

ÉCOLE NORMALE SUPÉRIEURE DE LYON

RAPPORT DE STAGE DE L3

---

**Analyse automatique des transferts  
de données dans l'accélération de  
code à l'aide de GPU**

---

Mines ParisTech  
CRI

Pierre Oechsel  
*encadré par* Claude Tadonki

1 juin 2017 — 1 août 2017

## Résumé

Au cours de mon stage au CRI de juin à juillet 2017 encadré par Mr. Claude Tadonki, j'ai développé un outil permettant l'optimisation de transferts de données en vue de faire de la parallélisation de code automatique via OPENACC. Ce rapport expliquera les étapes entreprises dans la conception d'un tel outil.

## Table des matières

<b>1</b>	<b>Importance des transferts et préliminaires sur notre outil</b>	<b>3</b>
1.1	Présentation de CUDA et OPENACC . . . . .	3
1.2	Importance des temps de transferts . . . . .	4
1.3	Définition précise du problème . . . . .	5
1.4	État de l'art . . . . .	6
<b>2</b>	<b>Détection de zones de tableaux</b>	<b>6</b>
2.1	Introduction à l'analyse statique . . . . .	6
2.2	Bases de l'outil . . . . .	7
2.3	Suivre les déclarations . . . . .	8
2.4	Détection du type d'accès des variables . . . . .	10
2.5	Générer le code en sortie . . . . .	10
<b>3</b>	<b>Gestion des conditions</b>	<b>11</b>
3.1	Notions de contraintes . . . . .	11
3.2	Lien avec l'analyse d'intervalles . . . . .	14
3.2.1	Algorithme . . . . .	14
3.2.2	Remarques et commentaires . . . . .	14
<b>4</b>	<b>Projections</b>	<b>15</b>
4.1	Introduction à la projection . . . . .	15
4.2	Intersection de deux accès . . . . .	16
4.3	Liens avec la littérature . . . . .	17
<b>5</b>	<b>Perspectives future et résultats</b>	<b>18</b>
5.1	Perspectives future . . . . .	18
5.2	Résultats . . . . .	18
<b>A</b>	<b>Contexte du stage</b>	<b>21</b>
<b>B</b>	<b>Code de l'outil</b>	<b>21</b>
<b>C</b>	<b>Exemple de programme en généré</b>	<b>21</b>
<b>D</b>	<b>Calcul matricielle</b>	<b>23</b>
<b>E</b>	<b>Extraits de Polybench</b>	<b>25</b>

# Introduction

Au cours de mon année de L3 à l'ENS de Lyon, j'ai eu l'opportunité d'effectuer mon stage de fin d'année au sein du centre de recherche en Informatique (CRI) des Mines ParisTech sous la direction de Mr Claude Tadonki.

Le sujet de mon stage changea au cours de celui-ci. Au début, il s'agissait de *Reconstruction d'images à grande échelle sur GPU*. Ainsi, lors des premières semaines de mon stage, je me suis renseigné sur le fonctionnement d'un GPU, et sa programmation avec CUDA et OPENACC. J'ai particulièrement veillé à apprendre les différentes possibilités d'optimisation au sein d'un programme CUDA. Rapidement mon encadrant a insisté sur la prépondérance des temps de transferts pour optimiser des programmes hybrides CPU-GPU. C'est ainsi que mon sujet de stage a évolué en *Analyse automatique des transferts de données dans l'accélération de code à l'aide de GPU*. Mon objectif a donc été de créer un outil permettant à partir d'une boucle d'un programme C d'optimiser les transferts de tableaux entre GPU et CPU en détectant les régions de tableaux à transférer ainsi que le sens de transfert (GPU vers CPU ou CPU vers GPU).

Il est à noter qu'au moment de commencer cet outil les notions d'analyse statique m'étaient complètement inconnues. J'ai commencé par une approche personnelle qui consistait à découvrir la problématique avant de me plonger dans la littérature. Néanmoins, au détour de discussions et après avoir élaboré certains algorithmes, j'ai réussi à trouver des antécédents adéquats dans le domaine. Régulièrement mon approche semblait similaire à des notions existantes dans la littérature. Je présenterai donc mon approche du problème en la comparant avec d'autres approches rencontrées plus tard au cours de mon stage.

# Remerciements

Je remercie tout particulièrement M. Claude Tadonki de m'avoir accueilli et m'avoir donné l'opportunité de découvrir la recherche au sein du CRI. Ses apports de connaissances et conseils m'ont été d'une précieuse aide au cours du stage. Je remercie également Mme. Corinne Ancourt pour le temps qu'elle m'a accordé et son accompagnement. Enfin, mes remerciements vont à Mme. Catherine le Caer pour son accueil et son aide logistique ainsi qu'à l'ensemble de l'équipe du CRI pour leur accueil chaleureux.

# 1 Importance des transferts et préliminaires sur notre outil



Dans cette première partie, nous introduisons rapidement la structure d'un GPU, puis à l'aide de l'exemple du calcul matriciel nous montrons l'importance d'optimiser les transferts GPU en comparant différentes implémentations d'un produit matriciel (sur CPU, en CUDA, ou en OPENACC). Cela nous permettra également d'introduire rapidement CUDA et OPENACC. Nous finirons en définissant précisément l'objectif de l'outil et les hypothèses que nous effectuerons.

## 1.1 Présentation de Cuda et OpenACC

Nous n'allons pas entrer dans les détails de l'architecture d'un GPU mais uniquement en faire un résumé ceci n'étant pas notre cadre d'étude principal. Néanmoins, les [refs whitemapers wikipedias] sont une bonne source d'information. Certains livres, notamment [7] permettent d'avoir un bon aperçu de la programmation d'un GPU et des problématiques associées.

L'architecture d'un GPU est fondamentalement différente de celle d'un CPU. Cela peut se comprendre historiquement : les GPUs sont apparus pour accélérer les calculs impliqués dans la génération d'images 3D. Ces opérations étant hautement parallélisables, les GPUs ont suivi le mouvement et se sont différenciés des CPUs. De nos jours, les GPUs sont devenus des accélérateurs facilement programmables, mais cela n'a pas toujours été le cas.

Les GPUs se distinguent des CPUs par leur nombre de core (de l'ordre du millier) et leur aptitude à exécuter en parallèle une même tâche. Ils disposent de leur propre DRAM, mais celle-ci à une haute latence d'accès. Enfin, les données doivent être transférées entre un GPU et un CPU. A l'heure actuelle ces transferts s'effectuent par une liaison PCIe. Le taux de transfert théorique culmine donc à 16 GB/s.

		
Architecture	Fermi	Pascal
Modèle	Tesla C2070	Tesla P100
Date de sortie	2010	2016
Cuda Cores	448	3584
Fréquence	1150Mhz	1126Mhz
Mémoire DRAM	6GB/s	16 GB/s
Bande passante DRAM	144 Gb/s	732 Gb/s

NVIDIA fut le premier constructeur à permettre à l'utilisateur de programmer complètement un GPU à l'aide de la technologie CUDA. Cette technologie offre un grand contrôle sur le GPU, permettant d'agir au plus bas niveau du GPU (transferts entre les différents types de mémoire, synchronisation des threads). Un exemple de code CUDA effectuant un calcul matriciel est donné en annexe D. Cependant, la technologie CUDA, bien que puissante et permettant un grand gain de performances vis-à-vis d'un CPU après

quelques efforts de développement, peut accroître la complexité d'un programme, rendant donc le code moins maintenable. Les gains en performance obtenus sur la partie parallélisable d'un programme sont négligeables par rapport au gain de performance global : si un programme contient 49% de code parallélisable, accélérer ce code d'un facteur 100 ne donnera un gain total en vitesse d'environ deux d'après la loi d'Amdahl). L'effort de porter un algorithme en utilisant CUDA n'est pas tout le temps rentable.

C'est ainsi que des solutions comme OPENACC sont apparues. OPENACC est un standard de programmation (compatible avec les langages C et Fortran) pour le calcul parallèle permettant de paralléliser un algorithme rapidement en insérant des directives dans le programme. Un exemple est donné avec le calcul matriciel en Annexe D. OPENACC cache la complexité de la parallélisation à travers des directives et générera au moment de la compilation la parallélisation. Les directives n'étant que des pragmas, le programme reste compatible avec un CPU (ce qui n'est pas le cas avec CUDA). Il n'y a donc plus besoin de maintenir deux versions différentes d'un même programme.

## 1.2 Importance des temps de transferts

Dans l'annexe D, nous présentons trois versions d'une multiplication matricielle : deux parallélisées, une avec CUDA, l'autre avec OPENACC, et une témoin fonctionnant sur GPU. Comme nous pouvons le voir les temps de transfert sont non négligeables par rapport au temps de fonctionnement des algorithmes. Ils peuvent même être un facteur limitant lors de la parallélisation d'un programme. Bien qu'il soit parfois possible de transférer les données de manière asynchrone ce n'est pas toujours possible.



FIGURE 1: Le transfert n'est pas asynchrone.

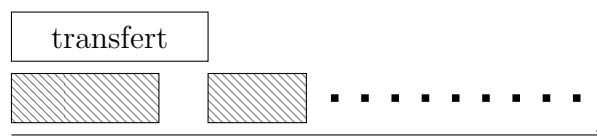


FIGURE 2: Le transfert est effectué de manière asynchrone.

Dans ces cas là, nous aimerions pouvoir disposer d'un outil se chargeant automatiquement de l'optimisation des transferts dans une partie de code parallélisable. Au cours de mon stage nous nous sommes basés sur l'implémentation d'OPENACC présente dans le compilateur PGI. N'ayant jamais réussi à faire fonctionner celle de GCC, nous n'avons pas pu effectuer de comparaison entre les deux.

PGI approxime déjà les parties de tableaux à transférer. Cette approximation est souvent large, voir peut créer des erreurs. Ci-dessous `copy` représente un transfert du GPU vers le CPU et du CPU vers le GPU. `copyin` est un transfert uniquement dans le sens CPU vers GPU.

```
#pragma acc kernels
for (int i = 0; i < 500; ++i) {
    if (i < 250)
        a[i] = 0;
}
```

```
>> 25, Generating implicit copy(a[:500])
```

Dans l'exemple ci-dessus, PGI détecte un accès en lecture et en écriture alors qu'il n'est qu'en écriture. De plus, il transfère tout le tableau alors que nous n'en utilisons que la moitié.

```
#pragma acc kernels
for (int i = 0; i < 500; ++i) {
    a[2*i] = 0;
}
```

```
>> 25, Generating implicit copy(a[:999])
```

PGI envoie tous le tableau même si cela n'était pas nécessaire car nous sommions les index pairs.

```
#pragma acc parallel loop
for (int i = 0; i < 9999; ++i) {
    if (i>=2)
        b = a[i-2];
}
```

```
>> 25, Generating implicit copyin(a[-2:9999])
```

PGI transfère les premiers éléments de *a* à partir de l'indice  $-2$ . Cela conduit logiquement à une erreur d'accès mémoire (segfaults).

### 1.3 Définition précise du problème

Nous souhaitons résoudre les différents problèmes que rencontre OPENACC ci-dessus. L'outil devra donc pouvoir, à partir de l'extrait d'un programme en C qui est censé être parallélisable (par exemple une boucle), détecter :

- Les accès aux variables (lecture, écriture ou lecture et écriture)
- Les régions des tableaux accédées ainsi que leur type d'accès
- Les projections dans les tableaux (transformer  $A[2*i]$  en  $B[i]$ )

Comme nous travaillons sur des parties de programme supposées être parallélisables, nous pouvons supposer qu'il n'y a pas d'aliasing (c'est à dire qu'aucune variable ne pointe vers la même zone mémoire qu'une autre). Nous supposons de plus que les fonctions appelées sont sans effets de bords.

Pour que les termes soient clairs dans la suite du rapport, nous définissons :

**Définition 1.1.** Régions Une région  $E$  d'un tableau  $n$ -dimensionnel  $A$  correspond à la donnée de  $E_1, \dots, E_n$ ,  $n$  sous ensembles de  $\mathbb{Z}$ , tel que  $E = E_1 * \dots * E_n$ .  $E$  représente les éléments  $A[i_1][\dots][i_n]$  pour tous  $i_1 \in E_1, \dots, i_n \in E_n$ .

**Définition 1.2.** Itérateur Un itérateur est une variable  $i$  parcourant un intervalle  $[i_l; i_h]$  avec un pas fixe.

## 1.4 État de l’art

Nous effectuons un rapide état de l’art de programmes générant des transferts automatiquement.

- OPENACC, dont nous avons déjà parlé, ne prévoit pas de générer automatiquement les transferts dans sa norme [2]. L’implémentation de PGI le fait, mais de façon non optimale dans certains cas.
- CGCM [5] est un outil développé par l’université de Princeton. Il génère automatiquement les transferts d’un programme parallélisable. Son approche est différente de la nôtre : l’outil utilise une combinaison d’inférence de type, d’analyse d’aliasing et de redéfinition de fonctions C de gestion mémoire (malloc, free) pour arriver à sa fin.
- PIPS [3], est un projet développé par le CRI d’analyse de programmes ayant débuté il y a plusieurs dizaines d’années sur lequel se base PAR4ALL. Il peut (entre autre) détecter les régions des tableaux accédées.

## 2 Détection de zones de tableaux

Dans cette première partie, nous décrivons le fonctionnement de la première version de l’outil. Nous expliquons le choix des technologies utilisées ainsi que le processus suivi pour obtenir un programme capable de générer des transferts de tableaux sans en gérer les conditions.

### 2.1 Introduction à l’analyse statique

Il arrive que l’on veuille obtenir des informations à propos d’un programme. Par exemple savoir si une variable peut être nulle, si une variable reste constante où encore dans quels intervalles évoluent les variables. Malheureusement, dû au lien fort avec le problème de l’arrêt, il n’est pas possible de déterminer les réponses précises à ces questions dans un cas général. L’analyse statique consiste donc à obtenir le maximum d’informa-

```
x = 17;
if (Machine fini (machine))
    x = 18;
```

FIGURE 3: Lien entre analyse statique (savoir si x est constante) et problème de l’arrêt

tion à une question. Pour cela nous effectuons des approximations. Généralement ces approximations sont dites *conservatives* : il n’y a pas de faux positifs. Dans l’exemple de la détection d’un pointeur nul nous pouvons ne pas détecter certains pointeurs nuls, mais tous les pointeurs détectés le sont. Le but est donc d’avoir les réponses les plus précises possible.

On distingue deux types d’analyses :

- Les *must*, où toutes les informations détectées doivent être vrais (on calcule une sous approximation).
- Les *may*, où les informations détectées peuvent être vrais (on calcule une sur approximation).

Nous voulons ici détecter les accès de façon *must* pour en envoyer le moins possible sur le GPU. En effet, comme nous cherchons à améliorer la détection que fait PGI, nous pouvons

nous permettre de ne pas détecter certains accès : PGI les détectera probablement même si non optimisés.

Enfin, nous effectuerons des analyses dites *intraprocédurale*. Nous analysons le code passé en argument sans connaître le reste du programme. Nous ne possédons donc aucune information sur les valeurs et types des variables non définies dans le bloc de code passé en argument ainsi que sur les fonctions annexes. Les analyses prenant en compte toutes les fonctions d'un coup sont dites *interprocédurale*.

## 2.2 Bases de l'outil

J'ai choisi le langage OCaml pour écrire l'outil, ce langage étant très pratique pour manipuler du code source.

**Parsing** La première étape du stage fut d'écrire un parser C. Celui-ci est écrit en suivant la norme C99 [1] à l'aide de yacc et lex. Nous obtenons en sortie de cette étape de parsing un AST (Abstract Syntax Tree). Il s'agit d'une structure de données représentant notre code source d'une manière aisément manipulable. La principale difficulté est qu'il a fallu parser les pragmas et les garder dans l'AST car elles peuvent être importantes (pour paralléliser avec OPENACC par exemple).

**Forme des expressions traitées** Plusieurs expressions dans le code vont nous être utiles. Il s'agit des indices d'accès aux tableaux, des conditions, ou encore des bornes des itérateurs. Dans les lignes ci-dessous, le symbole # désigne symboliquement différentes positions qui peuvent avoir ces expressions.

```
if (# < #)
for (int i = #; # < #; ++i)
A[#][#] = foo + B[#]
```

Nous avons choisi de nous concentrer sur des expressions de la forme :

$$\sum_{k=1}^n a_k * i_k + b \quad (1)$$

Les  $a_i$  peuvent être quelconque mais ne doivent pas dépendre de  $i_k$ . Les  $i_k$  sont des itérateurs (autrement dit des indices de boucles). Ces indices de boucles sont définis heuristiquement : une variable  $v$  est un itérateur si elle est présente dans une instruction de la forme `for (int v = expr; v < expr; ++v)` (ou tout autre équivalent). Le plus grand avantage de ces expressions est qu'étant donné des intervalles  $[[l_k, h_k]]$  encadrant les  $i_k$ , nous pouvons facilement déterminer leurs extrema. L'autre avantage est que ces expressions sont relativement courantes.

**Calcul formel :** Nous effectuons une passe de calcul formel pour vérifier qu'une expression est de la forme attendue. Cette passe sert également à retrouver les  $a_k$  et les  $b$  de chaque expression.

**Les soucis du calcul symbolique** L'analyse que nous effectuons étant intraprocédurale, nous ne connaissons pas la valeur de plusieurs variables. Il faut donc que nous restions en calcul symbolique tout au long de notre analyse, ce qui complique certaines opérations. Tous calcul et comparaison doivent être reportés dans le code généré : cela sera expliqué plus clairement plus tard.



## 2.3 Suivre les déclarations

Rapidement nous avons du faire face à une première difficulté. Les indexations ne se font pas toujours idéalement, les cas suivant sont possibles :

```
for (int i = 0; i < N; ++i) {
    int k = i;
    A[k] = ...;
}

for (int i = 0; i < N; ++i) {
    int width = W * i;
    for (int j = 0; j < N; ++j) {
        A[width + j] = ...;
    }
}
```

Dans les deux exemples, les indices `k` et `width + j` ne sont pas suffisamment explicites. Aucun des deux n'impliquent tous les itérateurs (`i` dans le premier cas, `i` et `j` dans le second). Pour résoudre ce problème, nous avons dû créer un algorithme permettant de suivre l'historique des déclarations. Il remplace les variables par les calculs ayant mené à ces valeurs. Par exemple, le programme à gauche est transformé en celui à droite.

```
int a = N + 3;
int b = a++ * 4;
int c = a + sqrt(b);

int a = N + 3;
int b = (N + 3) * 4;
int c = (N+3+1) + sqrt( (N+3) * 4 );
```

Cet algorithme est donné ci-dessous. L'algorithme consiste en une transformation de l'AST. Nous supposons que nous avons accès à une structure de données `Map` permettant d'associer à un identifiant une partie d'AST (typiquement représentant une expression arithmétique). Nous supposons également que nous possédons trois méthodes, `add`, `get` et `join`, dont le fonctionnement est décrit ci-dessous :

- `add(E, id, ast)` ajoute l'association entre `id` et `ast` dans l'environnement `E`
- `get(E, id)` retourne l'affectation de `id` si elle est présente dans `E`, sinon `id`
- `join(E1, E2)` va renvoyer un environnement présentant la fusion de `E1` et `E2`. Si un identifiant est présent dans les deux environnements, mais que les deux ASTs associés sont différents, alors il ne sera pas présent en sortie. Sinon il l'est.

A partir de là, l'algorithme final se présente sous la forme :

---

**Algorithme 1** Pseudocode de l'algorithme pour récupérer l'historique des déclarations

---

```
1: procedure DECLARATIONSHISTORY(E, ast)
2:   match ast
3:     with Identifiant(id) :
4:       return (E, get(E, id))
5:     with Operateur(left, right) :
6:       E', left' = DeclarationsExpand(E, left)
7:       E'', right' = DeclarationsExpand(E', right)
8:       return (E'', Operateur(left', right'))
9:     with IfThenElse(condition, if, else) :
10:      E', condition' = DeclarationsExpand(E, condition)
11:      E1, if' = DeclarationsExpand(E', if)
12:      E2, else' = DeclarationsExpand(E', else)
13:      return (join(E1, E2), IfThenElse(condition', if', else'))
14:     with While(condition, content) :
15:      E', condition' = DeclarationsExpand(E, condition)
16:      E'', content' = DeclarationsExpand(E', content)
17:      return (join(E, E''), While(condition', content'))
18:     with Affectation(id, expr) :
19:      E', expr' = DeclarationsExpand(E, expr)
20:      return (add(E', id, expr'), Affectation(id, expr'))
21:   end match
22: end procedure
```

---

Une attention particulière a du être prêtée aux post-incrémentations et pré-incrémentations (de même pour les décrémentations). Le comportement d'expression semblable à `++a * a++` ou encore `A[++i] = i++` étant non défini en C [1] nous avons regardé le comportement de telles expressions sur différents compilateurs C. Les résultats sous **GCC** et **Clang** étant différents pour de telles expressions (le comportement de **GCC** est même différent selon que `a` soit déclaré **volatile** ou pas, ce qui n'est pas le cas pour **Clang**), nous avons donc choisi de suivre le comportement de **Clang**. Ainsi, une expression `++a` ajoutera l'historique de `a+1` à l'identifiant `a` dans l'environnement, puis retournera l'historique de `a+1`. L'expression `a++` retournera juste l'historique de `a`, puis ajoutera l'historique de `a+1` à l'environnement.

**Remarque :** l'AST généré est utilisé uniquement pour récupérer les expressions qui nous intéressent. Si une de ces expressions comporte une variable déclarée dans notre procédure, alors nous n'avons pas pu avoir accès à un historique clair (cela peut arriver à cause d'une divergence causée par un `if`). Dans ce cas, nous renvoyons une erreur. Par exemple, dans le programme suivant, `a` change de valeur suivant la condition suivie. Lors du déroulement de l'historique, la valeur de `b` est donc `a`, qui est une variable définie au sein de la procédure.

```
for (int i = 0; i < N; ++i) {
    int a = 0;
    if (i < 4)
        a = 6;
```

```

    b = a;
}

```

**Limitations de cet algorithme** Cet algorithme présente une certaine faiblesse. Comme nous effectuons des comparaisons entre ASTs, après les instructions suivantes, `c` est non défini dans l’environnement bien qu’ayant la même valeur.

```

if (...) c = a+b; else c = b+a;

```

Plus généralement, cet algorithme est sensible aux divergences. La présence d’une boucle ou d’une condition au sein de laquelle une variable peut être modifiée peut compromettre son fonctionnement.

**Synthèse** Cette étape nous permet de récupérer de manière explicite l’indice d’accès aux tableaux ainsi que d’autres expressions utiles. Son but est d’enlever les variables intermédiaires pour ne garder que les dépendances aux variables hors du bloc de code accessible et aux itérateurs.

## 2.4 Détection du type d’accès des variables

Le point central de notre travail était de détecter les variables accédées en lecture, en écriture ou en lecture et écriture. L’algorithme résultant construit une liste de toutes les variables écrites ou lues (avec les indices d’accès correspondant). Nous ne conserverons de cette liste que les variables déclarées en dehors du code entré par l’utilisateur (donc les variables à transférer). L’algorithme en lui même est simple et ne mérite pas beaucoup d’explications. Une variable dont la valeur est modifiée et ajoutée (avec ses indices d’accès et les itérateurs associés) comme étant en écriture. Si elle est lue, elle est ajoutée en écriture. Dans le cas d’une incrémentation ou décrementation, elle est ajoutée en lecture et en écriture. Si une même variable est présente deux fois avec le même type d’accès, les mêmes indices d’accès et les mêmes itérateurs, alors nous supprimons l’une des deux occurrences.

## 2.5 Générer le code en sortie

Récapitulons : nous savons quelles sont les variables accédées en lecture, écriture ou en lecture et écriture. Nous savons également par rapport à quels indices ces accès se font. Il reste alors deux étapes : obtenir les intervalles accédés et générer les transferts associés.

**Envoyer des enveloppes convexes** Nous envoyons sur le GPU les enveloppes convexes des accès. Si un tableau est accédé en écriture entre les indices `[0;100]` et `[600;700]`, nous allons envoyer le tableau entre les indices `[0;700]`. Nous perdons en optimisations, mais en contrepartie cela facilite la gestion des tableaux multidimensionnels. Prenons le programme suivant. La géométrie du tableau à transférer plus délicate à déterminer.

```

for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        A[i * 100 + j][j] = ...;
    }
}

```

**Détermination des régions à transférer** Connaissant les intervalles  $[l_k; h_k]$  dans lesquels les itérateurs  $i_k$  sont inclus, il est aisé de calculer l'intervalle des accès d'un tableau de la forme 1. L'intervalle correspondant est :

$$\left[ \sum_{k=1}^n \min(a_k * l_k, a_k * h_k) + b; \sum_{k=1}^n \min(a_k * l_k, a_k * h_k) + b \right] \quad (2)$$

Pour les tableaux multi-dimensionnels nous utilisons le fait que l'enveloppe convexe de deux intervalles est le produit des deux intervalles.

**Chevauchement de zones en écriture / lecture** Il peut arriver qu'un même tableau ait deux zones, une écriture, l'autre en lecture) se chevauchant. Dans ce cas nous fusionnons les deux zones pour n'en former qu'une en lecture et écriture. Plus précisément le non chevauchement des zones en lecture et en écriture signifie qu'il n'y a pas de dépendances faisant obstacle à la parallélisation. L'inverse n'est pas vrai, déterminer les dépendances en lecture et écriture est un sujet délicat [6].

**Code généré :** Le programme génère beaucoup d'instructions C pour effectuer les transferts. Une partie est destinée à calculer dans quels intervalles se situent les itérateurs, une autre à déterminer quelles sont les régions de tableaux à transférer, et enfin une dernière partie pour générer les transferts. Nous donnons en annexe C un exemple de code généré commenté.

**Conclusion** Une fois que nous obtenons les enveloppes convexes des accès aux différentes zones d'un tableau, nous pouvons générer ces transferts. Nous effectuons une disjonction de cas en fonction des zones. Si les zones en lecture et écriture sont disjointes, nous sommes assurés que le parallélisme peut se faire et nous effectuons un transfert GPU vers CPU et un transfert CPU vers GPU. Sinon, nous transférons uniquement une zone de tableau en écriture et lecture. Le parallélisme n'est alors pas assuré.

## 3 Gestion des conditions

Une fois la première version de notre outil fini, nous avons cherché à améliorer notre outil pour gérer les conditions et donc gérer les programmes suivants (A est en écriture sur  $[0; N/2]$  et en lecture ailleurs) :

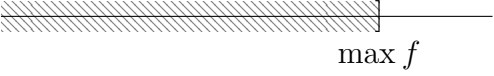
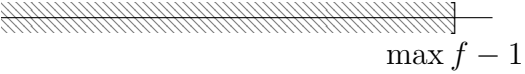
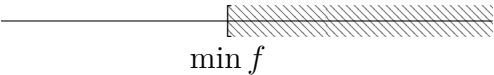
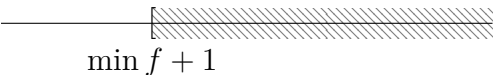
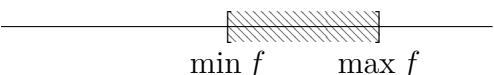
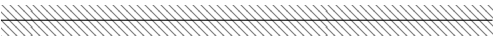
```
for (int i = 0; i < N; ++i) {
    if (i < N/2) {
        A[i] = ...;
    } else {
        foo(A[i]);
    }
}
```

### 3.1 Notions de contraintes

Jusqu'à présent, les intervalles dans lesquels évoluent les itérateurs correspondent à leurs intervalles de définition. Nous cherchons à restreindre ces intervalles en s'aidant des conditions présentes dans le code.

Nous introduisons la définition de **contrainte simple**.

**Définition 3.1.** Contrainte simple Étant donné une fonction  $f$  quelconque à  $n$  arguments, une variable entière  $i$  et un opérateur  $op$  appartenant à :  $\{\leq, \geq, =, \neq, <, >\}$ , nous associons à l'expression  $e = i \text{ op } f(x)$  un intervalle, noté  $Cs(e)$  parmi les suivants :

- Si  $op$  est  $\leq$ ,  $Cs(e) = [-\infty; \max f]$  
- Si  $op$  est  $<$ ,  $Cs(e) = [-\infty; \max f - 1]$  
- Si  $op$  est  $\geq$ ,  $Cs(e) = [\min f; +\infty]$  
- Si  $op$  est  $>$ ,  $Cs(e) = [\min f + 1; +\infty]$  
- Si  $op$  est  $=$ ,  $Cs(e) = [\min f; \max f]$  
- Si  $op$  est  $\neq$ ,  $Cs(e) = [-\infty; +\infty]$  

En pratique, nous nous intéresserons aux **Contrainte simple** provenant d'expression de la forme :

$$\sum_{k=1}^n a_k * i_k + b \text{ op } 0$$

D'une telle expression se déduisent  $n$  contraintes simples, il s'agit des (pour  $l \in [1, \dots, n]$ ) :

$$i_l \text{ op }', \frac{\sum_{k=1, k \neq l}^n a_k * i_k + b}{-a_l}$$

$op'$  correspond à l'opérateur  $op$  si  $-a_l > 0$ , sinon à l'opérateur conjugué.

### Notes sur cette définition

- La définition prend également en compte les expressions comme  $i + 2j < n$  (où  $i$  et  $j$  sont des itérateurs). Les opérations  $\min$  et  $\max$  sont justement utiles dans le cas où plusieurs itérateurs entrent en jeu. Dans ce cas précis, deux contraintes sont générées :  $i < n + \max(-2h_j, -2l_j)$  et  $j < \frac{n + \max(-h_j, -l_j)}{2}$ .
- Le cas des expressions de la forme  $i \neq j$  s'expliquent en supposant que  $i$  et  $j$  sont deux itérateurs. Prenons le programme suivant :

```

for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        if (i != j)
            A[i] = ...;
    }
}

```

Dans ce programme l'élément qui n'est pas accédé dans **A** change au fil des itérations. Une contrainte d'exclusion aurait mené à une erreur. Certes l'intervalle n'est pas le plus précis possible dans le cas où  $i$  (ou  $j$ ) est une constante. Mais dans

ce cas le tableau est coupé en deux et il faudrait deux appels pour le transférer sur le GPU (ou vers le CPU) en admettant que cela soit faisable. Le surcout d'ouverture d'une communication (2ms) pour économiser l'envoi d'un faible nombre d'éléments fait que cette opération n'est pas rentable.

Nous définissons également les **contraintes composées** :

**Définition 3.2.** Contrainte composée A toute opération booléenne  $e$  nous associons une contrainte composée (noté  $Cc(e)$ ) comme suivant :

- Si  $e = a \vee b$ ,  $Cc(e) = Cc(a) \cup Cc(b)$
- Si  $e = a \wedge b$ ,  $Cc(e) = Cc(a) \cap Cc(b)$
- Sinon  $Cc(e) = Cs(e)$

Enfin, nous appelons **contrainte** l'intersection de l'ensemble des contraintes composées agissant sur un itérateur.

Les régions accédées dans un tableau sont dorénavant déterminées à l'aide de ces contraintes. Par ailleurs, ces contraintes permettent de généraliser et de traiter l'initialisation de nos itérateurs de la même façon que l'on traite les conditions : à l'instruction `for (int i = 0; i < f(x); ++i)` correspond la contrainte composée  $[0; 0] \cap [-\infty; \max f + 1]$ .

Nous noterons également qu'une condition génère deux contraintes par itérateurs impliqués. Nous donnons un exemple ci-dessous :

```
if (i < 8 && j >= n/2) {
    // contrainte sur i: [-infini; 7]
    // contrainte sur j: [n/2; +infini]

} else {
    // contrainte sur i: [8; +infini]
    // contrainte sur j: [-infini; n/2-1]
}
```

**Génération de code** Les calculs devant être délégués lors de l'exécution du programme car nous ne connaissons pas la valeur de toutes les constantes, nous générons explicitement les opérations d'insertion, d'union ainsi que de génération des contraintes simples en sortie.

**Lien avec la littérature** Cette idée est très semblable avec le *domaine relationnel polyédrique* [4]. Nous entendons par *domaine* une abstraction de ce qui se passe dans le programme que l'on étudie. Il s'agit souvent d'une approximation afin que cela soit calculable. *Domaine relationnel* sous - entend que le domaine permet d'exprimer des relations entre les différentes variables. Par exemple qu'une variable  $x$  est de signe opposé à une variable  $y$ . Enfin ce domaine est dit *polyédral* : il contient des polyèdres convexes et fermés représentant les contraintes entre différentes variables. En notant  $x = (x_1, \dots, x_n)$   $n$  variables, un polyèdre convexe fermé correspond à une équation du type  $Ax \leq b$  (avec  $a$  une matrice et  $b$  un vecteur colonne). Ce domaine, bien que puissant, présente des performances faibles quand le nombre de variable devient grand. Plusieurs opérations élémentaires sur les polyèdres sont requises mais ces opérations peuvent avoir dans le pire des cas une complexité exponentielle en le nombre de variables [10].

Il existe un domaine intervalle. Moins précis que le *domaine relationnel polyédrique*, il est cependant aisé à manipuler. Nous en parlons plus en détail dans le point suivant.

## 3.2 Lien avec l'analyse d'intervalles

Vers la fin de mon stage, j'ai découvert l'existence d'algorithmes d'analyse d'intervalles [9]. Leur but, semblable à celui que nous avons eu avec les itérateurs, est de déterminer un intervalle dans lequel se situe chaque variable. Nous allons rapidement présenter l'approche utilisée dans ces algorithmes puis nous examinerons l'utilité de cet algorithme dans notre outil.

### 3.2.1 Algorithme

Le but de cet algorithme est de déterminer un intervalle  $[l_v, h_v]$  pour toutes variables  $v$  de tel sorte que  $v$  soit dans cet intervalle. Nous nous plaçons pour cela dans un treillis  $L$  contenant l'ensemble des intervalles entiers avec la relation d'inclusion. Ce treillis n'est pas de hauteur finie. Nous utilisons ensuite un algorithme *dataflow*. Ce type d'algorithme est expliqué clairement dans [9].

Au début du programme, nous affectons à toute variable l'intervalle  $[-\infty; +\infty]$ . Nous disposons d'opération permettant de calculer l'intervalle correspondant au produit de deux valeurs (mais aussi à l'addition, la soustraction, l'inverse). Malheureusement, comme le treillis n'est pas de hauteur finie, la convergence de l'algorithme n'est pas assurée, notamment dans le cas de boucles :

```
int y = 0;
while (true) {
    y = y+1;
}
```

Pour contrecarrer cet effet, nous allons nous placer dans un treillis  $L'$  à hauteur finie. Ce treillis contient  $-\infty$  et  $+\infty$  ainsi que, par exemple, les intervalles dont les bornes sont des entiers présents dans le programme.

Nous commençons par définir une opération de **widening**  $w$  qui va surestimer l'approximation des intervalles. Cette opération est effectuée sur chaque approximation de la fonction  $F$  dont nous voulons trouver le point fixe. Une opération de **narrowing** suivra une fois que la convergence est obtenue pour sous approximer notre sur-approximation, et ainsi se rapprocher de l'intervalle optimal.

### 3.2.2 Remarques et commentaires

Cet algorithme pourrait avantageusement remplacer l'étape déroulant l'historique des variables. Il permettrait de gérer des cas plus compliqués, par exemple les programmes suivants :

```
for (i=0; i < N; ++i) {
    if (...) {
        a = 3;
    } else {
        a = 2;
    }
    A[a * i] = ...;
}

for (i=0; i < N; ++i) {
    while (...) {
        a ++;
    }
    A[a * i] = ...;
}

for (i=0; i < N; ++i) {
    if (...) {
        a = b+c;
    } else {
        a = c+b;
    }
    A[a * i] = ...;
}
```

Le dernier exemple n'est pas compatible avec l'algorithme déroulant l'historique des variables. Comme  $c+b$  n'est pas représenté par le même nœud que l'AST, l'algorithme n'arrive pas à dérouler l'historique de  $a$  en dehors des conditions.

Enfin, avoir non plus des valeurs exactes pour les  $a_i$  mais des intervalles dans l'équation 1 ne complique pas substantiellement le calcul de l'approximation des accès.

Malgré ces raisons positives, nous n'avons pas implémenté cet algorithme pour plusieurs raisons :

- Cela complique la gestion des appels de fonctions simples (par exemple de  $\cos$ ,  $\sin$ ,  $\sqrt{\phantom{x}}$ , ou tout autre fonction mathématique). Trouver le maximum et le minimum d'une fonction quelconque dans un intervalle n'est pas une tâche aisée.
- Nous n'avons pas vu comment implémenter les étapes de widening et narrowing qui utilisent traditionnellement des comparaisons. L'analyse étant intraprocédurale, nous ne possédons pas d'information sur les paramètres des régions et des conditions.
- Les problématiques associées à ces algorithmes dépassent la portée de mon stage.

## 4 Projections

La projection était une des pistes d'optimisation que nous voulions mener. Au cours de cette partie nous expliquerons notre approche à cette question.

### 4.1 Introduction à la projection

Mr. Tadonki m'a suggéré au début de mon stage de gérer les projections au sein des tableaux. Nous donnons ci-dessous des exemples de projections pour comprendre ce dont il s'agit.

Avant projection	Après projection
<pre>for (int i = 0; i &lt; N; ++i)   A[3*i] = ...;</pre>	<pre>for (int i = 0; i &lt; N; ++i)   B[i] = ...;</pre>
<pre>for (int i = 0; i &lt; N; i += 3)   A[i] = ...;</pre>	<pre>for (int j = 0; j &lt; N/3; ++j)   B[j] = ...;</pre>

Effectuer ces projections nous permet d'économiser de la mémoire dans les tableaux à transférer, donc de raccourcir les temps de transferts.

Plus concrètement, une projection est une opération telle que :

**Définition 4.1.** Projection Soit  $i_1, \dots, i_m$   $m$  itérateurs et  $E_1, \dots, E_m$  leurs intervalles de définitions. La projection d'un tableau  $A$   $m$ -dimensionnel accédé aux indices  $f(i_1), \dots, i_m$  est un tableau  $B$  accédé aux indices  $i_1, \dots, i_m$  et pour lequel :

$$\forall i_1 \in E_1, \dots, \forall i_m \in E_m, A[f(i_1)] \cdots [f(i_m)] = B[i_1] \cdots [i_m]$$



**Restriction de notre étude** La définition précédente est très large. Comme dans le reste de nos réflexions, nous nous limitons à des fonctions  $f_i$  et  $g_i$  linéaire, c'est à dire de la forme  $ai + b$ . Nous n'avons pas traité le cas où les tableaux ont plus d'une dimension.

**Détermination des zones projetables** Il reste encore à savoir quelles sont les régions que nous pouvons projeter. Si deux régions du tableau avec des types d'accès différents ne sont pas accédées exactement aux mêmes indices, alors il ne peut pas y avoir de projections de ces zones. Si deux régions d'un tableau de même type d'accès ont des accès aux mêmes indices, alors il n'y a pas non plus de projection possible. Nous illustrons ci-dessous les différents cas. Nous donnons uniquement le corps de la boucle. Toutes les boucles sont ici de la forme : `for (int i = 0; i < N; ++i) { ... }`.

Corps de la boucle	Commentaire
<pre>A[2 * i] = ...; A[3 * i] = ....;</pre>	Nous ne pouvons pas effectuer de projection de A car les ensembles $\{2i, i \in [0, N]\}$ et $\{3i, i \in [0, N]\}$ ont des éléments en commun.
<pre>A[2*i] = ...; s += A[3*i];</pre>	Nous ne pouvons pas effectuer de projection de A car les ensembles $\{2i, i \in [0, N]\}$ et $\{3i, i \in [0, N]\}$ ont des éléments en commun.
<pre>A[2*i] = ...; s += A[2*i];</pre>	Nous pouvons effectuer une projection de A car les accès sont identiques.

**Remarque** Nous pouvons gérer finement les itérateurs n'itérant pas par pas de 1. Un itérateur  $i$  qui itère par pas de 2 est équivalent à un itérateur  $j$  itérant par pas de 1 mais utilisé avec l'expression  $2j$ . Ainsi, si un accès se fait pour  $i$  de la forme  $ai + b$ , il se fait pour  $j$  avec la forme  $2aj + b$ . Il faut néanmoins faire attentions aux bornes de l'intervalle dans lequel  $j$  est défini.

## 4.2 Intersection de deux accès

Dans cette sous partie nous expliquons comment déterminer si deux accès à un tableau sont disjoints. Supposons qu'un tableau A soit accédé aux indices  $\sum_{i=1}^n a_i x_i + b$  et aux indices  $\sum_{i=1}^{n'} a'_i x'_i + b'$ . Les  $x_i$  et  $x'_i$  sont des itérateurs. Nous souhaitons savoir si l'équation suivante possède une solution :

$$\sum_{i=1}^n a_i x_i + b = \sum_{i=1}^{n'} a'_i x'_i + b'$$

Nous pouvons remarquer que nous cherchons donc les solutions des équations de la forme (avec  $x_i$  des inconnues) :

$$\sum_{i=1}^n a_i x_i = b \tag{3}$$

**Remarque :** Lorsque nous voulons savoir si deux accès de la forme  $ai + b$  et  $a'i + b'$  (avec  $i \in [l; m]$ ) s'intersectent, nous voulons en fait résoudre l'équation  $ai + b = a'j + b'$  où  $i \in [l; m]$  et  $j \in [l; m]$ . La création d'un indice implicite  $j$  est nécessaire. Sans ce nouvel indice, nous regardons si les deux accès se font au même endroit lors de la même itération.

**Théorème 1** (Existence d'une solution). *Une équation linéaire de la forme 3 possède une solution si et seulement si  $\text{pgcd}(a_1, \dots, a_n) \mid b$*

**Théorème 2** (Ensemble des solutions d'une équation linéaire où  $n = 2$ ). *Étant donné une solution  $(r, s)$  de l'équation  $a_1x_1 + a_2x_2 = b$ , l'ensemble des solutions est  $\{(r + k\frac{a_2}{d}, s - k\frac{a_1}{d}), k \in \mathbb{N}\}$  avec  $d = \text{pgcd}(a_1, a_2)$*

Effectuer uniquement le test de divisibilité nous permet d'éliminer les cas aberrants où l'intersection ne se fait pas. Malheureusement, ce test seul ne suffit pas à tester l'intersection entre deux accès de la forme  $ai + b$  avec  $i$  dans un intervalle  $[l_i, h_i]$ . Pour prendre en compte l'intervalle dans lequel se situe les itérateurs nous utilisons le théorème précédent. Nous commençons par trouver une solution particulière à l'aide de l'algorithme d'Euclide étendu, puis nous cherchons à savoir s'il existe un  $k$  tel que les solutions soient dans les bons intervalles. L'algorithme d'intersection final est donné ci-dessous :

---

**Algorithme 2** Pseudocode de l'algorithme d'intersection entre deux accès

---

```

1: procedure INTERSECTIONSACCÈS( $a_1, a_2, b, (l_i, h_i), (l_j, h_j)$ )
2:    $d = \text{pgcd}(a_1, a_2)$ 
3:   if  $d \div b$  then
4:     else
5:        $(i_0, j_0) = \text{solution particulière}$ 
6:        $k'_{\min} = \text{valeur de } k \text{ minimal pour que } l_i \leq i_0 + \frac{a_2}{b}k$ 
7:        $k'_{\max} = \text{valeur de } k \text{ maximal pour que } i_0 + \frac{a_2}{b}k \leq h_i$ 
8:        $k''_{\min} = \text{valeur de } k \text{ minimal pour que } l_j \leq j_0 - \frac{a_2}{b}k$ 
9:        $k''_{\max} = \text{valeur de } k \text{ maximal pour que } j_0 - \frac{a_2}{b}k \leq h_j$ 
10:      if  $[k'_{\min}; k'_{\max}] \cap [k''_{\min}; k''_{\max}] = \emptyset$  then
11:        return False
12:      else
13:        return True
14:      end if
15:    return False
16:  end if
17: end procedure

```

---

### 4.3 Liens avec la littérature

Des tests d'intersection semblables existent dans la littérature. Ils sont même la pièce centrale de la détection de dépendances pour la parallélisation automatique de code [6]. En particulier, deux tests ressemblent à notre travail. Le premier est le test de divisibilité par le pgcd. Le second est une amélioration de notre test pour prendre en compte les intervalles dans lesquels sont contenus les itérateurs. Il s'agit du test de Banerjee-Wolfe [8]. Celui-ci va chercher à calculer les valeurs minimales (et maximales) que peut prendre l'expression  $\sum_{i=1}^n a_i x_i$  sur les intervalles où les  $x_i$  sont définis. Puis, en s'aidant du théorème

des valeurs intermédiaires le test déduit l'existence d'une intersection ou non. Il faut néanmoins remarquer que ce test peut renvoyer des faux positifs : il peut renvoyer vrai même si la solution est non entière. Mais, sous des hypothèses suffisantes, nous pouvons nous assurer de sa correction. En comparaison, notre test est plus limité sur les types d'expressions qu'il traite, mais renvoie uniquement des solutions entières.

## 5 Perspectives future et résultats

### 5.1 Perspectives future

Dans le futur plusieurs perspectives peuvent être envisagées.

- Implémenter la projection. Nous avons étudié cette technique mais ne l'avons pas implémentée par manque de temps et pour son cadre d'application relativement limité.
- Essayer de voir si une étape d'analyse d'intervalles pourrait faire progresser l'outil en lui faisant gérer plus de cas.
- Faire de l'analyse *interprocédurale* pour pouvoir gérer une gamme de fonctions plus étendues (notamment les fonctions avec des effets de bords).
- Étendre l'outil à d'autres technologies de parallélisation qu'OPENACC

### 5.2 Résultats

Nous donnons ci-dessous les transferts générés par quelques tests. Certains d'entre eux viennent de POLYBENCH et les extraits de code concernés sont disponible en annexe E.

Test	Résultat
<pre>for (int i = 0; i &lt; N; ++i) {     if (i &lt; 250)         A[i] = 0; }</pre>	Transfère GPU vers CPU de A[0 : 249]
<pre>for (int i = 0; i &lt; 9999; i++) {     if (i &gt;= 2)         b += A[i-2]; }</pre>	Transfère CPU vers GPU de A[0; 9997]
<pre>for (int i = 0; i &lt; N; i++) {     A[2*i] = 0; }</pre>	Transfère GPU vers CPU de A[0; 2 * (N-1)]

<pre> for (int i = 0; i &lt; N; i++) {     A[i+m] = A[i]; } </pre>	Si $m > N$ , transfère CPU vers GPU de $A[0; N-1]$ et transfère GPU vers CPU de $A[N; N+m-1]$ Sinon, transfère CPU vers GPU et GPU vers CPU de $A[0; N+m-1]$
Test extrait de POLYBENCH : Jacobi-2d	Transfère CPU vers GPU et GPU vers CPU de $A[0:_PB\_N-1][0:_PB\_N-1]$ et de $B[0:_PB\_N-1][0:_PB\_N-1]$
Test extrait de POLYBENCH : blas/gemm	Transfère CPU vers GPU de $A[0:_PB\_NI-1][0:_PB\_NK-1]$ et de $B[0:_PB\_NK-1][0:_PB\_NJ-1]$ Transfère GPU vers CPU de $C[0:_PB\_NI-1][0:_PB\_NJ-1]$

## Conclusion

En conclusion, nous avons développé un outil permettant de générer automatiquement les transferts de données entre GPU et CPU pour un bloc de code donné. Cet outil peut détecter quelles sont les variables et régions de tableau en lecture, écriture ou lecture et écriture. En étudiant les formes linéaires dans les accès aux tableaux, dans les conditions et les déclarations de boucles for, il peut détecter les régions des tableaux accédées. Enfin, il génère du code C permettant de décaler la génération des transferts durant l'exécution du programme, permettant ainsi de prendre en compte les valeurs des constantes inconnues dans le bloc de code entré par l'utilisateur.

Ce stage aura été l'occasion pour moi de concevoir un outil conséquent dans le langage OCaml à partir de rien. J'ai également pu découvrir différentes approches de la parallélisation de code (CUDA, OPENACC) et m'initier au domaine de l'analyse statique.

## Références

- [1] Norme c99. <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>. Accessed : 2017-13-08.
- [2] Norme openacc 2. [https://www.openacc.org/sites/default/files/inline-files/OpenACC\\_2pt5.pdf](https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf). Accessed : 2017-13-08.
- [3] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. Pips is not (just) polyhedral software adding gpu code generation in pips. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjunction with CGO 2011*, 2011.
- [4] Stefan Bygde. Abstract interpretation and abstract domains. 2006.
- [5] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic cpu-gpu communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151. ACM, 2011.
- [6] Tim Jacobson and Gregg Stubbendieck. Dependency analysis of for-loop structures for automatic parallelization of c code. In *36th Annual Midwest Instruction and Computing Symposium, MICS*, 2003.

- [7] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann, 2016.
- [8] Kleanthis Psarris. The banerjee–wolfe and gcd tests on exact data dependence information. *Journal of Parallel and Distributed computing*, 32(2) :119–138, 1996.
- [9] Michael I Schwartzbach. Lecture notes on static analysis. *Basic Research in Computer Science, University of Aarhus, Denmark*, 2008.
- [10] Gagandeep Singh, Markus Püschel, and Martin T Vechev. Fast polyhedra abstract domain. In *POPL*, pages 46–59, 2017.

## A Contexte du stage

Mon stage s'est déroulé au Centre de Recherche en Informatique des Mines Paris-Tech. Les activités de ce laboratoire se concentrent notamment autour des domaines de la programmation parallèle, de l'optimisation de code, d'outils d'analyse statique et de transformations de programmes ainsi que de langages spécifique à un domaine. L'équipe du centre est composée de 13 enseignants chercheurs et d'ingénieurs de recherches, de 3 post-doctorants et 3 doctorants. Plusieurs stagiaires étaient également présents lors de mon séjour.

## B Code de l'outil

En raison de sa taille (plusieurs milliers de lignes), le code OCaml de l'outil est disponible sur github à l'adresse :

<https://github.com/poechsel/StageL3>

## C Exemple de programme généré

Dans cette annexe, nous commentons le code généré par notre outil avec en entrée le programme suivant.

```
for (int i = 0; i < N; ++ i) {  
    A[i+m] = A[i];  
}
```

L'exemple est intéressant car montre l'utilité de décaler les calculs vers l'exécution. Selon la valeur de  $m$  il y a (ou non) un soucis de dépendance, donc le transfert ne doit pas se faire de la même manière. L'exemple est également relativement simple : le code généré reste d'une taille raisonnable, donc est facilement explicable.

Nous commençons par créer les itérateurs. Ils sont générés à partir des contraintes définis par la définition 3.2.

```
1 struct s_iterators it_list[2];  
2 it_list[1] = UNION_INTERVAL(MAKE_OP_INTERVAL("==", 0, 0, -1),  
3                             MAKE_OP_INTERVAL("==", 1*-1+N, 1*-1+N, -1));
```

Ensuite, nous initialisons les structures permettant de contenir les informations sur les régions.

```
4 struct s_infos s_A_infos;  
5 s_A_infos.f_rw.min = (int *) malloc(sizeof(int)*1);  
6 s_A_infos.f_rw.max = (int *) malloc(sizeof(int)*1);  
7 s_A_infos.f_r.min = (int *) malloc(sizeof(int)*1);  
8 s_A_infos.f_r.max = (int *) malloc(sizeof(int)*1);  
9 s_A_infos.f_w.min = (int *) malloc(sizeof(int)*1);  
10 s_A_infos.f_w.max = (int *) malloc(sizeof(int)*1);
```

Les trois lignes suivantes servent pour la parallélisation. Bizarrement, PGI refuse d'envoyer sur le GPU deux zones disjointe d'un tableau de manière séparé à l'aide de pragmas. Pour passer outre ce problème, la solution que nous avons retenue a été de créer volontairement de l'aliasing. Une structure aurait probablement été plus propre, mais PGI génère plusieurs transferts entre le GPU et CPU pour transférer une structure. Définir des variables distinctes permet donc de contourner le problème sans grandes pénalités en terme de performances.

```

11 int * s_A_f_rw = A;
12 int * s_A_f_r = A;
13 int * s_A_f_w = A;

```

Nous passons ensuite au calcul des régions proprement dit. Ici il y a deux régions à calculer. Celle en lecture, avec un accès uni-dimensionnel de la forme `i+m`, et celle en écriture, également uni-dimensionnel mais de la forme `i`.

```

14 {
15     int __a = min(1*it_list[1].min, 1*it_list[1].max)+m;
16     int __b = max(1*it_list[1].min, 1*it_list[1].max)+m;
17     s_A_infos.f_w.min[0] = min(__a, __b);
18     s_A_infos.f_w.max[0] = max(__a, __b);
19 }
20 {
21     int __a = min(1*it_list[1].min, 1*it_list[1].max);
22     int __b = max(1*it_list[1].min, 1*it_list[1].max);
23     s_A_infos.f_r.min[0] = min(__a, __b);
24     s_A_infos.f_r.max[0] = max(__a, __b);
25 }

```

Les transferts peuvent donc être générés. Nous commençons par vérifier que nos deux régions sont distinctes.

```

26 if (intersection_bounds(1, s_A_infos.f_w, s_A_infos.f_r)) {

```

Si ce n'est pas le cas, nous transférons le tableau en écriture et lecture. PGI décide si la parallélisation doit avoir lieu.

```

27     s_A_infos.f_rw.min[0] = min(s_A_infos.f_w.min[0], s_A_infos.f_r.min[0]);
28     s_A_infos.f_rw.max[0] = max(s_A_infos.f_w.max[0], s_A_infos.f_r.max[0]);
29     #pragma acc data pcopy(s_A_f_rw[s_A_infos.f_rw.min[0]: \
30         s_A_infos.f_rw.max[0]-s_A_infos.f_rw.min[0]+1])
31     {
32     #pragma acc kernels
33         for (int i = 0; i<N; ++ i) {
34             s_A_f_rw[i+m] = s_A_f_rw[i];
35         }
36     }
37 }

```

Sinon, les régions en écriture et en lecture sont disjointes. Nous transférons les deux séparément et nous "forçons" le parallélisme.

```

38 } else {
39 #pragma acc data pcopyin(s_A_f_r[s_A_infos.f_r.min[0]: \
40                          s_A_infos.f_r.max[0]-s_A_infos.f_r.min[0]+1])
41 #pragma acc data pcopyout(s_A_f_w[s_A_infos.f_w.min[0]: \
42                           s_A_infos.f_w.max[0]-s_A_infos.f_w.min[0]+1])
43 {
44 #pragma acc parallel loop
45     for (int i = 0; i < N; ++ i) {
46         s_A_f_w[i+m] = s_A_f_r[i];
47     }
48 }
49 }

```

## D Calcul matricielle

Dans cette annexe nous comparons différentes implémentations du calcul matriciel. Une en CUDA, une sur CPU, et une accélérée par OPENACC.

L'implémentation de la multiplication matricielle est donnée ci-dessous.

```

void matrixMultCpu(const int *__restrict A,
                  const int *__restrict B,
                  int *__restrict C,
                  const int size)
{
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            C[i * size + j] = 0;
            for (int k = 0; k < size; ++k) {
                C[i * size + j] += A[i * size + k] * B[k * size + j];
            }
        }
    }
}

```

Celle accélérée par OPENACC est la même que celle sur CPU, à quelques pragmas près.

```

void matrixMultGpu(const int *__restrict A,
                  const int *__restrict B,
                  int *__restrict C,
                  const int size)
{
    #pragma acc data pcopyin(A[0:size*size]) \
                    pcopyin(B[0:size*size]) \
                    pcopyout(C[0:size*size])
    #pragma acc kernels
    #pragma acc loop independent tile(16, 16)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            int sum = 0;

```



```

#pragma acc loop seq
    for (int k = 0; k < size; ++k) {
        sum += A[i * size + k] * B[k * size + j];
    }
    C[i * size + j] = sum;
}
}
}

```

Enfin, l'implémentation CUDA. Il n'y a que le kernel (donc le code exécute par les threads). Les mécanismes de transferts de données ne sont pas mis dans le code contrairement à la version OPENACC car l'alourdisant trop. Ici l'algorithme utilise la mémoire partagée pour aller plus vite. Cette mémoire possède des temps d'accès plus réduits que la mémoire globale, mais est disponible en plus faible quantité.

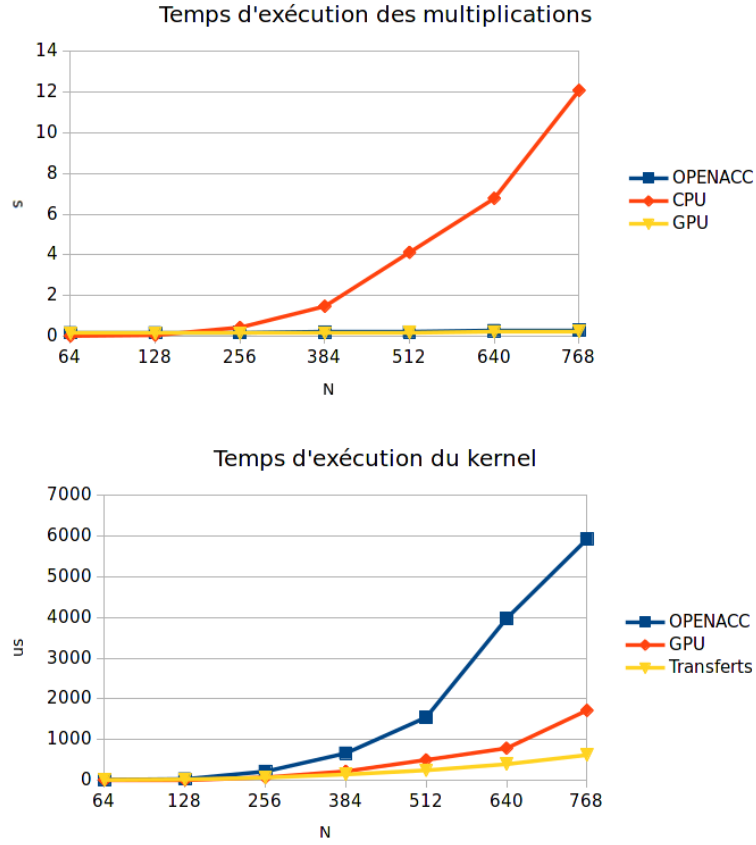
```

__global__ void matrixMultCuda(int *A, int *B, int *C, int n)
{
    __shared__ float Mds[KERNELSIZE][KERNELSIZE];
    __shared__ float Nds[KERNELSIZE][KERNELSIZE];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    float value = 0.0;
    int row = by * KERNELSIZE + ty;
    int col = bx * KERNELSIZE + tx;
    for (int m = 0; m < (n / KERNELSIZE); ++m) {
        Mds[ty][tx] = A[row * n + (m * KERNELSIZE + tx)];
        Nds[ty][tx] = B[(m * KERNELSIZE + ty) * n + col];
        __syncthreads();
        for (int e = 0; e < KERNELSIZE; ++e)
            value += Mds[ty][e] * Nds[e][tx];
        __syncthreads();
    }
    C[row * n + col] = value;
}

```

Nous pouvons remarquer que le programme CUDA a une complexité d'implémentation plus grande que celui d'OPENACC. Il est néanmoins plus optimisé que ce dernier.

Ci-dessous nous pouvons voir deux graphiques. Le premier compare les performances des trois implémentations. Pour les versions accélérées sur GPU (CUDA et OPENACC), les surcoûts de création du contexte GPU, les transferts et l'exécution de l'algorithme sont pris en compte. Le gain en performance des versions accélérées est flagrant. Le second affiche le temps d'exécution du kernel dans le cas de la version avec OPENACC et de celle avec CUDA. Comme nous pouvions nous y attendre, la version CUDA est plus rapide que celle OPENACC. Les temps de transferts sont non négligeables par rapport à ceux de l'exécution du kernel.



## E Extraits de Polybench

```

for (t = 0; t < _PB_TSTEPS; t++) {
    for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
            B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                           A[i][1+j] + A[1+i][j] + A[i-1][j]);
    for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
            A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                           B[i][1+j] + B[1+i][j] + B[i-1][j]);
}

```

FIGURE 4: POLYBENCH : Jacobi-2d

```

for (i = 0; i < _PB_NI; i++) {
    for (j = 0; j < _PB_NJ; j++)
        C[i][j] *= beta;
    for (k = 0; k < _PB_NK; k++) {
        for (j = 0; j < _PB_NJ; j++)
            C[i][j] += alpha * A[i][k] * B[k][j];
    }
}

```

FIGURE 5: POLYBENCH : blas/gemm