Retour d'expérience : portage d'une application haute-performance vers un langage de haut niveau

Mathias Bourgoin, Emmanuel Chailloux, Jean-Luc Lamotte

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Retour d'expérience : portage d'une application haute-performance vers un langage de haut niveau

Mathias Bourgoin & Emmanuel Chailloux & Jean Luc Lamotte

Laboratoire d'Informatique de Paris 6 (LIP6 - UMR 7606)
Université Pierre et Marie Curie Sorbonne Universités (UPMC - Paris 6)
4 place Jussieu, 75005 Paris, France
email : {Mathias.Bourgoin,Emmanuel.Chailloux,Jean-Luc.Lamotte}@lip6.fr

---

Résumé
La programmation généraliste des processeurs graphiques implique d'associer des unités de calculs graphiques hyperparallèles avec des CPU classiques dans le but d'accroître les performances d'applications communément traitées par ces derniers. Ces systèmes hybrides sont particulièrement complexes à programmer, en particulier pour en tirer de hautes performances. Afin d'en simplifier l'utilisation, des bibliothèques de haut niveau ont été développées. Cet article décrit l'utilisation d'une de ces bibliothèques pour effectuer le portage d'une application de calcul physique, `PROP`. A travers cet exemple, nous décrivons les outils utilisés, en particulier le langage OCaml et la bibliothèque SPOC, ainsi que le portage lui même. Nous étudions également les performances obtenues en les comparant à celle de la version d'origine écrite en Fortran qui exploite les GPU à l'aide de l'environnement Cuda. Les résultats montrent une meilleure fiabilité du programme grâce à OCaml associé à une réduction importante de la taille du code, pour des performances équivalente au programme d'origine.

Mots-clés : HPC, GPU, portage application Fortran, programmation haut niveau, OCaml

---

1. Introduction

Les processeurs graphiques (GPU) ont considérablement évolué ces dernières années. D'outils dédiés à la visualisation, ils sont désormais devenus des accélérateurs de calcul programmables. Dans ce cadre, des environnements de développement sont proposés afin de permettre l'écriture de programmes de plus en plus généralistes. Actuellement, les deux principaux environnements sont Cuda et OpenCL. Ils sont fournis avec des bibliothèques de bas niveau pour les langages C et C++. Pour aider les programmeurs, des bibliothèques de plus haut niveau ont vu le jour. Nous avons développé une bibliothèque pour la programmation GPGPU (Gerneral Purpose GPU) dédiée au langage de programmation OCaml dans le cadre du projet OpenGPU. Afin de vérifier sur un cas réaliste les bénéfices qu'elle apporte en abstraction et en maintenabilité, nous avons ici expérimenté son utilisation dans le cadre du portage d'une application de calcul physique exploitant déjà les GPU via l'environnement Cuda. Cette expérience nous a permis d'évaluer l'utilisation d'outils de haut niveau pour les GPU, du point de vue de l'écriture d'un programme haute performance réaliste. Ainsi, nous avons observé les gains en abstractions et en sûreté offerts au programmeur tout en vérifiant la conservation de l'efficacité. Ce document est organisé comme suit. La section 2 présente le programme étudié et son implantation. La section 3 introduit le langage OCaml et la bibliothèque utilisée pour le calcul GPU. Le portage et la description des bénéfices eux-mêmes liés au passage à un langage de haut niveau sont décrits section 4. Enfin, les résultats obtenus en matière de performance sont exposés en section 5 avant de conclure sur cette expérience et de présenter les travaux futurs.

2. Présentation du programme étudié

2.1. PROP

PROP est un programme de la suite 2DRMP [1] qui modélise la diffusion des électrons dans des atomes hydrogénoïdes et des ions à des énergies intermédiaires. L'objectif principal de PROP est de propager une $\mathcal{R}$-Matrice [2], $\mathfrak{R}$, dans un espace de configuration à deux électrons. La propagation est réalisée pour chaque énergie de diffusion. Celles-ci sont indépendantes les unes des autres. Chaque propagation est calculée par des équations dont les éléments principaux sont des multiplications de matrices, impliquant des sous-matrices de $\mathfrak{R}$. Au cours du programme, $\mathfrak{R}$ est modifiée : en particulier, sa taille augmente au cours de la propagation. PROP se situe au cœur de la suite 2DRMP il prend en entrée des fichiers produits par un autre programme de la suite et produit d'autres fichiers exploitables par le programme suivant. Ceci implique des étapes de lecture/écriture et de préparation des données en plus de l'étape de calcul.

2.2. Parallélisation et exploitation GPGPU

La suite 2DRMP fonctionne sur des machines séquentielles, sur des clusters hautes performances ou encore sur des super calculateurs. 2DRMP a fait l'objet d'optimisations pour les architectures parallèles à mémoire partagée, mais aussi à mémoire distribuée. PROP a, lui, par la suite, été modifié à plusieurs reprises pour profiter des dispositifs GPGPU (General Purpose GPU). La section suivante présente la programmation GPGPU puis l'implantation GPGPU du programme PROP.

2.2.1. Programmation GPGPU

La programmation GPGPU consiste à gérer des dispositifs spécifiques (dispositifs GPGPU) pour traiter des calculs généraux. Deux environnements de développement sont principalement utilisés : Cuda [3] et OpenCL [4]. Tous deux offrent des langages (extensions C/C++) pour écrire des programmes (appelés noyaux) qui s'exécutent sur les GPGPU. Ils disposent également d'API de bas niveau pour gérer noyaux et transferts de données entre CPU et GPGPU.

Deux environnements de développement. Cuda et OpenCL sont très similaires, mais incompatibles. Les données transférées *via* OpenCL ne peuvent pas être utilisées par les noyaux Cuda, et *vice versa*. De plus, certains périphériques sont incompatibles avec l'un des deux. Cuda est propriétaire, seuls les périphériques NVIDIA peuvent l'utiliser directement (certains outils tiers (comme CU2CL [5]) peuvent traduire automatiquement du code Cuda vers de l'OpenCL). Cependant, de nombreux outils et bibliothèques ont été développés pour Cuda, et peuvent profondément améliorer la productivité. La plupart des constructeurs offrent, actuellement, une implantation d'OpenCL, pour leurs GPGPU, ce qui en fait un meilleur choix pour la portabilité bien que développer des programmes efficaces pour des architectures GPGPU multiples implique des optimisations spécifiques pour chaque architecture. Avec deux environnements de développement incompatibles, atteindre de hautes performances tout en maintenant une grande portabilité est très difficile.

Noyaux. Pour écrire les noyaux, les deux environnements proposent des extensions C/C++. Toutefois, les GPGPU sont des architectures hautement parallèles qui exigent des modèles de programmation spécifiques. Cuda et OpenCL exploitent le *Stream Processing* (Traitement de flux) qui définit les GPGPU comme des multiprocesseurs, chacun constitué de nombreuses unités de calcul. Il simplifie la description du parallélisme en limitant les calculs à l'application d'un même noyau à chaque élément d'un ensemble de données (le "flux"). Chaque unité de calcul matérielle va exécuter le noyau sur un élément du flux de données en parallèle. Cuda et OpenCL exigent de décrire le GPGPU sous la forme d'une grille, contenant des blocs d'unités de calcul, qui exécuteront le noyau. Au lancement du noyau, ils associent cette description aux multiprocesseurs physiques du GPGPU, optimisant l'association des blocs afin d'améliorer l'occupation des multiprocesseurs et de maximiser le parallélisme.

Transferts. Combiner des architectures hétérogènes (CPU et GPGPU) implique des transferts de données, de la mémoire CPU vers celle des GPGPU. Ces dispositifs sont communément considérés comme des appareils invités, avec une mémoire dédiée, liés au CPU hôte à travers une interface PCI-E. Cette interface dispose d'une bande passante limitée en comparaison de celle de la mémoire des GPGPU. Ceci implique des transferts lents, qu'il devient aussi important d'optimiser que les noyaux de calcul, pour obtenir des hautes performances.

2.2.2. Implantation

PROP est un programme écrit en Fortran utilisant les bibliothèques BLAS et LAPACK pour les calculs de propagation de la $\mathcal{R}$-Matrice. *CAPS-Entreprise* a apporté une première modification à PROP dans le cadre d'un appel d'offre *GENCI* visant au portage d'applications. PROP avait été retenu pour un portage avec le compilateur HMPP. Lors de cette étude, l'équation de propagation a été remaniée pour manipuler des matrices plus grandes. En effet, pour des produits de matrices, les performances GPGPU s'accroissent avec la taille des matrices, diminuant la surcharge induite par le transfert des données. En utilisant le compilateur HMPP, CAPS a développé une version de PROP dans laquelle les produits de matrices s'effectuent sur GPGPU (via Cuda). Une seconde modification a eu lieu pour limiter les transferts entre CPU et GPU en localisant l'ensemble des calculs sur les dispositifs GPGPU [6]. Ce travail a permis de développer une version du programme PROP effectuant l'ensemble des calculs de propagation sur GPGPU. PROP effectue une série de calculs sur différentes sections de la $\mathcal{R}$-matrice d'entrée. Les données nécessaires au calcul sont lues dans des fichiers lors d'une phase de préparation des données. La préparation a lieu pour chaque section à évaluer. L'implantation étudiée utilise un système de *double buffering* pour permettre, pour chaque section, au calcul sur GPGPU de recouvrir les temps de lecture des données et de préparation du calcul de la section suivante. Elle utilise les bibliothèques Cublas et Magma pour la réalisation des calculs. L'implantation GPGPU de PROP utilise une couche de code C pour faire la liaison entre le code Fortran et la bibliothèque Cuda.

3. OCaml et la bibliothèque SPOC

3.1. OCaml

OCaml [7] est un langage de programmation généraliste, de haut niveau, développé par Inria. C'est un langage multiparadigmes, à la fois fonctionnel, impératif, modulaire et orienté objet. Il est fortement et statiquement typé. Il offre une gestion automatique de la mémoire associée à un glaneur de cellules (*Garbage Collector* - GC) incrémental performant. La distribution d'OCaml d'Inria propose deux types de compilateurs. L'un qui compile vers du code-octet portable (des machines virtuelles OCaml existent pour de nombreuses architectures) tandis que l'autre produit du code natif efficace. OCaml est par ailleurs complètement interopérable avec le langage C.
SPOC [8] est une bibliothèque pour OCaml gérant les dispositifs GPGPU et leur mémoire, et permettant aux développeurs d'effectuer des calculs GPGPU. Elle bénéficie particulièrement de plusieurs aspects d'OCaml. D'abord, étant une bibliothèque liant OCaml avec les bibliothèques C des environnements Cuda et OpenCL, elle utilise l'interopérabilité d'OCaml avec C. SPOC cible la programmation GPGPU ainsi que, la haute performance, ce qui rend nécessaire d'utiliser un langage de programmation déjà efficace pour des calculs séquentiels. Les compilateurs natifs d'OCaml offrent des hautes performances pour ces calculs. De plus, avec SPOC, nous souhaitons simplifier la programmation GPGPU tout en améliorant la fiabilité des logiciels et la productivité. La sûreté de typage améliore la fiabilité en détectant de nombreuses erreurs de programmation lors de la compilation, tandis que le gestionnaire mémoire assure la cohérence des données (pas de fuite mémoire ou de pointeurs fantômes) et limite l'occupation mémoire au cours de l'exécution. Par ailleurs, SPOC bénéficie directement des multiples paradigmes d'OCaml, combinant programmation objet pour l'exploitation des noyaux externes avec programmation séquentielle, fonctionnelle et modulaire pour la bibliothèque d'exécution. Enfin, SPOC exploite les outils d'extensions d'OCaml (ici Camlp4) pour simplifier la déclaration de noyaux externes, ces outils pourront à l'avenir permettre de développer un langage, dédié à la programmation des noyaux de calculs, intégré à OCaml.

3.2. SPOC

SPOC (`http://algo-prog.info/spoc`) consiste en une extension à OCaml associée à une bibliothèque d'exécution. L'extension permet la déclaration de noyaux GPGPU externes utilisables depuis un programme OCaml, tandis que la bibliothèque permet de manipuler ces noyaux ainsi que les données nécessaires à leur exécution. SPOC offre, de plus, une abstraction supplémentaire en unifiant les deux environnements de développement GPGPU en une même bibliothèque. Enfin, SPOC abstrait les transferts mémoires via l'utilisation de jeux de données spécifiques.

Environnements de développement. Comme différents environnements existent pour la programma-

tion GPGPU, il peut être difficile d'obtenir portabilité et efficacité. SPOC détecte, lors de son initialisation, l'ensemble des GPGPU compatibles avec Cuda ou OpenCL et les rend exploitables par le programme OCaml. Elle unifie aussi les deux environnements en s'adaptant lors de l'exécution en fonction du matériel utilisé et de l'environnement (Cuda ou OpenCL) correspondant. Ceci permet l'exécution d'un même programme sur différentes architectures, mais aussi d'utiliser de multiples GPGPU (y compris avec les deux environnements incompatibles) conjointement.

Noyaux. SPOC permet l'utilisation de noyaux externes écrits en assembleur Cuda ou en OpenCL C99. Elle offre une extension à OCaml pour déclarer ces noyaux, d'une manière similaire à celle qu'OCaml propose pour déclarer des fonctions C externes. Cette extension permet la vérification statique des types des arguments du noyau réduisant les risques d'erreurs difficiles à déboguer lors de l'exécution. La figure 1 présente un exemple simple de noyau OpenCL et comment l'utiliser avec SPOC. Ce noyau *vec_mul*, prend trois vecteurs (*__global float* *) stockés dans la mémoire globale du GPGPU en paramètre ainsi qu'un entier représentant la taille des vecteurs à additionner. Chaque unité de calcul (dont l'identifiant est obtenu *via get_global_id(0)*) va calculer la multiplication d'un seul élément de chaque vecteur. Le code OCaml associé permet l'utilisation d'un tel noyau dans un programme OCaml. Il consiste en la déclaration du nom du noyau suivi de son type et de deux chaînes de caractères. Les deux chaînes permettent d'identifier le fichier externe contenant le noyau ainsi que la fonction dans ce fichier correspondant au noyau . Ici, nous pouvons voir que les types des paramètres doivent être traduits pour être compatibles avec SPOC et OCaml de *__global float** vers *Spoc.Vector.float32*.

Transferts. Comme présenté en section 2.2.1, la programmation GPGPU implique des transferts mémoire entre CPU hôte et GPGPU invités. Cuda et OpenCL demandent des transferts explicites à travers des API de bas niveau. De plus, pour obtenir de hautes performances pour des programmes complexes, il est important d'optimiser l'ordonnancement de ces transferts. SPOC utilise le gestionnaire mémoire d'OCaml pour réduire le nombre de transferts, ne transférant les données que lorsque c'est nécessaire. Pour cela, SPOC introduit un type de jeux de données appelé *vectors*. A l'exécution, lors d'un accès à un *vector* (par le CPU ou lors du lancement d'un noyau), SPOC vérifie la position courante du *vector* et le déplace si nécessaire. Les *vectors* sont gérés par le gestionnaire mémoire d'OCaml qui les alloue/désalloue automatiquement, que ce soit dans la mémoire du CPU hôte ou des GPGPU invités. Ceci libère le programmeur de la gestion des transferts mémoire qui est une part importante, complexe et source de bogues de la programmation GPGPU de bas niveau, et permet de concentrer les efforts sur les calculs.

4. Portage

Afin de valider la démarche utilisée lors du développement de la bibliothèque SPOC et de vérifier le niveau de performance qu'elle permet d'obtenir, nous proposons un portage du programme `PROP` avec cette bibliothèque. Ce portage nous permet également d'évaluer les bénéfices liés à l'utilisation d'un langage de haut niveau pour la programmation GPGPU haute performance.

4.1. Démarche

Le programme `PROP` porté se divise en 3 grandes parties : la première, qui lit les données d'entrée dans des fichiers, prépare les tâches à accomplir et les lance, la seconde qui effectue le traitement sur GPGPU et la troisième qui correspond effectivement aux noyaux de calculs Cuda utilisés. `PROP` utilise à la fois des noyaux de calcul simples écrits en Cuda pour l'initialisation ou la copie de données, mais aussi des fonctions issues des bibliothèques Cublas [9] et Magma [10] afin d'effectuer les calculs. Nous avons concentré notre travail sur le portage des calculs (et non des lectures/écritures) depuis les langages For-

Code OpenCL	Code OCaml

`__kernel void vec_mul(__global const float * a,`
`__global const float * b,`
`__global float * c, int vector_size){`
`int nIndex = get_global_id(0);`
`if (nIndex < N) c[nIndex] = a[nIndex] * b[nIndex];}`
	`kernel vector_mul :`
	`Spoc.Vector.vfloat32 ->`
	`Spoc.Vector.vfloat32 ->`
	`Spoc.Vector.vfloat32 ->`
	`int -> unit = "kernel_file" "vec_mul"`

FIGURE 1 – Déclaration de noyau avec SPOC

tran/C et Cuda vers OCaml et Cuda en utilisant la bibliothèque SPOC (le portage complet sera effectué à moyen terme). Pour réaliser ce travail, nous avons d'abord effectué un *binding* des bibliothèques Cublas (V1) et Magma vers OCaml en s'appuyant sur SPOC pour la gestion des structures de données et des transferts entre CPU et GPGPU. Le programme une fois porté comprend alors une part de Fortran (initialisation et entrées/sorties), une part de C (glue entre Fortran et OCaml), une part d'OCaml (composition des calculs sur GPGPU avec SPOC) et une part de Cuda (noyaux de calculs).

4.2. Bénéfices du passage à un langage de haut niveau

L'utilisation d'un langage de haut niveau permet d'accroître la productivité et de faciliter le développement en libérant le programmeur de la verbosité des langages de bas niveau. L'utilisation de la bibliothèque SPOC a une incidence directe sur la façon d'écrire le programme et en particulier de traiter les données à transférer sur les GPGPU. Les allocations/libération et transferts de mémoire étant gérés automatiquement, une importante charge de travail est déportée du programme final vers la bibliothèque. En pratique, cela permet de n'avoir qu'à créer des *vectors* et à les utiliser soit pour du calcul sur CPU soit avec des noyaux GPGPU. `PROP` nécessite du calcul double précision pour offrir des résultats satisfaisants, là aussi l'utilisation d'OCaml et de sa sureté de typage nous assure de ne pas insérer des calculs en simple précision au sein du programme. Le portage réduit la taille du code de 31% des 3000+ lignes de code (hors commentaires) du calcul GPGPU, principalement par la suppression du code lié aux transferts. `PROP` ayant déjà fait l'objet d'importants efforts d'optimisations et s'appuyant principalement sur des bibliothèques de calcul optimisées, nous n'avons pas particulièrement profité d'OCaml comme langage de composition des calculs afin de simplifier l'expression des algorithmes employés. Dans le cas de l'écriture de nouveaux programmes, les fonctions de haut niveau et l'aspect multiparadigme d'OCaml permettront, néanmoins l'expression simplifiée d'algorithmes complexes.

5. Tests de performance

Nous avons comparé notre implantation utilisant OCaml et SPOC avec des implantations précédentes de `PROP`. Pour cela nous avons utilisé un système Linux (3.5.4-1) Fedora 17 64-bits avec un processeur Intel(R) Core(TM) i7-3770 CPU à 3.40GHz (Quad-Core + Hyperthreading), avec 8Go de mémoire DDR3, associé à un GPU Nvidia Tesla C2070 (driver 304.37 et Cuda_5.0.24) avec 6Go de mémoire, en exploitant les bibliothèques Intel MKL 2013.0.079 et Magma 1.2.1 et les compilateurs gcc-4.5.3 et OCaml-4.00.0.

Jeu de données	Taille de la $\mathcal{R}$-matrice locale	Taille de la $\mathcal{R}$-Matrice globale finale	Nombre d'énergies de diffusion
Petit	90x90	360x360	6
Moyen	90x90	360x360	64
Grand	383x383	7660x7660	6

FIGURE 2 – Caractéristiques des jeux de données

Jeu de tests	Temps d'exécution		
	Fortran	OCaml	
Petit	1,49s	3,36s	+125%
Moyen	10,70s	26,58s	+148%

(a) Cas *Petit* et *Moyen*

Version	Temps d'exécution	Accéleration / Fortran		
		CPU 1	CPU 4	GPU
CPU 1 cœur	71m11s	1	0,51	0,22
GPU Fortran	15m51s	4,49	2,29	1
GPU OCaml	**19m55s**	3,57	1,82	0,80

(b) Cas *Grand*

FIGURE 3 – Performances

Afin de vérifier les performances obtenues avec SPOC et OCaml, nous avons comparé notre implantation GPGPU avec l'implantation précédente en Fortran (sur CPU et GPGPU avec Cuda) sur différents jeux de données (décrits fig. 2).

5.1. Résultats

La figure 3 présente les temps obtenus pour les différents cas étudiés. Afin de mesurer l'efficacité du gestionnaire mémoire d'OCaml et de son utilisation par la bibliothèque SPOC, nous avons aussi étudié

pour le cas *Grand* l'occupation mémoire à la fois sur GPGPU et CPU au cours de l'exécution du programme. La figure 4 présente, à gauche, les résultats obtenus avec le programme d'origine et, à droite, ceux du programme porté. On peut y voir que l'occupation mémoire GPU varie entre 1Go et 1,8Go sur le programme d'origine et entre 1Go et 2,2Go avec le programme OCaml. La consommation mémoire CPU est stable en avec le programme d'origine et augmente au cours de l'exécution avec le programme OCaml pour atteindre 1Go.

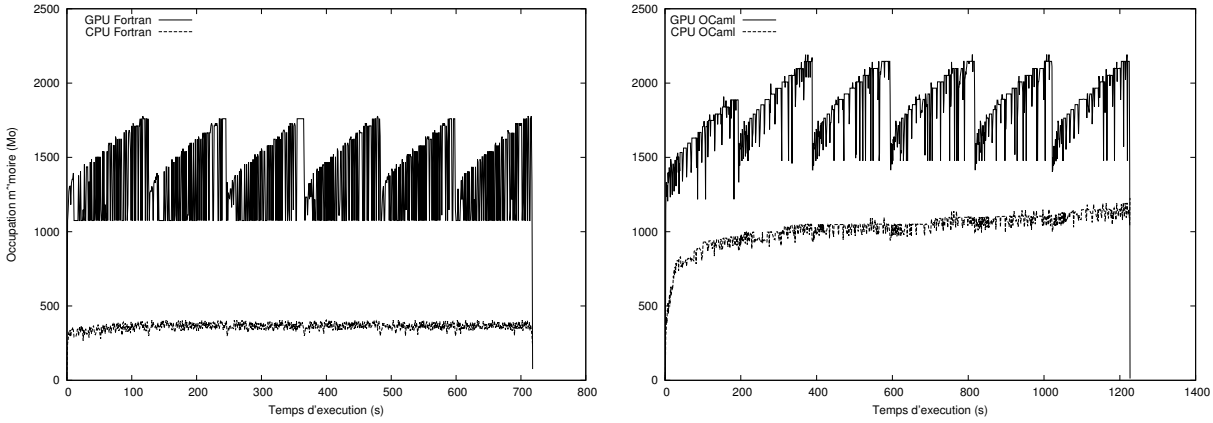

FIGURE 4 – Occupation mémoire cas *Grand*

Temps d'exécution. Lors du développement de la version d'origine du programme, l'utilisation des GPGPU permettait d'obtenir une accélération de $\times 15$ comparé à la version n'utilisant qu'un cœur du CPU. Elle est ici de $\times 4.49$ car le CPU utilisé a évolué entre les tests (passant d'un Intel Q8200 à un Core i7-3770) alors que le GPGPU est resté de même génération. On peut supposer qu'en exploitant des GPGPU de la récente génération Kepler, les accélérations seront plus importantes. Notre implantation avec OCaml offre pour les cas *Petit* et *Moyen* des performances plus faibles que pour le cas *Grand* en comparaison avec la version d'origine. De plus, ces performances diminuent en passant du cas *Petit* au cas *Moyen*. Ceci s'explique principalement par le fait qu'entre ces deux cas, seul le nombre d'énergies varie. Pour chaque énergie, les mêmes calculs seront réalisés. Ceci indique que le coût du lancement des noyaux avec SPOC peut influer pour des quantités de calcul GPGPU faibles. Ceci est confirmé avec le cas *Grand* qui avec moins d'énergies et d'importantes quantités de calcul voit notre implantation atteindre 80% des performances du programme d'origine. Cette expérience montre que l'utilisation d'un langage de haut niveau, qui simplifie l'exploitation des GPGPU et l'expression du programme, permet d'obtenir des performances proches de celles obtenues après de nombreux efforts d'optimisation avec un langage de programmation de bas niveau.

Occupation mémoire. On constate sur la figure 4 que le GC d'OCaml libère efficacement la mémoire sur le GPGPU, l'amplitude d'occupation mémoire étant similaire pour les deux versions du programme. Cependant, la mémoire CPU occupée est plus importante avec la version OCaml du programme qu'avec la version Fortran. En effet, le programme originel utilise de nombreuses matrices temporaires pour les calculs. Celles-ci ne sont pas utilisées pour le résultat final, et peuvent donc, être allouées uniquement sur le GPU où elles resteront le temps du calcul avant d'être désallouées. SPOC ne permet que d'exprimer des *vectors*, c'est-à-dire des ensembles de données transférables. Ces *vectors* sont, par défaut, alloués en mémoire CPU. Une option (à préciser lors de la création d'un *vector*) permet de l'allouer directement en mémoire GPGPU (et donc d'éviter un transfert couteux et inutile dans le cas de vecteurs utilisés uniquement sur GPGPU). Cependant, SPOC réserve un espace en mémoire CPU pour tout vecteur créé afin de permettre son rapatriement automatique, lors d'un accès par le CPU à ce vecteur, ou lors d'une libération de la mémoire GPGPU par le GC d'OCaml. Ceci implique que pour chaque matrice temporaire uniquement allouée en mémoire GPGPU dans le programme d'origine, nous avons, avec SPOC, une

allocation de mémoire supplémentaire côté CPU. Celle-ci pourra disparaitre via des abstractions permettant la composition de calculs, optimisant les transferts et placement de données automatiquement.

6. Travaux Connexes

Autour de C/C++ et Fortran. Les outils fournis avec Cuda et OpenCL concernent majoritairement les langages C/C++ et Fortran. En particulier, de nombreuses bibliothèques de calcul (comme CUBLAS ou Magma) sont fournies pour simplifier l'expression de programmes numériques complexes ou le portage de programme existants s'appuyant sur des bibliothèques similaires dédiées aux CPU. Cependant, afin d'apporter davantage d'abstraction d'autres bibliothèques ont été développées. C'est le cas par exemple de StarPU[11] qui permet à la manière de SPOC d'affranchir le développeur de la gestion explicite des transferts mémoire en y ajoutant des outils de description de graphes de dépendances entre tâches et données associé à un ordonnanceur pour optimiser la répartition des tâches sur les architectures hétérogènes. D'autres bibliothèques, comme SkePU[12] ou SkelCL[13], offrent un ensemble de squelettes de parallélisme optimisés pour l'exploitation des GPGPU, simplifiant la production de programme GPGPU complexes et efficace.

Autour des langages fonctionnels. Accelerate[14] et Repa[15] permettent d'exploiter les GPGPU avec Haskell. Ces bibliothèques s'appuie sur des opérateurs sur les tableaux optimisés pour les GPU. Elles permettent ainsi d'obtenir de bons niveaux de performances mais limitent l'expression du code exécutable sur GPGPU à ces seuls opérateurs. ScalaCL[16] agit de même, transformant les opérations *map* et *reduce* appliquées à des tableaux dans le langage Scala en noyaux OpenCL. ScalaCL évalue par ailleurs le coût des kernels pour décider, dynamiquement de l'exécution d'un calcul sur CPU ou GPU.

7. Conclusion et perspectives

7.1. Conclusion

La programmation GPGPU est complexe : tirer toutes les performances de ces architectures demande d'optimiser les calculs parallèles à réaliser, mais aussi les transferts entre CPU et GPGPU. L'utilisation d'un langage de haut niveau permet de libérer le programmeur de la complexité des API de bas niveau tout en apportant des outils pour simplifier et vérifier la programmation. Notre expérimentation, qui visait à effectuer le portage d'une application de calcul scientifique, optimisée pour les GPGPU, depuis Fortran, vers OCaml, nous a permis de vérifier les bénéfices obtenus pour le programmeur et la conservation d'un haut niveau de performances. L'utilisation de la bibliothèque SPOC pour OCaml décharge le programme de la gestion des transferts mémoire et nous avons pu vérifier que l'occupation mémoire de la version OCaml restait comparable (sur GPGPU) avec celle de la version d'origine. Cette expérience nous a par ailleurs permis de mesurer les performances de la bibliothèque SPOC pour l'écriture de programmes scientifiques réalistes. Elle permet d'atteindre un haut niveau de performance comparable à l'écriture d'un programme Fortran très optimisé. Dans cet exemple elle permet de réduire le code de 31% pour des performances de 80% de celles du code d'origine. Afin d'améliorer l'efficacité, dans l'objectif d'atteindre le niveau du code d'origine, nous avons pu mettre en évidence des optimisations supplémentaires pour la bibliothèque SPOC qui feront l'objet de travaux futurs.

7.2. Perspectives

Autour de `PROP`**.** Une dernière version de `PROP` en Fortran exploitait la capacité des GPGPU récents à exécuter plusieurs noyaux en parallèle pour réaliser le calcul de plusieurs énergies en parallèle sur une seule carte afin d'améliorer les performances. On pourra profiter des 6Go de mémoire de la C2070 pour réaliser jusqu'à trois calculs parallèles pour le jeu de données *Grand*. Pour aller plus loin, nous pourrons profiter de la capacité de SPOC à manipuler de multiples GPGPU en parallèle pour offrir une version multi-GPU de `PROP` en OCaml. Les calculs pour chaque énergie étant totalement indépendants, chaque GPGPU pourra traiter une énergie. Pour aller plus loin, la bibliothèque MagmaCL et sa sous bibliothèque MagmaBLAS offrent l'ensemble des fonctions utilisées par notre implantation de `PROP` en ciblant OpenCL au lieu de Cuda. SPOC étant portable et permettant de manipuler uniformément les dispositifs compatibles avec OpenCL ou Cuda, une extension à notre *binding* de Magma pour exploiter MagmaCL nous permettra d'avoir une version portable Cuda/OpenCL de `PROP` en OCaml. Associée à la version multi-GPU, nous aurons ainsi une version capable de fonctionner sur des machines très

hétérogènes capables d'exploiter les GPUs mais aussi les multiples cœurs des CPU récents via OpenCL. Nous souhaitons, de plus, étudier les capacités d'OCaml à nous aider à exprimer une distribution du calcul sur plusieurs machines parallèles. Ceci dans l'objectif d'avoir une version portable, hybride avec plusieurs niveaux de parallélisme de `PROP`.

Autour de SPOC. La bibliothèque SPOC offre de bonnes performances. Cette première expérience avec un programme réaliste nous permet de proposer de nouvelles optimisations, en particulier, du point de vue de l'occupation mémoire CPU (comme vu section 5) et du chargement des noyaux de calculs. De ce côté, SPOC doit actuellement, lors du premier lancement d'un noyau, lire le fichier contenant le noyau et le compiler avant de l'exécuter (le noyau compilé est ensuite stocké en mémoire pour de futures utilisations). Une amélioration visera à utiliser des noyaux précompilés liés lors de la phase de compilation avec le programme OCaml. Par ailleurs, afin d'offrir plus d'unicité et de permettre de nouvelles optimisations automatisées, nous souhaitons étudier le développement d'un langage de programmation intégré à OCaml dédié à la programmation GPGPU. Ceci nous permettra d'exprimer l'ensemble du programme dans un seul langage tout en simplifiant la liaison entre les données CPU et GPGPU.

Remerciements

Les auteurs remercient Stan Scott et le département "High Performance and Distributed Computing" de Queen's University of Belfast (Royaume-Uni) pour la bibliothèque 2DRMP et le programme `PROP` ainsi que Rachid Habel de Télécom SudParis pour le partage de ses connaissances sur `PROP`.
Ces travaux font partie du projet OpenGPU et sont partiellement financé par le pôle SYSTEMATIC PARIS REGION SYSTEMS & ICT CLUSTER (`http://opengpu.net/`).

Bibliographie

1. NS Scott, MP Scott, PG Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou. 2DRMP : A suite of two-dimensional R-matrix propagation codes. *Computer Physics Communications*, 180(12) :2424–2449, 2009.
2. PG Burke, CJ Noble, and P. Scott. R-matrix theory of electron scattering at intermediate energies. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 410(1839) :289–310, 1987.
3. C. Nvidia. Compute unified device architecture programming guide. 2007.
4. A. Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1 :l1–15, 2009.
5. G. Martinez, M. Gardner, and W. Feng. CU2CL : A CUDA-to-OpenCL Translator for Multi-and Many-core Architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2011.
6. P. Fortin, R. Habel, F. Jezequel, JL Lamotte, and NS Scott. Deployment on GPUs of an application in computational atomic physics. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1359–1366. IEEE, 2011.
7. Xavier Leroy. The Objective Caml system release 4.00 : Documentation and user's manual. Technical report, Inria, 2012. `http://caml.inria.fr`.
8. M Bourgoin, E Chailloux, and JL Lamotte. SPOC : GPGPU Programming through Stream Processing with OCaml. *Parallel Processing Letters*, 22(2) :1240007, May 2012.
9. C. NVIDIA. Cublas Library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.
10. S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. *Univ. of Tennessee, Knoxville, Univ. of California, Berkeley, Univ. of Colorado, Denver*, 2009.
11. C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. Starpu : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 2011.
12. Johan Enmyren and Christoph W. Kessler. Skepu : a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14. ACM, 2010.
13. M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl-a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1176–1182. IEEE, 2011.
14. M.M.T. Chakravarty, G. Keller, S. Lee, T.L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM, 2011.
15. G. Keller, M.M.T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices*, 45(9) :261–272, 2010.
16. R. Beck, H.W. Larsen, T. Jensen, and B. Thomsen. Extending Scala with General Purpose GPU Programming. Technical report, Adlborg University, Departement of Computer Science, 2011.