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

Compilation et optimisation statique des communications hôte-accélérateur

Mehdi AMINI[1,2], Fabien COELHO[2], François IRIGOIN[2] et Ronan KERYELL[1]

[1] HPC Project, Meudon, France
[2] MINES ParisTech/CRI, Fontainebleau, France

Résumé
La puissance de calcul disponible dans les machines hybrides à base d'accélérateurs matériels nécessite de réécrire les programmes selon un modèle complexe et réparti : les données traitées doivent être copiées entre mémoire de l'hôte et mémoire de l'accélérateur. Cette contrainte est régulièrement identifiée comme *le* goulet d'étranglement. Nous proposons une transformation automatique que nous avons implémentée comme une passe de compilation au sein de PIPS/PAR4ALL qui génère statiquement des instructions de copies entre l'hôte et l'accélérateur selon deux stratégies « au plus tôt » et « au plus tard ». De plus les communications inutiles avec l'hôte résultant d'une réutilisation de données entre deux appels de noyaux sont évitées. Alors que les instructions initiant les transferts mémoire sont générées statiquement, un support exécutif associe lorsque nécessaire les tableaux de données de l'hôte à des zones mémoires sur l'accélérateur. Nous présentons les gains obtenus avec des expériences.

Mots-clés : Parallélisation automatique, optimisation des communications, compilation source-à-source, architectures parallèles hétérogènes, GPU

1. Introduction

Les machines hybrides à base d'accélérateurs matériels se généralisent pour améliorer les performances d'exécution des codes massivement parallèles. Déjà en 2008, *Roadrunner* équipé d'accélérateurs *PowerXCell 8i* s'installait en tête du classement TOP500. L'édition de novembre 2010 inclut dans son top 5 trois hybrides accompagnés de GPU (accélérateurs graphiques) NVIDIA. Ces accélérateurs matériels hautement parallèles permettent potentiellement de meilleurs rapports performance/prix et calcul/consommation comparés aux processeurs multicœurs conventionnels. On assiste au même phénomène dans le domaine des systèmes embarqués (NVIDIA Tegra...) et le futur proche semble hétérogène.

Malheureusement le modèle de programmation utilisé est complexe : avec des mémoires distribuées, le code exécuté sur l'accélérateur ne peut souvent pas accéder à la mémoire de l'hôte et réciproquement. Des communications explicites doivent être lancées pour permettre d'y accéder, souvent via des bus lents (bus PCI à 8 Go/s) et c'est *le* goulet d'étranglement des machines hybrides [7, 17, 5]. Si des solutions récentes permettent de se passer de cette copie explicite, le passage via le bus PCI est toujours obligatoire.

Nous proposons avec PAR4ALL [12] une initiative open-source pour fédérer des efforts autour de compilateurs permettant de paralléliser automatiquement des applications vers des architectures hybrides. La stratégie de compilation de base génère un code parallèle correct mais non efficace en général à cause de communications redondantes hôte-accélérateur.

Des études ont déjà été réalisées sur l'optimisation des communications dans le cadre de machines à mémoire répartie, par exemple en utilisant la fusion de messages dans le cadre SPDD (*Single Program Distributed Data*) [9] ou avec des analyses de flots de données fondées sur des régions de tableau pour éliminer les communications redondantes et masquer les communications restantes avec des calculs [10].

Nous proposons d'appliquer des techniques similaires au déport de calcul du couple hôte-accélérateur. Dans cet article, nous présentons succintement les approches existantes visant à exploiter les accélérateurs (§ 2), puis nous identifions des cas pratiques de simulation numérique pouvant profiter de la présence d'un accélérateur et nous montrons les limitations d'une transformation automatique en l'absence d'optimisation spécifique aux communications (§ 3). Nous présentons ensuite notre analyse de

flot de données et la génération statique des appels DMA de transferts entre hôte et accélérateur (§ 4), enfin nous évaluons notre solution sur un code de simulation numérique cosmologique ainsi que sur une résolution itérative suivant une méthode de JACOBI (§ 5).

2. Transformations automatisées ou semi-automatisées pour accélérateurs matériels

La programmation des accélérateurs est un travail manuel fastidieux pour le programmeur. Au mieux il disposera de langages proches du C tels que CUDA ou OPENCL ou d'API ; au pire il devra s'approcher de l'assembleur voire de langages de synthèse de circuits spécifiques tels que VHDL. NVIDIA CUDA est une extension du langage C propriétaire, limitée aux GPU NVIDIA. Le standard OPENCL propose une API permettant de s'abstraire de la machine cible, mais les constructeurs peuvent proposer des extensions propriétaires, rendant *de facto* le code final spécifique à un accélérateur ou une famille d'accélérateur. Les circuits de types FPGA sont eux généralement élaborés en VHDL, même s'il existe des outils de génération tels que le compilateur C-to-VHDL d'Altera `c2h`.

2.1. Approche semi-automatisée

Des compilateurs récents offrent une approche incrémentale pour le portage de code vers les accélérateurs matériels. C'est le cas de PGI [18] qui propose à l'utilisateur d'identifier à l'aide de directives insérées dans le code source les portions de code sujettes à être déportées sur l'accélérateur. Le compilateur se charge de la génération automatique du code du noyau, éventuellement aidé par des directives supplémentaires concernant notamment allocation et copie des données.

C'est également le cas du compilateur de CAPS à l'aide des directives HMPP [4]. Il permet à l'utilisateur de décrire l'ensemble des paramètres nécessaires à la génération de code. Les possibilités de spécialisation et d'optimisation sont accrues, mais avec le même revers que les extensions propriétaires d'OPENCL, c'est à dire un code spécifique à une architecture.

Des initiatives ont également été proposées pour la transformation de code annoté pour OPENMP vers CUDA [15, 16]. Le modèle de programmation des GPUs et le paradigme hôte-accélérateur a conduit rapidement aux limites de l'approche, OPENMP étant conçu pour les machines à mémoire partagée. Des travaux ultérieurs ont abouti à proposer une extension à OPENMP pour prendre en compte la spécificité de CUDA [14] (ce qui n'est pas sans rappeler HICUDA [11]) et aboutissant à nouveau à un code spécialisé.

2.2. Approche entièrement automatisée : PAR4ALL

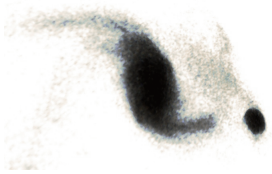
PAR4ALL est une initiative open-source aidant le développement de compilateurs paralléliseurs, basés sur des transformations source-à-source [12]. La version actuelle permet de générer de l'OPENMP à partir de C et Fortran pour les architectures à mémoire partagée et vers CUDA pour les accélérateurs matériels de type GPU NVIDIA. La génération d'OPENCL est en cours de finalisation. Le cœur du projet est le système de transformation de programme PIPS [13].

Le processus de transformation automatisée dans PAR4ALL repose fortement sur les capacités des passes de PIPS. Ce dernier utilise une librairie d'algèbre linéaire [1] pour manipuler une représentation polyédrique des abstractions d'un programme [8] et détecter les nids de boucles parallèles. Ces boucles sont ensuite extraites dans des fonctions marquées exécutables sur le GPU, ce qu'on appelle des noyaux dans la terminologie de programmation hybride. Des analyses de régions convexes de tableau [6] sont utilisées pour définir les données utilisées et produites par un noyau. Un prélude et un postlude sont ajoutés autour du noyau pour allouer la mémoire nécessaire et copier les données utilisées de la mémoire de l'hôte vers le GPU, puis les données produites du GPU vers la mémoire de l'hôte.

3. Application *Stars-PM*

Nous allons illustrer le processus des transformations automatiques de base de PAR4ALL sur un exemple de simulation numérique, un code cosmologique à N-corps selon une méthode *Particle-Mesh* intitulée *Stars-PM*. La version séquentielle a été écrite en C à l'Observatoire Astronomique de Strasbourg et a fait l'objet d'une étude pour un portage et une optimisation manuelle sur GPU [2]. Le principe consiste à modéliser les interactions gravitationnelles entre des particules dans l'espace. La résolution tridimensionnelle de ces interactions est réalisée par interpolation sur une discrétisation de l'espace.

Lors de l'exécution, les conditions initiales sont lues depuis un fichier, puis une boucle principale itère



(a) Exemple de simulation cosmologique où un satellite génère des bras en spirales dans un disque galactique.

```
int main(int argc, char *argv[]) {
  /* Read data from a file */
  init_data(argv[1]);

  /* Main temporal loop */
  for(t = 0; t < T; t+=DT)
    iteration(...);

  /* Output the results to a file */
  write_data(argv[2]);

  return 0;
}
```

(b) Synoptique de code minimal représentatif de simulation numérique.

```
void iteration(coord pos[NP][NP][NP],
               coord vel[NP][NP][NP],
               float dens[NP][NP][NP],
               int data[NP][NP][NP],
               int histo[NP][NP][NP]) {

  /* Découpe l'espace tridimensionnel
     selon une grille régulière */
  discretisation(pos, data);
  /* Calcul de la densité sur la grille */
  histogram(data, histo);
  /* Calcul du potentiel sur la grille
     dans l'espace de Fourier */
  potential(histo, dens);
  /* Calcul dans chaque dimension de la force
     et application à la vitesse des particules */
  forcex(dens, force);
  updatevel(vel, force, data, 0, dt);
  forcey(dens, force);
  updatevel(vel, force, data, 1, dt);
  forcez(dens, force);
  updatevel(vel, force, data, 2, dt);
  /* Déplacement des particules */
  updatepos(pos, vel);
}
```

(c) Structure d'un pas de temps dans l'exemple de simulation cosmologique.

FIGURE 1 – Exemple de programme de simulation cosmologique.

sur des pas de temps successifs. Ce sont les itérations de cette boucle qui représentent le traitement effectué. Après un nombre de pas de temps initialement défini, la boucle se termine et le résultat est enregistré dans un fichier. Cette structure générale de programme (voir code simplifié figure 1b) se rencontre typiquement dans le cadre des simulations numériques. Le contenu du pas de temps est illustré sur la figure 1c dans le cas du programme de cosmologie.

3.1. Exemple de transformation

Comme introduit en section 2.2, le processus de transformation automatisée dans PAR4ALL est basé sur la détection des nids de boucles parallèles. Ces boucles sont ensuite extraites pour former des noyaux.
Le code simplifié de la fonction discretization (pos, data) reproduit avant et après transformation sur la figure 2 illustre ce processus. Le nid de boucles est identifié comme parallèle et sélectionné pour être transformé en noyau. Le corps de la fonction est extrait dans une nouvelle fonction qui s'exécutera sur le GPU, et le nid de boucle est remplacé par un lancement de ce noyau. Les analyses de régions $\mathcal{IN}$ et $\mathcal{OUT}$ [6] permettent d'établir quelles zones de tableau doivent être copiées en entrée et en sortie du noyau.

3.2. Mise en évidence des limites de l'approche

Les copies de données vers la mémoire du GPU se font par des transferts DMA depuis la RAM au dessus du bus PCI-express. Ce dernier offre actuellement une bande passante théorique de 8 Go/s. Ce débit peut paraître considérable mais est en fait faible par rapport au débit disponible entre le GPU et sa mémoire, de l'ordre de 150 Go/s. Cette limitation peut donc annuler totalement l'intérêt du GPU.
Dans la pratique il faut compter un temps de démarrage et on obtient un maximum de 5,7 Go/s de l'hôte vers le GPU et 6,2 Go/s dans l'autre sens. Ce débit s'obtient à partir de quelques dizaines de Mo, mais diminue avec de plus petites quantités de données à transférer ; d'environ 4 Go/s pour 1 Mo de données, on tombe à quelques centaines de Mo/s pour quelques Ko de données. De plus ce débit est à réduire de moitié quand les zones mémoires à transférer ne sont pas allouées de manière *épinglée*, c'est-à-dire sous forme d'une zone contiguë en mémoire physique et avec des pages que le système d'exploitation ne peut délester sur disque.

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]){
  int i, j, k;
  float x, y, z;
  for (i = 0; i < NP; i++)
    for (j = 0; j < NP; j++)
      for (k = 0; k < NP; k++) {
        x = pos[i][j][k].x;
        y = pos[i][j][k].y;
        z = pos[i][j][k].z;
        data[i][j][k] = (int)(x/DX)*NP*NP
                      + (int)(y/DX)*NP
                      + (int)(z/DX);
      }
}
```

```
void discretization(coord pos[NP][NP][NP],int data[NP][NP][NP]) {
 // Pointers to memory on accelerator:
 coord (*pos0)[NP][NP][NP] = (coord (*)[NP][NP][NP]) 0;
 int (*data0)[NP][NP][NP] = (int (*)[NP][NP][NP]) 0;
 P4A_accel_malloc((void **) &data0, sizeof(int)*NP*NP*NP);
 P4A_accel_malloc((void **) &pos0, sizeof(coord)*NP*NP*NP);
 P4A_copy_to_accel(sizeof(coord)*NP*NP*NP, pos, *pos0);
 P4A_call_accel_kernel_2d(discretization_kernel,NP,NP,*pos0,*data0);
 P4A_copy_from_accel(sizeof(int)*NP*NP*NP, data, *data0);
 P4A_accel_free(data0);
 P4A_accel_free(pos0);
}
// The kernel corresponding to previous loop body
P4A_accel_kernel discretization_kernel( coord *pos, int *data ) {
   int k; float x, y, z;
   int i = P4A_vp_1; //  P4A_vp_* are mapped from CUDA BlockIdx.*
   int j = P4A_vp_0; //  and ThreadIdx.* to loop indices
   // Iteration clamping to avoid GPU iteration overrun:
   if (i<=127&&j<=127)
      for(k = 0; k < NP; k += 1) {
         x = (*(pos+k+NP*NP*i+NP*j)).x;
         y = (*(pos+k+NP*NP*i+NP*j)).y;
         z = (*(pos+k+NP*NP*i+NP*j)).z;
         *(data+k+NP*NP*i+NP*j) = (int)(x/DX)*NP*NP
                                + (int)(y/DX)*NP
                                + (int)(z/DX);
      }
}
```

FIGURE 2 – Code de la fonction discretization avant et après transformation automatique.

En réalisant des tests avec une taille de référence pour notre espace, un cube de 128 cellules de coté et autant de particules que de cellules, nous avons pour la fonction discretization une copie vers le GPU des positions des particules, soit 25 Mo, et une copie depuis le GPU de l'assocation particule-cellule, soit 8 Mo. Le temps nécessaire pour effectuer ces deux copies est 5ms, et même 12ms si l'ECC (correction d'erreurs sur la mémoire) est activée. Avec l'ECC, il faut également compter 10ms supplémentaires pour les allocations et désallocations mémoire. L'exécution du noyau quant à elle s'effectue en 0,37ms. On constate que les copies de données représentent le plus gros potentiel de gain, ce qui justifie notre proposition.

3.3. Constat

En étudiant le code de la fonction iteration (fig. 1c) de cette application, on constate que chaque étape utilise des données qui ont été traitées à l'étape précédente. Le code parallélisé passe alors beaucoup de temps à copier des données depuis le GPU pour les y renvoyer immédiatement après. De plus, d'une manière générale, dans les codes de simulations numériques fonctionnant sur un schéma tel qu'indiqué figure 1b, les données de l'itération suivante auront été générées à l'itération précédente. Dans le meilleur des cas, on peut donc conserver les données sur le GPU durant toute l'exécution de la boucle.

4. Méthode d'optimisation

Nous proposons dans le compilateur une analyse qui estime l'emplacement global des tableaux entre l'hôte et le GPU à chaque endroit du code source. Nous décidons ensuite statiquement à quels endroits générer les copies. La stratégie actuelle consiste à lancer les copies de l'hôte vers le GPU au plus tôt et à l'inverse les copies du GPU vers l'hôte au plus tard.

4.1. Définitions

L'analyse que nous proposons s'intitule *kernel data mapping*, se place du point de vue de l'hôte et consiste à calculer les ensembles suivants :

- $\mathcal{U}_A^{\geq}$ associe à chaque instruction les tableaux dont il est certain que la prochaine *u*tilisation se fera sur l'*a*ccélérateur ;
- $\mathcal{D}_A^{\leq}$ associe à chaque instruction les tableaux dont la dernière *d*éfinition a été faite sur l'*a*ccélérateur et qui n'ont pas été utilisés par l'hôte entre temps ;

– $\mathcal{T}_{H\to A}$ associe à chaque instruction les tableaux qu'il faut transférer vers l'accélérateur juste après exécution de l'instruction ;
– $\mathcal{T}_{H\leftarrow A}$ associe à chaque instruction les tableaux qu'il faut transférer depuis l'accélérateur juste avant exécution de l'instruction.

Ces ensembles sont initialement vides pour toute instruction du programme.

4.2. Construction intra-procédurale

L'analyse construit d'abord l'ensemble $\mathcal{D}_A^<$ en passe avant. Un tableau est défini sur le GPU en une instruction I ssi c'est le cas de toutes les instructions la précédant directement dans le graphe de contrôle et si le tableau n'est ni lu ni écrit par l'hôte (c'est à dire pas dans les ensembles respectifs $\mathcal{R}(I)$ ou $\mathcal{W}(I)$, calculés par PIPS) :

$$\mathcal{D}_A^<(I) = \Big(\bigcap_{I'\in prec(I)} \mathcal{D}_A^<(I')\Big) - \mathcal{R}(I) - \mathcal{W}(I) \quad (1)$$

L'initialisation se fera au premier appel I_k de noyau k avec l'ensemble des tableaux écrits par le noyau qui sont utilisés ultérieurement (ci-après dénommé ensemble $\mathcal{OUT}$). L'équation suivante s'applique à chaque site d'appel de noyau :

$$\mathcal{D}_A^<(I_k) = \mathcal{OUT}(I_k) \cup \bigcap_{I'\in prec(I_k)} \mathcal{D}_A^<(I') \quad (2)$$

Une passe arrière est ensuite réalisée pour construire $\mathcal{U}_A^>$. Un tableau en une instruction I aura sa prochaine utilisation sur l'accélérateur ssi c'est le cas pour tous les instructions directement suivantes dans le graphe de contrôle à conditions qu'il ne soit pas écrit par I.

$$\mathcal{U}_A^>(I) = \Big(\bigcup_{I'\in succ(I)} \mathcal{U}_A^>(I')\Big) - \mathcal{W}(I) \quad (3)$$

Comme précédemment, $\mathcal{U}_A^>$ est initialement vide et est construit au niveau des appels aux noyaux à partir des tableaux nécessaires au calcul $\mathcal{IN}(I_k)$ et des tableaux écrits. Ces derniers doivent être copiés sur le GPU si on ne peut pas établir qu'il seront écrits en intégralité par le noyau. En effet dans ce cas lors d'une éventuellement copie du GPU vers l'hôte après exécution, les données non écrites par le GPU et donc présentes uniquement sur le CPU seraient perdues :

$$\mathcal{U}_A^>(I_k) = \mathcal{IN}(I_k) \cup \mathcal{W}(I_k) \cup \bigcup_{I'\in succ(I_k)} \mathcal{U}_A^>(I') \quad (4)$$

Un tableau doit être transmis de l'accélérateur vers l'hôte après une instruction I si sa dernière définition est sur l'accélérateur mais que ce n'est pas le cas dans au moins une instruction suivante directe :

$$\mathcal{T}_{H\leftarrow A}(I) = \mathcal{D}_A^<(I) - \bigcap_{I'\in succ(I)} \mathcal{D}_A^<(I') \quad (5)$$

Cet ensemble est directement utilisé pour générer une opération de copie au plus tard.
Un tableau doit être transmis depuis l'hôte vers l'accélérateur si la prochaine utilisation est sur l'accélérateur. Pour réaliser la communication au plus tôt, on la démarre après l'endroit de son écriture $\mathcal{W}(I)$:

$$\mathcal{T}_{H\to A}(I) = \mathcal{W}(I) \cap \bigcup_{I'\in succ(I)} \mathcal{U}_A^>(I') \quad (6)$$

4.3. Interprocéduralité

Les appels aux noyaux de calculs étant potentiellement enfouis profondément dans des appels de fonction, une réutilisation de données entre noyaux de calcul nécessite une analyse interprocédurale. Cette

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]) {
  //generated variable
  int posSize = NP*NP*NP*sizeof(coord);
  coord *pos0 = P4A_runtime_resolve(pos,posSize);

  int dataSize = NP*NP*NP*sizeof(int);
  int *data0 = P4A_runtime_resolve(pos,dataSize);

  // Call kernel
  P4A_call_accel_kernel_2d(discretization_kernel,
                           NP, NP, *pos0, *data0);
}
```

```
int main(int argc, char *argv[]) {
  /* .... declarations .... */

  /* Read data from a file */
  init_data(argv[1], ....);

  int posSize = NP*NP*NP*sizeof(coord);
  P4A_runtime_copy_to_accel(pos, posSize);

  /* main loop */
  for(t = 0; t < T; t+=DT)
    iteration(...);

  P4A_runtime_copy_from_accel(pos, posSize);

  /* output */
  write_data(argv[2],....);

  return 0;
}
```

FIGURE 3 – Code simplifié des fonctions discretization et main après optimisation des communications.

situation est illustrée par la fonction itération (fig. 1c) pour laquelle chaque étape correspond à un ou plusieurs noyaux de calcul dans les fonctions appelées.

Notre approche est de faire une analyse arrière sur le graphe des appels. Pour chaque fonction f un ensemble *summary* $\overline{\mathcal{D}^{<}_{\mathbb{A}}}(f)$ et $\overline{\mathcal{U}^{>}_{\mathbb{A}}}(f)$ est produit qui résume l'information au niveau des tableaux de la fonction. Ces ensembles servent à déterminer l'impact sur les ensembles $\mathcal{D}^{<}_{\mathbb{A}}$ et $\mathcal{U}^{>}_{\mathbb{A}}$ au niveau de l'appel dans les appelants. En effet lorsqu'on rencontre lors de l'analyse un appel c de fonction f, chacun des arguments de la fonction figurant dans $\overline{\mathcal{U}^{>}_{\mathbb{A}}}$ doit être copié sur le GPU avant l'appel, car la première utilisation suivant l'appel sera dans un noyau. Un argument qui figure dans $\overline{\mathcal{D}^{<}_{\mathbb{A}}}$ a été produit sur le GPU et n'a pas encore été rapatrié lorsque l'appel se termine. On devra le copier depuis l'accélérateur vers l'hôte avant toute utilisation.

Les équations 1 et 3 sont ainsi modifiées pour les appels de fonction en rajoutant un opérateur de traduction entre paramètres d'appels et paramètres formels $\text{trans}_{f->c}$:

$$\mathcal{D}^{<}_{\mathbb{A}}(c) = \left(\text{trans}_{f->c}(\overline{\mathcal{D}^{<}_{\mathbb{A}}}(f)) \cup \bigcap_{I' \in \text{prec}(c)} \mathcal{D}^{<}_{\mathbb{A}}(I')\right) - \mathcal{R}(c) - \mathcal{W}(c) \tag{7}$$

$$\mathcal{U}^{>}_{\mathbb{A}}(c) = \left(\text{trans}_{f->c}(\overline{\mathcal{U}^{>}_{\mathbb{A}}}(f)) \cup \bigcup_{I' \in \text{succ}(c)} \mathcal{U}^{>}_{\mathbb{A}}(I')\right) - \mathcal{W}(c) \tag{8}$$

On observe par comparaison du code optimisé de manière inter-procédurale (fig. 3) par rapport à l'approche locale (figure 2) que nous avons éliminé tout transfert.

4.4. Support exécutif (*runtime*)

Notre compilateur PAR4ALL fournit un support exécutif léger pour faciliter certaines opérations liées à notre approche, telle que la gestion de l'allocation mémoire. Cet exécutif est construit autour d'une table d'association d'une adresse sur l'hôte à une adresse sur l'accélérateur qui nous permet de la souplesse dans la gestion des allocations mémoire. Nous n'avons pas besoin de changer le code utilisateur, et seuls nos appels de noyaux et nos instructions de copies sont modifiés pour faire appel à l'exécutif.

La figure 3 montre le code simplifié de la fonction discretization après transformation. Le code est grandement simplifiée par rapport à la version précédente. On observe l'appel à l'exécutif pour retrouver l'adresse de la zone mémoire sur l'accélérateur qui aura été préallablement allouée pour les tableaux pos et data. Si cette allocation n'a jamais été réalisée, l'exécutif s'en charge à ce moment.

L'exécutif de PAR4ALL inclut également une structure de données permettant de convertir les appels à la librairie de traitement FFT `fftw3` vers des appels à `cufftw`, son équivalent CUDA.

Temps d'exécution	Simulation Cosmo.			Jacobi
	32	64	128	
Séquentiel	0,68	6,30	98,4	48,6
OPENMP 6 *threads*	0,16	1,28	16,6	13,8
CUDA base	0,84	4.72	25,9	95,6
Communications optimisées	0,10	0.32	2.1	3,8
Optimisation manuelles	0,05	0,26	1,8	

FIGURE 4 – Mesure du temps d'exécution en secondes des différentes versions de Jacobi et du programme de simulation cosmologique sur différentes tailles de jeux de données. Les transferts entre l'hôte et l'accélérateur sont inclus mais pas le chargement des jeux de données depuis le disque.

5. Expériences

Nous avons mesuré l'impact de notre schéma d'optimisation de communication sur la simulation *Stars-PM* présentée section 3. Nous avons utilisé des jeux de données de plusieurs tailles, allant de 32^3 à 128^3 particules. Pour chaque taille de jeux de données nous avons exécuté :
– le code séquentiel original,
– le code parallélisé automatiquement en OPENMP
– le code parallélisé sur GPU avec CUDA sans optimisation de communication,
– le code parallélisé sur GPU avec CUDA avec notre schéma d'optimisation,
– le code parallélisé sur GPU manuellement.
La génération du code des noyaux est réalisée sans optimisation particulière, et les noyaux sont exactement identiques pour les 2 versions automatiques sous CUDA. Le SDK NVIDIACUDA 3.2 pour la partie GPU, et la vestion 4.4.5 de GCC avec le niveau d'optimisaton *-O3* ont été utilisés.
Nous avons également évalué notre schéma sur un programme qui applique 400 itérations de JACOBI sur une image chargée dans un fichier. Le tableau figure 4 indique les mesures que nous avons réalisées sur une machine équipée de 2 Xeon Nehalem X5670 soit 12 cœurs à 2,93 GHz et d'un GPU NVIDIA Tesla C2060. Nous utilisons 6 threads en OPENMP, ce qui est la meilleure configuration testée.
L'impact de l'optimisation des communications est flagrant, avec une accélération moyenne de 11,8 par rapport à la version de base sur la simulation cosmologique, et de 25 sur l'exemple de JACOBI.
Nous avons également réimplémenté ce dernier exemple en utilisant la librairie *Starpu* [3] qui offre une approche totalement dynamique, obligeant toutefois à un effort supplémentaire lors du développement. Nous avons pu vérifier que les communications étaient supprimées par *Starpu* et nous avons obtenu un temps d'exécution de 10,3 secondes, soit près de 3 fois plus long que notre optimisation statique entièrement automatisée.

6. Conclusion

Avec le déploiement des accélérateurs matériels, le recours à des transformations automatisées ou semi-automatisées à l'aide de directives prend une importance croissante.
Nous avons montré que l'impact des communications est important dans l'utilisation d'accélérateurs sur des codes massivement parallèles tels que les simulations numériques. L'optimisation des communications est donc un point clé dans l'obtention de performances.
Nous avons présenté une méthode d'optimisation qui répond à cette problématique et l'avons implémentée dans PIPS et PAR4ALL. Nous l'avons expérimentée sur un code de simulation numérique réelle ainsi que sur un code plus simple réalisant une relaxation selon la méthode de JACOBI. Nos mesures montrent une accélération jusqu'à un facteur 14 par rapport à une parallélisation naïve, et jusqu'à un facteur 8 par rapport à une version multi-processeurs OPENMP sur une machine bi-hexa-cœurs.
Au delà de ces premiers résultats expérimentaux, nous allons valider notre approche sur de nouvelles simulations et l'améliorer. Nous pensons également comparer notre méthode à ce que propose PGI et HMPP, bien que ceux-ci demande un effort de programmation de la part du développeur en comparaison de notre méthode entièrement automatique.

Les performances des processeurs multi-cœurs nous incitent à orienter nos efforts futurs vers une gestion du partage du travail entre l'hôte et l'accélérateur plutôt que d'avoir l'hôte qui attend la fin des calculs sur l'accélérateur, ouvrant des perspectives intéressantes sur la détermination des régions de tableaux à transférer.
Comme notre modèle actuel suppose que l'accélérateur dispose de suffisamment de mémoire pour contenir l'intégralité des tableaux, nous pensons gérer automatiquement la libération de mémoire sur l'accélérateur en cas de besoin pour permettre l'adaptation automatique des codes à des accélérateurs à mémoire plus modeste.

7. Remerciement

Les auteurs tiennent à remercier Béatrice CREUSILLET et Pierre JOUVELOT pour leur relecture attentive et leurs suggestions, ainsi que Serge GUELTON pour l'important travail sur la génération des noyaux de calcul dans PIPS, Dominique Aubert pour nous avoir permis d'exploiter son code de simulation cosmologique et de manière générale toute l'équipe PIPS et PAR4ALL pour le travail accompli.

Bibliographie

1. Corinne Ancourt, Fabien Coelho, François Irigoin, et Ronan Keryell. A linear algebra framework for static High Performance Fortran code distribution. *Scientific Programming*, 6(1) :3–27, 1997.
2. Dominique Aubert, Mehdi Amini, et Romaric David. A particle-mesh integrator for galactic dynamics powered by GPGPUs. In *International Conference on Computational Science : Part I*, ICCS '09.
3. Cédric Augonnet, Samuel Thibault, Raymond Namyst, et Pierre-André Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23 :187–198, February 2011.
4. Francois Bodin et Stephane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program.*, 17 :325–336, December 2009.
5. Yifeng Chen, Xiang Cui, et Hong Mei. Large-scale FFT on GPU clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 315–324, 2010.
6. Béatrice Creusillet et François Irigoin. Interprocedural array region analyses. *Int. J. Parallel Program.*, 24(6) :513–546, 1996.
7. Wenbin Fang, Bingsheng He, et Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3 :670–680, September 2010.
8. Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22, 1988.
9. Hans Michael Gerndt et Hans Peter Zima. Optimizing communication in SUPERB. In *Proceedings of the joint international conference on Vector and parallel processing*, CONPAR 90-VAPP IV, 1990.
10. Chun Gong, Rajiv Gupta, et Rami Melhem. Compilation techniques for optimizing communication on distributed-memory systems. ICPP '93, 1993.
11. Tianyi David Han et Tarek S. Abdelrahman. hiCUDA : a high-level directive-based language for GPU programming. In *Proceedings of GPGPU-2*. ACM, 2009.
12. HPC Project. Par4All initiative for automatic parallelization. `http://www.par4all.org`.
13. François Irigoin, Pierre Jouvelot, et Rémi Triolet. Semantical interprocedural parallelization : an overview of the PIPS project. ICS '91, pages 244–251, 1991.
14. Seyong Lee et Rudolf Eigenmann. OpenMPC : Extended OpenMP programming and tuning for GPUs. SC '10, pages 1–11, 2010.
15. Seyong Lee, Seung-Jai Min, et Rudolf Eigenmann. OpenMP to GPGPU : a compiler framework for automatic translation and optimization. PPoPP '09, pages 101–110, 2009.
16. Satoshi Ohshima, Shoichi Hirasawa, et Hiroki Honda. OMPCUDA : OpenMP execution framework for CUDA based on omni OpenMP compiler. In *Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 161–173. 2010.
17. Cleomar Pereira da Silva, Leandro F. Cupertino, Daniel Chevitarese, Marco Aurelio C. Pacheco, et Cristiana Bentes. Exploring data streaming to improve 3d FFT implementation on multiple GPUs. In *International Symposium on Computer Architecture and High Performance Computing Workshops*, 2010.
18. Michael Wolfe. Implementing the PGI accelerator model. In *GPGPU*, pages 43–50, 2010.