Ordonnancement polyédrique incrémental

Thaïs Baudon

12 juillet 2018

Résumé

La parallélisation automatique de programmes est nécessaire pour exploiter la performance des processeurs multi-cœurs. Apollo est une plateforme de parallélisation spéculative, qui utilise l'outil Pluto pour transformer le programme en cours d'exécution. Les transformations appliquées déterminent la qualité de la solution obtenue. L'objectif de ce stage est d'établir une stratégie de compilation simultanée de plusieurs solutions et de gestion des priorités entre ces solutions.

Mots-clefs : compilation, optimisation et parallélisation automatiques, parallélisation à la volée

Référence : Stage effectué du 28 mai au 24 août 2018 au laboratoire ICube sous la direction de Philippe Clauss, INRIA CAMUS, ICube.

Table des matières

Introduction **1**

1 Modèle polyédrique **2**
- 1.1 Définitions . . . 2
- 1.2 Représentation d'un programme . . . 2
- 1.3 Dépendances . . . 4
  - 1.3.1 Représentation . . . 4
- 1.4 Ordonnancement . . . 5
- 1.5 Transformations . . . 5
  - 1.5.1 Exemples de transformations . . . 6
- 1.6 Limites du modèle polyédrique . . . 7

2 Parallélisation à la volée **7**
- 2.1 Apollo . . . 7
- 2.2 Approche incrémentale . . . 8

3 Implémentation du poly-ordonnanceur **10**
- 3.1 Étude des temps de compilation source-à-source et d'exécution . . . 10
- 3.2 Implémentation du poly-ordonnanceur . . . 11

Introduction

Les solutions consistant à augmenter la fréquence des micro-processeurs et/ou à réduire la taille des transistors utilisés ne sont plus applicables en raison des limites physiques de puissance et de miniaturisation. La solution mise en œuvre est d'utiliser des processeurs comportant plusieurs cœurs, pouvant effectuer des opérations simultanément. Les processeurs multi-cœurs sont désormais présents dans la grande majorité des machines. Leur intérêt est cependant limité lorsque les programmes exécutés sont uniquement séquentiels, et donc théoriquement restreints à une exécution sur un seul cœur. Pour exploiter leur performance, il est nécessaire que les programmes contiennent des informations sur les parties pouvant être exécutées sur plusieurs cœurs. Cependant, le développement de programmes parallèles est complexe, long et source d'erreurs. En effet, la parallélisation d'un programme séquentiel impose d'identifier les dépendances de données qui existent entre les instructions, puis de trouver une transformation permettant de paralléliser le programme qui tient compte de ces dépendances. Il est donc nécessaire de fournir des outils permettant de paralléliser automatiquement des programmes séquentiels.

Le modèle polyédrique [5] est un modèle mathématique de représentation des programmes. Il permet d'optimiser et de paralléliser un programme en appliquant des transformations sur sa représentation dans le modèle pour générer un programme optimisé. *Pluto* [7] est un outil de parallélisation et d'optimisation, qui sélectionne et applique automatiquement des transformations sur la représentation polyédrique d'un programme.

Ce modèle comporte néanmoins des limites : seuls les programmes répondant à certains critères (*boucles affines*) peuvent être représentés de façon exacte ; de plus, les transformations appliquées doivent être valables pour toutes les situations possibles lors de l'exécution. *Apollo* [1, 3, 9] est une plate-forme de parallélisation spéculative de boucles qui permet de contourner ces limites, en observant le comportement d'un programme lors de son exécution pour construire un modèle de prédiction compatible avec le modèle polyédrique. Ce modèle est ensuite optimisé et parallélisé par Pluto. Cette stratégie permet de paralléliser des programmes qui ne peuvent pas être représentés de façon exacte dans le modèle polyédrique par un compilateur statique, mais dont le comportement à l'exécution se rapproche de celui d'un programme constitué de boucles affines. D'autre part, il n'est plus nécessaire de trouver une transformation valide pour toutes les exécutions possibles, car le choix de la transformation s'effectue après avoir observé l'exécution réelle et construit un modèle de prédiction.

Les optimisations réalisées par Pluto sont déterminées par des options, qui précisent quelles transformations appliquer. Ces options influencent donc la qualité des solutions obtenues, ainsi que le temps nécessaire pour les générer.

L'objectif de ce travail est de mettre en place un système de *poly-ordonnancement*, avec une approche incrémentale qui consiste à générer plusieurs solutions avec plusieurs instances de Pluto en parallèle. Il consistera à étudier les temps de compilation et d'exécution associés à différentes combinaisons d'options, puis à établir une stratégie de gestion des priorités entre les solutions générées.

Après une introduction au modèle polyédrique en section 1, la section 2 présentera l'étude réalisée sur les temps de compilation et d'exécution pour différentes options de Pluto, puis la stratégie de gestion des priorités établie.

1 Modèle polyédrique

Le modèle polyédrique [5] est un modèle mathématique de représentation des programmes constitués de boucles affines, c'est-à-dire de boucles imbriquées dont les bornes et les indices des références mémoire sont des fonctions affines des indices des boucles externes et de constantes. Ce modèle permet l'analyse exacte des dépendances entre les itérations et instructions, afin d'appliquer des transformations qui permettent d'extraire du parallélisme et d'améliorer la localité des données tout en préservant la sémantique du programme.

La performance du programme dépend d'une part du parallélisme mis en évidence, et d'autre part de la localité des données, c'est-à-dire de la réutilisation de données déjà utilisées dans le passé (*localité temporelle*) et de l'utilisation de données situées dans un emplacement mémoire proche d'un emplacement accédé récemment (*localité spatiale*).

Pour cela, le programme est analysé pour obtenir une représentation polyédrique. Des transformations affines sont ensuite appliquées sur cette représentation, afin de générer un code optimisé et parallélisé sémantiquement équivalent au programme séquentiel.

1.1 Définitions

Demi-espace affine Un hyperplan affine $H = \{\vec{x} \in \mathbb{R}^m \mid \vec{v} \cdot \vec{x} = k\}$ (avec $\vec{v} \in \mathbb{R}^m \backslash \{\vec{0}\}$ et $k \in \mathbb{R}$) divise l'espace $\mathbb{R}^m$ en deux demi-espaces affines $H_1 = \{\vec{x} \in \mathbb{R}^m \mid \vec{v} \cdot \vec{x} \leq k\}$ et $H_2 = \{\vec{x} \in \mathbb{R}^m \mid \vec{v} \cdot \vec{x} \geq k\}$.

Polyèdre Un polyèdre (de $\mathbb{R}^m$) est une intersection finie de demi-espaces, c'est-à-dire un ensemble de la forme :

$$\{\vec{x} \in \mathbb{R}^m \mid A\vec{x} + \vec{b} \geq \vec{0}\}$$

SCoP Le modèle polyédrique est adapté aux programmes qui présentent les caractéristiques suivantes :

1. Chaque boucle itère selon un seul indice, dont les bornes sont des fonctions affines des indices des boucles englobantes
2. Les accès mémoire sont uniquement effectués sur des scalaires ou sur des tableaux (éventuellement multi-dimensionnels) indexés par des fonctions affines des indices des boucles englobantes.

Ces boucles imbriquées sont analysées précisément, en tenant compte des dépendances de données qui existent entre les instructions et entre les itérations. Une *SCoP* (*Static Control Part*) est une partie du code composée de boucles imbriquées qui correspondent à ces critères. Dans une *SCoP*, les accès mémoire, les bornes des boucles et les conditions sont des fonctions affines des itérateurs des boucles englobantes et des paramètres dont la valeur est constante à l'exécution. Un exemple de *SCoP* est donné figure 1.

1.2 Représentation d'un programme

Domaines d'itération Soit S une instruction contenue dans n boucles imbriquées. Les indices de ces boucles définissent plusieurs *instances* de cette instruction. Chaque instance correspond à une exécution particulière de l'instruction, associée aux valeurs des indices

```
for (i = 1; i <= N; i++)
{
    for (j = 1; j <= i; j++)
    {
        if (i + j <= M + 1 || i >= M)
            // S0
            A[i][j] = A[i-1][j] + 1;

        // S1
        B[i][j] = A[j][N + 1 - i];
    }
}
```

FIGURE 1 – Exemple de *SCoP*

des boucles lors de son exécution. Le vecteur de ces indices est appelé *vecteur d'itération* de l'instance. L'ensemble des vecteurs d'itération de toutes les instances d'une instruction constitue son *domaine d'itération*, qui représente l'ensemble des instances et est défini par les contraintes que doivent satisfaire les indices des boucles pour que l'instruction soit exécutée. Le domaine d'itération de S est modélisé par un polyèdre (ou une union de polyèdres) de dimension n :

$$\mathcal{D}^S(\vec{p}) = \{\vec{x} \in \mathbb{Z}^n \mid A\vec{x} + B\vec{p} + \vec{b} \geq \vec{0}\}$$

L'exécution séquentielle du programme correspond à l'exécution des instances selon l'ordre lexicographique de leurs vecteurs d'itération. Le domaine d'itération de l'instruction S_0 de la *SCoP* de la figure 1 est représenté par la figure 2.

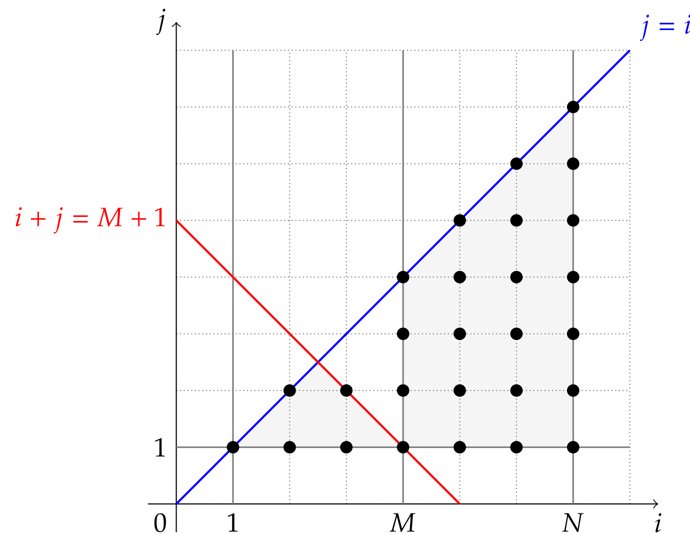

FIGURE 2 – Domaine d'itération de l'instruction S_0

Fonctions d'accès mémoire Les accès mémoire (en lecture ou en écriture) effectués par une instruction sont représentés par des fonctions affines des indices et des paramètres des boucles qui la contiennent. Ces fonctions, appelées *fonctions d'accès*, permettent

d'analyser précisément les dépendances de données. Chaque fonction est de la forme

$$f(\vec{x}) = F\vec{x} + \vec{f}$$

où $F \in \mathbb{Z}^{l \times d}$ est la matrice des coefficients des indices de tableaux, l la dimension du tableau accédé, d la *profondeur de boucle* de l'instruction, $\vec{x}$ le vecteur d'itération et $\vec{f}$ un vecteur constant. Chaque accès mémoire réalisé par l'instruction est associé à une fonction d'accès.

La fonction correspondant à l'accès en lecture de l'instruction S_1 de l'exemple est :

$$f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ N+1 \end{pmatrix} = \begin{pmatrix} j \\ N+1-i \end{pmatrix}$$

1.3 Dépendances

L'analyse des dépendances de données permet d'appliquer des transformations sur les boucles tout en préservant la sémantique du programme.

Définition Soient S et T deux instructions. T dépend de S, et on note $T\delta S$, s'il existe deux instances $S(\vec{x})$ et $T(\vec{y})$, et un emplacement mémoire m, tels que :

1. $S(\vec{x})$ et $T(\vec{y})$ accèdent à m
2. Au moins une des deux instances y accède en écriture
3. $S(\vec{x})$ est exécutée avant $T(\vec{y})$ dans l'ordre de l'exécution séquentielle du programme.

Pour toute transformation du programme qui conserve sa sémantique, l'ordre d'exécution relatif de deux instances dépendantes dans le programme transformé doit rester le même que dans le programme séquentiel.

Classification des dépendances Les dépendances peuvent être classifiées selon les types d'accès mémoire concernés :

- *RAW* (*read after write*) ou dépendance de flot ;
- *WAR* (*write after read*) ou anti-dépendance ;
- *WAW* (*write after write*) ou dépendance de sortie ;
- *RAR* (*read after read*) ou dépendance d'entrée, qui ne correspond pas à la définition de dépendance.

Les dépendances *RAW* sont intrinsèques au programme et ne peuvent pas être supprimées. Les dépendances *WAR* et *WAW* peuvent être contournées par des méthodes telles que le renommage ou l'expansion de tableaux [5]. Les dépendances *RAR* ne sont pas considérées comme des dépendances car aucune écriture n'est réalisée. Il peut toutefois être utile de les prendre en compte pour effectuer certaines optimisations pouvant améliorer la localité des données, en réduisant la distance de réutilisation des données.

1.3.1 Représentation

Les dépendances peuvent être représentées selon deux approches : les *vecteurs de distance* et les *polyèdres de dépendances*.

Représentation par des vecteurs de distance Un *vecteur de distance* $\vec{d}$ représente la distance entre une instruction source $S(\vec{x})$ et une instruction cible $T(\vec{y})$ qui dépend de l'instruction source. Il est défini par : $\vec{d} = \vec{y} - \vec{x}$. Le *vecteur de direction* indique la direction de la dépendance. Il s'agit du signe du vecteur de distance, c'est-à-dire du vecteur des signes de ses composantes. Ces deux vecteurs sont toujours lexicographiquement positifs ou nuls, c'est-à-dire que leurs premières composantes non nulles sont positives. Soit d_l la première composante non nulle de $\vec{d} = (d_1, ..., d_{h+1})$. Si $1 \leq l \leq h$, alors la dépendance est portée par la boucle de profondeur l. Sinon, $l = h + 1$, $\vec{d} = \vec{0}$ et la dépendance n'est portée par aucune boucle.

Polyèdre de dépendance Les dépendances peuvent être représentées par un graphe orienté dont chaque sommet correspond à une instruction et chaque arête à une dépendance entre deux instructions. Pour chaque arête, la relation entre les instances des deux instructions dépendantes est modélisée par un *polyèdre de dépendance*. La dimension de ce polyèdre est égale à la somme des dimensions des vecteurs d'itération et des dimensions correspondant aux paramètres et aux constantes. Les conditions exactes d'existence d'une dépendance entre deux instances particulières des instructions sont déterminées par des inégalités affines, issues des fonctions d'accès mémoire.

1.4 Ordonnancement

Fonction d'ordonnancement La *fonction d'ordonnancement* d'une instruction S associe à chaque instance de S une *date logique* qui définit l'ordre d'exécution des instances de S par rapport aux autres instructions. Cette fonction est de la forme :

$$\theta^S(\vec{x}) = \Theta^S \vec{x} + \vec{t}$$

avec $\vec{x} \in \mathcal{D}^S$, Θ^S une *matrice d'ordonnancement* et $\vec{t}$ un vecteur constant. Deux instances peuvent être exécutées en parallèle si leurs dates logiques sont égales. Si la date logique d'une instance $S(\vec{x_S})$ est lexicographiquement strictement inférieure à la date logique d'une autre instance $T(\vec{x_T})$, alors $S(\vec{x_S})$ doit être exécutée avant $T(\vec{x_T})$.

1.5 Transformations

Les transformations visent à réordonner les itérations et les instructions afin d'améliorer la localité des données ou de mettre en évidence du parallélisme. Une transformation associe à chaque instance une date logique dans le domaine d'itération transformé, qui conserve la sémantique du programme original, pour aboutir à un nouvel ordre d'exécution des instructions.

Forme affine du lemme de Farkas Soit $\mathcal{D}$ un polyèdre non vide défini par le système d'inégalités affines $A\vec{x} + \vec{b} \geq \vec{0}$. Alors toute fonction affine $f(\vec{x})$ est positive ou nulle sur $\mathcal{D}$ *ssi* c'est une combinaison linéaire positive ou nulle des faces de $\mathcal{D}$, c'est-à-dire :

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}(A\vec{x} + \vec{b})$$

avec $\lambda_0 \geq 0$ et toutes les composantes de $\vec{\lambda}$ positives ou nulles.

Ce résultat permet de caractériser l'ensemble des transformations valides, qui respectent l'ensemble des dépendances.

Application d'une transformation Une transformation peut être représentée par une fonction/matrice d'ordonnancement qui associe à chaque instance une nouvelle date logique. Les transformations légales sont celles qui garantissent que la différence entre les dates logiques de deux instances dépendantes est toujours strictement positive.

Soient S et T deux instructions telles que $T\delta S$. Soit $\mathcal{P}_e$ le polyèdre de dépendance associé.

$$\mathcal{P}_e = \{\vec{x} \mid A\vec{x} + \vec{b} \geq \vec{0}\}$$

Pour deux itérations dépendantes $\vec{x_i}$ et $\vec{x_j}$, la différence entre les ordonnancements $\Theta^S\vec{x_i}$ et $\Theta^T\vec{x_j}$ doit être positive :

$$\forall(\vec{x_i}, \vec{x_j}) \in \mathcal{P}_e, \Theta^T\vec{x_j} - \Theta^S\vec{x_i} \geq \vec{0}$$

D'après le lemme de Farkas :

$$\forall\vec{x_i}, \vec{x_j} \in \mathcal{P}_e, \Theta^T\vec{x_j} - \Theta^S\vec{x_i} = \lambda_0 + \vec{\lambda}(A(\vec{x_i}|\vec{x_j}) + \vec{b}), \text{avec } \lambda_0 \geq 0 \text{ et } \vec{\lambda} \geq \vec{0}$$

L'ensemble des solutions définit l'espace des ordonnancements légaux *selon cette dépendance*.

Une boucle de niveau l est parallélisable seulement s'il n'existe aucune dépendance entre deux instances appartenant à des itérations différentes.

1.5.1 Exemples de transformations

Ces transformations peuvent être effectuées par le compilateur *Pluto* (voir la section 2).

Pavage Le *pavage* est une transformation qui consiste à partitionner le domaine d'itération en *pavés*, c'est-à-dire en groupes d'itérations de même forme et de même taille [10]. L'intérêt du pavage est de regrouper les itérations en pavés qui peuvent être exécutés par un cœur de processeur de façon atomique, c'est-à-dire sans communication avec d'autres cœurs et en stockant toutes les données nécessaires dans le cache [2].

Torsion de boucles Cette transformation consiste à réordonner les itérations de manière à éliminer les dépendances entre les itérations des boucles internes en déplaçant ces dépendances au niveau de la boucle externe, en remplaçant l'indice d'une boucle par une combinaison linéaire d'autres indices.

Fusion de boucles

Génération du code transformé Le problème est de trouver un code qui visite chaque point entier de chaque polyèdre une et une seule fois en respectant l'ordre lexicographique de certaines dimensions. L'outil *CLooG* (utilisé par le compilateur source-à-source *Pluto*) est un générateur de code qui réalise ce travail d'analyse de polyèdres [4].

Pluto [7] est un compilateur source-à-source qui permet d'optimiser et de paralléliser automatiquement des programmes contenant des *SCoP*, en y appliquant des transformations affines [2].

1.6 Limites du modèle polyédrique

Ce modèle est applicable uniquement aux programmes constitués de boucles affines (*SCoP*). Cette restriction exclut une grande partie des programmes (par exemple ceux contenant des accès mémoire par pointeurs, des boucles *while* . . .). De plus, il est nécessaire de trouver une transformation valable dans toutes les situations potentiellement rencontrées lors de l'exécution. Or il est en général impossible de connaître précisément le comportement du programme avant l'exécution.

Apollo est une plateforme de parallélisation automatique de boucles imbriquées qui ne peuvent pas être traitées par les compilateurs statiques, par exemple des boucles *while*, comportant des accès mémoire avec des pointeurs . . . La stratégie employée est spéculative : l'observation de l'exécution du programme sur quelques itérations permet de modéliser son comportement par des *SCoP*, et donc d'utiliser le modèle polyédrique pour le paralléliser et l'optimiser [1].

2 Parallélisation à la volée

2.1 Apollo

La plate-forme Apollo [1, 3, 9] permet de contourner certaines limites du modèle polyédrique, par une stratégie spéculative. Celle-ci consiste à observer le comportement du programme (qui n'est pas forcément constitué de *SCoP*) au cours de son exécution pendant quelques itérations de sa boucle principale. Cette observation permet de construire un modèle de prédiction de son comportement compatible avec le modèle polyédrique. Des transformations sont ensuite sélectionnées et appliquées sur ce modèle pour aboutir à un programme optimisé. Un mécanisme d'exécution en tranches de boucles permet d'interrompre l'exécution du programme original et de reprendre l'exécution avec le programme optimisé. Afin de réduire l'impact du temps de compilation à la volée, le programme original séquentiel est exécuté en parallèle de la génération du programme parallélisé [1].

Ainsi, les programmes dont la structure ne correspond pas à une *SCoP* mais dont le comportement à l'exécution peut être modélisé par des boucles affines peuvent être parallélisés. De plus, les transformations appliquées sont déterminées par le comportement réel du programme exécuté ; il n'est donc plus nécessaire de trouver une transformation valable dans toutes les situations potentiellement rencontrées lors de l'exécution.

La sélection et l'application des transformations sont réalisées par le compilateur source-à-source Pluto, qui permet de paralléliser et d'optimiser automatiquement des programmes contenant des *SCoP*. Les transformations appliquées par Pluto, et donc les solutions obtenues, sont déterminées par des options. La combinaison d'options choisie influence la qualité de la solution générée par Pluto, ainsi que son temps de génération.

Apollo est constitué d'une composante statique et d'une composante dynamique (voir la figure 3). La composante statique analyse les parties du programme à optimiser (appelées *DCoP*, pour *Dynamic Control Parts*) et génère des parties de code élémentaires qui seront utilisées par le module dynamique pour construire un programme optimisé lors de l'exécution [3]. La composante dynamique est intégrée à l'exécutable généré par la partie statique et dirige la compilation à la volée et l'exécution des *DCoP*.

La boucle externe du nid de boucles à optimiser est divisée en *tranches* (ou *chunks*) de plusieurs itérations. Chaque tranche est associée à une version du programme, instrumentée, séquentielle ou optimisée (voir la figure 4).

La première tranche d'exécution est associée à une version *instrumentée* du programme. Les accès mémoire réalisés lors de l'exécution, les scalaires et les bornes des boucles qui n'ont pas pu être analysés lors de la compilation statique sont enregistrés. Les informations issues de cette exécution instrumentée sont utilisées pour construire un modèle de prédiction linéaire des accès mémoire, des scalaires et des bornes des boucles du programme. L'analyse du programme réalisée lors de la compilation (statique) et le modèle de prédiction permettent d'identifier les dépendances entre les accès mémoire du programme et de construire le polyèdre de dépendance associé [9].

Pluto est ensuite utilisé pour déterminer un ordonnancement légal à partir du polyèdre de dépendance. Le programme original est exécuté pendant la génération de l'ordonnancement afin de masquer une partie du coût de la transformation. *CLooG* est utilisé pour déterminer le parcours de chaque point du polyèdre transformé généré par Pluto, afin de construire un nouveau programme à partir de parties élémentaires de code (générées lors de la compilation statique) et d'instructions de vérification de la spéculation [3]. Le code généré est ensuite "converti" en représentation intermédiaire LLVM, puis compilé. La compilation *juste-à-temps* (*JIT*) du programme optimisé permet également d'appliquer des optimisations supplémentaires.

Pendant l'exécution du programme optimisé, chaque *thread* d'exécution vérifie que le modèle de prédiction est valide, c'est-à-dire que toutes les dépendances sont respectées. Si ce n'est pas le cas, un retour en arrière (*rollback*) vers la dernière situation valide (c'est-à-dire à la fin du *chunk* précédent) est réalisé, et un nouveau modèle de prédiction est construit à partir d'une nouvelle exécution instrumentée. À la fin de chaque *chunk* d'exécution optimisée, le modèle de prédiction est valide et l'état courant est sauvegardé en vue de l'exécution du prochain *chunk* (voir la figure 5).

2.2 Approche incrémentale

L'approche incrémentale implémentée consiste à générer simultanément plusieurs solutions avec des combinaisons d'options différentes. Une solution de faible qualité mais qui est générée en peu de temps permet d'obtenir rapidement un programme optimisé. D'autres solutions de meilleure qualité, dont la compilation nécessite plus de temps, sont générées en parallèle. Ces solutions pourront remplacer la première solution dès la fin de leur compilation. Cette approche nécessite de définir une stratégie de gestion des priorités entre les différentes solutions. Une stratégie immédiate consiste à lancer simultanément une instance de Pluto pour chaque solution à générer. Dès qu'une solution est prête, si sa priorité est supérieure à celle de la solution en cours d'exécution, alors elle remplace la solution précédente. Si plusieurs solutions sont disponibles, seule celle de plus haute priorité est retenue.

Cette stratégie naïve présente plusieurs inconvénients. Les priorités définies à partir des résultats de l'étude peuvent ne pas correspondre aux performances réelles des solutions sur un programme en particulier. En ne retenant que les solutions de plus haute priorité, c'est-à-dire de meilleure performance "théorique", cette stratégie pourrait éliminer des solutions qui étaient en réalité de meilleure qualité, malgré leur plus faible priorité. Afin de reconnaître les solutions réellement plus performantes, le temps d'exécution des différentes solutions doit être mesuré

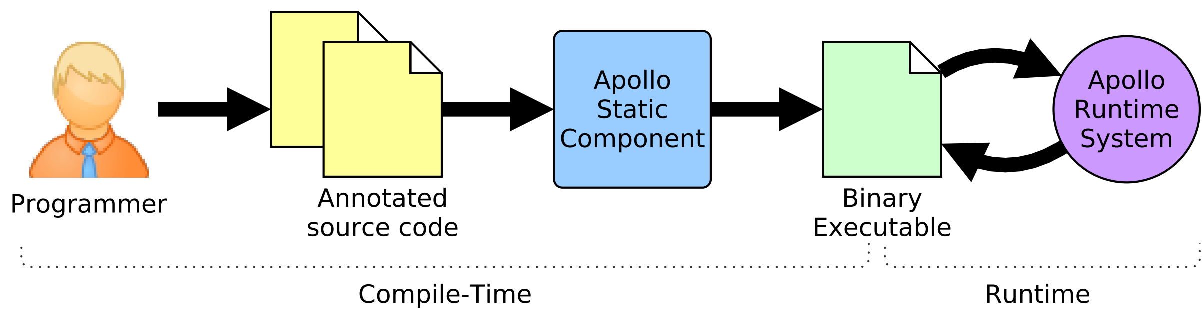

FIGURE 3 – Parties statique et dynamique d'Apollo

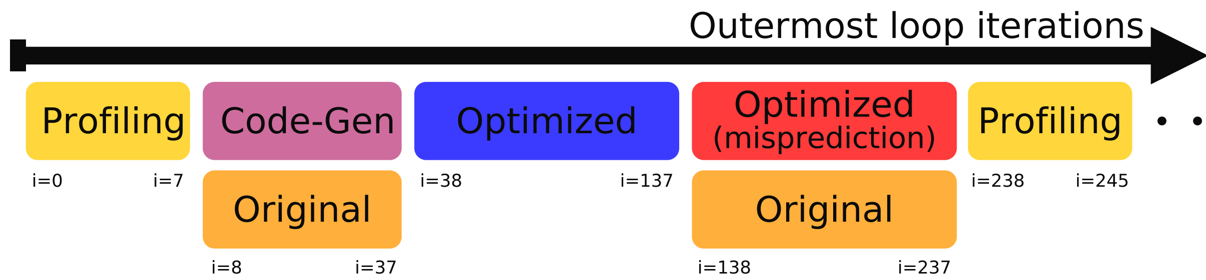

FIGURE 4 – Exécution en tranches de la boucle externe

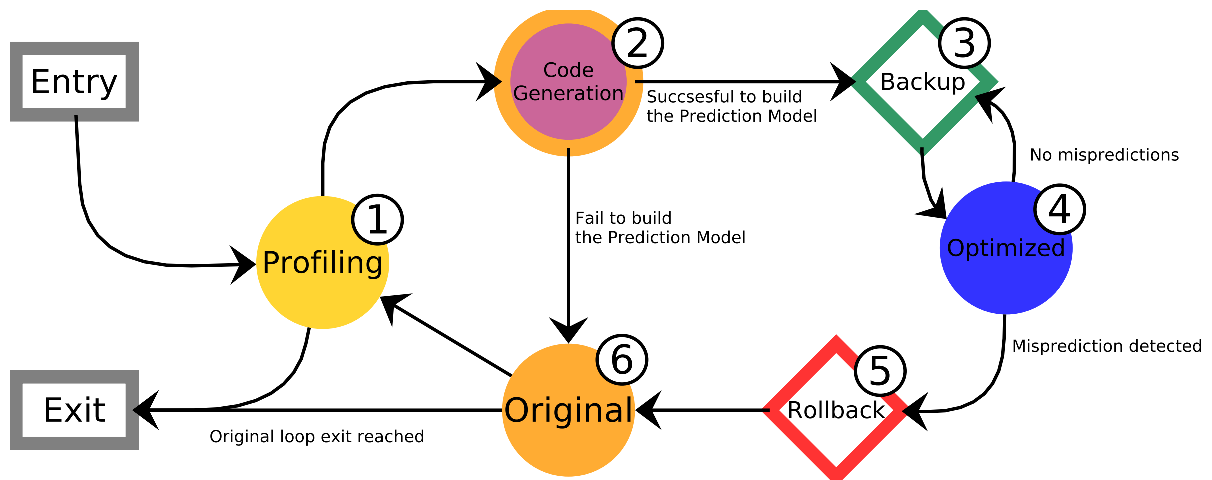

FIGURE 5 – Module dynamique d'Apollo

sur quelques itérations de la boucle externe du programme, pour établir un classement des solutions issu des temps d'exécution effectifs.

De plus, selon le nombre de combinaisons d'options retenues et le nombre de solutions pouvant être compilées en parallèle (qui dépend du nombre de *threads* disponibles), il n'est pas toujours possible de générer simultanément l'intégralité des solutions. Il est donc nécessaire d'établir des priorités qui dépendent non seulement de la performance attendue, mais aussi du temps nécessaire pour compiler chaque solution.

L'étude réalisée sur les programmes du banc d'essai *PolyBench* [8] a permis de sélectionner certaines combinaisons d'options et d'estimer leur qualité ainsi que leur temps de compilation. Un nombre limité de *threads* est disponible pour compiler ces solutions. Dans un premier temps, au moins un *thread* doit être utilisé pour générer une solution avec un faible temps de compilation afin d'obtenir une solution optimisée le plus rapidement possible.

La première exécution d'une nouvelle solution concerne un nombre réduit d'itérations de la boucle externe. Au cours de cette tranche d'exécution, la durée de l'exécution et le nombre d'instructions exécutées sont mesurés. Ces mesures permettent d'évaluer la qualité réelle de la solution. Si la nouvelle solution est de moindre qualité que la solution précédente, alors elle est supprimée. Sinon, elle est plus performante (ou au moins aussi performante) que la solution précédente et la remplace. Les solutions qui n'ont pas encore été exécutées sont prioritaires par rapport aux solutions déjà exécutées, afin de connaître la performance de toutes les solutions générées.

3 Implémentation du poly-ordonnanceur

3.1 Étude des temps de compilation source-à-source et d'exécution

Afin d'établir une hiérarchie entre les différentes combinaisons d'options, il est nécessaire d'évaluer la qualité (qui correspond au temps d'exécution) ainsi que le temps de compilation de chaque solution. L'étude des temps de compilation et d'exécution a été réalisée sur la collection de programmes *PolyBench*, qui regroupe une trentaine de programmes contenant des *SCoP* [8].

Pour chaque programme et chaque combinaison d'options, l'évaluation du temps nécessaire à la génération du programme transformé correspond à la moyenne des durées de trois compilations source-à-source successives. Les temps de compilation supérieurs à 30 secondes n'ont pas été pris en compte.

L'option `--parallel` de Pluto introduit des directives de compilation *OpenMP* dans le code pour permettre une exécution parallèle du programme. Le temps d'exécution des exécutables obtenus dépend donc du nombre de *threads* disponibles, ainsi que des cœurs de processeur qui correspondent à ces *threads*, car la distance de réutilisation des données est influencée par le partage de certains caches entre différents cœurs. Ces programmes ont été exécutés successivement avec 6 puis 12 *threads*, sur six cœurs de processeur (Intel Xeon X5650 (2.67GHz)), partageant un même cache L3 de 12 ko et disposant chacun d'un cache L2 de 256 ko, d'un cache de données L1 de 32 ko et d'un cache d'instructions L1 de 32 ko, avec respectivement un et deux *thread(s)* par cœur pour les exécutions avec 6 et 12 *threads*.

Le temps d'exécution dépend également de la taille des données d'entrée. *PolyBench* permet de définir la taille du jeu de données lors de la compilation. Chaque fichier source généré par

Pluto a été compilé pour trois tailles de données différentes *M*, *L* et *XL*, avec le compilateur *GCC* et les options d'optimisation `-O3 -march=native -mtune=native -ftree-vectorize`.

Pour chaque exécutable obtenu et chaque nombre de *threads* disponibles, le temps d'exécution évalué correspond à la moyenne de trois exécutions successives. Les temps d'exécution supérieurs à 70 secondes n'ont pas été pris en compte.

Les options de Pluto correspondent à certaines transformations appliquées à la représentation polyédrique du programme (voir la section 1.5.1) :

`--tile` : pavage des domaines d'itération.

`--tile --l2tile` : pavage supplémentaire pour le cache L2.

`--intratileopt` : optimisation de l'ordre d'exécution des instances dans un pavé pour maximiser la localité des données.

`--parallel` : parallélisation du code avec des directives *OpenMP*.

`--nofuse`, `--smartfuse`, `--maxfuse` : détermine la stratégie de fusion des boucles.

`--rar` : prise en compte des dépendances *RAR* (*read-after-read*).

`--lastwriter` : ne pas prendre en compte les dépendances transitives ; un seul niveau de dépendance est pris en compte.

`--iss` : (*Index Set Splitting*) division des domaines d'itération de chaque instruction en plusieurs parties dont les ordonnancements sont indépendants [6].

`--diamond-tile`, `--part-diamond-tile` : méthode de pavage permettant de commencer l'exécution de tous les pavés d'une face du domaine d'itération simultanément.

Cette étude a permis d'estimer l'effet des différentes options sur la qualité et le temps de compilation des solutions, afin d'identifier des combinaisons d'options pertinentes et de leur attribuer des priorités en fonction des temps mesurés.

Les options `--parallel` et `--intratileopt` permettent d'améliorer la qualité des solutions obtenues, pour un coût négligeable. `--tile` permet pour certains des programmes testés d'obtenir une solution de meilleure qualité ; le coût de `--tile --l2tile` est plus important que celui de `--tile` et le temps d'exécution des solutions obtenues est pour la majorité des programmes testés plus élevé qu'avec `--tile`. Les figures 6 et 7 représentent le coût et la qualité des solutions obtenues pour les options `--tile` et `--tile --l2tile` par rapport aux solutions obtenues sans ces options.

3.2 Implémentation du poly-ordonnanceur

La stratégie présentée en 2.2 a été implémentée dans un prototype basé sur la plateforme Apollo.

Chaque *thread* de compilation génère la solution dont le temps de compilation estimé est le plus court parmi les solutions qui ne sont pas générées par un autre *thread*. Plusieurs de ces *threads* peuvent utiliser Pluto simultanément pour appliquer des transformations qui correspondent aux options associées à la solution générée. En revanche, la compilation *JIT* est limitée à un seul *thread* de compilation, pour des raisons techniques non résolues à ce stade.

La performance effective des solutions est estimée en mesurant le temps d'exécution de chaque solution sur une tranche d'un certain nombre d'itérations de la boucle externe. Cependant, le temps moyen d'exécution d'une itération de la boucle externe n'est pas toujours

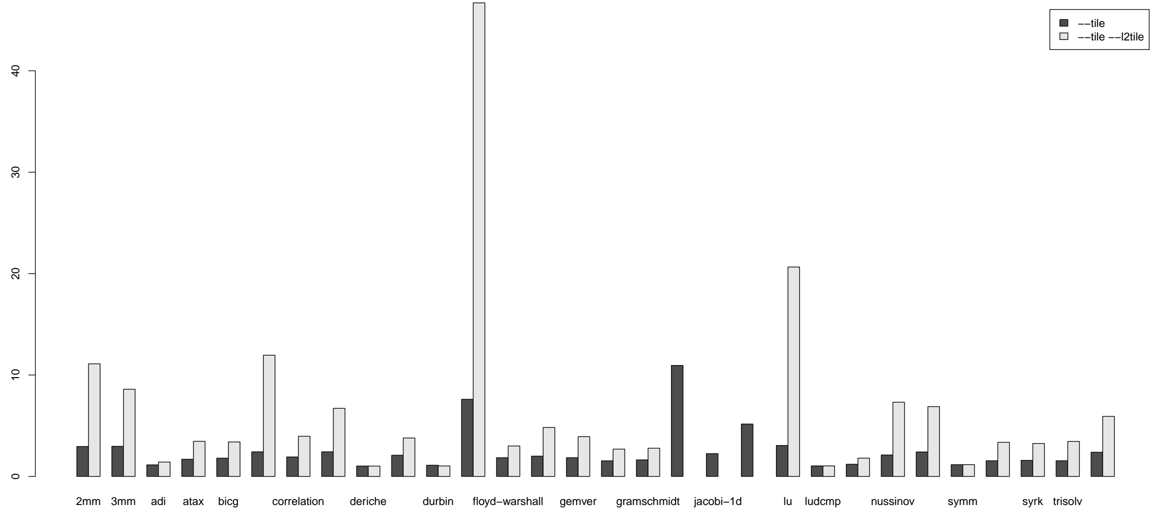

FIGURE 6 – Rapport des temps de compilation des solutions obtenues avec et sans les options –tile et –tile –l2tile

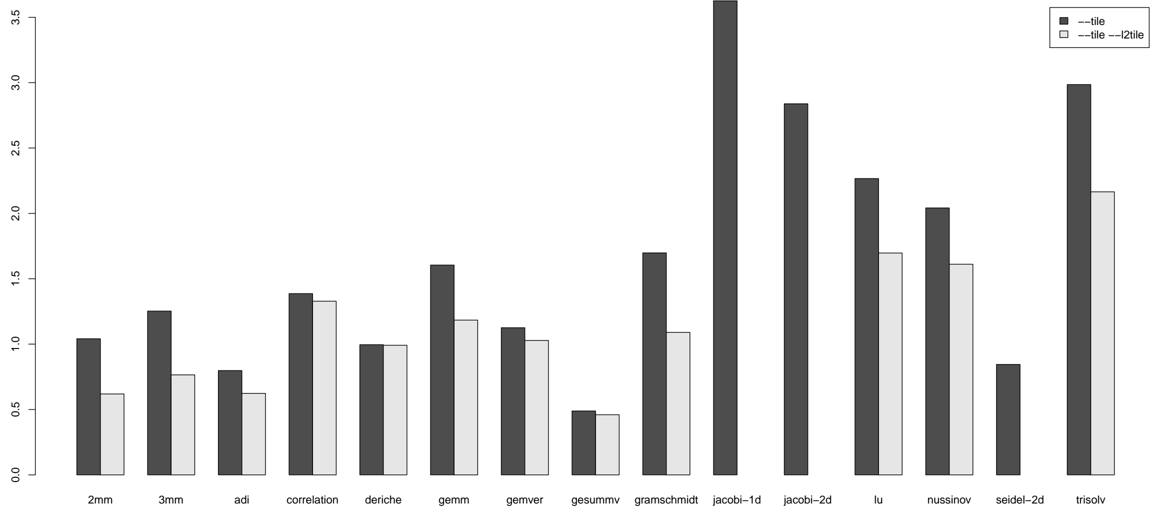

FIGURE 7 – Rapport des temps d'exécution des solutions obtenues sans et avec les options –tile et –tile –l2tile

représentatif de la performance réelle d'une solution. En effet, toutes les itérations externes ne contiennent pas le même nombre d'opérations ou d'itérations des boucles internes. Le nombre (approximatif) d'instructions exécutées permet d'obtenir une mesure plus représentative.

Pour cela, lors de la génération du code d'une boucle parallélisable, le nombre d'instructions (en représentation intermédiaire LLVM) contenues dans le corps de la boucle est mémorisé. Lors de l'exécution de cette boucle, chaque *thread* d'exécution ajoute ce nombre d'instructions à un compteur pour chaque itération exécutée. La somme de ces compteurs permet d'estimer le nombre total d'instructions exécutées.

Ce prototype présente toutefois un surcoût important par rapport à Apollo, dû à de nombreuses opérations coûteuses réalisées au cours de la génération des solutions et de leur exécution.

Conclusion

L'objectif de ce stage était d'étudier l'opportunité d'une approche incrémentale d'utilisation de Pluto pour la parallélisation à la volée de programmes, en vue d'implémenter un système de *poly-ordonnancement*, qui sélectionne des solutions optimisées de bonne qualité tout en minimisant l'impact du temps de compilation, dans la plateforme Apollo.

Pour cela, une étude des temps de compilation et d'exécution des solutions générées par Pluto a été nécessaire afin de définir une stratégie de gestion des priorités entre les solutions. Les premiers tests, réalisés sur les programmes de PolyBench avec un ensemble restreint d'options de Pluto, avaient permis de définir une première stratégie naïve et d'identifier les problèmes inhérents à cette stratégie. Cette stratégie avait été implémentée dans un premier prototype logiciel approximatif, qui mesurait le temps d'exécution effectif des solutions générées par Pluto afin de retenir la plus performante selon ces mesures, sans gestion des priorités entre les solutions.

Par la suite, une nouvelle stratégie a été implémentée dans la plateforme Apollo. Elle introduit cependant un surcoût important, qui pourrait être réduit. Cette stratégie pourrait être améliorée, par exemple en commencant à générer une solution coûteuse mais de haute qualité dès le début de l'exécution lorsqu'au moins deux threads sont disponibles, au lieu de générer les solutions par ordre croissant de coût estimé.

References

[1] *APOLLO, Automatic speculative POLyhedral Loop Optimizer*. URL: `http://apollo.gforge.inria.fr`.

[2] Uday Bondhugula et al. « Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model ». In: *Compiler Construction*. 2008.

[3] Juan Manuel Martinez Caamaño, Willy Wolff, and Philippe Clauss. « Code Bones: Fast and Flexible Code Generation for Dynamic and Speculative Polyhedral Optimization ». In: *Euro-Par 2016*. 2016.

[4] *CLooG, the Chunky Loop Generator*. URL: `http://www.cloog.org`.

[5] Paul Feautrier and Christian Lengauer. « The Polyhedron Model ». In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011.

[6] Martin Griebl, Paul Feautrier, and Christian Lengauer. « Index Set Splitting ». In: *International Journal of Parallel Programming* (2000).

[7] *PLUTO - An automatic parallelizer and locality optimizer for affine loop nests*. URL: `http://pluto-compiler.sourceforge.net`.

[8] *PolyBench/C, the Polyhedral Benchmark suite*. URL: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench`.

[9] Aravind Sukumaran-Rajam. « Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation ». PhD thesis. Université de Strasbourg, France, Nov. 2015.

[10] Jingling Xue. *Loop Tiling for Parallelism*. Springer US, 2000.