placement sur GPU

Sommaire

L'architecture des GPU décrite dans le chapitre 1 met en évidence d'une part, une divergence vis-à-vis des architectures CPU classiques et, d'autre part, une utilisation massive du parallélisme. Par nécessité, cette divergence architecturale a engendré la création de nouveaux langages et librairies décrits dans la section 1.4. Ces derniers permettent au programmeur d'utiliser les spécificités nouvelles du GPU. Quant au parallélisme intensif, le défi a été non seulement d'adapter le parallélisme de tâches propre aux CPUs, mais aussi d'étendre l'héritage des instructions et architectures vectorielles. La possible segmentation

des espaces mémoire hôte/accélérateur engendre de plus le besoin de gérer le transfert des données utilisées par le GPU.

Au final, ces changements se traduisent par une rupture dans les processus classiques de développement des applications. Cette rupture est liée à la nécessité d'apprentissage de l'architecture particulière des GPUs, à la complexité de "penser" un programme de façon parallèle dès sa création, et à l'héritage des librairies d'algorithmes et des langages séquentiels. Ces problématiques, amplifiées par la complexité des hiérarchies multiples et du parallélisme massif des GPUs, peuvent être traitées en employant la parallélisation automatique. Paul Feautrier en a fait l'état des lieux [146] en 2002, soit avant l'ère du HPC sur GPU. Il évoquait parmi les solutions, l'utilisation du modèle polyédrique ainsi que ses limites pour le placement d'algorithmes sur architecture parallèle. Cependant, les Static COntrol Part (SCOP)s, les transformations unimodulaires d'espaces affines, les bornes de boucles statiques ou les algorithmes réguliers étaient déjà à cette époque autant de limites à l'emploi des méthodes polyédriques de compilation.

Les solutions décrites dans ce chapitre ont été développées dans le but de simplifier la transition CPU/GPU et d'améliorer le placement ainsi que les performances des algorithmes sur GPU. Ces solutions considèrent les enjeux de la parallélisation pour le placement à gros grain et de la vectorisation pour le placement à grain fin sur GPU. On retrouve dans chacune d'elles, une partie ou la totalité d'un axe méthodologique commun dont la synthèse des étapes est la suivante :

La première étape d'*analyse* consiste à transformer le code source en une représentation intermédiaire qui servira de base pour différentes analyses, transformations et optimisations. Cette représentation peut prendre différentes formes telle que l'Abstract Syntaxic Tree (AST). Les zones de code affines et à contrôle statique peuvent être aussi traduites en contraintes polyédriques qui sont utiles à l'application de transformations unimodulaires.

L'étape de *transformation*, va permettre de placer un algorithme sur le GPU. Si cette étape assure un comportement fonctionnel identique de l'algorithme, elle ne garantit cependant pas toujours l'optimalité du placement sur l'architecture. Différentes entrées peuvent alimenter cette étape. Des compilateurs tels que PPCG (2.2.3), PIPS (2.2.2) ou encore Bones (2.3.4) utilisent les représentations intermédiaires précédemment mentionnées. ToGPU (2.2.5) pour sa part utilise directement le code source sur lequel des *matchers*[1] seront appliqués pour identifier des *patterns* de codes[2] spécifiques. SkePU (2.3.1) ou OpenMPC (2.1.5) de leur côté utilisent des directives annotées par le développeur dans le code source original. Au final, le format de données retenu sera analysé pour en extraire les dépendances de données qui permettront d'assurer la légalité des transformations de code. Ces transformations adaptent ainsi les algorithmes séquentiels aux contraintes architecturales des GPUs. On distingue trois types d'approche pour cette étape. La transformation au moyen de directives, décrite dans la section 2.1, la transformation automatique de code, détaillée dans la section 2.2, et l'utilisation de squelettes développée dans la section 2.3.

L'étape d'*optimisation* du placement permet d'améliorer l'adéquation du problème algorithmique traité avec les contraintes architecturales du GPU ciblé. L'objectif est ici de raffiner les transformations de code pour chercher à atteindre la *peak performance* de l'architecture retenue, en fonction de ses spécifications. Il est à noter que plusieurs des solutions de transformation de code effectuent aussi de l'optimisation de placement. Dans le cadre de la section 2.4, nous présenterons des solutions effectuant exclusivement de l'optimisation de placement sur un code déjà transformé pour GPU.

Enfin, l'étape de *génération de code* consiste à générer un code source compilable à

1. Formulation permettant d'identifier par correspondance un modèle
2. Struture de code notable

partir de la représentation intermédiaire transformée. La problématique de cette dernière étape repose sur la détermination d'une traduction optimale parmi l'ensemble des traductions possibles, en fonction des contraintes architecturales du GPU. Dans le cadre des accélérateurs tels que les GPUs, cette étape de génération de code va effectuer l'*outlining* de code vers un *kernel*, ajouter les instructions de mise en œuvre des *kernels* créés et aussi générer les transferts de données entre accélérateurs et hôte dans le cas d'espaces mémoire disjoints.

Pour conclure, nous noterons qu'en parallèle des trois approches présentées dans ce chapitre, il en existe une quatrième fondée sur les DSL et les API. Certaines solutions [88, 35, 32, 34, 21, 100] permettent dans leur domaine spécifique de décrire la résolution d'un problème selon une sémantique donnée. La problématique de transformation est alors déportée vers la définition d'une sémantique appropriée, non seulement pour la résolution du problème concerné, mais aussi pour intégrer les caractéristiques de l'architecture ciblée. Dans leur forme la plus basique, les squelettes peuvent être assimilés dans certains cas à une API. Cette quatrième approche ne sera volontairement pas développée dans cet état de l'art, car, dans le cadre de cette thèse, nous nous sommes concentrés sur les transformations automatisées de codes séquentiels.

2.1 Transformation par annotation de directives

Les solutions présentées dans cette section permettent de transformer un code source en programme parallèle grâce à des directives placées dans le code source original. L'utilisation d'annotations permet de conserver la forme originale du code séquentiel. Dans le cas où une architecture autre que le GPU viendrait à être utilisée, le code source original resterait ainsi compilable, en ignorant les directives ajoutées. Cela représente un plus indéniable en terme de portabilité.

Au final, ces directives correspondent à un langage haut niveau. Celui-ci permet de demander des transformations à des compilateurs intégrant un interpréteur pour le langage de directives utilisé. Deux approches se distinguent au sujet des directives dans cet état de l'art. Certaines solutions comme *OpenMP C to Cuda* ou *OpenMPC* décrites dans les sections 2.1.4 et 2.1.5, ont choisi d'adapter aux GPUs les spécifications du standard Open Multi-Processing (OpenMP) présenté en section 2.1.3. Le sujet est loin d'être trivial car OpenMP a été initialement défini pour répondre aux problématiques des CPUs. D'autres en revanche, comme *Mint* ou *hiCUDA* décrits dans les sections 2.1.6 et 2.1.2, proposent leur propre modèle de directives, ainsi que l'interpréteur associé. Au final, OpenACC, décrit dans la section 2.1.8, a permis de converger vers un nouveau standard de directives mais pour accélérateurs cette fois. Comme pour OpenMP, le consortium d'OpenACC fournit une spécification de langage de directives. Différentes implémentations d'interpréteurs sont disponibles comme celle de *PGI accelerator* en section 2.1.9.

2.1.1 HMPP

Hybrid Multicore Parallel Programming (HMPP) [50, 54] est un compilateur commercial qui a été développé par CAPS entreprise. Il permet de porter sur GPU du code C ou Fortran annoté au moyen de directives de type *pragma hmpp*. Ces directives permettent de gérer les transferts de données, ainsi que des fonctions appelées *codelets* pour un accélérateur cible. Dans le cadre des GPUs, ces fonctions correspondent aux *kernels*. HMPP gère l'utilisation simultanée de plusieurs accélérateurs de calculs au sein d'une architecture globale hétérogène. Pour cela, il permet au programmeur de découper un programme

en plusieurs sous-ensembles, chacun d'entre-eux étant adapté à une architecture cible. La distribution des calculs sur les différents GPU est réalisée de manière dynamique par le *runtime* de HMPP. Dans le cas où aucun GPU ne serait disponible, l'implémentation *native* est exécutée sur le CPU.

HMPP ne semble plus actif en 2018. Un standard portant le nom d'*Open HMPP* a été dérivé de la version 2.3 de HMPP. L'entreprise CAPS ayant été impliquée dans le développement du standard OpenACC avant de faire faillite, nous considérerons OpenACC présenté dans le chapitre 2.1.8 comme l'évolution de HMPP.

2.1.2 hiCUDA

hiCUDA [71, 72, 73] est un compilateur source-à-source fondé sur Open64. Ce compilateur utilise en entrée du code séquentiel C ou C++ annoté au moyen d'un langage de haut niveau. Les zones de code ainsi délimitées par des directives *'#pragma hicuda'* sont transformées de manière inter-procédurale en *kernels* CUDA afin d'être exécutés sur GPU. La phase d'analyse, et notamment l'analyse des dépendances, reste manuelle. Le placement des directives dans le code source original se fait au moyen d'un outil d'analyse ou par un programmeur ayant évalué le contenu du code source. Seules les analyses des régions de tableaux et des flots de données sont automatisées par les modules afférents d'Open64. De ce fait, hiCUDA permet de calculer les espaces mémoires utilisés par chaque *kernel*. Au final, cette donnée permet de ne transférer que les espaces ou sous-espaces mémoires nécessaires sur le GPU. Enfin, hiCUDA est complémentaire à CUDA-Lite présenté dans le chapitre 2.4.1. Celui-ci permet d'améliorer l'utilisation des *shared memory* dans le GPU à partir de code CUDA.

Dans le cadre de hiCUDA, leur méthodologie de portage d'algorithmes séquentiels sur GPU [72] est composée de cinq étapes :

1. l'*outlining* permettant de créer le *kernel* encapsulant le code source annoté,
2. le *tiling* pour définir le placement des *threads* sur les niveaux hiérarchiques du GPU,
3. la génération des communications entre le CPU et le GPU,
4. l'optimisation de la bande passante d'accès aux données par sélection des mémoires du GPU,
5. l'optimisation du code source intégré dans le *kernel*.

Les directives développées par Han et Abdelrahman [71] sont réparties dans deux catégories : le modèle de calculs et le modèle de données. Le modèle de calculs fournit quatre actions :

— la création et l'encapsulation de code séquentiel dans un *kernel*,
— le partitionnement cyclique ou en blocs des boucles sur les *blocks* de *threads* du GPU ou uniquement une distribution cyclique sur les *threads* du GPU pour des soucis de coalescence des accès aux données,
— le placement de code sur un unique *thread* pour chaque *block*,
— la synchronisation des *threads* au moyen de barrières.

Le modèle de données fournit trois actions permettant d'utiliser la mémoire constante, la mémoire globale et la *shared memory*. La gestion des tableaux de taille dynamique est aussi permise par une directive associée dans le modèle de données. Enfin, hiCUDA ne gère pas l'utilisation de la *texture memory* sur les GPUs.

2.1.3 OpenMP

OpenMP est aujourd'hui devenu un standard pour la parallélisation de code sur CPU à partir de code C, C++ ou Fortran. Le consortium *OpenMP Architecture Review Board* responsable d'OpenMP, publie régulièrement les spécifications d'un langage haut niveau définissant des directives. Le placement dans le code source original de ces directives peut être effectuée manuellement par le développeur ou au moyen d'un paralléliseur automatique de code tel que ROSE [140, 141] ou Pluto [28, 29]. Des compilateurs tels que GCC, Intel ISL ou encore LLVM [86] intègrent un interpréteur de directives OpenMP dont le rôle est d'appliquer les transformations de code adéquates au placement du code original sur architecture parallèle. Le modèle OpenMP *fork-join*, caractérisé par un *thread* maître dirigeant un *pool de threads worker* est comparable au fonctionnement du GPU. Ainsi, la révision 4.0 des spécifications d'OpenMP publiée en 2013 apporte de nouvelles directives permettant aux compilateurs de déporter des portions de code sur différents types d'accélérateurs dont les GPUs [99]. Enfin, on retiendra qu'OpenMP est principalement basé sur le parallélisme de boucles comme pour les GPUs.

2.1.4 "OpenMP C to CUDA"

OpenMP C to CUDA [110, 109] est un compilateur source-à-source permettant de transformer automatiquement un code annoté avec des directives OpenMP en code CUDA. Le compilateur OMPi [49] a servi de base pour son implémentation et les *Scanner* et *Parser* de code d'OMPi ont été étendus pour prendre en compte les directives Cuda.

Concernant les transformations de code, la directive *omp parallel* est transformée en générant dans l'ordre : les allocations mémoires sur GPU, les transferts des données du CPU vers les zones mémoire GPU précédemment allouées, l'appel du ou des *kernels* concernés, les transferts de données du GPU vers le CPU et enfin les désallocations mémoire sur GPU. Les boucles annotées avec la directive *omp for* sont transformées en un *kernel* CUDA en utilisant les techniques d'*outlining* détaillées à la suite. OpenMP C to CUDA est cependant limité à un unique niveau de boucle *omp for*. L'analyse des bornes et du pas d'itération de la boucle *for* concernée permet de définir la quantité de *threads* CUDA à allouer. Le corps de la boucle *for* est modifié avant d'être transféré dans le *kernel* CUDA. Les indices de boucles sont ainsi recalculés pour s'adapter à l'architecture en *blocks/threads* du GPU et les variables sont modifiées en fonction de leur statut OpenMP, *shared* ou *private*.

Les variables scalaires et les tableaux de type *shared* sont transférés dans la mémoire globale du GPU avant le lancement du *kernel* et sont récupérés ensuite par le CPU. Pour chaque variable scalaire *private*, un tableau de la taille du nombre de *threads* est alloué dans la mémoire globale du GPU. De ce fait chaque *thread* aura sa propre variable privée. Si cette technique permet de réduire l'occupation des registres, les performances devraient cependant être pénalisées du fait de l'utilisation systématique de la mémoire globale du GPU. Enfin, il est à noter que seuls les tableaux à bornes statiques sont pris en compte. À notre connaissance, aucune optimisation visant à réduire la redondance des transferts mémoire n'est utilisée. Pour les portions de code CPU, afin de conserver une compatibilité avec le standard C99, la configuration et le lancement des *kernels* CUDA utilisent l'implémentation *CUDA driver API* [125] du langage CUDA détaillée dans la section 1.4.3.

2.1.5 OpenMPC

Comme OpenMP C to CUDA détaillé au chapitre 2.1.4, OpenMPC [90, 89, 91] est un compilateur source-à-source permettant de transformer automatiquement un code annoté

avec l'API OpenMP en code CUDA. Il aborde notamment les problématiques liées à l'utilisation de directives spécifiquement définies pour CPU afin de générer du code GPU. Le challenge repose ici sur la gestion des divergences architecturales entre CPU et GPU. La méthodologie de traduction et d'optimisation source-à-source se décompose en étapes successives développées en utilisant le framework de compilation Cetus [87]. Le code source est analysé par le *Cetus Parser* afin de générer une représentation intermédiaire au format Cetus IR qui servira de base de travail pour les transformations. OpenMPC a la particularité de réutiliser les directives standards OpenMP. Ainsi, les directives OpenMP *parallel* sont interprétées comme des *kernels* CUDA potentiels et chaque itération de directives *for* ou *sections* est distribuée sur les *threads* du GPU. Les directives de synchronisation donnent lieu à une séparation du *kernel* concerné en deux *sous-kernels* afin de respecter les dépendances de données. Des directives supplémentaires peuvent aussi être fournies par l'utilisateur au moyen d'un fichier annexe, ce qui permet de ne pas modifier les directives OpenMP existantes. Celles-ci seront appliquées pour chacune des régions de code correspondantes. De même, un fichier définissant des variables d'environnement peut être utilisé pour définir les spécifications du GPU ciblé lors des phases d'optimisation et de transformation de code.

OpenMPC permet d'améliorer la coalescence des données entre *threads* dans un *block* au moyen de deux transformations de code. La première, *parallel loop-swap*, permute deux boucles parallèles au sein d'un même nid lorsque celles-ci sont régulières et mal ordonnancées afin d'essayer de se ramener à un pas unitaire d'accès au données. Dans le même but, la seconde, *loop-collapsing*, compresse deux boucles de profondeurs différentes lorsque les fonctions d'accès aux données sont au contraire irrégulières. Ce cas provient généralement de l'utilisation de code à contrôle dynamique ou de tableaux à accès indirect. Les fonctions d'accès ne peuvent alors être résolues lors de la compilation.

Une analyse interprocédurale de flot de données est utilisée pour gérer les transferts mémoires entre CPU et GPU [89]. Celle-ci analyse de plus la cohérence entre les mémoires des deux architectures et permet ainsi de réduire la quantité de données envoyée à chaque transfert mais aussi de supprimer les transferts mémoire inutiles entre deux *kernels* exécutés.

Afin de converger vers une solution de placement optimale, deux algorithmes ont été définis. Le premier, *Search Space Pruning*, est un algorithme de simplification d'espace de recherche en fonction des variables d'environnement et des directives OpenMPC supplémentaires fournies par l'utilisateur. Le second, *Tuning Configuration Generation*, permet de parcourir l'espace de recherche et de générer pour chaque point, le paramétrage algorithmique afférent.

Avant l'étape finale de génération de code CUDA, plusieurs optimisations sont effectuées par OpenMPC. Si l'espace de stockage concerné le permet, les données réutilisées dans un *kernel* sont placées en *constant memory* dans l'hypothèse où celles-ci sont uniquement lues. Dans le cas contraire, la *shared memory* est utilisée pour stocker ces données. Enfin, le parcours d'une matrice, dans le sens opposé à celui du stockage de ses éléments, est optimisé en stockant le résultat de la transposée de cette même matrice, dans la *shared memory*. Cette transformation permet de recréer des accès coalescents entre *threads* au moyen d'une amélioration de la localité spatiale.

2.1.6 Mint

Mint [154] est un compilateur source-à-source qui traite des directives de type *pragma C*. Celles-ci permettent d'orienter le compilateur dans la transformation d'un code source séquentiel en langage C vers un code CUDA pour GPU. Mint est spécialisé pour les algo-

rithmes de calculs de stencils jusqu'à trois dimensions. Cette approche spécifique permet de simplifier et surtout d'améliorer la qualité du placement sur GPU. Son modèle de programmation est fortement inspiré d'OpenMP, présenté dans la section 2.1.3, et est découpé selon 2 axes. Le premier concerne la parallélisation et le placement de boucles et le second traite des transferts entre mémoires. Dans le cadre de ce modèle, cinq pragmas ont ainsi été définis dont quatre sont hérités d'OpenMP :

mint parallel pour la délimitation de régions de code parallèles,

mint for pour la parallélisation de nids de boucles,

mint barrier pour la synchronisation des *threads* parallélisés,

mint single pour gérer les portions de code séquentiel au sein de régions parallèles et

mint copy pour les transferts mémoires.

Mint utilise le framework de compilation ROSE [141, 140] qui permet de générer et de manipuler des arbres syntaxiques abstraits. La méthodologie de placement débute ainsi par l'utilisation du parseur de code de ROSE. Celui-ci prend en entrée un code source en langage C annoté au moyen des pragmas Mint définis précédemment. Ceux-ci vont ensuite être analysés et reconnus par les *handlers* fournis par Mint qui alimenteront l'étape de transformation du code séquentiel en programme CUDA. Cette étape importante est décomposable en cinq sous-étapes :

1. Les tailles des grilles de calculs et des *blocks* de *threads* sont tout d'abord déterminées à partir des spécifications des nids de boucles.
2. L'*outlining* visant à transformer les nids de boucles en *kernels* est effectué en parallèle de la précédente étape.
3. La segmentation des variables en fonction de leur portée est ensuite effectuée. Cette étape différencie ainsi les variables locales, les paramètres de *kernel* et les vecteurs de données ayant une portée globale dans l'application. Dans ce dernier cas, la génération des transferts mémoires est nécessaire.
4. Une corrélation est ensuite effectuée entre les copies mémoires demandées au moyen du pragma *Mint copy* et les vecteurs de données.
5. Enfin, le module de placement de *threads* génère les identifiants de *threads* à partir des itérations de boucles concernées.

L'étape suivante dans la méthodologie associée à Mint est une étape optionnelle d'optimisation. Un analyseur identifie la forme du *stencil* étudié et en tire son empreinte mémoire. Un optimiseur mémoire permet ensuite de transformer les communications de données en mémoire globale vers la *shared memory* ou mieux encore, vers les registres. Un aggrégateur de boucles déroule alors les boucles de plus faible granularité selon un paramètre défini par l'utilisateur, le *chunk size*, afin d'améliorer la réutilisation de données ainsi que la coalescence des accès mémoire au sein d'un même *thread*. La dernière étape enfin consiste à générer au moyen de Rose le code hôte auquel vient s'ajouter le code CUDA dérivé.

2.1.7 GPSME

GPSME [163] est un ensemble d'outils regroupés dans le framework de compilation ROSE. GPSME est fondé sur une version modifiée de Mint [154], à laquelle ont été ajoutées quelques extensions supplémentaires, telles que l'analyse de code C++ ou la gestion des codes sources fragmentés en fichiers multiples. La méthodologie de portage sur GPU dérivé de GPSME dépend donc de celle de Mint. Celle-ci ayant été décrite dans la section 2.1.6, nous ne la détaillerons pas ici. GPSME permet au final la génération de code CUDA ou

OpenCL à partir de code C ou C++. Le manque d'information et d'articles sur le sujet ainsi que la disparition du site internet du projet [3] laissent à penser que GPSME n'est plus actif.

2.1.8 OpenACC

Open ACCelerators (OpenACC) est un nouveau standard pour la parallélisation de programmes au moyen de directives. Celui-ci cible la catégorie architecturale des accélérateurs dont le GPU fait partie. De manière globale, son modèle est très similaire à celui d'OpenMP. Un consortium composé de Cray, PGI, Caps et Nvidia établit les spécifications d'un langage haut niveau permettant de manipuler des directives par annotation dans le code source original. Les langages supportés sont C, C++ et Fortran. Différents compilateurs ont intégré un traducteur de directives OpenACC pour la parallélisation de code. Parmi toutes les implémentations existantes, nous pouvons citer : le compilateur PGI Accelerator décrit dans la section 2.1.9, le compilateur CRAY, Open ARC [92], OpenUH [95, 152, 151], RoseACC ou encore GCC 7.

2.1.9 PGI Accelerator

PGI Accelerator [11, 166] est un compilateur commercial racheté par Nvidia en 2013. Il est basé sur un modèle de directives par pragmas comparable à OpenMP et incorpore un traducteur supportant le modèle de directives défini par OpenACC, présenté dans la section 2.1.8. Ce compilateur source-à-source permet de transformer du code C/C++ ou Fortran annoté en code CUDA, à partir d'analyses statiques orientées par les directives insérées dans le code source. Il est capable de calculer, par l'intermédiaire de *PGI Profiler*, quelques métriques telles que le taux d'occupation des *threads* dans le GPU, la quantification des accès mémoire ou encore le nombre de registres utilisés. Les directives interprétées par PGI Accelerator sont réparties en trois catégories :

les régions de données , qui permettent de transférer des données entre l'hôte et l'accélérateur en dehors des régions de calculs,

les régions de calculs , qui délimitent une portion de code contenant des boucles à placer sur l'accélérateur et

les directives de boucles , qui permettent de placer manuellement et à grain très fin les itérations de boucles sur l'accélérateur en agissant notamment sur les paramètres du tiling.

Le compilateur PGI est toujours d'actualité. Pourtant il n'existe pas beaucoup de publications à son sujet, notamment sur la méthodologie de portage à utiliser. Un article de 2010 de Wolfe [166] nous donne cependant quelques informations sur la fonction de placement. Son but premier est de distribuer les boucles sur les différents niveaux hiérarchiques du GPU. PGI accelerator peut ainsi :

— réordonnancer les boucles séquentielles pour maximiser la réutilisation de données et ainsi réduire la quantité de communications,
— modifier la taille et la forme des tuiles pour maximiser le parallélisme,
— détecter les données à placer en *shared memory* pour améliorer la bande passante des accès mémoire ou encore
— modifier l'affectation des index de *threads* pour améliorer la coalescence des accès mémoire.

3. http ://www.gp-sme.eu

Au final, le choix global des transformations à appliquer est effectué en utilisant un arbre de décision. Celui-ci considère par ordre décroissant d'importance :

1. la quantité minimale de mouvement des données,
2. la régularité des accès mémoire avec le *stride* le plus faible,
3. la taille maximale des *blocks* de *threads*,
4. la quantité maximale de *blocks* dans une grille de calcul,
5. l'empreinte mémoire la plus faible et pour finir,
6. la génération de code la plus simple à effectuer.

Le déroulage de boucles pour modifier la granularité des *threads* est cependant traité manuellement par l'opérateur.

2.2 Transformation automatique de code

Les compilateurs à parallélisation automatique de code permettent de transformer un code source séquentiel en code parallèle avec une intervention humaine minimale. Ils emploient ainsi un analyseur de code qui va permettre de générer une représentation intermédiaire pouvant prendre différentes formes dont celle d'un AST. *C-to-CUDA*, *PPCG*, PIPS et R-Stream exploitent en complément une abstraction polyédrique pour leurs transformations de code. *Togpu* en section 2.2.5 est cependant une exception et travaille sur le code source directement au moyen de *matchers*.

PIPS se différencie en étant capable de travailler sur des programmes entiers grâce à ses analyses inter-procédurales alors que les autres solutions demeurent intra-procédurales.

2.2.1 C-to-CUDA

Baskaran *et al.* [23] ont développé un système de transformation automatique de code C séquentiel en code CUDA intitulé *C-to-CUDA*. Leur recherche porte exclusivement sur des programmes possédant des nids de boucles avec des espaces d'itérations et des fonctions d'accès aux éléments de tableaux affines. Cette restriction est justifiée par l'utilisation du modèle polyédrique. En effet, C-to-CUDA utilise Pluto [28, 29] pour l'analyse des dépendances, le calcul des ordonnancements et l'application de transformations affines. Le *loop skewing* est employé pour générer des boucles parallèles. Le générateur de code polyédrique Chunky LOOp Generator (CLOOG) [24] est utilisé à la fin du processus pour la génération de code comportant des tuiles multi-niveaux adaptées aux architectures GPU et exploitant les informations polyédriques. Les *kernels* CUDA ainsi générés doivent cependant être placés manuellement dans le code original. Il reste notamment à générer les communications CPU/GPU ainsi que les appels et le paramétrage des différents *kernels*.

2.2.2 PIPS et Par4All

Programming Integrated Parallel System (PIPS) est un *framework* de compilation source-à-source effectuant des analyses et transformations de code. Il permet par exemple d'appliquer de la parallélisation automatique ou de la vectorisation dans le but de faciliter l'utilisation des architectures parallèles. PIPS accepte en entrée du code séquentiel en langage Fortran ou C et permet de générer en plus un code cible OpenCL, OpenMP, Message Passing Interface (MPI) ou encore CUDA. Il utilise une représentation polyédrique pour exprimer l'ensemble des informations collectées sur les structures du code source, l'analyse des dépendances ainsi que les analyses de régions de tableaux convexes [36]. Contrairement

à beaucoup d'autres outils, les analyses ont la particularité d'être interprocédurales. PIPS est aussi capable de générer et d'optimiser automatiquement les transferts de données [18] nécessaires au maintien de la cohérence mémoire pour les architectures hétérogènes telle qu'un CPU couplé à un GPU.

Amini *et al.* [17, 16] ont travaillé sur la génération de code OpenCL et CUDA pour GPU dans le cadre du projet Par4all [38, 19]. Leur approche commence par une phase de recherche des boucles au sein du programme, en retenant pour chaque nid de boucles, la boucle la plus interne. Si celle-ci est détectée comme étant parallèle, son corps est encapsulé dans un *kernel*, en utilisant le principe d'*outlining*. Pour aider la prise de décision quant-au placement d'une partie des calculs du nid de boucles sur GPU, deux analyses sont effectuées. La première correspond à l'estimation de la complexité du nid de boucles selon un modèle polynomial de temps d'exécution. La seconde est l'utilisation des régions convexes de tableaux afin d'en déduire une estimation polynomiale de l'empreinte mémoire des boucles portées sur GPU. Cette quantité est fortement corrélée à la quantité de données devant être transférée entre le CPU et le GPU. À la suite de ces deux analyses, les nids de boucles dont le temps de calcul est supérieur au temps de transfert des données sont retenus. Les fonctions de transfert des données et d'appel de *kernels* sont enfin générées et l'entête de la boucle parallèle de plus faible profondeur est remplacée par un appel à la fonction *outlinée* dont le contenu sera placé sur GPU. Dans le cas où l'analyse des régions de tableaux retourne une empreinte mémoire de nid de boucles trop élevée pour tenir dans la mémoire du GPU, un découpage de l'espace d'itérations au moyen d'un *tiling* est effectué. L'appel au *kernel* se fera en plusieurs étapes, chacune ayant une empreinte mémoire compatible avec la mémoire du GPU.

Au final, Par4all est un paralléliseur et un optimiseur automatique de code source séquentiel. Il a été développé par la société HPC Project, qui a été intégrée dans la société Silkan. Par4all a reposé dès son lancement sur le compilateur PIPS pour les phases d'analyse et de transformations. Il est ainsi capable de générer du code utilisant les API OpenMP, CUDA ou OpenCL. Cependant son interface ouverte et open source permet de modifier les outils utilisés. Ainsi, par exemple, l'optimiseur polyédrique ou encore l'extracteur de *features* pourraient être étendus par des contributions.

2.2.3 PPCG

Polyhedral Parallel Code Generator (PPCG) [160, 159] est un compilateur source-à-source utilisant les techniques polyédriques pour générer dans le cadre des GPUs, du code OpenCL ou CUDA. Un ensemble de transformations affines sont utilisées pour générer du code gérant le parallélisme à niveaux multiples ainsi que les différentes hiérarchies mémoires disponibles sur les GPUs. En conséquence du choix d'utilisation d'un modèle polyédrique, les indices de boucles et les bornes de boucles doivent être affines et le code doit être à contrôle statique. De plus, PPCG n'a pas de portée inter-procédurale dans le cadre de ses analyses. La gestion des tailles des tuiles est laissée à discrétion de l'utilisateur et l'*unrolling* des boucles n'est pas supporté. En revanche, le *tiling* à niveaux multiples est bien géré. Les nids de boucles imparfaitement imbriquées sont résolus par un découpage en plusieurs *kernels* GPU. Au final, toutes les boucles parallèles sont placées sur GPU. Seuls les boucles externes et séquentielles ainsi que le code de contrôle du GPU généré est maintenu sur l'hôte. Il n'existe donc pas de critère d'évaluation ou de prédiction des temps d'exécution permettant de rejeter un code placé sur GPU moins performant que sa version originale sur CPU. Enfin, la sélection des nids de boucles à paralléliser se fait manuellement au moyen de directives.

La méthodologie de PPCG se décompose en cinq étapes :

1. La première étape consiste à extraire du code source original une modélisation polyédrique représentant le domaine d'itération des boucles, les relations d'accès aux données ainsi que l'ordre d'exécution des instructions. L'outil Polyhedral Extraction Tool (PET) [158] est utilisé dans ce but.
2. Dans un second temps, une analyse des dépendances est effectuée en utilisant l'Integer Set Library (ISL) [156] sur le modèle polyédrique précédemment extrait.
3. À partir des dépendances identifiées, le code est à présent réordonnancé selon les critères suivants :
   — Une recherche des boucles parallèles est effectuée.
   — Une succession de transformations par tuilage est effectuée sur les nids de boucles compatibles afin d'exploiter les niveaux hiérarchiques multiples des unités de calcul et des mémoires du GPU. Un dérivé de l'algorithme de Pluto [28, 29] est utilisé pour cela avec ISL.
4. Les données sont ensuite réparties dans les différentes mémoires du GPU.
   — Les transferts mémoires entre la mémoire globale du GPU et la mémoire du CPU sont identifiés.
   — Les espaces de données pouvant être scalarisés sont transférés de la mémoire globale vers les registres.
   — De même, les données réutilisées dans plusieurs *kernels* sont déplacées de la mémoire globale vers les *shared memory*.
5. Enfin, ISL est utilisé pour générer le code GPU et le code CPU, ce que CLOOG [24] seul ne peut pas faire.

2.2.4 R-Stream

R-Stream [103] est un compilateur et optimiseur polyédrique commercial permettant de transformer automatiquement du code C ou Fortran en code OpenMP, OpenCL ou encore CUDA. Le portage sur GPU en particulier a été étudié par Leung *et al.* [94]. R-Stream autorise l'adressage de GPU multiples et est capable de générer automatiquement les communications mémoires ainsi que le code nécessaire à la mise en œuvre des *kernels* GPU générés.

La première étape de R-Stream consiste à parser le code source ciblé afin d'en extraire une représentation intermédiaire. Plusieurs types de boucles ainsi que les accès mémoire par pointeur ou tableau sont reconnus lors de cette étape. Le code ne doit cependant pas avoir été déjà optimisé. Ainsi, la réutilisation de variables ou encore les accès multi-dimensionnels ayant été linéarisés sont cités comme des exemples pouvant interférer avec les méthodes de portage et d'optimisation de R-Stream.

À la suite de la génération de la représentation intermédiaire, une première phase d'optimisation scalaire a lieu. C'est notamment durant cette étape que les appels de fonctions au sein de nid de boucles sont *inlinées*.

Dans un second temps, les espaces d'itération, les fonctions d'accès aux tableaux ainsi que les dépendances provenant de la représentation intermédiaire vont être extraits dans un modèle polyédrique. Ce dernier va servir à rechercher une solution optimale prenant en compte le parallélisme, la localité des données et la contiguïté des accès mémoires. Une version modifiée de Pluto [28, 29] est utilisée pour la manipulation de ce modèle. L'ajout de R-Stream réside dans la recherche d'un bon compromis entre la fusion de boucle, le parallélisme et la contiguïté des accès mémoire. Concernant le placement, le parallélisme à gros grain va venir alimenter les *blocks* ainsi que les *threads* du GPU, tandis que le parallélisme à grain fin est utilisé au sein même des *threads* à des fins de vectorisation ce

qui améliore la contiguïté des données [155]. La modification de l'agencement des données en mémoire est aussi utilisée pour permettre d'améliorer la vectorisation et la coalescence des accès aux données. Les espaces d'itérations sont ensuite transformés par tuilage afin de s'adapter à l'architecture à hiérarchies multiples des GPUs. Les caractéristiques architecturales du GPU ciblé sont spécifiées dans un fichier XML joint. La taille des tuiles est alors calculée pour atteindre un bon équilibre entre calculs et communications et respecter la contrainte que l'empreinte mémoire des tuiles tienne dans la *shared memory*. Afin de réduire les temps d'accès mémoire, les données utilisées plusieurs fois au sein d'un *block* de *threads* sont déplacées de la mémoire globale à la *shared memory*. Les instructions de synchronisation sont enfin placées afin de respecter les dépendances et ainsi la fonctionnalité du code.

Une seconde étape d'optimisation scalaire est effectuée afin de dérouler les boucles de plus faible granularité. Cette action permet de réduire la pression exercée par les entêtes de boucles sur les *dispatchers* d'instructions du GPU. Concrètement cela se traduit par une amélioration globale des performances liée à une meilleure alimentation des pipelines d'instructions.

Enfin, la dernière étape permet de générer les communications CPU/GPU, le code des *kernels*, ainsi que le code de contrôle pour l'utilisation du GPU.

Au final, nous noterons que cette méthodologie présente une approche en une passe unique de transformations décomposée en plusieurs étapes. Plusieurs articles ont été publiés au sujet de R-Stream en général. Cependant, le compilateur de *Reservoir Lab* n'est pas librement accessible.

2.2.5 Togpu

Marangoni et Wischgoll ont développé un outil de transformation source-à-source intitulé *Togpu* [98]. Celui-ci a été conçu pour gérer la transformation automatique d'un code C++ en CUDA en s'appuyant sur la *libTooling* de *Clang* [85] dans *LLVM* [86]. Les auteurs n'apportent pas de contribution particulière quant à la méthode de parallélisation automatique employée et s'appuient sur l'algorithme de vectorisation déjà présent dans *Clang*. Afin de gérer les transformations de codes à appliquer, ils utilisent un pipeline de transformations configuré manuellement en fonction de l'algorithme ciblé. Les transformations ordonnancées séquentiellement sont appliquées en fonction de *matcher* sur le code source. Ceux-ci sont dérivés des *AST Matchers* de *Clang* qui sont chargés de rechercher et de valider certains patterns au sein du code source. Si les *matchers* retournent des résultats, ils déclenchent les transformations correspondantes dans le pipeline.

Togpu est un outil de placement sur GPU Nvidia. Il n'effectue pas d'optimisation ni de spécialisation du code source. Pour le moment, il a été évalué avec des algorithmes de traitement d'images et est limité à certains algorithmes sélectionnés par les auteurs [98] uniquement. De plus, le code source en entrée doit avoir des tableaux mémoire de taille fixe ainsi que des boucles *for* exclusivement. Enfin la portée d'application de l'outil demeure intra-procédurale.

2.3 Squelettes algorithmiques

Les squelettes algorithmiques sont des patterns classiques de manipulation de données que l'on retrouve dans plusieurs implémentations algorithmiques. L'idée est qu'une personne experte conçoive une librairie de squelettes optimisés pour une architecture donnée. On retrouve ainsi pour le GPU des implémentations de squelettes pour OpenCL et CUDA.

SkelCL décrit en section 2.3.2 fournit ainsi une implémentation de squelettes en OpenCL. Certaines librairies peuvent aussi être spécialisées pour un domaine d'applications donné. Bones, détaillé en section 2.3.4, a ainsi été étudié pour répondre aux besoins courants du traitement d'images.

La complexité des squelettes réside dans la recherche d'un bon compromis entre une librairie trop spécialisée qui fournit un très grand nombre de squelettes aux performances optimales et une librairie trop généraliste qui propose un nombre de squelettes réduit, mais qui facilite ainsi la programmation de l'algorithme. À titre d'exemple, Thurst, détaillé en section 2.3.3, mise sur une bibliothèque intégrant une grande variété de squelettes, minimisant ainsi la programmation des algorithmes mais complexifiant le choix du squelette approprié. À l'opposé SkePU, détaillé en section 2.3.1, fournit un nombre réduit de squelettes, augmentant ainsi l'implication de l'utilisateur pour le développement des algorithmes, mais simplifiant la mise en œuvre des squelettes. Dans tous les cas, il est à noter que la résolution d'un problème algorithmique passe par l'identification manuelle du squelette adapté. Enfin, l'encapsulation de code au sein de squelettes a globalement un effet de "cloche de verre" sur les algorithmes intégrés. Ce point soulève un problème d'optimisation entre squelettes notamment pour améliorer la localité temporelle des données ou pour améliorer le taux d'occupation des architectures. Bones utilise cependant une phase d'analyse permettant de détecter automatiquement, lors de la compilation, le squelette adapté. De plus, il permet au moyen d'un modèle polyédrique d'effectuer une phase d'optimisation entre squelettes. Bones utilise ainsi les techniques propres aux transformateurs automatiques de code pour contourner les problématiques des squelettes (voir section 2.3.4).

2.3.1 SkePU/SkePU2

SkePU [145, 53] est une librairie de squelettes implémentés en CUDA, OpenCL et OpenMP. Elle est capable de gérer les architectures multi-cœurs et notamment un à plusieurs GPUs. Inspirée de BlockLib [14], dont elle partage la même structure de squelettes algorithmiques, SkePU se distingue en étant adaptée au langage C++ et en ciblant d'autres architectures comme le GPU.

SkePU utilise des *smart containers* [45] pour la gestion des données en mémoire. Ceux-ci encapsulent et interprètent les données de façon mono-dimensionnelle au moyen de vecteurs ou bi-dimensionnelle grâce aux matrices. Les vecteurs sont dérivés de la classe *vector* de la Standard Template Library (STL) et partage la même interface pour une meilleure inter-opérabilité. Les *smart containers* apportent une gestion intelligente des mémoires hétérogènes. Les données sont ainsi maintenues dans la mémoire de l'architecture hôte, le CPU et des fragments de vecteurs ou matrices correspondant à l'espace de données à calculer sont transférés vers le GPU. Dans le cas où plusieurs GPUs sont utilisés, les vecteurs ou matrices utilisés en entrée de squelette sont équitablement répartis par le compilateur entre chaque GPU. La cohérence des données entre l'architecture hôte et les accélérateurs se fait au moyen d'un protocole Modified Shared Invalid (MSI) et les transferts entre mémoires sont de type *paresseux*. Ainsi, les données modifiées [4] par un GPU restent dans son espace mémoire et les transferts entre mémoires hétérogènes sont uniquement effectués lors de l'expression d'une demande d'accès à une donnée invalidée [5]. Cette technique permet de minimiser la quantité de données transférées et d'éliminer les transferts redondants ce qui présente un avantage certain pour les anciennes générations

4. statut *modified*
5. statut *invalid*

de GPU qui utilisaient des transferts synchrones. Aujourd'hui, il est courant de trouver en parallèle des unités de calcul, des unités dédiées pour les transferts mémoire dans les GPUs. Ce modèle architectural permet alors d'effectuer des transferts mémoire asynchrones. Dans un souci d'efficacité, les demandes de transfert mémoire asynchrone doivent être exprimées de façon anticipée pour que les temps de transferts soient masqués par les temps de calcul. En exprimant ces demandes de transfert quand le besoin est détecté, SkePU ne permet pas de masquer les temps de transferts mémoire.

Le paramétrage des squelettes se fait au moyen de macros qui sont traduites au moment de la compilation par le préprocesseur C en fonction de l'architecture cible sélectionnée. Cette méthode souffre cependant d'un problème de rigidité au niveau des paramètres dans l'entête des fonctions.

SkePU inclut la gestion des *kernels* pour les GPUs. L'utilisateur n'a ainsi pas besoin de développer la mise en œuvre de ces *kernels* et peut alors se focaliser sur le développement d'algorithmes. OpenCL fonctionnant sur un modèle de compilation JIT, SkePU met en plus à disposition de l'utilisateur la génération et la compilation de code dynamique pour ce cas particulier. Enfin, les librairies de squelettes inclus dans SkePU sont réparties en 7 catégories, constituant les squelettes de base :

Map qui, pour 1 à 3 vecteurs de taille n en entrée génère 1 vecteur de taille n en sortie.

Reduce qui, utilisée conjointement avec un opérateur binaire associatif et commutatif permet d'obtenir un scalaire à partir d'un vecteur de taille n.

MapReduce qui applique une fonction *Map* sur plusieurs vecteurs de taille n suivi d'une fonction *Reduce* pour obtenir au final un seul scalaire.

MapOverlap qui est une fonction *Map* mais avec une fenêtre de voisinage glissante. Les bords de l'espace mémoire sont interprétés par réplication ou par affectation d'une constante. La taille de la fenêtre glissante est limitée par le nombre de *threads* dans un *block* ainsi que par la quantité de *shared memory* disponible dans le GPU.

MapArray qui, à partir de deux vecteurs de taille n, permet de comparer chaque élément du premier vecteur à l'ensemble des éléments du second vecteur. Le résultat est un vecteur de taille n.

Scan qui est une forme généralisée du pattern *prefix sum* pour des opérateurs binaires associatifs.

Generate enfin, qui permet d'initialiser un vecteur de taille n.

Le choix entre les architectures CPU ou GPU est confié à un modèle d'exécution [52] qui, en fonction de la taille des données à traiter, permet de choisir une implémentation de squelette ainsi que son paramétrage adéquate. La stratégie du modèle d'exécution est générée en utilisant une approche par machine learning [44] à partir d'un modèle entraîné par micro-benchmarking. Cette approche soulève cependant un risque de spécialisation trop avancée pour un modèle de GPU donné. Afin de converger vers une solution en un temps raisonnable, une heuristique [47] a aussi été utilisé. Enfin, une association avec StarPU [20, 21] a été réalisée pour une approche à placement dynamique de tâches sur architecture hybride [46].

Skepu2 [56, 57, 55] apporte plusieurs nouveautés dont l'utilisation de C++11. L'utilisation des macros du préprocesseur C pour la mise en œuvre des squelettes est remplacée dans SkePU2 par une transformation source-à-source au moyen de Clang [85] lors de la pré-compilation. Cette modification permet d'améliorer la détection, au moment de la compilation, des problèmes liés aux signatures de fonctions des squelettes. L'utilisation des

templates à paramètres variables de C++11 apporte aussi plus de souplesse à l'utilisation des squelettes. Le nombre de vecteurs et matrices utilisés en entrée et sortie de squelette n'est ainsi plus fixé par les squelettes de base mais par les besoins de l'algorithme. La classification de ces mêmes squelettes a alors été revue. Les fonctions *MapArray* et *Generate* sont supprimées au profit de la fonction *Map* qui devient plus générique et la fonction *Call* qui fournit une simple fonction appelable, a été ajoutée.

2.3.2 SkelCL

SkelCL [147] fournit un ensemble de squelettes OpenCL afin d'effectuer de la manipulation de données selon les fonctions :

map qui applique une opération à chaque élément d'un vecteur de taille n pour générer un vecteur de même taille n,

zip qui combine deux vecteurs de même taille n en un unique vecteur de taille n,

reduce qui réduit selon une opération donnée, les données d'un vecteur de taille n en un scalaire et

scan qui pour chaque élément i d'un vecteur de taille n cumule une opération sur les i premiers éléments du même vecteur tel que $0 \leq i < n$ et retourne un vecteur de taille n.

OpenCL acceptant des architectures parallèles variées telles que les GPUs et les CPUs, SkelCL fournit une classe C/C++ *Vector* représentant indifféremment un espace mémoire contiguë. Ces *Vectors* sont ainsi utilisés en entrée et en sortie de squelette et un mécanisme permet de vérifier si un transfert de données entre mémoires hétérogènes est nécessaire. SkelCL permet ainsi de porter du code sur GPU au moyen d'OpenCL et est capable de gérer le placement sur plusieurs GPUs.

Enfin, OpenCL ayant un modèle de compilation de type JIT, la compilation des *kernels* s'effectue à l'exécution du programme hôte. À l'opposé, CUDA peut compiler ses *kernels* en même temps que la compilation du code hôte, comme décrit en section 1.4.3. Cette différence se traduit dans l'approche de SkelCL par un placement nécessitant plus d'instructions notamment avec l'utilisation de la classe *Vector*. De plus, une partie des analyses est ainsi différée au moment de l'exécution du programme pour une approche plus dynamique.

2.3.3 Thrust

Mise à disposition par Nvidia, Thrust [128, 25] est une librairie de squelettes C++ fondée sur la STL. Elle réutilise notamment le conteneur de données *Vector* en distinguant les espaces mémoires de l'hôte et de l'accélérateur au moyen de vecteurs *host* et de vecteurs *device*. Les transferts de données entre ces deux types de vecteurs sont effectués à la discrétion de l'utilisateur du fait qu'il n'y ait aucun automatisme sur ce sujet. Thrust fournit aussi plusieurs squelettes algorithmiques parallèles pour CPU et GPU, et la plupart ont un équivalant dans la STL. L'ensemble des algorithmes fournis sont répartis dans cinq catégories :

Sorting pour classer des données selon un ordonnanceur donné,

Reordering qui permet de modifier un espace de données selon un prédicat,

Reductions pour réduire un espace de données en une valeur unique au moyen d'un opérateur binaire,

Scan/Prefix Sums pour rechercher un élément dans un espace de données,

Transformations qui applique une opération pour chaque élément d'un espace de données en sortie.

La philosophie de Thrust diffère de celle des autres solutions présentées dans cette section. La librairie de Nvidia, a pour vocation de simplifier la programmation parallèle sur GPU en fournissant d'une part, une API simplifiée comparée à celle de CUDA et d'autre part, une collection d'algorithmes optimisés sans avoir nécessairement besoin de programmer les squelettes utilisés.

2.3.4 Bones

Nugteren et Corporaal [115] se sont intéressés à la classification d'algorithmes de traitement d'images. Ils ont extrait d'OpenCV une classification de patterns de traitement d'images que l'on retrouve régulièrement au sein de la librairie. Chacun de ces patterns algorithmiques, tel que celui de la réduction de données par exemple, correspond à un squelette dont l'implémentation est optimisé pour une architecture particulière. Dans leur étude initiale, Nutgeren et Corporaal n'incorporaient pas de phase d'analyse automatisée. Le développeur devait ainsi s'assurer de la validité d'utilisation d'un squelette donné en fonction des dépendances existantes au sein du code source. Une fois le squelette sélectionné, celui-ci devait encore être complété par le développeur selon les spécifications fonctionnelles souhaitées.

Les squelettes ont été définis selon les quatre classes d'accès mémoire :
— les accès pixel à pixel
— les accès voisinage à pixel
— les accès de type réduction globale vers un scalaire
— les accès de type réduction globale vers un vecteur

Enfin, plusieurs règles de réécriture ont été définies. Celles-ci sont basées sur du *pattern matching*, implémenté par expression régulière et appliqué sur le code original. Ces règles lorsqu'elles peuvent s'appliquer, permettent de transformer des accès en mémoire globale en accès vers la *shared memory*, de linéariser les accès aux tableaux multi-dimensionnels ou encore de définir la taille des *blocks* de *threads*. Elles servent cependant uniquement au placement sur GPU et n'ont pas de rôle d'optimisation.

Ces travaux de classification ont servi de base pour lancer Bones [113, 111, 114]. Bones est un compilateur source-à-source, permettant de mettre en œuvre les squelettes algorithmiques précédemment décrits pour générer du code CUDA ou OpenCL. Les architectures parallèles ainsi ciblées sont les GPUs Nvidia et AMD ainsi que les CPUs de type x86. Bones a été écrit en Ruby et s'appuie sur la librairie C Abstract Syntax Tree (CAST) pour transformer le langage C utilisé en entrée en AST. L'utilisation des squelettes de programmation s'effectue en délimitant les portions de code concernées au moyen de directives de type pragma C. Le processus de compilation de Bones passe par quatre étapes :

1. Dans un premier temps, CAST est utilisé en tant que préprocesseur pour extraire les zones annotées et les transformer en AST.
2. À partir de ces zones extraites, une analyse de l'AST est effectuée pour distinguer les allocations statiques des allocations dynamiques ou encore pour calculer les empreintes mémoire.
3. Des transformations de code liées à l'*outlining* sont alors appliquées.
4. Enfin, l'application des *patterns* de squelettes définis par les annotations par l'utilisateur conclut ce processus.

Au final, le code généré comprend alors plusieurs *kernels* ainsi que le code nécessaire à leur utilisation tel que les transferts mémoire. La sélection des squelettes lors de l'étape

4 a depuis été automatisé [118] grâce à une classification automatique [116, 117]. De plus la génération de code a été étendue [114] à OpenMP pour les CPUs et HLS-C pour les FPGA. Un ensemble d'optimisations a aussi été ajouté [114] :

Le *threads coarsening* correspond à de la fusion de *threads*. La localité des données peut ainsi être améliorée ce qui se traduit par un temps d'accès réduit pour les données communes, au prix d'une réduction du parallélisme.

La scalarisation d'accès mémoire permet d'augmenter l'utilisation des registres tout en profitant de la bande passante mémoire la plus élevée du GPU. Cette optimisation est cependant limitée par la quantité réduite de registres par *thread*.

La réduction mono-dimensionelle des nids de boucles permet d'augmenter les itérations de boucles et d'améliorer la quantité de tâches placées sur le GPU.

La fusion des transferts de données entre l'accélérateur et son hôte permet d'éliminer les transferts redondants sous certaines conditions.

La fusion des *kernels* permet dans certains cas de réduire les coûts de lancement des *kernels*, d'améliorer la localité des données et d'éliminer des calculs redondants.

Enfin, Bones incorpore une classification modulaire et paramétrable des squelettes selon une nomenclature décrite dans l'article [112]. Celle-ci permet de résoudre le dilemme de sélection entre un classification à grain fin, qui apporte de meilleures performances, et une classification à grain grossier, qui apporte une utilisation plus simple des squelettes.

2.4 Optimiseurs GPU

Trois approches issues de l'état de l'art du placement sur GPU viennent d'être abordées dans les sections 2.1, 2.2 et 2.3. Certaines des solutions qui ont été décrites abordent le sujet de l'optimisation des performances en raffinant le placement initialement défini. Ce raffinement peut alors prendre en compte l'utilisation de spécificités architecturales telle que la *shared memory*, ainsi que des contraintes architecturales telle que la quantité de registres disponibles. Le modèle polyédrique est une solution courante dans cet état de l'art pour rechercher une solution de placement optimale tout en prenant en compte l'ensemble des contraintes d'une architecture donnée. Concrètement, les transformations effectuées dans ce cadre se traduisent par une amélioration des temps d'exécution sur GPU. Dans le cadre de cette section, nous ne reviendrons pas sur les solutions déjà abordées. Leur présentation respective intègre déjà les informations relatives à l'optimisation du placement. En revanche, *CUDA-Lite*, présenté en section 2.4.1, ainsi que la solution de Yang *et al.* (2.4.2) se différencient en utilisant en entrée un code déjà placé sur GPU au moyen de CUDA. En conséquence, ces deux solutions s'intéressent exclusivement à l'optimisation du placement sur GPU. Enfin, *GPUCC* détaillé en section 2.4.3 est un compilateur source vers cible. De même, il accepte en entrée un code CUDA. Mais à la différence des solutions présentées dans cette section, il génère un binaire exécutable pour GPU Nvidia. Il incorpore de plus une étape de transformation de code afin d'améliorer le placement sur GPU.

2.4.1 CUDA-Lite

Ueng *et al.* [153] ont développé un outil nommé *CUDA-Lite* pour effectuer des transformations source-à-source de programmes CUDA. Il permet au moyen de directives dans le code original, d'améliorer l'utilisation des hiérarchies mémoire au sein des GPUs Nvidia. Le code source en entrée doit être écrit en langage *C* et CUDA et ne doit employer