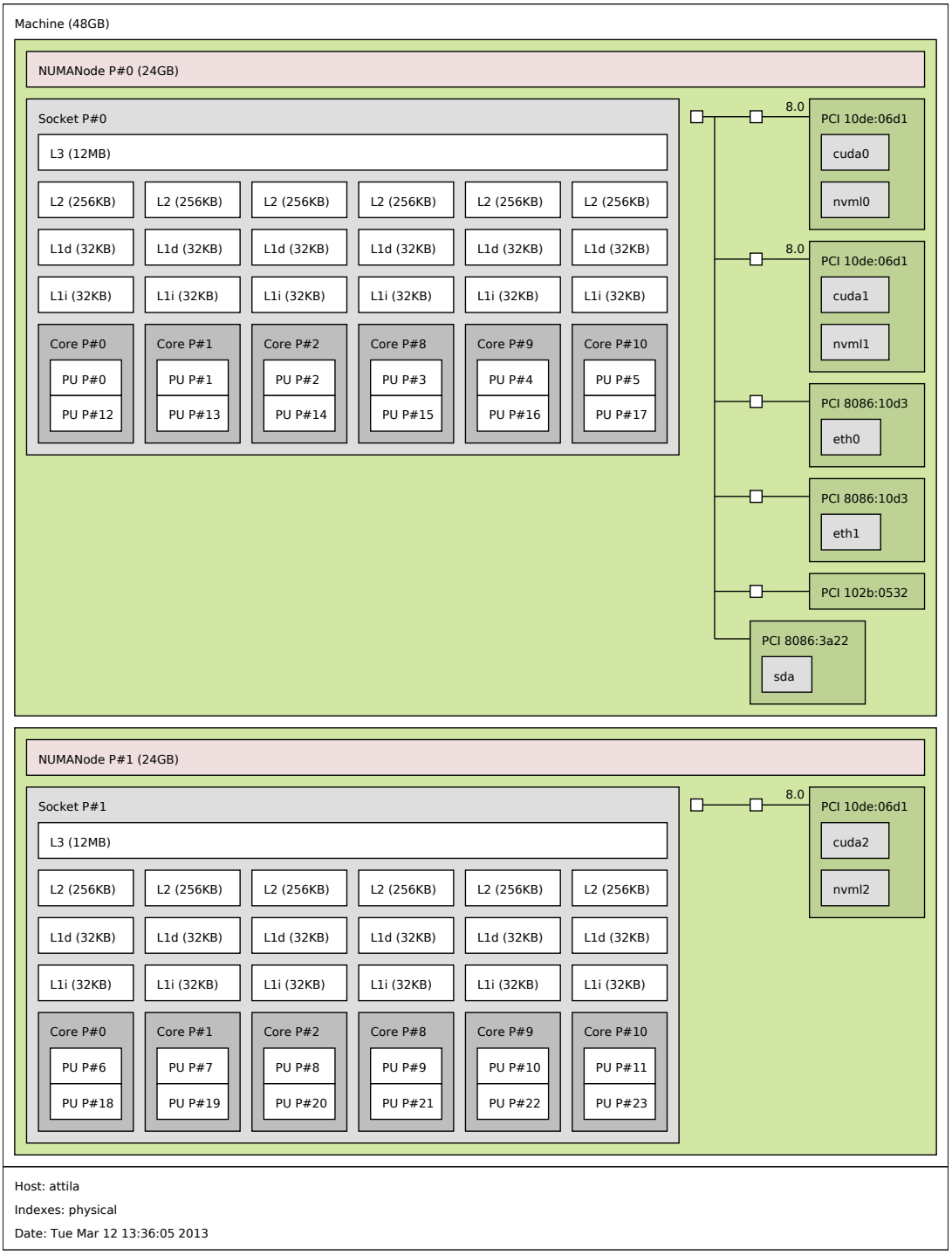


Figure I.5: NUMA architecture with three GPUs. The host is a multi-core with 12 dual-threaded cores (6 cores per NUMA socket). Two GPUs are connected to the first socket and the third is connected to the second socket, hence NUIOA factors are likely to be observed.

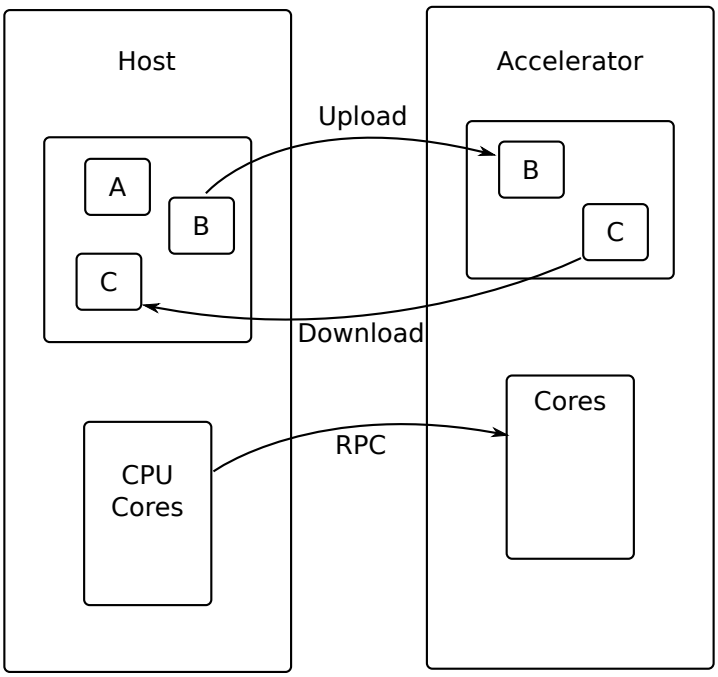


Figure I.6: Offload Model

tion network similarly to NUMA architectures. As DMA are often used to perform IO (hard disk controllers, network interface controllers...), it is called *Non-Uniform Input-Output Access* (NUIOA). In Figure I.5, network interface controllers are connected to the first socket, thus NUIOA effects are likely to be observable. Some architectures contain additional units to enhance transfers between devices. INTEL QuickData Technology [133] – part of INTEL I/O Acceleration Technology (I/OAT) – adds a DMA unit that can be used to perform DMA transfers between network interface controllers (NIC) and memory units as well as between memory units (for instance from one in a given NUMA socket to another). INTEL Direct Cache Access (DCA) technology allows controllers such as NICs to explicitly preload data into caches to speed up processing unit accesses to these data that are supposed to quickly follow. [78]

If the configuration of the platform (i.e. number and kind of devices) is known at compilation time, it is possible to statically assign codes that executed by an accelerator and codes executed by the host multi-core. For instance, Sequoia [54] is an extension for C language that has been mostly used to program the Cell. It lets programmers define functions that perform computations (*leaf tasks*) and functions that structure parallelism (*inner tasks*). For each target architecture, a mapping file is used to indicate the kind of task (leaf or inner) that will be executed by processing units corresponding to each level of the memory hierarchy.

An alternative to the static mapping of tasks on devices is to use a runtime system to dynamically schedule them. We distinguish three methods to schedule kernels executions on accelerators and data transfers between host memory and accelerator memories: offload, command graph and task graph.

Listing I.3: HMPP Basic Usage Example

```
#pragma hmpp foo codelet, target=CUDA, args[a].io=inout
void foo(int a[10], const int b[10]) {
   ...
}

int A[10], B[10];
int main(void) {
  for (int j = 0; j < 2; ++j) {
    #pragma hmpp foo callsite
    foo(A, B);
  }
  return 0;
}
```

I.3.3.1 Offloading Model

The offloading model is the one used by OpenACC and OpenHMPP. It basically supposes that a single accelerator is used, thus the memory model can be much simpler. Figure I.6 shows the memory model that is used where data can be uploaded on the accelerator and then downloaded in host memory. At some points in the host program, computations are offloaded on the accelerator, similarly to remote procedure calls (RPC). The execution can either be synchronous or asynchronous. Input data are transfered in accelerator memory before kernel execution if necessary and output data are transfered back in host memory after kernel execution if required. Listing I.3 presents an host code using HMPP to offload the function `foo` on a GPU supporting CUDA.

An issue with this model is that data transfers do not overlap with host code or kernel executions. Explicit data transfer commands or prefetch hints have to be used to get best performance. HMPP provides `advancedload` and `delegatedstore` annotations to trigger an upload data transfer in advance and to postpone a download data transfer respectively. Extensions to this model are provided to support multiple devices. HMPP lets applications choose onto which devices allocations have to be made and kernels are executed by devices that own the data. However, for a full low-level support of architectures with multiple accelerators it is often better to resort to frameworks such as OpenCL.

I.3.3.2 Command Graph Based Frameworks

The offload approach works well when only a single accelerator is used and when precise control of the data transfers is not required. Otherwise, it is often better to use another approach that gives full control of the devices (memory management, kernel management, data transfer management, etc.) to the application such as the command graph approach. With this approach, applications dynamically build a graph of commands that are executed asynchronously by the different devices in parallel. The edges in the command graph indicate dependencies between them so that a command is executed only when all of its dependencies

have completed. OpenCL [69] specification implements the command graph model. It defines a programming interface (API) for the *host* to submit commands to one or several *computing devices*. Several OpenCL devices from different vendors can be available on a given architecture. Each vendor provides an implementation of the OpenCL specification, called a *platform*, to support its devices. The mechanism used to expose all platforms available for an application is called the Installable Client Driver (ICD). From the ICD and the platforms, the applications can retrieve platform information, in particular the list of available devices for this platform.

Devices that need to be synchronized, to share or exchange data can be grouped into *context* entities. *Only devices from the same platform can belong to the same context*. The data buffers required for the execution of computing kernels are created and associated with a context, and are lazily allocated to a device memory when needed. *Commands* are executed by computing devices and are of three different types: kernel execution, memory transfer and synchronization. These commands can be submitted to *command queues*. *Each command queue can only be associated with a single device*. Commands are issues in-order or out-of-order, depending on the queue type. Barriers can be submitted to out-of-order command queues to enforce some ordering. Synchronization between commands submitted to queues associated with different devices is possible using *event* entities *as long as command queues belong to the same context*. Events give applications a finer control of the dependencies between commands. As such they subsume command queue ordering and barriers. Each command can be associated with an event, raised when then command completes. The dependences between commands can be defined through the list of events that a command is waiting for. Listing I.4 shows a basic OpenCL host code to execute a kernel on the first available OpenCL device.

The OpenCL ICD is badly designed in that it does not allow devices of different platforms to be easily synchronized. Moreover the OpenCL API is a very low-level one and many extensions have been conceived to make it easier to use, for instance by providing a more unified platform, automatic kernel scheduling, automatic kernel partitioning...

IBM's OpenCL Common Runtime [80] provides a unified OpenCL platform consisting of all devices provided by other available implementations. Multicoreware's GMAC (Global Memory for Accelerator) [105] allows OpenCL applications to use a single address space for both GPU and CPU kernels. Used with TM (Task Manager), it also supports automatic kernel scheduling on heterogeneous architectures using a custom programming interface. Kim *et al.* [89] proposes an OpenCL framework that considers all available GPUs as a single GPU. Their framework expresses code for a single GPU and partitions the work-groups among the different devices, so that all devices have the same amount of work. Their approach does not handle heterogeneity among GPUs, not a hybrid architecture with CPUs and GPUs, and the workload distribution is static. Besides, data dependences between tasks are not considered since work-groups are all independent. De La Lama *et al.*[91] propose a compound OpenCL device in order to statically divide the work of one kernel among the different devices. Maestro[124] is a unifying framework for OpenCL, providing scheduling strategies to hide communication latencies with computation. Maestro proposes one unifying device for heterogeneous hardware. Automatic load balance is achieved thanks to an auto-tuned performance model, obtained through benchmarking at install-time. This mechanism also helps to adapt the size of the data chunks given as parameters to kernels.

Mechanisms for automatic scheduling and load-balancing of the OpenCL kernels on multi-devices architectures have been investigated. A static approach to load partitioning and schedul-

Listing I.4: OpenCL host code example

```
// List platforms and devices
clGetPlatformIDs(..., &platforms);
clGetDeviceIDs(platforms[0], ..., &devices);

// Create a context for devices of the first platform
context = clCreateContext(devices, ...);

// Load and build a kernel
program = clCreateProgramWithSource(...);
clBuildProgram(program, ...);
kernel = clCreateKernel(program,...);

// Create a command queue on the first device
queue = clCreateCommandQueue(context, devices[0], ...);

// Allocate buffers
b1 = clCreateBuffer(context, ...);
b2 = clCreateBuffer(context, ...);

// Initialize the input buffer "b1"
clEnqueueWriteBuffer(queue, b1, ..., &inputData, ...);

// Configure and execute the kernel
clSetKernelArg(kernel, 0, ..., b1);
clSetKernelArg(kernel, 1, ..., b2);
clEnqueueNDRangeKernel(queue, kernel, ...);

// Retrieve result from output buffer "b2"
clEnqueueReadBuffer(queue, b2, ..., &outputData, ...);

// Wait for the completion of asynchronous commands:
// write buffer, kernel execution and read buffer
clFinish(queue);

// Release entities
clReleaseBuffer(b1);
clReleaseBuffer(b2);
clReleaseKernel(kernel);
clReleaseProgram(program);
```

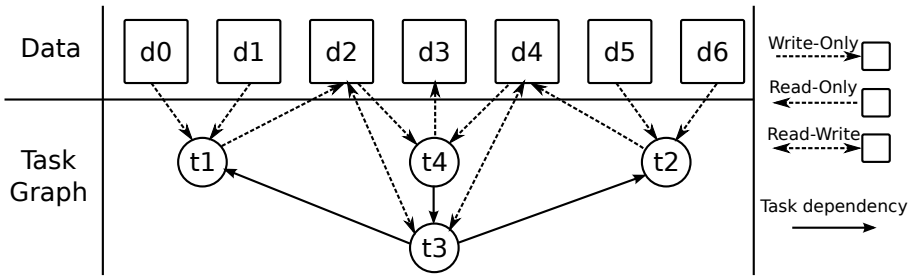


Figure I.7: Example of task-graph

ing is presented by Grewe and O'Boyle [67]. At runtime, the decision to schedule the code on CPU or GPU uses a predictive model, based on decision trees built at compile time from micro-benchmarks. However, the case of multiple GPU is not directly handled, and the decision to schedule a code to a device does not take into account memory affinity considerations. Besides, some recent works use OpenCL as the target language for other high-level languages (for instance, CAPS HMPP [52] and PGI [136]). Grewe *et al.* [68] propose to use OpenMP parallel programs to program heterogeneous CPU/GPU architectures, based on their previous work on static predictive model. Boyer *et al.*[29] propose a dynamic load balancing approach, based on an adaptive chunking of data over the heterogeneous devices and the scheduling of the kernels. The technique proposed focuses on how to adapt the execution of one kernel on multiple devices.

Amongst other OpenCL extensions, SnuCL [88] is an OpenCL framework for clusters of CPUs and GPUs. The OpenCL API is extended so as to include functions similar to MPI. Besides, the OpenCL code is either translated into C for CPU, or Cuda for GPUs. The SnuCL runtime does not offer automatic scheduling between CPUs and GPUs. Moreover, SnuCL does not handle multi-device on the same node.

In Chapter II, we present our OpenCL implementation called SOCL that was among the first ones to provide a unified platform allowing devices of different platforms to be easily used jointly. In addition, by using OpenCL contexts as scheduling contexts, it lets applications use automatic kernel scheduling in a controlled manner. Besides, it automatically handles device memories and uses host memory as a swap memory to evict data from device memories when required. Finally, preliminary support for automatic granularity adaptation is supported too. SOCL is at the junction of frameworks using the command graph model and task graph based runtime systems.

I.3.3.3 Task Graph Based Runtime Systems

The command graph model as implemented by OpenCL lets applications explicitly schedule commands on available devices of an heterogeneous architectures. As it is extremely tedious to do properly and efficiently, a more abstract task graph model has been used by several frameworks such as StarPU [17] or StarSS [18, 113]. With these frameworks, applications dynamically build a task graph that is similar to a command graph containing only kernel execution commands not associated with a specific device. The runtime system is responsible of scheduling kernel executions and data transfers appropriately to ensure both correctness and performance. Figure I.7 shows a task graph and data used by each task.

Listing I.5: StarPU codelet definition example

```
struct starpu_codelet cl =
{
  .where = STARPU_CPU | STARPU_CUDA | STARPU_OPENCL,

  /* CPU implementation of the codelet */
  .cpu_funcs = {
     sgemm_cpu_func
  #ifdef __SSE__
     , sgemm_sse_func
  #endif
     , NULL},

  /* CUDA implementation of the codelet */
  .cuda_funcs = {sgemm_cuda_func, NULL},

  /* OpenCL implementation of the codelet */
  .opencl_funcs = {sgemm_opencl_func, NULL},

  .nbuffers = 2,
  .modes = {STARPU_R, STARPU_RW},
};
```

A task is not exactly the same thing as a kernel. Indeed, in order to be able to schedule a task on different devices of an heterogeneous architecture, several kernels performing the same operation but on different architectures have to be grouped together to compose a task. For instance, SGEMM kernels written in CUDA, OpenCL and C can compose a SGEMM task. Listing I.5 shows how a task is usually defined with StarPU (in this case it is called a `codelet`) while Listing I.6 shows how it is defined with StarSS.

Listing I.7 shows how a task is created and submitted using StarPU API. First, data are registered in host memory. StarPU uses a shared-object memory model and transfers data as required in the different device memories. Then a task entity is created associated with the `cl` codelet and using the previously registered data and is submitted. Tasks are executed asynchronously thus the `task_wait_for_all` primitive is used to wait for its completion. Finally, data are unregistered, meaning that the host memory is ensured to contain data in a coherent state. To make a task depends on some others, StarPU provides explicit and implicit mechanisms. When the implicit mode is enabled, task dependencies are inferred from task data access modes and task submission order: the last task accessing a data in write mode has its identifier stored alongside the data and subsequent access to this data by another task adds a dependence between the two tasks. Listing I.8 shows how to explicitly declare dependencies between several tasks. Task `t2` will be executed only when both `t1` and `t3` have completed. Nevertheless, StarPU may perform some data transfers for `t2` in advance to speed up the execution.

StarPU uses performance models and mainly the heterogeneous earliest finish time (HEFT)

Listing I.6: StarSS codelet definition example

```
// Task prototype
#pragma css task input(A) inout(B)
void sgemm(float A[N], float B[N]);

// Task implementations

#pragma css task implements(sgemm) target device(smp)
void sgemm_cpu_func(float A[N], float B[N]) { ... }

#ifdef __SSE__
#pragma css task implements(sgemm) target device(smp)
void sgemm_sse_func(float A[N], float B[N]) { ... }
#endif

#pragma css task implements(sgemm) target device(cuda)
void sgemm_cuda_func(float A[N], float B[N]) { ... }

#pragma css task implements(sgemm) target device(opencl)
void sgemm_opencl_func(float A[N], float B[N]) { ... }
```

scheduling algorithm. XKaapi [61] provides a similar programming model but uses work-stealing scheduling strategies.

By using these runtime systems, applications do not have to explicitly manage device memories nor to explicitly schedule kernels on accelerators. Hence, they loose the ability to perform load-balancing with granularity adaptation. However, very few runtime systems provide support for this. SOCL that we present in Chapter II faced the same issue. We concluded that the lack of anticipation inherent to the dynamic creation of the task graph by the host program in code opaque to the runtime system made it difficult to introduce efficient automatic granularity adaptation mechanisms. In Chapter III we use a model based on parallel functional programming that gives the runtime system a better knowledge of the task graph so that it can perform tranformations on it more easily, granularity adaptation being one of them.

I.4 Generic Approaches to Parallel Programming

Models and frameworks presented in the previous sections have mostly been designed as evolutions of the existing ones in order to support new constraints introduced by newer architectures. In this section we present approaches that aim to be generic enough to be fully independent of the target architectures. Relying on high-level approaches liberated from the Von Neumann model is an old debate regularly rekindled [19, 35, 51] especially now that it becomes nearly unmanageable to use low-level approaches to program heterogeneous architectures or that their model has been subverted so that there is no point in using them anymore.

In addition to performance and scalability, objectives of these high-level approaches are:

Listing I.7: StarPU Example

```
float *matrix, *vector, *mult;
float *correctResult;
unsigned int mem_size_matrix, mem_size_vector, mem_size_mult;

starpu_data_handle_t matrix_handle, vector_handle, mult_handle;
int ret, submit;

// StarPU initialization
starpu_init(NULL);

// Data registration
starpu_matrix_data_register(&matrix_handle, 0, matrix, width, width, height, 4);
starpu_vector_data_register(&vector_handle, 0, vector, width, 4);
starpu_vector_data_register(&mult_handle, 0, mult, height, 4);

// Task creation and configuration
struct starpu_task *task = starpu_task_create();
task->cl = &cl;
task->callback_func = NULL;
task->handles[0] = matrix_handle;
task->handles[1] = vector_handle;
task->handles[2] = mult_handle;

// Task submission
submit = starpu_task_submit(task);

// Waiting for task completion
starpu_task_wait_for_all();

// Data unregistration
starpu_data_unregister(matrix_handle);
starpu_data_unregister(vector_handle);
starpu_data_unregister(mult_handle);

starpu_shutdown();
```

Listing I.8: StarPU explicit dependency declaration

```
t1 = starpu_task_create();
t2 = starpu_task_create();
t3 = starpu_task_create();

struct starpu_task * deps[] = {t1,t3};
starpu_task_declare_deps_array(t2, 2, deps);
```

- Correctness: programmers avoid most bugs that are introduced during the mapping of high-level concepts to a low-level language. Proofs are easier to establish with high-level models closer to mathematics. High-level models are often deterministic which avoids most bugs due to the parallelization.

- Composability: for instance, models based on functional programming are intrinsically composable because they provide information hiding (i.e. task internal memory cannot be modified by another task), context independence (i.e. referential transparency) and argument noninterference (i.e. immutable data) [51].

- Productivity: programmers should not waste their time dealing with architectural issues and scientists should not need to be experts in parallel computing in addition to their primary science domain to have access to results obtained with high-performance architectures.

- Maintainability: programs written using high-level languages are often much easier to understand than their low-level counterparts as original intents of the programmers have not been as much obfuscated during their transposition in actual code. In particular, there is no boilerplate code to support different architectures.

- Portability: codes that are architecture agnostic are inherently portable as long as a compiler for the target architecture exists.

- High-level optimizations: high-level programming models are more amenable to high-level optimizations that can change whole algorithms. For instance, domain specific languages can use properties of the domain to switch from an algorithm to another, avoid some computations or provide rewrite rules that increase the degree of parallelism.

High-Level approaches are used to represent task graphs that are not unrolled, which is one of the issue faced by task graph based runtime systems presented in the previous section: when too many tasks are submitted, the overhead of the scheduler can be too high. DAGuE [27] and parametric task graphs (PTG) [47] are frameworks that use their own pseudo-language to describe task graphs that are converted into forms amenable to compiler analyses and efficient execution by a runtime system.

I.4.1 Parallel Functional Programming

Functional programming can be used to write programs that are implicitly parallel and that can thus be evaluated in parallel [73, 74, 86, 111, 114, 120]. Indeed side effect free functions can be computed in any order hence expressions composing programs can be evaluated in parallel (function parameters for instance). Purely functional programs are easier to debug thanks to their deterministic nature: their behavior is the same when they are executed sequentially and when they are evaluated in parallel. Program results do not depend on the runtime system and its task scheduling strategies.

Eden [30, 95] and Glasgow Parallel Haskell (GpH) [132] are parallel functional language that extend the non-strict functional language Haskell. PMLS [104] is a parallel implementation of the strict functional language ML. Task creations are explicit in Eden, GpH introduces

new primitives (`par` and `pseq`) that helps the compiler detecting parallelism and PMLS identifies parallelism by detecting certain higher-order functions. A comparison of the three approaches is done in [94]. A drawback of this approach is that granularity may be too fine. Spawning a thread for each expression is too costly and may annihilate the performance gain that would be expected. Several strategies to fusion several fine tasks into coarser ones have been established. [73]

In Chapter III we use parallel functional programming to represent task graphs where tasks are coarse grained so that we are not subject to the granularity issue mentioned above. On the contrary, it allows us to provide a solution to the granularity issue faced by task graph based runtime systems (cf Section I.3.3.3) by substituting some function applications in the program by equivalent functional expressions involving smaller tasks.

I.4.2 Higher-Order Data Parallel Operators

Higher-order functions on collection of data – *data-parallel operators* – are often intrinsically parallel. These higher-order functions are also called *algorithmic skeletons* or *skeletons* for short as they encapsulate patterns of code. A recent survey of these approaches can be found in [65] as well as a manifesto for bringing skeletal approaches into mainstream practice [44].

Typical operators are:

- `map f`: apply the function `f` to every element of the collection and create a new collection containing the produced values.
- `reduce f`: reduce a collection of values to a single one by successively applying the `f` function to two elements. Collection is supposed to be non-empty and if `f` is associative, the reduction can be performed in logarithmic time by using a binary reduction tree.
- `scan f`: also called *prefix sum*, this operator applied to a collection produces a new collection where the i^{th} value contains the result of the reduction (using `f`) up until the i^{th} element of the input collection. [24]

Two kinds of data-parallel models are distinguished: *nested data parallelism* allows collections to contain other collections while *flat data parallelism* does not.

MICROSOFT Accelerator [130] and Sh library [102] are examples of frameworks that support flat data-parallel operators on matrices. Initially Sh targeted GPU programmable shaders through graphics API (at the time GPU were not generically programmable). RapidMind [41] was the commercial successor of Sh and has been bought by INTEL to be included into INTEL Ct [64]. Ct has now been integrated into INTEL ArBB [76, 108] framework and do not target GPUs anymore but multi-core CPUs. HArBB/EmbARBB [127, 128] is an Haskell embedded DSL that uses ArBB's runtime system internally. Intel has retired ArBB in October 2012. Qilin [97] is a similar framework that builds a DAG from the use of a special C++ array type and that dynamically generates INTEL TBB and NVIDIA CUDA codes.

NESL [23] is a framework that supports nested data parallelism. It uses *flattening* to transform nested data parallelism into flat data parallelism [25]. SETL [121]is a language that supports two kinds of data containers: unordered sets and tuples. Nested data parallelism is supported and data parallel operators can be used on these containers.

Data Parallel Haskell (DPH) is a data-parallel framework for Haskell. It supports both flat (Repa [85]) and nested data parallelism (Nepal [38]). Accelerate [37] and Obsidian [129] are other Haskell DSLs that compile data-parallel codes into CUDA/OpenCL codes for GPUs.

Similarly to Repa, Single Assignment C (SAC) [66] is a functional language with a syntax borrowed from C language specialized for flat data parallelism that supports *shape polymorphism*: the same data-parallel operators can be used for several collection shapes. It has been influenced by Sisal [35, 60] that implemented several mechanisms to avoid data duplications in a functional language.

HaskSkel [75] is a library of parallel skeletons implemented using parallel strategies of the parallel functional language GpH [132].

I.4.3 Bird-Meertens Formalism

Bird-Meertens Formalism (BMF) [22] is a calculus of functions that allows programs to be derived from naive program specifications. Functions over various data types are defined together with their algebraic properties. By using *categorical data types* (CDT), functions and properties can be generalized for every data types [55]. Examples of containers that can be represented as categorical data types include lists, finite sets, bags, trees, arrays... [123]

Concatenation list (i.e. a data type `[a]` with two constructors, `single :: a → [a]` and `(++) :: [a] → [a] → [a]`) is a typical example of a categorical data type (i.e. regular recursive type). Most operations on it have a semantically equivalent canonic form $reduce\ \phi\ .\ map\ \rho$ where ϕ is associative. This form can be efficiently parallelized and is called *catamorphism* in category theory [55].

MapReduce [50] is a framework conceived by GOOGLE based on the categorical properties of concatenation lists. Programmers define a catamorphism on concatenation lists by providing only the two functions ρ and ϕ found in the canonic representation $reduce\ \phi\ .\ map\ \rho$ of the catamorphism. MapReduce's runtime system distributes tasks on a whole hierarchical cluster of workstations. Hadoop [134] is a free implementation of MapReduce. Grex [21] is another implementation that targets GPUs.

Bird-Meertens formalism is used to transform programs. For instance, the "Cons-Snoc" (CS) method [63] tries to infer from the same algorithm applied to two kinds of data types (cons-list and snoc-list) a parallel algorithm for a concatenation-list data type. It uses a generalization method of the term-rewriting theory to find ϕ and ρ candidates in the catamorphism $reduce\ \phi\ .\ map\ \rho$. Then it uses a theorem prover to prove that ϕ is associative.

Another transformation consists in inserting a data `split` operator followed by a data `join` operator somewhere in a BMF expression [10, 135]. This operation is equivalent to inserting the identity function, hence semantically neutral. Then, rules are used to transform the expression into a more efficient parallel one. For instance, the `join` operator is delayed as much as possible or removed if possible (e.g. replaced by a reduction). Adl [8, 9] is a functional data-parallel language that is compiled into a BMF representation to be optimized. The mapping between Adl and BMF is formalized in [7].

In Chapter III we show how we could use BMF expressions to optimize functional expressions in the case of automatic granularity adaptation (e.g. linear algebra operations that work

on matrix tiles). Transformation rules are used to remove bottlenecks such as matrix recomposition from its tiles that could have been distributed in several memories to be computed in parallel. We also sketch a method to automatically infer alternative functional expressions with a higher degree of parallelism from a simpler BMF expression. In particular, we show that we may be able to keep partitioning factors (i.e. number of tiles in each dimensions for a matrix) algebraic and to infer constraints on them so that the runtime system could dynamically choose them in a restricted search space.

I.4.4 Algorithmic Choice

Different algorithms can be used to solve the same problem. They may be characterized by the trade-offs they make regarding complexity, memory consumption, accuracy... Parallel algorithms have additional factors such as the amount of communication, degree of parallelism... It is thus hard to select an algorithm for a given architecture. Additionally, algorithms can often be combined especially with *divide-and-conquer* (i.e. recursive) approaches that may select different algorithms at different levels of the recursion. Some high-level frameworks perform *algorithmic choice* as they strive to select the best algorithmic combination to meet some goals. Typically, the most efficient algorithm for a given architecture that compute a result with a bounded minimal accuracy.

Algorithmic choice can also be used to find appropriate data layouts. Different algorithms on different architectures have different performance results with different data layouts. The combinatorial explosion of choices leads to a search space of possibilities that can only be realistically explored in an automatic manner. Execution trace analyze, auto-tuning and micro-benchmarking are methods that can be used to evaluate codes that have been generated. Genetic approaches may also be used to combine algorithms.

QIRAL [20] is a framework developed for physicists using Quantum ChromoDynamics (QCD) that takes formulae written using LaTeX and compiles them into C codes. It is designed to easily evaluate several preconditioning methods and several data layouts. PetaBricks [13] is a framework that supports several specifications (algorithms) for the same computation using a high-level language. Some specifications are automatically inferred and recursive ones are allowed. The framework benchmarks the different algorithms and uses auto-tuning to select the most appropriate ones. Using recursive definitions, it can combine several approaches (iterative, direct...) depending on the input granularity. Finally, it also handles *accuracy choice* by letting programmers specify the desired output accuracy. The runtime system not only selects algorithms using a performance criterion but also based on the accuracy they yield. An example of PetaBricks code for matrix multiplication is given in Listing I.9.

I.5 Conclusion

In this chapter, we described how evolution of architectures from the Von Neumann model to heterogeneous architectures composed of several many-core accelerators has influenced programming models and frameworks. A complementary survey of the different frameworks can be found in [31], especially regarding Cell and FPGA programming. On heterogeneous architectures, frameworks based on the task graph model (cf Section I.3.3.3) offer good perfor-

Listing I.9: Matrix Multiplication with PetaBricks (excerpt from [13])

```
transform MatrixMultiply
from A[c, h] , B[w, c]
to AB[w, h]
{
   //Base case, compute a single element
   to (AB.cell(x,y) out)
   from (A.row(y) a, B.column(x) b) {
       out = dot(a,b);
   }

   //Recursively decompose in c
   to (AB ab)
   from (A.region (0, 0, c/2, h) a1 ,
         A.region (c/2, 0, c, h) a2 ,
         B.region (0, 0, w, c/2) b1 ,
         B.region (0, c/2, w, c) b2) {
       ab = MatrixAdd (MatrixMultiply (a1, b1),
                       MatrixMultiply (a2, b2));
   }

   //Recursively decompose in w
   to (AB.region(0, 0, w/2, h) ab1,
       AB.region(w/2, 0, w, h) ab2)
   from (A a,
         B.region (0, 0, w/2, c) b1,
         B.region (w/2, 0, w, c) b2) {
       ab1 = MatrixMultiply(a, b1);
       ab2 = MatrixMultiply(a, b2);
   }

   //Recursively decompose in h
   to (AB.region (0, 0, w, h/2) ab1,
       AB.region (0, h/2, w, h) ab2)
   from (A.region (0, 0, c, h/2) a1,
         A.region (0, h/2, c, h) a2,
         B b) {
       ab1 = MatrixMultiply(a1, b);
       ab2 = MatrixMultiply(a2, b);
   }
}
```

mance and a quite simple programming model. However there is no recognized specification or standard based on this model. Only OpenCL and OpenACC are specifications endorsed by many vendors. However OpenCL is based on the command graph model which is much harder to use correctly and efficiently and OpenACC is based on an offloading model that is much simpler to use but that does not offer good support for multi-accelerator setups.

Task graph based runtime systems raise new issues that cannot be easily tackled in current programming models. For instance, the granularity adaptation issue that consists in substituting a coarse task with finer ones is an optimization that has an impact on other tasks of the task graph using the same data as the partitioned task. Inter-task optimizations are hard to implement in runtime systems when task graphs are dynamically built in unpredictable ways. In the next two chapters we try to improve on the existing solutions in the following ways:

- In Chapter II we propose an OpenCL implementation that strive to integrate all of the benefits obtained with task graph based runtime systems while still remaining very close to the standard. In particular it provides a unified platform with support for automatic kernel scheduling, automatic device memory management and preliminary support for kernel granularity adaptation.
- In Chapter III we show that by using another approach to describe task graph than the one used in most task graph based runtime systems, we can tackle issues they face such as granularity adaptation or static task graph optimization. In particular, our model is based on parallel functional programming so that we benefit from using both a high-level approach and high-performance kernels written in low-level languages. We present our implementation of a runtime system supporting this programming model to schedule kernels on heterogeneous architectures.

Notes

Many people in the high-performance computing community seem to agree that using several models at the same time on clusters of heterogeneous architectures such as MPI, OpenMP and OpenCL/OpenACC is not viable in the long term and that we should strive to use more abstract models which provide more information to compilers and runtime systems. However as there is no recognized mature alternative and as most people do not want to invest into experimental frameworks – especially for applications that have to last for years – there is a kind of chicken and egg issue and it is hard to predict when a mature alternative will be available. This prudence is well-founded as some frameworks such as High-Performance Fortran (HPF) and Intel ArBB are examples of frameworks that have failed: ArBB has been retired by Intel without notice in October 2012; HPF is an example of a language that was expected with high expectations and whose compilers had to be rushed out prematurely to comply to the impatience of the HPC community. Consequently disappointing preliminary results led to a fall in its acceptance and to its abandon as a failed project. [87]

In this survey of programming languages, models and frameworks, we limited ourselves to the ones used by the high-performance computing community. More general surveys can be found in [118, 122].

CHAPTER II

SOCL: AUTOMATIC SCHEDULING WITHIN OPENCL

Chapter abstract

In this chapter, we present SOCL, our extended OpenCL implementation that provides several mechanisms to ease the exploitation of heterogeneous architectures. SOCL unifies every installed OpenCL implementation into a single platform, hence making mechanisms constrained to each platform (device synchronization, etc.) available for all of them equally.

It provides automatic device memory management so that host memory is automatically used to swap out buffers in order to make room in device memory. Given these two prerequisites, SOCL can provide its best feature, namely automatic kernel scheduling. OpenCL context use has been extended to become scheduling contexts allowing command queues to be associated with contexts instead of a specific device. Finally, SOCL provides a preliminary support for automatic granularity adaptation.

II.1 Introduction

There are currently two specifications that are widely supported by runtime systems for heterogeneous architectures, namely OpenACC and OpenCL (cf previous chapter). The former is simpler to use as its scope is limited to architectures containing a single accelerator (cf Section "Scope" in [71]) and its interface is composed of compiler annotations (pragmas) for C and Fortran codes. On the contrary, OpenCL fully supports architectures with multiple accelerators because it gives application full control on accelerators: management of memory allocations and releases, data transfers, kernel compilations and executions...

Programming portable applications with OpenCL is very difficult. Accelerators can be very different one from the other and there can be many of them in an architecture. Most of the properties of the accelerators are exposed to applications through the OpenCL Platform API (memory hierarchy, number of cores...). Knowing what to do given these information to efficiently schedule kernels on devices remains hard. Moreover, some information is missing such as bandwidths and latencies of the links between host memory and device memories.

In this chapter we present our OpenCL implementation called SOCL. Our goal with this implementation is to bring dynamic architecture adaptation features into an OpenCL framework. From an OpenCL programmer perspective, our framework only diverges in minor ways from the OpenCL specification and is thus straightforward to use. Moreover, we strived to extend existing OpenCL concepts (e.g. contexts become scheduling contexts) instead of introducing new ones so that some of these extensions could be included in a future OpenCL specification revision. Unlike most other implementations, SOCL is not dedicated to a specific kind of device nor to a specific hardware vendor: it simply sits on top of other OpenCL implementations to support hardware accelerators. SOCL is both an OpenCL implementation and an OpenCL client application at the same time: the former because it can be used through the OpenCL API and the latter because it uses other implementations through the OpenCL API too.

II.2 Toward Automatic Multi-Device Support into OpenCL

In this section, we present extensions to the OpenCL specification that are provided by SOCL in order to automatically support multi-device architectures:

- Unified OpenCL platform: SOCL exposes all devices of the other OpenCL implementation into its own OpenCL platform. It gives access to all of the mechanisms provided by OpenCL (that are constrained into each platform by design) to all devices equally. For example, synchronizations using events are now available between all devices.

- Automatic device memory management: buffers in SOCL are not necessarily attached to a specific device nor context. The runtime system automatically transfers them as required in device memories. Coherency is loosely ensured among the different memories. In particular, the host memory is used to store buffers that have to be evicted from a device memory to make room for other buffers.
- Automatic kernel scheduling on multiple devices: OpenCL command queues have been extended in SOCL so that they no longer need to be associated with a specific device. Instead, the context they are associated with is said to be scheduling context and command submitted in the queue are automatically scheduled on a device of the context. Each scheduling context may have its own scheduling policy.
- Preliminary support for automatic granularity adaptation: with SOCL, applications can associate a function with each kernel that takes a partitioning factor as parameter. During the execution, SOCL can choose to execute the given function instead of the kernel. It is supposed that the kernel is executed more than once so that the runtime system automatically explores the search space of the partitioning factor to select the best one.

II.2.1 Unified OpenCL Platform

SOCL is an implementation of the OpenCL specification. As such, it can be used like any other OpenCL implementation with the OpenCL host API. As the Installable Client Driver (ICD) extension is supported, it can be installed side-by-side with other OpenCL implementations and applications can dynamically choose to use it or not among available OpenCL platforms.

OpenCL specification defines how the installable client driver (ICD) interface can be used to exploit simultaneously several OpenCL implementations provided by different vendors. It is basically a kind of multiplexer that forwards OpenCL calls to the appropriate implementation based on manipulated entities. As such, entities of the different implementations cannot be interchanged. SOCL has been conceived as a much more integrated multiplexer of OpenCL implementations than the ICD as it provides a unified platform: all the glue has been added to make all entities of every other installed OpenCL implementation appear as if they belong to the same platform.

SOCL uses the ICD internally as shown on Figure II.1 to present all devices of every other platform in its own platform. Devices of other platforms can be used through the SOCL platform as they would through their own platform. However, using them through the SOCL platform brings several benefits. First, entities can be shared amongst devices from different vendors. In particular, it is much easier for application programmers to be able to use synchronization mechanisms (events, barriers, etc.) with all devices while these mechanisms were originally only provided for devices belonging to the same platform. Second, SOCL automatic device memory management and automatic scheduling (presented in the following sections) rely on this unified platform to follow the OpenCL specification as closely as possible. For instance, the specification states that contexts must only contain devices belonging to the same platform: this constraint is still true with SOCL scheduling contexts, except that by using the SOCL platform, devices originating from different platforms can be grouped into the same context.

Listing II.1 shows how to select the SOCL platform. If there is no other OpenCL implemen-

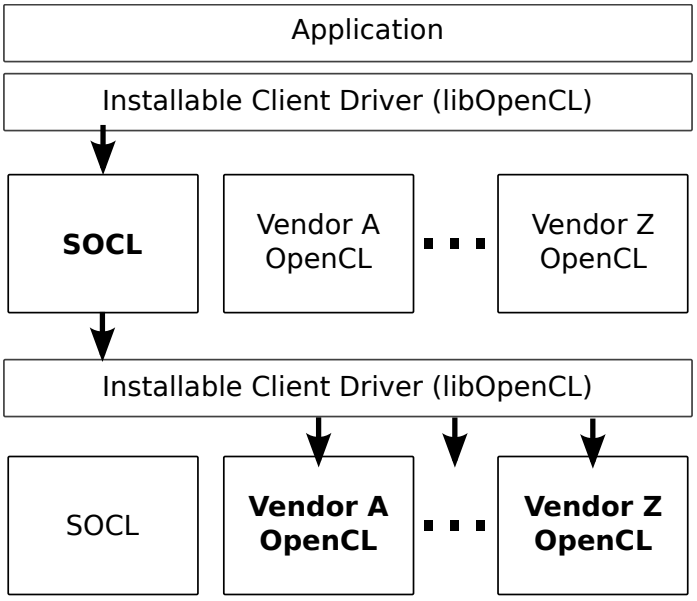


Figure II.1: SOCL unified platform uses OpenCL implementations from other vendors and can be used as any other implementation through the installable client driver (ICD).

tation available, SOCL platform does not contain any device. Otherwise, devices of the other platforms are all available through the SOCL unified platform. Note that we do not check returned error codes in this example but it is highly advised to do it in real codes.

II.2.2 Automatic Device Memory Management

OpenCL buffers are not directly allocated in device memories. Instead, they are virtually allocated associated with a context (a group of devices) and they are lazily allocated in device memory when a command require them. OpenCL 1.1 adds a new API to force the migration of a buffer in a specified device memory which was lacking in OpenCL 1.0. Hence, lazy allocation makes it difficult to manage device memories as allocation failures can be triggered by almost every command using buffers.

SOCL relieves programmers from encountering most of these errors. It is very rare that an allocation fails with SOCL because it uses the host memory as a swap memory space where to store buffers evicted from device memories. As a consequence of the automatic device memory management, a lot of cumbersome buffer management has become unnecessary in applications. For instance, if a buffer has to be used by several kernels on several devices, it only have to be allocated once. The runtime system ensures that it will be transferred appropriately into device memories where the kernels are scheduled and its content will remain coherent automatically.

Suppose that several kernels use the same input data. With classic OpenCL, a buffer has to be allocated and initialized per device. On the contrary, SOCL only requires the creation of a single buffer that can be used simultaneously by different kernels (if they access it in read mode). It will be automatically duplicated as required on the different devices. Listing II.2 shows both approaches. Note that the command queue used to submit the WriteBuffer com-

Listing II.1: SOCL Platform Selection

```
/* Retrieve the number of available platforms */
cl_uint platformCount;
clGetPlatformIDs(0, NULL, &platformCount);

/* Retrieve platform IDs */
cl_platform_id platforms[platformCount];
clGetPlatformIds(platformCount, platforms, NULL);

/* Select SOCL platform */
cl_platform_id platform;
for (i=0; i<platformCount; i++) {
   cl_uint vendorSize;
   clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, 0, NULL, &vendorSize);
   char vendor[vendorSize];
   clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, vendorSize, vendor, NULL);
   if (strcmp(vendor, "INRIA") ==  0) {
      platform = platforms[i]
      break;
   }
}
```

mand in the SOCL case can be attached to any device, or no device at all (cf next section on automatic command scheduling). Finally, the recommended way to initialize a buffer with SOCL is to use the `CL_MEM_USE_HOST_PTR` flag at buffer creation. The buffer is then created as if it had been swapped out in host memory, without performing any data transfer.

II.2.3 Automatic Scheduling

OpenCL lets applications perform kernel scheduling explicitly by submitting commands into command queues attached to devices. As previously mentioned, it is very hard to do it correctly because many factors have to be taken into account (device properties, characteristics of the interconnect between the host and the different devices…). Moreover, early choices that are made can have dramatic impacts on performance. Kernel scheduling should also take into account data that are already present in device memories, kernels that are already compiled, etc. Task graph based runtime systems presented in Section I.3.3.3 provide a different programming model alongside a custom API.

In SOCL, automatic command scheduling is integrated by transforming the OpenCL semantics a little bit so that contexts are now *scheduling contexts*. While in OpenCL command queues are each associated with a context (i.e. a group of devices that can share entities) and to a device which execute the submitted commands, in SOCL, we make the association to a specific device optional. Command queues are then only associated with a context that we call scheduling context. Any command enqueued into a command queue that is not associated with a device will be executed by one of the device of the associated scheduling context.

Listing II.2: OpenCL Buffer Allocation Example

```
/* Duplication of a buffer in every device memory */

/* Without SOCL */
cl_mem b[ndevices];
for (i=0; i<ndevices; i++) {
   b[i] = clCreateBuffer(ctx,...)
   clEnqueueWriteBuffer(cq[i], b, 1, 0, size, ptr, 0, NULL, NULL);
}

/* With SOCL */
cl_mem b = clCreateBuffer(ctx,...)
clEnqueueWriteBuffer(cq, b, 1, 0, size, ptr, 0, NULL, NULL)

/* With SOCL using CL_MEM_USE_HOST_PTR flag */
cl_mem b = clCreateBuffer(ctx, CL_MEM_USE_HOST_PTR, size, ptr, NULL);
```

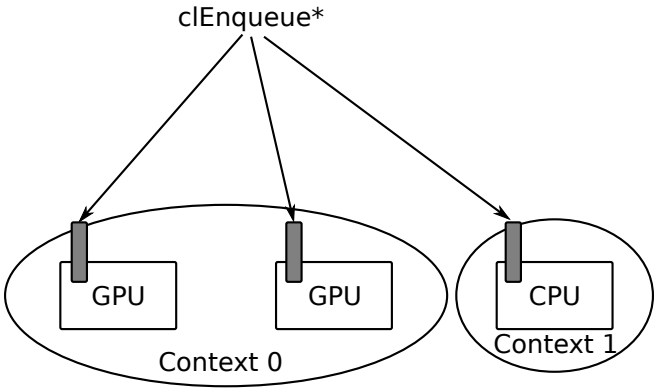
Each scheduling context can be parameterized using context specific properties. For instance, applications can select the scheduling policy that is used to schedule commands on devices. Figure II.2 shows the two different cases, where command queues are attached to devices as in standard OpenCL (a) or to scheduling contexts as proposed by SOCL (b). Note that it is still possible to attach command queues to devices with SOCL and that scheduling contexts are optional. An application can be incrementally converted to using them while ensuring that it does not imply performance loss.

Thanks to the unified platform, applications are free to create as many contexts as they want and to organize them the way it suits them the most. For instance, devices having a common characteristic may be grouped together in the same context. One context containing GPUs and accelerators such as Intel MIC could be dedicated to heavy data-parallel kernels and another containing CPUs and MICs would be used to compute kernels that are known to be more sequential or to perform a lot of control (jumps, loops...). The code in Listing II.3 shows how this example would look like.

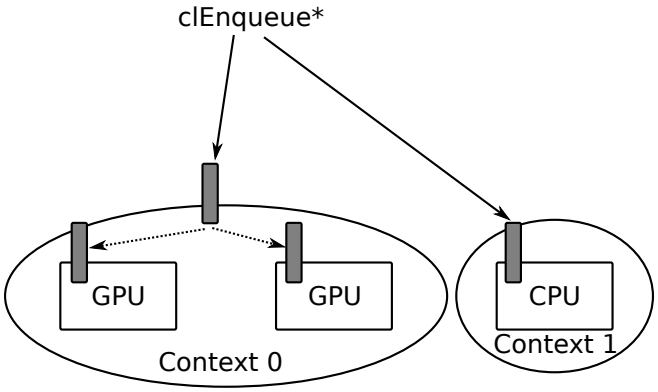
A predefined set of scheduling strategies assigning commands to devices is built in SOCL. They can be selected through context properties. For instance, the code in Listing II.4 selects the "heft" scheduling policy, corresponding to the Heterogeneous Earliest Finish Time heuristic [131]. Other available heuristics are referenced in [82], and additional strategies can be user-defined if need be.

II.2.4 Automatic Granularity Adaptation

With automatic command scheduling, applications slightly loose the control on where commands are executed. For kernel execution commands, it can be problematic as applications may not want to execute exactly the same kernel depending on the targeted device. Generic OpenCL kernels – that do not use device specific capabilities – may not have good performance



(a) Command queues attached to devices (OpenCL Specification)



(b) Command queues optionally attached to scheduling contexts (SOCL extension)

Figure II.2: (a) Command queues can only be attached to a device (b) Command queues can be attached to contexts and SOCL automatically schedules commands on devices in the context

Listing II.3: Context queue creation example. Scheduling and load-balancing of commands submitted in these queues are automatically handled by SOCL.

```
cl_context ctx1, ctx2;
cl_command_queue cq1, cq2;

ctx1 = clCreateContextFromType(NULL,
   CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_ACCELERATOR,
   NULL, NULL, NULL);
ctx2 = clCreateContextFromType(NULL,
   CL_DEVICE_TYPE_CPU | CL_DEVICE_TYPE_ACCELERATOR,
   NULL, NULL, NULL);

cq1 = clCreateCommandQueue(ctx1, NULL, 0,
 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
cq2 = clCreateCommandQueue(ctx2, NULL, 0,
 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
```

Listing II.4: Context properties are used to select the scheduling policy.

```
cl_context_properties properties[] = { CL_CONTEXT_SCHEDULER_SOCL, "heft", 0 };

cl_context ctx = clCreateContextFromType( properties, CL_DEVICE_TYPE_CPU,
   NULL, NULL, NULL);
```

compared to kernels that have been tuned for a specific architecture, taking into account specificities such as cache sizes, number of cores... The code of the kernels, as well as the size of their dataset, have to be adapted to the device architecture. Adapting granularity is a common issue for applications that aim to be portable. Given an architecture composed of several heterogeneous computing devices, and computation that could be split into smaller parts, it is often difficult to estimate how the partition scheme will impact performance. In order to deliver high performance, we believe task granularity adaptation should be done dynamically at runtime by the runtime system. This is why SOCL provides several mechanisms that offer preliminary support for automatic granularity adaptation.

As a first step toward adapting kernels to devices, SOCL provides additional compilation constants defined during kernel compilation and specific to the targeted device. They may be used in kernel source code to help the user to define device-dependent codes. For instance, the constant `SOCL_DEVICE_TYPE_GPU` is defined for every GPU device. As kernels can be compiled per device at runtime, this specialization can also be done explicitly by the user. This allows a finer code adaptation to the devices. However this approach is limited in that it cannot be used to change the granularity of a task.

SOCL lets applications associate a *divide function* with each kernel. On kernel submission, the runtime system may decide either to execute this function in place of the kernel, or to execute the kernel as usual. The divide function, executed by the host, takes a parameter called the *partitioning factor* and issues several smaller kernels equivalent to the initial kernel. In general, these kernels define a task graph that corresponds to a decomposition of the initial kernel computation. The partitioning factor drives how this initial computation is divided into smaller ones. The range of values for this factor is associated with the kernel by the application and SOCL selects the value dynamically from this range. It means that each time a command is issued for the execution of a kernel, the runtime can decide to run instead the divide function associated with this kernel (on the host) with a partitioning factor of its choice.

SOCL uses a granularity adaptation strategy to explore the partitioning factor range each time a kernel with an associated divide function is submitted, with the same input sizes. In a first phase, it saves statistics about execution time for each partitioning factor. In a second phase it uses and updates these statistics to choose appropriate partitioning factors. These statistics are still updated in the second phase to dynamically adapt to the execution environment (e.g. device load, etc.). In the future, several granularity adaptation strategies could be implemented, just like several scheduling strategies are available. Each kernel could use the most appropriate strategy.

The divide function has the prototype presented in Listing II.5. Parameters are respectively the command queue that should be used to enqueue the kernels, the partitioning factor, an event that must be set as dependency to sub-kernels. The last parameter allows the di-

Listing II.5: Divide function prototype

```
cl_int (*)(cl_command_queue cq, cl_uint partitioning_factor,
           const cl_event before, cl_event *after)
```

Listing II.6: Definition of a divide function for a kernel

```
#define FUNC  -1
#define RANGE -2

cl_uint sz = sizeof(void *);
cl_uint range= 5;

clSetKernelArg(kernel, FUNC,  sz, part_func);
clSetKernelArg(kernel, RANGE, sz, &range);
```

vide function to return an event indicating that every sub-kernel has completed (typically an event associated with a marker command). The return value indicates if an error occurred or not. Listing II.6 shows how an application can associate the divide function (`part_func`) and the partitioning factor range ($[1, 5]$). For this preliminary version, we choose to use the OpenCL API function `clSetKernelArg` with a specific argument index. A cleaner way to do this would be to introduce extension functions using OpenCL extension API. See NBody example in Section II.4.5 for a complete example using SOCL automatic granularity adaptation capabilities.

II.3 Implementation

SOCL is a full OpenCL 1.0 implementation (without imaging support) written in C. It implements the installable client driver (ICD) extension so that it can be used jointly with other OpenCL implementations, hence applications can choose to use it or not at runtime. SOCL takes advantage of an existing runtime system (namely StarPU) to schedule tasks on devices. It avoids reimplementing some features provided by this other runtime system. We chose StarPU but other runtime systems such as the one of StarSs could have been used equally.

StarPU supports several drivers such as CUDA, CPU, OpenCL, Xeon Phi... SOCL initializes StarPU with only the OpenCL driver enabled. For each OpenCL kernel entity that is created, SOCL creates a StarPU codelet as in Listing I.5 but with a single OpenCL implementation. The number of parameters (i.e. buffers) to the kernel/codelet can be easily obtained by calling the `clGetKernelInfo` API with the `CL_KERNEL_NUM_ARGS` flag. Parameter access modes are much trickier to deal with. OpenCL 1.2 provides a new `clGetKernelArgInfo` API that could be called with the `CL_KERNEL_ARG_ACCESS_QUALIFIER` flag to obtain each parameter access mode. However, in order to support older OpenCL implementations, SOCL infers kernel parameter access modes from flags (`CL_MEM_READ_ONLY...`) that have been used to create the buffers that are passed as kernel parameters.

OpenCL provides two mechanisms to coordinate command execution: command queues and events. That is implicit dependencies (in the case of in-order queues or synchronization commands such as barriers) and explicit dependencies (events). SOCL converts all of these dependencies into explicit StarPU dependencies using "tag" mechanism (not covered in this document, see [82]).

Support for scheduling contexts has been recently integrated into StarPU [79]. SOCL uses them so that a scheduling context is associated with each OpenCL context. Context properties set on context creation are used to configure StarPU's scheduling contexts (selection of the scheduler to use, etc.). In addition, StarPU provides a task property that permits applications to specify on which device a task has to be executed. When a kernel is submitted through SOCL in a command queue associated with a device, SOCL sets this property appropriately. When the command queue is only associated with a context, SOCL lets StarPU schedules the task on any device contained in the context using StarPU's scheduling contexts.

StarPU supports several data types (matrix, vector...) but we only use the one called "variable" that corresponds to OpenCL buffers as it is only defined by its size. StarPU only provides a single way to create and initialize a data, namely registering. It consists in indicating to the runtime system that a region of the host memory should be used to initialize a new buffer. It is a very cheap operation and the only constraint is that applications do not access the memory area directly thereafter. SOCL uses this mechanism when a buffer is created with the `CL_USE_HOST_PTR` flag. Thus, it is the recommended way to create and initialize a buffer with SOCL. Another way to create and initialize a buffer is to use the `CL_MEM_COPY_HOST_PTR` flag on buffer creation. In this case, SOCL duplicates (using `memcpy`) the memory region containing initializing data and then uses StarPU's data registering mechanism on the duplicate region.

SOCL also supports data transfers provided by the `ReadBuffer` and `WriteBuffer` commands. The latter may also be used to initialize a buffer but with the risk of an overhead due to the data transfer to the selected (manually or automatically) OpenCL device. `ReadBuffer` and `WriteBuffer` commands are currently implemented as StarPU tasks that perform data transfer instead of executing a kernel. The benefit of this approach is that it is then very easy to integrate data transfers in a StarPU task graph, something that is lacking in StarPU. The drawback of this approach is that the device executing the transfer task does not execute any kernel during the transfer.

OpenCL uses a reference counting mechanism to release entities when every command using them has completed and when the host application has indicated that it will not use it anymore. Consequently, a command completion can trigger the release of several entities. SOCL implements this mechanism and uses a specific "garbage collection" thread to release entities in order to circumvent restrictions on what can be done in StarPU task completion callback functions. For instance, a task that releases itself in its own callback is not allowed by StarPU but is one of the many chain reaction that may happen with OpenCL reference counting mechanism and that is correctly handled by our implementation.

II.4 Evaluation

SOCL performance has been evaluated with different applications. In our experiments, we have used the following hardware platforms:

- Hannibal: Intel Xeon X5550 2.67GHz with 24GB, 3 NVidia Quadro FX 5800
- Alaric: Intel Xeon E5-2650 2.00GHz with 32GB, 2 AMD Radeon HD 7900
- Averell1: Intel Xeon E5-2650 2.00GHz with 64GB, 2 NVidia Tesla M2075

The software comprises Linux 3.2, AMD APP 2.7, Intel OpenCL SDK 1.5 and Nvidia CUDA 4.0.1.

II.4.1 Matrix Multiplication

Many numerical algorithms use matrix multiplications, hence it is crucial to to be able to perform it as fast as possible. In addition, it is a good example of a regular problem because matrices can be split in tiles computed independently and the number of operations required to compute each block only depends on the block dimensions.

The code used for this test performs a matrix multiplication between two matrices containing random single-precision floating-point values. Parallelization of the operation $C \leftarrow A \times B$ has been obtained by splitting matrices A and C in blocks of lines while matrix B is not partitioned. With SOCL, a scheduling context is created so that kernels are automatically scheduled on all the available devices. We compared this approach with a static round-robin distribution well suited for regular problems of this kind. Note that the kernel used for this test has not been extensively tuned for the target architecture.

Figure II.1 shows results obtained for a fixed size of the blocks (64 rows) when three GPUs are used only and when an additional CPU is used (Hannibal architecture has been used for these tests). As expected given the sub-optimal kernel used, absolute performance is not on par with highly tuned kernels such as those used by MAGMA [5]. Nevertheless, this example shows that performance with three homogeneous GPUs is better with SOCL than with a round-robin distribution that is well-suited for this kind of parallel pattern. It is explained by the fact that while accelerators are homogeneous, links between them and the host memory may not have the same capabilities or NUIOA effects may occur. SOCL dynamically adapts to these constraints. When Intel OpenCL CPU implementation is used in conjunction with NVidia's one, performance obtained with static round-robin distribution fall dramatically because the performance of the kernel is very bad on the CPU. A static distribution would need to take this into account and to perform load-balancing in order to ensure that a smaller amount of kernels are scheduled on the CPU. Comparatively, SOCL correctly adapts to the different setup and benefits from the additional CPU as a performance increase of 19% is observed.

This example illustrates automatic performance portability that can be achieved with SOCL. Even when problems are regular and when a static round-robin distribution seems appropriate on an homogeneous architectures, some factors such as the interconnects between host and devices can make it heterogeneous in practice. In this case as in the case of other heteroge-

Table II.1: Single-Precision Floating-Point Matrix Multiplication Performance Results (A matrix dimensions are $16k \times 64k$, B matrix dimensions are $16k \times 16k$, blocks contain 64 rows)

	3 GPUs	3 GPUs + 1 CPU
Round-Robin	440 GFlops	30.3 GFlops
SOCL	459 GFlops	546.8 GFlops

neous architectures (e.g. GPUs and CPU used jointly), SOCL can dynamically adapt to the architecture.

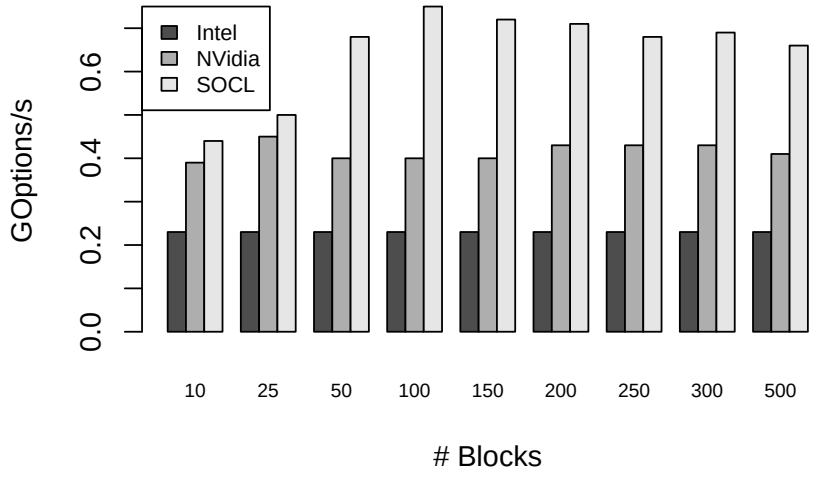
II.4.2 Black-Scholes

The Black Scholes model is used by some option market participants to estimate option prices. Given three arrays of n values, it computes two new arrays of n values. It can be easily parallelized in any number of blocks of any size. With this test we show how automatic device memory management lets SOCL automatically handle cases where the problem size outgrows the device memory capacity. We also present a case where data are reused from one iteration to another to show that SOCL takes into account data that are available in each memory in its scheduling policies.

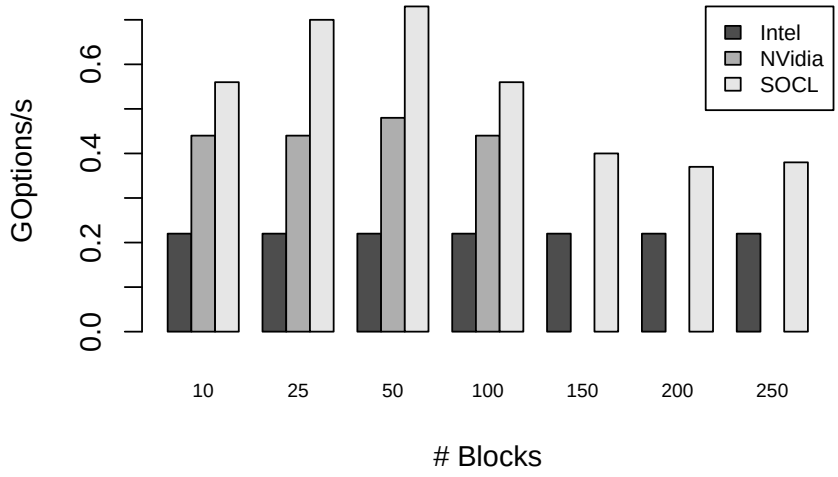
For this test, we used the kernel provided in NVidia OpenCL SDK that uses single-precision floating-point values. Similarily with the matrix multiplication, we also compared SOCL with a static round-robin distribution of the kernels.

Figures II.3a, II.3b and II.3c present performance obtained on Hannibal with blocks of fixed size of 1 million, 5 million and 25 million options. The first two tests have been performed using a round-robin distribution of the blocks with Intel and NVidia OpenCL implementations. When the size and the number of blocks are too high, the graphic cards do not have enough memory to allocate required buffers. Since NVidia's OpenCL implementation is not able to automatically manage memory on the GPU, it fails in these cases, which explains why some results are missing. In contrast, when it uses graphic cards, SOCL resorts to swap into host memory if necessary and is able to handle these cases. The last test presents performance obtained with the automatic scheduling mode of SOCL. The blocks are scheduled on any device (GPU or CPU). We observe that on this example, automatic scheduling always gives better performance than the naive round-robin approach, nearly doubling performance in the case of 1M options (for 100 blocks). This is due to the fact that both computing devices (CPU and GPU) are used, which neither NVidia nor Intel implementation of OpenCL is able to achieve.

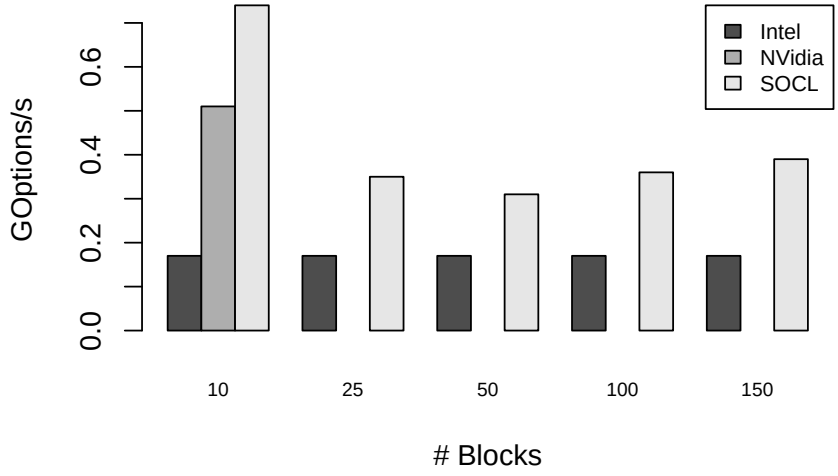
Figure II.4 presents the results we obtained by performing 10 iterations on the same data using the same kernel. This test illustrates the benefits of automatic memory management associated with the scheduling, when there is some temporal locality. The test was conducted on Averell1 with a total option count of 25 millions. This time we used two different scheduling algorithms – eager and heft – to show the impact of choosing the appropriate one. We can see that the heft algorithm clearly outperforms the other approaches in this case, and avoids unnecessary memory transfers. Indeed, this algorithm takes into account memories into which data are stored to schedule tasks. This advantage comes with very little impact on the original OpenCL code, since it only requires to define a scheduling strategy to the context. Note that the number of iterations has been chosen arbitrarily and that the performance gap increases



(a) 1M Options



(b) 5M Options



(c) 25M Options

Figure II.3: Performance of Black Scholes algorithm on Hannibal with blocks containing 1 million (a), 5 millions (b) and 25 millions of options (c).

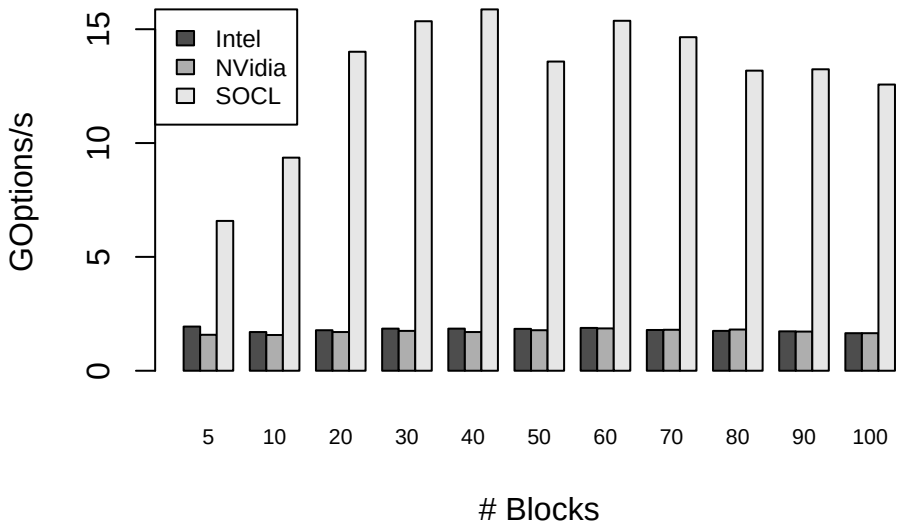


Figure II.4: Performance of 10 iterations (i.e. with data reuse) of the Black Scholes algorithm on Averell1 with a total option count of 25 millions.

between SOCL and the static approaches when the number of iterations is raised.

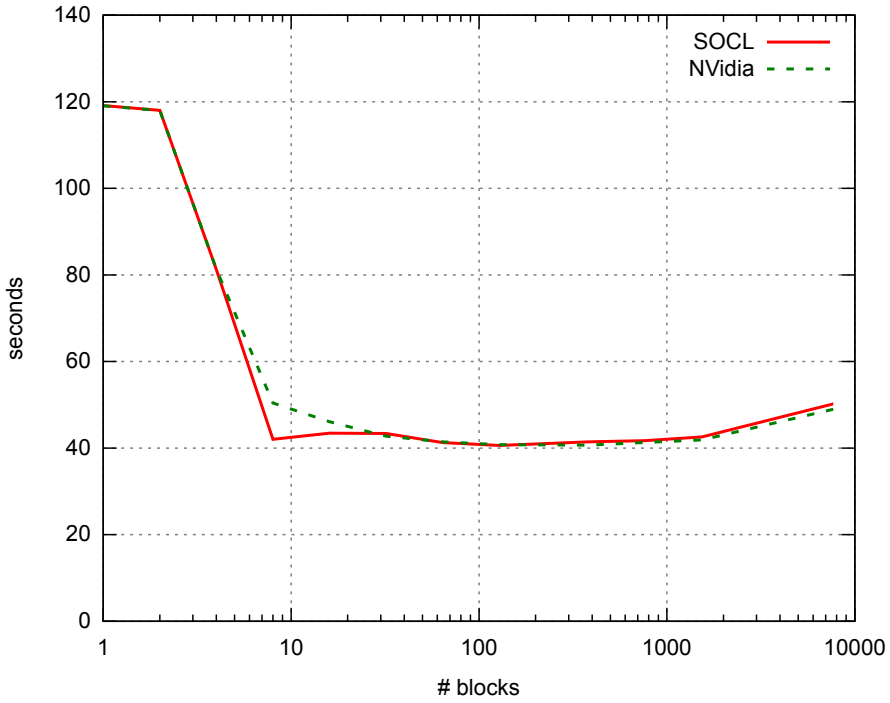
Overall, this example shows the following properties of the SOCL platform: (1) the swapping mechanism allowing large computations to be performed on GPUs, on contrary to NVidia OpenCL implementation; (2) an efficient scheduling strategy in case of data reuse with the heft scheduler; (3) a performance gain up to 85% without data reuse and even higher in case of data reuse.

II.4.3 Mandelbrot Set Image Generation

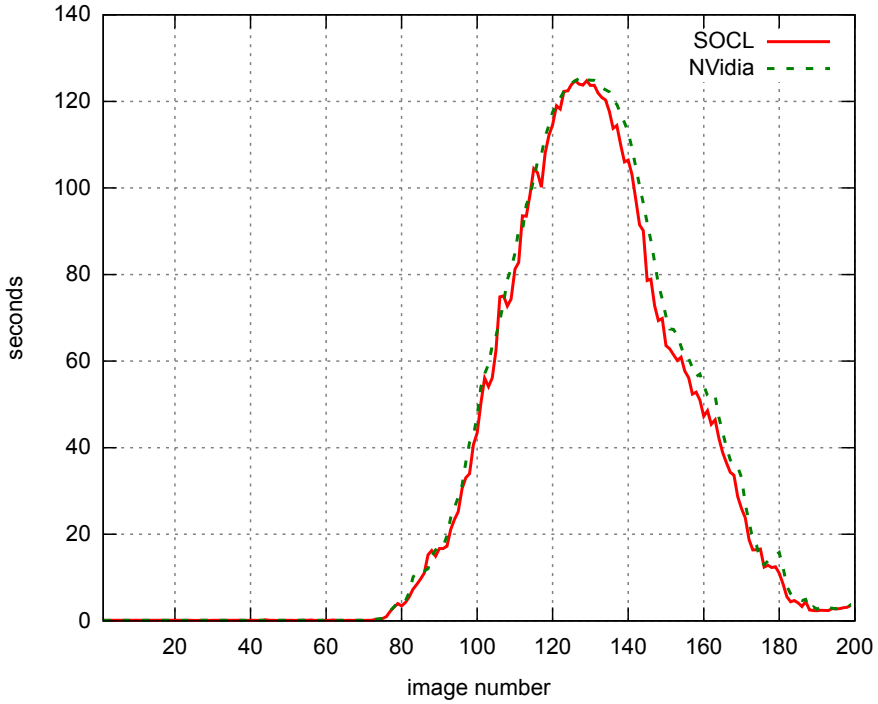
Generation of images representing a portion of the Mandelbrot set is a good example of code that requires load-balancing. Compared to matrix-multiplication and Black-Scholes examples, it is irregular because the time taken to compute each pixel of the resulting image only depends on the coordinate of the represented point in the Mandelbrot set. Depending on the position of the view plane in the Mandelbrot set, some zones in the image can take much longer to compute than some others.

For this example, we generate an image containing 79 millions of pixels. We split the whole image into block of lines that are distributed on available devices. We compare SOCL automatic scheduling with a static round-robin distribution. For the first test, we chose a position in the Mandelbrot so that an unbalance between the time required to compute each block of lines is observed. For the second test, we generate 200 images by moving and zooming into the Mandelbrot set.

We executed this example on Hannibal architecture with only the 3 GPUs enabled. Results of the first test are shown in Figure II.5a. It shows the time taken to generate the whole image given the number of blocks of lines. Results of the second test are shown in Figure II.5b. We can observe that SOCL obtains comparable performance results in both cases.



(a) Computation time given a partition factor for a chosen image



(b) Computation time for 200 images with a fixed number of blocks (16)

Figure II.5: Mandelbrot set image generation on Hannibal with 3 GPUs (lower is better)

The example shows that SOCL automatic scheduling mechanisms work well, especially in the case of irregular problems. We show an example of an irregular problem executed on an heterogeneous architecture in the following LuxRender example.

II.4.4 LuxRender

LuxRender [99] is a rendering engine that simulates the flow of light using physical equations and produces realistic images. LuxRays is a part of LuxRender that deals with ray intersection using OpenCL to benefit from accelerator devices. SLG2 (SmallLuxGPU2) is an application that performs rendering using LuxRays and returns some performance metrics. SLG2 can only use a single OpenCL platform at a time. As such, it is a good example of an application that could benefit from SOCL property of grouping every available device in a single unified OpenCL platform.

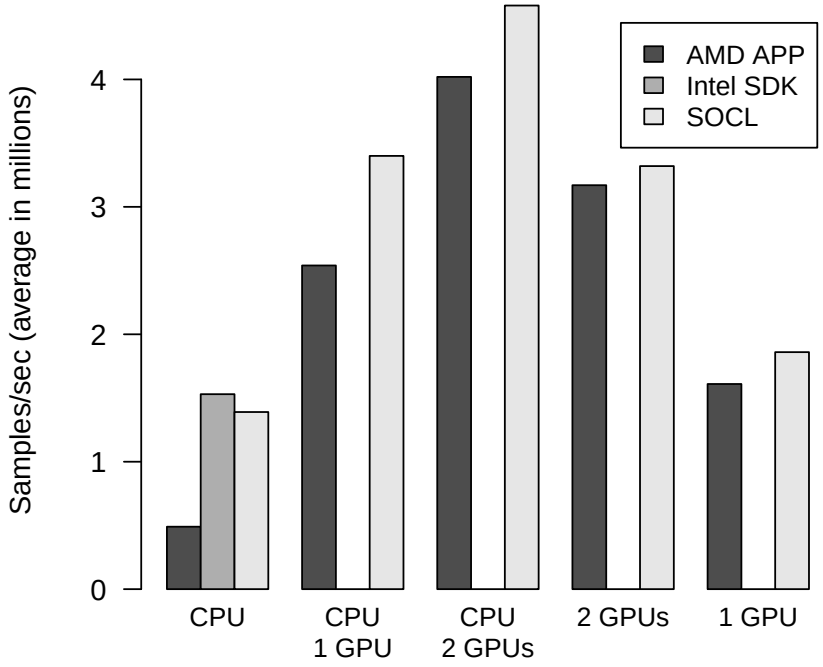
For this experiment, we did not write any OpenCL code, we used the existing SLG2 OpenCL code unmodified, once for each OpenCL platform. We used SLG2 batch mode to avoid latencies that would have been introduced if the rendered frame was displayed and we arbitrarily fixed the execution time to 120 seconds. The longer the execution time, the better the rendered image quality because the number of computed samples is higher. SLG2 also supports usual CPU compute threads but we disabled those because they would have interfered with OpenCL CPU devices. SLG2 has been configured to render the provided "luxball" scene with the default parameters. As SLG2 code has not been modified, we configured SOCL to fake a big context containing all devices with automatic scheduling enabled and we bypassed command queue associations to specific devices. Hence, kernels submitted by SLG2 can be arbitrarily scheduled on any device by SOCL.

The average amount of samples computed per second for each OpenCL platform on two architectures is shown in Figure II.6. When a single device is used (CPU or GPU), SOCL generally introduces only a small overhead compared to the direct use of the vendor OpenCL implementation. However it could even perform better in some cases (e.g. with a single AMD GPU) presumably thanks to a better data pre-fetching strategy. On Alaric architecture, we can see that the CPU is better handled with the Intel OpenCL implementation than with the AMD one. Best performance is obtained with the SOCL platform using only GPUs through the AMD implementation and the CPU through Intel's one. On Averell1 architecture, best performance is also obtained with SOCL when it commingles NVIDIA and Intel implementations.

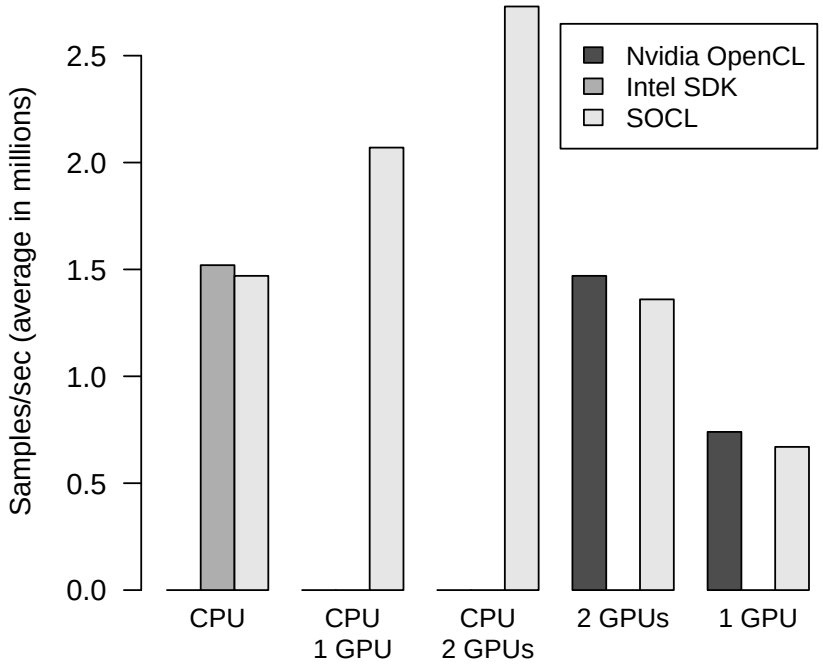
This example shows that an OpenCL application designed for using a single OpenCL platform can directly benefit from using the SOCL unified platform without any change in its code. It automatically uses several OpenCL implementations. In addition, this example shows that SOCL outperforms the AMD implementation when using GPUs and CPU at the same time by using the Intel OpenCL implementation for the CPU instead of AMD's one.

II.4.5 N-body

The OTOO simulation code [107] is an simulation of a dynamical system of particles, for astrophysical simulations. We use this example to put into practice the automatic granularity adaptation mechanisms provided by SOCL.



(a) Results on Alaric



(b) Results on Averell1

Figure II.6: LuxRender rendering benchmark indicates the average number of samples rendered per second (higher is better)

The OpenCL code is written for multiple devices, however we had to adapt it to be able to launch more kernels than the number of devices and to use command queues with out-of-order execution mode. The benchmark computes 20 iterations over 4000K particles, and for each iteration (i.e. each time step), the same computation is performed (a force calculation for each particle of the system using a tree method). In the OTOO simulation code, the force calculations for different particles are completely independent of each other. Therefore, the computation can be simply cut into smaller computations by distributing the particles to kernels. The kernels use shared read-only buffers describing all the particles and the tree, and each kernel uses its own write-only buffer to store the resulting force of each particle.

For the adaptive execution, the partitioning factor range is [1,5] (see Listing II.6). An overview of a way to write the code of the *divide function* `part_func` is given in Listing II.7. The real partitioning factor set used here is {4, 8, 16, 32, 64}. So, during the first 5 iterations of the 20 iterations, the number of kernels launched in the `part_func` function, will be successively a power of 2, from 4. During this first phase of the execution, statistics about the execution time of the `part_func` function are saved. For the remaining iterations, the `part_func` function is called with a partitioning factor in the range [1,5], determined by the SOCL run time support.

We assess the code performance on 4 heterogeneous devices (1 CPU and 3 GPUs) on the Hannibal architecture. The difficulty for this benchmark and for this heterogeneous architecture is to balance the load between the multi-core CPU and the 3 GPUs. This is a strong scalability issue: the same amount of computation has to be divided among all devices. As GPUs are faster on this kernel than the CPU, too few kernels creates either load imbalance between CPU and GPU or leads to use only GPUs. Too many kernels creates a high level of parallelism but reduces the computation efficiency (and the parallelism inside kernels) for each device.

The number of kernels launched for each iteration can be along 1 to 64. Figure II.7 shows performance gains when the number of kernels ranges from 2 to 64 (for all 20 iterations), compared to the version running one kernel per iteration. The last experiment (Adaptive) shows the speed-up when the number of kernels launched is adapted dynamically by SOCL. All speed-ups are evaluated for two scheduling heuristics, Eager and Heft. The figures show that the heuristics do not propose a perfect load-balancing strategy for all numbers of kernels. The efficiency starts to reduce after 32 kernels (resp. 8 kernels) for the Eager scheduling (resp. the Heft scheduling), for this size of input.

Overall, this shows that using SOCL adaptive strategy is an efficient way to automatically find an appropriate level of decomposition for the computation so as to avoid load-imbalance. Besides, this is also a way to overcome the shortcomings of scheduling heuristics.

II.5 Conclusion

OpenCL is a widespread specification to program heterogeneous architectures that has been implemented by many hardware vendors. However, it remains very hard to use efficiently because it is very low-level and it forces applications to perform a lot of cumbersome device management. In addition, the installable client driver (ICD) extension that allows several OpenCL implementations to be used jointly is not well integrated: entities of the different implementations cannot be mixed up and in particular devices cannot be synchronized using OpenCL

Listing II.7: NBody Partitioning Function

```
cl_int part_func(cl_command_queue cq, cl_uint partitioning_factor,
                 const cl_event before, cl_event *after) {

  // Enqueue commands transferring relevant data in input buffers (read-only).
  // These commands depend on the "before" event. (not shown)

  // Compute the real partitioning factor from the given factor that belongs to
  // the partitioning range
  factor = pow(2, partitioning_factor+1);

  // Compute the number of particles per block
  size = number_of_particles / factor;

  // Kernel executions must happen after transfers
  clEnqueueBarrierWithWaitList(cq,0,NULL,NULL);

  // Launch kernels
  for (i=0; i < factor; i++) {
      offset = size * i;

      // Set kernel arguments: buffers and offset in the input. (not shown)

      // Enqueue kernel execution command
      clEnqueueNDRangeKernel(cq, kernel, 1, NULL, global, local,
                             0, NULL, &evs[i]);

      // Enqueue a command to transfer output data that depends
      // on evs[i] (not shown)
  }

  // Set "after" marker event
  clEnqueueBarrierWithWaitList(cq,0,NULL,after);
}
```

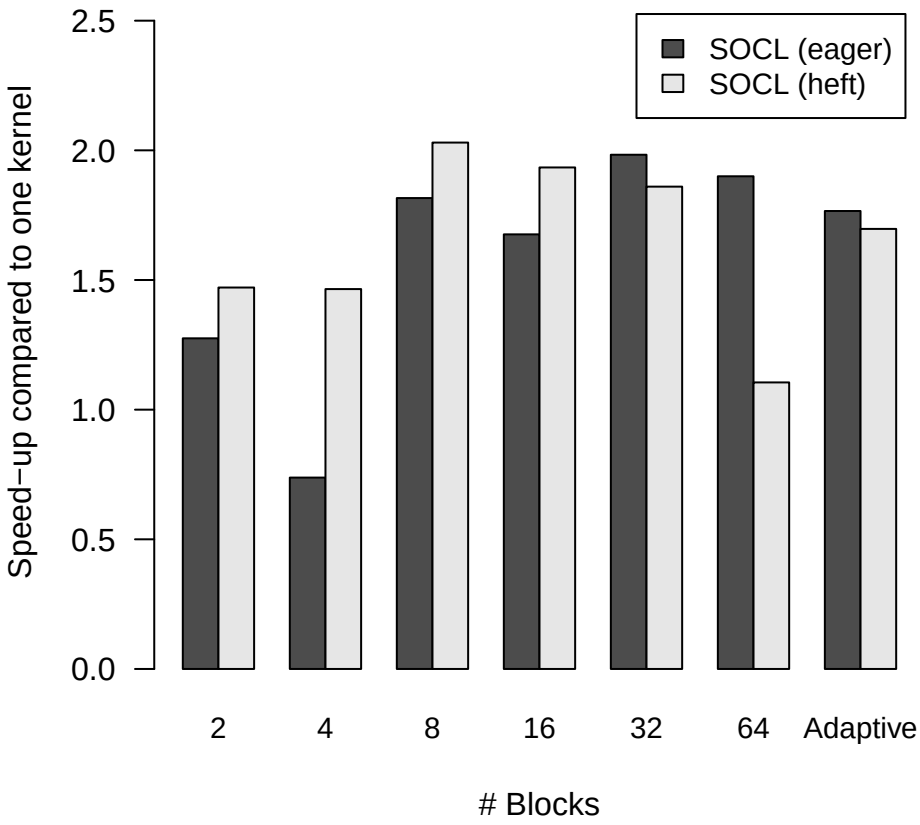


Figure II.7: Performance speed-ups for NBody benchmark when the number of kernels is changed from 2 to 64 (by power of 2) and when using the adaptive granularity approach, on Hannibal. Speed-ups are relative to the performance of one kernel.

mechanisms (events...) if they do not belong to the same platform.

In this chapter we presented SOCL, our OpenCL implementation that provides several extensions to OpenCL in order to alleviate its shortcomings. SOCL provides: (1) a unified OpenCL platform; (2) automatic device memory management; (3) automatic command scheduling; (4) preliminary support for automatic granularity adaptation.

The unified OpenCL platform is a non-intrusive extension that allows applications to use SOCL instead of the ICD to multiplex OpenCL implementations. LuxRender example shows that application can benefit from it without changing a single line of its code. Automatic device memory management is also transparent to applications. Black-Scholes example shows that problems whose size does not fit into device memory can be supported with SOCL as it automatically swaps out buffers in host memory. Automatic command scheduling and automatic granularity adaptation require applications to explicitly modify their codes to use them. By extending OpenCL contexts into scheduling contexts, modifications remain very light.

SOCL provides a mechanism to automatically adapt the granularity of the kernels. The runtime system can call partitioning functions with different partitioning factors selected in a given range. This mechanism is especially suited for iterative applications such as the N-Body benchmark we used as after a few iterations, the partitioning factor that gives best performance is selected. While promising, the use of a callback function to submit the alternative task graph does not allow the runtime system to predict if it is wise to partition the kernel or not. Moreover, if several consecutive kernels are partitioned, a bottleneck is introduced because all data are recomposed after the execution of the first partitioned kernel and then are partitioned again for the second kernel. Finally, OpenCL semantics is very poor regarding data partitioning. For instance, it is not easy to partition at the OpenCL level a matrix contained in a buffer into tiles with a non null stride. Two-dimensional copy APIs such as `clEnqueueWriteBufferRect` can be used but may introduce superfluous data transfers.

In the next chapter, we used a more declarative approach to declare task graphs so that the runtime system can dynamically transform it. Thanks to referential transparency of the functional programming model, the runtime can substitute a function application with an equivalent functional expression. This mechanism is used to substitute a kernel execution with an equivalent task graph.

CHAPTER III

HETEROGENEOUS PARALLEL FUNCTIONAL PROGRAMMING

Chapter abstract

In this chapter, we present a programming model that combines both parallel functional programming and task graph based runtime systems for heterogeneous architectures. We show that by expressing task graphs as functional programs, we can transform them at compilation time and at execution time to enhance performance. We present a runtime system called ViperVM that evaluates functional programs by performing parallel graph reduction in addition to management of heterogeneous devices (memory allocation and release, data transfers, kernel scheduling…).

III.1 Introduction

The heterogeneous parallel functional programming approach that we present in this chapter is designed to provide a high-level programming interface while still leveraging high-performance computational kernels that can be tuned by hand. We use a functional language to *describe* task graphs because pure functional programs are isomorphic to graphs of super-combinators (i.e. pure functions that only work on their parameters) and thus very similar to graphs of computational kernels for accelerators.

For instance, consider the functional expression `(f m) + (g m)` where `f` and `g` are two functions and `m` is a data. Thanks to properties of functional programs, we know that we can evaluate both function applications in parallel without even knowing what `f` and `g` are doing. In addition, we can associate a computational kernel with a function identifier such as `f` or `g` so that evaluating the previous expression triggers the scheduling of the associated kernels on available devices (CPU, GPU...) as well as appropriate data transfers.

Associating kernels to function identifiers is at the core of the model we propose but we go further: we can associate both a kernel and a functional expression with the same identifier. In this case, when a application of the identifier is evaluated, the runtime system can choose either to schedule the associated kernel or to evaluate (still in parallel) the associated expression. A concrete example would be to associate a linear algebra kernel (e.g. matrix multiplication) and an expression describing an equivalent algorithm with a finer granularity (e.g. tiled matrix multiplication) which itself involves linear algebra operations, hence granularity adaptation may occur recursively. Finding heuristics to know when it is appropriate for the runtime system to evaluate fine-grained expressions instead of coarse-grained kernels remains a hard issue though.

There are often several ways to write functional expressions equivalent to a given kernel with finer granularities. For instance, tile sizes in a tiled matrix multiplication algorithm can be chosen almost arbitrarily. Hence we sketch out a way for users to declare some variables that are to be automatically set by the runtime system or the compiler (cf algebraic partitioning factors). When several functions are composed, such as two consecutive matrix multiplications (`(a * b) * c`), partitioning factors for the three matrices `a`, `b` and `c` should be chosen so that no bottleneck is introduced. Basically, if bad factors are used, the (partitioned) result of the first tiled multiplication have to be repartitioned in a different manner in order to be compatible with the next one, potentially incurring additional data transfer overhead. We propose transformation rules on pure functional codes to avoid these bottlenecks (cf partitioning factor unification).

Finally, data-parallel functional expressions – that do not involve other kernels but describe at the finest granularity the operation that is performed – can also be associated with function identifiers. Several other works strive to generate efficient kernels from these expressions and could be integrated into our approach. Additionally, we could also use these expressions (that are often easy to write) to automatically infer algorithms equivalent to a kernel but with a finer granularity. Basically, we automatically introduce in functional programs application of the partitioning operator immediately followed by the unpartitioning operator to input data, hence it does not change the program meaning. Using transformation rules, we can delay or remove the application of the unpartitioning operator, so that we obtain an algorithm that works with partitioned inputs. We show that this approach works in simple cases but we still

need to prove that we can generalize it and that the rules we have lead to a sound rewriting system (congruence, convergence...).

III.1.1 Rationale

Our intuition to consider using an approach based on parallel functional programming on heterogeneous architectures comes from a set of observations. The more general ones are about the very nature of high-performance applications. They can often be thought of as functions (programs) consuming a huge data set and producing another as a result. Their execution can last for weeks without any user interaction, which avoids one of the main difficulties faced by functional programs (i.e. handling of non-deterministic side-effects produced by the user).

More specific observations can be made about the similarities between parallel functional programming and the way heterogeneous architectures are programmed using task-graph based runtime systems (cf Section I.3.3.3):

Side-effects Tasks composing task-graphs do not perform uncontrolled side effects. They can only modify buffers accessed in write mode but this is usually not enforced by an effect system. Similarly, pure functions cannot perform side-effects and in addition effect systems are used to track effectful functions.

Graph Host programs for most task graph based runtime systems dynamically build graphs of tasks. On the contrary, a pure functional program is itself a graph of supercombinators (i.e. functions that only depend on their parameters, hence that have no free variable). By assimilating kernels and (some) supercombinators, a pure functional program can describe a kind of graph of kernels.

Dependencies Dependencies between tasks often reflect data dependencies (read-after-write and write-after-write). Implicit inference uses task submission order, hence imposing some kind of sequential evaluation of the host program to be correct. In pure functional programs, dependencies between expressions are only data dependencies. In addition, there is no need for a sequential evaluation of the program, hence a parallel evaluation is possible.

Granularity In both approaches, fine grained parallelism is constrained into computational kernels while coarse grain parallelism (coordination) is either modelled with a task graph or a pure functional program both evaluated in parallel. Hence the common issue of a too fine granularity (i.e. thread management overhead is too high compared to the performance gain obtained with parallelism) encountered with implicit parallel functional programming is presumably avoided (or is faced in the same way by the task graph approach).

Memory model Task graph based runtime systems use a shared-object memory model so that objects are only accessed by their handles and can be transferred and stored by the runtime system in any physical memory. Objects can be lazily released using a reference counting strategy when the host program has stated it will not use them anymore. In functional programs, data are implicitly accessed by reference and are automatically released by a garbage-collector without presuming where data are stored (hence permitting copying garbage collection strategies for instance).

Execution Task graph based runtime systems have to select tasks to be executed among tasks ready to be executed (i.e. whose dependencies are fulfilled). Similarly, runtime systems for parallel functional programming have to select expressions to reduce among reducible expressions (i.e. expressions that are not in some normal form, depending on the reduction order).

III.1.2 Related Works

Our approach is related to the following other works.

- Clear separation between the coordination model and the computational model: as conceptualized in [62] and similarly to task-graph based runtime systems, there is a distinction between codes that perform computations (computational kernels) and codes that coordinate the execution of these kernels. As we use functional programming for the coordination code, our approach is in the continuity of Darlington's work [48, 49] that uses functional skeletons and the Manticore project [56] which uses explicit parallel functional programming for the coordination of implicit nested data-parallel functional computations.

- Parallel functional programming: pure functional programming is used to write coordination codes in our approach. Our runtime system uses a parallel evaluation strategy to interpret them [73].

- Shared-object memory model: our approach uses a memory model inspired from the shared-object memory model. To our knowledge, the model name has been coined for Jade in [90] and the model has been reused in the context of heterogeneous architecture programming, for instance in StarPU [17].

- Bird-Meertens formalism: in order to transform task graphs expressed as pure functional programs, we use a method inspired from the Bird-Meertens formalism [22]. We provide rewriting rules for functional expressions involving matrices (linear algebra algorithms) to transform them into more efficient expressions. In particular we would like to use it to automatically derive from a simple expression of an algorithm corresponding to a kernel (e.g. a matrix multiplication) another algorithm involving several kernels (e.g. blocked (tiled) matrix multiplication).

- Algorithmic choice: several kernels and functions are associated with the same identifier. We combine the following approaches:

  - at most one kernel per architecture (CUDA, OpenCL, CPU...), as in StarSS [18]
  - more than one kernel per architecture, as in StarPU [17]
  - several algorithms, as in PetaBricks [13]
  - high-level data-parallel kernels both for kernel generation, as in [37, 92, 100, 127, 130], and as input kernel description for task graph transformations

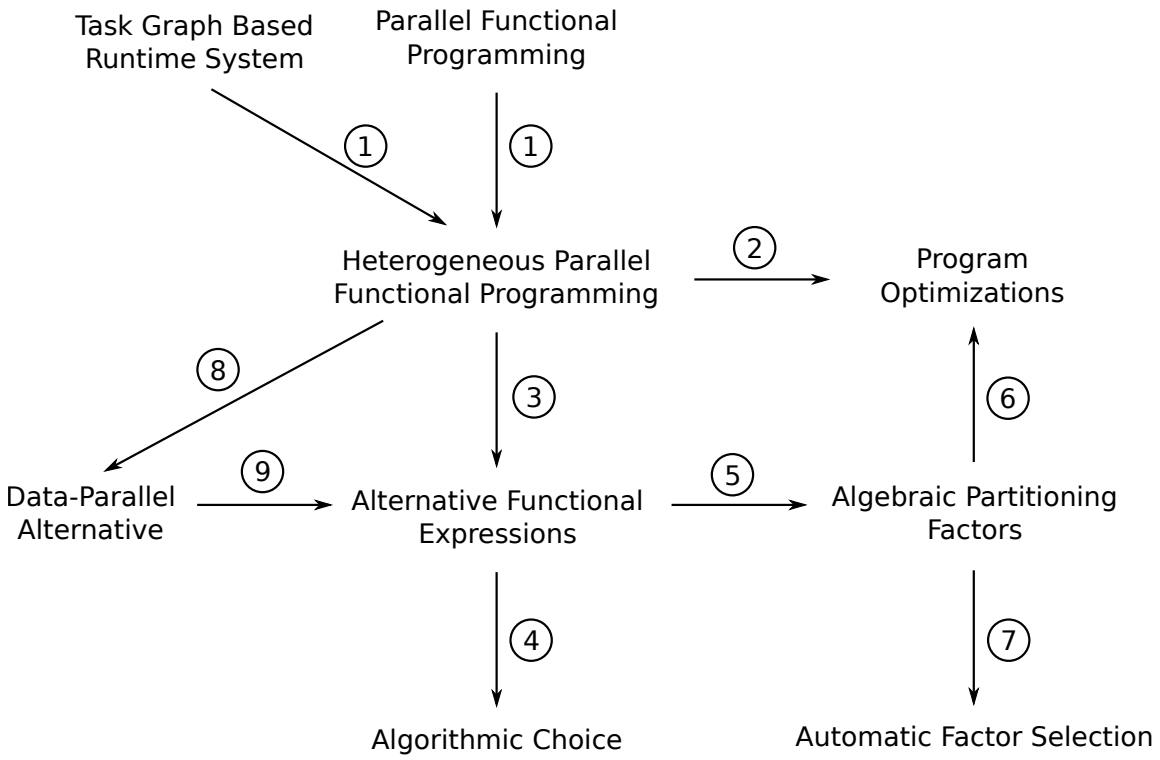


Figure III.1: Integration of the different concepts into the proposed approach

III.1.3 Overview of the Involved Concepts

To our knowledge, it is the first time that parallel functional programming is used to coordinate kernel executions on current heterogeneous architectures (GPU, Cell BE, MIC, etc.) by combining it with the shared-object memory model used by task graph based runtime systems (cf Section I.3.3.3).

Figure III.1 sums up how the different concepts are integrated into our approach. The heterogeneous parallel functional programming approach is basically composed of a task graph based runtime system where the task graph is described using pure functional programming (arrows labelled with "1"). Parallel evaluation of the functional programs is performed by the runtime system. This model uses supercombinator graphs instead of task graphs, hence programmers can use control (`if`) and functions in the code interpreted by the runtime system, avoiding opaque host code. Transformations on pure functional programs can be used to increase parallelism (arrow labelled with "2").

Some function identifiers in the pure functional codes that are evaluated in parallel are associated with computational kernels. That is, when the runtime system wants to reduce them, it has to schedule one of the associated kernel on one of the available devices. We extend this model to allow the association of alternative functional expressions (arrow labelled with "3"). That is, the runtime system now has to choose between the different kernels and an expression written in pure functional code (e.g. a matrix multiplication identifier can be associated with matrix multiplication kernels for each kind of device and to a functional expression for the

blocked matrix multiplication algorithm). If more than one alternative functional expression is given, we fall in the algorithmic choice category (arrow labelled with "4").

We want the runtime system to automatically adapt the granularity not only by selecting between a kernel or a functional expression to execute, but also by choosing partitioning factors, hence performing some kind of auto-tuning. We sketch out a formalism involving algebraic partitioning factors (arrow labelled with "5"). These are variables whose values are to be automatically set by the runtime system (or a compiler). We define some rules to optimize expressions containing algebraic partitioning factors (arrow labelled with "6"), in particular in case of function composition. For granularity adaptation purpose, it allows the selection of factors that are compatible with successive kernel executions, avoiding data recomposition and repartitioning bottleneck. Finally we sketch out some methods that could be (re)used for the partitioning factors to be automatically selected by the framework (arrow labelled with "7").

It is possible to define kernel alternatives that are written with high-order data-parallel operators (arrow labelled with "8"). A lot of other works have been conducted to generate efficient computational kernels for different architectures (CUDA, OpenCL...) and we could reuse them. Additionally, these high-level kernel representations could be used to automatically infer alternative algorithms to kernels by automatically inserting partition operators and using transformation rules to produce relevant codes (arrow labelled with "9").

III.2 Heterogeneous Parallel Functional Programming Model

Our approach involves three programming levels:

Computation Computational kernels executed by accelerators and CPUs can be written using low-level languages such as C, Fortran, OpenCL, CUDA, etc.

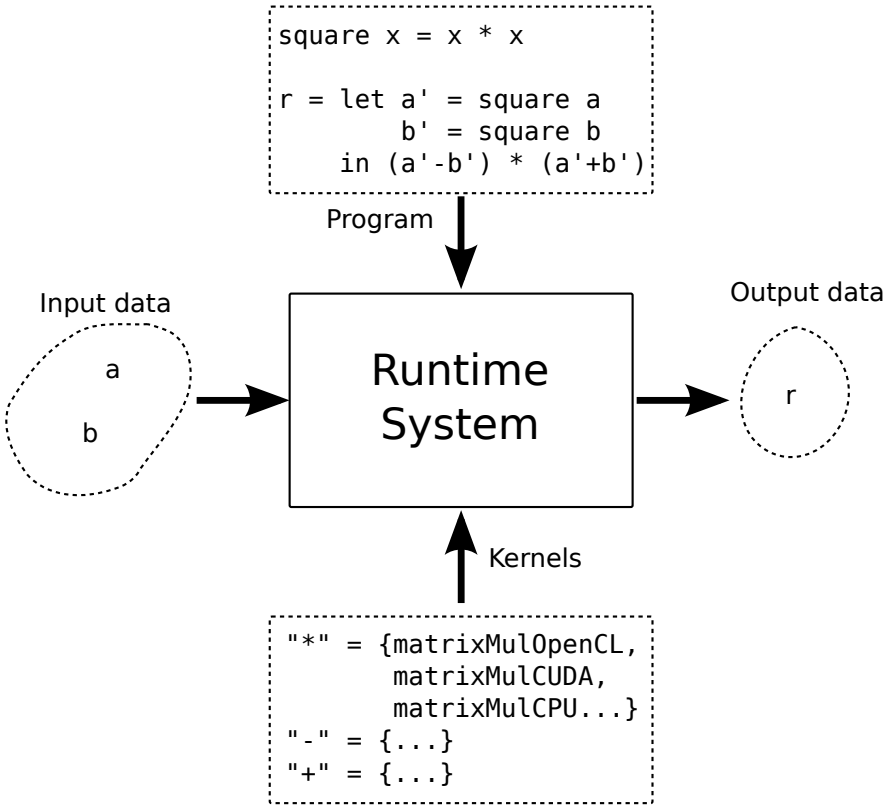
Configuration An host code is used to configure the runtime system and to perform IOs. As our runtime system is written in Haskell and presents itself as a library, host codes are written using Haskell.

Coordination Pure functional programs coordinate the execution of the kernels. They are interpreted by a runtime system using parallel evaluation strategies. This programming level is at the core of our work.

III.2.1 Configuring an Execution

In our approach, the role of the host code is limited to the configuration of the execution. Once the execution has started, it only has to wait for its completion. Figure III.2 shows the elements required for an execution: the runtime system has to be configured, some input data and some kernels have to be registered and a pure functional program to transform input data into output data has to be provided. Listing III.1 shows how the host code looks like with our runtime system ViperVM. The pure functional program is stored in a file called "codelet.vvm". The host code configures the platform to use OpenCL and selects the "eager" scheduling strategy. Then two matrices `a` and `b` are initialized and registered so that identifiers `a` and `b` refer to them in the functional program. Some computational kernels on matrices are registered too so

Figure III.2: Overview of the model



that symbols +, − and ⋆ are associated with appropriate single-precision floating-point matrix operations. Then the program source is loaded and evaluated. This last step is the one that can last for weeks depending on the evaluated program. Finally, here we display the resulting matrix but real programs will most likely store it on disk.

III.2.2 Parallel Evaluation of Functional Coordination Programs

Pure functional programs given to the runtime system are evaluated in parallel. Figure III.3 shows the task graph associated with the program used in Listing III.1. Its degree of parallelism is two because `a'` and `b'` can be evaluated in parallel, then matrix addition and matrix subtraction are performed in parallel two. When the runtime system has to reduce an application (in the lambda calculus sense) where the identifier is associated with a kernel (e.g. +, − or ⋆ in our example), it schedules the execution on a device selected with the configured scheduling strategy, then it performs appropriate data transfers and it executes the kernel on the device it has selected. When kernel execution completes, the application is substituted with the data handle of the result in the functional program and the reduction can continue using this data. Table III.1 shows the different steps of a potential execution on two accelerators. The content of each memory is indicated (supposedly, no garbage collection occurs during this execution). Figure III.4 shows the different reduction steps on the task graph.

Compared to the task graph model, in the heterogeneous parallel functional model the control flow is not opaque to the runtime system. *Functions* can be defined as in the previous

Listing III.1: Basic usage of the runtime system

```
-------------------------------------------------------------
-- codelet.vvm
-------------------------------------------------------------
square x = x * x

main = let a' = square a
           b' = square b
       in (a' - b') * (a' + b')

-------------------------------------------------------------
-- Main.hs
-------------------------------------------------------------
let config = Configuration {
       libraryOpenCL = "libOpenCL.so"
    }

-- Initialize the runtime system
pf ← initPlatform config
rt ← initRuntime pf eagerScheduler

-- Initialize some data
a ← initFloatMatrix rt [[1.0, 2.0, 3.0],
                        [4.0, 5.0, 6.0],
                        [7.0, 8.0, 9.0]]

b ← initFloatMatrix rt [[1.0, 4.0, 7.0],
                        [2.0, 5.0, 8.0],
                        [3.0, 6.0, 9.0]]

-- Register kernels and input data
builtins ← loadBuiltins rt [
   ("+", floatMatrixAddBuiltin),
   ("-", floatMatrixSubBuiltin),
   ("*", floatMatrixMulBuiltin),
   ("a", dataBuiltin a),
   ("b", dataBuiltin b)]

-- Load the functional program from file
src ← readFile "codelet.vvm"

-- Evaluate the program
r ← eval builtins src

-- Display the result
printFloatMatrix rt r
```

Figure III.3: Task graph corresponding to the example in Listing III.1

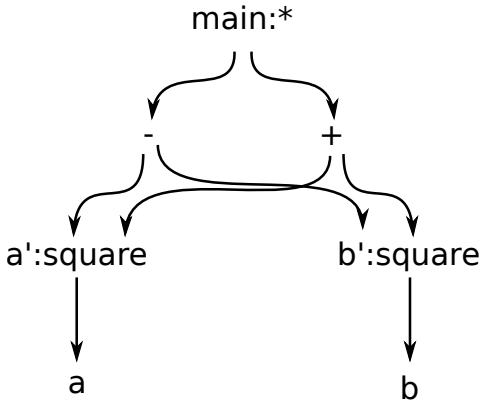
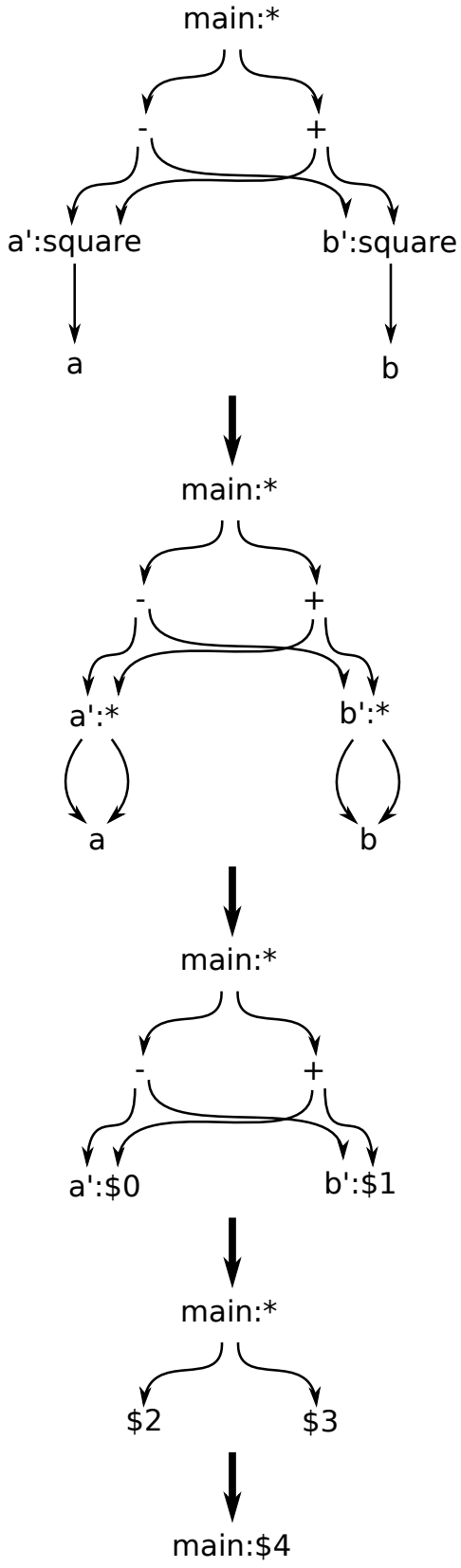


Table III.1: Execution steps of the task graph in Figure III.3

Actions	**Host**	**Device 1**	**Device 2**
Initial state	{a, b}	{}	{}
Data transfers	{a, b}	{a}	{b}
Placeholder allocations	{a, b}	{a,$0}	{b,$1}
Kernel executions ($\$0 \leftarrow a * a,\ \$1 \leftarrow b * b$)	{a, b}	{a,$0}	{b,$1}
Reductions ($a' \leftarrow \$0,\ b' \leftarrow \$1$)	{a, b}	{a,a'}	{b,b'}
Data transfers	{a, b}	{a,a',b'}	{b,b',a'}
Placeholder allocations	{a, b}	{a,a',b',$2}	{b,b',a',$3}
Kernel execution ($\$2 \leftarrow a' - b',\ \$3 \leftarrow a' + b'$)	{a, b}	{a,a',b',$2}	{b,b',a',$3}
Reductions ($a' - b' \leftarrow \$2,\ a' + b' \leftarrow \$3$)	{a, b}	{a,a',b',$2}	{b,b',a',$3}
Data transfer	{a, b}	{a,a',b',$2,$3}	{b,b',a',$3}
Placeholder allocation	{a, b}	{a,a',b',$2,$3,$4}	{b,b',a',$3}
Kernel execution ($\$4 \leftarrow \$2 * \$3$)	{a, b}	{a,a',b',$2,$3,$4}	{b,b',a',$3}
Reduction ($main \leftarrow \$4$)	{a, b}	{a,a',b',$2,$3,main}	{b,b',a',$3}
Data transfer	{a, b, main}	{a,a',b',$2,$3,main}	{b,b',a',$3}

Figure III.4: Reduction steps of the task graph in Figure III.3



example where `square` is a function not associated with a kernel. Moreover conditionals are available with the `if` operator. This operator combined with recursive sub-routines can be used to implement loops. In the future, runtime systems could decide to speculatively execute one or both branches of the `if` or at least to prefetch some data.

The memory model used by the heterogeneous parallel functional model is similar to the shared-object model except that objects are immutable. It is thus much easier to maintain coherency between the different memories because there is no need for a protocol like MESI. For each data handle, it is sufficient to know if there is an instance of the data in the considered memory and its content is ensured to be valid. Data have to be automatically collected as in other functional languages. There is a clear distinction between data that are manipulated by kernels (that cannot contain references to other data) and data manipulated by the functional program (that are supposed to remain small as they are only used for control, not for heavy computation). Hence, garbage collection algorithms could avoid stop-the-world policies as kernels can continue their execution while the reduction of the functional program is stopped.

III.2.2.1 Built-in "kernel" functions

Compared to usual parallel functional programming, in our model we associate computational kernels that can be executed by accelerators with some identifiers that can be used like functions in the functional code. For instance, in the following pseudo-code we associate three kernels with the `matrixMul` identifier so that when it is applied to two data, the runtime system will execute one of the three kernels on an accelerator of its choice.

```
matrixMul = {matrixMulOpenCL,
             matrixMulCUDA,
             matrixMulCPU}
```

We now consider cases where we associate additional *alternative functional expressions* with the same symbols. We may have the following situations:

- A symbol is associated with one or more kernels: it is the current situation where runtime systems have to select the most appropriate kernel to execute (for instance using performance models).
- A symbol is associated with a data-parallel functional expression used to generate kernels.
- A symbol is associated with several functional expressions that involve several kernels (algorithmic choice).
- All cases at the same time, that is a symbol is associated with some kernels and to some functional expressions. It is the situation we consider now where the runtime system has to choose whether it should execute a kernel or an alternative functional expression. In particular, alternative expressions can have a more fine grained granularity and this mechanism can be used to automatically perform granularity adaptation. In addition, we consider transformations that can be performed on these alternative functional expressions.

In our previous example, the matrix multiplication symbol was associated with several kernels (OpenCL, CUDA and CPU). We now allow it to be associated with other functional

Listing III.2: Matrix multiplication written with higher-order data-parallel operators

```
matrixMulDP a b = outerWith dotProduct (rows a) (columns b)

dotProduct xs ys = reduce (+) (zipWith (*) xs ys)
outerWith f xs ys = map (\x → map (\y → f x y) ys) xs
```

Listing III.3: `outerWith` operation (alternative code)

```
outerWith f xs ys = zipWith2D f xs' ys'
   where
      xs' = replicate (size ys) xs
      ys' = transpose (replicate (size xs) ys)

zipWith2D f xs ys = zipWith (zipWith f) xs ys
```

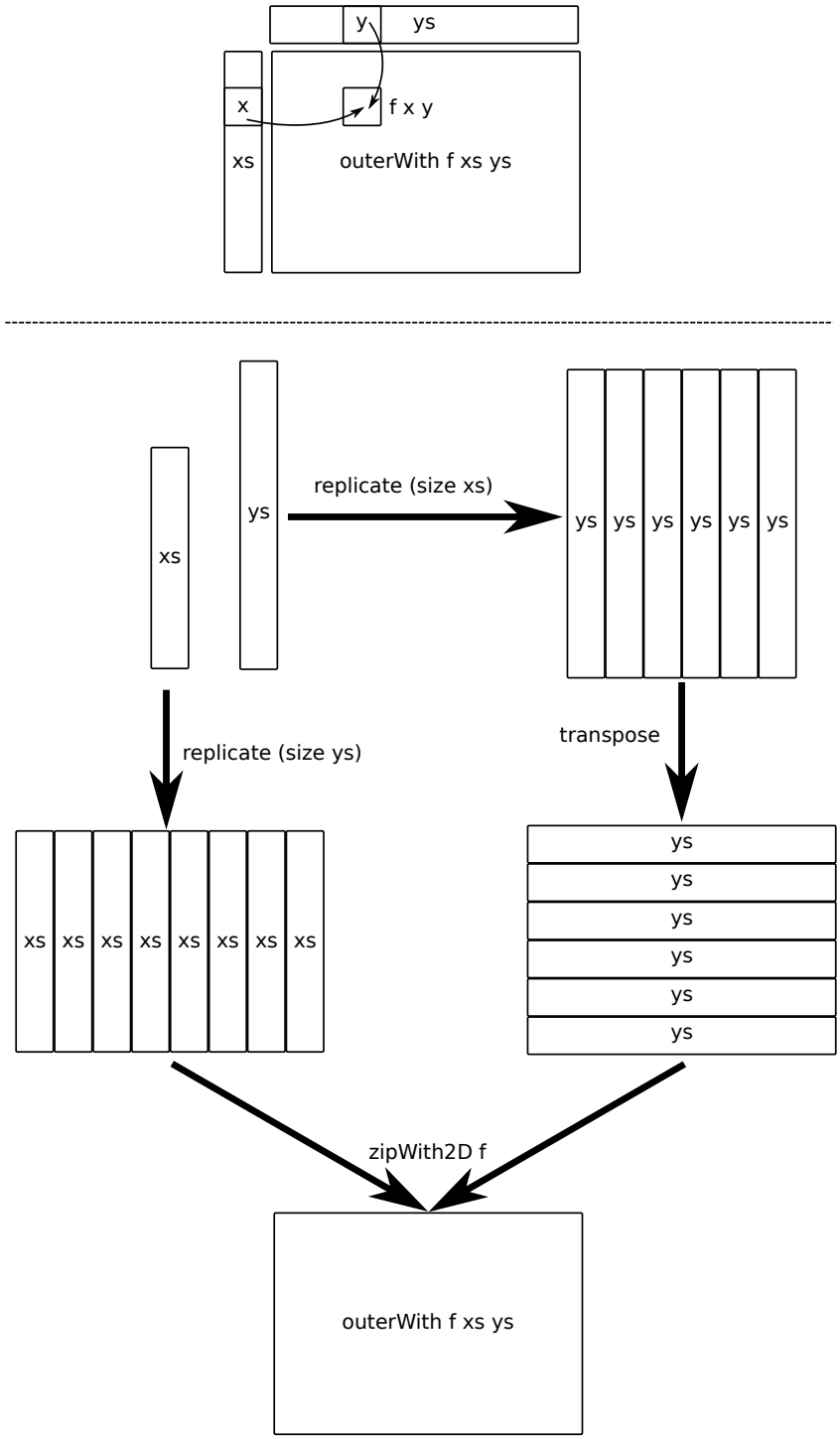
alternatives defined in the remainder of this section.

```
matrixMul = {matrixMulOpenCL,
             matrixMulCUDA,
             matrixMulCPU,
             matrixMulDP,
             matrixMulN,
             matrixMulTiled...}
```

Data-Parallel Alternative The first alternative we consider is the data-parallel one (`matrixMulDP`) that can be used to generate kernels. Data-parallel kernels can be written using higher-order operators such as `map`, `reduce` or `zipWith`. For instance, Listing III.2 shows how a matrix multiplication can be written. `outerWith` is a generalization of the outer product of two vectors where the product can be replaced with any operation. It can alternatively be written as in Listing III.3 (inspired from [85]) and as represented in Figure III.5.

Algorithmic Choice Matrix multiplication can be written with several algorithms equivalent to the previous one but that operate on matrix tiles. In Listing I.9 we show different algorithms for the matrix multiplication as they are defined in PetaBricks [13]. Listing III.4 shows the equivalent code to define the same algorithms using functional programming (note that the first variant has been omitted as it is equivalent to the previously defined `matrixMulDP`). In our code we use `split` and `unsplit` operators to partition and recompose (respectively) a matrix. Parameters of these operators indicate the number of tiles in each dimension and are called *partitioning factors*. Parameters could have been omitted for `unsplit` as the number of tiles in each dimension is given by the dimensions of the matrix that is recomposed, however we impose this syntax to use rewriting rules later in this chapter.

Figure III.5: outerWith operation



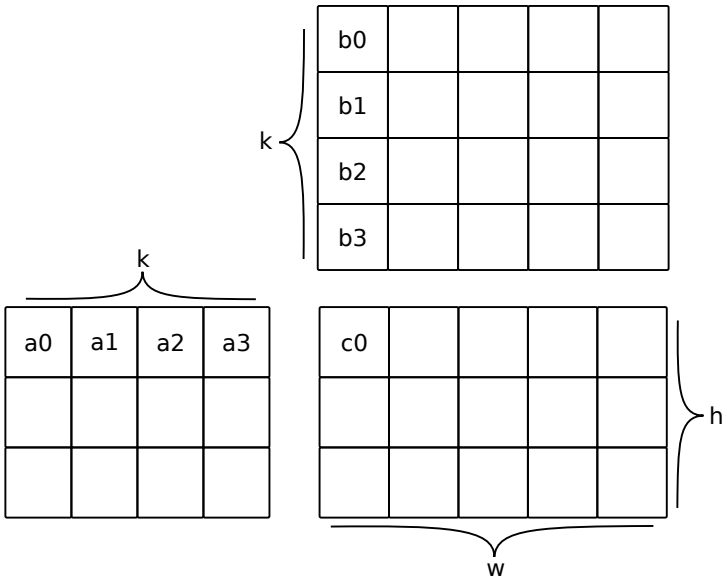
Listing III.4: Matrix multiplication algorithmic choice

```
matrixMul1 a b = matrixAdd (matrixMul a1 b1) (matrixMul a2 b2)
   where
      [[a1,a2]] = split 2 1 a
      [[b1],[b2]] = split 1 2 b

matrixMul2 a b = unsplit 2 1 [[ab1,ab2]]
   where
      [[b1,b2]] = split 2 1 b
      ab1 = matrixMul a b1
      ab2 = matrixMul a b2

matrixMul3 a b = unsplit 1 2 [[ab1],[ab2]]
   where
      [[a1],[a2]] = split 1 2 a
      ab1 = matrixMul a1 b
      ab2 = matrixMul a2 b
```

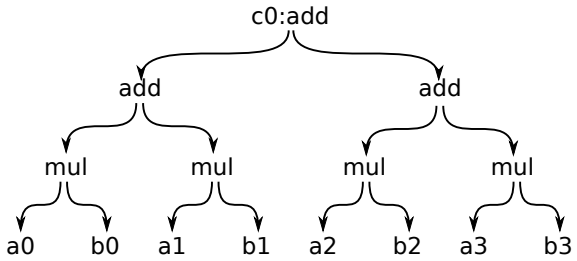
Listing III.5: Alternative to `matrixMul` with fixed partitioning factors

```
matrixMul4 a b = unsplit w h (f a' b')
   where
      (w,h,k) = (5,3,4)
      a' = split k h a
      b' = split w k b
      f as bs = outerWith dotProduct as (transpose bs)
      dotProduct xs ys = reduce matrixAdd (zipWith matrixMul xs ys)
```

Figure III.6: Tiled matrix multiplication (cf Listing III.5)




```
c0 = dotProduct [a0,a1,a2,a3] [b0,b1,b2,b3]
   = reduce matrixAdd (zipWith matrixMul [a0,a1,a2,a3] [b0,b1,b2,b3])
```



Listing III.6: Alternative functional expression to `matrixMul` with algebraic partitioning factors

```
matrixMulTiled a b = \ (# w) (# h) (# k) → unsplit w h (f a' b')
   where
      a' = split k h a
      b' = split w k b
      f as bs = outerWith dotProduct as (transpose bs)
      dotProduct xs ys = reduce matrixAdd (zipWith matrixMul xs ys)
```

Algebraic Partitioning Factors More algorithmic variants could be defined, for instance by using different partitioning factors as `matrixMul4` in Listing III.5 (represented in Figure III.6). However the fact that the granularity is fixed in the functional expressions is an issue because: (1) we want the runtime system to have the ability to choose the granularity; (2) fixing the granularity can lead to partitioning conflicts when two tasks or more use the same data and induce superfluous data transfers when data are recomposed and partitioned again. Hence we introduce *algebraic partitioning factors* as a mean for programmers to use partitioning factors without explicitly assigning them a value. We write them using a syntax similar to type-arguments in GHC Core as in the following example:

```
f = \ (# w) (# h) → unsplit w h . f' . split w h
```

This syntax allows us to give a scope and a name to partitioning factors. Declared partitioning factor variables can appear anywhere as usual variables with the `Integer` type. As a consequence, we can rewrite the previous example without specifying partitioning factor values (cf `matrixMulTiled` in Listing III.6). Expressions using algebraic partitioning factors can be composed using variable renaming if necessary to avoid name clashes. For instance, suppose the runtime system wants to replace both `matrixMul` with `matrixMulTiled` in the expression: `r = matrixMul a (matrixMul b c)`, Listing III.7 shows how partitioning factors are merged without conflict by using variable renaming.

Partitioning Factors Selection Before executing programs containing algebraic partitioning factors, their values have to be set by a compiler or by the runtime system. Several kinds of strategies can be envisioned based on different criteria to perform this auto-tuning: platform metrics (number of processing units and their capabilities, sizes of the memory units, transfer capabilities between memories), kernel execution time statistics for each kind of device (performance models), runtime system state (available memory in each memory, processing unit occupation, number of tasks ready to be executed, etc.), code metrics (degree of parallelism, etc.).

The search space to determine the values of the partitioning factors can be very large. We could imagine adding constraints on partitioning factors so that only valid values are selected as in the following code:

```
f = \ (# w) (# h) | (w mod 2 == 0) → unsplit w h . f' . split w h
```

The advantage is that it is easy to combine constraint, for instance when several variables are unified. The major drawback is that it may take a long time for the system to find correct

Listing III.7: `matrixMul` composition with partitioning factors

```
r = matrixMul a (matrixMul b c)

r = \ (# w) (# h) (# k) → unsplit w h (f a' tmp)
      where
         a' = split k h a
         tmp = split w k (matrixMul b c)
         f as bs = outerWith dotProduct as (transpose bs)
         dotProduct xs ys = reduce matrixAdd (zipWith matrixMul xs ys)

r = \ (# w) (# h) (# k) (# w2) (# h2) (# k2) → unsplit w h (f a' tmp)
      where
         a' = split k h a
         b' = split k2 h2 b
         c' = split w2 k2 c
         tmp = (split w k . unsplit w2 h2) (f b' c')
         f as bs = outerWith dotProduct as (transpose bs)
         dotProduct xs ys = reduce matrixAdd (zipWith martixMul xs ys)
```

Listing III.8: Partitioning factors with type ascription

```
\ (# w :: EvenInteger) (# h) → unsplit w h . f . split w h

instance Arbitrary EvenInteger where
   arbitrary = do
      a ← arbitrary
      return (2 * a)
```

values that fulfill every constraint.

Another solution, inspired from QuickCheck's `Arbitrary` class [42], is to add explicit type ascriptions for partitioning factors so that custom value generators are used. Listing III.8 shows how to generate even integers with this method for `w`. The drawback of this approach is that generators do not compose well: if a constraint is inferred that two variables must have the same value but that they both use their own generator like in Listing III.9, we would have to find a "super-generator" that would generate valid values for `A` and `B` at the same time.

Automatic Alternative Inference Starting with the data-parallel representation of a kernel, we want to transform it to automatically generate equivalent alternative task graphs involving partitioning factors. Basically we could use Bird-Meertens formalism (cf I.4.3) to transform functional expressions after partitioning operators have been inserted. The function that partitions a data and then recomposes it is equivalent to the identity function. It is thus valid to insert `unsplit w h . split w h` at various positions in a functional expression. Then by using transformations, we can try to remove or delay the `unsplit` operation as much as

Listing III.9: Partitioning factors with type ascriptions and constraints

```
\ (# w :: A) (# h :: B) | (w == h) → unsplit w h . f . split w h

instance Arbitrary A where
   arbitrary = ...

instance Arbitrary B where
   arbitrary = ...
```

Listing III.10: Automatic insertion of partitioning factors in matrix multiplication code

```
matrixMul a b = outerWith dotProduct (rows a) (columns b)

-- Introducing "unsplit w h . split w h" for each parameter
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
      outerWith dotProduct (rows a') (columns b')
   where
      a' = (unsplit w1 h1 . split w1 h1) a
      b' = (unsplit w2 h2 . split w2 h2) b

-- Several transformation steps...

-- Result of the derivation
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith dotProduct' a' (columns b'))
  where
    dotProduct' x y = reduce matrixAdd (zipWith matrixMul x y)
    a' = split w1 h1 a
    b' = split h1 h2 b
```

possible and to perform the `split` operation as soon as possible. For instance, we can use this mechanism on function parameters to transform a function working on whole matrix into a function working on tiles. Listing III.10 shows this method applied to the matrix multiplication. The full derivation is given in Appendix D. We observe that we could automatically find the block-matrix multiplication algorithm with algebraic partitioning factors as defined previously.

III.2.2.2 Optimizations for Pure Functional Coordination Programs

Some computational kernels can be efficiently implemented by modifying data in place. That is, they use some data with read-write access mode. The heterogeneous parallel functional model relies on functional programming and manipulated data in programs are immutable. However compilers and runtime systems can internally use computational kernels that perform *destructive updates* as long as it is non observable in program codes. A conservative ap-

proach to use these kernels is to duplicate data accessed with read-write access mode before their use.

As data duplications are costly, they should be avoided as much as possible. This problem is well known in the functional programming community because implementations of immutable arrays using implicit array copying face the same issue. Static analyses can be performed to detect data that can be safely modified in-place. In the expression `r = f (g a b) c`, the data produced by the `g` function can be safely modified in-place by the `f` function as there is no way for it to be used elsewhere in the program as it has no identifier. It exists some solutions to statically ensure that the optimization is taking place [117]. For instance *linear type systems* or *monadic encapsulation* of the destructive update.

In addition, optimizations could also be performed at execution time. For instance, if reference counting is used and if a data has a single reference left then it can be modified in-place. Additionally, if a data is duplicated into several physical memories, it could be cheaper to detach one data instance, to modify it in-place and to attach it to the output data handle. Finally, scheduling policies may decide to give an higher priority to tasks that do not modify data in-place. By doing this, several tasks could read a data before it gets modified by a task scheduled with a lower priority.

By using functional programs to represent task graphs, we can transform them (at compilation time for instance) in order to schedule optimized task graphs. For instance, we can enhance the degree of parallelism, avoid duplication of work with common sub-expression elimination, etc. In a preliminary prototype of ViperVM, called HaskellPU (cf Appendix A), we used GHC's rewrite rule mechanism to increase parallelism. For instance, we used the following rule to ensure that consecutive matrix additions (which are associative) are not performed sequentially. Instead we replace the addition sequence with a reduction that is implemented as a binary tree. This example shows how transformations on the functional program has an impact on the scheduled task graph.

```
"reduce_plus"     forall x y z.  x + y + z = reduce (+) [x,y,z]
"reduce_plus_add" forall xs y. (reduce (+) xs) + y = reduce (+) (xs ++ [y])
```

The runtime system should strive to remove any superfluous data recomposition (`unsplit`). The following equation can be used to substitute the left-hand side with the right-hand side that is the identity function:

```
split w h . unsplit w h = id
```

It often happens that the pattern encountered is `split w1 h1 . unsplit w2 h2` as in Listing III.7. In this case, the runtime system may decide to replace two or more partitioning factors by a single one in order to apply the rule: $w1 = w2$ and $h1 = h2$. We call this *partitioning factor unification*. Constraints are merged and therefore must be compatible (i.e. there must be at least one acceptable value for each partitioning factor). Listing III.11 shows the result of this optimization applied to the previous `matrixMul` composition example (Listing III.7).

Listing III.11: Partitioning factor unification example

```
r = \ (# w) (# h) (# k) (# w2) (# h2) (# k2) →unsplit w h (f a' tmp)
    where
      a' = split k h a
      b' = split k2 h2 b
      c' = split w2 k2 c
      tmp = (split w k . unsplit w2 h2) (f b' c')
      f as bs = outerWith dotProduct as (transpose bs)
      dotProduct xs ys = reduce matrixAdd (zipWith lmatrixMul xs ys)

-- Partitioning factor unification
r = \ (# w) (# h) (# k) (# k2) →unsplit w h (f a' tmp)
    where
      a' = split k h a
      b' = split k2 k b
      c' = split w k2 c
      tmp = (split w k . unsplit w k) (f b' c')
      f as bs = outerWith dotProduct as (transpose bs)
      dotProduct xs ys = reduce matrixAdd (zipWith matrixMul xs ys)

-- Simplification rule applied
r = \ (# w) (# h) (# k) (# k2) →unsplit w h (f a' tmp)
    where
      a' = split k h a
      b' = split k2 k b
      c' = split w k2 c
      tmp = f b' c'
      f as bs = outerWith dotProduct as (transpose bs)
      dotProduct xs ys = reduce matrixAdd (zipWith matrixMul xs ys)
```

III.3 ViperVM

ViperVM is a runtime system that we are developing from scratch to support the programming model proposed in this chapter. In our HaskellPU prototype (cf Appends A), we combined two existing softwares: GHC for the functional programming part of the approach and StarPU for kernel scheduling and memory management of heterogeneous architectures. With ViperVM we want to avoid some pitfalls of this previous approach:

- We want to avoid the creation of an intermediate task graph representation in memory. That is, we want to control the evaluation of the pure functional program so that the latter suffice to itself as it already represents a graph.
- We want to be able to perform transformations on functional programs both at compilation time and at execution time.
- We want the runtime system to support capabilities related to the functional model we use and that are not provided and often not easily implementable in task graph based runtime systems (e.g. garbage collector, destructive updates...). Additionally, as the memory model we use is more constrained (immutable data), complex coherency mechanisms that are typically implemented in task graph based runtime systems are superfluous.

ViperVM provides an abstraction layer called *generic platform* that can be used to inspect and to control the components of the architecture. Three entities are at the core of the generic platform: memory, link and processor. The topology of the architecture is exposed as a network of memories interconnected with links. To each memory, some processors able to process data stored in it are attached (or none if it is a memory such as a hard disk).

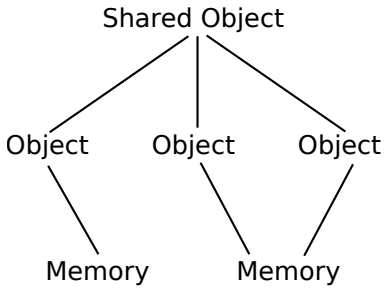
Internally, ViperVM uses the foreign function interface (FFI) to have access to low-level framework interfaces in order to control accelerators. A generic platform module is provided to the other modules: each enabled driver (Host, OpenCL, CUDA...) returns a set of *memories*, a set of *links* between memories and a set of *processors* attached to some memories. A generic API is provided by this module to allocate buffers in memories, to transfer data through links and to execute kernels on processors. The host driver is mandatory because it is used by other drivers as it returns a single memory entity corresponding to the host memory and a single loop-back link to perform data duplications in host memory.

Users have to configure the runtime system to choose device drivers to use – OpenCL, CUDA, NUMA...– and to set driver specific options. The current implementation only supports OpenCL and options are limited to the OpenCL library path as shown in Listing III.1. Note that we decided to use dynamic linking for drivers because it is much harder to distribute software, especially in binary form (e.g. as a Linux distribution package) when it can be statically linked to several optional libraries chosen at compilation time and that may not be available in other environments.

III.3.1 Implementation

In this section, we describe the different modules that compose ViperVM.

Figure III.7: Shared-Object Memory Model



Data Management Our runtime system uses a shared-object memory model: data manipulated by functional programs are only handles associated with a set of effective objects that can be replicated in any memory. On top of the generic platform module, a `SharedObjectManager` module can be used to allocate a shared object, to attach an instance to a shared object or to detach an instance. Figure III.7 represents a shared object that is associated with three objects. These objects contain the same data and are distributed in two memories, in particular the object is duplicated in the second memory. Duplicated objects can be stored differently (e.g. different count of padding bytes for matrices...).

An `ObjectManager` module provides primitives to allocate and release objects and to transfer the content of an object into another over a link between two memories (or using the loop-back link). Shared objects are considered *immutable* from a user perspective in the sense that they reference objects containing the same immutable content. To modify an object, it must first be detached from its associated shared object (if any). When it has been modified, it can be attached to another shared object (usually a freshly allocated one). A locking mechanism is used to ensure that objects are not modified concurrently. However, simultaneous read accesses are permitted.

In high-performance applications, it is common to work on sub-data. For instance, matrices may be partitioned in tiles that are processed in parallel by different processors. Our runtime system provides support for sub-data by allowing a shared object to indicate how it relates to another. When an operation uses a shared object specifying this information, it can use objects of the other shared object instead of its own objects. In fact, it may not have any object of its own. This mechanism lets applications work with a data and its sub-data at the same time.

Currently, the supported data types are vectors and matrices of single-precision floating-point values. Once the implementation is mature enough, we will add support for other data types. Regular matrix partitioning in tiles is supported. Each tile is then a sub-data with the same matrix type as its parent but with different dimensions.

Kernel Scheduling Schedulers are implemented as modules using both the generic platform and data manager modules. We provide two basic schedulers: round-robin and eager. The first one distributes kernels as they are submitted in a cyclic fashion to the available processors. The second one lets idle processors peek into a queue of kernels ready to be executed and that is shared by all of them. We have not yet implemented the whole machinery required for advanced scheduling heuristics such as HEFT. It would basically consist in selecting for each task the device that minimizes its predicted termination date (taking into account predicted

data transfer durations, etc.), thus it would be necessary to store previous kernel execution times and to predict future kernel execution and data transfer durations.

In the current implementation, kernels are compiled lazily the first time they are scheduled on a processor. Other scheduling strategies could choose to compile kernels for all devices at once and/or to store binaries for future executions. In addition, some accelerators such as the Cell BE would require the kernel binary to be explicitly loaded on the accelerator before being executed. Several strategies to efficiently handle kernels that are already loaded on an accelerator could be envisioned.

Parallel Graph Reducer ViperVM provides a `Graph` module based on software transactional memory (STM) so that graph nodes can be modified concurrently. For this preliminary release we implemented a parallel reducer for graphs of supercombinators based on the template instantiation approach presented in Peyton-Jones and Lester book [112]. Thus we use a lazy evaluation order where the outermost reducible expression is reduced first. Currently, the main difference is that when we need to evaluate parameters of a built-in (e.g. kernels) operation, we evaluate them in parallel, spawning a light thread per parameter.

Once kernel parameters have been evaluated, the kernel and its parameters (data handles and immediate data) are submitted to the selected scheduler. Upon kernel completion, the function call associated with the kernel is substituted with the resulting expression, generally a data handle or an immediate data. It is up to the kernel scheduler to return a functional expression that is not in weak head normal form (WHNF) such as a function application. In this case the parallel graph reducer will reduce the returned expression. This mechanism could be used by schedulers to perform task granularity adaptation.

Parser ViperVM is language neutral as long as the input language is purely functional and can be converted into a graph of supercombinators. Parsers transform functional expression that are given to them as strings into graphs that can be involved in the program execution. In Listing III.1, we use a parser for a Haskell-like syntax to parse the program.

The following constructions are supported both by the parser and by the graph reducer. Top-level functions can be defined. For instance, the following code define two functions `f` and `main`.

```
main = f a b

f x y = x*y + y*y
```

Anonymous functions are also supported. For instance, in the following code we define an anonymous function equivalent to the previous `f` that is applied without giving it a name:

```
main = (\ x y → x*y + y*y) a b
```

It is possible to introduce new variables in the functional expression by using `let` operator. The following code uses `let` to introduce two variables `c` and `d`.

```
g a b = let c = a * b
            d = b * a
        in (c - d) * (c + d)
```

`let` operator is important because it allows to define graphs instead of trees: variables introduced with this operator can be used more than once in an expression but the expressions they reference are only evaluated once.

Conditional execution is supported with the `if` operator. It takes three arguments: a condition expression and two expressions for the different branches. The condition is evaluated and depending on its result one of the two branches is evaluated. In the following code we use an hypothetical `isSymmetric` function to check if a matrix is symmetric in order to use a more efficient code to compute its square.

```
square m = if (isSymmetric m) then (ssyrk m) else (sgemm m m)
```

Kernel Library and Built-ins ViperVM provides a library of simple dense linear algebra kernels that can be easily used. Matrix multiplication and addition operations used in Listing III.1 come from this library and only have to be associated with appropriate symbols to be used in a functional program.

A built-in list data type is supported alongside basic list operations (`head`, `tail`, `cons...`). This data type can be used to represent matrices as list of list of tiles and to write algorithm that use tiling. In particular, `split` built-in can be used to partition a matrix into tiles of the specified dimensions. The result is given as a list of list of matrices. Another important list operation is `reduce` that can be used to reduce a non-empty list in parallel by using a balanced binary tree pattern. It is not a built-in and its code is the following one:

```
reduce f [] = error "Empty list"
reduce f xs = head (reduce' f xs)

reduce' _ [] = []
reduce' _ [x] = [x]
reduce' f (x1:x2:xs) = reduce' f xs'
   where
      xs' = (f x1 x2) : reduce' f xs
```

Listing III.12 shows how a tiled matrix multiplication can be written using list operations. `sgemm` takes two matrices (`x` and `y`) and three tile dimensions (`w`, `h` and `k`) as parameters. `x` is partitioned using `split` in blocks of dimensions $k \times h$ and `y` in blocks of dimensions $w \times k$. `sgemm'` is called with the partitioned matrices as parameters and the result is recomposed into a single dense matrix with `unsplit`. `outerWith f` performs an operation similar to an outer product of two vectors but where the multiplication is replaced with `f`, its first parameter. `sgemm'` uses it to combine every row of `xs` with every columns of `ys` using `dotProduct` function. The latter is a dot product on matrices using `add` and `mul` matrix addition and multiplication kernels, respectively. For completeness, note that `'()` is the Lisp syntax to create an empty list, `null` returns true if the given parameter is an empty list and `map` applies its first parameter to every element of the list given in second parameter.

As the parallel graph reducer performs lazy evaluation and stops on weak head normal forms, results may not be what users would expect. For instance, if we want to compute a list of values such as `[1,(1+1),3]`, the parallel graph reducer will stop on the list constructor without evaluating anything. To force the evaluation, the built-in function `deepseq` can be used so that `deepseq [1,(1+1),3]` is correctly reduced to the normal form `[1,2,3]`. Note

Listing III.12: Tiled Matrix Multiplication

```
sgemm x y w h k = unsplit w h (sgemm' x' y')
   where
      x' = split k h x
      y' = split w k y

sgemm' xs ys = outerWith dotProduct xs (transpose ys)

dotProduct xs ys = reduce (+) (zipWith (*) xs ys)

outerWith f xs ys =  g xs ys
   where g = map (\ y → map (\ x → f x y))

transpose [[]] = []
transpose xs = map head xs : transpose (map tail xs)

zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

that list elements are then evaluated in parallel.

III.3.2 Evaluation

In this section we present preliminary performance results we obtained on several linear algebra algorithms. A performance comparison with state-of-the-art task graph based runtime systems would not be fair at this point of advancement of the project because advanced scheduling strategies using performance models, data prefetching, etc. have not yet been implemented. Moreover, the kernels we use are very naive and absolute performance is not comparable with what it is possible to achieve on the test architecture.

For these tests, we used an Intel Sandy Bridge E5-2650 dual-processor architecture with a total amount of 16 hyper-threaded cores and 64 gigabytes of memory. Three NVidia Tesla M2075 GPUs are used as accelerators. CPU and GPUs are used through the OpenCL implementation provided by each hardware vendors. In the case of the CPU, it means that commands such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` are used by ViperVM, hence superfluous data copies are performed in host memory, incurring additional overhead.

III.3.2.1 Matrix Addition

Matrix addition is an operation that is performed element-wise. As such, it is very easy to write an equivalent algorithm that works on tiles of the input matrices. In Listing III.13, input matrices `a` and `b` are partitioned in tiles of dimensions `(w,h)`. Then `+` is applied to every element-wise couple of tiles of `a` and `b` with the `zipWith2D` function, hence associated matrix

Listing III.13: Matrix Addition

```
main = unsplit w h (zipWith2D (+) a' b')
   where
      a' = split w h a
      b' = split w h b

zipWith2D f xs ys = zipWith (zipWith f) xs ys
```

Table III.2: Matrix Addition (tile dimensions: 8192x8192)

Input dimensions	ViperVM 3 GPUs + CPU	ViperVM 3 GPUs	StarPU 3 GPUs
16K x 16K	1.9s	2.1s	1.4s
24K x 24K	4.0s	4.4s	2.9s

addition kernels are to be used to compute each tile of the result. Finally, `unsplit` recomposes the resulting tiles to form a dense matrix.

For comparison, Listing III.14 presents the host code using StarPU that creates the same task graph. Compared to the functional program, the code is much more verbose, in particular the data `c` used to store the result of the matrix addition has to be explicitly allocated and partitioned before kernels are submitted. Moreover, tasks working on matrix tiles have to be completed before `c` can be unpartitioned, hence the `task_wait_for_all` synchronization. A solution to avoid this barrier is to use callbacks associated with tasks but the code becomes even more cumbersome.

In order to test the substitution mechanism during the evaluation of the functional program, we have integrated a naive granularity adaptation strategy to the + operation: if one of the dimensions of the input matrices is greater than 8192, we substitute the + application with the previous equivalent code working on tiles where tile dimension is arbitrarily set to 8192. Hence the user code in the ViperVM case is just `a + b`.

Performance results are presented in Table III.2 in comparison with performance obtained with StarPU. As StarPU does not support data transfers with a stride introduced in OpenCL 1.1, we have used an equivalent CUDA implementation of our OpenCL kernel, hence results with StarPU only use the GPUs (we checked that both kernel execution times are the same). These results show that our runtime system still has an additional overhead of 50% compared to StarPU on this example.

III.3.2.2 Matrix Multiplication

Matrix multiplication algorithm working on matrix tiles has already been presented in Listing III.12 and the main part is reproduced here:

```
main = unsplit w h (f a' b')
   where
      a' = split k h a
```

Listing III.14: Tiled matrix addition example using StarPU

```
struct starpu_data_filter f = {
   .filter_func = starpu_matrix_filter_vertical_block,
   .nchildren = nw
};

struct starpu_data_filter f2 = {
   .filter_func = starpu_matrix_filter_block,
   .nchildren = nh
};

starpu_data_map_filters(a, 2, &f, &f2);
starpu_data_map_filters(b, 2, &f, &f2);
starpu_data_map_filters(c, 2, &f, &f2);

for (i=0; i<w; i++) {
   for (j=0; j<h; j++) {
      starpu_data_handle_t sa = starpu_data_get_sub_data(a, 2, i, j);
      starpu_data_handle_t sb = starpu_data_get_sub_data(b, 2, i, j);
      starpu_data_handle_t sc = starpu_data_get_sub_data(c, 2, i, j);

      starpu_insert_task(&add, STARPU_R, sa, STARPU_R, sb, STARPU_W, sc, 0);
   }
}

starpu_task_wait_for_all();
starpu_data_unpartition(c, 0);
```

Table III.3: Matrix Multiplication (4096x4096)

w x h	1024x1024	4096x1024	1024x4096
GPU (1x)	4.5s	4.4s	4.3s
GPU (2x)	3.6s	2.9s	3.2s
GPU (3x)	3.1s	2.5s	3.3s
CPU	31s	36s	35s
GPU (3x) + CPU	3.3s	3.7s	10s

```
      b' = split w k b

f xs ys = outerWith dotProduct xs (transpose ys)
```

With this second example, we show that ViperVM benefits from the simple eager scheduling strategy it uses when performing a tiled matrix multiplication. Performance results are reported in Table III.3 for different values of `w` and `h` and we can observe that performance increases with the number of GPUs in almost all cases. The matrix multiplication kernel we implemented is not optimized and is particularly slow on the CPU so that using it actually decreases the overall performance. Hence adding the CPU shows a shortcoming of the eager scheduling strategy. We would need to use performance models and to implement a better scheduling strategy (such as HEFT) to avoid giving a task to the CPU if there are not enough tasks for the GPUs.

III.4 Conclusion

In this chapter, we presented a programming model based on parallel functional programming that can be used to schedule computational kernels on heterogeneous architectures. By using functional programs, we are able to optimize task graphs both statically and dynamically: we showed that rewrite rules can be used statically to enhance the degree of parallelism of the graph and that some kind of granularity adaptation can be performed at execution time by replacing a task with an equivalent task graph. Thanks to referential transparency, the latter substitution is performed safely.

ViperVM is our current implementation of a runtime system for this programming model. It still lacks some important features but is already able to perform parallel reduction of some functional programs and to schedule kernels on devices appropriately. In the current development state, performance is not yet on par with those of equivalent codes written for task graph based runtime systems. However it is very promising, especially regarding productivity as user codes using functional programming are much more closer to mathematics.

Summary of our contributions:

- A programming model that combines parallel functional programming for coarse-grained parallelism (task graph description) and low-level kernels (CUDA, OpenCL, C, Fortran...) for fine-grained parallelism
- A runtime system providing this programming model and able to substitute a kernel exe-

cution with an alternative functional expression. This mechanism is envisioned to be used to perform automatic granularity adaptation.

- A mechanism called ”algebraic partitioning factors” to integrate scoped partitioning factors in functional expressions that are to be set by the runtime system and a method called ”unification” to enhance composed functional expressions and remove superfluous data recompositions.
- Transformations rules that could be used to automatically infer alternative functional expressions containing algebraic partitioning factors for a kernel from a high-level data-parallel functional representation of the kernel.

Notes

Most of the work on granularity adaptation presented in this chapter has not yet been implemented into our runtime system. More work is needed to see how it can be generalized and automatized. In particular, we have to find heuristics for the runtime system to automatically set partitioning factor values. We also have to check if transformation rules given in Appendix B are general enough and can be systematically applied in an appropriate order to find relevant algorithms.

CONCLUSION AND PERSPECTIVES

Heterogeneous architectures are being increasingly used but are challenging to exploit efficiently. They are the most difficult to program because they are the most general: they are not hierarchical and can be composed of every kinds of devices, with different processing capabilities and memory capacities, interconnected with different networks to the host.

OpenCL is the first widespread specification of an application programming interface (API) dedicated to exploit heterogeneous architectures by using a command graph model. Our first contribution is an OpenCL implementation called SOCL that offers additional functionalities to application programmers compared to usual OpenCL implementations: a unified OpenCL platform, automatic device memory management, automatic command scheduling and a preliminary support for kernel granularity adaptation. In Chapter II, we described the implementation and we showed results obtained with several representative applications.

From our experience in implementing SOCL, we note that it is very hard to deal with automatic granularity adaptation with the programming model proposed. In Chapter III we presented a programming model based on parallel functional programming that alleviates some of the drawbacks of the low-level approaches and gives the framework the ability to perform task graph transformations both at compilation time and at execution time. In particular we show that this approach can be used to substitute a coarse-grained task with an equivalent task graph involving fine-grained tasks. We showed that by describing kernels with data-parallel functional operators we could automatically derive alternative task graphs. Additionally, we pointed out how algorithmic choice could be combined with our approach.

Contributions

Specifically, our contributions are:

- SOCL: an implementation of the OpenCL standard extended to provide a unified platform, automatic device memory management, automatic command scheduling and preliminary support for automatic kernel granularity adaptation.
- Heterogeneous Parallel Functional Programming model: a programming model combining parallel functional programming, high-performance hand-tuned computational kernels and a runtime system for heterogeneous architectures.

- ViperVM: a preliminary implementation of a runtime system supporting the Heterogeneous Parallel Functional Programming model

Perspectives

With heterogeneous architectures, a point of no return has been reached: it is now generally too costly to write high-performance applications using low-level frameworks for a specific architecture (hence making it not portable). Runtime systems are used to conglomerate most architecture concerns into a single place and force application developers to use different programming models. However there is a proliferation of runtime systems and it is hard for application developers to choose the one to use. We need a **common low-level framework** to precisely expose heterogeneous architecture models. A framework such as Hwloc [33] integrates preliminary support for accelerator (GPUs) detection but it should be extended to provide low-level capabilities to applications (buffer allocation and release, data transfer...). Moreover a **taxonomy of architectures** should be established because it is no longer relevant to select computational kernels on a "is it a GPU or CPU" basis (as it is done in OpenCL) because the distance between the two kinds of architectures is blurred.

On top of this low-level runtime system, we want to provide a **complete implementation of the heterogeneous parallel functional programming model** presented in Chapter III. The first step is to make it on par with current task-graph based runtime systems. More drivers should be implemented in addition to the OpenCL one (CUDA, x86, CELL, out-of-core, networking...). A library of kernels has to be constituted. It should contain low-level kernels (written in CUDA, OpenCL, C, etc.) alongside alternative functional expressions used to perform some optimizations (granularity adaptation, etc.). These expressions can also be used to generate kernels from a high-level data-parallel representation by borrowing techniques from Nepal [38], Accelerate [37], Accelerator [130]....

The current implementation of ViperVM is closer to a proof of concept than to a mature runtime system. Some engineering time will be devoted to implement missing functionalities (garbage collection, other drivers, etc.). In addition, there are several further directions we would like to explore:

Granularity Adaptation We would like to develop heuristics and static optimizations so that the runtime system could automatically perform granularity adaptation. Ideally, we would like the runtime system to automatically choose partitioning factors. Thus, we need to work on algebraic transformations of the functional programs so that constraints on factors can be inferred to reduce the search space domain. Work on granularity adaptation using functional expressions (both statically and dynamically) should be implemented. Granularity adaptation policies will have to be evaluated empirically because it is difficult to predict their behaviors. Additional factors could be used to select transformations such as accuracy (as found in PetaBricks [13]). Dependent types could be used to track data sizes and to enhance granularity adaptation decisions. In particular, it could avoid having to take some of them at runtime.

Scheduling Policies We would like to experiment different scheduling policies. In particular, we could use lookahead now offered by the more declarative model to execute in priority

tasks that reduce the memory pressure or whose results are required by many other tasks. Speculative execution is another mechanism that we would like to evaluate in this context.

Destructive Updates Kernels performing destructive updates are commonly found especially in codes using dense linear algebra. We want to allow their use while minimizing the number of data duplications. The shared-object memory model may offer new optimization opportunities as data may already be duplicated in different memories. Interaction with the garbage collection algorithm could be interesting too: if there is only one remaining task using a data then it can perform a destructive update.

Automatic Benchmarking QuickCheck is an Haskell library used for testing purpose. Given a function prototype, it automatically generates test cases (i.e. sets of input data) and function results are checked against constraints. We would like to use a similar mechanism to automatically generate micro-benchmarks for the computational kernels. Additionally, we could automatically check that different kernels performing the same operation on different architectures return correct results.

Check-Pointing With our model, the runtime system controls the parallel reduction of the program. We would like to provide a way to stop the reduction and to store the current state of the program on disk alongside the data required to continue the execution. Thanks to functional purity, it may be possible to automatically perform check-pointing without interrupting the reduction. We would have to lock data while they are being stored to avoid destructive updates.

Fault-Tolerance A fault-tolerance mechanism could be easily implemented by executing each task a few times (potentially on different devices). The functional nature of the program makes it easy to perform automatically because the runtime system is already aware of the data that should not be modified to keep the non-observable data immutability property. A technique similar to automatic check-pointing could be used so that when several tasks are executed performing the same computation are executed, the program could continue by using the first result returned and only be restarted at this point later on if a mistake is detected in the result that has been used.

APPENDIX A

HASKELLPU

A preliminary work of ours has been to make an experiment with Glasgow Haskell Compiler (GHC) rewrite rules in order to show that task graphs could be enhanced with this approach. We wrote an Haskell module called HaskellPU which provides an opaque `Matrix` data type alongside several operations on it. Matrix data wraps a StarPU data handler and a boolean state indicating whether the data has been already computed or not. A set of matrix operations backed by efficient BLAS for GPUs and CPUs and grouped into some StarPU "codelets" are provided. Evaluations of matrix operations by GHC's runtime system trigger task submissions to StarPU using `unsafePerformIO` API. Tasks are then executed asynchronously and on their completion data states are set accordingly. Because of Haskell lazy evaluation order, a `compute` function is provided to force the evaluation of an expression (i.e. task submission to StarPU) and a `wait` function is provided to synchronously wait for a data to be computed. StarPU relies on explicit data release function calls performed by applications to know when it can safely free some buffers. Initially applications were responsible to call these methods only when every tasks using the data to be released have completed. In HaskellPU, wrappers for StarPU data are released asynchronously by the garbage collector. However, as computations are performed asynchronously, the associated StarPU data should not be released at the same time as it may still be in use. To fix this issue, we added a new functionality in StarPU that allows applications to indicate to StarPU that a data will no longer be used by the application and that it can release it as soon as every submitted task using it has completed. We use this method into garbage collector hooks associated with HaskellPU data.

Listing A.1 shows some of the functions provided by HaskellPU and Listing A.2 is an example of a Haskell program using them. Several BLAS operations are provided such as matrix addition, subtraction and multiplication as well as Cholesky factorization on symmetric matrices (`potrf`) and matrix transposition. In addition, HaskellPU provides two methods to initialize matrices: `floatMatrixSet` that sets the same value to every cell and `floatMatrixInit` that applies a given function to each matrix coordinate to get the value of each cell. As `Matrix Float` type implements an instance of the `Num` type class, usual operators (`+`, `-`, `*`...) can be

Listing A.1: Some of the functions exported by HaskellPU module

```
-- Matrix initialization function
floatMatrixInit :: (Word → Word → Float) → Word → Word → Matrix Float
floatMatrixInit f width height = ...

floatMatrixSet :: Float → Word → Word → Matrix Float
floatMatrixSet value width height = ...

-- Functions backed by BLAS kernels
floatMatrixAdd :: Matrix Float → Matrix Float → Matrix Float
floatMatrixSub :: Matrix Float → Matrix Float → Matrix Float
floatMatrixMul :: Matrix Float → Matrix Float → Matrix Float
floatMatrixPotrf :: Matrix Float → Matrix Float
floatMatrixTranspose :: Matrix Float → Matrix Float
```

Listing A.2: Example using HaskellPU

```
identityMatrix n = floatMatrixInit (\x y → if (x == y) then 1.0 else 0.0) n n
hilbertMatrix n  = floatMatrixInit (\x y → 1.0/(fromIntegral (x + y) + 1.0)) n n

main = do
   let
      a = floatMatrixInit (\x y → fromIntegral (10 + x*2 + y)) 20 20
      b = identityMatrix 20
      c = hilbertMatrix 20
   computeSync $ (a + b) * (floatMatrixPotrf c)
```

used and appropriate BLAS kernels are used. `computeSync` is used to force the evaluation of an expression and to wait for the result.

Glasgow Haskell Compiler (GHC) provides rewrite rules that can be used to detect patterns in codes and to replace them with another code. We use this mechanism to show that the declarative (functional) way of declaring a task graph permit static optimizations that are cannot be provided by frameworks relying on a dynamically built task graph. As an example, the two following rules transform successive matrix additions into a single parallel reduction using the matrix addition operation. This rule is sound because the matrix addition is associative, hence a binary tree can be used to perform the reduction. By doing this, we introduce parallelism in the generated task graph as shown on Figure A.1. Indeed this rule automatically transforms a sequential code into a parallel one.

```
"reduce_plus"     forall x y z.  x + y + z = reduce (+) [x,y,z]
"reduce_plus_add" forall xs y. (reduce (+) xs) + y = reduce (+) (xs ++ [y])
```

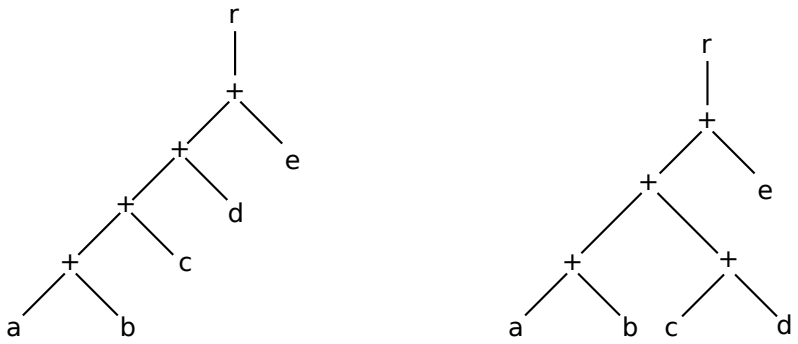


Figure A.1: Application of the rewrite rules to the expression: `r = a + b + c + d + e`. Without the rewrite rules (left), the height of the tree is in O(n); with the rewrite rules (right), the height of the tree is in O(log n).

APPENDIX B

REWRITE RULES

In this appendix we present several rewrite rules that can be used by runtime systems and compilers to transform functional expressions, especially those containing (un)partitioning operators (`split` and `unsplit`). For each equation, any occurrence of the left-hand side of the equal sign can be replaced by the expression on the right-hand side, and vice versa. Nevertheless rules are given so that it is often better for parallelism to use the former order (i.e. left-hand side replaced by right-hand side). To simplify the expression of some programs and of some rewrite rules, we first introduce several helper functions.

```
map2D :: (a → b) → Matrix a → Matrix a
map2D f = map (map f)

zipWith2D :: (a → b → c) → Matrix a → Matrix b → Matrix c
zipWith2D f = zipWith (zipWith f)

rows :: Matrix a → Vector (Vector a)
rows = id

columns :: Matrix a → Vector (Vector a)
columns = transpose
```

`map2D` is the matrix equivalent of the vector `map` function. It applies a function to every element of a matrix. `zipWith2D` is the matrix equivalent of the vector `zipWith` function: it combines two matrices element-wise with the given function. `rows` and `columns` respectively return a vector of rows and a vector of columns from a matrix. As matrices are defined using the row-major convention, there is nothing to do to return rows. Returning columns is equivalent to transposing the matrix.

The following first set of rewrite rules try to delay or to avoid the execution of the costly `concat` operation that joins several vectors into a single one. The syntax `f^n` indicates a composition of `f` with itself repeated `n` times.

```
-- reduce/concat
reduce f . concat h = reduce f . map (reduce f)

-- map/concat
map f . concat v = concat v . map (map f)

-- zipWith/concat
zipWith f (concat h a) (concat h b) = concat h $ zipWith2D f a b

-- outerWith/concat
outerWith f (concat i a) (concat j b)
  = unsplit i j $ outerWith (outerWith f) a b

-- outerWith/unsplit
outerWith f (unsplit h1 v1 a) (unsplit h2 v2 b)
  = unsplit h1 h2 $ outerWith (outerWith f') (map transpose a) (map transpose b)
    where
      f' x y = f (concat v1 x) (concat v2 y)

-- zipWith2D/unsplit
zipWith2D f (unsplit h v a) (unsplit h v b)
  = unsplit h v $ zipWith2D (zipWith2D f) a b

-- transpose/unsplit
transpose . unsplit h v = unsplit v h . transpose . map2D transpose

-- map^n(concat)
map^n (concat h . f) = map^n (concat h) . map^n f

-- zipWith/map^n(concat)
zipWith (map^n (concat h) . g) a b = map (map^n (concat h)) $ zipWith g a b
```

The following rules delay or avoid the execution of the costly `transpose` operation.

```
-- transpose/transpose
transpose . transpose = id

-- transpose/outerWith
transpose $ outerWith f a b = outerWith (flip f) b a

-- zipWith2D/transpose
zipWith2D f (transpose a) (transpose b) = transpose . zipWith2D f a b

-- map(transpose)
```

```
map (transpose . f) = map transpose . map f

-- outerWith(zipWith)/transpose
outerWith (zipWith f) (transpose a) (transpose b)
   = map transpose $ transpose $ zipWith (outerWith f) a b

-- outerWith(outerWith)/transpose A
outerWith (outerWith f) (transpose a) b
  = transpose $ map transpose $ transpose $ outerWith (outerWith f) a b

-- outerWith(outerWith)/transpose B
outerWith (outerWith f) a (transpose b)
  = map (transpose . map transpose . transpose) $ outerWith (outerWith f) a b

-- outerWith(outerWith)/transpose AB
outerWith (outerWith f) (transpose a) (transpose b)
  =  tr $ outerWith (outerWith f) a b
    where
      tr = map transpose . map2D transpose . transpose . map transpose

-- zipWith/map*(transpose)
zipWith (map* transpose . g) a b = map (map* transpose) $ zipWith g a b
```

The following rules delay the execution of `reduce`.

```
-- map(reduce)
map (reduce f . g) = reduce (zipWith f) . transpose . map g

-- outerWith(reduce)
outerWith (\x y → reduce f $ g x y) = map2D (reduce f) . outerWith g
```

Some other rules to add rewriting occurence opportunities.

```
-- map/map: f != concat, f != reduce, f != transpose
map f . map g = map (f.g)

-- map/zipWith: f != concat, f != transpose
map f . zipWith g = zipWith (f.g)

-- map/zipWith2D: f != concat, f != transpose
map f . zipWith2D g = zipWith (f. zipWith g)

-- zipWith/map
zipWith f (map g a) (map h b) = zipWith (\x y → f (g x) (h y)) a b

-- outerWith/map
outerWith f (map g a) (map h b) = outerWith (\x y → f (g x) (h y)) a b
```

```
-- Block matrix transposition
transpose . map2D transpose = map2D transpose . transpose
```

APPENDIX C

MATRIX ADDITION DERIVATION

```
matrixAdd :: Num a => Matrix a →Matrix a →Matrix a
matrixAdd a b = zipWith2D (+) a b

-- Partitioning
= \ (# h1) (# v1) (# h2) (# v2) →zipWith2D (+) a' b'
  where
    a' = (unsplit h1 v1 . split2D h1 v1) a
    b' = (unsplit h2 v2 . split2D h2 v2) b

-- Unification: h1=h2, v1=v2
= \ (# h) (# v) →zipWith2D (+) a' b'
  where
    a' = (unsplit h v . split2D h v) a
    b' = (unsplit h v . split2D h v) b

-- "zipWith2D/unsplit" rule
= \ (# h) (# v) →unsplit h v (zipWith2D (+) a' b')
  where
    a' = split2D h v a
    b' = split2D h v b
```

APPENDIX D

MATRIX MULTIPLICATION DERIVATION

```
matrixMul :: Num a => Matrix a →Matrix a →Matrix a
matrixMul a b = outerWith dotProduct (rows a) (columns b)

dotProduct xs ys = reduce (+) (zipWith (*) xs ys)

-- Partitioning parameters
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
     outerWith dotProduct (rows a') (columns b')
  where
    a' = (unsplit w1 h1 . split w1 h1) a
    b' = (unsplit w2 h2 . split w2 h2) b

-- By definition of rows and columns
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
   outerWith dotProduct a' b'
  where
    a' = (unsplit w1 h1 . split w1 h1) a
    b' = (transpose . unsplit w2 h2 . split w2 h2) b

-- "transpose/unsplit" rule
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
   outerWith dotProduct a' b'
  where
    a' = (unsplit w1 h1 . split w1 h1) a
    b' = (unsplit h2 w2 . transpose . map2D transpose . split w2 h2) b

-- "outerWith/unsplit" rule
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = dotProduct (concat h1 x) (concat w2 y)
```

```
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split w2 h2) b

-- By definition of "dotProduct"
matrixMul' a b = \ (# w1) (# h1) (# w2) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = reduce (+) (zipWith (*) (concat h1 x) (concat w2 y))
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split w2 h2) b

-- Partitioning factor unification: h1 = w2
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = reduce (+) (zipWith (*) (concat h1 x) (concat h1 y))
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "zipWith/concat" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = (reduce (+) . concat h1) (zipWith2D (*) x y)
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "reduce/concat" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = (reduce (+) . map (reduce (+))) (zipWith2D (*) x y)
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "map/zipWith2D" rule and "dotProduct" identification
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith (outerWith f) a' b')
  where
    f x y = reduce (+) (zipWith dotProduct x y)
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "outerWith(reduce)" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
     g = map2D (reduce (+))
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- By definition of "map2D"
matrixMul' a b = \ (# w1) (# h1) (# h2) →unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
    g = map (map (reduce (+)))
    a' = (map transpose . split w1 h1) a
```

```
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "map(reduce)" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
    g = map (reduce (zipWith (+)) . transpose)
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "map(reduce)" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
    g = reduce (zipWith (zipWith (+)) . transpose . map transpose
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- By definition of "zipWith2D"
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
    g = reduce (zipWith2D (+)) . transpose . map transpose
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "matrixAdd" identification
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) x y)
    g = reduce matrixAdd . transpose . map transpose
    a' = (map transpose . split w1 h1) a
    b' = (map transpose . transpose . map2D transpose . split h1 h2) b

-- "outerWith/map" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = g (outerWith (zipWith dotProduct) (transpose x) (transpose y))
    g = reduce matrixAdd . transpose . map transpose
    a' = split w1 h1 a
    b' = (transpose . map2D transpose . split h1 h2) b

-- "outerWith(zipWith)/transpose" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = (g . map transpose . transpose) (zipWith (outerWith dotProduct) x y)
    g = reduce matrixAdd . transpose . map transpose
    a' = split w1 h1 a
    b' = (transpose . map2D transpose . split h1 h2) b

-- "map/map" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = (g . transpose) (zipWith (outerWith dotProduct) x y)
    g = reduce matrixAdd . transpose . map (transpose . transpose)
    a' = split w1 h1 a
    b' = (transpose . map2D transpose . split h1 h2) b
```

```
-- "transpose/transpose" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = (g . transpose) (zipWith (outerWith dotProduct) x y)
    g = reduce matrixAdd . transpose
    a' = split w1 h1 a
    b' = (transpose . map2D transpose . split h1 h2) b

-- "transpose/transpose" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith (outerWith dotProduct) x y)
    a' = split w1 h1 a
    b' = (transpose . map2D transpose . split h1 h2) b

-- "Block matrix transposition" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith (outerWith dotProduct) x y)
    a' = split w1 h1 a
    b' = (map2D transpose . transpose . split h1 h2) b

-- By definition of "map2D"
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith (outerWith dotProduct) x y)
    a' = split w1 h1 a
    b' = (map (map transpose) . transpose . split h1 h2) b

-- "outerWith/map" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith h x (map transpose y))
    h x y = outerWith dotProduct x y
    a' = split w1 h1 a
    b' = (transpose . split h1 h2) b

-- "zipWith/map" rule
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith h x y)
    h x y = outerWith dotProduct x (transpose y)
    a' = split w1 h1 a
    b' = (transpose . split h1 h2) b

-- Identification of "rows" and "columns"
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith h x y)
    h x y = outerWith dotProduct (rows x) (columns y))
    a' = split w1 h1 a
    b' = (transpose . split h1 h2) b

-- "matrixMul" identification
matrixMul' a b = \ (# w1) (# h1) (# h2) -> unsplit w1 h2 (outerWith f a' b')
  where
    f x y = reduce matrixAdd (zipWith matrixMul x y)
    a' = split w1 h1 a
```

```
    b' = (transpose . split h1 h2) b

-- "columns'" identification
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith f a' (columns b'))
  where
    f x y = reduce matrixAdd (zipWith matrixMul x y)
    a' = split w1 h1 a
    b' = split h1 h2 b

-- "dotProduct'" identification
matrixMul' a b = \ (# w1) (# h1) (# h2) →
    unsplit w1 h2 (outerWith dotProduct' a' (columns b'))
  where
     dotProduct' x y = reduce matrixAdd (zipWith matrixMul x y)
    a' = split w1 h1 a
    b' = split h1 h2 b
```

REFERENCES

[1] URL: http://www.top500.org/.

[2] URL: http://www.green500.org/.

[3] URL: http://www.tilera.com/.

[4] URL: http://www.kalray.eu/.

[5] Emmanuel Agullo et al. "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for". In: 2010.

[6] Thomas William Ainsworth and Timothy Mark Pinkston. "Characterizing the Cell EIB On-Chip Network". In: *IEEE Micro* 27.5 (Sept. 2007), pp. 6–14. ISSN: 0272-1732. DOI: 10.1109/MM.2007.81.

[7] Brad Alexander. *Mapping Adl to the Bird-Meertens Formalism*. Tech. rep. 1994.

[8] Brad Alexander, Dean Engelhardt, and Andrew Wendelborn. "A supercomputer implementation of a functional data parallel language". In: (1994).

[9] Brad Alexander, Dean Engelhardt, and Andrew Wendelborn. *An overview of the Adl language project*. 1997.

[10] Brad Alexander and Andrew Wendelborn. "Automated transformation of BMF programs". In: *In The First International Workshop on Object Systems and Software Architectures*. 2004, pp. 133–141.

[11] *AMD Stream Computing User Guide*. AMD. Dec. 2008.

[12] M. Amini et al. "Par4All: From convex array regions to heterogeneous computing". In: *2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)*. 2012.

[13] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, June 2009.

[14] OpenMP ARB. *OpenMP Application Programming Interface 3.1*. July 2011.

[15] Joe Armstrong et al. *Concurrent Programming in ERLANG*. 1993.

[16] Cédric Augonnet. "Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective". PhD thesis. Ph. D. dissertation, Université Bordeaux 1, 351 cours de la Libération—33405 TALENCE cedex, 2011.

[17] Cédric Augonnet et al. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”. In: *Concurrency and Computation: Practice and Experience* (2010).

[18] Eduard Ayguadé et al. “An extension of the StarSs programming model for platforms with multiple GPUs”. In: *Euro-Par 2009 Parallel Processing* (2009), pp. 851–862.

[19] John Backus. “Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs”. In: (1977).

[20] Denis Barthou et al. “QIRAL: A High Level Language for Lattice QCD Code Generation”. In: (Aug. 2012).

[21] Can Basaran and Kang Kyoung-Don. “Grex: An efficient MapReduce framework for graphics processing units”. In: *Journal of Parallel and Distributed Computing* (2013), pages. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2013.01.004.

[22] Richard Bird et al. *Lectures on constructive functional programming*. Oxford University Computing Laboratory, Programming Research Group, 1988.

[23] Guy Blelloch. *Nesl: A Nested Data-Parallel Language, 3.1*. 1995.

[24] Guy E Blelloch. “Prefix sums and their applications”. In: *Synthesis of Parallel Algorithms* (1991), pp. 35–60.

[25] Guy E Blelloch and Gary W Sabot. “Compiling Collection-Oriented Languages onto Massively Parallel Computers”. In: (1989).

[26] Robert D Blumofe et al. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[27] George Bosilca et al. “DAGuE: A generic distributed DAG engine for high performance computing”. In: *Parallel Computing* 38.1 (2011), pp. 37–51.

[28] G. Bosilca et al. “Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA”. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. 2011, pp. 1432–1441. DOI: 10.1109/IPDPS.2011.299.

[29] Michael Boyer et al. “Load balancing in a changing world: dealing with heterogeneity and performance variability”. In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2013, p. 21.

[30] S. Breitinger et al. “The Eden coordination model for distributed memory systems”. In: *High-Level Programming Models and Supportive Environments, 1997. Proceedings., Second International Workshop on*. 1997, pp. 120–124. DOI: 10.1109/HIPS.1997.582964.

[31] Andre R Brodtkorb et al. “State-of-the-art in heterogeneous computing”. In: *Scientific Programming* 18.1 (2010), pp. 1–33.

[32] François Broquedis et al. “ForestGOMP: an efficient OpenMP environment for NUMA architectures”. In: *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Müller and Eduard Ayguade* 38.5 (2010), pp. 418–439. DOI: 10.1007/s10766-010-0136-3.

[33] François Broquedis et al. “hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italie, Feb. 2010. DOI: 10.1109/PDP.2010.67.

[34] Ian Buck et al. “Brook for GPUs: stream computing on graphics hardware”. In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH ’04. Los Angeles, California: ACM, 2004, pp. 777–786. DOI: 10.1145/1186562.1015800.

[35] David Cann. “Retire Fortran? A debate rekindled”. In: *Supercomputing ’91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1991, pp. 264–272.

[36] CAPS. *OpenHMPP Concepts and Directives 2.5*. Nov. 2012.

[37] Manuel MT Chakravarty et al. “Accelerating Haskell array codes with multicore GPUs”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[38] Manuel M.T. Chakravarty et al. “Nepal - Nested Data-Parallelism in Haskell”. In: *Euro-Par ’01*. Springer-Verlag, 2001, pp. 524–534.

[39] B.L. Chamberlain et al. “The case for high-level parallel programming in ZPL”. In: *Computational Science Engineering, IEEE* 5.3 (1998), pp. 76–86. ISSN: 1070-9924. DOI: 10.1109/99.714604.

[40] Brad Chamberlain. “A Brief Overview of Chapel (revision 1.0)”. In: *To be published*. Jan. 2013.

[41] Iris Christadler and Volker Weinberg. “RapidMind: Portability across Architectures and Its Limitations”. In: *Facing the Multicore-Challenge*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Vol. 6310. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 4–15. ISBN: 978-3-642-16232-9.

[42] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046.

[43] Carsten Clauss et al. “Evaluation and improvements of programming models for the Intel SCC many-core processor”. In: *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE. 2011, pp. 525–532.

[44] Murray Cole. “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming”. In: *Parallel Computing* 30.3 (2004), pp. 389–406. ISSN: 0167-8191. DOI: 10.1016/j.parco.2003.12.002.

[45] UPC Consortium. *UPC Language Specifications, 1.2*. May 2005.

[46] Intel Corporation. *Intel® Xeon Phi Coprocessor System Software Developers Guide*. May 2013.

[47] Michel Cosnard and Michel Loi. “Automatic task graph generation techniques”. In: *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*. Vol. 2. IEEE. 1995, pp. 113–122.

[48] John Darlington et al. “Functional skeletons for parallel coordination”. In: *EURO-PAR ’95 Parallel Processing*. Ed. by Seif Haridi, Khayri Ali, and Peter Magnusson. Vol. 966. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 55–66. ISBN: 978-3-540-60247-7. DOI: 10.1007/BFb0020455.

[49] John Darlington et al. “Parallel Programming Using Skeleton Functions”. In: *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*. PARLE ’93. London, UK, UK: Springer-Verlag, 1993, pp. 146–160. ISBN: 3-540-56891-3.

[50] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Vol. 6. Dec. 2004, pp. 137–150.

[51] Peter J. Denning and Jack B. Dennis. “The resurgence of parallelism”. In: *Commun. ACM* 53.6 (June 2010), pp. 30–32. ISSN: 0001-0782. DOI: 10.1145/1743546.1743560.

[52] R. Dolbeau, S. Bihan, and F. Bodin. “HMPP: A hybrid multi-core parallel programming environment”. In: 2007.

[53] Ulrich Drepper and Ingo Molnar. “The native POSIX thread library for Linux”. In: *White Paper, Red Hat* (2003).

[54] Kayvon Fatahalian et al. “Sequoia: Programming the Memory Hierarchy”. In: *SC 2006 Conference, Proceedings of the ACM/IEEE*. Nov. 2006, p. 4. DOI: 10.1109/SC.2006.55.

[55] Jörg Fischer, Sergei Gorlatch, and Holger Bischof. “Foundations of data-parallel skeletons”. In: *Patterns and skeletons for parallel and distributed computing*. Ed. by Fethi A. Rabhi and Sergei Gorlatch. London, UK, UK: Springer-Verlag, 2003. Chap. 1, pp. 1–27. ISBN: 1-85233-506-8.

[56] Matthew Fluet et al. “Manticore: A heterogeneous parallel language”. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM. 2007, pp. 37–44. URL: http://dl.acm.org/citation.cfm?id=1248656.

[57] High Performance Fortran Forum. “High Performance Fortran Language Specification. Version 2.0”. In: Jan. 1997.

[58] MPI Forum. *MPI: A Message-Passing Interface Standard 3.0*. Sept. 2012.

[59] Stefan Frenz et al. “A Practical Comparison of Cluster Operating Systems Implementing Sequential and Transactional Consistency”. In: *Distributed and Parallel Computing*. Ed. by Michael Hobbs, AndrzejM. Goscinski, and Wanlei Zhou. Vol. 3719. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 23–33. ISBN: 978-3-540-29235-7. DOI: 10.1007/11564621_3.

[60] J.-L. Gaudiot et al. “The Sisal model of functional programming and its implementation”. In: *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*. 1997, pp. 112–123. DOI: 10.1109/AISPAS.1997.581640.

[61] Thierry Gautier et al. “XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures”. In: *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 2013. URL: http://hal.inria.fr/hal-00799904/.

[62] David Gelernter and Nicholas Carriero. “Coordination languages and their significance”. In: *Communications of the ACM* 35.2 (1992), p. 96. URL: http://dl.acm.org/citation.cfm?id=376083.

[63] Alfons Geser and Sergei Gorlatch. “Parallelizing Functional Programs by Generalization”. In: *Journal of Functional Programming*. Springer-Verlag, 1997, pp. 46–60.

[64] Anwar Ghuloum et al. “Ct: A Flexible Parallel Programming Model for Tera-scale Architectures”. In: Intel, 2007.

[65] H. González-Vélez and M. Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”. In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[66] Clemens Grelck and Sven-Bodo Scholz. "SAC: From high-level programming with arrays to efficient parallel execution". In: *Parallel processing letters* 13.3 (2003), pp. 401–412.

[67] Dominik Grewe and Michael F. P. O'Boyle. "A static task partitioning approach for heterogeneous systems using OpenCL". In: *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*. CC'11/ETAPS'11. Saarbrücken, Germany: Springer-Verlag, 2011, pp. 286–305. ISBN: 978-3-642-19860-1.

[68] Dominik Grewe, Zheng Wand, and Michael F.P. O'Boyle. "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems". In: *ACM/IEEE International Symposium on Code Generation and Optimization*. Shenzen, China, Feb. 2013.

[69] Khronos Group. *OpenCL Specification, 1.2*. Nov. 2011.

[70] Khronos Group. *SPIR 1.0 Specification for OpenCL*. Aug. 2012.

[71] OpenACC Group. *The OpenACC Application Programming Interface*. 2011.

[72] Philipp Haller and Martin Odersky. "Scala Actors: Unifying thread-based and event-based programming". In: *Theoretical Computer Science* 410.2–3 (2009). Distributed Computing Techniques, pp. 202–220. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.09.019.

[73] Kevin Hammond. "Parallel Functional Programming: An Introduction". In: *International Symposium on Parallel Symbolic Computation*. Hagenberg/Linz, Austria: World Scientific, Sept. 1994.

[74] Kevin Hammond and Greg Michelson, eds. *Research Directions in Parallel Functional Programming*. London, UK, UK: Springer-Verlag, 2000. ISBN: 1852330929.

[75] Kevin Hammond and Álvaro Rebón Portillo. "HaskSkel: Algorithmic skeletons in haskell". In: *Implementation of Functional Languages* (2000), pp. 181–198.

[76] Alexander Heinecke et al. "Towards High-Performance Implementations of a Custom HPC Kernel Using ® Array Building Blocks". In: *Facing the Multicore - Challenge II*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Vol. 7174. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 36–47. ISBN: 978-3-642-30396-8. DOI: 10.1007/978-3-642-30397-5_4.

[77] P. Hilfinger et al. *Titanium Language Reference Manual, 2.20*. Aug. 2006.

[78] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. "Direct Cache Access for High Bandwidth Network I/O". In: *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 50–59. ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.23.

[79] Andra-Ecaterina Hugo et al. "Composing multiple StarPU applications over heterogeneous machines: a supervised approach". Anglais. In: *Third International Workshop on Accelerators and Hybrid Exascale Systems*. Boston, États-Unis, May 2013. URL: http://hal.inria.fr/hal-00824514.

[80] IBM. *OpenCL Common Runtime for Linux on x86 Architecture (version 0.1)*. 2011.

[81] IBM. *SPE Runtime Management Library Version 2.2*. 2.2. 2007.

[82] Inria. *StarPU Handbook*. http://runtime.bordeaux.inria.fr/StarPU/. 2013.

[83] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. “TreeMatch : Un algorithme de placement de processus sur architectures multicœurs”. French. In: *RenPAR - 21e Rencontres Francophones du Parallélisme*. Grenoble, France, Jan. 2013.

[84] Laxmikant V. Kale and Sanjeev Krishnan. “CHARM++: a portable concurrent object oriented system based on C++”. In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. OOPSLA ’93. Washington, D.C., USA: ACM, 1993, pp. 91–108. ISBN: 0-89791-587-9. DOI: 10.1145/165854.165874.

[85] Gabriele Keller et al. “Regular, shape-polymorphic, parallel arrays in Haskell”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ICFP ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272. ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863582.

[86] P.H. Kelly. *Functional programming for loosely-coupled multiprocessors*. MIT Press, 1989.

[87] Ken Kennedy, Charles Koelbel, and Hans Zima. “The rise and fall of High Performance Fortran: an historical object lesson”. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL III. San Diego, California: ACM, 2007, pages. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238851.

[88] Jungwon Kim et al. “SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters”. In: *Proceedings of the 26th ACM international conference on Supercomputing*. ICS ’12. San Servolo Island, Venice, Italy: ACM, 2012, pp. 341–352. ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304623.

[89] J. Kim et al. “Achieving a single compute device image in OpenCL for multiple GPUs”. In: *SIGPLAN Notices* 46.8 (2011), p. 277.

[90] Monica S Lam and Martin C Rinard. “Coarse-grain parallel programming in Jade”. In: *ACM SIGPLAN Notices*. Vol. 26. 7. ACM. 1991, pp. 94–105. URL: http://dl.acm.org/citation.cfm?id=109636.

[91] Carlos S de la Lama et al. “Static Multi-device Load Balancing for OpenCL”. In: *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE. 2012, pp. 675–682.

[92] Sean Lee et al. “GPU kernels as data-parallel array computations in Haskell”. In: *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*. 2009.

[93] Calvin Lin and Lawrence Snyder. “ZPL: An array sublanguage”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Vol. 768. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 96–114. ISBN: 978-3-540-57659-4. DOI: 10.1007/3-540-57659-2_6.

[94] H.-W. Loidl et al. “Comparing Parallel Functional Languages: Programming and Performance”. In: *Higher Order Symbol. Comput.* 16.3 (Sept. 2003), pp. 203–251. ISSN: 1388-3690. DOI: 10.1023/A:1025641323400.

[95] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. “Parallel functional programming in Eden”. In: *Journal of Functional Programming* 15.3 (2005), pp. 431–476.

[96] R. Lottiaux et al. “OpenMosix, OpenSSI and Kerrighed: a comparative study”. In: *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*. Vol. 2. May 2005, 1016–1023 Vol. 2. DOI: 10.1109/CCGRID.2005.1558672.

[97] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 45–55. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669121.

[98] Ewing Lusk and Katherine Yelick. "Languages for high-productivity computing: the DARPA HPCS language project". In: *Parallel Processing Letters* 17.01 (2007), pp. 89–102.

[99] LuxRender. *GPL physically based renderer*. 2013. URL: http://www.luxrender.net.

[100] Geoffrey Mainland and Greg Morrisett. "Nikola: embedding compiled GPU functions in Haskell". In: *Proceedings of the third ACM Haskell symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 67–78. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863533.

[101] William R. Mark et al. "Cg: a system for programming graphics hardware in a C-like language". In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 896–907. ISSN: 0730-0301. DOI: 10.1145/882262.882362.

[102] Michael D McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Wellesley, 2004.

[103] Diego Melpignano et al. "Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: ACM, 2012, pp. 1137–1142. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228568.

[104] Greg Michaelson et al. "Nested algorithmic skeletons from higher order functions". In: *PARALLEL ALGORITHMS AND APPLICATION* 16.3 (2001), pp. 181–206.

[105] Inc. Multicoreware. *GMAC: Global Memory for Accelerator, TM: Task Manager*. 2011. URL: http://www.multicorewareinc.com.

[106] M. Nakao et al. "Productivity and Performance of Global-View Programming with XcalableMP PGAS Language". In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. 2012, pp. 402–409. DOI: 10.1109/CCGrid.2012.118.

[107] Naohito Nakasato et al. "Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems". In: *CoRR* abs/1206.1199 (2012).

[108] C.J. Newburn et al. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. Apr. 2011, pp. 224–235. DOI: 10.1109/CGO.2011.5764690.

[109] Robert W. Numrich and John Reid. "Co-array Fortran for parallel programming". In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. ISSN: 1061-7264. DOI: 10.1145/289918.289920.

[110] NVIDIA. *CUDA C Programming Guide*. 5.0. Oct. 2012.

[111] S. L. Peyton Jones. "Parallel Implementations of Functional Programming Languages". In: *Comput. J.* 32.2 (Apr. 1989), pp. 175–186. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.175.

[112] Simon L Peyton Jones and David R Lester. *Implementing functional languages: a tutorial*. Prentice-Hall, Inc., 1992.

[113] Judit Planas et al. "Hierarchical task-based programming with StarSs". In: *International Journal of High Performance Computing Applications* 23.3 (2009), pp. 284–299.

[114] R. Plasmeijer, M. Van Eekelen, and MJ Plasmeijer. *Functional programming and parallel graph rewriting*. Vol. 857. Addison-wesley, 1993.

[115] Martin C. Rinard and Monica S. Lam. "The design, implementation, and evaluation of Jade". In: *ACM Trans. Program. Lang. Syst.* 20.3 (May 1998), pp. 483–545. ISSN: 0164-0925. DOI: 10.1145/291889.291893.

[116] M.C. Rinard, D.J. Scales, and M.S. Lam. "Heterogeneous parallel programming in Jade". In: *Supercomputing'92. Proceedings*. IEEE. 1992, pp. 245–256.

[117] Paul Roe and Andrew Wendelborn. *Implicit Array Copying: Prevention is Better than Cure*. 1992.

[118] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN: 0-262-22069-5.

[119] Vijay Saraswat et al. "Report on the Programming Language X10. Version 2.3". In: Feb. 2013.

[120] Wolfgang Schreiner. "Parallel Functional Programming - an Annotated Bibliography". In: (1993).

[121] J. T. Schwartz et al. *Programming with sets; an introduction to SETL*. New York, NY, USA: Springer-Verlag New York, Inc., 1986. ISBN: 0-387-96399-5.

[122] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann Pub, 2009.

[123] David B Skillicorn. *Foundations of parallel programming*. Vol. 6. Cambridge University Press, 2005.

[124] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. "Maestro: data orchestration and tuning for OpenCL devices". In: *Euro-Par 2010-Parallel Processing*. Springer, 2010, pp. 275–286.

[125] Inc. Sun Microsystems. "The Fortress Language Specification. Version 1.0". In: Sun, Mar. 2007.

[126] MIT Supercomputing Technologies Group. *Cilk Manual 5.4.6*.

[127] Bo Joel Svensson and Ryan Newton. "Programming Future Parallel Architectures with Haskell and Intel ArBB". In: (2011).

[128] Bo Joel Svensson and Mary Sheeran. "Parallel programming in Haskell almost for free: an embedding of intel's array building blocks". In: *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*. FHPC '12. Copenhagen, Denmark: ACM, 2012, pp. 3–14. ISBN: 978-1-4503-1577-7. DOI: 10.1145/2364474.2364477.

[129] Joel Svensson, Mary Sheeran, and Koen Claessen. "Obsidian: A domain specific embedded language for parallel programming of graphics processors". In: *Implementation and Application of Functional Languages* (2011), pp. 156–173.

[130] David Tarditi, Sidd Puri, and Jose Oglesby. "Accelerator: using data parallelism to program GPUs for general-purpose uses". In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. San Jose, California, USA: ACM, 2006, pp. 325–335. ISBN: 1-59593-451-0. DOI: http://doi.acm.org/10.1145/1168857.1168898.

[131] H. Topcuoglu, S. Hariri, and Min-You Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *Parallel and Distributed Systems, IEEE Transactions on* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219. DOI: 10.1109/71.993206.

[132] P. W. Trinder et al. "Algorithm + strategy = parallelism". In: *J. Funct. Program.* 8.1 (Jan. 1998), pp. 23–60. ISSN: 0956-7968. DOI: 10.1017/S0956796897002967.

[133] K. Vaidyanathan and D.K. Panda. "Benefits of I/O Acceleration Technology (I/OAT) in Clusters". In: *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*. Apr. 2007, pp. 220–229. DOI: 10.1109/ISPASS.2007.363752.

[134] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[135] Joseph Windows. *Automated Parallelisation of code written in the Bird-Meertens Formalism*. 2003.

[136] Michael Wolfe. *Implementing the PGI accelerator model*. In GPGPU. 2010.

[137] Ying Zhang et al. "Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications". In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. 2011, pp. 205–215. DOI: 10.1109/IISWC.2011.6114180.

[138] Gengbin Zheng et al. "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers". In: *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*. San Diego, California, USA, Sept. 2010.