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

Compilation pour cibles hétérogènes: automatisation des analyses, transformations et décisions nécessaires

Serge GUELTON[1], Ronan KERYELL[2] et François IRIGOIN[3]

[1] Institut Télécom, Télécom Bretagne, Info/HPCAS, Plouzané, France
[2] HPC Project, Meudon, France
[3] MINES ParisTech, CRI, Fontainebleau, France

Résumé
Les accélérateurs matériels, telles les cartes FPGA ou les cartes graphiques, apportent une alternative ou un complément intéressant aux processeurs multi-cœurs classiques pour de nombreuses applications scientifiques. Il est cependant coûteux et difficile d'y porter des applications existantes ; et les compilateurs standards, traditionnellement portés sur la génération de code pour processeurs séquentiels, ne disposent pas des abstractions nécessaires à la génération automatique et re-ciblable de code pour ces nouvelles cibles. Cet article présente un ensemble de transformations de code de haut niveau reposant sur une abstraction à plusieurs niveaux de l'architecture des accélérateurs actuels et permettant de construire des compilateurs spécifiques à chaque cible en se basant sur une infrastructure commune. Ces transformations ont été utilisées pour construire avec PIPS deux compilateurs complètement automatisés pour un processeur embarqué à base de FPGA et pour GPU NVIDIA avec PAR4ALL.

Mots-clés : compilation source-à-source, parallélisation automatique, GPU, accélérateur matériel, calcul hétérogène

1. Contexte

Deux possibilités s'offrent actuellement à qui veut accélérer une application, et toutes deux sont basées sur l'utilisation du parallélisme : les processeurs multi-cœurs classiques et les accélérateurs matériels. La seconde possibilité touche un champ d'application moins vaste que la première mais offre un rapport accélération par euro et accélération par Watt potentiellement plus intéressant. Cependant ces accélérateurs matériels souffrent de limitations caractéristiques :
– difficulté et coût du portage d'applications (coût du ticket d'entrée) ;
– diversité des machines cibles impliquant une multiplication des programmes sources ;
– coût de sortie de technologie si on veut porter un programme optimisé d'une architecture à l'autre.
Afin de pérenniser l'investissement dans un accélérateur, il est important qu'un utilisateur puisse s'affranchir de ces trois contraintes : une approche automatique, quand elle est possible, est alors particulièrement intéressante car elle offre la possibilité de ne maintenir qu'un seul code et de laisser à l'outil la charge de générer le code spécifique à chaque architecture.
La génération automatique ou semi-automatique de code pour accélérateur matériel connaît d'ailleurs un regain d'intérêt ces dernières années, visant à combler le fossé existant entre les performances offertes par les cartes graphiques (GPU) et leur difficulté de programmation. On peut noter que sur les 10 premières machines du top500 de novembre 2010, 4 sont hétérogènes, dont la première, le cluster Tianhe-1a, utilisant des GPU Fermi de NVIDIA. NVIDIA propose le langage CUDA [15], une extension du langage C89, et le compilateur `nvcc` pour générer du code spécifique à ses GPU. Ceci limite naturellement à une famille d'accélérateurs et impose de surcroît une réécriture des cœurs de calculs. *A contrario* OPENCL [7] propose une API et un langage étendant C99 et proche de CUDA mais permettant de s'abstraire de la machine cible, ce qui permet d'avoir un code portable mais ne garantit pas la portabilité des performances. Par exemple [11] démontre sur un ensemble de noyaux qu'une implémentation OPENCL peut s'exécuter entre 10 et 40% moins vite que le noyau CUDA équivalent, temps de transferts inclus. Le compilateur de

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
      if( j==0 ) out[i][j]=MIN(in[i][j],in[i][j+1]);
      else if(j==m-1) out[i][j]=MIN(in[i][j-1],in[i][j]);
      else out[i][j]= MIN(in[i][j-1],in[i][j],in[i][j+1]);
}
```

FIGURE 1 – Exemple de calcul d'érosion horizontale.

PGI [18] propose une approche semi-automatique où l'utilisateur identifie les noyaux de calcul à l'aide de directives insérées dans le code d'origine. Le compilateur se charge de la génération automatique du code du noyau, éventuellement aidé par des directives supplémentaires permettant de prendre le relais des analyses quand celles-ci ne sont pas suffisantes. Cette approche permet un portage incrémental des applications. Le compilateur HMPP de CAPS Entreprise [5] utilise également une approche à base de directive, mais l'utilisateur doit décrire l'ensemble des paramètres nécessaires à la génération de code, avec des possibilités de spécialisation et d'optimisation accrues. Dans [13] les auteurs utilisent les informations présentes dans les directives OPENMP comme base pour générer du code CUDA. La génération de code pour FPGA, malgré certains points communs, a connu des développements différents avec par exemple le compilateur `c2h` de Altera, ou Mitrion-C [12], qui prennent en entrée un sous ensemble du langage C pour générer du VHDL. [6, 8, 1] donnent de nombreux éclaircissements sur les moyens qui peuvent être employés pour améliorer le fonctionnement de tels outils.

Dans ce contexte, il paraît difficile de fournir un outil capable de prendre en entrée un unique code source, même annoté, et d'en dériver autant de versions que d'architectures cibles. Dans cet article, nous proposons une approche qui consiste en l'analyse des différentes contraintes qu'une architecture exerce sur un code (e.g. taille mémoire ou nombre de nœuds de calcul) et en l'utilisation de transformations de code pour satisfaire ces contraintes. Ce couplage contrainte/transformations offre la possibilité de construire facilement un compilateur dédié en réutilisant de nombreuses transformations, ce qui permet de produire autant de compilateurs que d'architectures cibles.

Pour illustrer nos propos, les transformations de code et analyses présentées dans cet article sont explicitées sur un cas test représentatif d'une application de traitement de signal : une érosion horizontale dont le code est donnée en figure 1. Toute les transformations présentées sont implémentées dans l'infrastructure de compilation source-à-source PIPS [9, 10] et les fragments de code exhibés sont présentés tels que produits par le compilateur au remaniement de mise en page près.

Ce document est organisé de la manière suivante : la section 2 décrit le modèle commun considéré, la section 3 propose une méthode pour estimer l'intensité de calcul d'une instruction, la section 4 décrit l'étape d'extraction des noyaux de calculs, la section 5 présente le processus de génération des communications tandis que la section 6 propose une technique pour prendre en compte les contraintes de taille mémoire. Des résultats expérimentaux sont présentés en section 7 et la dernière section conclut et propose des perspectives.

2. Modélisation

Que ce soient les cartes graphiques ou les cartes FPGA, les accélérateurs hybrides se placent dans un paradigme maître-travailleur, où un processeur **hôte** délègue le calcul intensif à un **accélérateur** en utilisant une séquence d'appel de type chargement - travail - déchargement (*load - work - store*). Cette séquence est une constante du calcul hybride et est à la base des propositions faites dans cet article.

Le chargement et le déchargement sont effectués par des opérateurs de copie, qui peuvent être asynchrones ou non suivant la cible. Ces opérateurs peuvent travailler sur des mémoires avec accès à n dimensions, où n est fixé par l'architecture cible. De plus, il peut y avoir des contraintes spécifiques supplémentaires concernant l'alignement des données ou la taille de données transférées. La taille de la mémoire cible a aussi une influence sur cette séquence car il est impossible de transférer en cumulé plus de données que ce que peut contenir la mémoire de l'accélérateur. L'accélération peut venir d'une

	Quadro FX 2700M	Ter@pix
accélération	MIMD +SIMD	SIMD
nombre de nœuds	48	128
dimension DMA	1d	2d
taille DMA	pas de contrainte	$k \times 128$
mémoire accélérateur (*octets*)	512 M (global RAM)	1024

FIGURE 2 – Caractéristiques des accélérateurs Quadro et Terapix.

implémentation spécifique de certaines fonctionnalités ou de l'utilisation massive du parallélisme. Le tableau 2 résume les paramètres qui sont significatifs pour le calcul hybride pour deux cibles : le processeur vectoriel embarqué Ter@pix [3] de THALES et une carte graphique NVIDIA Quadro FX 2700M bas de gamme pour avoir une consommation électrique comparable à un accélérateur embarqué.
De nombreux autres paramètres doivent ensuite être pris en compte, mais ils sont spécifiques à chaque cible et offrent moins de possibilité d'agrégation. On citera par exemple l'absence de division entière sur Ter@pix ou l'agrégation des accès mémoire (*coalescing*) sur carte graphique.

3. Estimation de l'intensité de calcul

La calcul hybride nécessite l'identification de noyaux de calculs candidats à une exécution efficace sur accélérateur matériel. Nous choisissons de modéliser par la formule (1) le temps d'exécution t_n d'un noyau de calcul exécuté à l'instruction I dans l'état mémoire σ :

$$t_n(I,\sigma) = \tau_0 + \frac{V(I,\sigma)}{B} + \frac{t_s(I,\sigma)}{a_{th}} \tag{1}$$

où $V(I,\sigma)$ est le volume mémoire utilisé par l'instruction I en fonction de σ et B est la bande passante entre l'hôte et l'accélérateur. τ_0 est un surcoût constant lié par exemple à l'initialisation d'un transfert, $t_s(I,\sigma)$ est le temps d'exécution séquentiel de I depuis l'état σ sur l'hôte, fonction de σ et a_{th} est l'accélération théorique procurée par l'accélérateur hors communication. Pour que la déportation d'un calcul sur accélérateur soit intéressante, il faut $t_n(I,\sigma) < t_s(I,\sigma)$ impliquant $a_{th} > 1$. Cela se traduit par la condition (2) reformulée en (3)

$$\tau_0 + \frac{V(I,\sigma)}{B} < t_s(I,\sigma) \times \frac{a_{th}-1}{a_{th}} \tag{2}$$

$$(\frac{1}{B} + \frac{\tau_0}{V(I,\sigma)}) \times \frac{a_{th}}{(a_{th}-1)} < \frac{t_s(I,\sigma)}{V(I,\sigma)} \tag{3}$$

L'étude asymptotique de l'équation (3) nous renseigne alors sur la viabilité de la déportation du noyau. On peut obtenir une approximation de $t_s(I,\sigma)$ en utilisant la méthode d'estimation de temps de calcul décrite dans [19]. Pour des codes structurés, on obtient ainsi un polynôme fonction de différentes variables du programme à l'état σ. On peut obtenir une approximation de $V(I,\sigma)$ en se basant sur l'analyse de régions de tableaux convexes [4]. Appliquée sur une instruction I à l'état σ, elle fournit une approximation de l'ensemble des données lues ou écrites par I sous forme de système de contrainte. Il est alors possible de calculer le volume du polyèdre associé [2] afin d'obtenir $V(I,\sigma)$ sous forme polynomiale.
La figure 3 montre le résultat de ces analyses pour le code de la figure 1. On y trouve $t_s(I,\sigma) = 20.25 \times m \times n + 4 \times n + 3$ et $V(I,\sigma) = 2 \times m \times n$ soit un quotient qui tend vers 10.125 quand $m \to \infty$ et $n \to \infty$. Dans ce cas, l'éligibilité du noyau est fonction de la bande passante B.

4. Extraction de fonctions

Une fois les fragments de code intéressants identifiés, il est nécessaire de les transformer en fonctions car cela permet de bien séparer le code à exécuter sur l'hôte et le code à exécuter sur la cible. Si l'on construit le processus d'extraction de telle sorte que les fonctions extraites n'utilisent pas de variable globale ni

```
//    20.25*m.n + 4*n + 3 (SUMMARY)
void erode(int n, int m, int in[n][m], int out[n][m])
```

(a) Analyse de complexité

```
//  <in[PHI1][PHI2]-R-MAY-{0<=PHI1, PHI1+1<=n, 0<=PHI2, 1<=PHI2+m,
//  PHI2<=m, 1<=m}>
//  <out[PHI1][PHI2]-W-MAY-{0<=PHI1, PHI1+1<=n, 0<=PHI2, PHI2+1<=m}>
void erode(int n, int m, int in[n][m], int out[n][m])
```

(b) Analyse de régions de tableaux

FIGURE 3 – Analyses de complexité et de région de tableaux sur un code d'érosion.

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
   for(int i = 0; i <= n-1; i += 1)
      noyau(m, n, i, in, out);
}
void noyau(int m, int n, int i, int in[n][m], int out[n][m]) {
   for(int j = 0; j <= m-1; j += 1)
      if (j==0) out[i][j] = MIN(in[i][j], in[i][j+1]);
      else if (j==m-1) out[i][j] = MIN(in[i][j-1], in[i][j]);
      else out[i][j] = MIN(in[i][j-1], in[i][j], in[i][j+1]);
}
```

FIGURE 4 – Extraction de fonction sur un noyau d'érosion.

de valeur de retour, c'est aussi une façon d'isoler les variables qui seront locales à l'accélérateur (celles déclarées dans les fonctions extraites) et celles qui seront utilisées pour communiquer entre l'hôte et l'accélérateur (celles passées en paramètre de fonction). Le processus d'extraction utilise la formule (4) inspirée de [14] :

$$\mathrm{ExternalVars}(S) = \mathrm{ReferencedVars}(S) - \big(\mathrm{DeclaredVars}(S) \cup \mathrm{PrivateVars}(S)\big) \tag{4}$$

S est un fragment de code, $\mathrm{ExternalVars}$ symbolise les variables à passer en paramètre à la fonction créée à partir de S, $\mathrm{ReferencedVars}(S)$ est l'ensemble des variables référencées par S, $\mathrm{DeclaredVars}(S)$ est l'ensemble des variables déclarées dans S avec visibilité limitée à S et $\mathrm{PrivateVars}(S)$ est l'ensemble des variables privées de S. $\mathrm{ReferencedVars}$ et $\mathrm{DeclaredVars}$ sont construits par récursion sur la représentation interne de S et $\mathrm{PrivateVars}$ est construit par une phase de privatisation de variables [17].
Lors du processus d'extraction, il est également important de prendre en compte les dépendances de types, notamment à cause des tableaux à taille variable du langage C99. Ainsi, l'ensemble $\mathrm{ExternalVars}(S)$ doit être augmenté de l'ensemble des variables nécessaires à la définition de chacun des types des variables qu'il contient. La figure 4 illustre ce comportement sur l'exemple de la figure 1 où l'on extrait la boucle interne dans une nouvelle fonction noyau.

5. Génération des communications

Une fois le noyau de calcul identifié et extrait dans une fonction séparée, il est nécessaire de générer les communications et les allocations mémoires nécessaires à l'appel distant. Pour cela, une technique appelée **isolation d'instructions** est introduite. Étant donnée une instruction I, pour chaque variable v référencée par I il est possible de calculer une approximation des régions de tableaux lues (resp. écrites) par cette instruction pour cette variable, que l'on note $\mathcal{R}_r(v)$ (resp. $\mathcal{R}_w(v)$) pour les régions lues (resp. écrites). Ces régions sont soit exactes $\mathcal{R}_r^{=}(v)$ soit sur-approximées $\mathcal{R}_r^{\succeq}(v)$ (représentées par -*MAY* dans

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  int (*out0)[n][m] =  0, (*in0)[n][m+1] =  0;
  P4A_accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
  P4A_accel_malloc((void **) &out0, sizeof(int)*n*m);
  P4A_copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0,, &in[0][0], *in0);
  P4A_copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
  for(int i = 0; i <= n-1; i += 1)
    for(int j = 0; j <= m-1; j += 1)
      if (j==0) (*out0)[i][j] = MIN((*in0)[i][j], (*in0)[i][j+1]);
      else if (j==m-1) (*out0)[i][j] = MIN((*in0)[i][j-1], (*in0)[i][j]);
      else (*out0)[i][j] = MIN((*in0)[i][j-1],(*in0)[i][j],(*in0)[i][j+1]);
  P4A_copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
  P4A_accel_free(in0);
  P4A_accel_free(out0);
}
```

FIGURE 5 – Illustration du comportement de l'isolation d'instructions.

la sortie de PIPS) si on n'arrive pas à calculer exactement la région. Il existe un lien fort entre ces régions de tableau et les transferts de données.

Transferts sortants Il faut copier de l'accélérateur vers l'hôte toutes les données potentiellement écrites :

$$\mathcal{T}_{H\leftarrow A}(I) = \{\mathcal{R}_w(v) \,|\, v \in I\} \tag{5}$$

Transferts entrants Il faut copier de l'hôte vers l'accélérateur toutes les données potentiellement lues :

$$\mathcal{T}_{H\rightarrow A}(I) = \{\mathcal{R}_r(v) \,|\, v \in I\}$$

En fait il faut copier toutes les données pour lesquelles on n'a pas une garantie d'écriture par I ; dans le cas contraire on pourrait transférer depuis l'accélérateur des données non initialisées.

$$\mathcal{T}_{H\rightarrow A}(I) = \{\mathcal{R}_r(v) \,|\, v \in I\} \cup \{\mathcal{R}_w^{\succeq}(v) \,|\, v \in I \wedge \nexists\, \mathcal{R}_w^{=}(v)\} \tag{6}$$

En se basant sur les équations (5) et (6) il est possible d'allouer de nouvelles variables sur l'accélérateur, de générer les opérations de copie entre les anciennes variables et les nouvelles variables et d'effectuer les changements de base nécessaires. La figure 5 illustre cette transformation : on peut y voir l'effet du remplacement de variables, la génération des allocations et des transferts de données 2D.

6. Contraintes mémoires

De nombreux accélérateurs embarqués ont une taille mémoire limitée qui ne peut être ignorée lors de la génération de code. S'il n'y a pas assez d'espace mémoire pour exécuter l'ensemble des calculs en une passe, il faut effectuer le calcul par morceau. C'est indispensable dans le cas de Ter@pix.

Afin de satisfaire ces contraintes mémoire, il est nécessaire de découper l'espace d'itération des noyaux de calculs, ce qui se fait généralement par pavage. Supposons que le noyau de calcul s'exprime sous la forme d'un nid de boucles I parfaitement imbriquées de profondeur n. Afin de connaître les valeurs de pavage, on procède en deux temps : on introduit n valeurs symboliques $p_1, \ldots, p_n$ pour représenter les tailles des blocs de calcul et on effectue un pavage rectangulaire paramétré par ces valeurs, ce qui génère n boucles externes et n boucles internes. On note I_{int} l'instruction portant les boucles internes et σ_{int} l'état mémoire pris avant l'exécution de cette instruction. L'idée est d'exécuter les boucles internes sur l'accélérateur après avoir fixé les p_k de telle sorte que l'empreinte mémoire de ces boucles ne dépasse par une valeur seuil fixée par le matériel. Pour cela on calcule l'empreinte mémoire des boucles internes $V(I_{int}, \sigma_{int})$ et l'on veut trouver une combinaison des $p_1, \ldots, p_n$ qui satisfasse la condition (7), où V_{max} est l'espace mémoire maximum disponible sur l'accélérateur.

$$V(I_{int}, \sigma_{int}) < V_{max} \tag{7}$$

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  int p_1, p_2;
  for(int it = 0; it <= n-1-(p_1-1); it += p_1)
    for(int jt = 0; jt <= m-1-(p_2-1); jt += p_2)
// <in[PHI1][PHI2]-R-MAY-{it<=PHI1, PHI1+1<=it+p_1, PHI1+1<=n,
//  jt<=PHI2+1, PHI2<=jt+p_2, 2jt+1<=PHI2+m, PHI2<=m, 1<=p_2, 0<=m, 0<=n}>
// <out[PHI1][PHI2]-W-MAY-{it<=PHI1, PHI1+1<=it+p_1, PHI1+1<=n,
//   jt<=PHI2, PHI2+1<=jt+p_2, PHI2+1<=m, 0<=m, 0<=n}>
      for(int i = it; i <= MIN(it+p_1, n-1+1)-1; i += 1)
        for(int j = jt; j <= MIN(jt+p_2, m-1+1)-1; j += 1)
          if (j==0) out[i][j] = MIN(in[i][j], in[i][j+1]);
          else if (j==m-1) out[i][j] = MIN(in[i][j-1], in[i][j]);
          else out[i][j] = MIN(in[i][j-1], in[i][j], in[i][j+1]);
}
```

FIGURE 6 – Pavage paramétrique de la boucle externe.

Indirectement, cela fera apparaître des contraintes sur les p_k. D'autres contraintes peuvent peser sur les p_k. Par exemple pour un accélérateur vectoriel, on demande à ce que p_1 soit fixé par la taille de vecteur. La figure 6 illustre le pavage sur notre exemple et montre le résultat de l'analyse des régions de tableau. On peut en déduire l'expression (8) de l'empreinte mémoire de I_{int} en fonction de p_1, p_2.

$$V(I_{int}, \sigma_{int}) = 2 \times p_1 \times p_2 \tag{8}$$

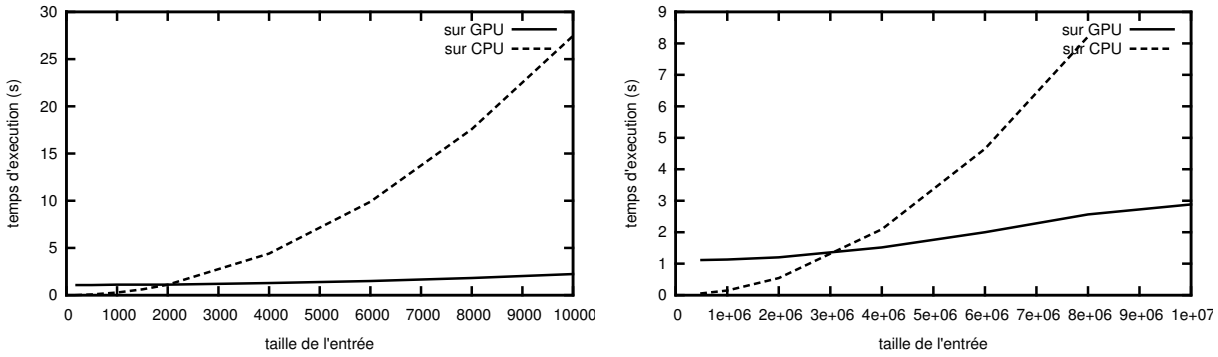
Dans le cas de Ter@pix, on doit fixer $p_1 = 128$, 128 étant le nombre de processeurs de l'accélérateur, et $V_{max} = 1024$ par nœud, ce qui nous donne une expression directe pour $p_2 = \frac{1024 \times 128}{128 \times 2} = 512$.

7. Applications

Les transformations présentées dans les sections précédentes ont été utilisées pour construire avec PIPS deux compilateurs pour deux accélérateurs radicalement différents : un accélérateur à base FPGA dédié au traitement du signal développé par THALES, Ter@pix [3], et les cartes graphiques programmables de NVIDIA. Le premier est un prototype de recherche développé dans le cadre du projet ANR FREIA ; le second est PAR4ALL développé par la jeune pousse HPC Project. Chacun de ces compilateurs utilise les briques de base décrites précédemment pour la partie hôte et utilise des transformations spécifiques pour prendre en compte les autres contraintes au niveau de l'accélérateur.

7.1. Ter@pix

Cet accélérateur est prévu pour fonctionner sur un petit nombre de noyaux de traitement de signal prédéfinis ou sur une combinaison de ces noyaux. La chaîne de compilation pré-existante demandait aux développeurs d'écrire manuellement les noyaux de calcul dans un assembleur spécifique et les appels à ces noyaux en C. Ces noyaux ont été classifiés de la façon suivante : les opérateurs point à point (somme d'images, maximum de deux images etc), les opérateurs avec patron (érosion, dilatation, convolution), les opérateurs avec réduction (puissance moyenne) et avec indirection (histogramme). Un prototype de compilateur automatique pour cette machine, `terapyps`, a été développé en se basant sur le chaînage de l'extraction de noyau, pavage symbolique et génération des communications, ce qui a permis de traiter les deux premières classes d'application. Les opérateurs avec réduction ont été implémentés en parallélisant la réduction pour se ramener à l'un des deux cas précédents. Les noyaux avec indirection ne permettent pas d'obtenir des analyses de régions de tableau suffisamment fines pour qu'elles soient utiles à nos transformations. Notre compilateur génère du code pour l'assembleur cible et pour la gestion des appels de l'hôte. L'essentiel des efforts s'étant concentrés sur le compilateur pour GPU présentés dans la section suivante, nous n'avons pas encore de résultats de simulation au cycle près pour cette cible, malgré des fichiers assembleurs correctement générés.



(a) Convolution (b) FIR

FIGURE 7 – Temps d'exécution sur GPU pour des noyaux de traitements d'images générés par `p4a`.

7.2. GPU

Bien que les approches type CUDA ou OPENCL facilitent la programmation sur GPU, elles sont loin d'être automatisées. Un compilateur C vers CUDA a été développé en utilisant les transformations décrites dans cet article pour la partie hôte, et des transformations spécifiques côté accélérateur qui dépassent le cadre de cet article. Ce compilateur nommé `p4a` est développé dans le cadre de l'initiative Open Source PAR4ALL [16] menée par HPC Project. Il n'utilise pas le pavage symbolique qui ne s'avère généralement pas nécessaire pour les cartes graphiques disposant d'une mémoire globale de grande capacité. Sa validation comprend les noyaux de traitements du signal suivant : une convolution (fenêtre de taille 5) et un filtre à réponse impulsionnelle finie (fenêtre de taille $\frac{n}{1000}$). L'érosion n'apparaît pas car elle est refusée lors de l'estimation d'intensité de calcul **pour cette cible** et donc est exécutée plus efficacement sur l'hôte. Les courbes de performance pour ces noyaux sont données dans la figure 7. Les mesures sont effectuées sur une machine de bureau hébergeant une Debian/testing avec un noyaux Linux 2.6.30-1amd64 et deux Intel Core2 à 2,4 GHz. Le compilateur CUDA 3.2 est utilisé et le code généré s'exécute sur une carte Quadro FX 2700M. Les codes sources sont compilés avec la commande `p4a -cuda` *`input_file.c`* `-o` *`binary`* ; la compilation est complètement automatisée. La version de développement de PAR4ALL est utilisée conjointement à la version de développement de PIPS. Les temps donnés mesurent l'exécution complète du programme : lecture des données, appel du noyau et affichage du résultat, en prenant la médiane sur 100 exécutions. Ces résultats sont encourageants et montrent qu'une approche automatique de la génération de code pour GPU est possible. Les résultats sont encore meilleurs avec une carte plus récente offrant des performances mémoires et calculs mais ceci nous éloigne ici de la gamme de Ter@pix. Le principal avantage étant la compatibilité au niveau source avec le code séquentiel, sans ajout de directives. L'estimation de l'intensité de calcul a permis de valider le comportement asymptotique des fonctions considérées, mais pourrait être raffinée en introduisant un test dynamique permettant de choisir entre l'implémentation séquentielle et l'implémentation parallèle.

8. Conclusion et perspectives

Les contributions de cet article sont la définition de quatre transformations correspondant à autant de contraintes matérielles pesant sur des codes candidats au calcul hybride. L'**estimation de l'intensité de calcul** d'une boucle est utilisée pour prendre en compte le compromis entre temps d'exécution et temps de transfert des données ; l'**extraction de fonction** permet d'isoler une instruction dans une nouvelle fonction ; l'**isolation d'instructions** place les instructions dans un nouvel espace mémoire et le **pavage symbolique** couplé à un **calcul d'empreinte mémoire** permet de prendre en compte les contraintes de mémoire limitée que l'on rencontre sur accélérateur. Ces transformations sont utilisées pour construire deux compilateurs automatiques pour le langage C, `terapyps` et `p4a`, pour un accélérateur à base de FPGA et pour cartes graphiques. Le compilateur `p4a` a été validé sur différents noyaux de traitement du signal et applications scientifiques, démontrant qu'une approche automatique est viable.

Des travaux sont actuellement en cours sur l'optimisation globale des allocations mémoires et des transferts de données qui ont été générées localement. De même, l'utilisation de transferts asynchrones pourrait permettre de masquer une partie des temps de communication. Les interactions entre ces techniques et la prise en compte des contraintes mémoires ou l'estimation du temps de calcul offrent des sujets d'étude intéressants.

Bibliographie

1. Christophe Alias, Alain Darte, et Alexandra Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators an experience with the Altera C2H. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 329–332, 2010.
2. Alexander I. Barvinok. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension Is Fixed. In *Symposium on Foundations of Computer Science (FOCS)*, pages 566–572, 1993.
3. Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gérard Gaillat, Olivier Ruch, et Pascal Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *Design Automation and Test in Europe (DATE'2008)*, pages 610–615. IEEE, 2008.
4. Béatrice Creusillet et Francois Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6) :513–546, 1996.
5. CAPS Entreprise. HMPP Workbench. `http://www.caps-entreprise.com/hmpp.html`.
6. Gildas Genest, Richard Chamberlain, et Robin J. Bruce. Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler : DIME-C. In *Adaptive Hardware and Systems (AHS)*, pages 280–286, 2007.
7. Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 2010.
8. Zhi Guo, Walid Najjar, et Betul Buyukkurt. Efficient hardware code generation for FPGAs. *Transactions on Architecture and Code Optimization (TACO)*, 5(1) :1–26, 2008.
9. François Irigoin, Pierre Jouvelot, et Rémi Triolet. Semantical interprocedural parallelization : an overview of the PIPS project. In *International Conference on Supercomputing (ICS)*, pages 244–251, 1991.
10. François Irigoin, Frédérique Silber-Chaussumier, Ronan Keryell, et Serge Guelton. PIPS Tutorial at PPoPP 2010. `http://pips4u.org/doc/tutorial`.
11. Kamran Karimi, Neil G. Dickson, et Firas Hamze. A Performance Comparison of CUDA and OpenCL. *The Computing Research Repository (CoRR)*, abs/1005.2581, 2010.
12. Volodymyr V. Kindratenko, Robert J. Brunner, et Adam D. Myers. Mitrion-C Application Development on SGI Altix 350/RC100. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 239–250, 2007.
13. Seyong Lee, Seung-Jai Min, et Rudolf Eigenmann. OpenMP to GPGPU : a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.
14. Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, et Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume Lecture Notes in Computer Science (LNCS), Newark, DE, USA, 2009.
15. NVIDIA. *NVIDIA CUDA Reference Manual 3.2*. `http://www.nvidia.com/object/cuda_develop.html`, 2011.
16. HPC Project. Par4All Initiative. `http://www.par4all.org`.
17. Peng Tu et David A. Padua. Automatic Array Privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.
18. Michael Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.
19. Lei Zhou. Complexity Estimation in the PIPS Parallel Programming Environment. In *Conference on Algorithms and Hardware for Parallel Processing (CONPAR)*, pages 845–846, 1992.