Générateur de code multi-temps et optimisation de code multi-objectifs

Victor Lomüller

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Victor LOMÜLLER

Thèse dirigée par **Henri-Pierre CHARLES**

préparée au sein **Laboratoire Infrastructure et Atelier Logiciel pour Puces, CEA Grenoble**
et de **l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"**

Générateur de code multi-temps et optimisation de code multi-objectifs

Thèse soutenue publiquement le ,
devant le jury composé de :

M. Jean-François Méhaut
Professeur à l'Université Joseph Fourier, Président
M. Gaël Thomas
Professeur à Télécom Sud Paris, Rapporteur
M. François Bodin
Professeur à l'Université de Rennes I, Rapporteur
M^me^ Karine Heydemann
Maître de Conférences à l'Université Pierre et Marie Curie, Examinatrice
M. Albert Cohen
Directeur de recherche à l'INRIA, Examinateur
M. Thierry Lepley
Ingénieur recherche senior à Nvidia, Examinateur
M. Ayal Zaks
Ingénieur recherche senior à Intel Israël, Examinateur
M. Henri-Pierre Charles
Directeur de recherche au CEA LIST, Directeur de thèse

Sommaire

Table des figures

Liste des tableaux

Table des listages

Remerciements

Avant de rentrer dans le cœur de cette thèse, fruit de mes 3 ans de travail au CEA. Je tiens à remercier tout particulièrement mon directeur de thèse Henri-Pierre Charles, qui a accepté et cru dans ma capacité à mener à bien cette thèse malgré un profile quelque peu atypique pour un doctorant. Sa patience, ses conseils et sa disponibilité ont été précieux, surtout dans les moments de doutes qui ont parsemé cette thèse.

Je tiens à remercier Monsieur Jean-François Méhaut d'avoir présidé mon jury de thèse.

Je tiens à remercier Messieurs Gaël Thomas et François Bodin d'avoir acceptés d'être mes rapporteurs et membres de mon jury.

Je tiens à remercier Madame Karine Heydemann et Messieurs Albert Cohen, Thierry Lepley et Ayal Zaks d'avoir acceptés d'être membre de mon jury.

Je remercie l'ensemble de l'équipe du LIALP, passé et présent. La très bonne ambiance du laboratoire a rendu ces trois années de thèses prolifique et agréable. Cela a été un très grand plaisir de travailler entouré de cette équipe compétente et passionnée. Je tiens particulièrement à remercier Damien Couroussé et Fernando Endo pour leur très grande aide dans mes travaux.

Je tiens particulièrement à remercier mes collègues et amis rencontrés dans ce laboratoire, merci à vous tous d'avoir été là, de m'avoir soutenu et pour tous les moments passés ensemble. Je remercie tous particulièrement Ivan Llopard pour tous le temps pris pour nos discussions et son aide sur LLVM, gracias !

Je tiens à remercier Bertrand Lejeune pour son éternelle bonne humeur et pour son soutien infaillible, même 13 ans après. Merci à Sandy Saupique pour son aide et de toujours être là, malgré le temps, la distance et mon caractère.

Je souhaite remercier mes grands-parents pour leur grande affection.

Je souhaite enfin terminer en remerciant ma mère et mon frère. Par leurs analyses fines des choses, ils ont su trouver les mots et puiser en moi les ressources pour aller de l'avant dans les moments de doutes. C'est pourquoi je souhaiterais leur dédier cette thèse.

Misaotra

Djarama

תודה

ありがとう

감사합니다

Дякую

teşekkürler

Merci

Thank you

شكرا

Gracias

Mulțumesc

Obrigado

Dieureudieuf

Grazie

Mahalo

Chapitre 1

Introduction

Sommaire

La compilation est l'un des sujets d'étude fondamentale de l'informatique. Apparu avec les langages de haut niveau, et quasiment dès ses origines, elle a été utilisée pour améliorer les performances des applications. Tentant de suivre l'évolution des architectures et des nouveaux usages, la compilation s'est rapidement heurtée à des difficultés à pleinement exploiter les capacités du matériel. Depuis la naissance de l'informatique, nous sommes passés d'un monde réservé à des domaines précis (cryptanalyse avec *Colossus*) à un monde connecté et distribué utilisant des architectures variées et hétérogènes. Les objectifs ont aussi évolué, autrefois centré sur la vitesse d'exécution, ils intègrent maintenant des problématiques de mémoires, de consommations énergétiques ou de portabilité. Autant d'objectifs qui vont mener à des approches différentes.

Les stratégies de génération de code ont progressivement évoluée. À l'origine purement statique, les stratégies emploient aujourd'hui de plus en plus de dynamisme dû à différents besoins :

Portabilité : la segmentation du marché en matière de matériel impose aux concepteurs de fournir des solutions portables de leurs applications. Cela a pour corollaire que lors du développement d'une application on ne sait pas nécessairement sur quel jeu d'instruction cette application sera exécutée.

Performances : il y a un besoin d'avoir des applications rapides, réactives, économes en énergie ou avec une empreinte mémoire acceptable. Pour la vitesse d'exécution, on parle aujourd'hui de *bogue de performance*, l'application est fonctionnelle mais trop lente, entraînant une sensation d'application chancelante. Avec l'explosion du marché des smartphones et celui grandissant de *l'Internet des Objets* (IoT), les besoins en performances incluent aujourd'hui la consommation électrique et l'empreinte mémoire.

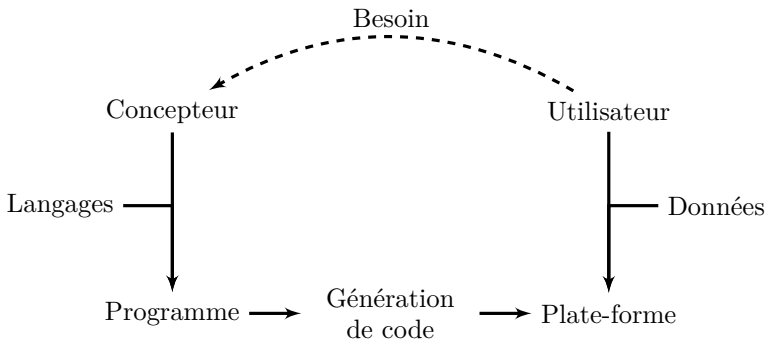


FIGURE 1.1 – Développement d'une application.

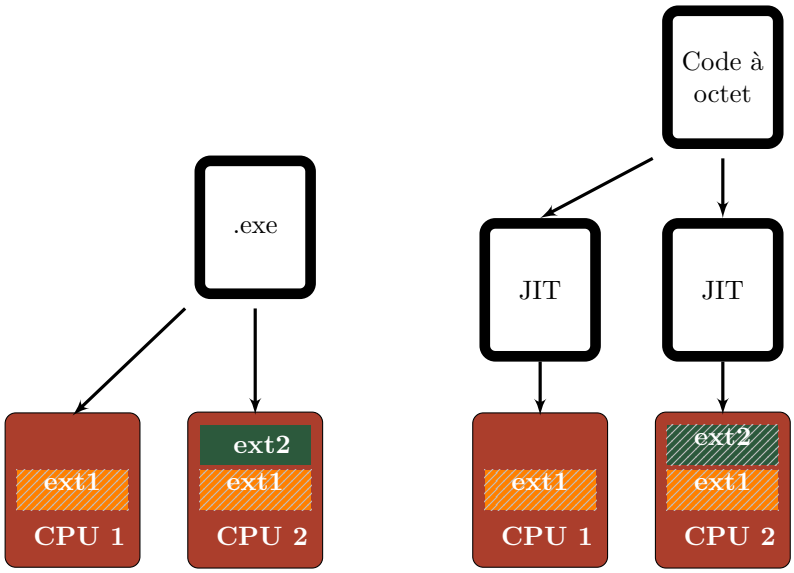
1.1 Contexte et nouveaux challenges

La figure 1.1 présente un cycle de développement d'une application. Pour répondre à un besoin utilisateur, le *concepteur* va créer une solution en utilisant divers langages pour générer une application pour une plate-forme (physique ou virtuelle). L'application sera utilisée par l'utilisateur qui fournira les données sur lesquelles travailler. Cependant, pour obtenir de bonnes performances, se pose la question de la plate-forme cible et la question de la sensibilité aux données.

La question de la plate-forme peut se poser d'un point de vue de la portabilité qui trouve une réponse dans la virtualisation. La question de la performance vis-à-vis de la plate-forme est plus délicate. Quand la plate-forme est parfaitement connue cela autorise l'optimisation et la génération statique du code. Mais la durée de vie d'un logiciel est plus longue que celle du matériel et donc il est fort probable que la plate-forme change pendant la vie de l'application. L'utilisation de technologie de virtualisation peut alors permettre aux applications de suivre les évolutions de la plate-forme [20]. La figure 1.2 illustre l'avantage de l'utilisation de la virtualisation dans le cas du suivi de l'évolution d'une architecture. Dans le premier cas (figure 1.2a), une application est compilée pour le CPU 1, pour lequel le compilateur sait qu'il existe une extension et sera donc en mesure de l'exploiter. Quant au CPU 2, il est de la même famille que le CPU 1 mais propose une extension supplémentaire. Le code ayant été généré de façon statique le programme n'est pas en mesure de l'utiliser. Dans le deuxième cas (figure 1.2b), l'application est compilée dans un code à octet. Le compilateur juste-à-temps présent sur les deux plates-formes a la pleine connaissance des extensions présentes sur les deux CPU, et est donc en mesure de créer un code utilisant les extensions présentes sur la machine.

Néanmoins, l'utilisation de ces techniques de virtualisation ne permet pas garantir les performances et la pleine utilisation du matériel. Si on prend le cas des GPU ("*Graphical Processing Units*", Unité de Traitement Graphique), les constructeurs fournissent des couches de virtualisation afin de garantir une portabilité *fonctionnelle* des bases de code. Par contre les performances obtenues ne seront pas nécessairement à la hauteur des capacités du matériel.

La sensibilité aux données de certains algorithmes a été elle aussi changée avec l'arrivée des architectures massivement multi-cœurs. Le traitement étant découpé en fonction des données, celles-ci influent sur le nombre de processus



(a) Approche purement statique. (b) Approche avec virtualisation.

FIGURE 1.2 – Illustration de l'avantage de la virtualisation dans le suivi de l'évolution d'une architecture.

obtenus, les accès mémoires *etc.*

Enfin, les techniques de virtualisation ne sont utilisables que s'il existe des ressources allouables à ces fins de générations. Ces outils vont prendre une place non négligeable dans le système, consommer de la RAM, des cycles CPU et de l'énergie. Dans le contexte de l'IoT par exemple, ces ressources coûtent cher du fait de leurs raretés. Mais même pour des systèmes moins contraints, il peut être difficile d'utiliser ces techniques à des fins d'adaptation de l'algorithme aux données.

1.2 Contributions

Cette thèse a été développée autour de 2 axes :

Dans le contexte des multi-cœurs : Nous avons étudié le comportement des programmes sur GPU Nvidia, en cherchant à savoir :
- S'il existe une sensibilité aux données des algorithmes, y compris ceux précédemment réputés insensibles dans un contexte mono-cœur ?
- Comment remédier à cette sensibilité ?
- Comment s'adapter tout en autorisant la possibilité de suivre les évolutions ?

Dans le contexte d'architectures contraintes : La réduction des budgets qu'il est possible d'allouer à des systèmes contraints nous a poussés à étudier l'impact de la génération et de la spécialisation de code. Nous avons donc cherché à savoir quels étaient les impacts en matière de gains et coûts sur la vitesse d'exécution, la mémoire et la consommation énergétique d'un noyau de calcul.

1.3 Plan du manuscrit

Le chapitre 2 présentera *un état de l'art de la compilation multi-temps*. Nous présenterons une définition de la compilation multi-temps et nous replacerons les différentes technologies de compilation et de génération de code en fonction de son temps d'action. Cette partie permettra de faire un large balayage sur les technologies développées, les raisons ainsi que les avantages et inconvénients à intervenir sur ces moments.

Le chapitre 3 présentera les *limites de l'état de l'art que nous avons identifiées et qui sont les points de départ des contributions*. Nous verrons le problème de la non prise en compte des données dans l'optimisation des performances pour la GEMM sur GPU. Nous soulèverons aussi les questions sur comment optimiser cet algorithme et la gestion de la taille de la bibliothèque ainsi créée. Nous nous intéresserons aussi aux métriques qui sont le point de départ du second axe de la contribution : l'étude multi-métriques de la génération et spécialisation de code.

Le chapitre 4 présentera les outillages qui ont été développés pour répondre aux questions posées. Dans cette partie il ne sera question que de présenter les outils et la logique sur laquelle ils ont été construits. Les outils présentés sont `deGoal` et `Kahuna`, tous deux présentant une approche spécifique de la génération de code lors de l'exécution. Le premier, `deGoal`, vise à laisser au programmeur un large contrôle sur le code qu'il souhaite générer à l'exécution. Le second, `Kahuna`, vise à faciliter le développement de générateurs de code à l'exécution, indépendants d'un langage d'entrée, portables et faciles à recibler.

Le chapitre 5 présentera les résultats d'une étude sur l'algorithme de la multiplication matricielle sur GPU Nvidia. Dans cette partie nous commencerons par présenter l'implémentation de la GEMM sur GPU et les méthodes d'ajustement des performances. Nous présenterons ensuite une limitation de l'approche vis-à-vis des performances obtenues dans certaines conditions. Il sera montré la méthodologie développée pour répondre à cette limitation suivie de son évaluation.

Le chapitre 6 se concentrera sur une plate-forme embarquée et une étude sur l'optimisation par la spécialisation. Nous étudierons le comportement des outils présentés dans le chapitre 4 ainsi que des approches basées sur le cadriciel LLVM. Dans ce chapitre, nous nous concentrons sur les gains et coûts de plusieurs approches de la génération de code selon plusieurs métriques. L'étude présentera les résultats du point de vue de la vitesse d'exécution mais aussi de la consommation énergétique et de l'utilisation mémoire.

Enfin le chapitre 7 conclura cette thèse. Nous récapitulerons les contributions présentées dans ce manuscrit, les travaux restant à faire. Nous remettrons aussi ces travaux dans un contexte plus global, et présenterons à cette occasion des pistes d'évolutions de ces travaux et la façon dont ils peuvent s'insérer dans les tendances de l'évolution des architectures.

Chapitre 2

État de l'art

Sommaire

L'évolution des technologies et du marché mène à de nombreux changements dans la façon d'appréhender la conception des logiciels. Pour répondre aux besoins croissants en puissance de calcul, les puces sont passées d'une architecture mono-cœur à une architecture multi-cœurs, voire massivement multi-cœurs, soulevant de nouveaux problèmes dans la façon de les programmer. L'explosion du marché des systèmes embarqués grand public pousse les fabricants vers des architectures devant allier puissance de calcul, consommation électrique faible et bas coût.

Comme nous l'avons évoqué dans le chapitre précédent, le support et l'optimisation des applications sur les architectures reposent en partie sur les compilateurs et la génération efficace du code. Dans ce chapitre, nous présenterons l'état de l'art de la génération de code multi-temps. Le but est de balayer les

différentes techniques de compilation et génération de code en fonction de quand ils interviennent dans la vie de l'application. Nous commencerons par définir la notion de compilation et de temps de compilation. Nous introduirons aussi les représentations intermédiaires qui est un point important dans la compilation et l'optimisation. Nous présenterons ensuite deux architectures distinctes par leurs objectifs et que nous utiliserons dans la suite de ce manuscrit. Nous ferons ensuite un tour des différentes techniques utilisées dans l'état de l'art pour décrire, transformer, optimiser et faire évoluer les applications.

2.1 Les temps de compilations

2.1.1 Introduction à la compilation et transformation de code

Aho et al. [3] définissent de façon générique le compilateur comme étant *"un programme lisant un programme dans un langage, le langage source, et le traduit en un programme équivalant dans un autre langage, le langage cible"*. Bien qu'étant large et simple, elle englobe bien le spectre des différentes méthodes existantes.

Le premier compilateur est apparu dans les années 50 avec le *A-0 System* développé par Grace Hopper pour l'UNIVAC [89]. Ce compilateur n'effectuait que des opérations d'assemblage de routines qu'il avait préalablement chargé. À la fin des années 50, le premier compilateur complet et optimisant pour le langage FORTRAN apparut permettant ainsi la création de programmes exécutables sans le passage par la représentation assembleur. Au fil des années, les compilateurs ont continué à se développer en parallèle des langages, levant de nouvelles problématiques d'optimisations à mesure que les langages deviennent de plus haut niveau. La conception de ces compilateurs a aussi évolué vers une plus grande modularité et une architecture plus ou moins commune aujourd'hui.

Les compilateurs modernes sont le plus souvent architecturés autour de 3 étages (illustré dans la figure 2.1) :

Partie frontale (*front-end*) : étage chargé de la lecture du langage source, du rapport des erreurs, d'optimisations spécifiques au langage et de la traduction vers la représentation intermédiaire (RI) de l'étage suivant. On pourra aussi désigner cette partie "*le frontal*" (du compilateur).

Partie intermédiaire (*middle-end*) : étage chargé des analyses et des transformations indépendantes du langage cible du programme dans la RI.

Partie arrière (*back-end*) : étage chargé des optimisations spécifiques au langage cible et de l'émission du programme dans le langage cible.

L'avantage de ce découpage, et notamment du passage par la RI, est qu'il permet de déconnecter le langage source du langage cible. Ainsi lors de l'ajout du support d'un nouveau langage source ou cible seul l'étage de la partie frontale, respectivement de la partie arrière, est à écrire.

La *partie frontale* construit une représentation du programme d'entrée dans une forme proche du langage pour s'assurer que le programme est bien formé (respect de la syntaxe et de la grammaire). Une fois cette phase de vérifications effectuée il peut éventuellement effectuer des transformations propres au langage ou au domaine. Enfin une phase de *traduction* ("*lowering*" dans le vocabulaire

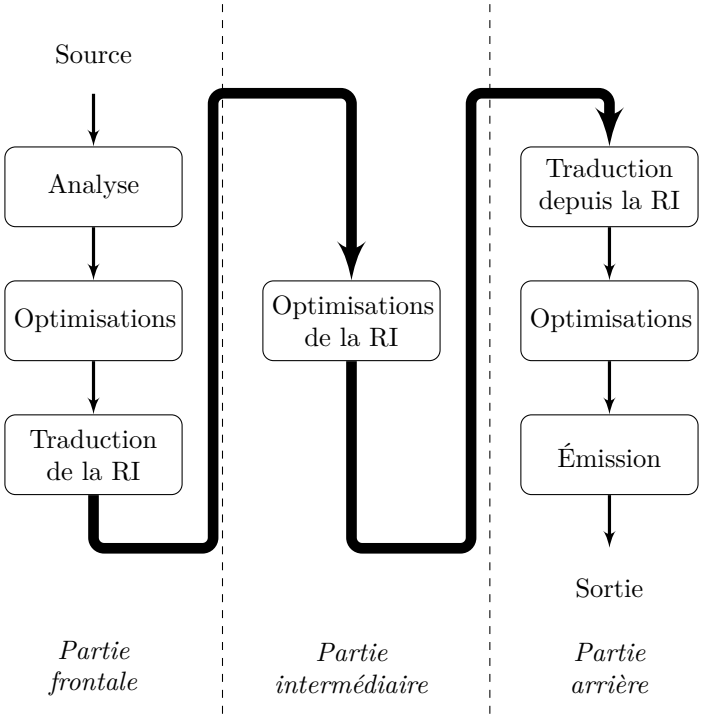


FIGURE 2.1 – Étages d'un compilateur.

de LLVM) permet de générer la représentation intermédiaire du programme dans celle de la partie intermédiaire du compilateur. Par *traduction*, on désigne le passage d'une représentation intermédiaire à une autre, normalement plus proche de la cible.

La *partie intermédiaire* orchestre les phases d'optimisations indépendantes des cibles. Durant ces phases, d'autres représentations peuvent être utilisées selon les besoins. Dans la pratique, des optimisations spécifiques à des cibles ou familles de cibles peuvent aussi être lancées à ce niveau de façon à pouvoir tirer parti de ses représentations intermédiaires.

La *partie arrière* va être responsable de la traduction du programme vers son langage. Cette étape effectue l'émission du code dans le langage de destination. Dans le cas de l'émission dans un langage machine, la partie arrière effectue une sélection d'instructions, l'allocation des registres, l'ordonnancement et des optimisations spécifiques à la cible comme le *peephole optimisation*, *if-conversion*, *strengh reduction etc.*

En règle générale, les compilateurs modernes vont utiliser plusieurs représentations intermédiaires suivant les opérations à effectuer (voir section 2.1.3). Par exemple, *LLVM* [51] utilise comme représentation intermédiaire *LLVA* [2] pour sa partie intermédiaire, le SDAG, MI (Machine Instruction) et MC (Machine Code) dans sa partie arrière. Le greffon *Polly* [40], qui effectue des transformations polyédriques dans *LLVM*, intervient dans la partie intermédiaire et utilise aussi plusieurs autres représentations du programme pour communiquer avec *CLooG*, l'outil d'optimisation polyédrique.

Selon la nature du langage source et cible, on peut désigner différents types de

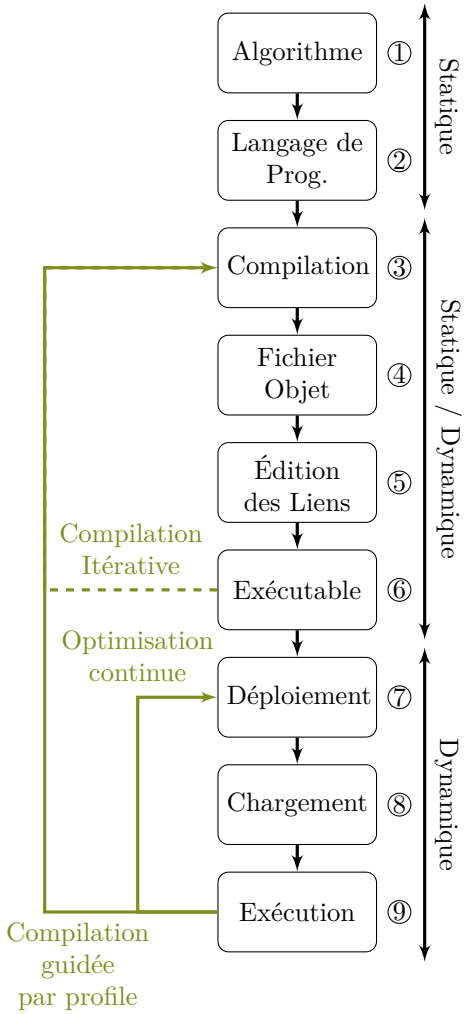


FIGURE 2.2 – Cycle de "vie" généraliste d'une application.

compilateurs que l'on pourra replacer dans les différents moments dans lesquels ils interviennent.

2.1.2 Introduction aux temps de compilations

Un *temps de compilation* est un moment dans le cycle de vie d'une application au cours duquel la connaissance de l'environnent se précise. Cette nouvelle connaissance crée des opportunités permettant d'optimiser les performances de l'application (vitesse d'exécution, utilisation mémoire ou encore consommation énergétique). Par exemple, lors de l'installation d'une application sur une plateforme, le programme a alors accès aux particularités matérielles de celle-ci. Cela peut aller de la connaissance de la taille du cache de données à la connaissance du jeu d'instruction (ISA, pour Instruction Set Architecture). Il est entendu que selon le type application certaines informations sont connues plus ou moins tôt

dans le cycle de vie.

La figure 2.2 présente de façon générique les différents moments de la vie d'une application. On peut la décomposer en 3 grands groupes :

Conception et écriture : Moment initial de la vie du programme. Il est pensé à un niveau fonctionnel puis algorithmique et enfin implémenté (moments 1 et 2).

Génération : Moment où des transformations sont effectuées sur le programme à partir de son implémentation en vue de générer un objet exploitable par une machine (moments 3 à 7).

Réalisation : Moment final où les instructions sont exécutées avec les données de l'utilisateur (moments 8 et 9).

Ces groupes ne constituent pas une séquence linéaire de moments et d'étapes de la conception à l'exécution du programme, c'est-à-dire qu'un groupe peut bénéficier d'informations tirées des autres groupes.

La *compilation itérative* (flèche verte pointillée de la figure 2.2) réutilise les informations accumulées lors de la compilation pour recompiler l'application. Les heuristiques servant lors de la compilation, comme l'estimation d'un facteur de déroulage, sont remplacés par ces informations, pouvant donner lieu à un code plus optimisé. Par exemple, pour dérouler partiellement une boucle, il faut connaître la taille du corps de cette boucle. À la première compilation, cette taille n'est pas connue car le code n'a pas encore été généré. Lors de la seconde itération, on peut réutiliser cette information pour dérouler de façon plus optimale cette boucle. Néanmoins, pour la dérouler de façon optimale, il nous faut, entre autre, connaître la taille du cache d'instruction, information souvent inconnue lors de la compilation statique.

La *compilation guidée par profile* ou *l'optimisation continue* (flèches vertes de la figure 2.2) sont 2 exemples de lien entre réalisation et génération. Dans le premier cas, après la fabrication de l'application, une phase de profilage est effectuée en utilisant un ensemble de tests. Les résultats du profilage vont permettre de relancer une phase de compilation et remplacer les heuristiques par ces résultats. Le problème de cette approche est que le résultat est un binaire invariant, c'est-à-dire qu'il sera dans l'impossibilité de suivre les évolutions de l'environnement et la façon dont l'application sera utilisée. Le deuxième cas effectue un profilage lors de l'exécution de l'application en condition réelle. Ces résultats sont réutilisés pour optimiser comme précédemment l'application et l'altère de façon à ce que la prochaine exécution puisse tirer parti des opportunités de la précédente exécution. L'inconvénient majeur de cette approche est celui de la sécurité : l'application va être amenée à être modifiée pour enregistrer les optimisations, ce qui n'est pas toujours possible selon les systèmes.

Ces groupes s'insèrent dans deux grandes phases temporelles :

Statique : Phase amont de l'installation pendant laquelle se déroule la **Conception et l'écriture**. Durant cette phase les connaissances sur les données et éventuellement la cible sont extrêmement limitées.

Dynamique : Phase où l'application va être déployée (installée) sur le système et exécutée. On peut la décomposer en 3 étapes :

- **Déploiement :** L'application est installée, on a alors une connaissance plus précise de l'environnement d'exécution.

Chargement : L'application est chargée en mémoire et liée à l'environnement (bibliothèques partagées ou variables d'environnements par exemple).

Exécution : L'application s'exécute avec les données à traiter.

La phase de **Génération** se déroule soit en totalité dans la phase *statique*, soit dans la phase *dynamique*, soit "étagée" entre les deux. Dans le cas d'un programme C, en règle générale la totalité de la génération est faite pendant la phase *statique*. Dans le cas de Python, la génération est faite au moment de *l'exécution*. Enfin dans le cas de Java, le code est transformé en code à octet pendant la phase *statique* et interprété ou transformé pendant *l'exécution*.

2.1.3 Introduction aux représentations intermédiaires

Pour effectuer ces optimisations, un compilateur va utiliser une ou plusieurs *représentations intermédiaires* (RI). Dans un compilateur moderne utilise une représentation quel que soit l'étage de la compilation (figure 2.1). Les RI sont un des points centraux dans l'optimisation et la génération de code. Ils permettent la manipulation du programme et assurent la communication entres les différentes phases de transformations. Il existe un grand nombre de représentations, chacune ayant leurs avantages et inconvénients [82].

Stanier et al. [82] classifient les RI en 3 groupes :

Linéaire : qui regroupe les représentations sous forme de pseudo-code, pouvant aller d'une représentation haut-niveau à du pseudo-assembleur.

Graphe cyclique : qui regroupe les représentations sous forme de graphe autorisant les cycles.

Graphe acyclique : qui regroupe les représentations sous forme de graphe interdisant les cycles.

Ces RI peuvent mettre en exergue des dépendances sur les données ou le contrôle. Enfin certaines représentations ne représentent pas l'ensemble du programme mais présentent des points d'intérêts en complément d'une autre représentation. La suite de cette section présente les RI qui sont manipulées pour les outils présentés et utilisés dans les chapitres suivant.

2.1.3.1 Représentations linéaires

Code à 3 adresses : Cette représentation est une suite d'instructions avec un des opérandes qui est défini par l'opérateur. Par exemple, l'expression $(a + b) * (b + a) * (c + d)$ pourrait être représentée de cette façon :

$$
\begin{aligned}
t_1 &\leftarrow a + b \\
t_2 &\leftarrow b + a \\
t_3 &\leftarrow c + d \\
t_4 &\leftarrow t_1 \times t_2 \\
t_4 &\leftarrow t_4 \times t_3
\end{aligned}
$$

Cette représentation est très usitée dans les compilateurs modernes mélangée à des représentations graphiques. L'avantage de cette représentation est sa lisibilité, mais surtout elle permet la modularité des compilateurs modernes en

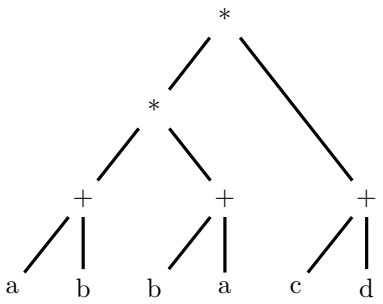


FIGURE 2.3 – Exemple d'AST : représentation de l'expression $(a+b)*(b+a)*(c+d)$.

autorisant une séparation nette entre les phases d'optimisations [82]. On peut citer les compilateurs LLVM [51] ou GCC [62] parmi les utilisateurs de ce type de représentation.

SSA : La forme *SSA* [23] ("Static Single Assignment") n'est pas une représentation intermédiaire en soit, mais une forme respectant certaines propriétés. Un programme dans une forme SSA garantit qu'une variable n'a qu'une seule définition et que toute utilisation est dominée par sa définition. Dans l'exemple précédent t_4 est défini à 2 fois, une forme correcte serait celle-ci :

$$t_4 \leftarrow t_1 \times t_2$$
$$t_5 \leftarrow t_4 \times t_3$$

Cette forme simplifie bon nombre d'optimisations, la plupart n'ont en effet pas besoin de connaître la valeur de l'opérande, mais où elle est définie [4]. Utilisée avec des représentations graphiques dérivées, cette forme est devenue un pilier de la compilation moderne à la fois dans les parties intermédiaires des compilateurs pour les optimisations génériques et pour les parties arrières qui peuvent aussi en tirer certains avantages, notamment pour l'allocation de registre [42].

2.1.3.2 Représentations sous forme de graphe

Arbres de syntaxe abstraite : Les *arbres de syntaxe abstraite*, ou *AST* ("Abstract Syntax Tree"), sont des représentations sous forme d'arbre (un graphe acyclique donc) utilisées par les parties frontales pour traiter leurs langages d'entrée. Les AST sont employés pour s'assurer de la bonne construction du programme : respect du système de typage, définition des variables *etc.* Cette représentation est en général très liée au langage d'entrée, certaines optimisations propres au langage peuvent être faites à ce niveau. Cependant cette représentation est assez pauvre et peu adaptées à la modularité des compilateurs modernes qui cherchent à se détacher du langage d'entrée pour les étages suivants. Pour cette raison, l'AST est finalement transformé dans une RI de la partie intermédiaire du compilateur mieux adaptée aux optimisations génériques

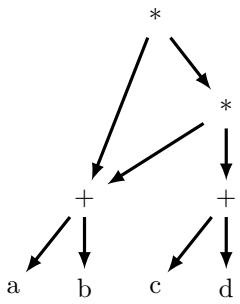


FIGURE 2.4 – Exemple de graphe orienté acyclique : représentation de l'expression $(a + b) * (b + a) * (c + d)$.

et indépendantes du langage d'entrée. Pour illustration, une représentation possible de l'expression arithmétique $(a + b) * (b + a) * (c + d)$ dans un AST est présentée dans la figure 2.3.

Graphes orientés acycliques : Un graphe orienté acyclique, ou *DAG* ("Directed Acyclic Graph"), est une représentation sous forme de graphe acyclique permettant de représenter la dépendance des données. Elle est assez similaire aux AST, mais permet de représenter les calculs redondants grâce à des noeuds qui ont plusieurs parents. La figure 2.4 représente la même expression que la figure 2.3, mais a regroupé les deux expressions $(a + b)$. Dans le compilateur LLVM [51], cette représentation est utilisée pour la sélection d'instructions, dans FFTW [32] pour la représentation du calcul de la FFT.

Graphes de flot de contrôle : Un graphe de flot de contrôle, ou *CFG* ("Control Flot Graph"), est une représentation sous forme de graphe cyclique permettant de représenter les dépendances de contrôle d'un programme.

Chaînes de définition-utilisations, utilisation-définitions et graphes SSA : Les chaînes de définition-utilisations (def-use) connectent une définition à toutes les opérations l'utilisant et qu'il peut atteindre, portant ainsi les informations du flot de donnée. Les chaînes utilisation-définitions (use-def) font les connections inverses en connectant une variable à toutes les définitions qui peuvent l'atteindre.

La forme SSA permet de simplifier grandement ces chaînes car une variable n'est défini qu'une seule fois, la chaîne de use-def est donc le lien unique entre une utilisation et sa définition. À partir de cette propriété, on peut construire le graphe SSA en suivant les chaînes de use-def.

2.2 Architectures

Avec l'évolution des usages des outils du numérique et les contraintes matérielles, les architectures ont fortement évolué pour répondre aux nouveaux challenges :

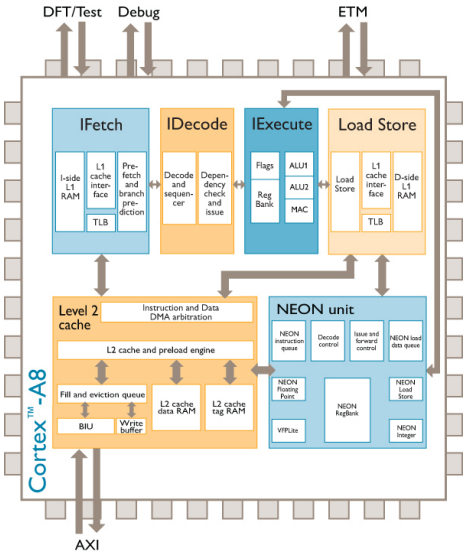


FIGURE 2.5 – Organisation d'un Cortex-A8.

— L'essor des applications embarqués, poussé par le marché des smartphones et des objets connectées.
— Besoin de puissance de calcul toujours croissante alors même que les architectures classiques ont atteint les limites en termes de fréquence de fonctionnement.
— Besoin de performances à moindre coût énergétique en cherchant à obtenir le meilleur rapport opérations par second et par Watt, là où autrefois seul le nombre d'opérations par second comptait.

Cette section présente deux architectures radicalement différentes dans l'objectif visé et qui sont utilisé par la suite. L'architecture du Cortex-A8 de ARM vise les applications embraquées type smartphone ou nœud de calcul basse consommation. Cette architecture cherche donc à allier puissance de calcul et consommation réduite. La seconde architecture est celle des GPU Fermi de Nvidia, seconde génération des cartes CUDA, qui elle vise les très hautes performances de calcul avec notamment en marché cible les systèmes HPC. On peut noter néanmoins que la préoccupation énergétique devient de plus en plus présente dans les systèmes HPC à cause notamment du passage à l'exascale (des systèmes capables d'exécuter 10^{12} opérations flottantes à la seconde).

2.2.1 Architecture Cortex-A8

Les processeurs ARM sont des processeurs RISC dont l'utilisation a très fortement augmenté avec l'arrivée des smartphones. D'une façon générale, ces processeurs ont à disposition 16 registres généraux au niveau utilisateur, des unités de chargement, de stockage et des unités arithmétiques. Des extensions existent pour le traitement des nombres flottants (VFP, NEON), des vecteurs (NEON), Java (Jazelle) *etc.* Deux jeux d'instructions existent pour ces architectures, le jeu d'instruction ARM sur 32 bits, et Thumb-1 sur 16 bits et son extension Thumb-2 sur 16 ou 32 bits. Le jeu d'instruction Thumb a été créé

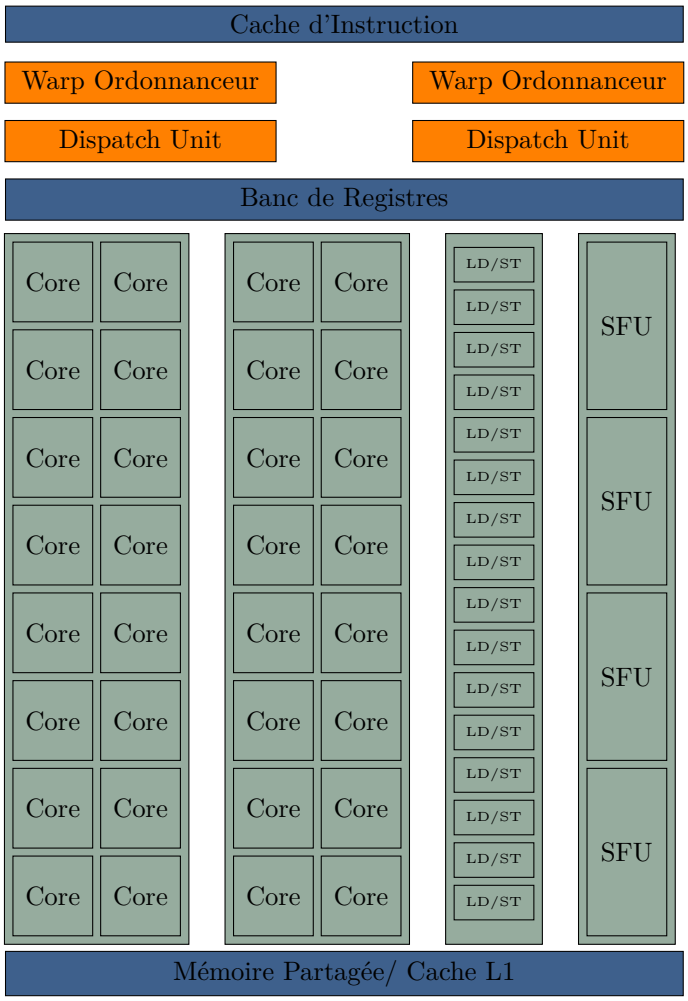


FIGURE 2.6 – Organisation d'un Streaming Multiprocessor (SM) dans l'architecture Fermi.

pour augmenter la densité de code et permet d'améliorer les performances en cas d'utilisation sur un système utilisant un bus mémoire de 16 bits.

Le Cortex-A8 est un processeur superscalaire exécutant les instructions dans l'ordre et pouvant adresser jusqu'à 2 instructions en simultané. L'organisation haut niveau du processeur est présenté dans la figure 2.5. Il implémente le jeu d'instruction armv7, avec notamment les extensions VFP3 (dans une version non pipelinée) et NEON pour le support des flottants et instructions SIMD.

2.2.2 Architecture des GPU CUDA

Les GPU Nvidia [69] sont organisés autour de plusieurs multiprocesseurs appelés "*Streaming Multiprocessors*" (SM) dont l'organisation interne est schématisée dans la figure 2.6. Ces SM utilisent une architecture dite "*Single Instruction Multiple Threads*" ou *SIMT* leur permettant de gérer et de manipuler

un grand nombre de processus (au sens processus léger ou "thread" en anglais). Les processus d'un bloc sont exécutés de manière concurrente dans un SM. Les processus d'un SM sont créés, managés, ordonnancés et exécutés par groupes de 32 processus appelés *"warp"*. Les processus d'un warp commencent ensemble à la même adresse mémoire, mais possèdent leurs propres registres d'états et pointeurs d'instructions. Les processus peuvent donc brancher et exécuter de façon indépendante. Néanmoins, les instructions sont émises au niveau du warp, c'est-à-dire que si un ou plusieurs processus du warp a un pointeur d'instruction différent de celui utilisé pour émettre l'instruction ce ou ces processus sont déactivés pour cette exécution, donnant lieu à des performances sous-optimales.

Dans cette thèse nous avons travaillé sur des GPU de la génération *Fermi*. Chaque SM de cette génération possède :

- 32 cœurs CUDA (unités de calcul de base), chacun implémentant une unité de calcul arithmétique et flottante complètement pipeliné. L'exécution se fait dans l'ordre et il n'existe aucune prédiction de branches ou d'exécution spéculative des instructions. Un SM peut lancer 2 warps en parallèle, ainsi 16 cœurs CUDA sont utilisés pour exécuter un warp.
- 16 unités de chargement et d'écriture ("LD/ST").
- 4 unités de calcul pour les fonctions spéciales (SFU, "*Special Function Units*"), tels que sinus, cosinus, racine carrée etc.

Un SM possède son propre banc de registres, une mémoire locale, un cache L1, un cache d'instruction et une mémoire partagée.

Les cœurs d'exécution CUDA ont une différence majeure par rapport à des cœurs CPU classique qui est une latence plus importante. Contrairement aux CPU, les GPU ne sont pas conçus pour optimiser cette latence mais pour maximiser le débit (*throughput*). Cette latence plus importante peut être cachée en utilisant le parallélisme au niveau instruction (ILP, pour *"Instruction Level Parallelism"*), comme sur les CPU, mais aussi par la capacité du GPU à changer de processus de façon efficace.

2.3 Temps de compilation et optimisations

2.3.1 Compilation statique : approche traditionnelle

Par *compilateur statique* on désigne les compilateurs qui effectuent leurs opérations lors de la phase statique. De part leurs positions dans le temps, leurs connaissances sur l'environnement d'exécution sont généralement limitées. Dans le cas le plus courant pour les compilateurs visant un langage machine seule l'ISA est connue mais la version précise ne l'est souvent pas, ce qui limite les possibilités de la pleine utilisation du jeu d'instruction, comme les instructions AVX sur x86 par exemple. Cette section s'attardera sur deux compilateurs statiques très repandus, et servant généralement de pierre angulaire à de nouvelles techniques.

2.3.1.1 LLVM : Low Level Virtual Machine

LLVM [51] est un cadriciel écrit en C++ pour la construction de machines virtuelles, de compilateurs statiques... LLVM permet de cibler plusieurs architectures comme x86, ARM ou MIPS. Il est construit comme une bibliothèque pour construire des compilateurs, il ne gère donc pas lui-même la partie frontale des compilateurs.

À l'origine du projet, le but était de pouvoir construire des machines virtuelles en utilisant cette infrastructure. Le projet a évolué au fil du temps vers un cadriciel de compilation plus classique. Ce cadriciel peut être utilisé dans des systèmes de compilation statique classique ou dans des systèmes de compilation juste-à-temps (JIT).

Représentations intermédiaires : LLVM utilise *LLVA* [2] (Low Level Virtual instruction set Architecture) comme représentation intermédiaire pour les optimisations de la partie intermédiaire. LLVA est une ISA, langage et architecture agnostique utilisant des registres typés et sous la forme SSA. Le modèle mémoire est partagé entre la pile et la mémoire globale pouvant résider dans différents espaces. La mémoire est allouée de façon explicite et les accès à la mémoire ne peuvent se faire que par les instructions *store* et *load*. Un aspect intéressant de l'ISA est dans la gestion de l'arithmétique des pointeurs. LLVA utilise une instruction spécifique, *getelementptr* (ou *GEP*), pour permettre les accès à des emplacements mémoires sans à avoir à penser à la taille des données (la cible n'étant pas forcément connue, la taille des données ne l'est pas non plus).

Au niveau de la partie arrière, LLVM va utiliser 3 autres représentations, adaptées aux opérations de chaque phase :

SDAG : Le code LLVA sous forme de DAG pour la sélection d'instructions.

MI : Une RI linéaire représentant les instructions machines et quelques informations de haut niveau comme les fonctions ou les blocs basiques. Cette représentation permet d'effectuer diverses optimisations spécifiques à la cible de façons similaires à ce qui peut être fait dans la partie intermédiaire. C'est dans cette représentation qu'est fait l'ordonnancement des instructions et l'allocation de registre.

MC : Une RI linéaire représentant les instructions machines encodées mais les informations haut-niveaux ont disparu. Cette représentation sert juste à émettre les instructions.

Écosystème LLVM : La figure 2.7 représente l'écosystème du cadriciel *LLVM*. Ce cadriciel inclut les deux derniers étages d'un compilateur (parties intermédiaires et arrières, voir figure 2.1). Ce cadriciel est utilisé par une multitude d'outils :

opt : un outil permettant d'orchestrer des passes d'optimisation sur le code à octet LLVA.

llc : un outil permettant la compilation statique d'application en LLVA.

lli : un outil permettant l'exécution juste-à-temps d'un programme LLVA (voir section 2.3.5).

lld : un outil permettant l'édition des liens.

clang : une partie frontale pour les langages de la famille du C (C / C++ / Objective-C / OpenCL. . .)

D'autres outils s'ajoutent pour former les outils classiques de la compilation comme les déassembleurs.

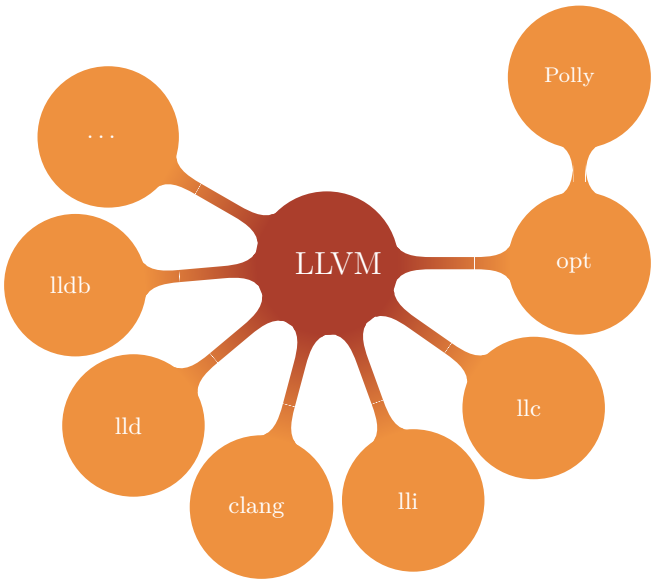


FIGURE 2.7 – Écosystème LLVM.

Cet outil est la base de nombreux compilateurs ou systèmes ayant recours à la génération de code. Parmi les compilateurs statiques, on peut notamment citer *nvcc* [68] pour la compilation du CUDA C, ou le SDK d'Intel pour OpenCL [45]. Toujours dans le domaine d'OpenCL, LLVA est par ailleurs le support de *SPIR* [47], une représentation intermédiaire portable destinée à devenir la représentation intermédiaire standard pour les programmes OpenCL.

2.3.1.2 GCC : GNU Compiler Collection

GCC est un compilateur gérant de nombreux langages comme le C/C++, l'ADA, le FORTRAN ou encore le Java et pouvant cibler de nombreuses architectures comme l'x86/x86-64, ARM ou PowerPC. Depuis ses débuts, fin des années 80, il s'est établi comme le compilateur statique de référence. Depuis sa version 4.5 il est aussi possible d'ajouter des passes d'optimisation sous forme de greffons. Parmi les multiples greffons existant, on peut citer *DragonEgg* utilisé comme partie arrière pour émettre le programme en LLVA et ainsi bénéficier des parties frontales de GCC pour LLVM.

Représentations intermédiaires : Historiquement GCC était basé sur 2 représentations, une sous forme d'arbre (*tree*) et dépendant du langage et une forme indépendante du langage proche de la machine appelé *RTL* ("*Register Transfert Language*"). La version actuelle de GCC se structure autour de 3 représentations [62] :

GENERIC : une représentation haut niveau mais indépendant des langages d'entrée. C'est la représentation de l'AST de la partie frontale débarrassé de toutes constructions spécifiques à l'entrée.

GIMPLE : la représentation intermédiaire pour la partie intermédiaire. GIMPLE est en réalité la représentation GENERIC avec des contraintes

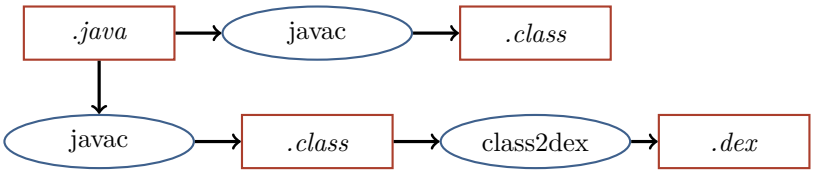


FIGURE 2.8 – Processus de compilation statique d'un programme Java (chemin haut) et Android (chemin bas).

sur sa structure, comme le passage en un code à 3 adresses et sous une forme SSA.

RTL : une représentation proche du langage lisp conçu pour la partie arrière.

Écosystème : GCC inclut un ensemble d'outils pour la génération et la manipulation des objets et des applications. Beaucoup sont devenus aujourd'hui des standards *de facto*. Parmi ces outils, on peut notamment citer les nombreuses parties frontales : *C*, *C++*, *ObjC*, *java*, *GO*, *Ada*. D'autres outils s'ajoutent pour former les outils classiques de la compilation, notamment les outils de manipulation des fichiers objets.

2.3.1.3 javac : Java Compiler

Les programmes écrient en Java sont compilés de façon statique mais le résultat produit est un code à octet portable [48], contrairement à des langages comme C dont la compilation est généralement liée à une architecture. L'outil *javac* (Java compiler) est un compilateur statique qui produit en résultat le programme dans une représentation intermédiaire. Le programme est stocké dans un format binaire appelé de façon générique "code à octet" (fichiers *.class*).

Les programmes Android sont écrits en Java et la compilation statique suit le même processus que pour un programme Java classique pour former les fichiers .class. Une fois le .class obtenu, ce code à octet est transformé en un code à octet pour la machine virtuelle *Dalvik* spécifique à l'environnement Android (fichiers DEX).

2.3.2 Compilation statique : approche source à source

Les *compilateurs source à source* sont les compilateurs ayant en langage cible un langage de haut niveau et qui effectue ces opérations lors de la phase statique. Contrairement à la compilation traditionnelle, ces compilateurs vont récrire le programme dans un langage de haut-niveau (possiblement le même). L'objectif recherché est généralement celui de la portabilité ou celui de la performance. En ciblant un langage de haut-niveau, ces compilateurs ne vont pas contraindre l'utilisation d'un compilateur spécifique lors de la génération du code machine, ce qui permet d'avoir une certaine portabilité.

2.3.2.1 ΣC

ΣC [36] est un langage de haut niveau visant la programmation d'architecture multi-cœurs. Ce langage est un langage dataflow conçue comme une

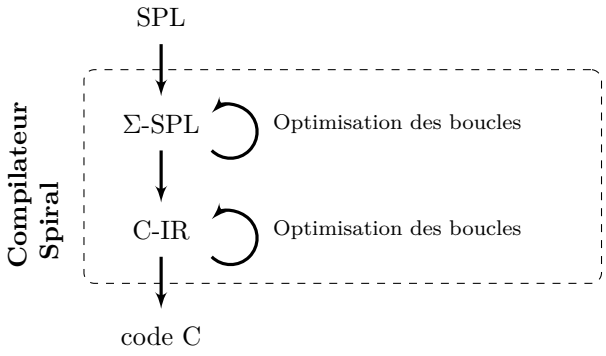


FIGURE 2.9 – Processus de compilation de Spiral.

extension du C. Dans ce langage, le programmeur va définir des agents ayant leurs propres processus ainsi que leurs entrées et sorties propres. Le compilateur réécrit ensuite ce code en programme C avec notamment :

— Le code métier des agents.
— L'instanciation des agents et les connections entre eux.
— Allocation de ressources.

2.3.2.2 Transformations polyédriques

Les transformations polyédriques sont des transformations permettant l'optimisation de boucles répondant à certains critères. Certains outils ont pour entrée et sortie du code C, la sortie ayant été optimisée et ordonnancée grâce à ces transformations.

Polly [40] et Graphite [86] permettent d'effectuer des transformations polyédrique directement dans les représentations intermédiaires de LLVM et GCC respectivement. Ces outils vont transformer la RI de leur compilateur dans celle de l'outil CLooG qui va optimiser le programme. Une fois les optimisations effectuées le programme est réinjecté dans le flot de la compilation.

2.3.2.3 Cas des Domain Specific Language

Spiral [76, 75] et FFTW [32, 31] sont deux bibliothèques de traitement du signal. FFTW est une bibliothèque dédiée à la transformée de Fourier, Spiral lui permet la génération arbitraire de transformées.

Pour créer des routines efficaces, FFTW manipule un petit DSL (Domain Specific Language) avec l'outil `genfft` qui lui permet de générer un certain nombre de routines, appelé "*codelet*", pour gérer au mieux les différents cas d'entrées possibles. Le code de la FFT est produit sous forme de *DAG* par genfft, puis simplifié et ordonnancé pour finir par être écrit en C ("unparse"). Ce code C est ensuite compilé de façon classique.

Spiral utilise un DSL appelé "*Signal Processing Language*" (SPL) [75, 73]. Le DSL SPL est exprimé en Scala, un langage fonctionnel, qui permet notamment une exécution symbolique des expressions. Le processus de compilation est décomposé en 2 étapes avec leurs représentations respectives : Σ-SPL puis en

```
1 extern int foo1(void);
2 extern void foo2(void);
3 extern void foo4(void);
```

LISTAGE 2.1 – Fichier d'en-tête.

```
1  #include "a.h"
2  static signed int i = 0;
3
4  void foo2(void) {
5    i = -1;
6  }
7
8  static int foo3() {
9    foo4();
10   return 10;
11 }
12
13 int foo1(void) {
14   int data = 0;
15   if (i < 0)
16     data = foo3();
17   data = data + 42;
18   return data;
19 }
```

LISTAGE 2.2 – Fichier C.

```
1  #include <stdio.h>
2  #include "a.h"
3
4  void foo4(void) {
5    printf("Hi\n");
6  }
7
8  int main() {
9    return foo1();
10 }
```

LISTAGE 2.3 – Fichier C avec le point d'entrée du programme.

```
1 int main() {
2   return 42;
3 }
```

LISTAGE 2.4 – Programme C équivalant après la passe LTO.

C-IR. Ces différents niveaux de représentations permettent l'optimisation des expressions et surtout tous les découpages et agencements possibles pour les boucles (fusion, fission, déroulage partiel, *etc.*). À la fin du processus de compilation, une routine C est générée, qui pourra ensuite être compilée par un compilateur standard.

2.3.3 Édition des liens

La compilation par module provoque un partitionnement et une isolation de la représentation du programme, interdisant certaines optimisations. Au moment de l'édition des liens tout le programme est connu ce qui lève de nouvelles opportunités d'optimisations. On appelle ces optimisations de ce moment *LTO* ("Link Time Optimisation").

Les compilateurs statiques GCC et LLVM possèdent tous deux une phase d'optimisation au moment de l'édition des liens et fonctionnent de façon similaire. GCC va sauvegarder la représentation GIMPLE des fonctions dans le fichier objet produit avec le code objet. Pour LLVM, c'est le code à octet LLVA

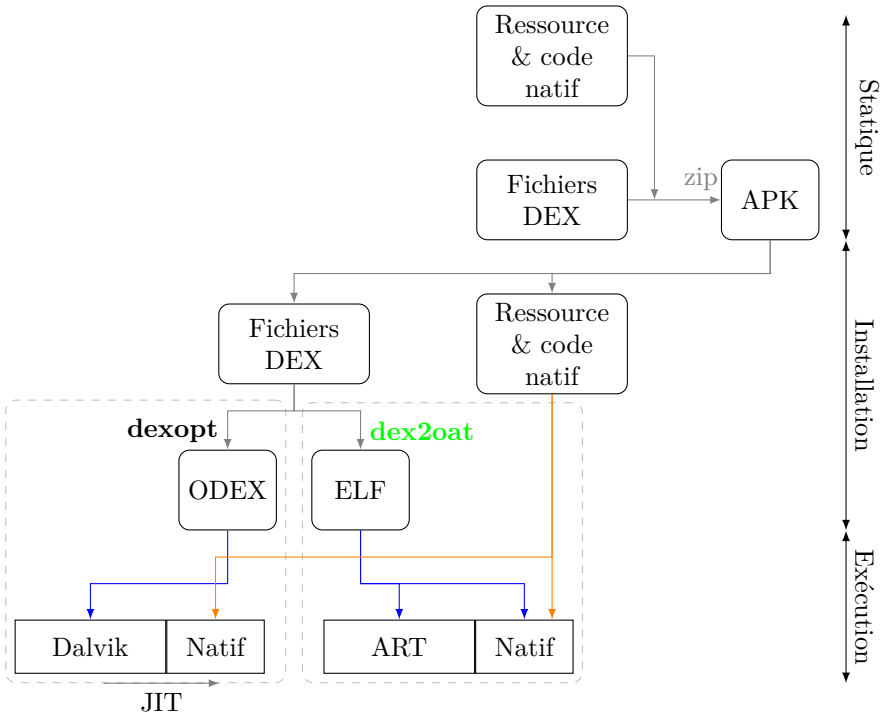


FIGURE 2.10 – Cycle de vie d'une application avec une plate-forme Android 4.4.

qui est sauvegardé.

Les listages 2.1, 2.2 et 2.3 illustrent des optimisations uniquement possibles grâce au LTO. Les fonctions `foo1`, `foo2` et `foo4` sont exportées, et donc utilisables dans un autre module, ce qui force donc le compilateur à garder `foo3`. Lors d'une phase de LTO, le compilateur identifie que `foo2` n'est jamais utilisée et peut donc être retirée, ce qui implique que `i` n'est jamais changé et donc `foo3` ne sera jamais appelée et en conséquence de quoi `foo4` n'est plus appelée dans le programme. Les fonctions `foo3` et `foo4` peuvent donc être supprimées. Finalement, `foo1` ne fait que retourner la valeur 42, le programme peut donc être transformé en celui présenté dans le listage 2.4.

2.3.4 Pré-déploiement et déploiement

Portabilité & Optimisations : Le cas de la plate-forme Android est intéressant du point de vue du déploiement. Lors de la génération de l'application, le code DEX (indépendant de la cible) est assemblé avec les ressources de l'application pour former un paquet *APK*. Cet APK est ensuite déployé sur les terminaux Android. Ce mode de développement permet de supporter la variété de plates-formes utilisant Android, parmi les plus répandues ARM (32 et 64 bits), x86, x86-64 ainsi que MIPS.

Lors de l'installation de l'application sur une plate-forme, deux modes sont possibles [1] :

1. À partir de la version 4.4 d'Android, dans les versions précédentes, seul Dalvik est

Dalvik : Le code est optimisé avec l'outil *dexopt* [35] pour produire un fichier ODEX (Optimized DEX). Parmi les optimisations effectuées, on trouve un réalignement des structures, de l'inlining, remplacement des accesseurs et modificateurs par des décalages, élimination de méthodes vides ou ajout de données pré-calculées.

ART runtime : Ici le code est compilé grâce à l'outil *dex2oat* pour produire un fichier ELF, avec le code DEX compilé en instructions natives [16].

Dans les deux cas, le code DEX originel est conservé afin de pouvoir tenir compte des évolutions des dépendances.

Avec l'arrivée des multi-cœurs et l'explosion des capacités des terminaux, il devient plus facile de faire de la compilation en amont. Cette approche permet de mieux utiliser les capacités du SOC, ce qui a motivé le développement de *ART*. Auparavant, les contraintes sur les terminaux rendaient difficile l'intégration de cette approche [16].

Tuning : Ce moment se situe entre les instants 6 et 7 de la figure 2.2. Il permet d'avoir la connaissance précise de code généré dans le cas d'applications compilées en langage machine, et de l'environnent d'exécution avec notamment la connaissance des unités de calcul (modèle, taille des caches . . .). Cela permet donc d'optimiser l'application afin qu'elle utilise au mieux l'architecture. Ce moment est intéressant pour les opportunités d'optimisations possibles, car le temps (durée) disponible pour effectuer des optimisations est encore large.

Dans le domaine de l'algèbre linéaire, ATLAS [90] (*Automatically Tuned Linear Algebra Software*) est l'une des bibliothèques de référence dans ce domaine pour CPU. Pour offrir les meilleures performances, ATLAS utilise le moment de *l'installation* sur la machine hôte pour tester différents agencements. Ceci afin de permettre la construction d'une bibliothèque utilisant au mieux les caches de la machine. ATLAS possède un petit générateur de code C qui génère des variantes de l'algorithme de base pour jouer avec différents paramètres comme les déroulages de boucle ou l'utilisation de certaines instructions. Le système de tuning d'ATLAS va rechercher les paramètres optimaux pour utiliser aux mieux les niveaux de caches, profondeur du pipeline *etc.* Une fois la meilleure configuration trouvée, la bibliothèque est alors construite autour de ces résultats. Au vu des grandes disparités des performances dans les modèles existants, notamment au niveau de la taille des caches et leurs fonctionnements (qui est le principal point d'optimisation d'ATLAS), cette bibliothèque est contrainte d'effectuer cette étape sur la machine sur laquelle elle tournera. Les performances obtenues avec cette bibliothèque permettent d'atteindre 50 à 95 % des capacités théoriques des machines testées.

Dans le domaine de l'algèbre linéaire pour GPU, MAGMA [65] est l'une des références pour les GPU CUDA. Les performances des premières versions ont été ajustées manuellement au fur et à mesure des évolutions de la première génération de GPU CUDA. Les changements de générations ont poussé l'équipe à développer *ASTRA* [49, 50] (*Automatic Stencil TuneR for Accelerators*), un système de tuning automatique de la bibliothèque. Grâce à la façon dont sont architecturés les GPU CUDA, il n'est pas nécessairement utile d'effectuer le tuning de l'application sur la machine cible directement, mais sur une machine

disponible.

type qui permettra de représenter une génération de carte. Cette propriété permet à MAGMA d'être tuner au moment du pré-déploiement et d'être distribué sans aucune autre action pour l'utilisateur.

FFTW utilise une approche similaire à ATLAS en testant différents candidats générés grâce à son compilateur source à source. Sur la machine hôte au moment de *l'installation*, ces variantes sont testées dans différentes conditions. Cela permet de rejeter les candidats les plus mauvais et de garder ceux qui seront utilisés dans les plans d'exécution.

Spiral va lui aussi intégrer une phase d'évaluation des routines générés par le compilateur. Contrairement à FFTW, Spiral utilise une boucle de rétroaction lors de l'évaluation. Des candidats sont générés puis évalués, ces résultats sont ensuite utilisés pour générer d'autres candidats afin de tenter d'améliorer les performances. Dans certains cas, Spiral intègre aussi une autre étape d'apprentissage pour fournir une bibliothèque adaptative. Dans le cas de la FFT, Spiral utilise un algorithme d'apprentissage machine pour permettre la génération d'une bibliothèque capable de choisir quelle variante utiliser sans une phase d'adaptation à l'exécution.

2.3.5 Chargement et exécution

Ce moment se situe sur les instants 8 et 9 de la figure 2.2. Comme sur le temps précédent, la plate-forme matérielle est connue de façon précise. Seulement les ressources disponibles pour le générateur de code sont fortement restreintes car il consomme inévitablement des ressources normalement utilisées pour faire tourner le code métier. Pour autant, l'augmentation des performances des plates-formes tant en termes de taille des mémoires que du nombre d'opérations par secondes qu'une unité de calcul peut effectuer, autorise aujourd'hui l'utilisation de techniques complexes. De façon générique, on désignera par *compilation dynamique* l'intervention d'un compilateur à ce moment de la vie de l'application.

La *compilation dynamique* peut prendre deux formes distinctes en fonction des objectifs :

Portabilité : le but n'est plus d'avoir un système maximisant les performances, mais de proposer des exécutables indépendants d'un jeu d'instruction. À l'exécution, une *machine virtuelle* va soit interpréter soit compiler ce programme de façon transparente pour l'utilisateur [2]. Les performances sont là aussi un enjeu important, mais le but est ici de minimiser l'impact du coût de la compilation.

Optimisation des performances : il s'agit ici d'avoir un système qui permet d'optimiser les performances de l'application. Historiquement, c'est ainsi que sont nées les compilateurs dynamiques.

2.3.5.1 La compilation dynamique pour la portabilité et l'interopérabilité

L'approche la plus commune aujourd'hui dans le domaine de la compilation dynamique est bien son utilisation pour la portabilité. De façon générale, ces technologies passent par l'utilisation d'une *machine virtuelle* qui va gérer les aspects de génération de code, gestion de la mémoire et des optimisations. Les

2. c'est-à-dire sans intervention de sa part.

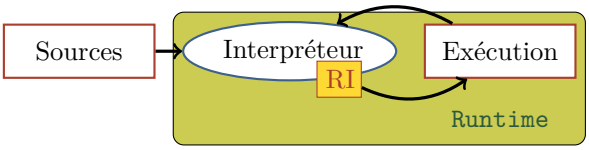


FIGURE 2.11 – Processus d'interprétation.

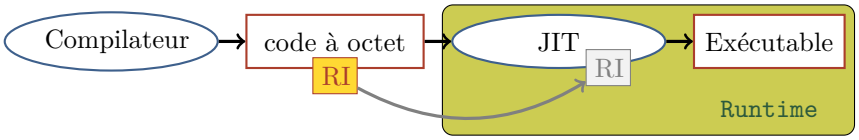


FIGURE 2.12 – Processus de compilation dans le cas d'un JIT.

figures 2.11 et 2.12 présentent de façon simplifiée les deux approches les plus courantes : celle de l'interprétation à partir des sources (par exemple *python*, *perl*, *JavaScript*...) et celle de la compilation juste-à-temps d'un code à octet (par exemple *Java* ou *LLVM*). Dans le cas de l'interprétation (figure 2.11), les sources sont directement données en entrée de l'interpréteur. L'interpréteur transforme la source dans une RI qu'il va interpréter directement ou il va compiler à la volée en instructions natives pour optimiser les performances. Dans le cas de la compilation juste-à-temps d'un code à octet (figure 2.12), les sources sont transformées en code à octet dans un temps statique. À l'exécution, le code à octet est soit interprété (comme dans le cas précédent), soit il est consommé par le compilateur juste-à-temps pour produire un code exécutable. Ces deux visions changent surtout avec le passage par une étape de précompilation pour le passage du niveau source à celui de code à octet.

Langages de scripts : Ces langages ne sont généralement pas compilés statiquement, mais lus, interprétés et éventuellement compilés dynamiquement. On peut citer notamment Python [77] ou JavaScript. Ce dernier occupe une place particulière avec l'émergence des technologies web. Beaucoup de sites dynamiques reposent sur une interprétation efficace du JavaScript. De façon classique, le code source en entrée est transformé dans une représentation intermédiaire pour être soit directement interprété, soit compilé en instructions natives pour les parties fréquemment utilisées.

Compilation juste-à-temps : Dans la catégorie de la compilation juste-à-temps, on peut citer la *Java Virtual Machine* (*JVM*) [48] sur laquelle s'est focalisé beaucoup d'efforts de recherches pour améliorer ses performances. Ces efforts se sont portés sur l'amélioration de l'interprétation, mais aussi sur la génération du programme en instructions natives. La génération en instructions natives pose aussi le développement de stratégie pour choisir quelles portions de code doivent être générées, à partir de quel moment et avec quel niveau d'optimisation. De façon connexe, on peut aussi citer *Dalvik* [35] que nous avons déjà vu sur les temps statiques et d'installations. La machine virtuelle *Dalvik* permet de compiler à la volée le code afin de le rendre aussi plus efficaces avec la contrainte supplémentaire de cibler des plates-formes potentiellement

très limitées. Les premières générations de terminaux avec Dalvik ne faisaient qu'interpréter le code, par manque de ressources disponibles pour effectuer de la compilation à l'installation ou à la volée. La mise à disposition de bibliothèques natives et optimisées dans le runtime d'Android permettait de limiter l'impact du surcoût de l'interprétation. Des études internes ont montré qu'environ 30 % de l'exécution d'un programme était de l'interprétation [35], le reste passant par ces bibliothèques. À partir de la version 2.2, la puissance disponible ayant largement évolué, Dalvik a été munie d'un JIT à la Java pour optimiser les points chauds et fournir de meilleurs performances.

Le cadriciel LLVM que nous avons déjà mentionné précédemment permet la construction de machines virtuelles en fournissant l'outillage nécessaire pour compiler à la volée le code à octet. VMKit [34] par exemple, est une bibliothèque permettant la construction rapide de machines virtuelles grâce à LLVM. Pour rejoindre les langages dynamiques comme javascript, on peut citer WebKit dont le moteur de compilation du JavaScript repose aussi sur LLVM [84]. Pour revenir sur Android mais toujours en lien avec LLVM, Google a inclus dans son écosystème *RenderScript* [80] un langage proche du C pour permettre le développement de routines critiques et portables. Ces routines sont statiquement compilées en LLVA et compilées en instructions natives au moment de l'exécution mais non interprétées.

Dans le domaine des GPU, il existe Ocelot [26], qui cherche à obtenir une portabilité des programmes CUDA à partir de la représentation intermédiaire de Nvidia de ces noyaux (PTX). Le système Ocelot permet la compilation et l'exécution d'un programme PTX sur les GPU Nvidia via l'API classique, mais aussi pour les GPU AMD en traduisant le code PTX en *IL* (l'équivalent du PTX pour les GPU AMD), ou en instructions x86 grâce à LLVM et un runtime spécifique.

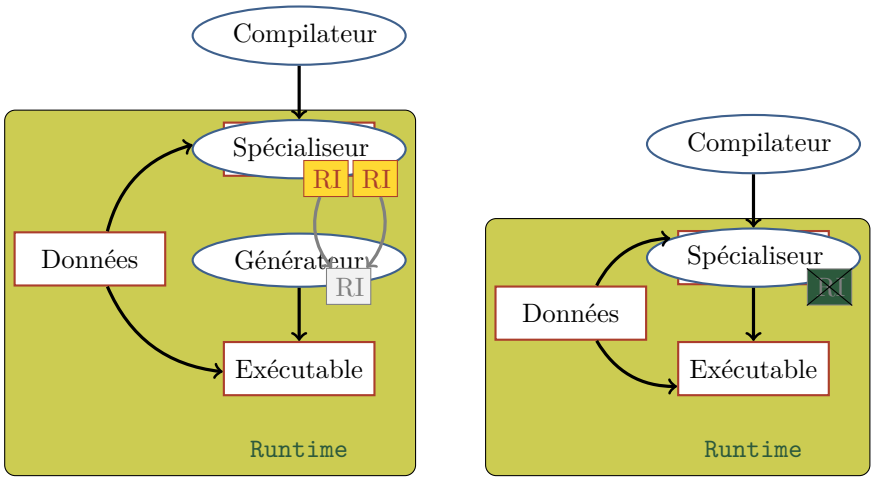
2.3.5.2 La compilation dynamique pour l'optimisation des performances

Ces approches se focalisent sur l'amélioration pure des performances, laissant de côté l'aspect portabilité des codes à octets ou sources interprétées. L'optimisation de l'application se fait par l'identification d'invariants dans le jeu de données traitées par l'algorithme, données qui ne sont pas connues ou non exploitables à la *compilation statique*. Par données non exploitables, on parle de données qui sont connues à la *compilation statique* mais dont l'utilisation pour des optimisations statiques sont difficiles car elles provoqueraient une explosion de la taille du code. Dans les approches précédentes, la compilation initiale du code à octet en code machine a un coût important, limitant la possibilité d'utiliser ces invariants. Historiquement, l'un des premiers travaux dans ce domaine a été la génération de code pour les expressions rationnelles [7], où plutôt que d'interpréter l'expression rationnelle pour la reconnaissance du motif, un code machine dédié était généré.

Ces spécialiseurs sont utilisés de deux façons :

Transparente : [3] une mini-machine virtuelle est alors insérée avec l'application pour permettre l'étude des points chauds avant leurs spécialisations ;

3. Le terme transparent désigne la non-modification des sources par l'utilisateur, mais peut impliquer un chaîne de compilation spécifique



(a) Spécialisation par assemble d'une représentation intermédiaire.

(b) Spécialisation par patron.

FIGURE 2.13 – Spécialisation légère (sans représentation intermédiaire).

Explicite : le programmeur va alors soit lui-même créer son spécialiseur, soit identifier les fonctions et les paramètres qu'il considère comme invariant à l'exécution.

En termes d'implémentation, deux grandes approches reviennent de façon récurrente, celles basées sur une représentation intermédiaire (RI), et donc utilise un JIT, soit par "*patron*", les portions à spécialiser sont alors précompilées et assemblées à l'exécution. Les approches basées sur une RI peuvent être schématisées par la figure 2.13a, lors de la compilation, le générateur de code (spécialiseur) est créé et embarque des morceaux de RI dans l'application. À l'exécution, ces moreaux de RI sont assemblées et compilées à la volée en utilisant les données de l'utilisateur. Les approches basées sur les patrons peuvent être schématisées par la figure 2.13b. Contrairement à l'approche précédente, le générateur de code (spécialiseur) embarque des morceaux de code précompilés mais pas de RI. À l'exécution, ces moreaux de code sont assemblés à la volée en utilisant les données de l'utilisateur.

2.3.5.3 Spécialisation dynamique explicite

Le principe de cette approche repose sur une identification manuelle des opportunités d'optimisation dynamique et notamment de la spécialisation par identification d'invariant à l'exécution. Dans l'état de l'art, deux courants majeurs se distinguent. Il y a ceux basées sur une représentation intermédiaire qui est assemblée et compilée à l'exécution et ceux basées sur une manipulation d'instructions ou bloc d'instructions binaires. La première offre plus de possibilité d'optimisations que la seconde mais à un coût supérieur. Les approches basées sur la manipulation d'instructions ou bloc d'instructions binaires sont des approches dites par *patrons*. Lors de la compilation statique, le code qui sera spécialisé est compilé à un niveau binaire et du code ciment est générée pour permettre l'allocation de la mémoire, l'assemblage du code, ajuster les sauts,

invalider les caches *etc.*

Les spécialiseurs vont être exprimés de deux façons : soit en exprimant la fonctionnalité voulue, soit en écrivant directement le générateur.

Approche par expression fonctionnelle : Cette approche repose sur l'expression de la fonction qui doit être réalisée une fois spécialisée, de la même façon que l'on réaliserait une évaluation partielle dans les langages fonctionnels. Beaucoup de techniques reposent sur ce principe, on peut notamment citer *Tempo* [52, 60, 21] et *DyC* [37, 38] dont le but est la spécialisation de programme écrit en C et *Fabius* [53, 54] qui a pour cible le ML. *Tempo* utilise une approche source à source (voir la partie 2.3.2), ce qui lui permet d'assurer la portabilité du système. Son utilisation repose sur l'utilisation d'un fichier de configuration décrivant les fonctions à spécialiser. *Tempo* va ensuite analyser les sources et produire un code source avec les routines de spécialisation pour les fonctions visées par le fichier de configuration. À charge de l'utilisateur de *Tempo* d'appeler ce spécialiseur. *DyC* est une extension du C et compilateur complet pour ce langage visant les *Alpha AXP*. Les extensions au langage C ont été ajoutées pour décrire les invariants des fonctions par l'utilisation de la pseudo-fonction (`make_static` par exemple). Avec *DyC*, le cache des fonctions spécialisées est géré de façon transparente par le système. Plusieurs stratégies de gestion de cache peuvent être employées, le choix est local à la fonction et est déclaré directement dans les sources. *Fabius* est un système optimisant pour un dérivé du ML. Le langage a été modifié pour exprimer les constantes à l'exécution. Les paramètres des fonctions sont séparés en 2 : la partie dynamique et la partie constante à l'exécution. À partir de ces informations le compilateur lance des passes d'analyses et de transformations similaires à ce qui est fait avec Tempo ou DyC.

Un peu en marge, mais toujours sur un principe similaire, on trouve des techniques spécialisées dans un domaine. *Tempo* a été employé pour spécialiser la pile réseau de systèmes embarqués [12]. L'approche est intéressante sur 2 points :

1. elle vise directement des systèmes embarqués et contraints (en capacité de calcul, mémoire et énergie), avec notamment une attention particulière pour la taille du code généré. Dans leur papier, ils obtiennent une réduction d'un facteur 20 de la taille du code généré par rapport au code générique et une amélioration de la vitesse d'exécution de l'ordre de 25 %.
2. la spécialisation est faite de façon déportée sur un serveur qui envoie le code spécialisé, permettant ainsi de contourner les contraintes de la plate-forme cible.

Approche par expression du générateur : Une autre approche est l'expression directe du code que l'on veut générer, à l'opposé donc des approches précédentes qui se focalisaient sur l'aspect fonctionnel. Dans ces approches, on a un mélange entre le code du spécialiseur et le code que l'on souhaite générer. Le code à générer est mêlé à celui du spécialiseur. Le programmeur peut ainsi avoir un contrôle total sur la façon dont les fonctions sont spécialisées et peut donc créer des optimisations spécifiques et plus difficilement réalisables ou généralisable de façon automatique.

On peut citer l'exemple de `C [74], une extension du langage C où le programmeur peut écrire directement son spécialiseur. Dans une fonction C standard, le programmeur écrit le code qu'il souhaite générer par morceau grâce au caractère `. Le caractère @ permet quant à lui d'insérer une valeur calculée par le générateur dans le code généré et permettre ainsi la spécialisation de valeur. Ces fonctions `C sont compilées grâce au compilateur *tcc* qui va préparer le code pour la génération dynamique pour un des deux moteurs à l'exécution *ICODE* et *VCODE*. Le moteur *VCODE* permet une génération rapide du code mais sans optimisation, *ICODE* effectue des transformations plus lourdes, produisant des routines plus optimisées au prix d'un temps de génération plus long.

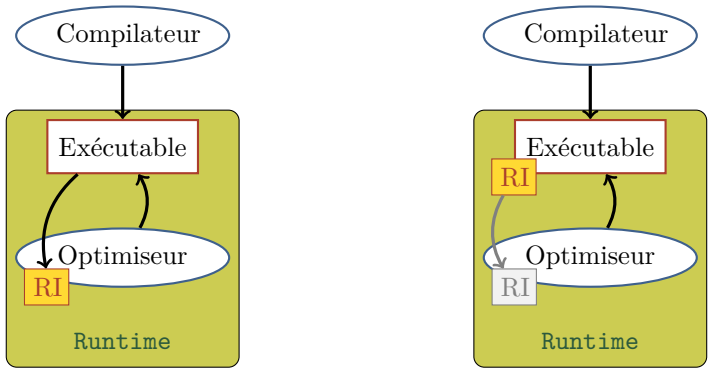
Toujours sur le même principe d'écrire le générateur, on trouve des approches qui vont exprimer le code à émettre à un niveau quasi-assembleur. Cette approche permet de garder un certain contrôle sur le code que l'on génère, c'est-à-dire que c'est le programmeur qui va choisir quelles instructions générer et dans quel ordre. Dans les travaux les plus anciens on trouve DCG [29] (à l'origine de `C), qui permet l'expression sous forme de macro l'expression de pseudo-instruction. Ces pseudo-instructions sont assemblées à l'exécution pour former un arbre dans l'IR du compilateur *lcc* [30] et le code machine est ensuite généré en utilisant une partie arrière non optimisant. Plus récemment, *Brifault et al.* [14] ont appliqué une approche similaire pour l'optimisation de noyaux de calcul multimédia. Les gains obtenus sont liés à la sélection manuelle des instructions, leurs ordonnancements et le déroulage des boucles effectués grâce aux informations connues uniquement à l'exécution. Des travaux ont fait suite à ceux de *Brifault et al.* avec HPBCG [19] pour obtenir notamment des générateurs plus faciles à recibler.

Toujours dans les méthodes basées sur les patrons, mais qui ne rentrent pas tout à fait dans les catégories précédentes, on peut citer les travaux de *Khan et al.* [46]. À la compilation statique, les paramètres des noyaux à spécialiser sont fixés avec différentes valeurs et compilés statiquement. Le système analyse les variantes et recherche les constantes qui évoluent de façon affines. À partir de cette analyse, un patron à "trou" générique est créé avec une routine calculant les constantes à insérer lors de l'exécution. Au moment de l'exécution, ces constantes sont calculées et le patron est modifié en-place de façon à produire une fonction valide. Cela permet d'avoir un temps de génération plus court tout en gardant un code efficace.

On peut aussi citer HIVE [91] qui est focalisé sur l'optimisation des requêtes sur les bases de données. HIVE applique des techniques de spécialisation basées sur des patrons pour optimiser le traitement des requêtes SQL. HIVE passe par la modification manuelle du moteur de la base de données, il n'y a donc pas ici de langage spécifique comme avec DyC par exemple. Les gains obtenus se situent entre 10 et 20 % en moyenne.

2.3.5.4 Spécialisation et optimisation dynamique transparente

Ne sachant pas à l'avance les points chauds à optimiser, l'application est instrumentée et compilée de façon statique pour permettre son exécution immédiate. L'instrumentation permettra de détecter les points critiques de l'application lors de son exécution. Ainsi, lors de l'exécution, lorsqu'un point chaud est détecté, il est alors possible de chercher à l'optimiser. Ce processus est schématisé dans les figures 2.14a et 2.14b.



(a) Approche par reconstruction de la représentation intermédiaire.

(b) Approche avec une représentation intermédiaire embarquée dans l'application.

FIGURE 2.14 – Spécialisation basée sur l'optimisation d'une représentation intermédiaire.

La première figure (2.14a) schématise l'approche basée sur la reconstruction (partielle) de la représentation intermédiaire. Elle offre l'avantage de ne pas faire augmenter la taille de l'application, mais présente un surcoût dû à la reconstruction de la structure de programme. C'est la stratégie utilisée par Dynamo [9]. Le but de Dynamo est de fournir un système optimisant basé sur les opportunités identifiées lors de l'exécution de l'application pour les processeurs HP PA-8000. Lors de l'exécution de l'application, des compteurs sur les adresses de branchement sont insérés. Arrivé à un certain seuil, la trace d'exécution est enregistrée dans un tampon. Cette trace est ensuite optimisée et insérée dans un cache de "*fragments*" pour une réutilisation future.

Un outil hybride avec l'approche précédente est *Calpa* [63]. C'est un outil qui permet d'annoter automatiquement un code C avec le système de *DyC*. Pour ce faire, Calpa procède par une phase de profilage en utilisant un ensemble de jeux de tests. À partir des résultats d'exécution sur ces jeux de tests, Calpa annote le code de façon automatique. Contrairement à Dynamo, dont le principe est l'optimisation de trace, Calpa fonctionne par anticipation en annotant un programme avec DyC. Calpa est donc incapable de s'adapter aux évolutions des usages de l'application. Un autre point faible propre aux systèmes faisant de la compilation guidée par profilage est que le jeu de données de test doit être représentatif de la façon dont sera utilisé l'application. S'il y a un biais dans le jeu de données, les annotations risquent de ne pas être insérées aux endroits opportuns, pouvant donner lieu à un ralentissement de l'application dû à son incapacité à amortir le coût de la génération.

Un problème récurant dans l'utilisation des techniques décrites précédemment est qu'un choix doit être fait entre niveau d'optimisation et temps de mise en route du système. Les approches basées sur les RI (LLVA ou code à octet Java) vont permettre un niveau d'optimisation élevé avec l'inconvénient que pour pouvoir tourner, il faut un certain temps pour que la RI soit transformée en instruction native. Les approches basées sur la reconstruction sont limitées par le fait qu'elle démarre d'une représentation bas niveau, et sont donc limitées dans leurs possibilités. *Nuzman et al.* [67] utilisent une approche lourde (dite

"Fatbin", schématisée dans la figure 2.14b) pour la compilation et optimisation dynamique de programmes C/C++ où la représentation intermédiaire est insérée avec le code natif de l'application. De cette façon, l'application obtenue est prête à être exécutée de façon classique, sans période de "préchauffage". Un système de profilage va lors de l'exécution détecter des opportunités d'optimisations. Ces morceaux identifiés sont optimisés en utilisant un compilateur à la volée. Le fait de posséder le code générique permet l'optimisation dans un processus dédié sans pour autant stopper l'application, ce qui n'était pas fait ou possible dans les approches précédentes. Cette approche provoque par contre une inflation de la taille du binaire d'un facteur 2 à 5 pour un gain moyen d'environ 8 %.

Un autre champ d'utilisation de ces techniques est la traduction binaire (pour passer de façon transparente d'un jeu d'instruction à un autre). Pour exécuter un binaire sur une machine possédant un autre jeu d'instruction, une machine virtuelle va effectuer la traduction du jeu d'instruction d'origine vers le jeu d'instruction de la plate-forme utilisée. *qemu* [10], dans sa version d'origine, passe d'une ISA à une autre via pseudo représentation intermédiaire. Cette architecture permet, comme pour les compilateurs classiques, de plus facilement cibler des ISA nouvelles tout en gardant le support des ISA en entrée.

2.3.5.5 Génération de code dynamique pour l'optimisation de la taille

La plupart des techniques que nous avons décrites jusqu'à présent était dédiées à l'optimisation des performances de la vitesse d'exécution et, dans certains cas, au détriment de l'empreinte mémoire de l'application. Il est vrai que pour des systèmes HPC l'empreinte mémoire n'est pas forcément un problème au vu des ressources disponibles. Dans le cas d'applications embarquées, l'octet peut rapidement devenir coûteux. Un exemple un peu extrême est celui du MSP430 : dans sa configuration la plus contrainte il n'y a que 512 octets de mémoire vive. De même, certaines plates-formes Arduino n'ont que 1 Ko de RAM.

Les travaux de *Heydemann et al.* [44] traitent ce problème en employant des techniques de compression du code avec un décodeur spécialisé et embarqué. Cela permet d'avoir un système spécialisé avec une empreinte mémoire réduite et un impact sur les performances limitées grâce une recherche automatique d'un compromis.

2.4 Rétroaction et anticipation

Dans la partie précédente, nous avons vu les différents grands moments de la vie d'une application et leurs utilisations dans des objectifs de portabilité ou de performances. Dans certains cas, l'utilisation du temps le plus adéquat au problème est impossible, souvent pour des raisons d'impact sur l'exécution de l'application mais aussi pour des raisons d'intégration dans les chaînes de compilations, de coûts, *etc.*.

2.4.1 Rétroaction

Parmi les outils et techniques que nous avons vues, certains utilisent une boucle de "rétroaction" pour effectuer des optimisations (en anglais "Feedback

Oriented Optimization"). Le principe est de réinjecter des connaissances acquises dans un temps situé en aval dans le but d'améliorer le processus. Parmi les outils cités, nous avons ceux ayant une phase de tuning dans la phase de déploiement. Par exemple Sprial va utiliser le retour sur les performances des variantes générées pour en générer de nouvelles qu'il espère meilleures.

Calpa est un autre exemple, qui utilise un ensemble d'apprentissages pour détecter des opportunités de spécialisation et va utiliser cette connaissance pour insérer des annotations *DyC* dans ce programme. Pour des optimisations purement statiques, le compilateur LLVM commence à être capable d'utiliser des traces d'exécution pour modifier ses heuristiques. Avec *GCC*, on peut citer *Interactive Compilation Interface* (ou *ICI*) intégré depuis la version 4.5. Celui-ci permet un contrôle plus fort sur le déroulement des opérations de GCC et notamment d'affiner les paramètres des heuristiques pour un contexte précis.

2.4.2 Anticipation

D'autres outils vont fonctionner par anticipation afin de *prédire* les paramètres ou les transformations à appliquer lors d'une optimisation. Dans les compilateurs statiques, *MILEPOST GCC* [33] est une modification de GCC utilisant *ICI* et utilisant des techniques d'apprentissages machines. Le but est de pouvoir extraire des caractéristiques des programmes compilés et ainsi apprendre et améliorer de façon automatique les passes d'optimisations. Coté GPU, on peut citer les travaux de *Bergstra et al.* [11] qui cherchent à prédire les performances (en *GFLOPS*) de noyaux générés. À partir d'un ensemble de noyaux de référence, un arbre de régression est construit permettant ainsi de "prédire" ses performances. L'avantage est ici le temps requis pour obtenir une réponse, de 0.1 seconde à 3 secondes pour estimer la performance, contre 1 à 3 minutes si l'on avait réalisé le test.

2.5 Conclusion

Dans ce chapitre nous avons abordé la notion de compilation et de génération de code multi-temps ainsi qu'une introduction sur les représentations intermédiaires, indispensables à la représentation des programmes et à la communication entre les différents temps. Nous avons ensuite effectué un état de l'art de la compilation et génération de code ainsi que sur la façon dont ils utilisent les différents moments de la vie d'une application.

Le tableau 2.1 reprend de façon synthétique les différentes approches présentées dans la section 2.3, avec les moments durant lesquels ils interviennent. Comme nous avons pu le voir, ces techniques prennent diverses formes avec des temps utilisés de différentes façons selon le niveau d'information disponible. Pour certaines, l'utilisation des temps est fortement liée à l'application ou au type d'architecture cible, obtenant généralement une amélioration des performances (au sens de l'amélioration du comportement en fonction d'une métrique) et ce de façon plus ou moins transparente.

Dans le prochain chapitre, nous introduirons des limitations de l'état de l'art dans le domaine des GPU dans l'optimisation de la GEMM et le problème de l'impact de la génération de code à l'exécution selon plusieurs métriques.

Outil	Compilation statique	Compilation sources à sources	Édition des liens	Pré-déploiment	Déploiment	Exécution	FDO
GCC	✓		✓				✓
LLVM	✓		✓			✓	✓
ΣC		✓					
Polly	✓						
Graphite	✓						
Spiral		✓			✓		✓
FFTW		✓			✓		✓
Android/Dalvik	✓				✓	✓	✓
Android/ART	✓				✓	✓	
ATLAS					✓		✓
MAGMA				✓			✓
Java	✓					✓	✓
Python						✓	✓
VMKit						✓	
Tempo	✓	✓				✓	
DyC	✓					✓	
`C	✓					✓	
Fabius	✓					✓	
HIVE	✓					✓	
DCG	✓					✓	
Brifault et al.	✓					✓	
HPBCG	✓					✓	
Khan et al.	✓	✓				✓	
Dynamo/DynamoRIO	✓					✓	✓
Calpa	✓	✓		✓		✓	✓
qemu						✓	
Nuzman et al.	✓					✓	✓
Heydemann et al.	✓					✓	

TABLE 2.1 – Synthèse des approches développées dans l'état de l'art.

Chapitre 3

Les verrous de la génération de code

Sommaire

Le chapitre précédent a montré la grande variété des stratégies d'optimisations et de générations de code exploitant des moments de la vie d'une application. Chacune cherchant une réponse à des problèmes variés en termes de performances (généralement vitesse d'exécution), et sur des cibles variées : de l'ordinateur de bureau, au système HPC et sur plates-formes embarquées. Il en ressort une difficulté de plus en plus élevée pour tirer complètement parti des capacités des architectures. Les raisons sont diverses : inadéquation des langages aux architectures émergentes, impossibilité (ou absence de volonté) de faire évoluer les bases de code ou le processus de construction de l'application [67] ou encore les besoins de portabilité.

Ce chapitre présente des limitations de l'état de l'art dans deux domaines. Le premier concerne le domaine des GPU. Nous présenterons une limitation liée à la non-utilisation des données pour optimiser les performances. Par le biais d'un exemple, nous verrons que dans certaines conditions les algorithmes ne peuvent tirer correctement parti de la gestion des processus offerte par les GPU. Le deuxième concerne plutôt le domaine des systèmes embarqués. Les méthodes de

spécialisations présentées dans l'état de l'art ne se sont concentrées que sur une métrique, généralement la vitesse d'exécution. Les autres contraintes comme l'énergie ou l'empreinte mémoire sont donc ignorées. Ces paramètres étant important dans ce domaine, cela lève la question l'impact de la spécialisation.

La suite du chapitre présentera dans un premier temps un contexte général. On présentera le fossé entre le matériel et les applications ainsi que les objectifs de compilations. Pour les GPU, nous présenterons ensuite le problème de fond et la limitation de l'état de l'art sur les performances de la GEMM. Nous présenterons enfin les problèmes généraux concernant la spécialisation et soulèverons la question de l'utilisation d'autres métriques dans l'étude des gains et coûts de ces systèmes. La conclusion fera enfin une synthèse de ces deux problématiques.

3.1 Contexte général

3.1.1 Applications contre matériel

Une analogie illustrant l'inadéquation des langages aux architectures est celle du "Golden Gate Bridge" faite par l'université de Californie à Berkeley [6] et qui est représentée dans la figure 3.1. Aux deux extrémités du pont nous avons les deux acteurs : *l'industrie* informatique (constructeurs) et les *utilisateurs* (clients). Les utilisateurs ont leurs processus métiers donnant des besoins *applicatifs*, pour lesquels l'industrie développe des réponses *matérielles*. Entre l'utilisation de ce matériel et les applications se trouvent les *logiciels* sur lesquels reposent l'interface entre ces deux mondes. Pour créer ces logiciels, il est nécessaire de passer par un *langage* pour décrire l'applicatif de l'utilisateur et une phase de *compilation* pour utiliser le matériel. Or, il y a un décalage entre les langages et les applications, duquel va découler une perte de sémantique (symbolisée par la descente du câble). À charge du compilateur d'essayer de réduire ces lacunes afin d'utiliser au mieux le matériel.

La complexification des architectures rend la pleine utilisation du matériel plus difficile. L'augmentation des tailles de caches ou du nombre de cœurs rendent certains algorithmes plus sensibles aux données. Or ce sont les utilisateurs qui détiennent ces données qui seront utilisées, et donc elles sont connues très tard dans la chaîne de réalisation de l'application. Un cas notable est celui de la FFT, avec notamment les travaux sur FFTW où diverses variantes sont générées lors de la compilation et un plan d'exécution est généré lors de l'exécution du programme quand le format des données est connu.

3.1.2 Objectifs de la génération de code

Les objectifs d'une application (couple matériel et logiciel) prennent aujourd'hui en compte des contraintes plus larges que le simple objectif de vitesse d'exécution. L'étude des performances des applications ont souvent été fait sous le prisme que d'une seule métrique avec éventuellement l'étude du coût vis-à-vis de cette métrique.

Le tableau 3.1 répertorie différents objectifs pour la compilation avec des métriques utilisables pour la mesure des performances. La colonne de gauche présente des objectifs qui peuvent être visés lors de la création d'une application. La colonne de droite présente des métriques qui peuvent être utilisés pour

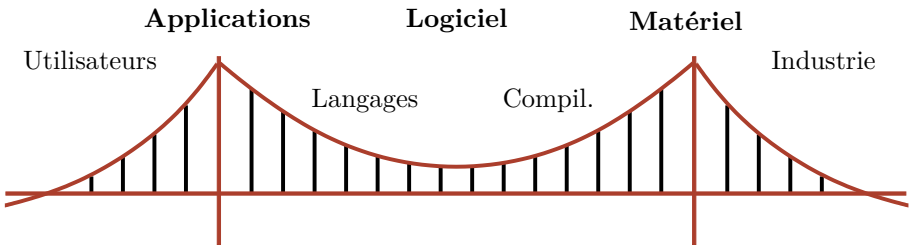


FIGURE 3.1 – Analogie du pont de Berkeley [6] (inspiré d'une vue du Golden Gate Bridge).

Objectif	Métrique
Vitesse d'exécution	Opérations par seconde Vitesse de réaction
Consommation énergétique	Watt Joules
Efficacité énergétique	Opérations par Watt Opérations par seconde et par Watt
Empreinte mémoire	Octet % d'augmentation
Portabilité	Coût de surcharge
Vitesse de développement	Temps de développement

TABLE 3.1 – Exemple d'objectifs de compilation et de métriques associées possibles.

optimiser ces objectifs. Pour un objectif donné, il peut exister plusieurs métriques différentes. Par exemple, si l'objectif est la vitesse d'exécution cela peut être interprété comme le nombre d'opérations par seconde ("throughput") mais aussi comme la vitesse de réaction (délai). Les choix d'optimisations peuvent diverger selon que l'on utilise l'une ou autre de ces métriques. Par exemple, si on optimise la vitesse d'exécution avec le délai comme métrique, des choix peuvent être fait pour réduire le temps d'attente en fournissant une réponse partielle par exemple. Cas choix ne vont peut-être pas permettre d'utiliser au maximum le matériel et donc maximiser le nombre d'opérations par seconde. De même, lorsque l'on souhaite optimiser selon plusieurs objectifs il peut naître une compétition entre eux, requérant donc un compromis.

Dans le domaine du calcul haute performance, les efforts étaient portés sur la vitesse d'exécution. Le TOP500 [85], utilise le FLOPS comme unique unité de mesure et classe ainsi les machines les plus performantes. Le classement Green500 [39], lancé en 2007, utilise le FLOPS par Watt comme unité de mesure, préférant ainsi l'efficacité énergétique comme indicateur de performance. Il en résulte un classement radicalement différent de celui du TOP500.

Sur les systèmes embarqués, les métriques de taille et de consommation vont avoir tendance à prendre le dessus sur la vitesse d'exécution. Cela a un impact sur la conception de ces systèmes sur la mémoire car elle est coûteuse et un facteur important de consommation. Cette mémoire est donc souvent limitée. La mémoire limitée a une conséquence importante sur la manière de concevoir les

applications et la chaîne de compilation doit être conservative dans ses optimisations afin que la taille de l'application soit raisonnable et avec des performances acceptables. Par exemple, dans les travaux de *Heydemann et al.* [44], l'optimisation se fait à la fois selon un objectif de réduction de l'empreinte mémoire et de vitesse d'exécution.

3.2 Lien entre données et performances sur GPU

3.2.1 Précision sur l'architecture des GPU CUDA

Les données ont leur importance dans l'optimisation. Sur CPU, la connaissance des données à traiter (fournies par l'utilisateur) ou des alignements (liées à l'exécution) peuvent permettre l'utilisation d'instructions plus rapides. La posture conservatrice des compilateurs statiques ne leur permet pas toujours de retenir ces instructions car elles pourraient rendre le code invalide. L'émergence du mutli-cœurs rend le phénomène plus important. La taille des données va jouer un rôle direct dans la distribution du calcul aux différents cœurs de la plate-forme, impactant la performance obtenue.

Sur les GPU CUDA, il est nécessaire d'avoir un grand nombre de processus pour obtenir de bonnes performances. Les calculs sont donc divisés et répartis sur une grille de calcul, divisant la charge de travail. Avec la latence importante des GPU CUDA, il est important d'utiliser au mieux les instructions (ILP) pour maximiser le parallélisme au niveau instruction et le nombre de processus. Aux débuts de CUDA, Nvidia conseillait d'ailleurs de maximiser le nombre de processus disponibles plutôt que de jouer sur l'ILP. Des travaux, notamment sur la GEMM pour CUDA, ont rapidement montré que maximiser le nombre de processus n'était pas une condition nécessaire pour obtenir les meilleures performances [88, 87].

Pour atteindre le pic de performances, il est important d'avoir suffisamment d'opérations à effectuer. Sur les générations 1.x et 2.0, cela peut être fait uniquement grâce au nombre de processus, mais avec la génération 2.1, il est indispensable d'avoir de l'ILP si on l'on souhaite dépasser les 2 tiers du pic. Pour occuper au maximum la bande passante, il est nécessaire que les processus chargent une certaine quantité de données chacun. Plus un processus va charger de données, moins il sera nécessaire d'avoir de processus pour occuper la bande passante.

Jouer avec l'ILP et le nombre de processus permet de jouer sur les ressources consommées par un processus. Ce qui déterminera le nombre de processus présent dans un SM. Si l'on ajoute les particularités des unités de chargements, le trafic avec les différents niveaux de mémoires ou les points de synchronisations, il en résulte un nombre important de possibilités de paramétrage.

3.2.2 Cas de la GEMM

Si l'on prend l'exemple de la multiplication matricielle, dans l'algorithme générique, la division du travail est fait selon la taille de la matrice résultat. C'est-à-dire que pour une multiplication $C = A \times B$, où A est de taille $M \times K$ et B de taille $K \times N$, la division du travail ce fait donc selon M et N. Dans le cas de `MAGMA` [49], une implémentation de référence pour les GPU CUDA, le

M	K	N	Nombre de processus	GFlops
64	1984	64	256	186
256	1984	256	2 304	435
640	1984	640	12 544	549
1600	1984	1600	73 984	594

TABLE 3.2 – Nombre de processus crées par `MAGMA` lors d'une multiplication matricielle avec des flottants simple précision et la performance obtenue sur une carte M2050.

choix sur la division du travail est fait de façon statique et avant le déploiement. Pour faire ce choix, une unique matrice est utilisée, de grande taille et de dimension équilibrée. Or, à l'exécution, cela peut poser un problème si les dimensions s'éloignent trop de ce cas de test. Le tableau 3.2 présente le nombre de processus créés par `MAGMA` pour effectuer le calcul sur un GPU de la génération Fermi. Les trois premières colonnes donnent les trois dimensions des matrices en entrée, la deuxième colonne donne le nombre de processus qui sont créés pour effectuer le calcul, enfin la dernière colonne présente la performance obtenue pour chaque cas. Pour la matrice la plus équilibrée dans ces dimensions, le nombre de processus est suffisant pour alimenter le GPU, mais plus elles deviennent étroites, moins le GPU est alimenté correctement.

L'adaptation du découpage de l'algorithme pourrait permettre d'améliorer ces performances. Pour ce faire, il faut cependant être capable d'inférer la configuration à utiliser, ce qui implique donc une exploration de l'espace des dimensions des matrices et la capacité d'utiliser ces résultats. Pour ce faire, cela nécessite aussi de pouvoir créer et gérer un certain nombre de variantes. Le faire statiquement pose le problème de l'empreinte mémoire car il y a un risque à faire augmenter fortement la taille de la bibliothèque. Il nous sera donc nécessaire de contrôler l'empreinte mémoire de la solution.

3.3 Gains et coûts de la spécialisation dynamique de code

3.3.1 Intérêt de la spécialisation dynamique

L'avantage le plus notable de la génération dynamique de code est la connaissance globale que le générateur possède sur son environnement. Nous avons accès à des informations sur le matériel et sur les données à traiter que ne possède pas le compilateur statique. La figure 3.2 schématise cette évolution de la connaissance du contexte avec quelques opportunités d'optimisations. Dans le meilleur cas au moment de l'écriture, nous pouvons être capables d'estimer la taille des données manipulées par le programme. Un expert sera capable d'estimer cette taille et d'insérer des indications dans le programme si le langage le permet. Par exemple certains compilateurs C supportent des *pragma* (généralement spécifique au compilateur) pouvant être utilisés par l'expert pour guider les optimisations. Lors de la compilation statique, la taille du corps des boucles est connue, cela donne la possibilité au compilateur de les dérouler partiellement. Mais souvent le compilateur n'a pas de connaissances précises sur la cible et les données,

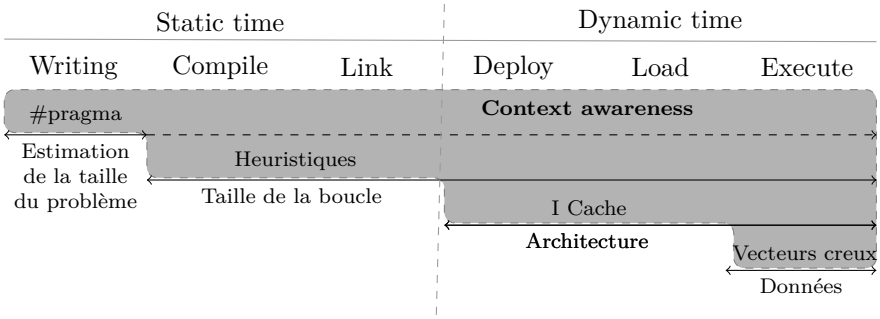


FIGURE 3.2 – Évolution de la connaissance du contexte.

le compilateur est donc contraint de se reposer sur des heuristiques. Au moment du déploiement et du chargement de l'application, ces informations sur la plate-forme deviennent connues. Malheureusement dans la grande majorité des cas, le programme est ici déjà vu comme un invariant, empêchant les optimisations utilisant ces informations. Finalement, lors de l'exécution du programme, l'utilisateur fournit les données qui sont exploitées par le programme. En cas d'invariant, il y a de nouveau une possibilité d'améliorer les performances du programme. Par exemple, le fait de savoir que les vecteurs manipulés sont creux permet de produire des boucles proprement déroulées en levant les gardes présentes dans la boucle. L'inconvénient d'intervenir à ce moment est la réduction des ressources disponibles (temps, mémoire, énergie) qui vont prendre sur celles normalement allouées pour l'exécution du programme (pour réaliser sa tâche).

La compilation dynamique va généralement intervenir pour résoudre 2 problématiques : *a)* portabilité ; *b)* optimisation des performances. Dans le premier cas, la plate-forme n'est connue qu'à l'installation ou qu'à l'exécution, ce qui laisse peu de possibilité pour anticiper la compilation. Dans la meilleure configuration, un processus de compilation l'aura converti en code à octet. L'optique sera donc ici de limiter le surcoût de l'interprétation et de la génération du code, lors de l'installation (compilation en amont, "ahead-of-time") ou à l'exécution (compilation juste-à-temps, "just-in-time"). La compilation lors de l'installation offre l'avantage de ne pas ralentir l'exécution. Il offre aussi la possibilité de recompiler certaines parties en fonction du comportement dynamique du programme ou des données. Avec la compilation juste-à-temps, cela est plus difficile à cause du coût initial de la génération. Dans le deuxième cas, la plate-forme est généralement connue ce qui offre la possibilité d'anticiper la compilation et les calculs.

Comme il a été vu dans le chapitre précédent, beaucoup de techniques ont été développées pour permettre la spécialisation de code lors de l'exécution. Ces techniques présentent diverses approches à la fois dans l'expression des programmes et dans la préparation de ces spécialiseurs. D'une façon générale, ces techniques étages les étapes de la génération du code afin de préparer la génération.

La compilation étagée permet de distribuer les étapes de génération du code dans le temps, au fur et à mesure que les informations sont connues. Les optimisations qui peuvent être effectuées ne requièrent pas toutes le même niveau d'informations sur le jeu d'instruction, les caractéristiques architecturales (la-

```
1  typedef int aint __attribute__ ((__aligned__(32)));
2  int reduction(int               n,
3                aint* __restrict  p,
4                aint* __restrict  q) {
5    aint res = 0;
6    while (n--) {
7      res += *p++ * *q++;
8    }
9    return res;
10 }
```

LISTAGE 3.1 – Implémentation d'une multiplication vecteur/vecteur.

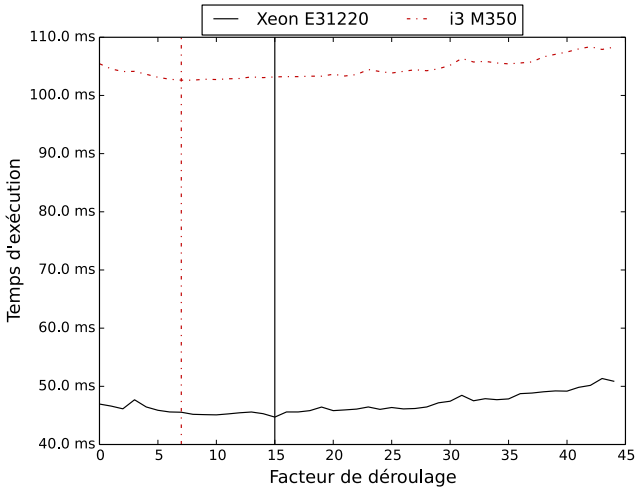


FIGURE 3.3 – Vitesse d'exécution du noyau 3.1 compilé avec clang 3.4 et avec différents facteurs de déroulage pour l'option -force-vector-unroll. Les lignes verticales montrent le point optimal de déroulage pour les deux architectures.

tence des instructions ou taille et fonctionnement du cache par exemple) ou les données (taille, valeurs, alignements).

DyC ou Tempo vont permettre d'identifier des constantes qui ne seront connues que lors de l'exécution. Une chaîne de transformation permet de préparer et spécialiser les routines de génération pour avoir un temps de spécialisation le plus faible possible.

3.3.2 Illustration

Le frein majeur à l'optimisation et à la génération purement statique du code est une connaissance limitée de l'environnement d'exécution. Un exemple classique est celui du déroulage de boucle. Le listage 3.1 présente une implémentation naïve d'une multiplication vecteur/vecteur en C. Son code est composé d'une simple boucle qui va parcourir les éléments des 2 vecteurs, les multiplier et accumuler le résultat dans une variable.

Pour optimiser, on peut utiliser le déroulage partiel de boucle, de cette fa-

```
1  .LBB0_3:
2    vld1.64 {d18, d19}, [r2:128]
3    subs  r6, r6, #4
4    add r2, r2, #16
5    vld1.64 {d20, d21}, [r3:128]
6    add r3, r3, #16
7    vmla.i32  q8, q10, q9
8    bne .LBB0_3
```

LISTAGE 3.2 – Code généré par clang/LLVM 3.4 pour le Cortex-A8.

```
1  .LBB0_3:
2    vld1.64 {d20, d21}, [r2:128]
3    add r7, r2, #16
4    vld1.64 {d22, d23}, [r3:128]
5    subs  r6, r6, #8
6    vmla.i32  q8, q11, q10
7    add r2, r2, #32
8    vld1.64 {d20, d21}, [r7:128]
9    add r7, r3, #16
10   vld1.64 {d22, d23}, [r7:128]
11   add r3, r3, #32
12   vmla.i32  q9, q11, q10
13   bne .LBB0_3
```

LISTAGE 3.3 – Code généré par clang/LLVM 3.4 pour le Cortex-A15.

çon, nous pouvons utiliser au mieux les registres, améliorer l'ordonnancement des instructions *etc.* La question est alors : jusqu'à quel point doit-on dérouler ? La figure 3.3 présente des résultats d'exécution pour cet algorithme avec différents facteurs de déroulages (en utilisant l'option de clang "-force-vector-unroll"). Ces codes ont été exécutés sur 2 CPU de la famille des X86-64 mais de modèles différents : un Xeon E31220 et un i3 M350. L'axe des abscisses représente les différents facteurs de déroulages, l'axe des ordonnées représente les temps d'exécution du noyau en seconde. Les lignes verticales représentent le point optimal de déroulage pour chaque exécution. Comme on peut le voir, il existe un point optimal de fonctionnement avant que les performances ne commencent à redescendre. Mais surtout, le point optimal de déroulage n'est pas le même sur les deux modèles.

Les listages 3.2 et 3.3 présentent le code généré par clang/LLVM 3.4 pour le noyau présenté dans le listage 3.1. Le listage 3.2 présuppose l'utilisation d'un Cortex-A8, le listage 3.3 présuppose lui l'utilisation d'un Cortex-A15. Pour des raisons de lisibilité, seul le cœur de la boucle est présenté dans ces listages, les codes d'entrée, de sortie et de sortie de la boucle ont été supprimé. L'architecture du Cortex-A15 étant plus performante que celle du Cortex-A8, LLVM autorise par défaut un déroulage partiel de la boucle et calcul le résultat de 2 vecteurs par itération, alors que pour le Cortex-A8, seul un vecteur est manipulé à chaque itération. Si la plate-forme n'est pas précisément connue, cela pose des problèmes lorsque l'on cherche à l'anticiper au moment statique. Il faut en effet prévoir les différentes possibilités faisant ainsi grossir l'application.

Ce genre d'optimisation pourrait être faite de façon efficace et portable lors de l'installation ou au moment de l'exécution. Par exemple Android avec ART utilise le moment de l'installation pour pouvoir générer le code pour la plate-forme utilisée, et donc il est à même de faire ce genre d'adaptation.

3.3.3 Synthèse

Comme nous l'avons tout au long de cette section, la génération et la spécialisation dynamique de code peuvent améliorer les performances d'une application. Cette approche permet aussi de mieux gérer différents objectifs. Avec l'exemple du déroulage de boucle nous avons vu que connaître le modèle du processeur permet de choisir le facteur de déroulage de la boucle. Le temps de l'installation est le moment à partir duquel on a la possibilité de choisir ce facteur et dérouler la boucle. Vouloir le faire plus tôt est impossible dû au nombre de variantes de x86-64 qui existent et cela imposerait aussi de spécialiser statique le code, risquant ainsi de faire exploser la taille de l'application.

L'utilisation de la génération de code dynamique permet de gagner en performance mais à un certain coût. Mais bien souvent, seulement une métrique est utilisée dans l'étude des bénéfices et coûts d'une technique. Dans la suite du manuscrit nous présenterons une étude multi-objectifs sur les gains et coûts de l'utilisation de la génération de code dynamique dans un cas d'étude.

3.4 Conclusion

Dans ce chapitre nous avons vu certaines limitations existantes dans l'état de l'art vis-à-vis de l'optimisation des performances. L'utilisation d'un unique temps de génération n'est pas la stratégie la plus efficace pour maximiser ces performances. L'évolution des besoins et les limitations architecturales imposent de devoir optimiser les applications selon plusieurs objectifs à la fois. Bien que ces objectifs ne soient pas nouveaux en soit, ils restent qu'ils ne sont que très rarement étudiés ensemble lors de l'étude de l'efficacité d'une optimisation.

Pour les GPU, nous avons introduit l'influence de la forme des données sur les performances de l'algorithme de la GEMM. Dans le chapitre 5 nous proposons une stratégie d'amélioration de ces performances en tenant compte à la fois de la vitesse d'exécution et de l'impact sur la taille de cette stratégie. Pour gérer la taille de cette stratégie, nous utiliserons un outil de génération de code à la volée : `deGoal`. Cet outil sera présenté dans le chapitre 4.

Nous avons aussi vu les bénéfices de la génération dynamique de code. Cependant, nous avons soulevé la question de la mesure des gains et coûts selon plusieurs métriques. Le chapitre 6 va présenter l'étude multi-objectifs sur un Cortex-A8. Dans ce chapitre, nous étudierons les bénéfices et coûts de plusieurs approches de la génération de la code et selon 3 métriques : *a)* La vitesse d'exécution ; *b)* L'utilisation de la mémoire ; *c)* La consommation énergétique. Parmi ces approches, nous étudierons notamment `Kahuna`, un outil capable de créer automatiquement des générateurs de code à la volée à partir d'un programme annoté. `Kahuna` sera présenté dans le chapitre 4, après `deGoal`.

Pour résumer la suite du manuscrit, le chapitre suivant présente les deux outils (`deGoal` et `Kahuna`) qui ont été développés pendant cette thèse. Dans le chapitre 5, nous étudierons l'optimisation de l'algorithme de la GEMM sur GPU

Nvidia. Enfin dans le chapitre 6, nous étudierons l'impact plusieurs méthodes de spécialisations sur un noyau de calcul sur un Cortex-A8.

Chapitre 4

Outillages développés

Sommaire

Le chapitre précédent a présenté des limitations dans l'utilisation du matériel et la gestion des métriques. Un premier point concerne l'optimisation des performances de la GEMM sur GPU avec la non prise en compte de la taille des données. Nous avons identifié des besoins en outils pour gérer l'utilisation de multiples variantes afin d'éviter l'explosion de la taille de la solution. Le deuxième point est la non prise en compte de certaines métriques (mémoire, énergie) dans l'utilisation de la spécialisation de code, ce qui nous amène à vouloir connaître leurs impacts dans le domaine des systèmes embarqués. Lors de cette étude nous allons étudier l'utilisation de plusieurs méthodes de génération de code, dont `deGoal` et `Kahuna`. Ce dernier outil offre une approche alternative de la génération par rapport à `deGoal`.

Ce chapitre présente deux outils que nous avons développés, `deGoal` et `Kahuna`, pour permettre la génération et la spécialisation de code lors de l'exécution (ou à la volée). `deGoal` est un outil issue de HPBCG [19] initialement développé par Henri-Pierre Charles. `deGoal` présente une approche de la génération de code orientée générateur. L'accent est mis sur la génération du code en exprimant le générateur de code grâce à l'utilisation d'un DSL, plutôt que d'exprimer directement le résultat d'un point de vue fonctionnel. Dans le cadre de la thèse, j'ai effectué une refonte de l'outil en Python, en ajoutant la possibilité d'ajout de greffons et créé la partie arrière pour les GPU CUDA. `Kahuna` présente une approche de la génération de code orientée fonctionnelle, c'est-à-dire que l'on va exprimer ce que la fonction doit réaliser une fois spécialisée. L'outil permet la génération automatique de spécialiseurs (générateur de code dédier à la spécialisation de routines) à partir d'un code source annoté. Cette génération automatique permet de créer des spécialiseurs plus facilement, avec un effort de programmation moins élevé.

Les sections suivantes présenteront la motivation au développement de ces outils, puis l'outil `deGoal`, son langage et son architecture ainsi que la partie arrière PTX (pour cibler les processeurs graphiques Nvidia), puis l'outil `Kahuna` avec ses annotations et les problèmes liés à la génération automatique du générateur.

4.1 Motivation et approches

Le but de ces outils est d'avoir un contrôle supplémentaire sur le moment où le code sera généré avec un coût raisonnable et adapté au moment. À mesure que l'on se rapproche de la réalisation (c'est-à-dire le moment où les instructions seront exécutées), nous disposons d'informations supplémentaires sur la plate-forme, l'environnement du programme ou encore les données utilisées. Ces informations peuvent être utilisées pour optimiser le programme, cependant le temps disponible ou acceptable se réduit à mesure que l'on se rapproche de la réalisation. Les stratégies les plus utilisées de génération de code emploient généralement un unique moment pour générer leurs code :

Compilation statique : cas le plus courant avec la possibilité de lancer des optimisations lourdes mais génériques.

Installation : employée par les méthodes ayant besoin de temps pour affiner les performances de certains noyaux.

Exécution : le plus souvent cela passe par un JIT et le but n'est généralement plus la performance mais la portabilité.

Les interactions entre ces temps sont en règle générale limitées, souvent à cause de la complexité de la mise en œuvre de ces systèmes.

Le listage 4.1 présente une fonction template C++ effectuant la multiplication d'une valeur passée en argument par le paramètre du template. Lorsque l'on souhaite utiliser ce template, on précise la valeur du paramètre `b` et le compilateur produit une fonction spécialisée pour cette valeur au moment de la compilation statique.

Le listage 4.2 présente une implémentation d'une fonction FIR en C. Cette fonction est une convolution sur le signal 1D `in` en utilisant le noyau `ker` et stocke le résultat dans le tableau `out`. Le noyau à appliquer pour le filtre est passé

```
1 template<int b>
2 int mul_func(int a) {
3   return a * b;
4 }
```

LISTAGE 4.1 – Fonction de multiplication sous forme de *template C++*.

```
1  void fir(int* out, const int* in, unsigned len, char* ker,
       unsigned ker_len) {
2    for (unsigned i = 0; i < len; i++) {
3      int s = 0;
4      unsigned idx = 0;
5      if (i < ker_len)
6        idx = len - ker_len + i + 1;
7      else
8        idx = i - ker_len + 1;
9      for (unsigned j = 0; j < ker_len; j++, idx = (idx+1) %
           len) {
10       s += in[idx] * ker[j];
11     }
12     out[i] = s;
13   }
14 }
```

LISTAGE 4.2 – Implémentation générique d'un filtre FIR (Finite Impluse Response). Les éléments du tableau `ker` sont considérés comme constants.

en paramètre de la fonction et force le compilateur à (re)charger à chaque tour de boucle les coefficients du filtre même s'ils ne changent pas. La compilation statique permettrait de générer une fonction efficace pour ce genre de fonction si les coefficients sont connus à ce moment. En connaissant le noyau (sa longueur et ses coefficients) on peut alors dérouler la boucle de façon optimale et injecter les constantes dans le code. Mais cette stratégie employée au moment de la compilation statique pose la question de l'explosion mémoire et de l'adaptation de la routine aux usages. L'utilisation de plusieurs noyaux de tailles différentes ou de coefficients différents va créer une multitude de versions, faisant ainsi grossir la taille de l'application. De plus, si les noyaux utilisés sont amenés à changer, nous serons alors contraints de nous rabattre sur une implémentation générique, laissant les optimisations existantes inutiles.

L'anticipation permettrait dans ces cas d'avoir, à moindre coût, la capacité de produire ces fonctions spécialisées pendant l'exécution. Cela évite donc d'avoir à prédire toutes les valeurs qui pourront, potentiellement, être utilisées. De même, l'utilisation du filtre peut évoluer sans pour autant impacter les performances finales car l'adaptation se fait au besoin. La maîtrise de ces moments de la génération du code est essentielle afin de pouvoir la déclencher au moment opportun. Selon les conditions, des méthodes plus ou moins lourdes peuvent être employées avec différents niveaux d'optimisations possibles.

La suite du chapitre présente deux outils permettant, avec deux approches distinctes, un contrôle sur le moment où l'on souhaite déclencher cette géné-

ration du code avec un coût faible. L'identification, dans la phase statique (cf. figure 3.2) des valeurs qui deviendront constante pendant la phase dynamique, permet d'anticiper la production d'un code spécialisé et de réduire fortement le coût de génération. Ce coût faible permet leurs utilisations pendant l'exécution et autorise l'adaptation à différents modes d'utilisations et aux évolutions de l'usage de l'application.

4.2 Approche orientée générateur : deGoal

4.2.1 Présentation

`deGoal` [17] est un outil permettant la création de générateurs de code embarqués dans des applications pour permettre, entre autres, leurs spécialisations. Ces générateurs de code, appelés *compilettes*, ont pour objectif de produire du code durant l'exécution du programme avec un coût en temps et en consommation énergétique et mémoire le plus faible possible. Les compilettes spécialisent des morceaux de programme préalablement identifiés en effectuant de la vectorisation, en modifiant le graphe de flot de contrôle ou en injectant des valeurs inconnues lors la compilation statique mais connues et constantes au moment de la spécialisation.

L'outil inclut :

Un DSL : un méta-assembleur vectoriel générique. Le langage inclut notamment la capacité d'injecter des valeurs qui ne sont connues qu'à l'exécution en tant qu'opérande immédiat. Permettant ainsi de limiter les opérations de chargement et réduire la pression sur les registres.

Un outil de transformation : qui effectue principalement la transformation du langage `deGoal` vers du code C.

Un mini langage de description d'architecture : pour la description de l'encodage des instructions ainsi qu'un outil de génération partiel de la partie arrière à partir de ces descriptions.

Pour créer des compilettes, le processus de compilation et de génération de code a été décomposé en 4 étapes (présentées dans la figure 4.1) :

Écriture de la compilette : elle est faite à l'aide un langage spécifique inséré dans du code C standard.

Réécriture de la compilette : un outil, `degoaltoc`, réécrit la compilette en C standard.

Compilation statique : la compilette est compilée de façon statique avec un compilateur C classique (comme *GCC* ou *clang*).

Exécution : durant l'exécution du programme, la compilette utilise les informations de la plate-forme et les données de l'utilisateur pour créer, à la volée, un noyau spécialisé.

4.2.2 Écriture des compilettes

L'écriture des compilettes est faite grâce à un langage spécifique, décrivant les instructions à générer à l'exécution, entrelacé dans du code C, qui permet de contrôler le flot de génération. Afin de simplifier la présentation, nous l'étudierons par le biais d'un exemple.

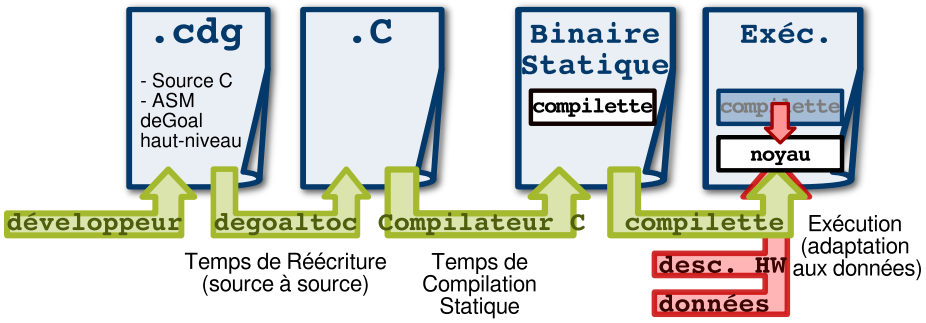


FIGURE 4.1 – Processus de compilation de `deGoal` (illustration de D. Couroussé).

```
1  typedef int (*pifi)(int);
2  pifi mul_compile(int b)
3  {
4    cdgInsnT *code= CDGALLOC(1024);
5    #[
6      Begin code Prelude
7      VectorType a int 32 1
8      RegAlloc a 1
9      Arg a0
10     mul a0, a0, #(b)
11     rtn
12     End
13   ]#;
14   return (pifi)code;
15 }
```

LISTAGE 4.3 – compilette produisant un code équivalent au listage 4.1.

Multiplication : Dans le code du listage 4.1 lorsque l'on souhaite utiliser ce template, on précisera la valeur du paramètre `b` et le compilateur produira alors une fonction spécialisée pour cette valeur au moment de la compilation statique.

Le listage 4.3 présente le code `deGoal` produisant la même fonction lors de l'exécution que le template C++ précédent (listage 4.1). Contrairement aux templates, la valeur de `b` n'a pas besoin d'être connue lors de la compilation statique.

Les symboles `#[` et `]#` (lignes 5 et 13) déclarent un bloc `deGoal` dans lequel est inséré les instructions à générer :

Ligne 6 : L'instruction `Begin` identifie le démarrage d'une *compilette*. L'argument `code` est un pointeur vers une zone mémoire où seront insérées les instructions et l'argument `Prelude` configure `deGoal` pour insérer le prolog et épilog de la fonction.

Ligne 7 : `VectorType` définit un type nommé `a`, de type entier de largeur 32 bits et longueur 1.

Ligne 8 : `RegAlloc` crée 1 registre de type et de nom `a`.

Ligne 9 : `Arg` définit le registre `a0` (définie de façon implicite à la ligne 8).

```
1  typedef int (*pifi)(int);
2  pifi op_compile(int b, char op)
3  {
4    cdgInsnT *code= CDGALLOC(1024);
5    #[
6      Begin code Prelude
7      VectorType a int 32 1
8      RegAlloc a 1
9      Arg a0
10   ]#
11   if (op == '+') #[ add a0, a0, #(b) ]#
12   else #[ mul a0, a0, #(b) ]#
13   #[
14     rtn
15     End
16   ]#;
17   return (pifi)code;
18 }
```

LISTAGE 4.4 – compilette similaire au listage 4.3 mais permettant, à l'exécution, de choisir entre une addition et une multiplication.

Ligne 10 : La construction `#(...)` permet d'évaluer une expression C et d'injecter son résultat en tant que valeur immédiate. À cette ligne, on multiplie le registre `a0` par le scalaire `b` donné en argument de la compilette et qui ne sera connue qu'à l'exécution.

Ligne 11 : La valeur `a0` est retournée.

Ligne 12 : `End` met fin à la génération du code de la fonction. À ce moment, `deGoal` effectue les opérations de finalisation, comme les calculs des sauts ou l'invalidation du cache.

Contrôle du flot de génération : Une compilette peut être composée de plusieurs blocs `deGoal` comme illustré dans le listage 4.4. Cette compilette reprend celle du listage 4.3 mais autorise la sélection de l'opérateur grâce à l'argument `op`. Le premier bloc `deGoal` se termine à la ligne 10, après cette ligne, nous revenons à du code C classique. Le `if` de la ligne 11 introduit une alternative et permet de choisir le bloc `deGoal` qui sera évalué et donc les instructions qui seront générées.

4.2.3 Réécriture : architecture de *degoaltoc*

degoaltoc est un outil de réécriture source à source permettant de transformer un code `deGoal` (fichier .cdg de la figure 4.1) en code C. Il est composé de 3 étages : l'analyse syntaxique, une phase de transformation et une phase de génération de code. Les deux derniers étages sont conçus pour être génériques et possèdent un certain nombre de crochets permettant l'inclusion d'analyses et de personnalisations en fonction des besoins de la cible.

Analyse syntaxique : Cet étage effectue l'analyse du code `deGoal` et construit l'arbre syntaxique abstrait. L'outil ne de contrôle que le code `deGoal`, le code C environnant n'est pas analysé pour le moment. Cet étage est indépendant de toute cible.

Analyse & transformation : Cet étage permet d'effectuer des analyses de l'arbre et de le modifier. Par défaut seulement la liaison déclaration/utilisation des variables et une vérification de type est effectuée. La liaison déclaration/utilisation des variables va s'assurer que les variables utilisées sont bien définies et leurs types valides. Dans la mesure du possible, cette phase s'assure aussi que les registres utilisés ne sont pas hors limites. Du faite de la définition du langage `deGoal` ces phases peuvent être imprécises. La vérification peut assez rapidement reposer sur le développeur, ce qui est la contre partie de la souplesse de l'outil dans l'écriture des optimisations. Si l'on reprend les exemples des listages 4.3 et 4.4, il est aisé de vérifier que l'on reste dans les limites car tout est adressé statiquement, mais `deGoal` autorise aussi l'indexation des registres grâce à des variables. Ainsi, ligne 10 du listage 4.3, on pourrait aussi écrire `#[ mul a(idx), a(idx), #(b) #]`, avec `idx`, une variable C définie préalablement. L'indexation faite par variable (`idx`) rend l'analyse plus difficile. Pour autant, il reste des gardes lors de l'exécution qui préviennent de l'utilisation de registres inexistants, évitant ainsi l'émission d'un code invalide ou des erreurs de segmentations.

Selon les besoins, une partie arrière peut ajouter ces propres passes d'analyse et de transformation par le biais de crochets. Cela permet à une partie arrière d'ajouter des définitions ou encore mettre en place certaines limitations.

Génération de code : Ce dernier étage produit le code C correspondant à la *compilette*. Les instructions `deGoal` sont remplacées par des appels à des fonctions de la partie arrière correspondant aux instructions. Ces fonctions répondent à une nomenclature précise qui dépend de l'instruction et des types des opérandes (registre ou valeur immédiate). La définition des fonctions est spécifique à chaque partie arrière et elles sont chargées de gérer les instructions `deGoal` pour la production de code binaire en fonction des capacités de l'architecture utilisée au moment de l'exécution.

4.2.4 Exécution

A l'exécution (dernier bloc de la figure 4.1), la compilette a accès aux données de l'utilisateur et aux capacités de l'architecture. Lors de la génération, la partie arrière prend en charge la sélection des instructions, l'allocation des registres et éventuellement l'ordonnancement. Ces étages privilégient la rapidité et une faible empreinte mémoire à la précision des transformations.

Si on reprend l'exemple du listage 4.3, les listages 4.5 et 4.6 présentent les codes produits à l'exécution pour différentes valeurs de `b` pour une cible ARM.

Dans le cas du listage 4.5, le jeu d'instruction d'ARM n'autorise pas les immédiats comme opérande pour une multiplication. La valeur est donc mise dans un registre libre avant d'effectuer l'opération. Dans le cas du listage 4.6, la multiplication par 2 peut être changée par une instruction de décalage de bits à gauche de 1 (l'instruction `LSL`), ce qui permet d'accélérer le calcul.

```
1 MOVS r1 , #10
2 MUL r0 , r0 , r1
3 BX r14
```

LISTAGE 4.5 – Exemple de code machine ARM produit par la compilette du listage 4.3 pour `b = 10`.

```
1 LSL r0 , r0 , #1
2 BX r14
```

LISTAGE 4.6 – Exemple de code machine ARM produit par la compilette du listage 4.3 pour `b = 2`.

4.2.5 Port PTX

Pour l'expérimentation présentée dans le chapitre 5, j'ai créé la partie arrière de `deGoal` ciblant le `PTX` pour la réalisation d'applications `CUDA` pour les GPU Nvidia avec les compilettes. Le PTX est une représentation intermédiaire décrivant des noyaux `CUDA` et utilisée par Nvidia pour cibler ses GPU.

Cette cible présente la difficulté d'avoir un modèle de programmation très différent des modèles classiques comme le C. Le modèle `CUDA` est orienté processus, il inclut donc des variables permettant l'identification du bloc et du processus courant. Il utilise aussi plusieurs niveaux de mémoire, managées explicitement (ex. mémoire partagée) ou non (ex. mémoire locale).

Les compilettes ayant pour but une génération de code avec un coût le plus bas possible, il n'est pas souhaitable d'inclure des analyses de code lors de l'exécution pour déterminer où doivent résider les données ou comment agencer les processus. Ces informations sont donc remontées au niveau du langage, par l'ajout de variables implicites (pour l'identification des processus) et d'instructions (principalement pour la gestion de la mémoire).

L'encodage binaire des instructions des GPU Nvidia (`SASS`) n'étant pas rendu public, les compilettes émettent les programmes dans la représentation intermédiaire PTX. Cette représentation est la RI de plus bas niveau qui soit documentée et disponible publiquement. Cette représentation est textuelle et nécessite une phase de compilation avec le JIT Nvidia, ralentissant la génération du code final.

4.3 Approche orientée fonctionnelle : Kahuna

4.3.1 Introduction

L'avantage de `deGoal` est le contrôle fin qu'a le développeur sur la génération de son code. Un expert peut donc facilement écrire son générateur grâce à `deGoal` et produire un code de qualité lors de l'exécution. Cependant ce modèle de génération est contraignant :

- Il utilise un DSL bas niveau, ce qui est peu adapté à certaines situations, et rend le développement au moins aussi long qu'écrire un code assembleur ;
- Les bogues d'un générateur dynamique de code sont plus difficiles à éliminer.

Dans la pratique, ce modèle est efficace pour produire de petits noyaux [55, 22] qui sont intégrés dans des tâches plus complexes.

Ce constat nous a poussé à développer `Kahuna` un outil plus simple d'utilisation assurant un temps de développement plus court ou une utilisation plus simple dans le cas d'une phase d'optimisation. La contrepartie étant qu'il ne permet pas un niveau de contrôle sur le code généré similaire à `deGoal`.

`Kahuna` repose aussi sur le principe d'identification de constantes dans un programme. À la différence de `deGoal`, l'outil part du niveau fonctionnel, c'est-à-dire la façon dont le code produit doit se comporter une fois spécialisé, et non comment le générer. Comme pour `deGoal`, c'est au programmeur du générateur de déterminer quand le code doit être spécialisé. En dehors de la façon d'exprimer ce qui doit être spécialisé, le processus suivi est assez similaire à `deGoal` (figure 4.1).

Cette section introduit donc `Kahuna` [58], une extension du cadriciel LLVM qui permet la génération de spécialiseurs de code à l'exécution. On commencera par introduire LLVM et la raison de son utilisation, pour ensuite présenter comment sont décrites les fonctions à spécialiser avec `Kahuna` et nous présenterons les différentes étapes de la génération et leurs fonctionnements.

4.3.2 Retour sur LLVM

LLVM (Low Level Virtual Machine) est un cadriciel conçu pour la fabrication de compilateurs. Le projet fournit une bibliothèque pour la manipulation de sa représentation intermédiaire (LLVA), l'optimisation du programme et la génération de code. LLVM supporte plusieurs cibles comme ARM (32/64-bits, Thumb), PTX, Hexagone...

Une série d'outils est fournie avec le projet, notamment :

— `opt` permettant de lancer des analyses et des optimisations génériques (c'est-à-dire indépendant de la cible) sur des programmes LLVA.
— `llc` permettant la génération de code machine à partir de LLVA.
— `lli` fournissant un moteur de JIT pour des programmes LLVA.

Le cadriciel LLVM est utilisé dans de nombreux projets académiques et industriels. Au niveau industriel, LLVM est notamment utilisé et commercialisé par :

— Apple, où la suite LLVM est devenue la chaîne de compilation par défaut. LLVM est aussi utilisé dans leur chaîne de traitement graphique ;
— Nvidia, qui l'utilise aujourd'hui pour son compilateur CUDA et OpenCL ;
— Intel, qui l'utilise aussi pour son compilateur OpenCL.

Son utilisation dans le cadre de la spécialisation de programme peut être intéressant du fait de son architecture. LLVM est un outil reconnu pour ses performances, tant du point de vue de la qualité du code généré que pour sa vitesse de génération de code. Cependant LLVM ne permet que la conception de compilateur ou JIT classique, et bien qu'il fasse bon usage des ressources, il reste néanmoins qu'il possède un surcoût non négligeable. Néanmoins, les optimisations offertes par LLVM peuvent être utilisées pour créer des spécialiseurs de fonctions performants.

4.3.3 Vocabulaire

Avant d'entrer dans la présentation de l'outil, nous introduisons ou rappelons certains éléments de vocabulaire. Ces éléments sont généralement connus mais

nécessitent quelques précisions car le sens qui leur est donné ici est souvent plus large qu'à l'accoutumée et peut entraîner quelques confusions.

Spécialiseur : Un *Spécialiseur* est un générateur de code à l'exécution spécialisant une routine. Ici, un spécialiseur est le générateur de code produit par `Kahuna` à partir du code annoté.

Constante : Une *Constante* est un élément dont la valeur ne change pas dans **une fenêtre temporelle** donnée, par exemple pendant le cycle de vie de l'application, l'exécution du programme ou encore l'exécution d'une tâche. En règle générale, nous distinguons deux types de constantes :

Constante statique : connue au moment de la compilation statique qui peut donc être manipulée par le compilateur.

Constante à l'exécution : inconnue au moment de la compilation statique mais qui une fois connue ne change pas de valeur. Ce type de valeur n'est pas ou peu manipulable par un compilateur statique car il est contraint de tenir une posture conservatrice.

Le terme *constante* est donc à prendre au sens large, la fenêtre temporelle sera précisée au besoin.

Variable : Une *Variable* est un élément dont la valeur change dans **une fenêtre temporelle** donnée. De façon triviale, tous ce qui n'est pas constante est variable.

Instruction dynamique : Une *Instruction dynamique* est une instruction dont le résultat ne peut pas être déterminé lors de la spécialisation.

Inconnue : Une *Inconnue* est une valeur immédiate d'une instruction dynamique qui n'est connue qu'à l'exécution.

Concepteur : Un *Concepteur* est celui qui va fournir à la chaîne `Kahuna` le code annoté, par exemple un expert ou une passe d'optimisation.

Fonction Kahuna : Une *Fonction Kahuna* est une fonction annotée pour `Kahuna`.

Site de production : Un *Site de Production* est une routine spécialisant du code pendant l'exécution.

Patron : Un *Patron* est un bloc d'instructions dynamiques qui sera utilisé par le spécialiseur pour générer le code.

Îlot : Un *Îlot* est un bloc d'instructions constantes qui sera intégré au spécialiseur.

4.3.4 Présentation du principe de fonctionnement

Le principe de fonctionnement est présenté figure 4.2. `Kahuna` utilise une approche par patrons comme *DyC* ou *Tempo*. Le *concepteur* va annoter le code source du programme pour identifier les constantes du programme et l'emplacement du générateur de code. Dans la figure, les concepteurs sont la partie frontale et la passe d'optimisation qui produisent un code source en LLVA annoté. Ce code annoté passe par la chaîne de compilation de `Kahuna` pour finalement produire un exécutable statique avec les patrons et le spécialiseur au format binaire. À l'exécution, le spécialiseur utilise les données et les patrons pour produire un noyau spécialisé. Le déclenchement de la spécialisation est à la charge du concepteur.

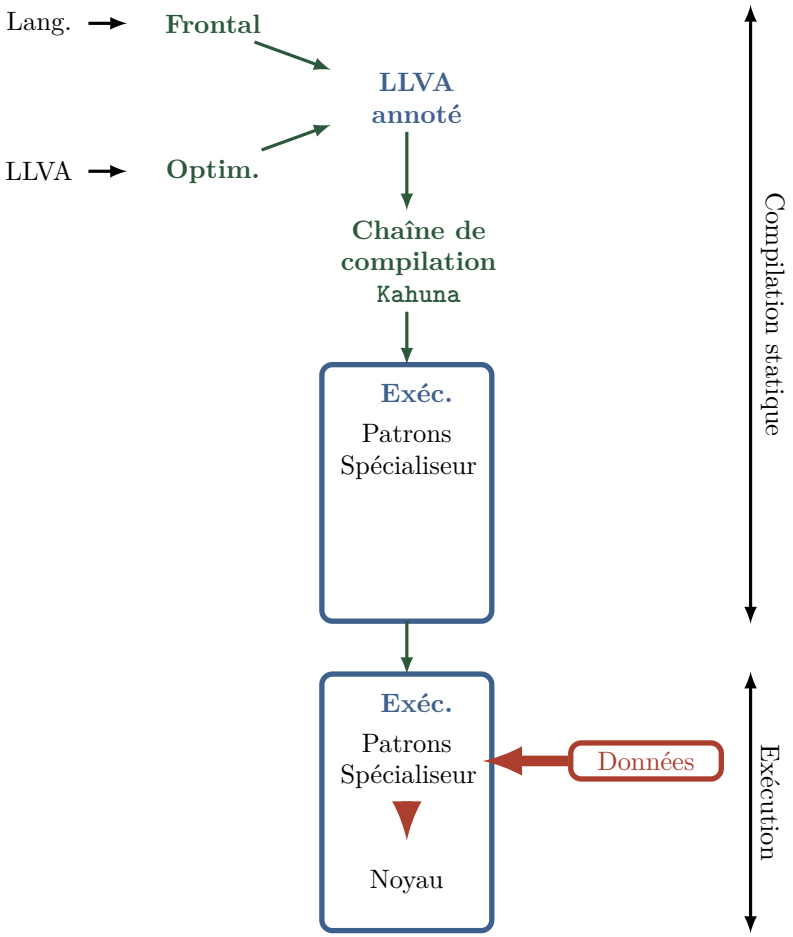


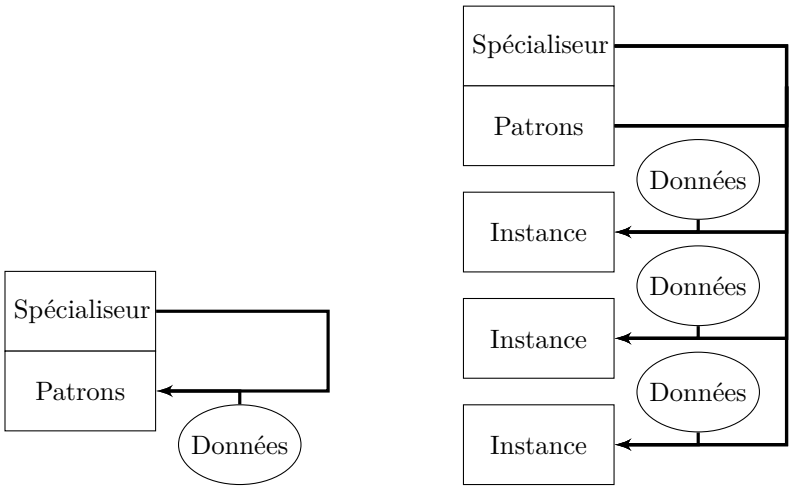
FIGURE 4.2 – Principe de fonctionnement de `Kahuna`.

`Kahuna` autorise deux modes de fonctionnement : spécialisation en-place (c'est-à-dire local) ou hors-place (c'est-à-dire dans une zone mémoire dédiée). La spécialisation en-place modifie le patron afin de produire une fonction valide (figure 4.3a), mais cela limite fortement les possibilités car elle est contrainte de faire les modifications localement, et ce, sans réordonner les instructions [1]. La spécialisation hors-place copie les instructions depuis le patron (figure 4.3b) et permet, à l'opposé du mode en-place, le réordonnancement avec un surcoût lié à la copie des instructions des patrons en sus du coût de la spécialisation en-place.

Les approches existantes dans l'état de l'art sont toutes basées sur un langage de haut niveau [2], C pour Tempo, DyC et `C et ML pour Fabius. Tempo et

1. Il est en réalité possible de réordonner les instructions, mais cela implique de conserver les informations de réordonnancement dans des structures de donnée, impliquant un surcoût mémoire non souhaitable dans ce contexte.

2. Langage est compris comme langage de haut niveau, c'est-à-dire un langage dans lequel un programmeur va volontiers utiliser pour écrire son programme, la RI restera la forme



(a) Spécialisation en-place. (b) Spécialisation hors-place.

FIGURE 4.3 – Les deux types de spécialisation de `Kahuna`.

DyC ont tous les deux leurs propres approches pour exprimer la spécialisation, construction de scénario pour le premier et annotation du code C pour le second. Tempo par son fonctionnement s'affranchit d'une adhérence au compilateur, DyC lui cumule à la fois une adhérence au langage et à son compilateur. `Kahuna` possède une adhérence à LLVM et naturellement à sa RI pour la création du spécialiseur à la compilation statique. Contrairement aux autres approches, il n'y a pas de d'adhérence avec un langage de haut niveau.

4.3.5 Description du flot de compilation

`Kahuna` a été conçu comme une extension de LLVM pour permettre une génération automatique de spécialiseur de code. La motivation première est d'avoir un outil complémentaire à `deGoal` plus facile d'utilisation, permettant de raccourcir les temps de développement et d'autoriser une interaction entre ces deux outils. L'extension fournit, en sus des modifications du cœur de LLVM, une série de passes d'analyses et de transformations dont le module de génération du générateur de code.

La chaîne de compilation de `Kahuna`, illustrée par la figure 4.4, part d'un programme écrit en LLVA annoté grâce à des fonctions intrinsèques. Cette RI est analysée lors d'une phase appelée "*temps de liaison*" ("*binding-time analysis*" en anglais). Cette phase permet de créer des *patrons*, qui serviront de base de code lors de la spécialisation, et des morceaux de code qui n'ont besoin d'être exécutés que lors de la spécialisation. Appelés *îlots*, ils seront donc insérés dans le spécialiseur de code. Les patrons sont compilés grâce à `llc` qui fournit les patrons au format binaire ainsi que des informations sur les instructions utilisées par ces patrons. Ces informations sont ensuite utilisées avec les îlots pour générer le spécialiseur.

adaptée au compilateur mais non à l'écriture d'un programme par un humain.

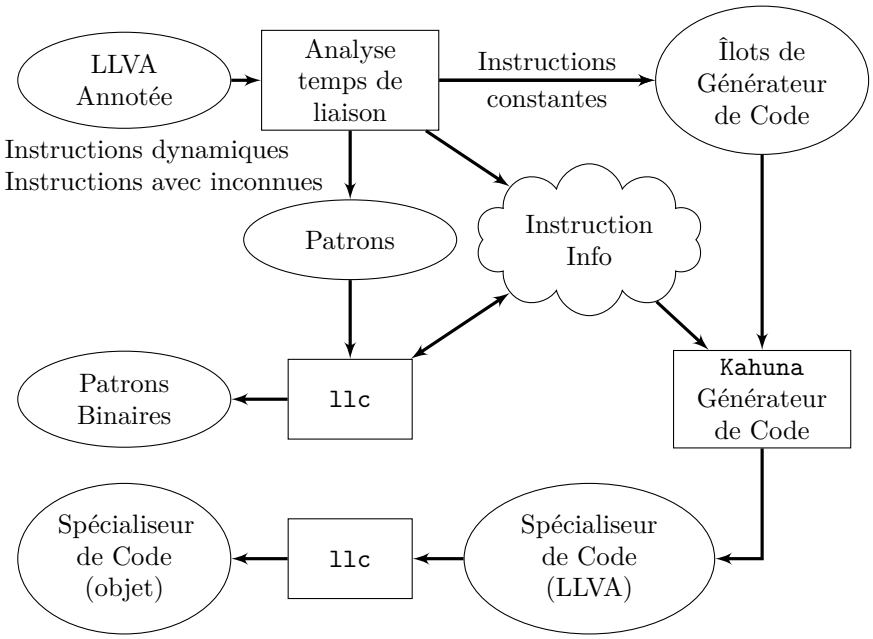


FIGURE 4.4 – Flot de compilation de `Kahuna`.

Architecture : `Kahuna` s'intègre dans LLVM à 2 niveaux :

- Dans l'infrastructure même où il intègre :
  - de nouvelles fonctions intrinsèques pour l'annotation du programme.
  - de nouvelles structures de données, pour la gestion du contexte et des informations relatives aux fonctions, blocs basiques et instructions.
  - les extensions dans l'étage de génération de code de LLVM pour la production des patrons.

  Ces modifications touchent le cœur de LLVM et donc ne peuvent que s'intégrer dans l'infrastructure.
- Sous forme de greffon, pour la manipulation du code lors de la compilation statique. Cela comprend toutes les passes d'analyses, de transformations et de génération propre à `Kahuna`. L'intérêt d'utiliser un greffon est la possibilité de séparer plus facilement les parties propres à `Kahuna` de celles propres à LLVM.

Dans le mode hors-ligne, il est aussi utilisable dans le cadre d'une application multi-processus.

Le but de `Kahuna` est la génération du spécialiseur de code. La gestion des codes spécialisés est considérée comme étant hors du domaine d'application de l'outil. Les applications et les capacités des cibles vont déterminer la manière dont sera utilisée le générateur, dans certains cas une seule instance sera utilisée à la fois, dans d'autres cela nécessitera un nombre indéterminable à l'avance. Dans ce dernier cas, il est indispensable de pouvoir gérer correctement le nombre d'instances disponibles, et ce en accord avec les possibilités de la plate-forme physique. De même le déclenchement de la spécialisation doit pouvoir être contrôlé pour se plier aux contraintes et besoins de la plate-forme ou de l'application.

Nom	Catégorie	Description
kahuna.irt.*	Identification des constantes	Constante de type entier
kahuna.fprt.*	Identification des constantes	Constante de type flottant
kahuna.ptr.*	Identification des constantes	Pointeur constant
kahuna.prodsite	Identification du générateur	Position dans le code où le générateur de code doit être inséré
kahuna.prodsite.iarg	Argument du générateur	Argument de type entier
kahuna.prodsite.farg	Argument du générateur	Argument de type flottant
kahuna.prodsite.parg	Argument du générateur	Argument de type pointeur

TABLE 4.1 – Liste des intrinsèques `Kahuna` servant à l'annotation des programmes.

4.3.6 Annotation du programme

Cette section introduit le système d'annotation du code LLVA. On traitera dans un premier temps la façon dont sont exprimées les fonctions kahuna et les constantes à l'exécution, puis comment sont exprimés les sites de productions avec les fonctions intrinsèques de LLVM. Puis dans un deuxième temps, nous détaillerons les annotations des constantes puis celles des sites de productions et nous terminerons avec la justification du choix de l'utilisation de ces fonctions intrinsèques.

`Kahuna` repose sur l'annotation du code LLVA pour exprimer et traiter des parties constantes à l'exécution d'une fonction. Une fonction kahuna peut être vu comme le "reflet" de la fonction spécialisée, c'est-à-dire que le prototype d'une fonction kahuna est celui de la fonction spécialisée. Par analogie avec les templates C++, le prototype de la fonction kahuna est constituée uniquement des arguments d'une fonction template C++. Les constantes à l'exécution sont récupérées grâce à l'appel de fonctions intrinsèques dans le corps de la fonction kahuna. L'expression équivalente du template C++ du listage 4.1 en fonction kahuna est présentée dans le listage 4.7. À la ligne 1 nous avons le prototype de la fonction kahuna qui correspond à l'argument de la fonction template. Le paramètre "b" de la fonction template est "*invoqué*" à la ligne 3 de la fonction kahuna, la valeur retournée est alors interprétée comme une constante à l'exécution. L'intrinsèque possède 2 arguments : le premier correspond à l'identifiant du site de production, le deuxième, l'identifiant de l'argument à utiliser.

L'expression de l'emplacement et de la configuration du *site de production* ce fait aussi par l'utilisation d'une fonction intrinsèque. Le listage 4.8 présente la déclaration et la configuration de l'emplacement du site de production pour spécialiser (en-place) la fonction kahuna du listage 4.7. La ligne 3 lie la variable `%var` avec la constante à l'exécution %b. La ligne 4 déclare l'endroit où doit être inséré le spécialiseur. Ces intrinsèques seront remplacés par le spécialiseur

```
1 define i32 @cst_mul(i32 %a) nounwind readnone {
2 entry:
3   %b = tail call i32 @llvm.kahuna.irtcst.off.i32(metadata !"
        code_gen", i32 0)
4   %mul = mul nsw i32 %b, %a
5   ret i32 %mul
6 }
```

LISTAGE 4.7 – Fonction kahuna équivalente à la fonction du listage 4.1.

```
1 define void @apply_cst_mul(i32 %cst) nounwind {
2 entry:
3   tail call void @llvm.kahuna.prodsite.iarg.i32(metadata !"
        code_gen", 0, %cst)
4   %0 = tail call i8* @llvm.kahuna.prodsite(metadata !"
        code_gen"), !kahunaKind !1
5   ret void
6 }
7 !1 = metadata !{metadata !"InPlaceSpecialization", i32 1}
```

LISTAGE 4.8 – Identification du site de production de la fonction kahuna du listage 4.7.

complet à l'issue des phases de générations.

L'annotation des programmes se fait donc uniquement via des fonctions intrinsèques qui offrent l'avantage d'être vues comme des fonctions. Cela assure ainsi une interopérabilitée avec les autres phases de LLVM sans avoir d'impact important sur les optimisations réalisées par LLVM. Le tableau 4.1 résume les principaux intrinsèques de `Kahuna`. Elles tombent dans 2 catégories :

- identification des variables constantes à l'exécution. Elles vont permettre d'invoquer des valeurs qui serons interprétées comme constante à l'exécution.
- identification du site de production (générateur de code à l'exécution). Elles vont permettre de définir la fonction dans laquelle le spécialiseur doit être créé et de configurer le spécialiseur.

Ces intrinsèques sont ceux que le concepteur utilise pour annoter le programme. D'autres types d'intrinsèques sont utilisés au fils des transformations et seront introduit en temps voulu.

4.3.6.1 Identification des constantes

Les intrinsèques d'identification de variables se déclinent en fonction du type qu'elles gèrent [3]. Dans tous les cas, chaque intrinsèque prend en arguments :

- L'identifiant du site de production auquel il est rattaché.
- L'argument qui doit être utilisé par le site de production.

La fonction produit une valeur qui est utilisée comme une variable par LLVM et donc est gérée de manière transparente. La variable étant issue d'une fonction sans effets de bord, certaines optimisations peuvent les utiliser normalement,

3. En raison du système de typage de LLVM.

notamment dans le cas de l'élimination du code mort (c'est-à-dire variable non utilisée).

4.3.6.2 Site de production

Un site de production est un emplacement dans le corps d'une fonction où sera inséré le spécialiseur. Le spécialiseur qui sera lancé à l'exécution est identifié par un intrinsèque. Le spécialiseur est inséré en lieu et place de l'intrinsèque dans le flot de la fonction. L'intérêt est de pouvoir insérer du code métier avant l'utilisation du spécialiseur comme l'extraction ou la transformation préalable de données.

Comme le montre le tableau 4.1, il y a deux types d'intrinsèques qui servent dans l'identification des sites de productions :

kahuna.prodsite : cet intrinsèque indique où doit être généré le spécialiseur. Cet intrinsèque porte en argument un identifiant et des méta-données portant sa configuration (en-place ou hors-place).

kahuna.prodsite. ?arg : ces intrinsèques [4] vont lier les constantes invoquées par les intrinsèques d'identification des constantes avec le spécialiseur. Ces intrinsèques prennent en argument l'identifiant du site de production auquel elles sont rattachées et l'identifiant de la constante utilisée dans la fonction kahuna. Cette approche permet d'effectuer un traitement préalable sur les données avant qu'elles ne soient utilisées par le spécialiseur.

Le listage 4.8 présente un exemple d'insertion du générateur de code. La fonction *apply_cst_mul* est une fonction standard qui est appelée par le concepteur pour spécialiser sa fonction. Le générateur de code sera intégré en lieu et place de l'intrinsèque à la ligne 4. L'instruction possède aussi des méta-données associée pour sa configuration, ici elle indique que la spécialisation doit être faite en place.

4.3.6.3 Justification de l'utilisation des intrinsèques

Deux choix auraient été possibles pour exprimer si une valeur est une constante ou non :

— soit en créant de nouveaux types qui porteraient cette information ;
— soit en utilisant les intrinsèques.

Le statut de "constante" étant une propriété de la valeur, cela aurait un sens qu'il soit reflété au niveau du type. D'un point de vue technique cela complique l'implémentation en raison la taille de LLVM (la version 3.4.2 fait plus d'un million de lignes de codes). En effet, l'ensemble des optimisations et des parties arrières aurai dû être modifiés pour tenir compte de cette propriété.

Grâce à l'utilisation du SSA, il est trivial de retrouver la définition d'une valeur dans le flot de contrôle de la fonction. Les valeurs définies par des intrinsèques sont donc facilement identifiables et il en va de même pour la propagation de leurs statuts de constante à l'exécution. Techniquement, les intrinsèques sont transparentes pour les optimisations car elles sont vues comme des fonctions avec des propriétés particulières et connues (présence d'effet de bord par

4. Cette approche est liée à une limitation technique de LLVM qui ne gère pas correctement les intrinsèques avec un nombre d'arguments variables.

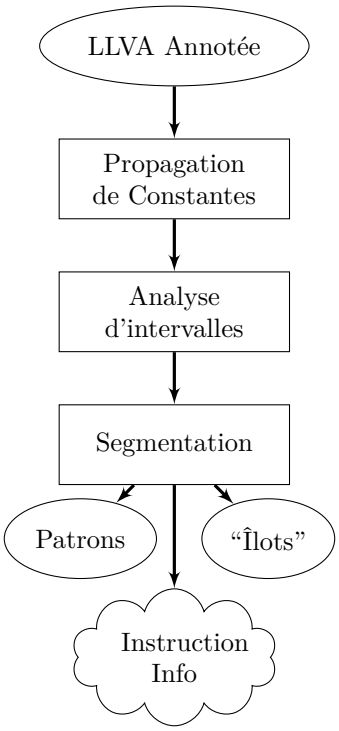


FIGURE 4.5 – Flot de travail de l'analyse du temps de liaison.

exemple). À défaut de pouvoir profiter correctement de toutes les optimisations, les intrinsèques ne font pas échouer le processus de compilation.

4.3.7 Temps de liaison

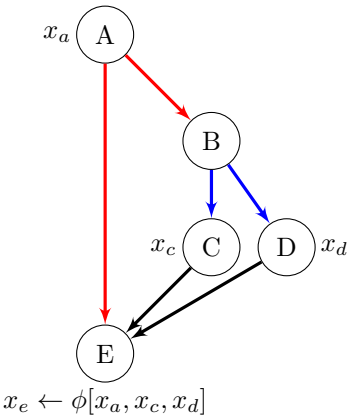
L'analyse du temps de liaison est une série de phases d'analyses et de transformations destinées principalement à identifier et à séparer les parties constantes des parties dynamiques d'un programme annoté. Les phases de cette analyse sont présentées dans la figure 4.5.

Une partie constante est composée d'instructions utilisant uniquement des constantes statiques ou des constantes connues à l'exécution. Une partie dynamique est composée d'instructions utilisant des variables et éventuellement des constantes. Dans le cas où une instruction dynamique utilise une constante connue à l'exécution, on parle alors d'une instruction avec inconnue.

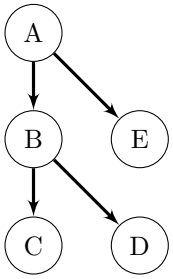
À l'issue de cette phase, les instructions dynamiques sont regroupées dans les *patrons*, les instructions constantes dans les *îlots* et une base de données (*instruction info*) qui contient des informations nécessaires à la liaison entre les patrons et les îlots.

4.3.7.1 Propagation de constantes

Cette phase analyse et étiquette les instructions en fonction de leurs statuts :
— Constante statique (C)
— Constante à l'exécution (CE)



(a) Graphe de flot de contrôle. Les arêtes rouges représentent un saut contrôlé par une constante. Les arêtes bleues représentent un saut contrôlé par une variable. Les arêtes noires représentent des sauts inconditionnels.



(b) Arbre de domination.

FIGURE 4.6 – Exemple de graphe de flot de contrôle.

— Dynamique (D) (c'est-à-dire variables dont la valeur ne peut être déterminée lors de la spécialisation)

Le statut d'un opérateur n'ayant pas d'effet de bord est déterminé par le statut de ses opérandes avec les règles suivantes :

	C	CE	D
C	C	CE	D
CE	CE	CE	D
D	D	D	D

Les appels de fonctions sont considérés comme constants (statique ou à l'exécution) si elles n'ont pas d'effets de bords et ses opérandes sont constants.

Instructions ϕ : Le principe de fonctionnement du SSA repose sur la propriété de domination, c'est-à-dire, toute utilisation d'un registre est dominé par sa définition. Aux frontières de domination, les instructions "ϕ" sont insérées pour prendre en compte le graphe de flot de contrôle [4]. Dans l'exemple présenté dans la figure 4.6a, considérons que les variables x_b, x_d et x_e sont constantes. Il apparaît rapidement que l'unique utilisation du statut des opérandes comme critère est insuffisant et qu'il nous faut connaître l'origine de la valeur, ou plus précisément, le chemin qui est parcouru entre l'entrée du programme et l'instruction ϕ. Comme pour la connaissance des valeurs, il est encore question du moment où ce chemin devient connu.

Le cas trivial est celui où l'un des opérandes de l'instruction est dynamique. Il existe alors au moins 1 cas où l'on ne pourra, à la génération du code pendant l'exécution, déterminer quelle sera la valeur de la variable. Le résultat est donc dynamique.

```
1 %0 = tail call i8 @llvm.kahuna.irtcst.off.i8(metadata !"
      code_gen", i32 0)
2 %conv = zext i8 %0 to i32
3 %mul = mul nsw i32 %conv, %val
```

LISTAGE 4.9 – Illustration de la coercition dans LLVM.

```
1 %0 = call i8 @llvm.kahuna.irtcst.input.mark.i8(...)
2 %conv = zext i8 %0 to i32
3 call void @llvm.kahuna.irtcst.mark.i32(...)
```

LISTAGE 4.10 – Instructions constantes du listage 4.9 allant dans un îlot.

```
1 %0 = tail call i32 @llvm.kahuna.irtcst.inh.i32(...)
2 %mul = mul nsw i32 %0, %val
```

LISTAGE 4.11 – Instructions dynamiques du listage 4.9 allant dans le patron.

Dans l'exemple présenté dans la figure 4.6a, ce qui nous intéresse est de savoir s'il existe un chemin avec au moins un branchement dynamique qui nous empêcherait de déterminer, lors de la spécialisation à l'exécution, la valeur d'une variable. Pour savoir le statut d'une instruction ϕ, une solution satisfaisante est de prendre le dominateur commun des prédécesseurs du nœud "ϕ". Soit x définie par l'instruction *phi* :

$$z = cdom(op(x))$$

où *cdom* renvoie le dominateur commun d'un ensemble de nœuds et *op* l'ensemble des opérandes du noeud ϕ définissant x. Soit X_y l'ensemble des nœuds entre z et y un des opérandes de x dominées par z.

$$y \in op(x); \quad z \in dom(y); \quad X_y = \{node(P_{z \to y})\}$$

Soit Y_y le sous-ensemble des nœuds de X_y tel que pour chaque nœud $n \in Y_y$ il existe un chemin entre z et x passant par n mais pas par y. On définit l'ensemble Y comme étant l'union des Y_y :

$$Y = \bigcup_{y \in op(x)} Y_y$$

Le statut de x est CE si et seulement si tous les branchements des noeuds de Y sont C ou CE. À contrario, le statut de x est D si et seulement s'il existe un branchement parmi les noeuds de X qui est D.

$$status(x) = \begin{cases} D & si \quad \exists n \in Y; \quad status(term(n)) = D \\ CE & sinon \end{cases}$$

4.3.7.2 Analyse d'intervalles

Un effet de bord de la formation des patrons et des îlots est la perte d'information sur les valeurs utilisées, notamment à cause de la coercition. Le listage 4.9 donne un exemple de coercition simple, mais courant dans les programmes LLVA. Lors de la phase de segmentation, les instructions lignes 1 et 2

seront supprimées et remplacées par un intrinsèque de type *i32* afin de respecter le système de type de LLVM. Les listages 4.10 et 4.11 montrent les instructions déplacées (respectivement, instructions constantes et instructions dynamiques), les fonctions intrinsèques présentes dans ces listages servent à faire le lien entre les îlots et patrons. Cette opération permet de simplifier la compilation des patrons mais provoque une perte d'information sur l'inconnue lors de la compilation statique. La posture conservatrice imposerait de gérer les valeurs tenant sur 32 bits alors que, de part la définition de l'intrinsèque à l'origine, les valeurs manipulées tiennent sur 8 bits.

Pour l'instant cette phase n'effectue que ce genre d'analyse simple, c'est-à-dire qu'on va garder le nombre de bits pouvant être changé. Dans l'exemple du listage 4.9, le registre %0 utilise 8 bits, le registre %conv est de type `i32` et utilise donc 32 bits, mais comme il est le résultat de la conversion de %0 en `i32`, il tient en réalité sur 8 bits.

Il pourrait être intéressant d'étudier et de mesurer l'impact d'analyse plus poussée, comme celle faite par [79], et qui permettrai de gérer au mieux les instructions pouvant être produites.

4.3.7.3 Segmentation

La dernière phase de l'analyse du temps de liaison est la séparation des différents éléments pour former les îlots (pour le site de production), les patrons et le lien entre ces deux éléments sous la forme d'une base de donnée.

Génération des îlots : Les constantes sont retirées du programme source et sont placées dans des "îlots" avec des entrées/sorties présentées sous forme d'intrinsèques. Les intrinsèques de sorties possèdent un identifiant pour permettre les connections avec les patrons.

Génération des patrons : Une fois les constantes identifiées, elles sont retirées du programme source et remplacées par des intrinsèques pour identifier les inconnues. Ces intrinsèques permettent de connecter la valeur produite par le site de production aux instructions dynamiques les utilisant, le lien étant assuré par la base de données sur les instructions et les intrinsèques de sorties des îlots.

Base de données des instructions : Les informations sur les intrinsèques qui ont été accumulées durant cette phase sont stockées dans une base de donnée, notamment les résultats de l'analyse d'intervalle.

4.3.8 Compilation des patrons

La compilation des patrons a été intégrée dans l'étage de génération de code (partie arrière) de LLVM. Plusieurs phases ont été modifiées pour prendre en compte les intrinsèques `Kahuna` présentés dans la section précédente, notamment la sélection d'instructions et la phase d'émission du code objet.

À l'issue de cette phase, les patrons sont stockés au format binaire. Certaines informations réutilisées dans la phase de génération du site de production sont stockées dans la base de données, notamment :

— Les symboles identifiant les patrons.

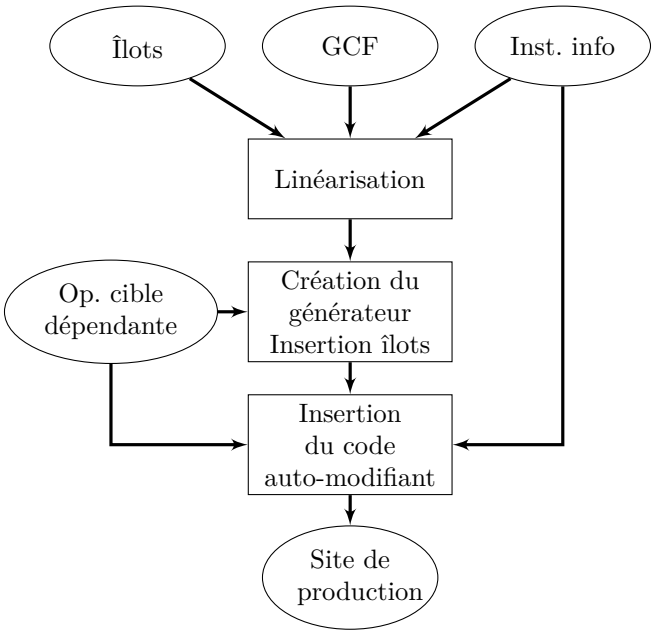


FIGURE 4.7 – Processus de génération du site de production.

— L'association entre les patrons et les blocs basiques de la représentation intermédiaire LLVM. Cette information est nécessaire car certaines optimisations de la partie arrière peuvent créer, supprimer ou fusionner des blocs basiques.
— Les opérations à effectuer pour l'encodage des instructions présentant des inconnues. Dans le cas où plusieurs instructions sont possibles, les contraintes à respecter sont aussi présentes.

Ces informations permettront la génération du site de production.

4.3.9 Génération du site de production

Cette dernière phase, spécifique à `Kahuna`, réunit les informations générées lors des précédentes phases pour générer le site de production. Elle se compose en trois grandes étapes : la linéarisation du flot de contrôle, la génération du spécialiseur avec l'inclusion des îlots, puis la création du code auto-modifiant pour gérer les inconnues. Cette dernière phase génère le site de production dans un programme LLVA portable.

Linéarisation : La phase de linéarisation ordonnance le graphe de flot de contrôle des fonctions à générer de façon à pouvoir visiter lors de l'exécution les blocs basiques requis sans avoir à maintenir des structures de contrôle. Les blocs basiques dont les instructions de branchement sont contrôlées par des variables imposent de visiter tous ces blocs basiques successeurs. A contrario, ceux qui dépendent de constantes nous permettent de ne visiter que le bloc basique successeur requit, ce qui nous permet de supprimer le code mort.

Dans l'exemple présenté en figure 4.6a, lorsque l'on visite le bloc basique A,

la constante déterminera si nous visiterons B ou C. Dans le cas où l'on visite C, nous devons alors visiter D et E.

Création du spécialiseur : En utilisant le résultat de la linéarisation, le spécialiseur est généré en prenant en compte les spécificités de la cible (comme l'alignement par exemple). En suivant le flot issu de la linéarisation, les îlots sont insérés dans le générateur en parallèle des copies de patrons dans le cas d'un générateur hors-place.

Insertion du code auto-modifiant : Cette étape insère les instructions permettant de fixer les inconnues des patrons aux emplacements des intrinsèques de sorties. Pour chaque couple intrinsèque de sortie et inconnue, cette phase produit :

Le code permettant la sélection des instructions : Selon la valeur prise par l'inconnue certaines instructions pourront être utilisées ou non. Les instructions utilisables ont été identifiées lors de la compilation des patrons avec les contraintes à respecter pour pouvoir les utiliser (voir section 4.3.8). Le générateur peut ainsi insérer les instructions les plus performantes si les constantes respectent les contraintes pour les utiliser, le cas échéant le générateur utilise une solution moins efficace mais plus souple sur les contraintes. Ce point était encore en développement au moment de la rédaction : la capacité de gérer les variantes au niveau du spécialiseur est faite, mais il reste encore à gérer correctement les variantes dans la partie arrière de LLVM. Nous y reviendrons dans le chapitre 6 lors de la discussion sur les évolutions possibles de `Kahuna`.

Le code encodant les instructions : Ces instructions sont ensuite écrites dans le code final en respectant les spécificités de l'architecture cible.

Dans le cas de la génération d'un spécialiseur en-place, les patrons binaires sont insérés dans une section spécifique avec les droits de lecture, écriture et exécutions.

Produit final : Finalement, on obtient le site de production écrit en LLVA. Ce code est optimisé en utilisant les passes standards de LLVM, puis il est compilé via `llc`. L'édition des liens entre le site de production et les patrons est faite avec un éditeur de liens standard. Pour éviter des effets indésirables ces optimisations sont désactivées afin d'éviter les transformations qui rendraient invalides les patrons ou les spécialiseurs.

4.3.10 Frontal

Des extensions dans `clang` pour C/C++ ont été réalisées pour l'utilisation de l'outil avec un langage de haut niveau. Cela nous permet de faciliter la réalisation des tests et des expérimentations. Ces extensions prennent la forme d'intrinsèques et expriment les mêmes fonctionnalités et sémantiques que pour `Kahuna`. À titre d'exemple, le listage 4.12 présente la forme C de l'exemple du listage 4.7 qui spécialise la fonction de multiplication. Seul la ligne 1 a été changée en fournissant les définitions et macro d'aide à l'utilisation de l'extension clang. La ligne 4 qui déclare une variable constante à l'exécution `b` de type `int` grâce à la macro `KAHUNA_INPUT_I`.

```
1 #include "kahuna/kahuna_cc.h"
2
3 int cst_mul(int a) {
4   KAHUNA_INPUT_I(int, b, "code_gen", 0);
5   return a * b;
6 }
```

LISTAGE 4.12 – Fonction C annotée. `KAHUNA_INPUT_I` est une macro visant a simplifier un peu l'usage de l'intrinsèque.

4.4 Conclusion

Dans ce chapitre nous avons présenté `deGoal` et `Kahuna`, les deux outils que nous avons développés. Ces deux outils utilisent deux approches distinctes mais complémentaires.

`deGoal` présente une approche orientée générateur. Le développeur exprime ce qu'il souhaite générer à la volée, lui laissant un contrôle fin sur le résultat final. Le langage et les outils associés permettent l'automatisation des tâches de générations et d'encodages des instructions, ce qui permet de ne pas faire accroître la complexité de la programmation comparée à l'écriture de programmes en assembleur.

`Kahuna` présente une approche fonctionnelle. D'une façon similaire aux templates C++, le développeur écrit l'aspect fonctionnel de la fonction à spécialiser. Il n'a donc aucun contrôle direct sur la façon dont le code doit être généré. Cette forme d'expression est plus simple et de plus haut niveau que `deGoal`, donnant lieu à des programmes plus simples et rapides à écrire.

L'idéal serait de pouvoir assurer une interaction entre `Kahuna` et `deGoal`, pouvant ainsi assurer une automatisation de la spécialisation sur des parties critiques mais facilement analysables et l'intervention d'un expert sur les parties plus complexes et hors de portée d'un outil automatique.

Dans les chapitres suivants nous traiterons des expérimentations effectuées avec ces outils. Dans le chapitre 5, nous utiliserons `deGoal` dans le cadre plus générale de l'optimisation de la GEMM sur GPU afin de contrôler la taille de la bibliothèque finale. Dans le chapitre 6, nous utiliserons `deGoal` et `Kahuna` dans le cadre d'une étude sur l'impact de l'utilisation de ces générateurs selon plusieurs métriques. Ces deux outils présente une approche distincte de la spécialisation de code, `Kahuna` offre moins de contrôle sur le code généré que `deGoal`, mais permet un développement plus rapide de l'application. Ce chapitre 6 permettra donc de montrer les avantages de ces approches sur l'utilisation des ressources mémoire, l'amélioration des vitesses d'exécution et la consommation énergétique.

Chapitre 5

Adaptation de l'algorithme du produit matriciel aux données sur GPU

Sommaire

Dans leurs course à la performance, les architectures classiques (CPU) faisant face aux limitations matérielles du silicium ont commencé à, modestement, augmenter les cœurs de calcul dans leurs puces. Le parallélisme devient alors une composante essentielle dans l'augmentation des performances des processeurs. Face à la progression plus lente de la puissance de calcul des cœurs de calculs des CPU, les processeurs graphiques (GPU) sont apparus comme une alternative viable pour améliorer de façon drastique les performances.

Les GPU sont des unités de calculs conçus pour gérer, manipuler une grande quantité de processus. Là où un CPU moderne gère efficacement moins de 20 processus, les GPU peuvent en gérer plusieurs centaines de milliers. Il en résulte une façon radicalement différente de penser le calcul sur ces plates-formes.

Dans le chapitre 3 nous avions vu qu'il existe une limitation dans l'exploitation du matériel avec l'algorithme de la GEMM liée à la non prise en compte de la tailles des données. Dans le chapitre 4 nous avons présenté `deGoal`, un générateur de code à la volée, dans le but de contrôler la génération de l'implémentation de la GEMM en fonction des données et ainsi contrôler la taille de la bibliothèque.

Dans ce chapitre, nous nous intéressons au lien entre taille des données et la façon de configurer ces algorithmes parallèles. Par configuration, nous entendons un ensemble de paramètre d'optimisations comme le tuillage ou la manière d'organiser les opérations de chargement. Ce lien est souvent ignoré car il est souvent difficile de l'exploiter à cause de la difficulté de la mise en œuvre d'un protocole de recherche, du temps requis et du coût mémoire. De plus, cela nécessite l'utilisation de nombreuses variantes pour un même algorithme, et donc la capacité de pouvoir les générer de façon efficaces, mais aussi d'être capable de choisir la bonne variante en fonction des données. La découverte des variantes à utiliser va donc ce faire grâce à un algorithme d'auto-tuning, un algorithme d'apprentissage permettra ensuite la création d'une fonction d'inférence pour déterminer quelle variante utiliser en fonction des données.

Ce chapitre commence par présenter la programmation CUDA et comment un noyau CUDA est exécuté sur la carte. Ceci nous permettra d'introduire les difficultés pour l'optimisation des performances sur ces architectures, plus particulièrement, l'équilibre entre nombre de processus déployés et l'ILP présent dans un noyau de calcul. Nous présenterons ensuite l'algorithme de la `GEMM`, qui sera l'objet de notre étude, son implémentation pour GPU CUDA et la mise en évidence du lien données et performances. On présentera ensuite la solution proposée. Nous présenterons ensuite le principe de l'auto-tuning et de l'apprentissage machine pour justifier leurs utilisations dans la solution. Nous ferons enfin son évaluation et nous terminerons sur les pistes d'évolution et d'amélioration possibles avant de conclure ce chapitre.

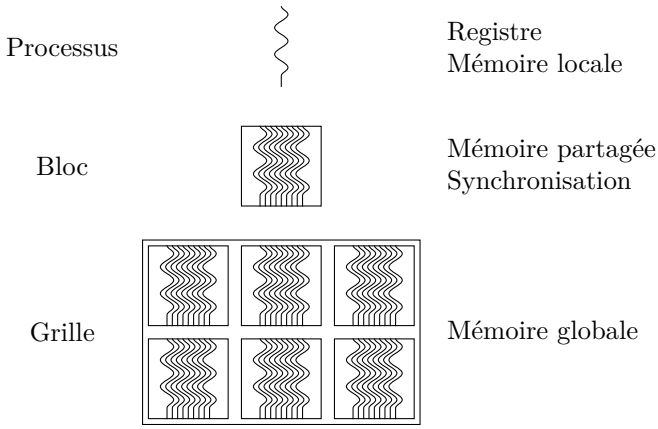


FIGURE 5.1 – Hiérarchie des processus dans CUDA.

5.1 Programmation des GPU

5.1.1 Présentation

Les GPU sont des unités de calculs dédiées au rendu graphique. Ces calculs étant hautement parallèles leurs architectures ont été spécialisées autour de cette idée. Entre les années 2003-2007, divers projets ont cherché à utiliser leurs capacités de calculs parallèles, BrookGPU [15] et Lib Sh [61] sont deux exemples très connus. L'architecture n'étant pas pensée pour être utilisée en dehors du cadre du calcul graphique. Ils subissaient quelques limitations, notamment vis-à-vis des calculs à virgule flottante qui soit étaient inexistantes soit non conformes avec la norme IEEE 754-2008.

Fin 2006, Nvidia introduisit CUDA (Compute Unified Device Architecture), des GPU capables de calculs génériques ainsi qu'un environnement complet de programmation pour ces cartes.

5.1.2 Programmation CUDA

En introduisant CUDA, Nvidia induisit une extension du C (CUDA C [70]) pour écrire des noyaux de calculs pour leurs GPU. La programmation CUDA est dite orientée processus, c'est-à-dire que le programmeur va écrire le comportement du programme au niveau processus. Les extensions du C restent assez simples et ont pour but de faire remonter des informations sur la gestion de la mémoire, l'identification du processus, et les barrières.

La logique de la programmation CUDA est orientée autour d'une hiérarchie à 3 niveaux représentés dans la figure 5.1 et où seul le processus (*thread*) est l'élément programmable. Un processus possède ses propres registres ainsi que d'une mémoire locale gérée par le compilateur. Les processus sont regroupés par *blocs*, les processus appartenant à un même bloc peuvent se synchroniser ou partager de l'information via une mémoire partagée adressable et gérée par le programmeur. Enfin les blocs forment une grille de calcul. Les communications et la synchronisation entre blocs sont impossibles.

D'autres modèles pouvant, *in fine*, produire des programmes CUDA tel que

des liaisons avec d'autres langages comme FORTRAN, ADA ou encore Haskell, ou des annotations permettant la parallélisation automatique tel `OpenACC` [41] ou `OmpSs` [27].

Outre l'extension C pour CUDA, le compilateur CUDA de Nvidia `nvcc` manipule une représentation intermédiaire public appelé `PTX` [72]. La description des noyaux en PTX suit le même modèle que celui du CUDA C, mais le PTX permet une écriture à un niveau très proche de l'assembleur. Cette représentation textuelle est accessible aux programmeurs, et peut être manipulée par `nvcc` au moment de la compilation statique ou utilisée par un JIT lors de l'exécution.

La bibliothèque CUDA intègre un JIT qui prend en entrée un programme `PTX` et fournit un noyau exécutable pour le matériel connecté (par défaut) ou un matériel spécifiquement identifié. Pour générer du code pour les GPU CUDA, `deGoal` génère dans la représentation intermédiaire PTX et donne la main à la bibliothèque CUDA pour la génération du noyau final. Le temps de génération du code binaire est non négligeable. Malheureusement le format binaire n'est pas publique, ce qui nous empêche de créer une partie arrière pour `deGoal` produisant directement ce code. Des projets de rétro-ingénieries sont en développement [1] mais ils ne sont pas encore complètement matures, rendant leurs utilisations difficiles.

Exécution d'un programme CUDA : Lors de l'exécution d'un noyau CUDA, les blocs de la grille de calcul sont distribués aux différents SM pour leurs exécutions. S'il y a plus de blocs de calcul que de SM, ils sont mis en attente jusqu'à ce qu'un SM soit disponible.

Sur Fermi, un SM peut traiter 2 blocs en même temps à condition que la somme des ressources consommées par ces 2 blocs ne dépasse pas les capacités du SM. Le fait de permettre à 2 blocs de résider dans un même SM permet d'augmenter le nombre de warps potentiellement disponible pour exécution et donc de masquer plus facilement la latence des instructions. Pour pouvoir augmenter l'ILP nous allons généralement augmenter la charge de travail d'un processus pour offrir plus d'instructions pour la chaîne de traitement (*pipeline*). Cela se traduit souvent par une augmentation de la consommation des ressources comme le nombre de registres, l'utilisation de la mémoire locale ou partagée. Et rentre donc en compétition avec le nombre de blocs pouvant résider dans un SM.

Un autre point important dans l'optimisation des performances est la méthode d'accès aux données. Lorsque qu'un processus cherche à charger ou stocker une donnée dans la mémoire globale, le GPU va systématiquement travailler par paquets de 128 octets. Afin de maximiser les performances, il est important que les chargements ou stockages des éléments soient regroupés de façon contiguë au sein d'un warp. Cela permet de ne pas sur-utiliser la bande-passante de la carte en chargeant des données inusitées.

Il existe donc un compromis entre la consommation des ressources (nécessaire pour augmenter l'ILP) et le nombre de processus, compromis dépendant de l'application. Dans la suite de ce chapitre, nous étudierons le lien entre donnée et performances pour l'algorithme de la GEMM sur CUDA, un aspect ignoré dans l'optimisation des performances. La taille des données va notamment influer sur le nombre de processus ou de bloc disponible lors du calcul et peut nécessiter des ajustements pour garder un niveau de performance optimal.

```
pour m ← 0 à M faire
    pour n ← 0 à N faire
        pour k ← 0 à K faire
            C_{m,n} += αA_{m,k} × B_{k,n}
        fin
        C_{m,n} = βC_{m,n}
    fin
fin
```

Algorithme 1 : GEMM dans sa forme canonique.

```
pour m' ← 0 à M par pas de M_blk faire
    pour n' ← 0 à N par pas de N_blk faire
        pour k' ← 0 à K par pas de K_blk faire
            pour m ← 0 à M_blk faire
                pour n ← 0 à N_blk faire
                    pour k ← 0 à K_blk faire
                        C_{m+m',n+n'} += αA_{m+m',k+k'} × B_{k+k',n+n'}
                    fin
                fin
            fin
        fin
        Multiplication de la tuile résultat par β.
    fin
fin
```

Algorithme 2 : GEMM avec tuiles.

5.2 Produit matriciel

5.2.1 Algorithme standard

Le standard `BLAS` définit un ensemble de routines pour effectuer des opérations d'algèbre linéaire courante, avec notamment la multiplication d'un vecteur par un scalaire ou la multiplication matricielle. La `GEMM` est la désignation de la multiplication matricielle générale. Elle effectue l'opération $C = \alpha A \times B + \beta C$ sur des matrices quelconques [1]. L'algorithme 1 présente la forme canonique de cette opération. L'algorithme calcule élément par élément le résultat, cette forme ne permet pas de tirer parti de la localité des données, et donc sur CPU, l'utilisation du cache n'est pas optimal.

Une technique pour optimiser les performances est d'améliorer la localité dans l'utilisation des données grâce au tuilage. Plutôt que de traiter le calcul du résultat élément par élément, on va traiter le calcul par tuile ou bloc (Algorithme 2). Pour limiter les effets des nouvelles boucles internes, celles-ci pourront être déroulées, évitant la pénalité des branchements et des conteurs de boucles (occupation de registres). On maximise ainsi l'utilisation des registres et le parallélisme au niveau instruction (pour remplir au mieux le pipeline et éviter qu'il ne cale). Ce type d'optimisation est communément utilisé sur presque toutes les architectures, y compris les GPU.

1. Mais bien entendue compatible pour cette opération.

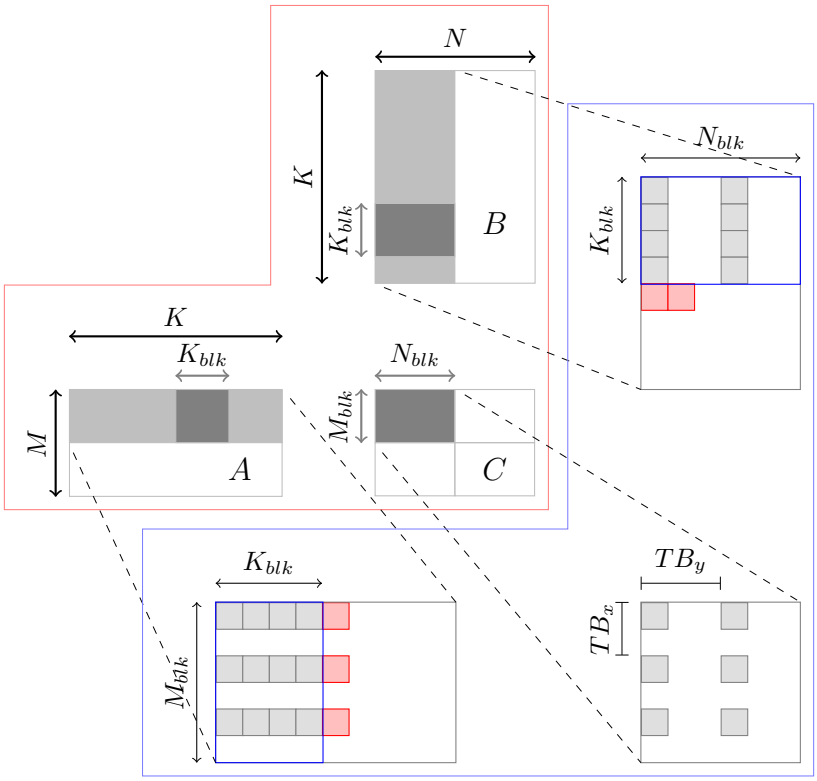


FIGURE 5.2 – Utilisation des données par le noyau d'exécution.

5.2.2 GEMM sur GPU

Pour cette expérimentation, nous nous sommes appuyés sur les résultats de `MAGMA` dans ce domaine. Notamment ceux de *Nath et al.* [64] et *Kurzak et al.* [49], pour respectivement la conception de l'algorithme et une méthode d'élagage pour le tuning des performances. Cette partie présente dans les grandes lignes l'algorithme de *Nath et al.* et *Kurzak et al.*, la section suivante s'intéressera à une limitation de l'approche de *Kurzak et al.*.

La figure 5.2 présente de façon schématique le principe des blocs et les mouvements mémoires effectués par l'algorithme 3 pour la multiplication $C = AB$ avec A de taille $M \times K$ et B de taille $K \times N$. Le schéma de l'algorithme le plus efficace sur GPU est similaire à ce qui est fait pour sur CPU (Algorithme 2). On retrouve cette même logique de bloc afin de maximiser un traitement local des données. Cependant les contraintes ne sont plus les mêmes que sur CPU où le fait de créer des blocs permettait de remplir les différents niveaux de cache. Sur les GPU, nous avons deux niveaux de caches partagés par les processus d'un bloc pour la L1 et commun à tous les blocs pour la L2, mais surtout une mémoire partagée et programmable.

Le cadre rouge (le macro-cadre supérieur) représente le découpage de la matrice au niveau du GPU. La matrice résultat C est découpée en blocs de $M_{blk} \times N_{blk}$, chaque bloc de la matrice C sera assigné à un bloc de calcul CUDA. Un bloc CUDA va calculer son résultat en itérant sur les données d'une bande

Entrée Stencil : $M_{blk}, K_{blk}, N_{blk}, TB_x, TB_y$
Entrée Stencil : $TB_{Ax}, TB_{Ay}, TB_{Bx}, TB_{By}$
Entrée Fonction : A, B, C, α, β, M, N, K
$C_{accumulateur} \leftarrow 0$
Charger une tuile de A et B de la mémoire globale vers la partagée
pour $k \leftarrow 0$ **à** $K - K_{blk}$ **par pas de** K_{blk} **faire**
  Charger une tuile de A et B de la mémoire globale dans des registres locaux
  pour $k \leftarrow 0$ **à** K_{blk} **faire**
    Charger r_x éléments de A de la mémoire partagée en registres
    Charger r_y éléments de B de la mémoire partagée en registres
    multiplier et accumuler dans $C_{accumulateur}$
  fin
  Sauvegarder les registres locaux dans la mémoire partagée
fin
pour $k \leftarrow 0$ **à** K_{blk} **faire**
  Charger r_x éléments de A de la mémoire partagée en registres
  Charger r_y éléments de B de la mémoire partagée en registres
  multiplier et accumuler dans $C_{accumulateur}$
fin
pour tous les $c \in C_{accumulateur}$ **faire**
  si *sûr* **alors**
    Sauvegarder $\alpha \times c + \beta \times C$ en résultat
  fin
fin

Algorithme 3 : GEMM stencil

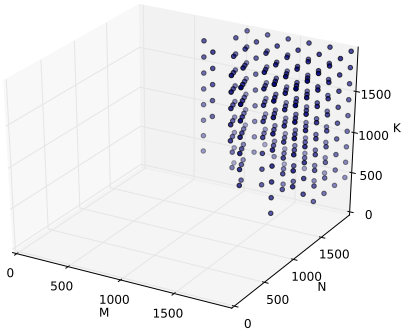
$M_{blk} \times K$ de A et d'une bande $K \times N_{blk}$ de B par pas de K_{blk}. Les sous-blocs $M_{blk} \times K_{blk}$ et $N_{blk} \times K_{blk}$ sont chargés de la mémoire globale vers la mémoire partagée et sont ensuite consommés par les différents processus du bloc.

Le cadre bleu (le macro-cadre inférieur) présente les mouvements des données effectués par un processus. TB_x et TB_y correspondent à la taille du bloc CUDA auquel est assigné une tuile de C. Les carrés gris de la tuile de C représentent les éléments calculés par un processus du bloc CUDA. Les carrés gris de A et B représentent les données de la mémoire partagée utilisées par le processus lors d'une itération. Les carrés rouges représentent les données chargées de la mémoire globale vers la mémoire partagée. Comme nous pouvons observer pour les données chargées par la matrice B, un processus ne va nécessairement charger les données qu'il va utiliser pour les calculs arithmétiques. Ces chargements redessinés permettent de limiter les pénalités.

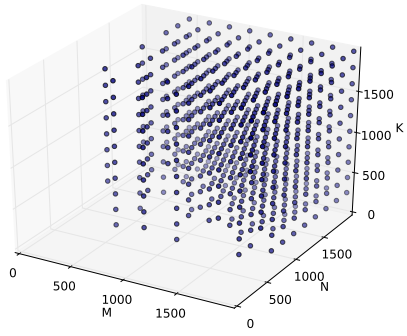
Ces différents paramètres vont affecter les performances de l'implémentation et ce pour diverses raisons :

— Nombre insuffisant de processus.
— Schéma de chargement induisant une trop forte pénalité.
— Une consommation trop forte des ressources (registres, mémoire partagée), interdisant à 2 blocs que résider dans un SM.
— Une ILP trop faible.

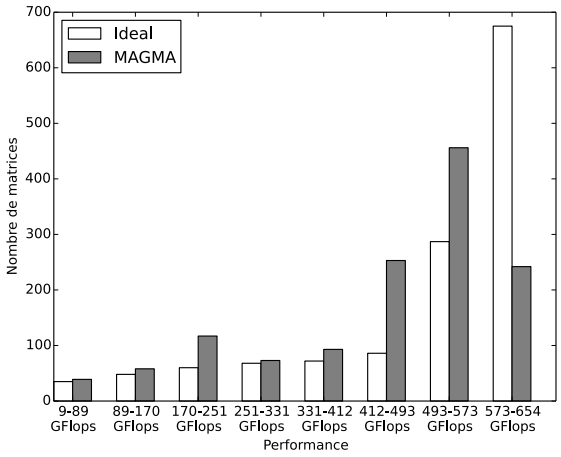
Kurzak et al. ont développé une technique de présélection efficace de configurations possibles pour l'algorithme afin de pouvoir éliminer les candidats qui



(a) `MAGMA`



(b) Cas idéal



(c) Répartition des performances de `MAGMA` et dans le cas idéal.

FIGURE 5.3 – Illustration du gain potentiel de performances pour la `SGEMM`.

produiront de façon sûre des implémentations inefficaces. Le principe est de choisir des noyaux qui ont une charge de travail minimale tous en s'assurant qu'au moins 2 blocs peuvent résider en même temps dans un SM. À la fin du processus de présélection, les candidats sélectionnés sont réduits à un nombre suffisamment bas pour permettre une évaluation systématique.

5.2.3 Problème d'adaptation aux données

Pour être efficaces, les GPU Nvidia ont besoin d'un grand nombre de processus et de parallélismes au niveau instruction pour masquer la latence d'exécution des instructions. Augmenter le parallélisme au niveau instruction implique l'augmentation de la charge de travail à effectuer et la consommation des ressources (registres et mémoire partagée) des processus. Cette augmentation implique la réduction du nombre de warps résidant dans un SM, et donc du nombre de processus disponible, jusqu'à l'impossibilité pour un SM d'avoir 2 blocs résidant en même temps, réduisant encore le nombre warps disponible.

Selon la taille des matrices et leurs formes, le nombre de processus peut changer radicalement en fonction des configurations choisies. Pour évaluer la

MAGMA								
Tailles Matrices			Configuration				Exécution	
M	N	K	M_{blk}	N_{blk}	TB_x	TB_y	Processus	GFlops
640	1 408	448	96	96	16	16	26 880	426
1 408	640	448	96	96	16	16	26 880	484
1 024	1 984	1 600	96	96	16	16	59 136	566
1 984	1 024	1 600	96	96	16	16	59 136	590
Idéal								
Tailles Matrices			Configuration				Exécution	
M	N	K	M_{blk}	N_{blk}	TB_x	TB_y	Processus	GFlops
640	1 408	448	80	64	16	8	22 528	631
1 408	640	448	96	48	16	8	26 880	549
1 024	1 984	1 600	80	64	16	8	51 584	625
1 984	1 024	1 600	80	64	16	8	51 200	622

TABLE 5.1 – Comparaison du nombre de processus et la performance obtenue pour MAGMA et le cas idéal.

perte de performance induite par l'utilisation d'une configuration unique, nous avons évalué les performances de MAGMA sur un ensemble de matrices $M \times K \times N$ de tailles diverses. Avec ces même matrices, nous avons effectué une recherche systématique de la meilleure configuration existante afin de maximiser les performances.

Les figures 5.3a, 5.3b et 5.3c présentent les résultats de cette évaluation. La figure 5.3c classe les matrices par performances obtenues (en GFlops) en 8 intervalles de 80 GFlops environ. Les barres représentent le nombre de matrices pour lesquels nous obtenons une performance située dans l'intervalle considéré. Les deux cubes (figures 5.3a et 5.3b) présentent les résultats sous la forme d'un cube dont les 3 dimensions représentent les 3 dimensions des matrices ($M \times K \times N$). Chaque point représente une matrice, placé dans l'espace en fonction de ces dimensions. La figure 5.3a présente les matrices avec lesquelles MAGMA atteint au moins 573 GFlops pour la SGEMM. La figure 5.3b présente les matrices avec lesquelles nous pouvons atteindre au moins 573 GFlops dans le cas idéal (recherche systématique de la meilleure configuration). Elle est visuellement parlante sur le lien qui existe entre taille des données et performances d'une configuration.

On constate que pour MAGMA l'essentiel des matrices est réparti dans les trois dernières catégories, avec une nette majorité dans l'avant dernière catégorie, c'est-à-dire avec des performances allant de 493 GFlops à 573 GFlops. Dans le cas idéal, l'essentiel des matrices est réparti dans les 2 dernières catégories avec une nette majorité dans la dernière catégorie, c'est-à-dire avec des performances allant de 573 GFlops à 654 GFlops.

Le tableau 5.1 illustre le besoin d'un compromis entre nombre de processus et charge de travail par processus. Ce tableau présente les performances de 4 matrices tirées des figures 5.3 avec :

— La taille des matrices : M, N et K.
— Le tuilage appliqué : M_{blk} et N_{blk} pour le découpage de la matrice résultat et TB_x et TB_y pour la taille des blocs composant la grille de calcul.

— Le nombre de processus créés lors du calcul.
— La performance finale obtenue en GFlops.

Les configurations pour les 4 matrices sont toutes distinctes [2] dans le cas idéal et donnent lieu à un nombre de processus différent comparé à `MAGMA`. Pour les 2 premières matrices, le nombre de processus est soit inférieur soit égal au cas idéal par rapport à `MAGMA`. Au contraire, pour les 2 derniers cas, le nombre de processus est inférieur à ceux créés par `MAGMA`. Mais dans tous les cas, les performances obtenues sont supérieures.

Par conséquent, afin de pouvoir maximiser les performances de l'application, il devient indispensable de s'adapter aux données, notamment aux formes des matrices. Cela pose donc naturellement 2 questions :

— Quelles sont les configurations que nous devons utiliser pour avoir les meilleures performances ?
— Comment choisir une configuration en fonction de la taille des matrices ?

Nous nous attacherons à répondre à ces deux questions dans les sections suivantes.

5.3 Solution proposée

La problématique centrale est donc de trouver un moyen de trouver la meilleure configuration en fonction des dimensions des matrices en entrées. De cette problématique, il en découle plusieurs sous-problèmes :

— L'espace des dimensions des matrices est trop large pour pouvoir espérer la meilleure configuration pour chacune d'entre elles, car cela prendrait un temps déraisonnable. Cet espace doit donc être exploré de façon partielle.
— Avec ces résultats partiels, il nous faut pouvoir inférer la configuration à appliquer en cas d'utilisation d'une matrice de dimension inconnue, c'est-à-dire.
— Enfin l'utilisation de N configurations provoque inévitablement la création de N instances différentes, posant un risque d'explosion de la taille de la bibliothèque si l'on utilise uniquement de la spécialisation statique.

Afin de pouvoir répondre au mieux à cette problématique d'adéquation entre configuration d'algorithme et données manipulées, nous avons développé une bibliothèque capable de s'adapter aux données reçues. La figure 5.4 présente les étapes de la construction de cette bibliothèque adaptative que nous avons développée. Sa construction est divisée en 3 temps distincts :

— Temps de développement : phase de conception et d'écriture de la *compilette*. Lors de cette phase nous allons créer le générateur qui sera capable de générer une implémentation de la GEMM à partir d'une configuration.
— Temps d'installation/déploiement : phase d'évaluation des performances et création de la couche d'adaptation pour le temps d'exécution. Lors de cette phase nous allons explorer une partie l'espace des matrices et des configurations. Les performances d'un nombre limité de matrices de tailles diverses seront évaluées avec toutes les configurations. À partir de ces résultats, nous utiliserons l'algorithme d'apprentissage C4.5 pour construire une fonction capable d'inférer la configuration à utiliser en fonction des dimensions des matrices.

2. Pour les deux derniers cas, seul les schémas d'accès à la mémoire diffèrent, mais ne sont pas inscrits par soucis de lisibilité.

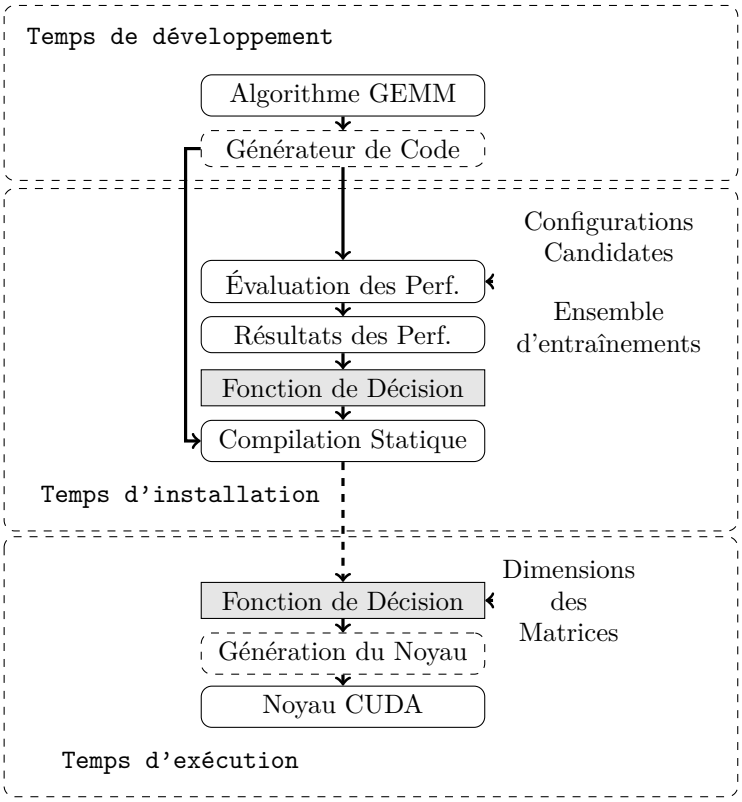


FIGURE 5.4 – Étapes de construction de la bibliothèque adaptative.

— Temps d'exécution : à ce moment les matrices sont connues et la bonne variante peut être sélectionnée afin de maximiser les performances.

5.3.1 Temps de développement

Temps classique de conception et développement de l'algorithme. Dans cette expérimentation, nous nous sommes concentrés sur la GEMM dont l'algorithme est décrit en Section 5.2.

À partir de cet algorithme, nous avons écrit la *compilette* correspondante. Le générateur de code permet de générer n'importe quelle implémentation de la `GEMM` pour toutes configurations valides. La *compilette* propage quelques constantes (comme la valeur de la taille des blocs par exemple), et déroule les boucles internes en fonction de la configuration donnée. Peu d'optimisations au niveau du code sont possibles en dehors de ces 2 précédentes, mais l'aspect intéressant de l'approche est l'impact bénéfique sur la taille de la bibliothèque finale.

5.3.2 Temps d'installation

À ce moment nous disposons encore de temps pour effectuer des optimisations et des évaluations : en l'occurrence, l'évaluation des variantes générées par

la *compilette* pour un ensemble de données. Avec ces résultats, nous pouvons construire la fonction de décision qui sera utilisée lors de l'exécution.

La première étape est de créer un ensemble de données de tests :

— Un ensemble de configurations pour la *compilette* : taille des tuiles, des blocs CUDA etc. Nous utilisons la méthode de *Kurzak et al.* pour construire cet ensemble.

— Un ensemble de matrices, qui serviront à mesurer une corrélation entre la forme des matrices et les configurations. Il est important que l'ensemble des matrices soit représentatif de l'espace utilisé par l'application afin d'être le plus efficace possible. Dans l'expérimentation, nous avons considéré l'espace borné $[64, 2000]$ pour chaque dimension dans lequel 1331 points ont été choisis.

À partir cet ensemble de test, nous pouvons récupérer l'ensemble des résultats des évaluations.

Avec les résultats des tests de performances, nous pouvons construire la fonction de décision. Cette fonction est construite grâce à l'algorithme C4.5 [78] (détaillé en section 5.5.3) qui permet la génération d'ensembles de règles facilement transformables en une fonction de complexité en $O(1)$.

Une fois cette fonction de décision créée, la bibliothèque peut être compilée statiquement.

5.3.3 Temps d'exécution

Ce temps combine les résultats des deux précédents temps. Une fois la forme des matrices connue, la fonction de décision choisit la configuration à utiliser pour obtenir les meilleures performances. La *compilette* génère ensuite le noyau CUDA qui peut ensuite être exécuté.

Le temps de génération n'est pas négligeable à cause de la contrainte sur `deGoal` d'utiliser le PTX et de devoir générer le noyau final par le JIT CUDA. Cependant pour masquer au mieux le coût de la génération, il est possible :

— de faire un cache des noyaux générés et de les réutiliser en cas de besoin.

— d'utiliser un noyau générique pendant la génération du noyau optimal s'il n'est pas disponible.

5.4 Auto-tuning

5.4.1 Présentation

La phase de tuning lors de l'optimisation d'un algorithme est la recherche des paramètres optimaux afin d'obtenir les meilleures performances d'une implémentation sur une architecture donnée. Ces paramètres sont dépendants des caractéristiques de la machine : taille des caches, vitesses d'accès à la mémoire, latence des instructions etc. Selon l'algorithme, ces paramètres peuvent être peu ou très sensibles aux données. Par exemple, l'algorithme de la FFT ("*Fast Fourier Transform*", transformée de Fourier rapide) ou la GEMM sur des matrices creusent sont réputées pour être dépendants aux données et donc les paramètres sont donc choisis en fonction des entrées.

L'auto-tuning est l'automatisation de la recherche de ces paramètres optimaux, l'avantage est d'obtenir une portabilité ou d'assurer une adaptation à

différentes architectures et à leurs évolutions sans intervention humaine.

5.4.2 Approche statique

Cette approche effectue l'évaluation des variantes et sélectionne la meilleure implémentation, lors de l'exécution aucune autre étape d'optimisation n'est effectuée. Cela présuppose donc une indépendance de l'algorithme par rapport aux données de l'utilisateur. Les implémentations hautes performances de la `GEMM` utilisent généralement cette approche statique, c'est le cas pour `ATLAS` [90] (pour CPU) et `MAGMA` [49] (pour GPU). `ATLAS` est optimisé lors de l'installation pour tenir compte du matériel et présuppose que le matériel ne changera pas.

5.4.3 Approche dynamique

Pour les algorithmes dépendants de la taille des entrées pour leurs performances, le choix de l'implémentation doit être faite à l'exécution. C'est le cas par exemple pour l'algorithme de la FFT, où plusieurs implémentations peuvent être utilisées en fonction des données. La bibliothèque FFTW [31] effectue une phase d'évaluation au moment du déploiement afin de connaître le fonctionnement des différentes variantes (appelées "codelet") et sélectionner les plus performantes.

Lors de l'exécution, la bibliothèque forme un *plan* en fonction de la taille des données et à partir des résultats obtenus au moment du déploiement. Le plan sélectionné est le plan rapide obtenu après l'évaluation de plusieurs plans candidats. Cette évaluation prend du temps, mais permet d'obtenir une version performante. Cette approche est efficace si la forme des données ne change pas.

Dans le cas qui nous intéresse, cette technique est difficilement employable car, d'une part, il n'existe pas d'approche par plan comme pour la FFT, et d'autre part, l'évaluation des variantes à l'exécution prendrait un temps non négligeable et donc serait difficile à amortir.

5.4.4 Approche avec apprentissage machine

Le principe est proche de l'approche précédente, mais cherche à éviter la phase d'évaluation et de recherche présente dans la précédente en "apprenant" des résultats de l'évaluation. Dans le cas du traitement du signal, `Spiral` [76, 24] est un exemple d'utilisation de cette stratégie. Diverses implémentations sont évaluées lors du déploiement, et ces résultats passent par un algorithme d'apprentissage qui génère une fonction de décision, évitant de cette façon la phase de recherche et le surcoût de l'approche dynamique, et peut donc être utilisé plus facilement si la taille des données change.

Cette solution nous permet de supporter le coût de génération du code plus facilement lors de l'exécution. C'est donc celle qui a été retenue, néanmoins il nous est maintenant nécessaire d'avoir une phase d'apprentissage adaptée pour être utilisable à l'exécution.

5.5 Apprentissage machine

5.5.1 Définition

L'apprentissage machine désigne un ensemble de techniques permettant à un programme de prendre une décision en fonction d'un ensemble de données qui lui aura été fourni. Les décisions sont prises en ségrégant les données grâce à des discriminateurs (ou caractéristiques) associés à ces données. Ces techniques couvrent plusieurs domaines, de la classification (c'est-à-dire affectation d'une instance à une "classe") à la réduction de dimension. Dans notre cas, ce qui nous intéresse c'est de pouvoir, à partir des dimensions d'une matrice, inférer la meilleure configuration pour notre algorithme (c'est-à-dire sa classe).

Cette section présente quelques approches d'apprentissage machine existantes et présentera la technique qui sera utilisée par la suite pour permettre l'adaptation aux données.

5.5.2 Différentes approches

Il existe différentes approches pour l'apprentissage machine :

k-NN : k plus proches voisins. C'est l'un des algorithmes les plus simples qui fonctionne sur un espace métrique. Pour inférer la classe d'une instance, l'algorithme choisit les k plus proches voisins de cette instance, la classe qui lui est affectée est celle la plus représentée parmi ces k voisins. Le gros inconvénient de cette approche est le temps de calcul requis par la fonction de décision pour inférer une classe et la taille qui impose d'embarquer l'ensemble des données d'apprentissages.

SVM : machine à vecteurs de support (Support Vector Machine). C'est une approche assez répandue, le principe est de trouver des hyperplans dans l'espace permettant de séparer 2 classes en maximisant une marge entre les représentants des 2 classes. Pour les cas plus complexes, l'algorithme peut utiliser des noyaux de calculs permettant de simplifier la séparation entre les représentants des classes. De même, il est peut-être étendu à une discrimination multi-classes. Bien que très efficace, il reste assez difficile à utiliser sans intervention humaine et nécessite de nombreux calculs pour inférer la classe d'une inconnue.

GMM : mixture de gaussienne (Gaussian Mixture Model). C'est une méthode paramétrique qui modélise la fonction de décision par une somme de distribution gaussienne. Chaque classe est donc représentée par une somme de fonctions gaussiennes, et une instance est affectée à la classe pour la laquelle il obtient le score le plus haut. Un des inconvénients de cette technique est d'avoir à déterminer à l'avance le nombre de gaussiennes requis pour modéliser une classe, ce qui rend son paramétrage initial délicat.

Arbre de décision : C'est une méthode non paramétrique, le choix dans l'assignation d'un élément à une classe est effectué en parcourant un arbre construit à partir de l'ensemble d'apprentissage. C'est cette approche qui est utilisée dans la solution, plus précisément nous utilisons l'algorithme C4.5 dont le principe de fonctionnement est décrit plus en détails dans la sous-section suivante.

Prévision	Température	Humidité	Venteux	Décision
soleil	85	85	non	pas de golf
soleil	80	90	oui	pas de golf
couvert	83	78	non	golf
pluie	70	96	non	golf
pluie	68	80	non	golf
pluie	65	70	oui	pas de golf
couvert	64	65	oui	golf
soleil	72	95	non	pas de golf
soleil	69	70	non	golf
pluie	75	80	non	golf
soleil	75	70	oui	golf
couvert	72	90	oui	golf
couvert	81	75	non	golf
pluie	71	80	oui	pas de golf

TABLE 5.2 – l'ensemble de données "weather" pour l'apprentissage machine.

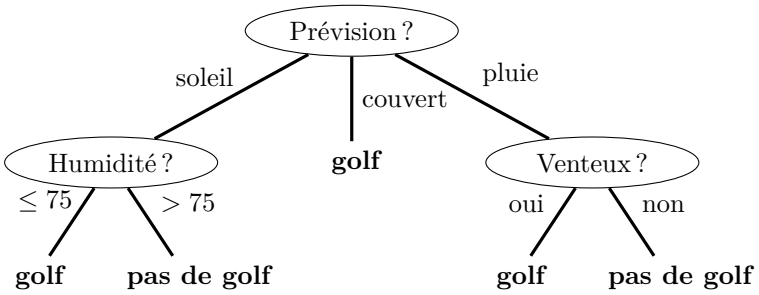


FIGURE 5.5 – Arbre de décision généré pour C4.5.

Pour la construction de la bibliothèque, nous avons donc choisi d'utiliser les arbres de décision car ils offrent l'avantage de ne nécessiter que peu de paramétrages (ce qui facilite une construction automatisée) et l'arbre produit peut être converti en un ensemble de règles. Le résultat peut donc être transformé en une fonction plus petite en termes d'empreinte mémoire et qui s'exécute en temps constant.

Comme décrit dans la section précédente, le tuning de l'algorithme se faisant hors-ligne, les résultats du tuning ont besoin d'être traitées pour être capables d'inférer correctement la configuration à utiliser à partir des tailles des matrices. Cette section présente donc un petit état de l'art des approches d'apprentissage machine qui a été fait pour les besoins de la construction de cette bibliothèque.

5.5.3 Arbre de décision C4.5

Dans notre solution, nous avons choisi l'algorithme C4.5 [78], un algorithme largement répandu. Il offre l'avantage de pouvoir produire l'ensemble des règles permettant l'inférence de classe, et surtout le support des attribues continues.

Comme le nom l'indique, le choix dans l'assignation d'un élément à une classe est effectué en parcourant un arbre construit à partir de l'ensemble d'ap-

prentissage. L'intérêt est double : *a)* ils simplifient la vue du problème (multi-dimensionnel) en lui donnant une représentation facilement visualisable ; *b)* ils permettent d'avoir un mécanisme d'inférence de nouvelle classe sous forme d'ensemble de règles.

Pour former l'arbre, ces algorithmes partitionnent l'ensemble des données en séparant de façon récursive l'espace. Une dimension est choisie de façon à séparer, de façon judicieuse, l'ensemble en plusieurs sous-ensembles. L'opération est ensuite répétée sur les nouveaux sous-ensembles, l'algorithme stoppe quand il ne reste plus qu'une classe dans l'ensemble. A l'origine, ces types d'algorithmes sont plutôt destinés à être utilisés sur des classes à discriminant symbolique (ex. "pluie" ou "soleil"). Leurs extensions aux discriminants continus est possible en séparant l'espace par intervalles.

La figure 5.5 (tiré de l'article [24]) présente le résultat de la construction de l'arbre de décision en utilisant C4.5 à partir de l'ensemble de données du tableau 5.2.

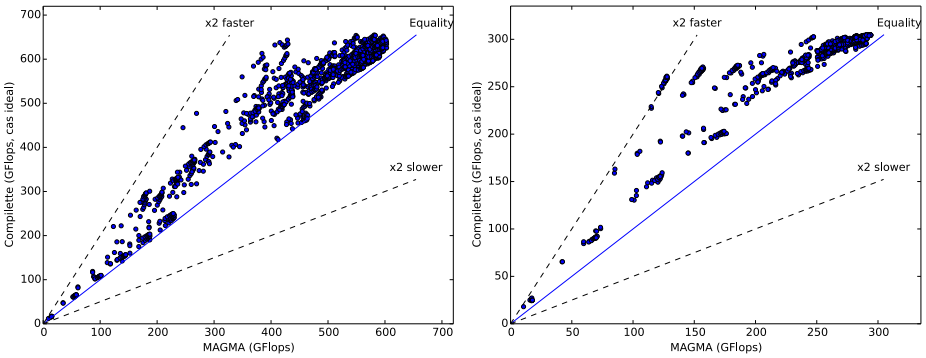
L'algorithme C4.5 ne permet que de discriminer l'espace via des plans. Dans la pratique cela peut être limitant quand les dimensions sont peu nombreuses (dans l'exemple du golf, tableau 5.2, les dimensions sont "Prévision", "Température", "Humidité" et "Venteux"). Dans notre cas nous avons à disposition 3 dimensions qui sont celles des matrices : M, K et N. Or il peut être intéressant de choisir la configuration à utiliser en fonction du nombre d'éléments des matrices ou du nombre d'opérations à effectuer. Ces informations sont facilement déterminables pour nous, mais elles vont rester cachées à C4.5. Ainsi pour faciliter l'apprentissage nous avons augmenté les dimensions de l'espace d'apprentissage en ajoutant ces types d'informations. L'intérêt et l'impact de cette technique sera discutée dans la partie présentant les résultats.

5.6 Résultats & discussion

5.6.1 Plate-forme et procédure d'évaluation

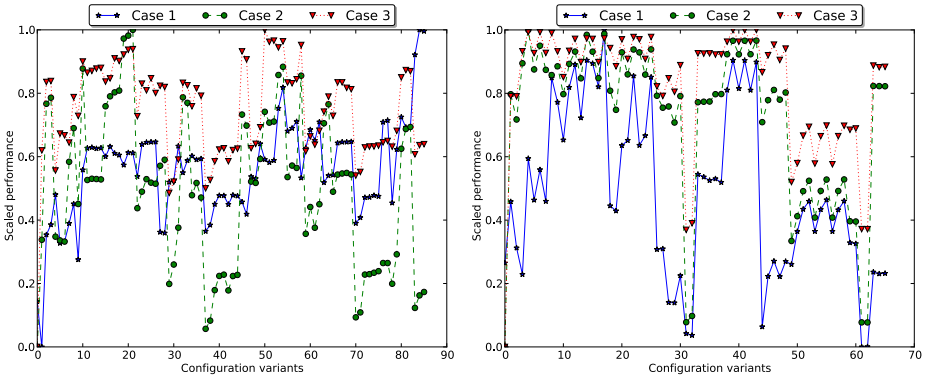
Les résultats présentés dans cette section ont été produit sur des serveurs *bullx* de `BULL`. Ces serveurs sont basés sur des Intel Xeon E5640 tournant à 2,67 GHz et d'une carte GPU Nvidia M2050. La carte Nvidia M2050 est basée sur une architecture Fermi (décrite dans la section 5.1) avec une performance crête à 1,03 TFlops pour les opérations à virgule flottante à simple précision et 515 GFlops pour les doubles précisions. Le système d'exploitation est un Linux 2.6.18, la boîte à outils CUDA utilisée est la version 4.2 et les drivers Nvidia 304.54.

Chaque cas de tests est exécuté 3 fois, ce qui semble être suffisant pour éliminer le bruit [49]. Les performances obtenues par la solution sont comparées avec `MAGMA` version 1.3. L'implémentation de la `GEMM` de Nvidia, présente dans leurs boite à outils [71], est en réalité plus efficace [49, 83]. Mais leur implémentation n'est pas divulguée et est implémentée directement en `SASS` (non divulgué lui aussi), une comparaison avec cette bibliothèque n'aurait donc pas été équitable et ne permet donc pas de mesurer le gain en performance.



(a) SGEMM (b) DGEMM

FIGURE 5.6 – Performance de la *compilette* avec adaptation à l'exécution dans le cas idéal par rapport aux performances de `MAGMA`.



(a) SGEMM (b) DGEMM

FIGURE 5.7 – Balayage des configurations pour 3 cas.

5.6.2 Exploration de l'espace des données

Cette partie présente les résultats obtenus pour la première partie du temps d'installation (deuxième bloc de la figure 5.4).

Deux espaces sont explorés, l'espace de configuration, dont la procédure de sélection des candidats est décrite par *Kurzak et al.* [49], et l'espace des matrices. Au vu des tailles des matrices manipulées par les GPU, l'adaptation de l'algorithme aux valeurs des coefficients des matrices est difficile au vu du temps requit pour générer le code correspondant. L'espace des matrices se limite donc uniquement aux tailles (M, K et N) et les matrices de tests ont des coefficients aléatoirement générés.

Pour l'expérimentation, nous avons considéré l'espace borné $[64, 2000]$ pour chaque dimension. Dans cet espace nous avons sélectionné 1331 points de façon pseudo-aléatoire : l'espace est discrétisé en sous-cube de 192 de côté et un point est pris aléatoirement dans chaque sous-cube. Prendre un point au hasard permet d'éviter de prendre des points particulier et donc d'introduire un biais, ce point pourrait être amélioré et est discuté dans la section 5.7. Les diverses

	Cas 1	Cas 2	Cas 3
M	256	256	1984
K	256	1216	1984
N	64	1600	1984
Performance max. (GFlops)	60	534	642
Performance `MAGMA` (GFlops)	52	474	602

TABLE 5.3 – 3 matrices testes de la figure 5.7 et leurs performances crête.

expérimentations nous ont montré que cela correspondait à un compromis entre le temps d'exploration et la précision. Cet espace est la zone où les performances sont le plus sensible aux données, au-delà, les GPU arrivent à saturation et le gain en performance est négligeable avec l'adaptation. La borne basse est très dépendante de l'architecture de la cible car elle correspond à la limite où les temps cumulés de transfert vers le GPU, le calcul sur GPU et le rapatriement du résultat sont plus faible que le calcul sur CPU. Elle est donc discutable dans le sens où elle doit être adaptée aux besoins et à l'architecture sur laquelle l'algorithme va tourner. Néanmoins, cette limite correspond aux performances obtenues sur nos machines de tests.

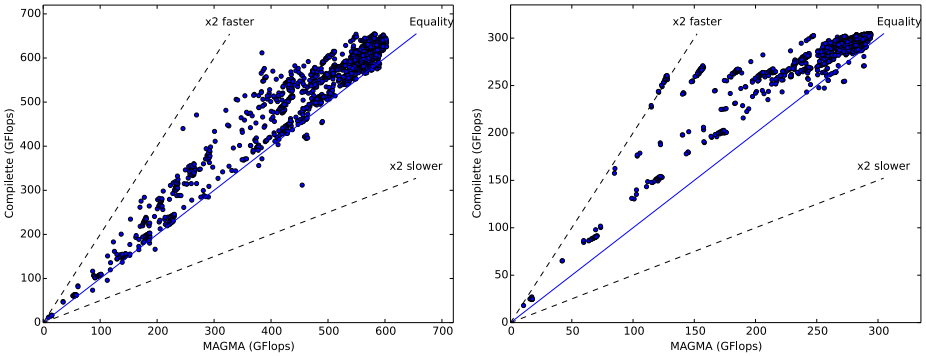
Les figures 5.6a et 5.6b présentent le gain en performance obtenu par la compilette vis-à-vis de `MAGMA` pour, respectivement, les flottants simple et double précision. Pour les deux figures, l'axe des abscisses représente les performances de `MAGMA`, l'axe des ordonnées les performances de la compilette. Chaque point représente la performance obtenue par une matrice avec `MAGMA` et la compilette. Plus le point est éloigné de la ligne pleine, plus le gain en performance obtenue avec la compilette est élevé. Les gains obtenus vont de 0% à 81% (16% en moyenne) pour la `SGEMM` et de 2% à 105% (14% en moyenne) pour la `DGEMM`. Il est intéressant de constater que le lien entre forme de matrice et configuration n'est pas évident. La figure 5.7 présente les performances obtenues lors du balayage des configurations pour la `SGEMM` et la `DGEMM` et ceux pour 3 matrices distinctes (présentées dans le tableau 5.3). Sur l'axe des abscisses nous avons les différentes configurations testées (l'ordre n'a pas d'importance) et sur l'axe ordonnées nous avons la performance mise à l'échelle sur l'intervalle $[1, 0]$ par rapport à la meilleure performance obtenue pour chaque cas. On constate qu'il existe des configurations un peu passe partout, c'est-à-dire qu'elles donnent des performances à peu près constantes quelle que soit la forme de la matrice, et d'autre donnant des performances optimales pour certaines matrices mais de très mauvais résultats pour d'autres. Cependant il semble difficile d'obtenir une règle pour choisir simplement quelle configuration utiliser en fonction des dimensions des matrices.

5.6.3 Génération de la fonction de décision

Les résultats du balayage sont difficilement exploitables de façon brute, à cause de leurs tailles (quantité de données) et d'un manque d'éléments caractérisants. Les seules informations connues lors de la sélection de la configuration sont uniquement les informations de tailles : $M \times K \times N$. Certaines informations discriminantes, comme le nombre d'éléments d'une matrice, nombre d'éléments en entrée ou en sortie, sont cachées à l'algorithme d'apprentissage et donc non

Algorithme	Caractéristiques dérivées	Nombre de règles	Configurations
`SGEMM`	Non	152	22
`SGEMM`	Oui	76	
`DGEMM`	Non	121	8
`DGEMM`	Oui	84	

TABLE 5.4 – Nombre de règles générées pour construire la fonction de décision, avec ou sans insertion de caractéristiques dérivées.



(a) SGEMM (b) DGEMM

FIGURE 5.8 – Performance de la bibliothèque avec adaptation à l'exécution par rapport aux performances de `MAGMA`.

exploitables. Pour faciliter l'apprentissage, nous avons augmenté l'espace des caractéristiques afin de laisser plus de liberté à l'algorithme pour discriminer l'espace. L'algorithme C4.5 ne permet que de discriminer l'espace via des plans, l'augmentation du nombre de caractéristiques permet d'augmenter le nombre de dimensions de l'espace, facilitant la création de ces plans. Ainsi, plutôt que de faire le choix sur 3 dimensions, C4.5 le fait sur 11 dimensions qui inclut le nombre d'éléments dans chaque matrice, nombre d'opérations effectuées *etc.* Ces dimensions sont intriquées, mais elles facilitent tout de même l'apprentissage.

Le tableau 5.4 présente le nombre de règles générées pour la fonction de décision avec ou sans génération des caractéristiques dérivées. Le nombre de règles s'en retrouve grandement réduit lorsque l'on ajoute les caractéristiques dérivées. L'impact sur les performances de la bibliothèque est présenté dans le tableau 5.5 et sera discuté dans la section suivante.

Une fois ces caractéristiques créées, nous avons utilisé l'algorithme de construction d'arbre de décision C4.5. De part sa nature, il permet de ne sélectionner que les caractéristiques qui lui sont utiles. L'utilisation de C4.5 produit une fonction de composée de 76 règles (pour 22 configurations distinctes) pour la `SGEMM` et de 84 règles (pour 8 configurations distinctes) pour la `DGEMM`.

5.6.4 Évaluation des performances de la bibliothèque

Amélioration des temps d'exécutions : La figure 5.8 présente les performances de la bibliothèque sur le jeu de matrices test. La figure se lie de la

Algorithme	Gain			
	Max. (%)	Min. (%)	Moyenne (%)	Médian (%)
`SGEMM` idéal	81	0	16	11
`SGEMM` & adaptation	79	-18	11	8
`DGEMM` idéal	105	2	14	7
`DGEMM` & adaptation	105	-7	13	7

TABLE 5.5 – Résumé des statistiques d'évaluation des performances de la bibliothèque. Le gain est calculé par rapport aux performances de `MAGMA`.

même façon que dans la figure 5.6, les points étant en dessous de la ligne pleine représente des cas de matrices pour lesquelles nous avons une perte de performance. Le tableau 5.5 reprend les gains extrémaux, moyens et médians par rapport à `MAGMA`. Les erreurs de classifications, inhérent à ce type de technique, peuvent entraîner dans certains cas une perte de performance, -18% pour la `SGEMM` et -7% pour la `DGEMM` dans le pire cas. Cependant, les gains maximaux sont conservés, 79% pour la `SGEMM` et 105% pour la `DGEMM`, et de façon globale, les performances sont améliorées de 11% en moyenne pour la `SGEMM` et 13% pour la `DGEMM`.

L'écart entre les gains idéaux et ceux obtenues par la bibliothèque est moins important pour la `DGEMM` que pour la `SGEMM`. Cette différence s'explique par le nombre de configurations intéressantes qui est moins importante pour le cas de la `DGEMM` (8 configurations retenues) que pour la `SGEMM` (22 configurations retenues). Comme on peut le constater sur la figures 5.7a et 5.7b, l'écart de performance entre 2 configurations est faible dans un grand nombre de cas pour la `DGEMM`. De plus, plus la taille des matrices grandit plus l'écart est faible en pourcentage. Pour ce qui est de la `SGEMM`, le balayage donne quelque chose de beaucoup plus chaotique provoquant ainsi une pénalité plus importante en cas d'erreur de décision.

Empreinte mémoire : Pour utiliser les *compilettes*, le programme embarque les routines de génération de code dans l'application. Les *compilettes* peuvent ainsi réduire l'empreinte mémoire de l'application comparée à l'effet de la spécialisation statique si la taille de la partie arrière embarquée n'excède pas la somme des fonctions spécialisées statiquement.

Pour montrer l'impact de l'empreinte mémoire de `deGoal` sur la taille de l'application, nous avons comparé deux applications qui récupèrent une variante de la GEMM parmi celles sélectionnées par la fonction de décision et l'exécutent sur un jeu de donnée. La version `deGoal` embarque la *compilette* pour la `[SD]GEMM` et les configurations qui seront utilisées. L'autre application embarque toutes les variantes de la `[SD]GEMM` compilées statiquement. Ces applications sont compilées en `-O3`. Il en résulte que le binaire utilisant `deGoal` pèse 57Kb et la version statique 2,7Mb. Cela correspond à une diminution de la taille (statique) de l'application d'un facteur 40.

Autre point important, la version statique embarque les noyaux au format "FATBIN", c'est-à-dire au format binaire et uniquement compilé pour les GPU `Fermi`. `deGoal` génère du code au format PTX, et bien que son utilisation engendre un surcoût supplémentaire lors de la génération du code, il assure une portabilité entre les générations.

```
Entrée Fonction : M, K, N
si Les matrices sont trop petites pour le GPU alors
    Exécuter sur CPU
sinon
    Transférer les données sur le GPU en arrière plan
    Version ← select_gpu_version(M, K, N)
    si Version n'est pas en cache alors
        GEMM ← generate(Version)
        mettre en cache GEMM
    sinon
        GEMM ← cache(Version)
    fin
    Exécuter GEMM sur le GPU
    Rapatrier le résultat
fin
```

Algorithme 4 : Pseudo-code de la sélection de la plate-forme de calcul.

Coût de la génération de code : Le coût de la génération de code à été jusqu'à présent ignoré. L'obligation de devoir passer par le compilateur JIT CUDA afin de générer le noyau final ralentit fortement la génération. En moyenne, le temps de génération du code (*compilette* plus compilateur JIT CUDA) est de 58ms. Le compilateur JIT CUDA possède plusieurs niveaux d'optimisations, mais pour ce noyau, ils n'ont pas d'effet significatif sur le temps de génération.

Pour que la génération soit complètement amortie, il faut en moyenne 800 utilisations pour la `SGEMM` et 200 utilisations pour la `DGEMM`, la valeur médiane est de 200 utilisations pour les deux versions. Le JIT CUDA occupe 99 % du temps pris par la génération de code. Avec un accès à la représentation binaire du jeu d'instruction `Fermi`, l'on peut espérer avoir un temps de génération aux alentours de 1 ou 2 % du temps actuel. Dans ce scénario, nettement plus favorable, l'amortissement serait autour 2 à 16 utilisations.

5.6.5 Intégration dans Scilab

Scilab [81] est un outil de calcul matriciel similaire à "*matlab*". Dans le cadre de ces travaux, la solution d'adaptation a été intégré dans l'outil afin d'en améliorer les performances [59].

Les opérations doivent à la fois pouvoir s'effectuer sur CPU et GPU, la sélection de la variante à utiliser se fait donc à 2 niveaux :

— Le calcul est-il assez lourd pour le GPU ?

— Quelle configuration utiliser pour l'exécution sur GPU ?

Afin de réduire au maximum les coûts de générations, un système de cache est mis en place afin de pouvoir réutiliser les variantes déjà générées. L'algorithme 4 présente le principe d'utilisation de la bibliothèque dans le cadre de *Scilab*. La première ligne teste détermine si l'on doit faire le calcul sur CPU ou GPU. Si la matrice est top petite, on exécute le calcul sur le CPU. Sinon nous transférons en arrière plan les données des matrices, puis on choisit la variante à utiliser. Si elle n'a pas été générée, nous la générons et mettons la matrice en cache, sinon nous récupérons la version en cache. Nous effectuons ensuite le calcul sur

le GPU et nous rapatrions le résultat.

5.7 Pistes d'évolution et améliorations

Cette section présente deux pistes d'évolutions possibles, sont présentées dans la figure 5.9. La figure reprend le processus de la figure 5.4 et positionne ces deux pistes dans ce processus.

5.7.1 Évaluation des performances

Une perspective pour améliorer la solution serait d'avoir une boucle de rétro-action lors de la phase d'évaluation des performances. L'évolution est illustrée en rouge dans la figure 5.9 (boites "Éval. de la précision" et "Création de nouveaux tests"). La sélection des candidats est suffisante et donne des solutions qui sont pour la grande majorités performantes. L'exploration de l'espace pourrait être améliorée en ayant une boucle entre les matrices de tests choisis et la précision de la fonction de décision.

Cela permettrait éventuellement d'améliorer les résultats de la fonction de décision (augmentation du gain moyen, diminution de l'impact des erreurs *etc.*) ou de diminuer le temps d'exploration en retirant des points de l'espace inutilement testés dans la version présentée.

Quelques outils, comme ASK (Adaptive Sampling Kit) [25], fournissent des outils pour effectuer cette boucle. Il n'est pas garanti que ce type d'outils permette en effet l'amélioration des résultats dans cet exemple précis, mais elle pourrait bien l'être avec d'autres applications.

5.7.2 Limitation de l'impact des erreurs de classification

Une autre perspective pour améliorer la solution serait de pouvoir simplifier l'espace avant l'apprentissage (boite "simplification" dans la figure 5.9). L'idée est d'essayer de trouver le compromis entre la configuration la plus performante pour un cas particulier et les résultats des cas voisins, et espérer ainsi trouver une solution où l'erreur de classification aurait un impact limité, mais toujours avec une solution performante.

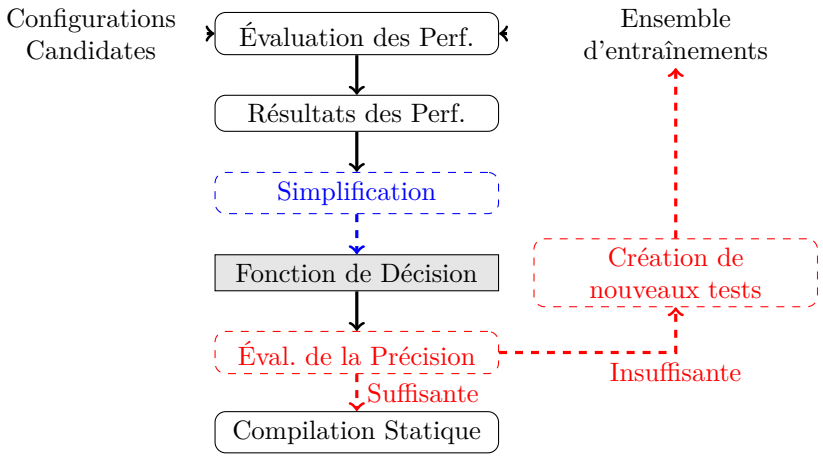


FIGURE 5.9 – Évolutions possibles lors de l'apprentissage.

5.8 Conclusion

Dans ce chapitre nous avons étudié l'impact des données sur GPU pour un algorithme particulier : la `GEMM`. Nous avons vu qu'en moyenne, il est possible d'avoir un gain de performance de 14 à 16% si l'on possède un oracle capable de déterminer quelle configuration appliquer à l'implémentation de la `DGEMM` et de la `SGEMM` respectivement.

À partir de ce constat nous avons construit une solution permettant de créer, de façon automatique, une couche d'adaptation moyennant une phase d'évaluation et d'apprentissage. La phase d'apprentissage permet d'inférer la configuration à utiliser en fonction de celles utilisées par des cas voisins évitant ainsi d'avoir à explorer tous les cas possibles, ce qui serai infaisable dans la pratique. L'utilisation de C4.5 permet de combiner un apprentissage rapide et la capacité de générer une fonction de décision sous forme d'une liste de règles, permettant un choix rapide. La construction de la bibliothèque est décomposée en *3 temps* :

— L'écriture : avec la création de la *compilette* permettant la génération d'une instance d'un noyau de la `GEMM` suivant une configuration quelconque.
— Le déploiement : ce qui permet d'évaluer les performances de différentes instances en fonction de la forme des matrices et de construire une couche adaptative en utilisant de l'apprentissage machine.
— L'exécution : à partir des tailles des matrices en entrée on génère l'instance de la `GEMM` la plus adaptée pour maximiser les performances.

Les performances de la bibliothèque ainsi construite donnent de bon résultats avec une augmentation moyenne des performances de 11% pour la `SGEMM` et de 13% pour la `DGEMM` avec des gains maximaux de 79% et 105% pour la `SGEMM` et la `DGEMM` respectivement. L'utilisation de `deGoal` nous permet aussi d'éviter une explosion de la taille du binaire. Par rapport à une spécialisation statique classique, `deGoal` permet de réduire la taille de la bibliothèque d'un facteur 40.

Chapitre 6

Étude multi-métriques de l'impact de la génération de code

Sommaire

`deGoal` est un outil de génération de code très flexible. Nous l'avons présenté dans le détail dans le chapitre 4 et utilisé dans le chapitre 5 pour contrôler la génération des variantes de la GEMM, réduisant ainsi la taille de la bibliothèque. `deGoal` a été conçu pour permettre l'écriture de générateur de code pouvant effectuer des transformations arbitraires (c'est-à-dire basées sur des opportunités propres à un algorithme ou à une architecture), mais cette flexibilité rend le

développement plus long. Dans beaucoup de cas de figure il n'est pas nécessaire d'avoir une telle flexibilité. Dans l'algorithme de la `GEMM` sur GPU (chapitre 5) peu de constantes sont à propager lors de l'exécution. Le résultat est que l'on a un grand nombre d'instructions qui ne varient pas selon les entrées, et pour celles en dehors de la boucle l'ordonnancement manuel n'a pas d'influence notable sur les performances. L'écriture à un niveau assembleur du noyau devient un frein plutôt qu'un avantage pour améliorer les performances à cause de l'allongement du temps d'écriture. Pour illustration, l'implémentation `deGoal` de la `GEMM` est composé de 93 instructions `deGoal`, dont 59 sont constantes (c'est-à-dire indépendantes des paramètres) et 22 uniquement liées à une propagation simple de constantes. L'écriture à un plus haut niveau de la fonction à générer permet un développement plus rapide des parties où la flexibilité de `deGoal` n'est pas nécessaire et permet d'obtenir un bon compromis entre temps de développement et performance. `Kahuna` permet la création automatique de spécialiseurs avec un faible coût de génération, comme `deGoal` peut les produire, créés de façon automatique à partir d'une représentation intermédiaire (RI) annotée.

La problématique étudiée dans ce chapitre est l'évaluation de l'efficacité de la génération et de la spécialisation de code à l'exécution dans un contexte embarqué. Nous avons vu dans l'état de l'art, qu'il existe un certains nombres d'approches connexes à `deGoal` et `Kahuna`. Dans l'état de l'art, ces approches sont centrées sur la vitesse d'exécution et le coût de la génération. Nous avons aussi présenté plusieurs types d'objectifs et des métriques associées (tableau 3.1, page 35). Nous étudions donc les gains et coûts de la génération et de la spécialisation de code selon plusieurs objectifs :

Vitesse d'exécution : avec comme métriques :
- Vitesse d'exécution des noyaux générés en opérations par seconde (gain) ;
- Vitesse de génération du code en cycle et en cycles par instruction générée (coût).

Utilisation de la mémoire : avec comme métriques :
- Taille des noyaux générés en nombre d'instructions (gain) ;
- Empreinte mémoire statique et dynamique en octet (coût) ;

Consommation énergétique :
- Énergie consommée par le noyau généré par rapport à un noyau de référence (gain) ;
- Énergie consommée pour la génération du code, en joule (coût).

Pour ce faire, nous présentons dans ce chapitre une évaluation croisée de 3 techniques de spécialisation (sections 6.1 à 6.3) :
- Une technique "naïve", basée sur le JIT LLVM. Elle est dite naïve car elle impose d'avoir, en plus de l'application, une machinerie importante et lourde au regard de la fonction considérée.
- En utilisant `deGoal`. Ce qui sera l'occasion de comparer d'un point de vue qualitatif les avantages et inconvénients de l'approche orientée générateur et l'approche orientée fonctionnelle.
- En utilisant `Kahuna`. Ce qui permet la comparaison du code généré avec la version précédente, car les deux sont basés sur la même partie arrière de LLVM.

La section 6.5 présentera des évolutions possibles de `Kahuna`. On s'intéressera notamment aux transformations qui seraient possibles de faire statiquement, les problématiques soulevées et des solutions envisageables.

Entrée Fonction : in, out, $longueur\_tampon$, $filtre$, $i1$, $i2$, $o1$, $o2$
pour $I = 1$ **à** $longueur\_tampon$ **faire**
  $sample \leftarrow in[I] \times filtre.b0 + i1 \times filtre.b1 + i2 \times filtre.b2 - o1 \times filtre.a1 - o2 \times filtre.a2$
  $out[I] \leftarrow sample$
  $i2 \leftarrow i1$
  $i1 \leftarrow in[I]$
  $o2 \leftarrow o1$
  $o1 \leftarrow sample$
fin
Retourner : $i1$, $i2$, $o1$, $o2$

Algorithme 5 : Filtre Passe-bande de SOX

6.1 Filtre biquad

6.1.1 Introduction

Cette section et les suivantes présentent une évaluation croisée de plusieurs approches de la génération de code suivant un benchmark basé sur l'application `SOX` [8].

`SOX` est un outil de traitement du signal destiné à pouvoir traiter des flux audio en ligne de commande. Il intègre une grande panoplie d'algorithmes. Dans le cadre de l'expérimentation, nous avons repris un de ces algorithmes, "*lsx_biquad_flow*" afin de tester nos méthodes de génération.

La suite de cette section présentera l'algorithme et les différentes applications de tests réalisées utilisant chacune une méthode de génération de code différent. Un seul algorithme est testé à cause du temps de développement requis pour créer la *compilette*, mais l'accent a été mis sur l'étude de différente méthodes de générations.

6.1.2 Algorithme du filtre SOX

L'algorithme 5 présente l'algorithme du filtre de `SOX` tel qu'il est implémenté par les programmes de tests des sections 6.2 et 6.3. L'algorithme est une simple boucle parcourant l'ensemble des coefficients d'un signal. À chaque étape on effectue la somme pondérée du coefficient courant avec les deux précédents coefficients du signal d'entrée et les deux derniers coefficients de sortie calculés. Au total, 9 opérations flottantes sont effectuées, 5 multiplications, 2 additions et 2 soustractions.

La spécialisation est faite sur les 5 valeurs du filtre, cela ne cause de pas changement au niveau du flot de contrôle de la fonction, mais réduit le nombre de chargements en forçant la supposition que le filtre ne change pendant l'exécution.

Le filtre est simple mais dans son implémentation naïve (celle que l'on trouve dans `SOX`) quelques points ne permettent pas d'optimiser correctement cette fonction. Dû aux problèmes de l'aliasing, les coefficients du filtre sont rechargés à chaque tour se qui ralentit la fonction. Avec LLVM, l'ordonnanceur donne même une solution peu efficace au niveau des chargements en ne tirant pas parti des capacités de l'architecture de notre test.

Générateur	Contrôle sur le code généré	Facilité de mise en place[1]	Potentiel d'optimisations
`LLVM & Spe`	-	+	++
`deGoal`	++	–	+++[2]
`Kahuna`	-	++	+

[1] Pour créer la routine permettant de passer de la représentation intermédiaire à une fonction utilisable.
[2] L'utilisation du potentiel dépend du programmeur.

TABLE 6.1 – Appréciation qualitative des capacités des générateurs de code.

6.1.3 Modèles de génération de code

Pour réaliser une évaluation globale des différents modèles de génération de code, nous testerons au total 7 approches : statique, en compilation juste-à-temps avec ou sans spécialisation du code et 2 niveaux d'optimisations avec `LLVM` et enfin la génération avec une faible surcharge avec `deGoal` et `Kahuna` (nos 2 approches décrites dans le chapitre 4). Le but est de montrer la compétitivité des approches bas-coût en matière de qualité du code généré, du générateur de code et de l'utilisation de la mémoire.

Les approches que nous considérons sont les suivantes :

`Statique` Version statique et originale du filtre, compilée en utilisant `clang` et `LLVM` avec l'option `-O3`. Cette application nous servira comme base de référence.

`LLVM O0` Version non spécialisée, compilée via le "`LLVM Execution Engine`" et avec le niveau d'optimisation `-O0`.

`LLVM O3` Même chose que précédemment, mais en utilisant le niveau d'optimisation `-O3`.

`LLVM & Spe O0` Version spécialisée à l'exécution au niveau de la RI LLVM et compilée via le "`LLVM Execution Engine`" et avec le niveau d'optimisation `-O0`.

`LLVM & Spe O3` Même chose que précédemment, mais en utilisant le niveau d'optimisation `-O3`.

`deGoal` Filtre spécialisé grâce à `deGoal` qui bénéficie notamment d'une sélection manuelle et spécifique des instructions.

`Kahuna` Filtre spécialisé grâce à `Kahuna` qui bénéficie de la génération automatique du spécialiseur une fois le code annoté.

Les capacités des spécialiseurs varient fortement tant en matière de contrôle sur le code généré que sur la mise en place ou que sur les optimisations potentielles. Le tableau 6.1 résume de façon qualitative les différentes approches. `LLVM & Spe` avec la machinerie LLVM a un fort potentiel d'optimisation sur le code spécialisé, là où `Kahuna` et `deGoal` sont contraints d'anticiper ces optimisations. Sur la mise en place de l'infrastructure, `Kahuna` effectue le travail de façon automatique une fois le code annoté, contrairement à `deGoal` où l'écriture se fait du point de vue du générateur et à un niveau assez bas. Dans le cas de `LLVM & Spe`, il "suffit" de produire le code LLVA et de lancer le JIT sur la RI. Enfin, le grand avantage

```
1  unsigned i;
2  signal_ty i1 = p->i1, i2 = p->i2, o1 = p->o1, o2 = p->o2;
3  for (i=0; i<veclen; i++, ibuf++)
4  {
5    signal_ty b0 = p->ker.b0;
6    signal_ty b1 = p->ker.b1;
7    signal_ty b2 = p->ker.b2;
8    signal_ty a1 = p->ker.a1;
9    signal_ty a2 = p->ker.a2;
10
11   signal_ty o0 = b0 * *ibuf
12     + b1 * i1
13     + b2 * i2
14     - a1 * o1
15     - a2 * o2;
16
17   i2 = i1;
18   i1 = *ibuf;
19   o2 = o1;
20   o1 = o0;
21
22   outbuf[i] = o0;
23 }
24 p->i1 = i1; p->i2 = i2, p->o1 = o1; p->o2 = o2;
```

LISTAGE 6.1 – Implémentation de base du filtre.

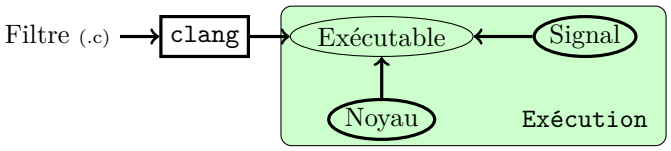
de `deGoal` sur les 2 autres générateurs est son contrôle sur le code généré, son code étant écrit à un niveau pseudo-ASM.

Les implémentations, faites dans différents langages, sont fonctionnellement équivalentes et suivent l'algorithme 5.

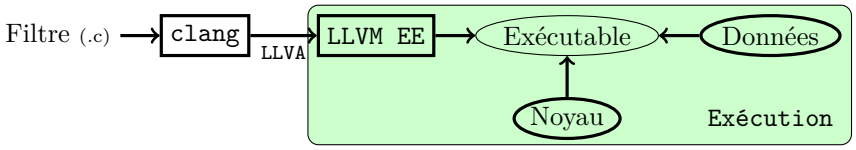
`Statique :` Cette version reprend directement l'implémentation de `SOX` (listage 6.1). L'approche est ici incapable de s'adapter dynamiquement. L'application est compilée statiquement de façon classique avec le niveau d'optimisation `-O3`, comme illustré dans la figure 6.1a.

`LLVM` Le processus de génération du code est illustré dans la figure 6.1b. Cette version reprend l'implémentation de `SOX` mais compilée en LLVA. Au moment de l'exécution, le code à octet est chargé et compilé pour la plate-forme en utilisant le `LLVM Execution Engine` pour obtenir une fonction utilisable. Le moteur JIT de LLVM permet la sélection du niveau d'optimisation, dans notre cas nous utiliserons soit l'option standard `-O0` soit `-O3`. Une fois ce code obtenu, il est utilisé de la même façon que pour la version `statique`. Aucune spécialisation n'est effectuée à l'exécution.

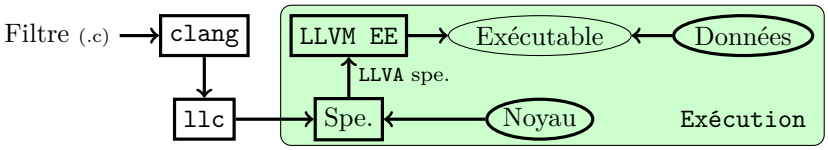
`LLVM & Spe` Le processus de génération du code est illustré dans la figure 6.1c. Cette version reprend l'implémentation de `SOX`. L'application est compilée en LLVA comme pour la version précédente, mais les chargements des coefficients du filtre sont replacés par des intrinsèques. Ce filtre modifié



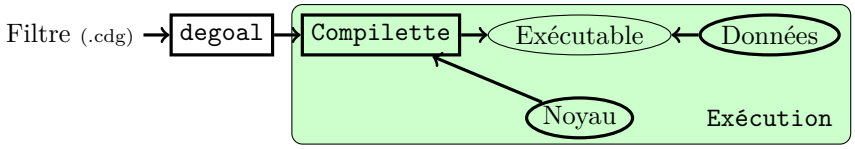
(a) Processus de génération de la version `statique`.



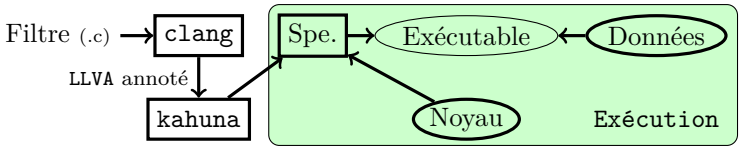
(b) Processus de génération de la version `LLVM`.



(c) Processus de génération de la version `LLVM & Spe`.



(d) Processus de génération de la version `deGoal`.



(e) Processus de génération de la version `Kahuna`.

FIGURE 6.1 – Les différents processus de générations utilisés et évalués.

est ensuite transformé grâce à la partie arrière de LLVM ciblant le C++ pour produire une routine utilisant l'interface de LLVM et qui produira le noyau spécialisé. Au moment de l'exécution, la routine produit le noyau spécialisé dans la RI de LLVM. Cette RI est ensuite compilée pour la plate-forme en utilisant le `LLVM Execution Engine` pour obtenir une fonction utilisable. Comme précédemment, nous utiliserons 2 options standard pour les niveaux d'optimisations : `-O0` et `-O3`.

`deGoal` Cette version reprend l'implémentation de `SOX` avec la spécialisation. Le processus de génération créé une *compilette* qui lors de l'exécution produit le code exécutable en utilisant le coefficient du noyau. Le processus de génération du code est illustré dans la figure 6.1d.

```
1 KAHUNA_INPUT(signal_ty, b0, "filter_build", 0);
2 KAHUNA_INPUT(signal_ty, b1, "filter_build", 1);
3 KAHUNA_INPUT(signal_ty, b2, "filter_build", 2);
4 KAHUNA_INPUT(signal_ty, a1, "filter_build", 3);
5 KAHUNA_INPUT(signal_ty, a2, "filter_build", 4);
```

Listage 6.2 – Implémentation Kahuna.

Kahuna L'implémentation de SOX a été légèrement modifié pour utiliser le frontal basé sur clang (laissant transparaître les intrinsèques Kahuna au niveau C, voir section 4.3.10). À la place des chargements des lignes 5 à 9 du listage 6.1, nous avons inséré les intrinsèques Kahuna pour faire une référence au spécialiseur (listage 6.2). L'implémentation est compilée dans la RI LLVM grâce à notre clang modifié, le code LLVM ainsi obtenu passe la chaîne de compilation de Kahuna normalement pour produire un générateur de code en-place (figure 6.1e).

6.2 Expérimentations

6.2.1 Métriques étudiées

Pour évaluer les différentes méthodes de génération, nous avons évalué les bénéfices et les coûts selon 3 caractéristiques :

- Vitesse d'exécution :
  - Code généré, c'est-à-dire l'évaluation de la qualité du code généré vis-à-vis de la vitesse d'exécution.
  - Générateur de code, c'est-à-dire le coût de la spécialisation vis-à-vis de la vitesse d'exécution de la méthode spécialisée.
- Mémoire :
  - Taille du programme statique, qui permettra de mesurer le coût de l'ajout du spécialiseur.
  - Taille du code généré.
  - Utilisation de la RAM et du tas.
- Consommation énergétique :
  - Code généré, c'est-à-dire l'évaluation de la qualité du code généré vis-à-vis de la consommation.
  - Générateur de code, c'est-à-dire le coût de la spécialisation vis-à-vis de la consommation de la méthode spécialisée.

Le protocole de test diffère selon la caractéristique considérée.

Dans la suite, nous procéderons donc à la mesure des performances des générateurs des points de vue des noyaux et des générateurs de code. Les implémentations du noyau étant semblables, leurs comparaisons peuvent se faire naturellement. Pour les générateurs, les 3 approches de spécialisations se sont pas à même de fournir les mêmes transformations. Ces différences sont plus du domaine du qualitatif (flexibilité par exemple) que du quantitatif, ce qui rend la pondération des résultats impossible et rend l'appréciation des résultats plus difficile.

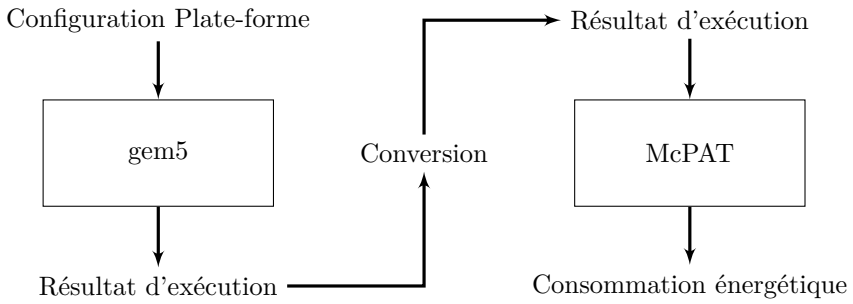


FIGURE 6.2 – Fonctionnement du couple gem5/McPAT.

6.2.2 Vitesses d'exécutions

Les mesures des vitesses d'exécution ont été effectuées sur une Beagleboard-xM, une plate-forme embarquant un Cortex-A8 [5] tournant à 800 MHz. Le CPU est un processeur avec une architecture dual-issue, exécutant les opérations dans l'ordre, et avec une unité de calcul de flottants. Les calculs flottants utilisent l'unité VPF3lite de la plate-forme. Le système d'exploitation est une Ubuntu 11.04 avec un noyau Linux version 3.9.11. Nous avons aussi désacitivé le système de gestion de l'énergie. Celui-ci peut faire varier la fréquence pour réduire la consommation, il risquait d'introduire un biais dans la mesure.

Les mesures sont faites en nombre de cycles CPU, obtenues grâce aux compteurs de performances du processeur. Chaque noyau est exécuté 10 fois et la mesure retenue est le temps d'exécution moyen de ces 10 exécutions.

6.2.3 Utilisation de la mémoire

La mesure de l'utilisation dynamique de la mémoire a été effectuée sur la beagleboard-xM en utilisant le cadriciel Valgrind [66] dans sa version 3.9 et son greffon `Massif`. Ce greffon permet d'obtenir une analyse précise de l'état du tas et de la pile.

Dans l'expérimentation nous retenons l'utilisation maximale de la mémoire (en octets) de l'application comme mesure.

6.2.4 Consommation énergétique

Les mesures de consommation énergétique ont été effectuées grâce au couple de simulateur gem5 [13] et McPAT [56], modifié afin de pouvoir simuler le comportement du Cortex-A8 [28]. gem5 est un simulateur micro-architectural produisant des statistiques sur les instructions émises. McPAT est un simulateur permettant l'estimation de la consommation énergétique de la puce. Le simulateur gem5 ne permet normalement que la simulation d'architecture out-of-order comme le Cortex-A9. Grâce au travail de F. Endo, les architectures in-order peuvent être simulées à partir du modèle out-of-order, ce modèle ayant été validé à l'aide d'une Beagleboard-xM [28]. Comme schématisé figure 6.2, à partir des résultats de simulation de gem5, McPAT réutilise ces résultats d'utilisation du processeur et effectue la simulation énergétique en prenant en compte notamment la température du cœur. Les paramètres du simulateur sont présentés dans

Paramètres		Valeur
Technology node		45 nm
Core clock		800 MHz
Memory controller and caches block size		64 bytes
DRAM	Size	256 MB
	Clock	166 MHz
	Latency[1]	65
L2	Size	256 kB
	A/L/M/W[1,2]	8/8/16/8
L1-I	Size	32 kB
	A/L/M/W[1,2]	4/1/1/0
L1-D	Size	32 kB
	A/L/M/W[1,2]	4/1/1/1
Global Branch Predictor	Entries	512
	Bits	2
Branch target buffer entries		512
Return address stack entries		8
I-TLB/D-TLB entries		32 each
Issue width		2
Execution stage depth		6
Pipeline stages		13
Instruction queue entries		16
Load store queue entries		12 each

[1] Les latences (latency) sont en nombre de cycles du cœur.

[2] A/L/M/W means Associativity/Latency/Miss status and handling registers/Write buffers.

TABLE 6.2 – Paramètres de notre plate-forme de simulation configuré de façon similaire à la Beagleboard-xM. Les termes ont été laissés en anglais, la traduction nuisant à la compréhension de ce point technique.

le tableau 6.2 pour la mesure de la consommation énergétique. La température initial du cœur a été fixé à 27 °C.

Les noyaux étant assez court, les simulateurs peuvent être imprécis dans la mesure de la consommation énergétique en valeur absolue. Il reste néanmoins possible de comparer la consommation relative (c'est-à-dire le noyau A consomme x % de plus que le noyau B). Pour cette raison la mesure de la consommation relevée sera la consommation relative à la consommation de la version `statique`.

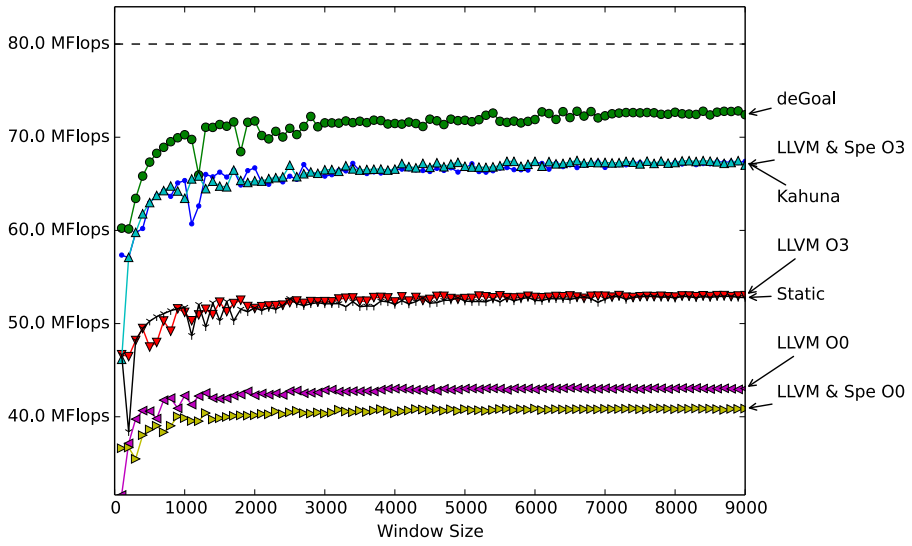


FIGURE 6.3 – Performance (en Flops) obtenue avec les différentes méthodes de génération de code. La ligne en pointillé représente la performance atteignable maximum théorique. Les valeurs les plus élevées sont les meilleures.

Approche	Nombre d'échantillons par Seconde	Gain (%)	Perf. crête (%)
`LLVM & Spe O3`	$7,5.10^6$	21	84
`deGoal`	8.10^6	27	91
`Kahuna`	$7,5.10^6$	21	84

TABLE 6.3 – Gain des différentes méthodes de génération par rapport à la version `statique` et positionnement par rapport à la performance crête.

6.3 Résultats & discussions

6.3.1 Vitesses d'exécutions

6.3.1.1 Vitesse d'exécution des codes générés

La figure 6.3 présente les performances atteintes par les différentes versions étudiées. Les performances sont mesurées en Flops, c'est-à-dire $\frac{9 \times Elts}{t}$ avec $Elts$ le nombre d'éléments traité et t le temps d'exécution en seconde, 9 étant le nombre d'opérations flottantes effectuées par itération. La ligne pointillée représente la performance théorique maximale du CPU. Tournant à 800 MHz avec une unité de calcul flottant non pipeliné et une latence de 10 cycles, cela donne une limite à environs 80 MFlops. Il est à noter que les versions `Kahuna` et `LLVM & Spe O3` se recouvrent, la même remarque s'appliquent pour les versions `statique` et `LLVM O3`. Le tableau 6.3 présente les gains en performance par rapport à la version `statique` (en pourcentage) avec la performance crête en nombre d'échantillons traité par seconde ($\frac{Flops}{9}$) ainsi que la performance par rapport à la performance théorique maximale. Le tableau ne représente que les versions plus rapides que la version `statique`.

Les versions `LLVM O0` et `LLVM & Spe O0` sont de façon peu surprenante les plus lentes du fait du manque d'optimisations, avec une perte de performance de 22 % et 29 % respectivement par rapport à la version `statique`, soit respectivement 53 et 51 % de la performance maximale théorique, là où la version `statique` est à 66 %. La spécialisation est même ici contre productive pour le cas de la version `LLVM & Spe O0` à cause de l'agencement des instructions de chargement dans la boucle. Le Cortex-A8 a la capacité de regrouper les chargements faits sur une même ligne de cache et de ne faire qu'un seul chargement améliorant ainsi les temps d'accès à la mémoire. L'agencement de ces versions ne tire pas partie de ce fait, et dans la version spécialisée la fonction cale de façon plus fréquente que dans la version non spécialisée.

Les versions `statique` et `LLVM O3` obtiennent les même performances, atteignant 66 % performance maximale théorique. Le noyau est produit par la même partie arrière dans les 2 cas, le code résultant est donc ici similaire.

La version `LLVM & Spe O3` est plus rapide que les versions précédentes. On obtient un gain de 21 % par rapport à la version `statique` obtenue par une réduction des instructions de chargement pendant les itérations, soit 84 % de la performance maximale théorique. Dans la version `statique` et `LLVM`, le compilateur ne peut pas légalement déplacer les instructions de chargement en dehors de la boucle car celles-ci peuvent changer (à cause de l'aliasing). Dans cette version, les coefficients sont déplacés hors de la boucle et mis dans un "pool" de constantes. Le "pool" de constantes est une zone mémoire située juste après la définition de la fonction regroupant les constantes ne pouvant tenir sur un immédiat. L'avantage est que les constantes sont regroupées de manière compacte et que le compilateur a la garantie de la constance des données et peut donc optimiser en conséquence. À l'exécution, le chargement est donc fait une fois et hors de la boucle, évitant ainsi les effets d'un mauvais ordonnancement des instructions qui faisaient caler l'application.

La version `Kahuna` obtient les même performances que la version `LLVM & Spe O3` avec un gain de 21 % par rapport à la version `statique` (84 % de la performance maximale théorique). Contrairement à la version `LLVM & Spe O3`, les constantes sont transférées via déplacement d'immédiats vers des registres ARM puis vers les registres de l'unité flottante. Dans les faits, cela n'a pas d'impact notable sur les performances. La génération du noyau avec `Kahuna` utilise, de manière indirecte, la même partie arrière que la version `LLVM & Spe O3`. Au vu de la simplicité de l'application dans ce cas, le code produit est donc très similaire. Les principaux changements sont au niveau du préambule de la boucle, donnant une variation (si elle existe) dans les performances négligeables.

La version `deGoal` est la plus rapide avec un gain de 27 % par rapport à la `statique`, soit 91 % de la performance maximale théorique. Ce gain est obtenu en grande partie pour les même raisons que pour `Kahuna` ou pour `LLVM & Spe O3`. Il reste néanmoins 6 points entre `deGoal` et les 2 méthodes de spécialisation basées sur un compilateur. `deGoal` permet de contrôler de façon fine les instructions générées. La version de `deGoal` utilise des opérations de multiplication et d'accumulation (`MAC`) qui sont aussi rapides qu'une simple multiplication [5]. Les versions `LLVM & Spe` et `Kahuna` ne font pas appel à ces opérations et sont donc ralenties par rapport à `deGoal`.

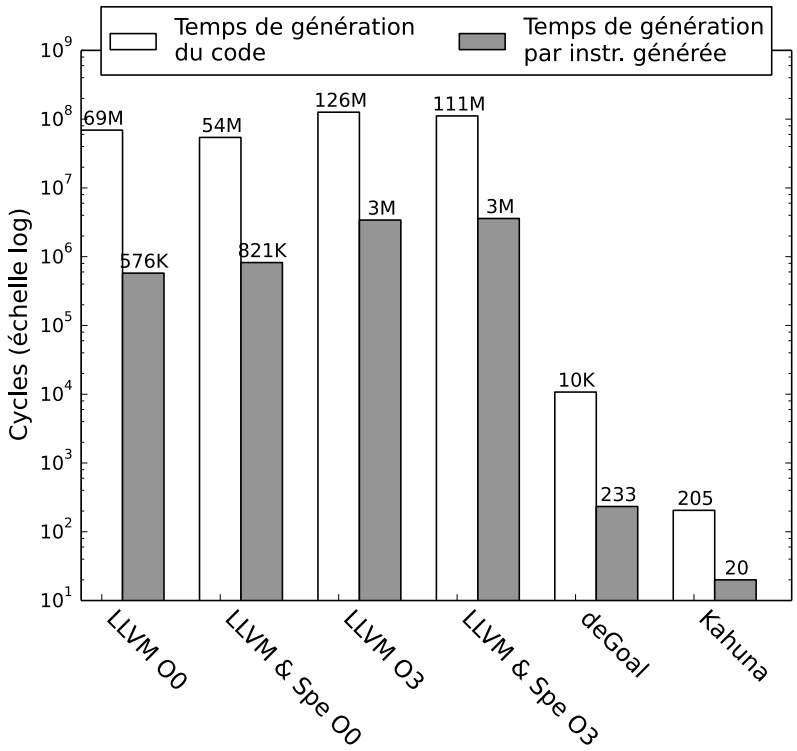


FIGURE 6.4 – Temps d'exécution du générateur de code (en échelle log). Les valeurs les plus faibles sont les meilleures.

6.3.1.2 Vitesse d'exécution des générateurs de code

La figure 6.4 présente sur une échelle logarithmique les temps d'exécution en cycles CPU des différents générateurs de code.

La spécialisation des générateurs `deGoal` et `Kahuna` permettent une génération très rapide du code. `deGoal`, avec sa stratégie hors-place, permet la génération du code à un rythme moyen de 233 cycles par instruction générée. Les instructions sont émises une par une, qu'elles soient invariantes (c'est-à-dire n'utilisant pas d'immédiat) ou nécessitant l'injection d'immédiat. `deGoal` effectuant une allocation de registre à l'exécution, il reste nécessaire de construire les instructions invariantes, car les registres utilisés sont inconnus lors de la compilation statique de la *compilette*. `Kahuna` dans son mode hors-place permet une génération du code à un rythme similaire à `deGoal`. Dans son mode en-place, le rythme moyen est de 20 cycles par instruction générée, ici seule les instructions utilisant des constantes à l'exécution sont émises, réduisant ainsi la quantité d'instructions à émettre à son strict minimum. Le code du générateur en-place ne va garder que les instructions à assembler avec l'immédiat et n'est donc composé que d'une série de `ou` bits à bits, `non` bits à bits, décalages de bits et instructions de stockage. À l'exécution aucun ordonnancement n'est effectué. Dans le cas de `Kahuna` il est fait à la compilation et dans le cas de `deGoal` il est fait manuellement avant la compilation.

La généricité de LLVM permet d'effectuer de nombreuses optimisations, mais

Approche	EPS	Temps de Génération (en s)	Éléments traitables par la version `statique`	Récupération du surcoût
`Statique`	$5,9.10^6$	N/A	N/A	N/A
`LLVM & Spe O3`	$7,5.10^6$	0.15	911 018	4 220 690
`deGoal`	8.10^6	1.32e-05	78	285
`Kahuna`	$7,5.10^6$	3.70e-07	2	10

TABLE 6.4 – Amortissement du surcoût de la génération de code, exprimé en charge de travail. "EPS" est le nombre d'échantillons traités par seconde (Échantillons Par Seconde). La colonne "Éléments traitables par la version `statique`" est le nombre d'échantillons qui auraient pu être traités par la version `statique` pendant la spécialisation. La récupération du surcoût est le nombre d'échantillons requis pour que le processus de spécialisation devienne bénéfique.

à un coût élevé. Contrairement à `deGoal` ou `Kahuna`, le but premier est celui de la portabilité [1] ce qui n'offre pas de possibilité d'anticipation dans la génération du code. Il n'est pas possible, par exemple, d'effectuer une partie de la sélection d'instructions à la compilation statique et le reste à la volée, notamment la sélection des pseudo-instructions dépendantes de la cible. En limitant les optimisations (`-O0`), LLVM requière quelques centaines de milliers de cycles par instruction, et avec le niveau maximum des optimisations (`-O3`), on passe dans l'ordre du million. Pour arriver au code binaire à partir de LLVA, LLVM va utiliser trois autres RI dans les différents étages : SDAG pour la sélection d'instruction, MI pour l'allocation des registres et l'ordonnancement et MC pour l'émission du code. De plus les algorithmes sont gourmands en calcul pour tenir compte des nombreuses contraintes. Dans le meilleur cas, il y a 3 ordres de magnitudes entre LLVM et `deGoal`. On remarque aussi que le temps de génération total du code est plus rapide pour la version `LLVM & Spe` que pour la version `LLVM` (environ 10 M de cycle).

6.3.1.3 Amortissement du coût de la spécialisation

Le tableau 6.4 présente l'amortissement des versions ayant une amélioration effective du temps d'exécution. L'amortissement du surcoût de la génération est exprimé en nombre d'échantillons nécessaires pour que la spécialisation devienne bénéfique pour l'application. Le tableau rappelle également le temps de génération des versions, la colonne "Éléments traitables par la version `statique`" étant la conversion du temps de génération en nombre d'échantillons que pourrait traiter la version `statique` pendant ce laps de temps (i.e. $EPS \times t$ avec t le temps de génération).

La stratégie en-place de `Kahuna` nécessite une charge de travail extrêmement faible pour amortir la génération du code (10 échantillons). `deGoal` nécessite une charge de travail faible pour amortir la génération du code (285 échantillons), mais offre l'avantage de pouvoir conserver les noyaux en fonction des besoins.

Si l'on part du principe que les échantillons sont ceux d'un flux audio échantillonné à une fréquence de 44 kHz, `Kahuna` a besoin d'un signal d'au moins

1. LLVA permet la production d'un code parfaitement portable, même si dans les faits, il est aisé de produire un code non portable, notamment à cause de l'arithmétique des pointeurs.

Approche	Taille statique du programme (en Kb)	Facteur d'augmentation
`Statique`	2.8	N/A
`LLVM`	13 586	4852
`LLVM & Spe`	13 586	4852
`deGoal`	7.9	2.82
`Kahuna`	3.3	1.17

TABLE 6.5 – Empreinte mémoire des approches en Kb (application plus l'outillage pour la spécialisation et la compilation). Pour les différentes versions de LLVM, la variation de la taille est seulement de quelques octets.

0.2 ms pour amortir la génération et `deGoal`, lui, a besoin de 6 ms. L'utilisation de `deGoal` par rapport à `Kahuna` nécessite une charge de travail d'environ 1 294 échantillons, soit 29 ms de flux audio. Au vu du temps de génération du code de LLVM, il en découle naturellement une charge de travail nécessaire pour amortir le coût de la génération nettement plus importante que `deGoal` ou `Kahuna`. À raison de plus de 4 millions d'échantillons, il faut environ 95 s de signal pour rentabiliser la génération du code dans ce contexte. L'amortissement de `Kahuna` et `deGoal` est suffisamment faible pour permettre une re-spécialisation à la volée du filtre, le changement pouvant passer inaperçu à l'oreille.

6.3.2 Utilisation de la mémoire

6.3.2.1 Taille statique des spécialiseurs

Le tableau 6.5 présente la taille, statique, des versions testées. L'utilisation d'un spécialiseur induit nécessairement un surcoût mémoire par rapport à la version générique de l'algorithme, car d'une manière ou d'une autre l'implémentation générique est présente avec les fonctions d'instanciation du code.

Dans sa stratégie en-place, `Kahuna` permet d'avoir une application dont la taille varie peu (environ 10%) comparée à l'approche hors-place de `deGoal`, dont la taille est presque triplé. L'augmentation de la taille du binaire avec `deGoal` est importante à cause de la taille du noyau qui est extrêmement faible.

Dans le cas des approches basées sur LLVM, la taille de la partie arrière (réduite à son minimum) occupe une place considérable. Cela étant, il est possible de faire partager la partie arrière, réduisant ainsi l'impact du surcoût mémoire dans le cas où plusieurs applications utilisent cette stratégie.

Dans le cas de `Kahuna` et `deGoal`, le partage n'est pas possible à cause de la spécialisation du générateur. Avec `deGoal` une partie de sa partie arrière pourrait être partagé (la partie génération des instructions) au prix d'un temps de génération très légèrement plus long. Dans le cas de `Kahuna`, la très grande majorité des instructions de spécialisation est générée, ce qui rend le partage plus limité.

6.3.2.2 Taille des codes générés

La figure 6.5 présente la taille des noyaux générés par les différentes stratégies de génération de code. Pour les 7 versions la figure présente le nombre

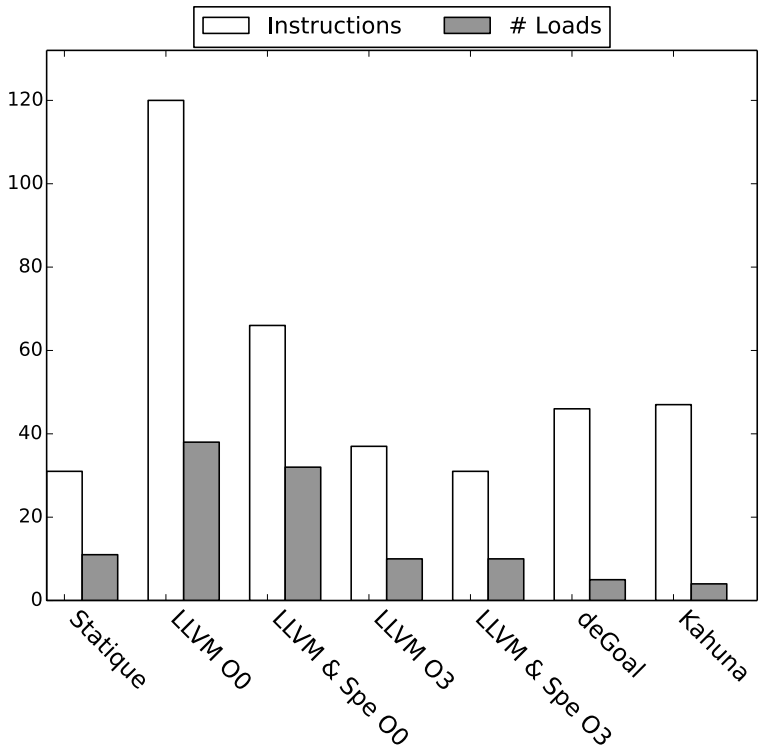


FIGURE 6.5 – Taille des noyaux de calcul générés : nombre d'instructions total et nombre d'instructions de chargement.

d'instructions générées (en blanc) et le nombre d'opérations de chargements (en gris).

Les versions `LLVM O0` et `LLVM & Spe O0` ont un nombre d'instructions élevé dû à l'absence d'optimisation. Le plus grand nombre d'instructions de chargement peut être expliqué par l'absence de la passe d'optimisation des opérations de chargements et stockages et la stratégie de l'allocateur de registre conçue pour être rapide et non pour fournir un code optimal.

Les versions `statiques`, `LLVM O3` et `LLVM & Spe O3` sont les versions qui génèrent le moins d'instructions. Le nombre d'instructions de chargements effectué est le même pour les 3 versions. La version `LLVM & Spe O3` stocke les coefficients spécialisés dans un pool de constantes. La différence dans les vitesses d'exécutions vient du fait que les chargements effectués pour `LLVM & Spe O3` se font avant l'entrée dans la boucle, contrairement aux deux autres qui les effectuent dans la boucle. De plus l'ordonnancement produit par LLVM n'est pas optimal pour le Cortex-A8 et vient donc aggraver la pénalité d'avoir à recharger les coefficients dans la boucle, faisant ainsi caler fréquemment le CPU.

`Kahuna` et `deGoal` produisent tous les deux des codes de tailles similaires et avec moins d'instructions de chargement que pour le groupe précédent. Ces outils insèrent les coefficients sous forme d'instructions de mouvement de données (`MOV`), ce qui augmente la taille du généré mais permet de réduire le nombre de chargements effectués.

Approche	Taille max. du tas	Taille max. du tas par instruction	Taille max. de la pile
`LLVM O0`	227 240	1 893	16 480
`LLVM & Spe O0`	229 224	3 473	15 864
`LLVM O3`	383 912	10 376	26 312
`LLVM & Spe O3`	386 112	12 455	22 744
`deGoal`	184	4	2 984
`Kahuna`	0	0	2 984

TABLE 6.6 – Consommation dynamique de la mémoire (en octet).

6.3.2.3 Utilisation dynamique de la mémoire

Le tableau 6.6 présente l'utilisation dynamique de la mémoire en octet, c'est-à-dire l'utilisation du tas et de la pile lors d'une exécution du générateur de code et du filtre tel qu'enregistré par Valgrind. Cet aspect est important dans les systèmes contraints où la taille de la RAM est limitée. Un effort doit donc être fait pour s'assurer de la bonne gestion du tas et de la pile.

La consommation du tas par LLVM est relativement modeste, culminant à une douzaine de Kb par instruction générée. Celle de `deGoal` est limité à l'allocation du tampon pour le code généré. Cette taille est optimisée manuellement pour allouer la taille strictement nécessaire afin d'éviter un gaspillage de la mémoire. L'optimisation manuelle de la taille est faite avec un effort relativement modeste au regard de la façon dont le code `deGoal` est écrit. De façon logique, `Kahuna` n'utilise pas le tas dans son mode en-ligne. Sur l'utilisation de la pile, `Kahuna` et `deGoal` est déterminé principalement par les appels faits par l'application de test.

La spécialisation du générateur de code diminue les besoins en mémoire, `Kahuna` et `deGoal` effectuant peu d'appels, gardant ainsi l'utilisation de la pile à un minimum. Il est tout de même à noter qu'un certain nombre de variables utilisées par ces systèmes sont statiques, c'est-à-dire qu'elles sont déjà comptabilisées dans la taille statique de l'application. Cela donne l'avantage d'être plus facilement mesurable (l'utilisation de la commande "`du`" dans un shell est suffisante) car ces variables sont incluses dans l'empreinte mémoire statique des applications.

6.3.3 Consommation énergétique

6.3.3.1 Consommation des codes générés

La figure 6.6 présente la consommation énergétique du code généré par éléments du processeur. La figure 6.7 présente la consommation énergétique du code généré par éléments du cœur de calcul. Enfin la figure 6.8 présente la consommation énergétique du code généré par éléments du cœur de calcul. Pour les 3 figures, les résultats présentés ont été mis à l'échelle par rapport à la version `statique`.

Les résultats peuvent être séparés en 3 groupes distincts :

1. `LLVM O0` et `LLVM & Spe O0` qui exécutent les plus grands nombres d'instructions, comme le montre la figure 6.8, et qui sont les versions qui ont la plus grande consommation énergétique.

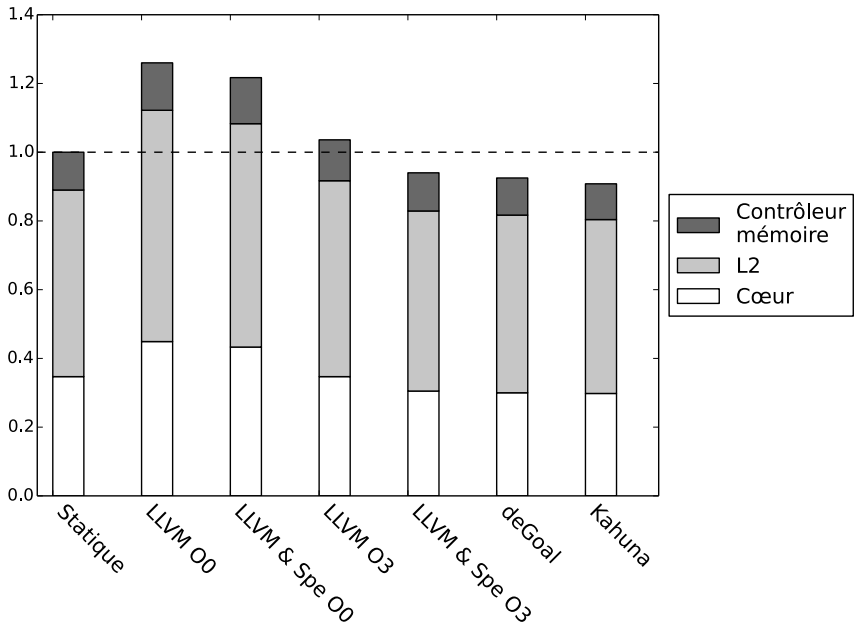


FIGURE 6.6 – Consommation énergétique des noyaux générés par composant du processeur. Les valeurs sont mises à l'échelle par rapport à la version `statique`. Les valeurs les plus faibles sont les meilleures.

2. `Statique` et `LLVM O3` qui donnent une consommation intermédiaire.
3. `LLVM & Spe O3`, `deGoal` et `Kahuna` qui donnent les noyaux plus efficaces avec une économie d'énergie d'environ 10 %.

À cette température d'exécution, la majorité de la consommation est dû aux courants de fuite (de l'ordre de 75 %). Par conséquent, les groupes 1 et 2 ayant un temps d'exécution plus long ils consomment donc plus que le groupe 3.

En termes de consommation dynamique, la réduction vient principalement du cœur. La figure 6.7 présente de façon plus détaillée la consommation au sein du cœur. La consommation du cœur a été séparée en 4 groupes :

EXE Unité d'exécution. Ce groupe inclut les bancs de registres entiers et flottants et les unités fonctionnelles.

IFU Unité de "fetch". Ce groupe inclut le cache L1 d'instruction, la prédiction des branchements et le décodage des instructions.

LSU Unité chargement et écriture. Ce groupe inclut les files de chargement et écriture ainsi que le cache L1 de donnée.

Autres Composants du processeur restants.

C'est le groupe "LSU" qui participe le plus au gain d'énergie. Ce groupe correspond à la consommation liée aux mouvements de données. L'ensemble des données testées est un ensemble de 9 000 échantillons soit 36 KB. La taille du cache L1 est de 32 KB, ce qui est presque suffisant pour faire tenir l'ensemble dans le cache. En conséquence, les requêtes de chargement aboutissent dans la

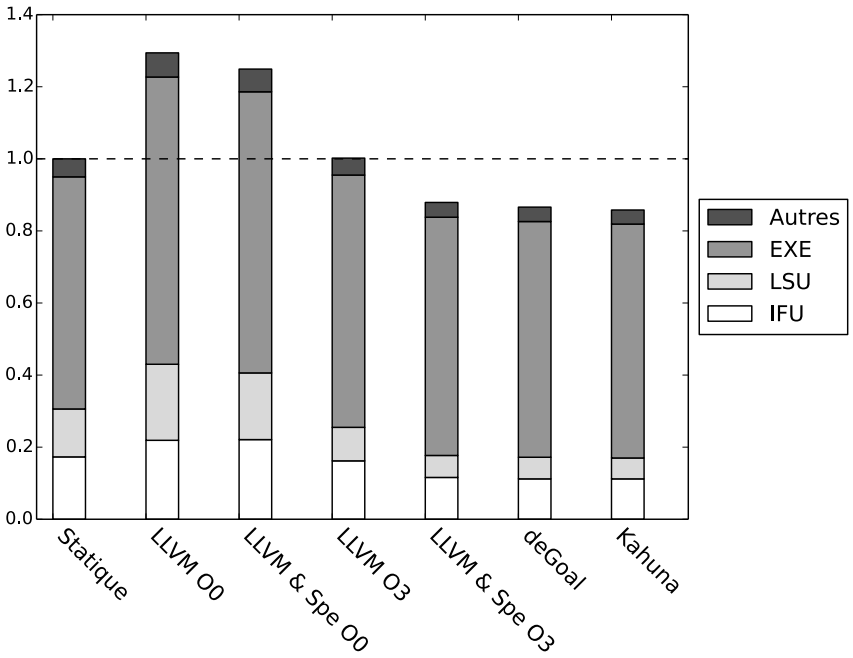


FIGURE 6.7 – Consommation énergétique des noyaux générés par étages du cœur. Les valeurs sont mises à l'échelle par rapport à la version `statique`. Les valeurs les plus faibles sont les meilleures.

grande majorité à un succès de cache ("cache hit") et la consommation induite est comptabilisée dans ce groupe et ne donne pas lieu à un accès au cache L2.

6.3.3.2 Consommation des générateurs de code

La figure 6.9 présente la consommation énergétique des générateurs de code sur un échelle log et mis à l'échelle par rapport à la `LLVM O0`. Comme pour l'exécution, le JIT de LLVM est le plus gourmand en termes d'énergie avec un ratio d'augmentation similaire entre les versions `O0` et `O3`. `Kahuna` grâce à son mode en-place permet un gain d'énergie important par rapport à `deGoal` (d'un facteur 10).

6.3.3.3 Amortissement du coût énergétique de la spécialisation

Le tableau 6.7 présente le coût de la génération sur le même principe que pour les mesures temporelles présentées tableau 6.4. Les gains énergétiques obtenus par la spécialisation du noyau étant plus faibles que ceux obtenus pour la vitesse d'exécution, l'amortissement devient plus difficile. Avec `Kahuna`, le cas le plus favorable, il est nécessaire d'avoir 79 échantillons contre 10 s'il l'on considère uniquement la vitesse d'exécution, soit 8 fois plus. Pour `deGoal` le facteur d'augmentation est de 7 et de 3 pour `LLVM & Spe O3`. Du fait du fonctionnement de gem5 et McPAT, il est possible que les spécialiseurs produits avec `Kahuna` et `deGoal` soient trop petits, et donc que l'erreur d'estimation de l'énergie soit plus

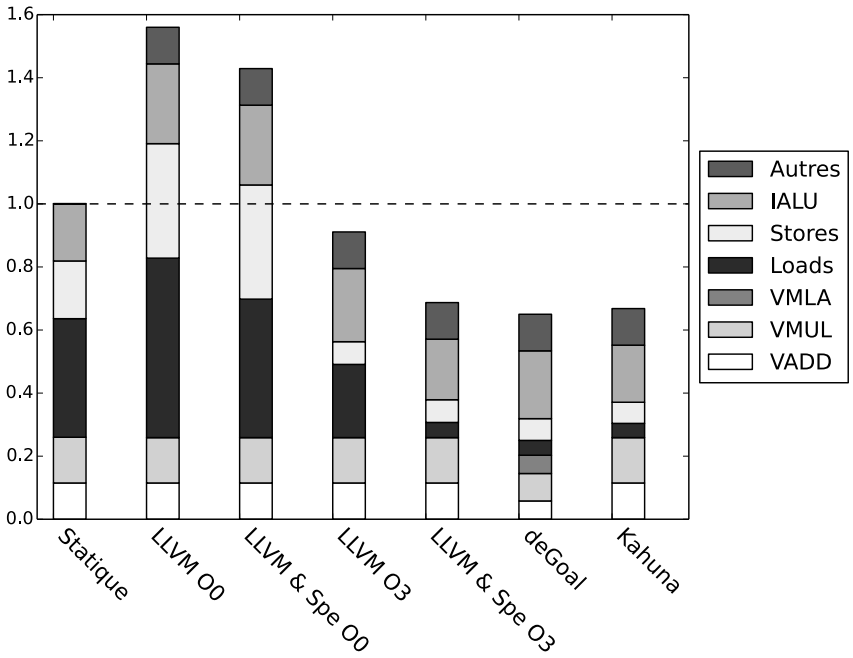


FIGURE 6.8 – Instructions émises (par type) par les noyaux générés. Les valeurs sont mises à l'échelle par rapport à la version `statique`. Les valeurs les plus faibles sont les meilleures.

grande que pour `LLVM & Spe`. Ceci pourrait expliquer la différence de facteurs entre ces versions.

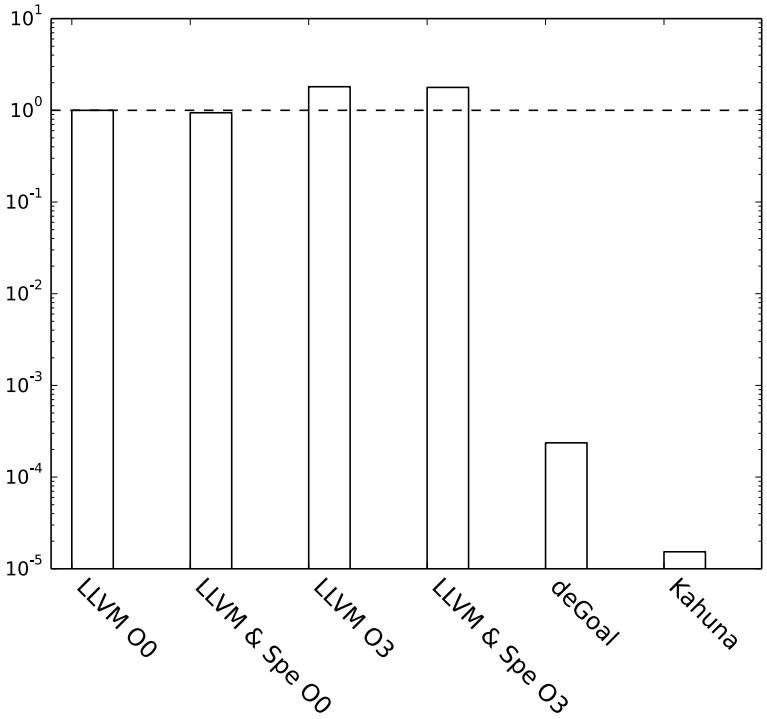


FIGURE 6.9 – Consommation énergétique des générateurs de code (en échelle log). Les valeurs sont mises à l'échelle par rapport à la version `LLVM O0`. Les valeurs les plus faibles sont les meilleures.

Approche	Éléments traitables par la version `statique`	Récupération du surcoût énergétique	Récupération du surcoût temporel	×
`LLVM & Spe O3`	846 309	13 998 419	4 220 690	3
`deGoal`	113	1 502	285	7
`Kahuna`	8	79	10	8

TABLE 6.7 – Amortissement (en nombre d'échantillons) du surcoût de la génération de code d'un point de vue énergétique.

6.4 Conclusion de l'expérimentation

Nous avons étudié plusieurs modèles de génération de code :

`Statique` avec aucune capacité d'adaptation ;

`LLVM` avec aucune capacité d'adaptation mais dont le code final est généré par le JIT ;

`LLVM & Spe` avec une spécialisation basée sur le JIT `LLVM` ;

`deGoal` avec une spécialisation à faible coût utilisant l'approche orientée générateur ;

`Kahuna` avec une spécialisation à faible coût utilisant l'approche fonctionnelle.

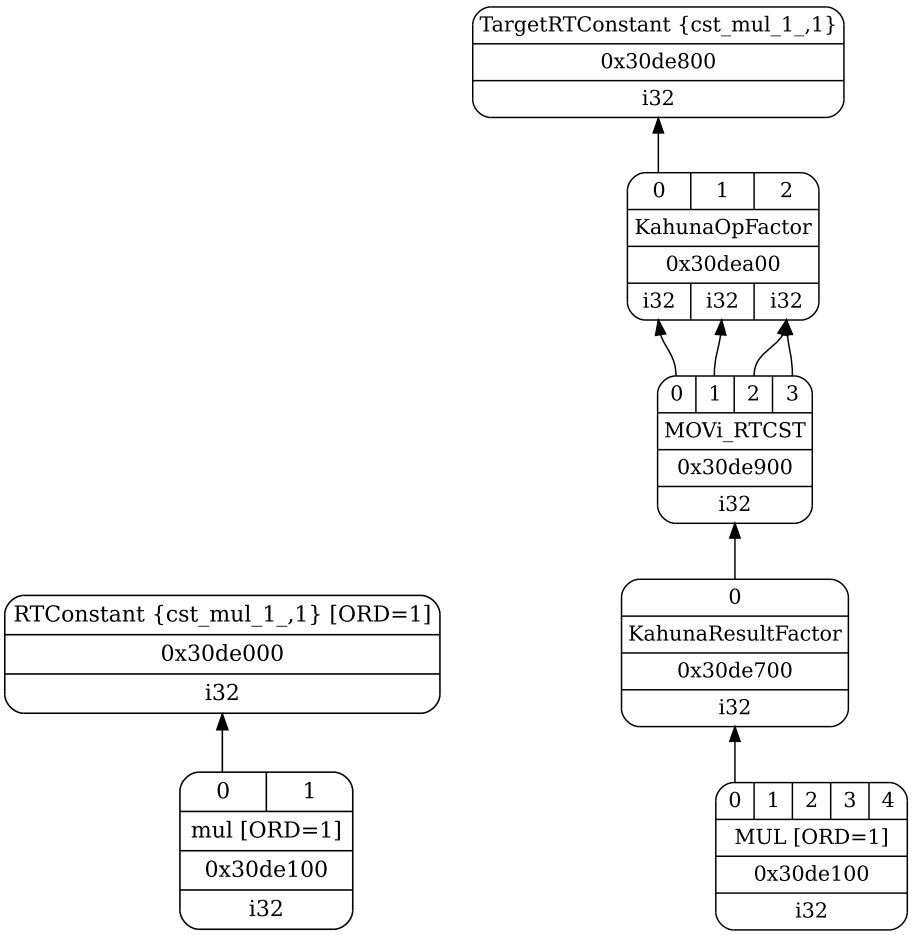
Et nous avons évalué les bénéfices et coûts pour chaque modèle de génération.

Du point de vu du noyau généré, nous avons obtenu un gain de vitesse de 21 % avec `LLVM & Spe` et `Kahuna`, et 27 % avec `deGoal`. Ce gain de vitesse permet aussi un gain de 10 % de l'énergie consommée par le noyau dans les 3 cas de spécialisations. L'approche orientée générateur de `deGoal` est l'approche la plus consommatrice en temps de développement pour obtenir un noyau performant. Mais son utilisation permet le gain le plus important grâce au fait que le code est ordonnancé de façon manuelle. Avec 21 % de gain, les approches automatiques sont une bonne alternative en permettant de réduire grandement le temps d'écriture. Les instructions utilisées par `deGoal` sont plus efficace que celles sélectionnées pour le sélecteur d'instruction de LLVM et donne les 6 points d'écart entre les deux approches.

D'un point de vue coût, le JIT `LLVM` n'étant pas prévu pour ce type d'utilisation, il donne des temps de génération très élevés pour la génération du code. La mesure se fait en centaines de milliers de cycles pour les cas favorables du point de vue temps de génération, mais ne donne pas de bon résultat à l'exécution du noyau généré à cause du manque d'optimisations. Dans le cas le plus favorable du point de vue de la vitesse d'exécution du code généré, le temps de génération est de l'ordre du million de cycles par instruction.

Les approches spécialisées donnent un coût fortement réduit du fait qu'elles sont adaptées à ce type d'utilisation. En mode en-place `Kahuna` permet la spécialisation plus rapide avec 20 cycles par instruction. `deGoal` permet de spécialiser le code avec une moyenne de 233 cycles par instruction en utilisant une approche hors-place.

Les 3 versions améliorant les performances ont aussi un gain de 10 % sur l'énergie consommée. Cette économie est principalement due à la réduction du temps d'exécution, et donc des gains obtenus précédemment. Du point de vue amortissement du coût énergétique de la spécialisation celui-ci est plus difficile que du point de vue du temps d'exécution. En nombre d'échantillons, il faut 3 fois plus d'échantillons à `LLVM & Spe` pour amortir la production de code, 8 fois plus pour `Kahuna` et 7 fois plus pour `deGoal`. Cependant, le coût temporel de la spécialisation étant déjà extrêmement faible pour `deGoal` et `Kahuna`, il reste toujours facilement amortissable du point énergétique.



(a) Extrait de la traduction du LLVA du listage 4.7 vers le SDAG LLVM.

(b) Factorisation des patrons après la sélection des instructions.

FIGURE 6.10 – Extrait de la sélection des instructions avec `Kahuna` issue du listage 4.7.

6.5 Évolution de `Kahuna`

6.5.1 Gestions des alternatives

Lors de la sélection des instructions, la machine à état de LLVM va chercher à faire correspondre le plus gros paquet d'instructions LLVM en instructions machines donnant lieu à une correspondance 1/1. Dans le cas de `Kahuna`, on cherche une correspondance 1/n dû au fait qu'il *peut* exister un environnement favorable pour une certaine sélection. La sélection doit donc être capable de choisir tous les patrons possibles jusqu'à pouvoir gérer tous les cas de valeurs possibles.

Pour illustrer notre propos, nous reprenons l'exemple du listage 4.7 de la section 4.3.7 (page 57) qui effectue une simple multiplication par un scalaire.

En ciblant ARM, nous n'avons pas la possibilité d'avoir une multiplication d'un registre par un immédiat, un `MOV` d'un immédiat vers un registre doit donc être inséré. Avec un immédiat signé, il y a 2 possibilités :

— Soit l'immédiat est positif, dans ce cas un `MOV` est inséré
— Soit l'immédiat est négatif, dans ce cas un `MVN` est inséré (écriture du non binaire de l'immédiat dans un registre)

Pour le moment, la machine à état produite par l'outil `tblgen` a été modifiée pour pouvoir supporter ce mode récursif. Pendant la sélection, la machine à état factorise les patrons entre des nœuds permettant d'exprimer l'alternative à l'exécution.

La figure 6.10 présente l'entrée et le résultat de la sélection d'instruction concernant la multiplication du listage 4.7. La figure 6.10a montre l'extrait du `SDAG` produit par la partie arrière de LLVM après la traduction, c'est-à-dire après la transition de LLVA (ici le listage 4.7) vers la représentation sous forme de graphe acyclique SDAG. L'opération de multiplication du listage est représenté par le nœud `mul` qui prend 2 opérandes : une constante à l'exécution (`%b`) et un registre ((`%a`)). La constante à l'exécution est représentée par un nœud `RTConstant`, le registre n'est pas montré pour des questions de lisibilité. La figure 6.10b présente le résultat de la sélection d'instruction :

— `mul` a été transformé en un `MUL` registre/registre.
— Le nœud `RTConstant` ne pouvant être utilisé directement, un nœud `MOVi_RTCST` a été ajouté qui représente un `MOVi` ARM émis à l'exécution.
— `TargetRTConstant` est le nœud `RTConstant` signalé comme sélectionné.

Pour symboliser les alternatives, deux nœuds ont été ajoutés, `KahunaOpFactor` qui factorise les opérandes des nœuds et `KahunaResultFactor` qui factorise les résultats des nœuds.

Les travaux restants sont de finir l'étape de sélection d'instruction pour exprimer les alternatives et de gérer ces alternatives dans les étages inférieurs, avec l'ordonnancement et l'allocation des registres.

6.5.2 Fonctionnalités souhaitables

Dans le cadre de la spécialisation hors-place, certaines optimisations peuvent être anticipées de façon relativement aisées et exécutées avec un coût faible lors de l'exécution.

6.5.2.1 Élimination du code mort

Bien que sur le principe peu complexe, où il "suffit" de n'évaluer que les blocs basiques voulus et ajuster les sauts, l'implémentation est plus difficile due aux nombreuses phases d'optimisations. Deux phases du flot de compilation doivent être utilisées :

1. Pendant la compilation des patrons. Les opérations de branchements gérées par des opérations constantes sont remplacées par des intrinsèques. Cela implique de rajouter l'outillage nécessaire de façon à ce que les passes LLVM existante interprètent correctement ces opérations dans la partie arrière.
2. Au moment de la linéarisation lors de la génération du spécialiseur. Seul un des blocs basiques est utile, et donc sera intégré dans la fonction finale.

6.5.2.2 Déroulage de boucle à l'exécution

Dans la forme actuelle de `Kahuna`, les itérateurs des boucles ne sont pas reconnus comme des constantes potentielles. D'un point de vue simplifié, il "suffirait" d'évaluer le bon nombre de fois le code de la boucle au moment de la spécialisation. Cela rejoint, sur un certain nombre de points, le problème précédent au niveau de la partie arrière. Un traitement spécifique doit être ajouté au moment de l'analyse au temps liaison pour limiter le facteur de déroulage.

6.5.2.3 Déroulage partiel de boucle à l'exécution

Dans beaucoup d'applications, notamment dans les convolutions, des gardes sont intégrées pour gérer les cas particuliers. Ces gardes dans ces boucles n'ont d'utilité que pour quelques itérations. Dérouler ces boucles pour éliminer les gardes aux cas extrêmes peut être fait à l'exécution quand les paramètres sont fixés, moyennant une analyse lors de l'analyse au temps de liaison en plus de l'optimisation précédente.

6.5.3 Évolutions

6.5.3.1 Interface avec `deGoal`

Comme montré en section 6.3.1.1, `deGoal` peut permettre d'avoir un gain substantiel par rapport à une approche automatique. Cependant il est difficile avec `deGoal` d'écrire de larges fonctions spécialisées (comme il peut l'être pour écrire un programme en assembleur). Il pourrait donc être intéressant de mélanger ces deux approches.

Dans certains langages, comme le C, il est possible d'insérer du code assembleur dans les régions critiques afin de pouvoir utiliser des instructions spécifiques et de contrôler l'ordre. Dans la même idée, des fonctions pouvant bénéficier de la spécialisation, mais où la sélection des instructions ou l'ordonnancement n'est pas critique, pourraient utiliser un système automatique comme `Kahuna`. Pour les parties plus critiques de ces fonctions, on pourrait alors basculer sur `deGoal` pour bénéficier localement d'un contrôle plus fin sur le code généré.

6.5.3.2 Réinstanciation des patrons

En cumulant l'instanciation hors-place et en-place, il pourrait être intéressant de cumuler les deux pour modifier un noyau déjà instancié. En cas de changement de quelques valeurs, il pourrait être bénéfique de ne modifier que ces valeurs. Les figures 6.4 et 6.9 montrent le gain potentiel (d'un facteur 10) entre les deux types d'instanciation.

Il reste cependant à évaluer le coût de cette évolution, car cela implique de suivre la façon dont le code a été créé et donc insérer des structures de contrôle dans le générateur pour s'assurer de l'intégrité du code réinstancié. Cela aura donc un impact en termes d'utilisation mémoire et sur la vitesse d'exécution du générateur.

6.5.3.3 Spécialisation itérative

L'idée est de spécialiser le spécialiseur dans le cas où les données sont fixées au fur et à mesure et n'évolue pas au même rythme dans le temps. La figure 6.11

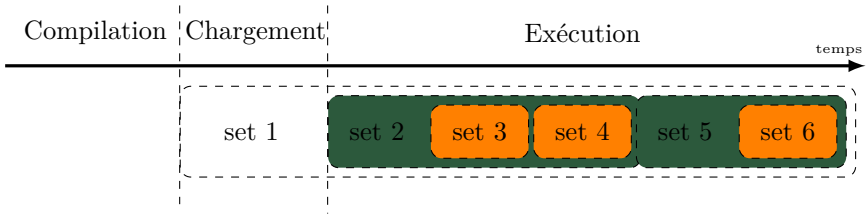


FIGURE 6.11 – Exemple d'évolution des ensembles de données au court du temps. Les différents ensembles ne changeants pas à la même fréquence, il est possible d'effectuer les transformations par étapes. De plus pour les "set 1", étant connue au moment du déploiement, il est possible de partie d'un plus haut niveau et d'effectuer des passes plus complexes.

schématise ce genre de besoin pour une spécialisation itérative. Au moment du chargement de l'application, on connaît l'environnement (set 1), ce qui permet de fixer une partie du spécialiseur. Lors de l'exécution, on a accès à une nouvelle partie des données (set 2) qui vient compléter celles connues précédemment (set 1), le spécialiseur est de nouveau spécialisé avec les informations du set 2. Enfin avec le set 3, on lance la spécialisation de la fonction, dès que ce dernier ensemble évolue (set 4), on peut respécialiser en utilisant implicitement le set 1 et le set 2, car ils ont été intégré au spécialiseur. Avec le set 5, le spécialiseur est réémis. Ceci permettrai d'accélérer le temps de génération si les ensembles intermédiaires ne change pas trop.

Du point de vue compilation multi-temps, cela permet de spécialiser en fonction de l'environnement au moment du chargement, notamment avec les capacités matérielles en raffinant la sélection d'instruction, ajuster des constantes de déroulage *etc.*

6.6 Conclusion

Dans ce chapitre, nous avons donc étudié différentes méthodes de génération de code. Nous avons montré les bénéfices de l'utilisation de spécialiseurs à faible coût du point de vue des trois caractéristiques suivantes : *a)* vitesse d'exécution ; *b)* mémoire ; *c)* consommation énergétique. Nous avons montré les gains possibles avec la spécialisation et le coût de cette spécialisation. Le point de vue énergétique est intéressant, les gains obtenus sont de l'ordre de 10% pour les spécialiseurs les plus performants. Cependant, dans le cas de JIT, le coût de la spécialisation est triplé et devient beaucoup plus difficile à amortir. Le coût de la spécialisation avec `Kahuna` et `deGoal` est multiplié par 8 et 7 respectivement par rapport au point de vue de la vitesse d'exécution, mais il reste facilement amortissable.

Le spécialiseur créé par `Kahuna` montre de bonnes performances, tant sur la vitesse d'exécution, que sur la consommation mémoire et énergétique. Cet outil permet ainsi de créer, facilement, des spécialiseurs générant un code performant avec un coût de génération faible. La dernière partie de ce chapitre a aussi été l'occasion de présenter les possibilités d'amélioration de l'outil afin de le rendre plus performant avec de nouvelles transformations ou usages.

`Kahuna` est un outil qui est conçu pour s'intégrer dans d'autres systèmes, que ce soit des parties frontales ou bien des phases d'optimisations faisant remontrer des informations sur les invariants d'un programme. Ce chapitre nous a permis de voir la possibilité de l'utiliser dans un frontal, et au moment de l'écriture, des travaux sont en cours pour créer des phases d'optimisations utilisant des résultats de profilages. Dans les évolutions envisagées, nous avons présenté le principe de la spécialisation itérative qui peut tirer parti de divers moments, notamment celui du chargement, en utilisant les informations du matériel présent par exemple ou les variables d'environnement.

Chapitre 7

Conclusion

Sommaire

Les architectures tentant de répondre aux nouveaux challenges pour tenir les promesses d'amélioration des performances répondent par une augmentation du nombre de cœurs et non plus par une augmentation de la fréquence. Pour faire face aux besoins croissants de puissance de calcul et aux besoins de réduction de la consommation énergétique, ces architectures se sont fortement complexifiées avec l'utilisation de différents niveaux de caches, l'augmentation du nombre de cœurs, l'hétérogénéité *etc.* Face à la rapide évolution de ces architectures, l'étagement des moments de génération de code offre une réponse adaptée pour l'utilisation de ces architectures. La génération dynamique de code notamment (moments à partir du déploiement) offre un grand potentiel tant du point de vue de la portabilité que de la pleine utilisation des possibilités du matériel. Des stratégies alliant portabilité et bonne utilisation du matériel commencent aussi à émerger (par exemple Android et le runtime ART), devenues possibles avec l'amélioration des techniques de génération de code et des ressources disponibles.

Avec l'augmentation du nombre de cœurs dans les nouvelles architectures certains algorithmes deviennent plus dépendants aux données par l'introduction d'un lien entre ces données et la façon de les paralléliser. Bien que la parallélisation ne soit pas un phénomène nouveau, son ampleur par le nombre de processus qu'il convient de gérer (même dans le cas d'une configuration modeste) entraînent de nouvelles problématiques sur la façon de découper les problèmes.

Les architectures embarquées et contraintes ne sont pas en reste, avec l'arrivée de l'IoT ou des "wearable technology". Elles sont contraintes par la mémoire et par l'énergie qui, dans certains cas, est la préoccupation majeure. Une solution utilisée est la spécialisation de l'architecture au type d'application voulue, ce qui entraîne nécessairement une modification des chaînes de compilations pour assurer les performances logicielles.

7.1 Résumé des réalisations

Les travaux effectués durant cette thèse ont été faits sur 2 axes :

- Sur le calcul haute performance, pour lequel j'ai effectué :
  - La réécriture de l'outil `deGoal` de façon à obtenir une architecture plus modulaire, indispensable à la gestion des architectures aussi diverses que les ARM, le xp70 ou les GPU CUDA.
  - L'identification de la sensibilité algorithme `GEMM` avec l'architecture et des formes des matrices.
  - Le développement d'une couche d'adaptation pouvant apprendre et adapter la configuration de l'algorithme en fonction des entrées.
  - Le contrôle de la taille de la bibliothèque grâce à `deGoal`.
  - L'intégration dans un démonstrateur Scilab.
- Sur les plates-formes embarquées :
  - Le développement d'un outil de génération automatique de spécialiseur.
  - Une étude de l'impact de la génération de code selon trois métriques : vitesse d'exécution, consommation mémoire et consommation énergétique.

Les publications réalisées en premier auteur lors de cette thèses sont [57, 58, 59] et en tant que coauteur [17, 18, 22].

7.1.1 Relation performances et données sur GPU

L'optimisation des performances sur les architectures modernes est un problème difficile. La complexification des architectures rend la tâche plus ardue et les GPU ne font pas exception. L'ajustement des performances sur GPU suivent des règles similaires aux CPU pour l'ordonnancement des instructions ou l'utilisation des caches mais doivent aussi composer avec des caractéristiques qui leur sont propres. La répartition des données aux processus, la charge de travail de chacun ou encore la gestion de la mémoire partagée sont autant de paramètres sur lesquels l'expert doit pouvoir jouer pour maximiser les performances.

Dans le cadre des travaux sur la `GEMM`, nous avons mis en évidence une sensibilité de l'algorithme aux formes des matrices données en entrée. Les précédentes approches ignoraient cette caractéristique, lui préférant un compromis dans la configuration choisie, entraînant une perte de performance de 14 à 16 % en moyenne. L'utilisation de ce type de stratégie est simple à mettre en place, et sur de grandes matrices aux dimensions équilibrées, l'utilisation d'une mauvaise configuration n'a qu'une influence limitée. Cependant avec la réduction des tailles de la matrice, ou lorsqu'une dimension devient plus importante que les autres, la performance décroît rapidement.

Nous avons donc amélioré les performances en explorant à la fois l'espace des configurations de l'algorithme et celui des tailles des matrices. Ce deuxième espace ne pouvant être exploré d'une manière exhaustive, il a été discrétisé de manière à garder une précision suffisante pour inférer les points non connus. À partir des résultats issus de l'exploration, nous avons créé une fonction de décision sélectionnant la variante la plus appropriée à une matrice quelconque. Les gains moyens obtenus sont entre 11 et 13 %, avec un doublement des performances dans certains cas. L'utilisation de la génération dynamique de code permet de regrouper les différentes variantes et d'éviter une explosion de l'empreinte mémoire. L'utilisation de `deGoal` dans ce contexte nous permet de réduire d'un facteur 40 la taille de la bibliothèque comparée à la spécialisation statique classique. Ces travaux ont fait l'objet des publications suivantes [18, 57, 59].

Le développement de `deGoal` a été fait pour tenir compte des particularités du modèle de programmation des GPU, modèle en rupture par rapport à ceux proposés par les architectures plus traditionnelles. Il tourne donc autour d'une architecture souple et modulaire visant à pouvoir faire remonter les particularités du modèle de programmation, notamment les identifiants des processus et la gestion des différents niveaux de mémoire. Une publication générique sur `deGoal` a été présentée à la conférence CC [17] et un chapitre de livre pour la présentation de l'utilisation de `deGoal` sur STHORM [22].

7.1.2 Génération automatique de spécialiseur

Les approches développées dans l'état de l'art pour développer des spécialiseurs se sont focalisées sur l'expression des invariants à l'exécution dans un langage. À l'origine avec des langages fonctionnels, elle s'est étendue à d'autres langages, le plus souvent le C. `deGoal` permet l'expression à un niveau instruction d'un code à générer, ces expressions étant insérées dans des fonctions C. L'insertion de `deGoal` dans un flot de compilation ou de réécriture est contrainte par l'utilisation du C, les besoins de sélection d'instructions et d'ordonnancement.

`Kahuna` est présent dans les parties intermédiaires et arrière de LLVM. Il garde une adhérence à un compilateur (contrairement à Tempo par exemple) mais se détache des langages de haut niveau, permettant l'utilisation par un frontal ou des passes d'optimisations sans repasser par un langage de plus haut niveau. Nous avons étudié ses capacités et les avons comparées à celles de `deGoal` et du JIT LLVM. Les gains obtenus par la spécialisation avec `Kahuna` sont similaires à ceux obtenues avec le JIT LLVM et `deGoal`. Dans le cas de la vitesse d'exécution, `deGoal` a tout de même permis un gain plus important.

La spécialisation des générateurs de code, avec `Kahuna` ou `deGoal`, permet de réduire à un minimum les coûts de spécialisation, que ce soit d'un point de vue vitesse de spécialisation, empreinte mémoire ou bien consommation énergétique. L'aspect consommation énergétique est important car le gain en consommation énergétique dû à la spécialisation est plus faible que le gain en vitesse d'exécution. De même, le coût énergétique est plus important et il en résulte inévitablement un amortissement plus difficile du coût de la spécialisation. Ces spécialiseurs bas-coût peuvent faciliter l'économie d'énergie par le biais de la spécialisation, mais il conviendrait d'étudier plus d'application pour vérifier cette hypothèse.

`Kahuna` a été présenté à ADAPT 2014 [58].

7.1.3 Volume de travail

Voici une estimation du volume du travail effectué :
- Réalisation de l'état de l'art : 6 mois.
- Travaux sur GPU : 12 mois, décomposés avec le portage de `deGoal` en Python et le ciblage des GPU Nvidia (1 mois), la caractérisation du lien entre les données et l'algorithme pour la GEMM et développement de la couche d'adaptation (7 mois), valorisation (4 mois).
- Travaux sur `Kahuna` : 11 mois, le développement s'est déroulé en 2 temps :
  - Utilisation de la partie intermédiaire de LLVM pour la génération de *compilettes* `deGoal`, environ 3 mois. Cette piste a été abandonnée, à cause notamment de problèmes d'ordonnancements.
  - Utilisation LLVM et clang pour la génération du modèle avec patron et présenté dans la thèse, environ 6 mois. La prise en main de la partie arrière, dense et moins bien documentée que la partie intermédiaire, a ralenti le développement.

  Il y a eu enfin 2 mois de valorisation pour `Kahuna`.
- Rédaction de la thèse : 4 mois.

7.2 Discussions

7.2.1 Relation performances et données sur GPU

L'un des points limitant avec l'utilisation du PTX est qu'il contraint `deGoal` à utiliser le compilateur à la volée de Nvidia. Dans l'expérimentation réalisée, ce compilateur représente 99 % du temps de génération du noyau de calcul. Selon le gain de performance obtenu avec la configuration, amortir le surcoût peut être difficile. Il faut de 200 à 800 exécutions pour totalement amortir le coût de la génération. Cependant avec ce schéma, l'utilisation du JIT comme passerelle offre l'avantage de garantir une portabilité fonctionnelle de la bibliothèque. Ainsi en cas de changement de matériel, cette bibliothèque peut toujours fonctionner et produire un résultat correct.

Les performances sont difficilement portables, et l'utilisation seule du PTX ne permet pas cette portabilité. Par contre ce type de fonctionnement offre une flexibilité dans le suivi de l'évolution de l'architecture. Notre bibliothèque *génère* le noyau de calcul à partir des paramètres de configurations choisis lors du déploiement. Mais le générateur permet la génération de noyaux arbitraires.

En cas de changement de configuration matérielle, il "suffit" donc de changer la fonction de décision et la base des configurations pour "*porter*" les performances vers ce matériel. Ce mode de fonctionnement permet aussi d'être fait sans divulguer les sources de l'application, car l'utilisateur n'a pas à compiler ou recompiler les sources.

7.2.2 Génération automatique de spécialiseur

L'idée de `Kahuna` est d'avoir un outil de génération de spécialiseur de code facile à recibler et indépendant d'un langage. Au moment de l'écriture de cette thèse, son développement n'était pas encore à un stade suffisamment avancé qui aurait permis d'avoir un outil "passe-partout". Cependant, il nous a permis d'avoir des points de comparaison par rapport à `deGoal` pour estimer un ratio

temps de développement et gain obtenu. Cela nous a aussi permis d'avoir une vision sur l'utilité de la spécialisation dans l'économie d'énergie. Dans notre expérimentation, les gains énergétiques des noyaux générés sont majoritairement liés aux gains sur la vitesse d'exécution. Mais cela a mis en évidence l'avantage de l'approche de la spécialisation de code employé par `Kahuna` et `deGoal` sur le coût énergétique de la spécialisation par rapport à une génération avec LLVM.

L'utilisation de LLVM permet d'avoir un déport et un contrôle plus important sur le moment de génération du code. On peut donc effectuer la construction du spécialiseur plus tardivement ou de façon plus adaptée qu'il ne l'est aujourd'hui avec les techniques de l'état de l'art. Il est possible avec cet outil de préparer la construction du spécialiseur à un temps statique (phase d'analyse du temps de liaison) sans pour autant connaître la cible [1]. Au moment du déploiement ou du chargement, la connaissance de l'architecture permet alors de finaliser la création du spécialiseur.

7.3 Perspectives

7.3.1 Évolution des architectures

Les architectures modernes ont fortement évolué pour répondre aux nouveaux usages des technologies de l'information. L'apparition des multi-cœurs, comme les GPU Nvidia mais aussi les puces MPPA de Kalray, Epiphany d'Adapteva ou encore STHORM de STMicroelectronics, ont permis une forte augmentation des capacités de calcul accompagnée d'une réduction de la consommation. Dans l'optique de répondre aux besoins croissants de l'industrie dans les besoins en puissance de calcul ou énergétiques, les architectures devraient se tourner de plus en plus vers des multi-cœurs hétérogènes, spécialisant ainsi les types calculs effectués. De cette façon l'industrie espère pouvoir assumer les besoins en puissance de calcul et garder une consommation énergétique acceptable.

L'utilisation d'architectures hétérogènes pose donc aussi la question du support logiciel, et surtout, l'interaction entre les compilateurs, le matériel et ces logiciels. Dans cette thèse, nous avons abordé la question des GPU offrant une grande puissance de calculs mais où les processus sont organisés en grilles et sont contraints d'effectuer le même calcul au même moment sous peine d'une forte pénalité sur les performances. Cela permet de développer un modèle analytique du comportement du matériel, mais contraint fortement les possibilités de calculs.

L'évolution de ces architectures ont poussé la création de langages dédiés à la programmation de ces dispositifs. Le langage OpenCL est sans doute aujourd'hui le plus utilisé ou du moins une implémentation de choix pour beaucoup. Il offre une possibilité unifiée de programmation de ces dispositifs. Néanmoins la question de l'utilisation optimale du matériel reste une question ouverte.

7.3.2 Utilité d'étager la compilation

Pour assurer la rentabilité du développement d'un logiciel il est indispensable d'assurer :

1. Il est entendu dans ce cas que le code LLVA doit être écrit de façon portable.

— De la performance, souvent assurée par la compilation dans l'ère du mono-processus.
— La sécurité de fonctionnement.
— Une certaine portabilité, soit pour permettre un déploiement sur des architectures variées, soit pour permettre l'évolution de la solution. Cela passe souvent par l'utilisation de composants modulaires et génériques, pouvant induire un surcoût à l'exécution si le programme n'est pas correctement optimisé de façon globale.

L'avantage des langages de haut niveau est l'expressivité qu'ils peuvent fournir pour un domaine d'application, offrant vitesse de développement, sécurité et possibilités d'optimisations. Ceci constitue une force dans le sens où se détacher de l'architecture permet une certaine portabilité de l'application. Par contre, plus le langage est de haut niveau, plus il est éloigné de l'architecture et plus il va requérir des transformations et des ajustements pour qu'il soit efficace sur la plate-forme finale. En répartissant les processus de génération en plusieurs moments, on va répartir la charge de calcul et focaliser les efforts d'optimisations sur le moment le plus approprié.

Cela présuppose tout de même d'avoir à disposition une chaîne de compilation capable de répartir le calcul, c'est-à-dire la capacité de manipuler plusieurs représentations et de les faire communiquer, et de choisir les moments pour lancer les optimisations. Cela implique aussi l'adaptation des optimisations aux moments durant lesquels ils sont exécutés afin que la consommation des ressources ne pénalise pas le système. Enfin l'étagement de la compilation permet d'appliquer des méthodes d'optimisations basées sur l'expérience tout en contrôlant les coûts de génération.

La compilation s'est souvent concentrée sur l'optimisation de la vitesse d'exécution. Les contraintes sur les systèmes modernes, notamment environnementales, impose de plus en plus de tenir compte de l'énergie consommée. Cela est vrai sur les systèmes embarqués mais aussi sur les systèmes HPC qui, en visant l'exascale, sont contraints de prendre en compte et de gérer la consommation. En effet, les infrastructures des réseaux électriques ayant une limite en termes de production et d'absorption de charge seront incapables de faire face à la demande si l'évolution suit les projections actuelles. La compilation étagée est aussi une réponse possible pour des optimisations de code multi-objectif, mais aussi adaptative à des besoins locaux et transients.

7.3.3 Poursuite des travaux à plus long terme

Les travaux développés dans cette thèse peuvent trouver leurs place dans des chaînes de compilation plus larges, en spécialisant de façon itérative l'application ou les noyaux.

Sur le thème de la portabilité des performances, point développé pour la GEMM sur GPU, certains groupes de travail étudient la prédiction des performances des noyaux calcul et devient, avec les nouvelles architectures un domaine important de recherche [43]. En extrayant des caractéristiques des noyaux de calcul, des modèles peuvent alors estimer la performance du noyau. Une intrication forte entre le spécialiseur et un modèle de performance permettrait, au déploiement, de s'adapter à l'environnement de façon quasi transparente. Au vu de la sensibilité des performances de l'algorithme aux données, il n'est pas dit qu'il serait possible d'obtenir les meilleures performances, mais cela consti-

tuerait une alternative à l'auto-tuner actuel. On éviterait ainsi une recherche longue et lourde en consommation des ressources de la plate-forme en cas de changement du matériel. En baissant les ressources requises pour déterminer les meilleurs candidats, il deviendrait alors possible de l'utiliser sur des systèmes embarqués.

Avec la génération automatique de spécialiseur, l'utilisation de LLVM permet un déport et une répartition plus simple de certains calculs. L'utilisation de `Kahuna` permet d'ajouter un grain plus fin dans la gestion des constantes inconnues à la compilation. Associé à une compilation en amont ou en juste-à-temps, cela autorise une réelle portabilité de `Kahuna` et un temps de génération acceptable. Il devient alors possible d'intégrer un mécanisme de spécialisation dans des réseaux de capteurs. Néanmoins, il reste à mettre en place des stratégies adéquates de gestion de l'énergie, la communication et la reprogrammation du système ayant un coût non négligeable sur ces appareils.

Bibliographie Personnelle

Conférences avec actes

1. *Scilab on a Hybrid Platform*
   ParCo 2013, MS Heterogeneous Architectures
   V. Lomüller, S. Ledru, H.P. Charles
2. *A LLVM Extension for the Generation of Low Overhead Runtime Program Specializer*
   ADAPT 2014
   V. Lomüller, H.P. Charles
3. *deGoal a Tool to Embed Dynamic Code Generators Into Applications*
   Compiler Construction 2014
   H.P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, R. Gauguey

Conférence sans acte

1. *Progressive Compilation : New Metrics and Usages*
   CPC 2013
   V. Lomüller, H.P. Charles

Chapitres de livres

1. *Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform*
   Smart Multicore Embedded Systems
   D. Couroussé, V. Lomüller, H.P. Charles
2. *Data Size and Data Type Dynamic GPU Code Generation*
   Patterns for Parallel Programming on GPUs
   H.P. Charles, V. Lomüller

Poster

1. *Maximizing GEMM Performances via Offline Heuristic Generation and Run-time Specialization*
   Advanced Computer Architecture and Compilation for High-Perfomance and Embedded Systems (ACACES 2013)
   V. Lomüller, H.P. Charles

Fait marquant

1. *Exploiter pleinement les performances hardware multicœurs par compilation dynamique*
   Un “fait marquant” CEA a été réalisé pour les résultats obtenues sur les GPU. Ce fait marquant a été intégré dans les rapports d'activités 2012 du CEA-LIST et celui de DIGITEO.

Bibliographie

[1] asfermi : An assembler for the NVIDIA Fermi Instruction Set. `https://code.google.com/p/asfermi/`, 2014.

[2] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla et Brian Gaeke : LLVA : A Low-level Virtual Instruction Set Architecture. *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman : *Compilers : Principles, Techniques and Tools (2nd Edition)*. Pearson New Internationnal Edition, 2007.

[4] Andrew W Appel : *Modern compiler implementation in ML*. Cambridge university press, 1998.

[5] ARM : *A8 Technical Reference Manual*, 2010.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams et Katherine A. Yelick : The Landscape of Parallel Computing Research : A View from Berkeley. Rapport technique UCB/EECS-2006-183, 2006.

[7] John Aycock : A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, juin 2003.

[8] Chris Bagwell, Rob Sykes et Pascal Giard : SoX - Sound eXchange - v14.4.1. `http://sox.sourceforge.net`, 2014.

[9] Vasanth Bala, Evelyn Duesterwald et Sanjeev Banerjia : Dynamo : A Transparent Dynamic Optimization System. *In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00*, 2000.

[10] Fabrice Bellard : QEMU, a Fast and Portable Dynamic Translator. *In USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[11] J. Bergstra, N. Pinto et D. Cox : Machine learning for predictive auto-tuning with boosted regression trees. *In Innovative Parallel Computing (InPar), 2012*, pages 1–9, May 2012.

[12] Sapan Bhatia, Charles Consel et Calton Pu : Remote Customization of Systems Code for Embedded Devices. *In Proceedings of the fourth ACM international conference on Embedded software - EMSOFT '04*, page 7, Jan 2004.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill et David A. Wood : The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, août 2011.

[14] K. Brifault et H-P Charles : Efficient data driven run-time code generation for multimedia applications. pages 1–7, 2004.

[15] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston et Pat Hanrahan : Brook for gpus. *In ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*, page 777, Jan 2004.

[16] Brian Carlstrom, Anwar Ghuloum et Ian Rogers : The ART runtime. Google I/O 2014, June 2014.

[17] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo et Rémy Gauguey : deGoal a Tool to Embed Dynamic Code Generators into Applications. *Compiler Construction*, 8409:107–112, Jan 2014.

[18] Henri-Pierre Charles et Victor Lomüller : *Patterns for Parallel Programming on GPUs*, chapitre Data Size and Data Type Dynamic GPU Code Generation. 2012.

[19] Henri-Pierre Charles et Khawar Sajjad : HPBCG High Performance Binary Code Generator, 2009.

[20] Albert Cohen et Erven Rohou : Processor virtualization and split compilation for heterogeneous multicore embedded systems. *Proceedings of the 47th Design Automation Conference on - DAC '10*, page 102, Jan 2010.

[21] Charles Consel, Julia L. Lawall et Anne-Françoise Le Meur : A Tour of Tempo : a Program Specializer for the C Language. *Science of Computer Programming*, 52(1-3):341–370, Jan 2004.

[22] Damien Couroussé, Victor Lomüller et Henri-Pierre Charles : Introduction to Dynamic Code Generation : An Experiment with Matrix Multiplication for the STHORM Platform. *Smart Multicore Embedded Systems*, pages 103–122, Jan 2014.

[23] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman et F. Kenneth Zadeck : Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, octobre 1991.

[24] Frédéric de Mesmay, Yevgen Voronenko et Markus Püschel : Offline Library Adaptation Using Automatically Generated Heuristics. pages 1–10, 2010.

[25] Pablo de Oliveira Castro, Eric Petit, Jean Christophe Beyler et William Jalby : ASK : Adaptive Sampling Kit for Performance Characterization. *In* Christos Kaklamanis, Theodore S. Papatheodorou et Paul G. Spirakis, éditeurs : *Euro-Par 2012 Parallel Processing - 18th International Conference*, volume 7484 de *Lecture Notes in Computer Science*, pages 89–101. Springer, 2012.

[26] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili et Nathan Clark : Ocelot : a dynamic optimization framework for

bulk-synchronous applications in heterogeneous systems. *In Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, page 353–364, New York, NY, USA, 2010. ACM.

[27] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell et Judit Planas : OmpSs : a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21:173–193, 2011-03-01 2011.

[28] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles : Micro-architectural Simulation of In-order and Out-of-order ARM Microprocessors with gem5. *In Proceedings - 2014 International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation, XIV-SAMOS*, 2014.

[29] Dawson R. Engler et Todd A. Proebsting : DCG : An Efficient, Retargetable Dynamic Code Generation System. *SIGOPS Oper. Syst. Rev.*, 28(5):263–272, novembre 1994.

[30] Christopher W. Fraser et David R. Hanson : A code generation interface for ansi c. *Softw. Pract. Exper.*, 21(9):963–988, août 1991.

[31] M. Frigo et S. G. Johnson : The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[32] Matteo Frigo : A Fast Fourier Transform Compiler. *In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[33] Grigori Fursin, Yuriy Kashnikov, AbdulWahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, ChristopherK.I. Williams et Michael O'Boyle : Milepost GCC : Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[34] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller et Bertil Folliot : VMKit : A Substrate for Managed Runtime Environments. *In Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM.

[35] Google Corporation : *Dalvik Optimization and Verification*, 2008. `https://android.googlesource.com/platform/dalvik/+/kitkat-release/docs/dexopt.html`.

[36] Thierry Goubier, Renaud Sirdey, Stéphane Louise et Vincent David : ΣC : A Programming Model and Language for Embedded Manycores. *In Algorithms and Architectures for Parallel Processing*, volume 7016 de *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2011.

[37] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers et Susan J. Eggers : DyC : An Expressive Annotation-Directed Dynamic Compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, octobre 2000.

[38] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers et Susan J. Eggers : The Benefits and Costs of DyC's Run-time Optimizations. *ACM Trans. Program. Lang. Syst.*, 22(5):932–972, septembre 2000.

[39] green500.org : Green500 list. `http://www.green500.org`.

[40] Tobias Grosser, Armin Groesslinger et Christian Lengauer : Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[41] OpenACC Working Group *et al.* : The openacc application programming interface, 2013.

[42] Sebastian Hack, Daniel Grund et Gerhard Goos : Register Allocation for Programs in SSA-Form. *In Compiler Construction*, volume 3923, pages 247–262, Jan 2006.

[43] Mary Hall, David Padua et Keshav Pingali : Compiler research : The next 50 years. *Commun. ACM*, 52(2):60–67, février 2009.

[44] Karine Heydemann, Francois Bodin et Henri-Pierre Charles : A software-only compression system for trading-offs between performance and code size. *In Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, SCOPES '05, pages 27–36, New York, NY, USA, 2005. ACM.

[45] Intel : Auto vectorization of OpenCL code with the Intel SDK for OpenCL Applications. `https://software.intel.com/en-us/articles/auto-vectorization-of-opencl-code-with-the-intel-opencl-sdk`, 2012.

[46] Minhaj Ahmad Khan, Henri-Pierre Charles et Denis Barthou : Improving Performance of Optimized Kernels Through Fast Instantiations of Templates. *Concurrency and Computation : Practice and Experience*, 21(1):59–70, Jan 2009.

[47] Khronos Group : The SPIR Specication : Standard Portable Intermediate Representation, Jan 2010.

[48] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell et David Cox : Design of the Java HotSpot ; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7 :1–7 :32, mai 2008.

[49] J. Kurzak, S. Tomov et J. Dongarra : Autotuning GEMM Kernels for the Fermi GPU. *Parallel and Distributed Systems, IEEE Transactions on*, 23(11):2045 –2057, nov. 2012.

[50] Jakub Kurzak, Piotr Luszczek, Stanimire Tomov et Jack Dongarra : Preliminary Results of Autotuning GEMM Kernels for the NVIDIA Kepler Architecture – GeForce GTX 680. Rapport technique, 2012.

[51] Chris Lattner et Vikram Adve : LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[52] Julia L. Lawall et Gilles Muller : Faster Run-time Specialized Code using Data Specialization. Rapport technique RR-3833, 1999.

[53] Mark Leone et Peter Lee : Lightweight Run-Time Code Generation. Rapport technique, Department of Computer Science, University of Melbourne, 1994.

[54] Mark Leone et Peter Lee : A Declarative Approach to Run-Time Code Generation. *In In Workshop on Compiler Support for System Software (WCSSS*, pages 8–17, 1996.

[55] Yves Lhuillier et Damien Couroussé : Embedded System Memory Allocator Optimization Using Dynamic Code Generation. *In Dynamic Compilation Everywhere, 2012*, 2012.

[56] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen et Norman P. Jouppi : The McPAT Framework for Multicore and Manycore Architectures : Simultaneously Modeling Power, Area, and Timing. *ACM Trans. Archit. Code Optim.*, 10(1):5 :1–5 :29, avril 2013.

[57] Victor Lomüller et Henri-Pierre Charles : Progressive compilation : New metrics and usages. *In 17th Workshop on Compilers for Parallel Computers, CPC 2013*, July 2013.

[58] Victor Lomüller et Henri-Pierre Charles : A LLVM Extension for the Generation of Low Overhead Runtime Program Specializer. *In Proceedings of International Workshop on Adaptive Self-tuning Computing Systems - ADAPT '14*, pages 14–16, Jan 2014.

[59] Victor Lomüller, Sylvestre Ledru et Henri-Pierre Charles : Scilab on a Hybrid Platform. *In Parallel Computing : Accelerating Computational Science and Engineering (CSE)*, Advances in Parallel Computing, pages 743–752. IOS Press, 2014.

[60] Renaud Marlet, Charles Consel et Philippe Boinot : Efficient Incremental Run-time Specialization for Free. *In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 281–292, New York, NY, USA, 1999. ACM.

[61] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan et Kevin Moule : Shader algebra. *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*, page 787, Jan 2004.

[62] Jason Merrill : GENERIC and GIMPLE : A New Tree Representation for Entire Functions. *In Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.

[63] Markus Mock, Craig Chambers et Susan J. Eggers : Calpa : A Tool for Automating Selective Dynamic Compilation. *In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 291–302, New York, NY, USA, 2000. ACM.

[64] Rajib Nath, Stanimire Tomov et Jack Dongarra : An Improved MAGMA GEMM For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, novembre 2010.

[65] Rajib Nath, Stanimire Tomov et Jack Dongarra : *BLAS for GPUs.* 2011.

[66] Nicholas Nethercote et Julian Seward : Valgrind : A framework for heavyweight dynamic binary instrumentation. *In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[67] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici et Jose Castanos : JIT Technology with C/C++ : Feedback-directed Dynamic Recompilation for Statically Compiled Languages. *ACM Trans. Archit. Code Optim.*, 10(4):59 :1–59 :25, décembre 2013.

[68] NVIDIA Corporation : CUDA LLVM Compiler. `https://developer.nvidia.com/cuda-llvm-compiler`.

[69] NVIDIA Corporation : *NVIDIA's Next Generation CUDA Compute Architecture : Fermi, Version 1.1*, 2009.

[70] NVIDIA Corporation : *NVIDIA CUDA C Programming Guide, Version 4.1*, 2011.

[71] NVIDIA Corporation : *CUDA Toolkit 4.1, CUBLAS Library*, 2012.

[72] NVIDIA Corporation : *Parallel Thread Execution ISA, Version 3.0*, 2012.

[73] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky et Markus Püschel : Spiral in Scala : Towards the Systematic Construction of Generators for Performance Libraries. *GPCE '13*, pages 125–134, Jan 2013.

[74] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler et M. Frans Kaashoek : `C and tcc : A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, Jan 1999.

[75] Markus Püschel, Franz Franchetti et Yevgen Voronenko : *Encyclopedia of Parallel Computing*, chapitre Spiral. Springer, 2011.

[76] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson et Nicholas Rizzolo : SPIRAL : Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[77] Python Software Foundation : Python website. `https://www.python.org`, 2014.

[78] J. Ross Quinlan : *C4.5 : programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[79] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues et Victor Hugo Sperle Campos : A fast and low-overhead technique to secure programs against integer overflows. *In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[80] Jason Sams et Tim Murray : High Performance Applications with RenderScript. Google I/O 2013, May 2013.

[81] Scilab Consortium : Scilab. `http://www.scilab.org`.

[82] James Stanier et Des Watson : Intermediate Representations in Imperative Compilers : A Survey. *ACM Comput. Surv.*, 45(3):26 :1–26 :27, juillet 2013.

[83] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao et Ninghui Sun : Fast implementation of dgemm on fermi gpu. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35 :1–35 :11, New York, NY, USA, 2011. ACM.

[84] The WebKit Open Source Project : Introducing the WebKit FTL JIT. `https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit`, May 2014.

[85] top500.org : Top500 supercomputer sites. `http://www.top500.org`.

[86] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, Ramakrishna Upadrasta *et al.* : GRAPHITE Two Years After : First Lessons Learned From Real-World Polyhedral Compilation. *In GCC Research Opportunities Workshop (GROW'10)*, 2010.

[87] Vasily Volkov : Better Performance at Lower Occupancy. Proceedings of the GPU Technology Conference, GTC, 2010.

[88] Vasily Volkov et James W. Demmel : Benchmarking gpus to tune dense linear algebra. *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31 :1–31 :11, Piscataway, NJ, USA, 2008. IEEE Press.

[89] Richard L. Wexelblat, éditeur. *History of Programming Languages I*. ACM, New York, NY, USA, 1981.

[90] R. Clint Whaley et Jack Dongarra : Automatically Tuned Linear Algebra Software. *In SuperComputing 1998 : High Performance Networking and Computing*, 1998.

[91] Rui Zhang, Saumya Debray et Richard T. Snodgrass : Micro-Specialization : Dynamic Code Specialization of Database Management Systems. *In Proceedings of the Tenth International Symposium on Code Generation and Optimization - CGO '12*, page 63, Jan 2012.

Résumé

La compilation est une étape indispensable dans la création d'applications performantes. Cette étape autorise l'utilisation de langages de haut niveau et indépendants de la cible tout en permettant d'obtenir de bonnes performances. Cependant, de nombreux freins empêchent les compilateurs d'optimiser au mieux les applications. Pour les compilateurs statiques, le frein majeur est la faible connaissance du contexte d'exécution, notamment sur l'architecture et les données utilisées. Cette connaissance du contexte se fait progressivement pendant le cycle de vie de l'application. Pour tenter d'utiliser au mieux les connaissances du contexte d'exécution, les compilateurs ont progressivement intégré des techniques de génération de code dynamique. Cependant ces techniques ne se focalisent que sur l'utilisation optimale du matériel et n'utilisent que très peu les données. Dans cette thèse, nous nous intéressons à l'utilisation des données dans le processus d'optimisation d'applications pour GPU Nvidia. Nous proposons une méthode utilisant différents moments pour créer des bibliothèques adaptatives capables de prendre en compte la taille des données. Ces bibliothèques peuvent alors fournir les noyaux de calcul les plus adapté au contexte. Sur l'algorithme de la GEMM, la méthode permet d'obtenir des gains pouvant atteindre 100 % tout en évitant une explosion de la taille du code. La thèse s'intéresse également aux gains et coûts de la génération de code lors de l'exécution, et ce du point de vue de la vitesse d'exécution, de l'empreinte mémoire et de la consommation énergétique. Nous proposons et étudions 2 approches de génération de code à l'exécution permettant la spécialisation de code avec un faible surcoût. Nous montrons que ces 2 approches permettent d'obtenir des gains en vitesse et en consommation comparables, voire supérieurs, à LLVM mais avec un coût moindre.

Abstract

Compilation is an essential step to create efficient applications. This step allows the use of high-level and target independent languages while maintaining good performances. However, many obstacle prevent compilers to fully optimize applications. For static compilers, the major obstacle is the poor knowledge of the execution context, particularly knowledge on the architecture and data. This knowledge is progressively known during the application life cycle. Compilers progressively integrated dynamic code generation techniques to be able to use this knowledge. However, those techniques usually focuses on improvement of hardware capabilities usage but don't take data into account. In this thesis, we investigate data usage in applications optimization process on Nvidia GPU. We present a method that uses different moments in the application life cycle to create adaptive libraries able to take into account data size. Those libraries can therefore provide more adapted kernels. With the GEMM algorithm, the method is able to provide gains up to 100 % while avoiding code size explosion. The thesis also investigate runtime code generation gains and costs from the execution speed, memory footprint and energy consumption point of view. We present and study 2 light-weight runtime code generation approaches that can specialize code. We show that those 2 approaches can obtain comparable, and even superior, gains compared to LLVM but at a lower cost.