THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à MINES ParisTech

Compilation efficace d'applications de traitement d'images pour processeurs manycore

École doctorale n°432

SCIENCES ET MÉTIERS DE L'INGÉNIEUR

Spécialité INFORMATIQUE TEMPS RÉEL, ROBOTIQUE ET AUTOMATIQUE

COMPOSITION DU JURY :

Fabrice Rastello
Inria, Rapporteur

Cédric Bastoul
Université de Strasbourg, Rapporteur

Antoniu Pop
University of Manchester, Examinateur

Dumitri Potop-Butucaru
Inria, Examinateur

Stéphane Louise
CEA LIST, Examinateur

François Irigoin
MINES ParisTech, Directeur de thèse

Fabien Coelho
MINES ParisTech, Maître de thèse

Benoît Dupont de Dinechin
Kalray, Invité

Michel Bilodeau
MINES ParisTech, Invité

Soutenue par **Pierre Guillou**
le mercredi 30 novembre 2016

Dirigée par **François Irigoin**
et par **Fabien Coelho**

MINES ParisTech | PSL RESEARCH UNIVERSITY PARIS

Compilation efficace d'applications de traitement d'images pour processeurs manycore

Pierre Guillou
MINES ParisTech, PSL Research University

Thèse de doctorat soutenue le mercredi 30 novembre 2016

Nous prendrons le temps de vivre
D'être libres, mon amour
Sans projets et sans habitudes
Nous pourrons rêver notre vie

Viens, je suis là, je n'attends que toi
Tout est possible, tout est permis

Viens, écoute ces mots qui vibrent
Sur les murs du mois de mai
Ils nous disent la certitude
Que tout peut changer un jour

Viens, je suis là, je n'attends que toi
Tout est possible, tout est permis

— Georges Moustaki, *Le temps de vivre*

Résumé

Nous assistons à une explosion du nombre d'appareils mobiles équipés de capteurs optiques : *smartphones*, tablettes, drones… préfigurent un Internet des objets imminent. De nouvelles applications de traitement d'images (filtres, compression, réalité augmentée) exploitent ces capteurs mais doivent répondre à des contraintes fortes de vitesse et d'efficacité énergétique. Les architectures modernes — processeurs manycore, GPUs,… — offrent un potentiel de performance, avec cependant une hausse sensible de la complexité de programmation.

L'ambition de cette thèse est de vérifier l'adéquation entre le domaine du traitement d'images et ces architectures modernes : concilier programmabilité, portabilité et performance reste encore aujourd'hui un défi. Le domaine du traitement d'images présente un fort parallélisme intrinsèque, qui peut potentiellement être exploité par les différents niveaux de parallélisme offerts par les architectures actuelles. Nous nous focalisons ici sur le domaine du traitement d'images par morphologie mathématique, et validons notre approche avec l'architecture manycore du processeur MPPA de la société Kalray.

Nous prouvons d'abord la faisabilité de chaînes de compilation intégrées, composées de compilateurs, bibliothèques et d'environnements d'exécution, qui à partir de langages de haut niveau tirent parti de différents accélérateurs matériels. Nous nous concentrons plus particulièrement sur les processeurs manycore, suivant les différents modèles de programmation : OpenMP ; langage flot de données ; OpenCL ; passage de messages. Trois chaînes de compilation sur quatre ont été réalisées, et sont accessibles à des applications écrites dans des langages spécifiques au domaine du traitement d'images intégrés à Python ou C. Elles améliorent grandement la portabilité de ces applications, désormais exécutables sur un plus large panel d'architectures cibles.

Ces chaînes de compilation nous ont ensuite permis de réaliser des expériences comparatives sur un jeu de sept applications de traitement d'images. Nous montrons que le processeur MPPA est en moyenne plus efficace énergétiquement qu'un ensemble d'accélérateurs matériels concurrents, et ceci particulièrement avec le modèle de programmation flot de données. Nous montrons que la compilation d'un langage spécifique intégré à Python vers un langage spécifique intégré à C permet d'augmenter la portabilité et d'améliorer les performances des applications écrites en Python.

Nos chaînes de compilation forment enfin un environnement logiciel complet dédié au développement d'applications de traitement d'images par morphologie mathématique, capable de cibler efficacement différentes architectures matérielles, dont le processeur MPPA, et proposant des interfaces dans des langages de haut niveau.

Mots clés : Langages spécifiques à un domaine, Compilation, Processeurs manycore, Traitement d'image, Modèles de programmation

Abstract

Many mobile devices now integrate optic sensors; smartphones, tablets, drones… are foreshadowing an impending Internet of Things (IoT). New image processing applications (filters, compression, augmented reality) are taking advantage of these sensors under strong constraints of speed and energy efficiency. Modern architectures, such as manycore processors or GPUs, offer good performance, but are hard to program.

This thesis aims at checking the adequacy between the image processing domain and these modern architectures: conciliating programmability, portability and performance is still a challenge today. Typical image processing applications feature strong, inherent parallelism, which can potentially be exploited by the various levels of hardware parallelism inside current architectures. We focus here on image processing based on mathematical morphology, and validate our approach using the manycore architecture of the Kalray MPPA processor.

We first prove that integrated compilation chains, composed of compilers, libraries and run-time systems, allow to take advantage of various hardware accelerators from high-level languages. We especially focus on manycore processors, through various programming models: OpenMP, data-flow language, OpenCL, and message passing. Three out of four compilation chains have been developed, and are available to applications written in domain-specific languages (DSL) embedded in C or Python. They greatly improve the portability of applications, which can now be executed on a large panel of target architectures.

Then, these compilation chains have allowed us to perform comparative experiments on a set of seven image processing applications. We show that the MPPA processor is on average more energy-efficient than competing hardware accelerators, especially with the data-flow programming model. We show that compiling a DSL embedded in Python to a DSL embedded in C increases both the portability and the performance of Python-written applications.

Thus, our compilation chains form a complete software environment dedicated to image processing application development. This environment is able to efficiently target several hardware architectures, among them the MPPA processor, and offers interfaces in high-level languages.

Keywords : Domain Specific Languages, Compilation, Manycore Processors, Image Processing, Programming Models

À propos de l'auteur

D'origine bretonne, Pierre Guillou a reçu le diplôme d'ingénieur de l'École des mines de Paris en 2013, avant de commencer un doctorat dans le domaine de l'informatique, dans le Centre de recherche en informatique de l'école, à Fontainebleau. Il a soutenu avec succès sa thèse, intitulée « compilation efficace d'applications de traitement d'images pour processeurs manycore », en novembre 2016, et a été subséquemment nommé docteur de l'Université de recherche Paris Sciences et Lettres.

Ses recherches se concentrent sur l'adéquation entre les architectures manycores modernes, notamment celle du processeur MPPA-256 du français Kalray, et le domaine du traitement d'images. Il a pour cela développé et étendus plusieurs outils, qui visent à automatiser le portage d'applications de traitement d'images écrites dans des langages de programmation de haut niveau vers les cibles architecturales complexes que sont ces processeurs manycores.

Au quotidien, il aime travailler avec flegme et professionnalisme sur son bureau debout, et s'appuie sur des logiciels archaïques, comme *emacs*, ou plus modernes, comme git ou zsh. Il s'intéresse, en particulier, à ces outils qui facilitent ou améliorent le confort de la programmation, comme le langage Rust (qu'il faudrait qu'il apprenne, un jour), ou bien les outils de coloration de ligne de commande (comme `colorlatex`), de formatage de code, d'analyse statique, etc. Il est aussi l'auteur de la feuille de style LaTeX qui génère la couverture PSL du présent mémoire.

Durant son temps libre, il aime jouer au baby foot et au billard, découvrir le Gâtinais à vélo, nager (mais pas dans la Seine !) ou bien monter à la capitale à la recherche de curiosités locales. Il commande souvent des *Piña Colada* à l'El Floridita local : le Glasgow. Vous pouvez parfois le retrouver à jouer de sa flûte traversière sur un parking la nuit.

Remerciements !

Je me livre, dans le présent chapitre, au périlleux exercice des remerciements ; que les oubliés me pardonnent si, par défaut de mémoire, ils n'y sont pas mentionnés.

Commençons par la fin : je remercie bien évidemment tous les membres de mon jury pour avoir fait le déplacement jusqu'à la capitale et m'avoir écouté parler pendant trois quarts d'heure avant de pouvoir me questionner. Ils m'ont, à mon grand soulagement, attribué ce titre de docteur tant convoité. Je veux bien sûr parler de MM. Cédric Bastoul, Fabrice Rastello, Antoniu Pop, Stéphane Louise, Dumitru Potop Butucaru, Benoît Dupont de Dinechin et Michel Bilodeau.

Recentrons nous désormais sur mon lieu de travail, le Centre de recherche en informatique de l'École des mines. Je remercie toute son équipe pour m'avoir accueilli durant les trois années qu'a duré ce doctorat. Ces remerciements s'adressent tout particulièrement à mes encadrants : Fabien Coelho, dont j'avais fait la connaissance dans mon cursus ingénieur et qui m'a poussé à entreprendre cette thèse, et bien entendu François Irigoin, mon directeur de thèse, qui m'a octroyé toute la liberté dont je n'aurais sans doute jamais rêvé dans mon travail, gage, je l'espère, de la qualité de celui-ci. Je n'oublie évidemment pas les autres permanents du CRI : Corinne, Claire et Catherine, dont les chaleureuses aides ont résolu nombre de mes problèmes, Pierre J. mon correcteur accrédité et aviateur favori (s'il reste des fautes, c'est la sienne), Benoît qui n'a malheureusement pas pu être le photographe officiel de ma soutenance, Emilio qui a partagé mon bureau et supporté mes silences, et enfin Olivier, Claude et Laurent.

Merci aux doctorants que j'y ai côtoyés : Vivien, qui m'a appris mes premiers rudiments de LaTeX, de git, de zsh et de baby foot, Nelson, avec qui j'ai partagé bon nombre de sorties pizza ainsi qu'une mémorable promenade à vélo, Pierre W., trop souvent à Paris, Florian, Adilla, Patryk et Bruno les derniers venus. Quelques membres croisés de façon plus temporaires méritent aussi une citation dans ces colonnes : je songe notamment à Arnaud, à Wajdi, à Dounia et à Amira.

Ce séjour de trois ans, dans le cadre exceptionnel de la cité impériale de Fontainebleau, a été l'occasion de nombreuses rencontres et échanges avec les doctorants, certains maintenant docteurs, des autres centres. Mention spéciale au centre de morphologie mathématique, à ses cours de danse latine et pauses méridiennes teintées de baby foot. Je pense plus particulièrement à Sébastien, notre fournisseur local de Dopamine, Vaïa notre géniale professeure de salsa et kizomba, J.B., dont on cherche toujours le mode muet, Gianni, qui ne nous a toujours pas présenté sa sœur, Robin le musicien matheux émo et Angélique, notre majesté des mouches radioactives et du gavage drôle de poches à la chocolatine. Mais également à

— mes adversaire aux manettes du baby : Théo le beauf barbu hipster, Jean-Charles, qui nous a quittés pour d'autres cieux plus perfides, Haisheng également nageur et cycliste, Amin notre maître-apprenti artificiel, et Albane, bientôt de retour à Fontainebleau ;
— par-delà la Seine, du côté d'Héricy, les anciens et actuels résidents de la « coloc » : Fanny, Enguerrand et Benjamin ;
— le clan des italiennes : Mariangela, Sara et Simona ;
— les casseurs lécheurs de cailloux : Dariouche, Arezki et Martin ;
— les joueurs de plateau : Luc et Serge ;
— les nouveaux doctorants et assimilés : Marine, Yuan, Kaiwenn, Aurélien, Rémi et Nico le co-bureau « plus belge la vie » ;
— les anciens : Antoine, Élise et Mauro, Borja et Joris ;
— et enfin les géo-portugais : Ricardo, Hector et Esteban.

Je tiens tout particulièrement à remercier Robin et Angélique, non pour leurs capacités inexistantes en matière d'informatique, mais pour leurs talents sociaux, musicaux, glasgophiles et conspirationnistes. Je tiens en outre à souligner le support et les encouragements que Robin m'a prodigués dès que l'absence d'inspiration se faisait sentir, au travers de son fameux leitmotiv « Va bosser feignasse ! », et ce malgré son rendez-vous manqué avec la grammaire du présent ouvrage. Angélique a, quant à elle, tenté de s'occuper d'autres aspects de ma vie, et depuis, je lui porte ses courses : l'évolution est notable. Petite pensée également pour les barmen du Glasgow, qui sans notre apport et celui de Théodore C., seraient certainement au chômage depuis longtemps, mais qui ont malgré eux contribué à l'ambiance alcoolo-studieuse commune à nombre de thèses bellifontaines. [1]

Enfin, je tiens à rendre hommage à ma famille, qui m'a soutenu malgré la distance, et n'a pas hésité à faire le déplacement en masse pour ma soutenance. Je tâcherai de leur rendre la pareille.

Sans plus attendre, je laisse le lecteur débuter la lecture de cet ouvrage par la table des matières — j'ai mobilisé toute la maîtrise de mon talent littéraire dans sa réalisation —, ou, pour les plus téméraires, directement par son chapitre introductif page 1. Bonne lecture !

1. Pour ma défense, cette phrase a été écrite par Robin, sous influence au moins caféinique (si ce n'est houblonnée), quand j'ai voulu en faire le correcteur officiel de ce mémoire. À consommer avec modération.

Table des matières

Table des figures

Liste des tableaux

Liste des extraits de code

Liste des algorithmes

Chapitre 1

Introduction

The Wheel of Time turns, and Ages come and pass, leaving memories that become legend. Legend fades to myth, and even myth is long forgotten when the Age that gave it birth comes again. In one Age, called the third age by some, an Age yet to come, an age long pass, a wind rose in the Mountains of Mist. The wind was not the beginning. There are neither beginnings or endings to the turning of the Wheel of Time. But it was a beginning.

— Robert Jordan, *The Wheel of Time*

L'informatique moderne, vecteur de changement sociétal, s'appuie sur des unités de calcul de plus en plus complexes. Face à l'apparition de phénomènes quantiques liés à l'augmentation de la finesse de gravure, des circuits spécialisés voient le jour. En parallèle, les usages évoluent : poussées par le succès des *smartphones*, la réalité virtuelle et la réalité augmentée sont désormais accessibles au grand public et les véhicules autonomes le seront d'ici au plus quelques années. Pouvoir rapidement écrire des applications qui s'exécutent avec de bonnes performances sur un maximum d'appareils est un enjeu économique critique. Au carrefour entre ces usages de plus en plus avancés, donc consommateurs de puissance et d'énergie, et ces architectures complexes, il est nécessaire de développer des environnements logiciels qui offrent facilité de programmation, portabilité et performance.

1.1 Microprocesseurs : vers la fin de l'évolution exponentielle ?

Le développement de l'informatique a mené à une révolution sociétale. Des *mainframes* aux *smartphones* et tablettes actuelles, en passant par les ordinateurs de bureau puis les ordinateurs portables, les progrès constants dans les performances des unités de calcul et les communications en réseau ont pavé la voie vers une explosion des usages numériques dans notre société. La loi de Moore [90], qui prédit le doublement du nombre de transistors tous les deux ans, est entrée dans la culture populaire et les évolutions récentes dans l'Internet des objets, les véhicules autonomes et autres *wearables* ne font que renforcer la croyance populaire en cette prédiction.

Pourtant, augmenter la fréquence des calculs dans les processeurs, comme ce fut le cas jusqu'au milieu des années 2000, provoque des pertes d'énergie sous forme de chaleur qu'il faut évacuer, sous peine de faire fondre le matériel. En outre, il est impossible d'augmenter indéfiniment la finesse de gravure des transistors : dès que leur taille devient inférieure

à quelques atomes, des phénomènes quantiques interfèrent avec leur fonctionnement normal. Pour contourner ces phénomènes physiques, les fabricants de processeurs se doivent d'innover.

C'est ainsi que les processeurs multicœurs ont été introduits au début du millénaire, comme solution à la course à la puissance de calcul. Ces derniers sont aujourd'hui présents dans tous les ordinateurs grand public actuels. L'émergence des processeurs graphiques a également permis de décharger le processeur principal des calculs simples et répétitifs. L'augmentation de la bande passante réseau a également rendu possible l'élaboration de superordinateurs, formés de plusieurs milliers voire millions de processeurs [117].

Toutes ces solutions ont l'inconvénient d'augmenter la complexité des architectures matérielles et, par conséquent, l'expérience nécessaire pour en tirer parti. Pour écrire des applications performantes, il est désormais nécessaire de connaître les caractéristiques des architectures sur lesquelles elles seront exécutées. Malheureusement, les innovations en matière d'architectures matérielles exotiques [111] n'arrangent en rien cette évolution.

De plus, le développement de nouveaux processeurs semble ralentir à mesure que la finesse de gravure tend à s'approcher du *mur quantique* : le fabricant Intel a fait évoluer sa stratégie « *Tick-Tock* » qu'il suivait depuis 2007 pour développer ses processeurs en introduisant une étape supplémentaire, pour laisser le temps à ses équipes de contourner les problèmes induits par la finesse de gravure. La fin de la loi de Moore semble proche. En parallèle, d'autres métriques, comme la consommation énergétique, concurrencent la performance brute des unités de calcul, entre autres pour les usages mobiles, où l'autonomie est un facteur critique, ou bien dans les datacenters et les superordinateurs, où la facture d'électricité devient un coût substantiel.

L'apparition de nouvelles architectures plus complexes et le ralentissement de la loi de Moore rendent plus ardu le travail des développeurs d'applications, qui ne peuvent plus se reposer sur l'augmentation continue des performances mais doivent optimiser leurs applications sur différentes configurations architecturales. Les processeurs manycore, évolution logique des processeurs multicœurs modernes, rassemblent ainsi des dizaines voire des centaines d'unités de calculs sur une même puce.

Figure 1.1 – Un processeur manycore : le MPPA de Kalray

À titre d'exemple, le processeur MPPA Manycore [7, 43], conçu par la société française Kalray, compte ainsi 256 cœurs de calculs, regroupés en seize nœuds de seize cœurs, qui communiquent à travers un réseau sur puce. Ce type d'architecture nouvelle présente différents niveaux de parallélisme — nœuds de calcul à mémoire partagée, mémoire distribuée

à l'échelle de la puce, utilisation comme accélérateur matériel aux côtés d'une machine hôte — et des contraintes fortes — mémoire répartie de taille très faible, entrées/sorties complexes — qui rendent son utilisation difficile. La complexité de l'architecture de ce processeur manycore est néanmoins le prix à payer pour sa faible consommation énergétique.

Fort heureusement, des modèles de programmation permettent de tirer parti d'une telle architecture, en facilitant la programmation et en rendant le code applicatif plus portable. Ces modèles laissent un contrôle plus ou moins avancé et explicite au développeur, qui devra alors choisir entre réaliser des optimisations spécifiques aux cibles matérielles, qui génèrent de meilleures performances, ou bien se cantonner à une implémentation générique portable mais moins performante.

1.2 Des applications et des usages toujours plus gourmands en performance

L'évolution des architectures matérielles, l'augmentation des performances dans les calculs et les accès mémoire et la réduction de la consommation énergétique ont permis le développement de nouveaux usages. La révolution des *smartphones* depuis l'introduction en 2007 du premier iPhone a apporté au public et concentré téléphone, accès Internet, écran tactile, localisation GPS et appareil photo dans un même objet capable de tenir dans la poche. Des applications innovantes sont apparues, profitant du modèle économique de ces nouveaux appareils, fondés sur des marchés d'applications payantes. Ces applications ont parfois transformé certains usages sociaux, à l'exemple de Twitter et Periscope, qui ont bouleversé le partage d'informations et d'actualités, ou Tinder, qui a révolutionné le monde des rencontres en ligne.

À la suite du succès de l'iPhone, Apple a introduit l'iPad en 2011, tablette tactile à mi-chemin du smartphone et de l'ordinateur portable. Dédiées principalement au divertissement et à la consommation de contenus, les tablettes tactiles ont également connu un large succès, quoique moindre, principalement dû au grand nombre d'applications compatibles. Depuis, les *smartwatches* et les bracelets fitness ont annoncé l'émergence de l'Internet des objets. Les objets du quotidien, connectés, se piloteront désormais depuis le téléphone ou la tablette.

Les performances des téléphones et des tablettes tactiles ont évolué en parallèle du matériel qui les compose. Les puces graphiques des téléphones actuels ont ainsi des capacités similaires à celles des consoles de salon d'il y a dix ans. Les jeux vidéos, comme Angry Birds, Candy Crush ou plus récemment Clash of Clans, se sont développés sur ces nouveaux supports. Caractérisés par leur contenu monétisé et la forte attractivité de leurs parties rapides, ces jeux ont rapporté un grand profit à leurs créateurs. Toutefois, l'usage vidéo-ludique tend à réduire l'autonomie des appareils, facteur critique de la mobilité. En réponse, les récents développements dans les processeurs centraux et graphiques évoluent vers une réduction de la consommation énergétique, et donc vers une meilleure efficacité.

Aujourd'hui, les puces graphiques des téléphones portables sont suffisamment puissantes pour proposer la réalité virtuelle (au travers par exemple du Samsung Gear VR) ou la réalité augmentée à tous. L'application phare au moment de l'écriture de ces lignes est en effet Pokémon Go, un jeu pour *smartphone* liant géolocalisation et réalité augmentée. Là encore, ces fonctionnalités nécessitent beaucoup d'énergie et nuisent à l'autonomie.

En parallèle de ces applications grand public, les systèmes embarqués présents dans la plupart des véhicules actuels, des voitures aux sondes spatiales en passant par les avions, régissent un grand nombre de fonctions, de la fusion de capteurs aux systèmes multimédia. Dans ces systèmes, la consommation énergétique joue en général un rôle critique.

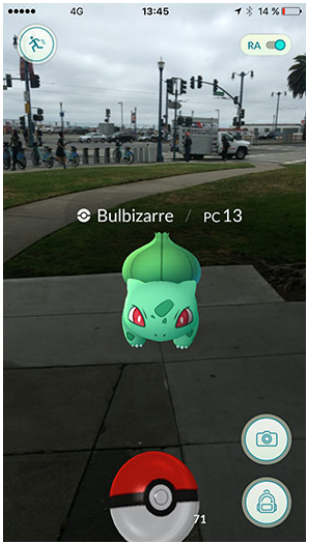


Figure 1.2 – Pokémon Go : un Bulbizarre en réalité augmentée — et une batterie bientôt vide

À l'autre bout du spectre, certaines disciplines scientifiques nécessitent une grande puissance de calcul afin de résoudre les équations de la chromodynamique quantique ou de la météorologie, par exemple, afin de déterminer l'impact du changement climatique. Les GAFAM (Google, Apple, Facebook, Amazon et Microsoft, grands acteurs du monde numérique actuel), les NATU (les nouveaux challengers que sont Netflix, Airbnb, Telsa et Uber) et les BATX (Baidu, Alibaba, Tencent et Xiaomi, les concurrents chinois) ont, pour gérer l'ensemble des données à leur disposition, fait construire des centres de données partout dans le monde : c'est l'informatique en nuage. Les serveurs contenus dans ces *datacenters* fonctionnent en continu et dégagent de la chaleur, qui doit être évacuée en permanence. Une diminution de quelques pour cent de la consommation énergétique de ces serveurs peut économiser beaucoup d'argent.

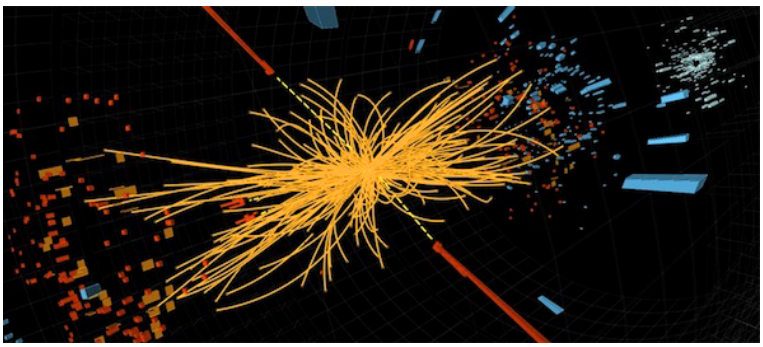


Figure 1.3 – Chaque collision du *Large Hadron Collider* du CERN génère de très grandes quantités de données

Domaine actuellement en plein essor, l'apprentissage artificiel requiert également une grande puissance de calcul, afin d'entraîner des réseaux de neurones de plus en plus profonds.

Les processeurs graphiques haut de gamme ciblent désormais ce domaine d'applications dans lequel ils excellent [38]. Les crypto-monnaies virtuelles, comme le Bitcoin [17], nécessitent, elles aussi, du matériel spécifique et de fortes quantité d'énergie pour valider les transactions et créer de la monnaie ex nihilo à travers le « minage ».

1.3 Le traitement d'images, un domaine applicatif en plein essor

L'émergence des smartphones a mis une caméra dans toutes les poches. De nombreuses applications s'appuient sur cette fonctionnalité ubiquitaire pour proposer, par exemple, des filtres Instagram ou de la reconnaissance faciale pour « tagger » ses amis sur Facebook.

Les avancées modernes dans le champ de la vision par ordinateur reposent principalement sur les techniques modernes d'apprentissage artificiel et, notamment, sur les réseaux de neurones profonds, champ d'étude exacerbant la concurrence entre les grandes entités du monde numérique — Google, Microsoft et Amazon ayant récemment ouvert au public leurs logiciels de *deep learning*.

Ces techniques de traitement d'images permettent notamment de reconnaître des objets automatiquement, voire de proposer une description à partir d'une image. Ces travaux sont clés dans le développement des véhicules autonomes, afin notamment de pouvoir déchiffrer la signalisation routière et repérer correctement les usagers à proximité.

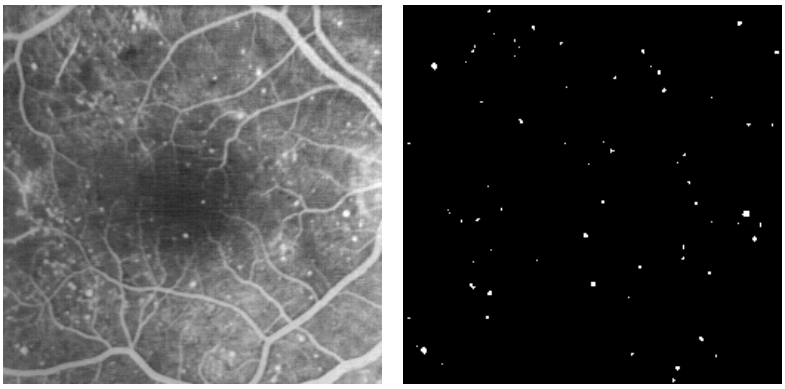
Des applications moins futuristes, comme la reconnaissance optique de caractères ou le post-traitement d'imagerie médicale, sont déjà utilisées à l'heure actuelle. La figure 1.4 montre ainsi une application d'extraction de plaque d'immatriculation d'une voiture, par exemple à partir d'une image issue d'un radar automatique. Couplée à un système de reconnaissance de caractères et à la base de données adéquate, cette application permet d'envoyer directement et automatiquement les amendes pour excès de vitesse à l'adresse du propriétaire du véhicule contrevenant et, par conséquent, contribue à la sécurité routière.

(a) Image initiale (b) Image finale

FIGURE 1.4 – Application de traitement d'images : extraction de plaque d'immatriculation

L'application illustrée par la figure 1.5 analyse une image de fond d'œil et détecte les lésions de la rétine, symptomatiques de la dégénérescence maculaire liée à l'âge, qui provoque une perte de la vision centrale. Détecter automatiquement et rapidement ces lésions permet de porter un diagnostic plus fiable et plus tôt.

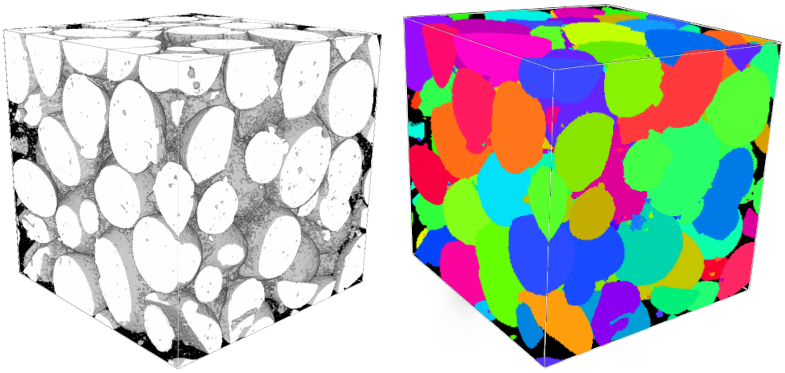
La figure 1.6 représente quant à elle la segmentation d'une image en trois dimensions obtenue par microtomographie à rayons X. Les grains représentés dans cette image proviennent de matériaux pyrotechniques. La segmentation consiste ici à détecter chaque grain indivi-



(a) Image initiale (b) Image finale

Figure 1.5 – Application de traitement d'images : détection de dégénérescence maculaire dans un fond d'œil

duellement, afin de pouvoir modéliser le matériel pour déterminer certaines caractéristiques physiques — ici, force de l'explosion ou rayon d'action.



(a) Image initiale (b) Image finale

Figure 1.6 – Application de traitement d'images : segmentation d'une acquisition 3D par microtomographie aux rayons X (avec l'aimable autorisation de Théodore Chabardès)

Les applications de traitement d'images s'appuient souvent sur des bibliothèques logicielles, qui implémentent différents opérateurs et fournissent des interfaces pour les réutiliser. L'utilisation de ces bibliothèques évite au développeur d'applications de réinventer la roue en implémentant à nouveau les opérateurs dont il a besoin. Les développeurs de bibliothèques, quant à eux, doivent fournir une interface de programmation de haut niveau, afin de faciliter l'usage de leur bibliothèque, ainsi que des versions optimisées de leurs opérateurs sur les processeurs modernes. Porter une bibliothèque sur une nouvelle architecture matérielle, comme un processeur graphique, est généralement long et coûteux : les opérateurs doivent être réécrits et optimisés pour l'architecture cible, en utilisant généralement un autre modèle de programmation. Deux bibliothèques de traitement d'images sont particulièrement utilisées dans cette thèse : SMIL [46, 47] et FREIA [15, 29].

Au vu de ses applications actuelles et futures, le traitement d'images est amené à jouer un grand rôle dans notre société en rendant possible l'automatisation de tâches complexes. Fournir des environnements logiciels capables de tirer efficacement parti des architectures modernes et à venir, tout en proposant une interface de haut niveau pour faciliter la programmation, est un enjeu clef.

1.4 La compilation, entre applications, bibliothèques et matériel

La compilation consiste à convertir un langage de programmation en un autre. Les compilateurs standards, par exemple GCC [109] ou LLVM [84], transforment des applications ou des bibliothèques écrites à la main dans des langages de programmation de haut niveau, comme C ou C++, vers des langages de programmation de plus bas niveau, comme l'assembleur ou le langage machine. Les compilateurs sont capables d'optimiser le code machine généré pour différentes cibles matérielles, en s'appuyant sur des représentations internes performantes. Cependant, l'espace d'optimisation se montre habituellement trop vaste pour être complètement parcouru, et le contrôle laissé aux développeurs par les langages de la famille du C empêche parfois certaines optimisations.

Les langages spécifiques à un domaine, en anglais DSLs (« Domain-Specific Languages »), sont des langages de haut niveau censément faciles à utiliser pour développer rapidement des applications dans un domaine donné, en limitant les connaissances nécessaires sur les architectures cibles. Les concepteurs de DSLs font le lien avec les cibles matérielles grâce au compilateur, qui va pouvoir implémenter des optimisations spécifiques au domaine et des optimisations spécifiques aux matériels. Certains DSLs reprennent la syntaxe d'un langage de programmation plus généraliste. On les qualifie alors de DSLs embarqués ou *embedded* DSLs (e-DSLs). Les interfaces de programmation de FREIA et de SMIL s'apparentent ainsi à des DSLs embarqués, le langage hôte étant C dans le cas de FREIA, et C++ ou Python pour SMIL.

Sous-domaine particulier, la compilation source-à-source consiste à générer, non pas un code machine ou assembleur, comme le font les compilateurs traditionnels, mais un nouveau code source. De tels compilateurs, à l'instar de PIPS [41, 75], permettent d'optimiser des codes sources, tout en gardant une sortie lisible, facilement modifiable et réutilisable par d'autres compilateurs en aval de la chaîne de compilation.

1.5 Sujet

Cette thèse vise la «*compilation efficace d'applications de traitement d'images pour processeurs manycore*» en se concentrant sur l'adéquation entre un domaine applicatif et de nouvelles architectures matérielles.

Compiler efficacement consiste à résoudre, au cœur des problématiques de développement logiciel actuels, les questions de programmabilité, de portabilité et de performances — aussi appelées les *3P* — des applications, qui sont souvent complexes à concilier. La *programmabilité* d'un langage de programmation est l'ensemble des facilités qu'il offre pour développer plus rapidement des applications : syntaxe expressive, structures de données de haut niveau, ramasse-miettes, paradigmes orienté objet ou fonctionnel, etc. Ces propriétés réduisent le code des applications et le rendent plus extensible et plus maintenable. La *portabilité* est la capacité d'un langage à pouvoir s'exécuter sur plusieurs cibles matérielles. Enfin, la *performance* d'une application est définie par les métriques de son temps d'exécution et de sa consommation d'énergie sur une cible matérielle donnée.

Concilier ces *3P* est, aujourd'hui encore, une gageure. Traditionnellement, la programmabilité est apportée par l'utilisation d'interfaces dans des langages de haut niveau, tandis qu'optimiser les temps d'exécution requiert un contrôle sur le matériel, qui n'est possible qu'au travers de langages de bas niveau. Ces optimisations sont généralement peu portables en dehors d'un ensemble restreint d'architectures matérielles conventionnelles.

Face aux limites physiques de la fréquence de calcul et de la finesse de gravure des processeurs modernes, de nouvelles architectures complexes voient le jour. Les processeurs manycore, qui rassemblent des dizaines, voire des centaines d'unités de calcul sur une même puce, se placent à mi-chemin entre les processeurs centraux conventionnels et les processeurs graphiques. Plusieurs modèles de programmation permettent d'en tirer parti.

Le traitement d'images, domaine industriel critique, est adapté à ce type d'architecture, puisque les applications exhibent différents niveaux de parallélisme : les pixels d'une même image peuvent être traités en même temps (parallélisme de données) et deux opérateurs peuvent être exécutés en parallèle (parallélisme de tâches). De nombreuses bibliothèques logicielles ont été développées, afin de faciliter la conception d'applications de traitement d'images, mais un travail de portage est souvent nécessaire pour cibler les architectures modernes.

Nous nous focalisons sur un cas particulier du traitement d'images, au travers de la morphologie mathématique [108, 44], et d'une architecture matérielle manycore spécifique, celle du processeur MPPA [7, 43] de Kalray. Le MPPA est, en effet, un nouvel accélérateur matériel exhibant un grand potentiel en puissance de calcul, grâce à ses 256 cœurs, puissance cependant très complexe à exploiter, à cause des différents niveaux de parallélisme qui le composent. Cependant, les applications de traitement d'images présentent également des opportunités de parallélisation, et sont compatibles avec les multiples niveaux de parallélisme du MPPA.

Cette thèse se propose de compiler efficacement, donc de résoudre le problème des *3P* — programmabilité, portabilité et performance —, en se focalisant sur des applications de traitement d'images par morphologie mathématique exprimées dans des langages de haut niveau, et en automatisant les portages au travers de techniques de compilation modernes, en particulier à destination d'accélérateurs matériels, notamment le processeur manycore MPPA sur lequel nous validons notre approche.

1.6 Contributions

Nous avons prouvé par construction la faisabilité de chaînes de compilation assurant programmabilité, portabilité et performance pour des applications de traitement d'images exécutées sur des accélérateurs matériels. Nous avons en effet développé durant cette thèse un ensemble de compilateurs, bibliothèques et environnements d'exécution, afin de porter des applications de ce domaine sur ces cibles. Ces différents outils, associés à un jeu de sept applications de traitement d'images, que nous avons en partie élaboré, nous permettent de réaliser des expériences, qui valident les performances de notre approche.

Nous avons ainsi conçu le compilateur `smiltofreia`, qui convertit des applications d'un DSL de haut niveau embarqué dans Python vers un DSL de plus bas niveau, qui s'appuie sur C et qui peut s'exécuter sur plusieurs cibles matérielles. Cela permet de combiner la programmabilité offerte par le DSL de haut niveau à la portabilité du DSL cible. Nous montrons que notre solution permet d'écrire des applications plus facilement et ne provoque pas de perte de performance.

Nos chaînes de compilation ciblent plus particulièrement le processeur manycore MPPA, qui présente différents niveaux de parallélisme. Ces derniers peuvent s'exploiter suivant

trois modèles de programmation :

— un modèle à mémoire partagée pour les nœuds de calcul, associé ou non à un modèle de communication pour le réseau sur puce,
— un modèle flot de données et
— un modèle pour architectures hétérogènes.

Pour chaque modèle, nous avons développé ou étendu des briques logicielles, grâce auxquelles nous avons pu tester nos sept applications de traitement d'images. Nous avons mesuré les temps d'exécution de nos applications sur le MPPA et sur d'autres processeurs. Nous montrons ainsi que le MPPA est, pour chacun des modèles testés, une des cibles matérielles les plus économes en énergie.

Une seconde génération de processeurs MPPA a été mise sur le marché au cours de cette thèse. Nous avons ainsi pu comparer cette nouvelle génération à la précédente, sur les modèles de programmation concernés, et constater les améliorations en matière de performance.

Grâce à nos expériences, nous pouvons réaliser une synthèse comparative des modèles de programmation étudiés, afin de déterminer le ou les modèles les plus performants. Nous montrons que le modèle flot de données, bien que non supporté sur le MPPA de seconde génération, est le modèle le plus efficace énergétiquement.

Les outils développés pendant cette thèse s'intègrent dans un environnement logiciel global, nommé `freiacc`, qui permet de compiler des applications de traitement d'images écrites dans un DSL de haut niveau vers divers accélérateurs matériels, dont le processeur MPPA.

1.7 Structure de la thèse

Cette thèse est découpée en huit chapitres — en sus de la présente introduction.

Le chapitre 2 et le chapitre 3 exposent les problématiques inhérentes aux domaines des architectures matérielles et du traitement d'images, à l'intersection desquels se placent nos travaux. Ces deux chapitres détaillent également, pour l'un, l'architecture du processeur MPPA et, pour l'autre, la théorie de la morphologie mathématique, cas d'usage de cette thèse, ainsi que deux bibliothèques logicielles qui implémentent les opérateurs issus de cette théorie, SMIL et FREIA.

Le chapitre 4 est consacré au compilateur de langages spécifiques `smiltofreia`, qui concilie la programmabilité d'un langage de haut niveau à la performance et la portabilité d'un environnement logiciel spécifique aux accélérateurs matériels.

Le chapitre 5 est principalement dédié au portage de la bibliothèque SMIL sur un nœud de calcul du MPPA, facilité par la prise en charge native par SMIL du modèle OpenMP pour les architectures à mémoire partagée.

Dans le chapitre 6, nous nous focalisons sur le modèle flot de données et, plus spécifiquement, sur le langage Sigma-C développé en parallèle du MPPA.

Le chapitre 7 décrit le portage sur le processeur MPPA de la cible OpenCL préexistante de FREIA.

Nous rassemblons, dans le chapitre 8, les résultats des développements et expérimentations effectués dans les précédents chapitres, afin de déterminer le modèle de programmation le plus adapté au MPPA.

Enfin, nous concluons, au chapitre 9, en résumant l'ensemble de nos travaux. Nous résumons finalement les progrès restant à accomplir, à savoir l'utilisation en parallèle de tous les clusters de calcul du MPPA avec OpenMP et des communications inter-cluster, ainsi que le port d'un algorithme de segmentation sur le MPPA, algorithme qui sera bientôt intégré à SMIL.

Chapitre 2

Évolutions matérielles et modèles de programmation

Votre âme est un paysage choisi
Que vont charmant masques et bergamasques
Jouant du luth et dansant et quasi
Tristes sous leurs déguisements fantasques.

Tout en chantant sur le mode mineur
L'amour vainqueur et la vie opportune
Ils n'ont pas l'air de croire à leur bonheur
Et leur chanson se mêle au clair de lune,

Au calme clair de lune triste et beau,
Qui fait rêver les oiseaux dans les arbres
Et sangloter d'extase les jets d'eau,
Les grands jets d'eau sveltes parmi les marbres.

— Paul Verlaine*, Clair de lune*

Malgré une évolution continuelle, les processeurs conventionnels se sont avérés trop généraux pour certains calculs spécialisés, ce qui a notamment conduit à l'introduction des processeurs graphiques, aujourd'hui très répandus. Pourtant, à l'ère du *Big Data* et de l'Internet des objets, ces architectures matérielles éprouvées atteignent leurs limites, et de nouvelles architectures innovantes voient le jour. Les processeurs manycore tentent de combiner le parallélisme offert par les processeurs graphiques aux fonctionnalités des processeurs centraux. Leurs concepteurs offrent le support de modèles de programmation, des interfaces qui permettent de pallier la complexité de ces nouvelles architectures. Ces modèles promettent une relative portabilité sur des architectures semblables, au prix d'une complexité de programmation accrue par rapport à un code séquentiel traditionnel. Les performances offertes par ces modèles sont souvent dépendantes des machines cibles.

La section 2.1 est consacrée à l'évolution de ces architectures matérielles et présente les innovations récentes que sont les processeurs manycore. En section 2.2, nous présentons un panorama des différents modèles de programmation associés à ces architectures.

2.1 Des architectures innovantes

Devant la complexité du développement de nouveaux processeurs causée par les barrières physiques, les fabricants innovent en concevant de nouvelles architectures moins performantes mais plus efficaces, et les constructeurs associent plusieurs architectures pour en tirer les bénéfices. Aujourd'hui, trois paradigmes d'architectures matérielles prédominent :

les architectures à mémoire partagée disposent plusieurs unités de calcul autour d'une mémoire unique et accessible par tous ;

les architectures à mémoire distribuée proposent des unités de calcul qui possèdent une mémoire propre et où accéder à la mémoire des autres unités dépend de la topologie du réseau ;

les architectures hétérogènes associent des types d'unités de calcul différents, par exemple un processeur central couplé à un accélérateur graphique disposant d'une mémoire propre.

Ces différents paradigmes peuvent être employés dans la même machine : un nœud de calcul dans un superordinateur (mémoire distribuée) peut se composer d'un processeur principal à plusieurs cœurs (mémoire partagée) pouvant délocaliser ses calculs à un accélérateur dédié (architecture hétérogène). L'utilisation de ces paradigmes est liée à l'évolution des architectures matérielles.

2.1.1 Les processeurs multicœurs et l'évolution vers la mobilité

Le processeur central a longtemps fait figure de principale unité de calcul des ordinateurs. Il a, au cours des années, été l'objet de différentes innovations et évolutions qui ont optimisé son fonctionnement : exécution dans le désordre, unités de calcul vectorielles ou prédiction de branches. À la suite des progrès technologiques, la fréquence des calculs des processeurs a régulièrement augmenté, jusqu'au moment où la dissipation thermique par effet Joule est devenue trop importante et le refroidissement, trop coûteux. Dans l'impossibilité d'augmenter davantage cette fréquence, les concepteurs de processeurs ont dupliqué leurs unités de calcul, introduisant de fait les processeurs multicœurs. Cette multiplication des cœurs a été rendue possible par l'évolution continue des techniques de gravure, qui augmentent le nombre de transistors sur une même surface. Cependant, les programmes et applications qui s'exécutaient sur un processeur à un seul cœur doivent être adaptés à ce nouveau paradigme, et cette tâche nécessite parfois de réécrire de larges portions des applications.

Seconde évolution notable, l'augmentation de l'efficacité énergétique a permis le développement des ordinateurs portables, puis des smartphones et, désormais, des tablettes tactiles. Les innovations récentes consistent à associer deux unités de calcul de différentes performance et consommation énergétique. Ainsi, l'architecture *big.LITTLE* du fabricant de processeur ARM [6] associe un cœur à faible consommation énergétique et un second cœur dédié aux applications calculatoires, à destination des smartphones. Par défaut, c'est le cœur *LITTLE* qui exécute les applications de base ; le cœur *big* n'est démarré qu'à la demande, de sorte à minimiser les dépenses énergétiques.

2.1.2 Les processeurs graphiques : toujours plus de complexité

Les processeurs graphiques, introduits dans les années 1990, sont composés de centaines, voire de milliers, d'unités de calcul très basiques. Ils sont donc très efficaces pour des calculs répétitifs, par exemple des traitements graphiques. De nouvelles interfaces de programmation

et de nouvelles capacités, comme des unités de calcul à virgule flottante double précision, en ont fait des accélérateurs matériels de choix pour les calculs scientifiques.

Cependant, les processeurs graphiques souffrent de leur grande consommation énergétique par rapport à des processeurs standards et de la complexité de leur programmation. Ceux-ci ne peuvent s'utiliser, en effet, que dans le cadre d'une architecture hétérogène dans laquelle un processeur central déporte les données et délègue les calculs à l'accélérateur.

2.1.3 L'avènement des processeurs manycore

Créés afin de concurrencer les processeurs graphiques, les processeurs manycore sont l'évolution naturelle des processeurs multicœurs. Ils comportent de plusieurs dizaines à plusieurs centaines de cœurs de calcul reliés par des réseaux sur puce à faible latence. Ces cœurs sont cependant plus performants que ceux d'un processeur graphique, ce qui leur permet d'exécuter des calculs plus complexes.

Nous décrivons plus bas deux exemples de tels processeurs : le Xeon Phi d'Intel et le MPPA Manycore de Kalray, qui joue un rôle majeur dans cette thèse.

L'Intel Xeon Phi

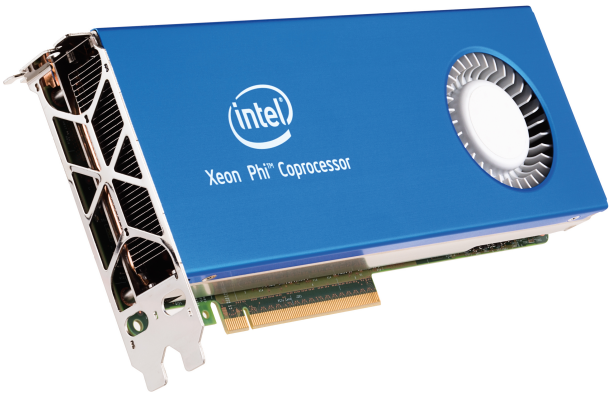


FIGURE 2.1 – Le processeur manycore Intel Xeon Phi

La gamme Xeon Phi [74] du fondeur américain Intel se compose de processeurs manycore comptant entre 60 et 70 cœurs par puce. Ces processeurs sont généralement utilisés en tant qu'accélérateur au côté d'un processeur central classique. Ainsi, le superordinateur chinois Tianhe-2, qui a tenu la tête du TOP500 [117] de 2013 à 2015, utilise des coprocesseurs Xeon Phi.

Ces processeurs consomment en général moins qu'un processeur graphique, mais comportent en contrepartie moins de cœurs. Développer des applications sur ces processeurs est aussi moins complexe, puisqu'Intel fournit une suite logicielle complète et supporte les modèles de programmation standards pour architectures à mémoire partagée et architectures à mémoire distribuée. Par conséquent toute application s'exécutant sur un processeur central Intel peut également s'exécuter sur un Xeon Phi sans modification de code. Un travail d'optimisation des performances reste toutefois nécessaire.

La soixantaine de cœurs que comporte un Xeon Phi peuvent exécuter simultanément jusqu'à quatre *threads* et disposent d'une unité de calcul vectorielle de 512 bits. Ils sont connectés à un réseau d'interconnexion en anneau, qui transfère les données entre les cœurs,

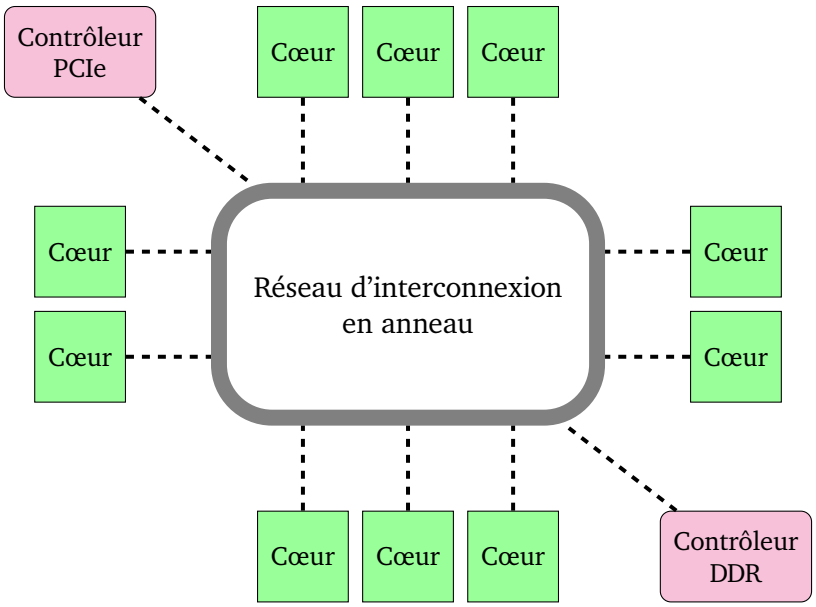


Figure 2.2 – L'architecture du processeur Xeon Phi

la mémoire embarquée — pouvant atteindre plusieurs centaines de Go — et le processeur hôte. Ce processeur peut consommer jusqu'à plus de 200 W. La figure 2.2 exhibe ainsi de façon schématique l'architecture des processeurs Xeon Phi.

Le MPPA Manycore de Kalray

Le processeur MPPA Manycore, conçu par la société française Kalray, expose un fort parallélisme grâce à ses 256 cœurs. Ce processeur, dont la première version a été mise sur le marché en 2013, se différencie par sa relativement faible consommation énergétique, qui s'élève à environ 10 W. De par ses caractéristiques, ce processeur est principalement destiné à des usages hautes performances et/ou embarqués.

Les 256 cœurs de calcul du MPPA sont répartis dans seize clusters de calcul, qui communiquent les uns aux autres à travers deux réseaux sur puce toriques, le premier étant dédié aux transferts de données avec une forte bande passante, le second, aux communications avec une faible latence. Quatre clusters supplémentaires, situés en périphérie de la puce, gèrent deux à deux les interfaces d'entrées/sorties. Deux de ces clusters sont ainsi consacrés aux interfaces PCI-Express, par exemple pour communiquer avec un processeur hôte, et DDR, afin d'accéder à une mémoire globale attachée, tandis que les deux autres sont dédiés aux interfaces réseau Ethernet et Interlaken [113], cette dernière permettant de connecter efficacement plusieurs processeurs MPPA entre eux. La figure 2.3 schématise ainsi l'agencement spatial global de cette puce.

Chaque cluster de calcul comporte seize cœurs, plus un dix-septième, nommé *System Core* ou *Resource Manager* selon les cas. Celui-ci exécute un système d'exploitation basique fournissant une interface pour les communications avec le réseau sur puce, ainsi que pour la gestion des processus sur les différents cœurs. Ces cœurs sont caractérisés par leur architecture VLIW (« Very Long Instruction Word »), qui leur permet d'exécuter plusieurs instructions simultanément.

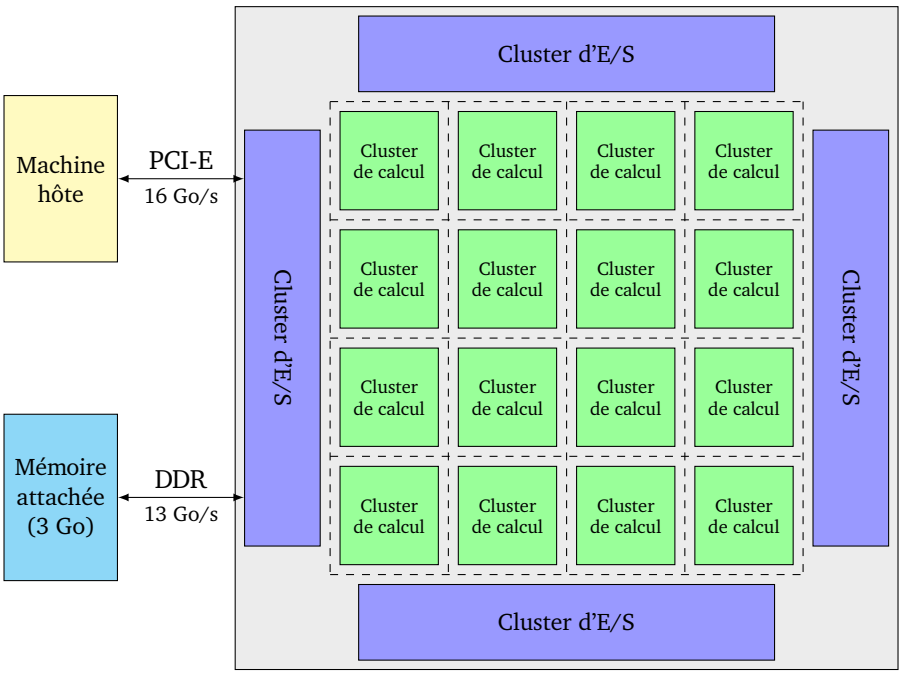


FIGURE 2.3 – Le processeur MPPA-256

Chaque cluster de calcul dispose d'une mémoire locale d'une capacité de 2 Mo, partagée entre les seize cœurs. Cette mémoire sert également pour le système d'exploitation du dix-septième cœur, le *Resource Manager*, ainsi que pour le code binaire des applications, réduisant de fait la capacité disponible pour les données. Le diagramme de la figure 2.4 résume l'architecture d'un cluster de calcul.

Un cluster de calcul s'apparente donc à une architecture à mémoire partagée, tandis qu'à l'échelle de la puce entière, chaque cluster correspond à un nœud d'un système distribué. Enfin, la puce MPPA peut s'utiliser comme accélérateur matériel d'un processeur central, formant ainsi un système hétérogène.

Le processeur MPPA en est aujourd'hui à sa deuxième génération, nom de code « Bostan ». Cette génération est disponible depuis le début 2016 et apporte des gains de performance substantiels par rapport à la première génération, « Andey ». Par exemple, la fréquence d'horloge augmente et passe de 400 MHz à 600 MHz.

Afin de développer des applications sur cette puce, Kalray fournit une suite logicielle appelée MPPA AccessCore, fondée sur l'environnement de développement Eclipse [50] et sur le compilateur GCC [109]. De nouvelles fonctionnalités apparaissent régulièrement dans cette suite logicielle, afin d'améliorer la compatibilité avec les modèles de programmation supportés, ainsi que les performances globales.

Le processeur MPPA de Kalray joue un rôle majeur dans cette thèse : c'est sur cette plateforme complexe que seront portées et exécutées plusieurs applications de traitement d'images, pour en tester les capacités.

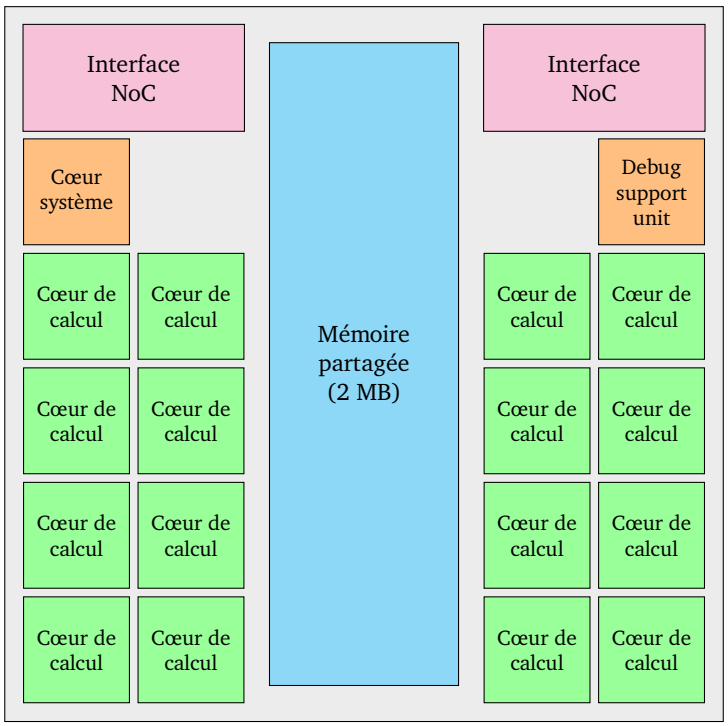


Figure 2.4 – Un des seize clusters de calcul du MPPA-256

2.2 Modèles de programmation et architectures matérielles

Afin de faciliter le développement d'applications performantes et efficaces sur ces nouvelles architectures, différents modèles de programmations ont été conçus. Ceux-ci permettent de cibler d'un coup, si pris en charge, toute une catégorie d'architectures matérielles.

2.2.1 Processeurs multicœurs et programmation parallèle explicite

Les processeurs multicœurs ont permis l'avènement des architectures à mémoire partagée. De nouveaux concepts, comme la programmation par fils d'exécution, ou *threads*, ont émergé, les POSIX Threads [66] en étant l'actuelle référence. Ces modèles de programmation nécessitent toutefois de repenser de façon extensive le fonctionnement des applications ou des bibliothèques.

Pour faciliter la parallélisation des boucles de calcul, la programmation par directives, dont OpenMP [19] est le meilleur représentant, a permis de tirer parti des processeurs multicœurs à moindre frais, tout en conservant le fonctionnement historique sur un seul cœur. La directive placée en tête de boucle dans l'extrait 2.1 demande ainsi l'exécution parallèle de celle-ci.

Le chapitre 5 traitera plus en détails des architectures à mémoire partagée, ainsi que du modèle de programmation OpenMP.

```
12 #pragma omp parallel for
13   for (i = 0; i < N; i++)
14     c[i] = a[i] + b[i];
```

Extrait 2.1 – Addition de vecteurs parallélisée en OpenMP

2.2.2 Passage de messages et mémoire unifiée dans les architectures distribuées

Les superordinateurs actuels sont composés de dizaines de milliers de nœuds de calcul communiquant par un réseau d'interconnexion. Pour communiquer à travers ce réseau, le modèle de programmation par passage de messages permet d'abstraire les communications spécifiques au matériel sous une interface commune. Cependant, les programmes doivent être conçus à l'origine pour tirer parti de ce modèle de programmation, bien que des techniques de génération automatique de code distribué soient aujourd'hui à l'étude [68, 69, 87]. La solution industrielle de référence pour le passage de messages est sans conteste l'interface MPI [49] (« *Message Passing Interface* »), Une interface de programmation par passage de messages est décrite dans le chapitre 9, afin de communiquer entre plusieurs clusters du processeurs MPPA de Kalray.

PGAS [97] est un autre modèle de programmation pour les architectures distribuées. Celui-ci consiste à abstraire les transferts de données et de commandes et à fournir une vision « unifiée » de la mémoire.

2.2.3 Le flot de données : le parallélisme de tâches explicite

Le modèle flot de données, contrairement aux modèles de programmation pré-cités, n'est pas spécifiquement lié à un type d'architecture, mais plutôt à certaines classes d'applications. Il consiste à diviser une application en différentes tâches qui consomment et produisent des données de granularité prédéfinie. Les applications flot de données fournissent certaines assurances quant à la sûreté d'exécution, surtout si les quantités de données échangées entre tâches sont constantes [86]. C'est pourquoi ce modèle est assez utilisé dans un contexte industriel.

Une application prend donc la forme d'une chaîne de ces tâches, ce qui correspond à un pipeline pouvant s'exécuter en parallèle. Des constructions spécifiques permettent de diviser et de fusionner les données, ce qui génère ainsi du parallélisme de données. Un graphe flot de données typique est présenté dans la figure 2.5.

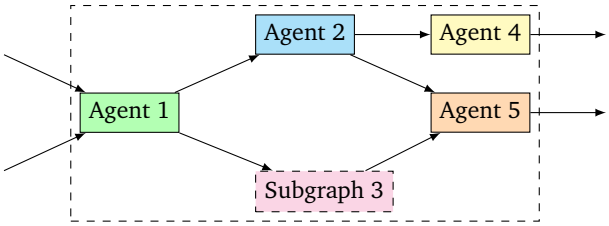


Figure 2.5 – Graphe flot de données

Ce modèle sera étudié de façon plus extensive dans le chapitre 6.

2.2.4 La délocalisation des calculs dans les architectures hétérogènes

Les architectures hétérogènes sont généralement constituées d'un processeur central relié à un ou plusieurs accélérateurs matériels dédiés disposant de leur propre mémoire. Les processeurs graphiques appartiennent à cette catégorie. Des modèles de programmation ont vu le jour pour cibler efficacement ces accélérateurs. OpenCL [60] ou CUDA [37] en sont les figures de proue. Ces modèles bas niveau définissent des noyaux de calcul qui seront exécutés par le ou les accélérateurs, tandis que le processeur central, aussi appelé processeur hôte, se charge des transferts de données et de l'ordonnancement de ces noyaux de calcul. Le processeur principal peut, en parallèle, se charger d'une partie des calculs, mais l'équilibrage de ceux-ci reste encore complexe. Le chapitre 7 est dédié à l'utilisation d'OpenCL pour délocaliser les calculs sur l'accélérateur manycore de Kalray.

2.2.5 Les interfaces de haut niveau : le confort au détriment de la flexibilité ?

Les modèles de programmation proposés ci-dessus nécessitent en général des connaissances pointues des architectures ciblées et de la parallélisation des calculs. De plus, ces modèles de programmation sont en général liés et dédiés à un type d'architecture ; les utiliser dans un autre contexte est inefficace.

Depuis plusieurs années, des modèles de haut niveau tentent de percer afin de faciliter la prise en charge des architectures modernes. Parmi ces nouveaux modèles, OpenACC [94] propose des directives préprocesseur afin de délocaliser les calculs sur accélérateur dans le cadre d'une architecture hétérogène. La directive en tête de boucle dans l'extrait 2.2 permet de déporter automatiquement l'exécution de celle-ci sur tout accélérateur respectant la spécification. Similairement, la version 4 d'OpenMP [18], datant de 2013, ajoute des directives à destination des accélérateurs matériels, unifiant de fait les modèles de programmation pour architecture à mémoire partagée et pour architecture hétérogène.

```
11 #pragma acc kernels copyin(a[0 : N], b[0 : N]), copyout(c[0 : N])
12   for (i = 0; i < N; i++)
13     c[i] = a[i] + b[i];
```

Extrait 2.2 – Addition de vecteurs déportée en OpenACC

Ces modèles de haut niveau facilitent la programmation en proposant des constructions plus simples que leurs équivalents de plus bas niveau. Grâce à eux, il est possible, en annotant du code déjà fonctionnel, de le porter à moindre frais sur les accélérateurs supportés. Toutefois, ces interfaces de haut niveau peuvent s'avérer limitées pour représenter la complexité des architectures cibles et, donc, atteindre les performances d'un modèle de plus bas niveau.

2.3 Conclusion

Face aux limites physiques, provoquées par les effets quantiques dès que la finesse de gravure atteint quelques nanomètres, les concepteurs de processeurs augmentent la complexité de leurs architectures. Les architectures modernes comme le processeur manycore MPPA exposent différents niveaux de parallélisme : cœurs VLIW à instructions vectorielles, nœuds de calcul à mémoire partagée, cluster à mémoire distribuée et interconnexion par réseau sur puce et accélérateur en lien avec un hôte.

Pour exploiter ces différents paradigmes, plusieurs modèles de programmation existent. Des modèles de bas niveau comme OpenCL rendent le code portable sur différentes classes d'accélérateurs matériels, mais des optimisations spécifiques au matériel restent nécessaires pour atteindre de bonnes performances. À l'inverse, certains modèles de haut niveau facilitent la programmation et sont portables sur un plus large ensemble d'architectures, généralement aux dépens de la performance.

L'enjeu de cette thèse est de concilier facilité de programmation, portabilité et performance sur l'architecture complexe du processeur MPPA. Nous utilisons, pour ce faire, les différents modèles de programmation compatibles avec cette architecture, à savoir OpenMP sur les nœuds de calcul à mémoire partagée dans le chapitre 5, le langage flot de données Sigma-C dans le chapitre 6 et la spécification OpenCL (chapitre 7).

Chapitre 3

Traitement d'images et bibliothèques logicielles

Le temps irrévocable a fui, l'heure s'achève.
Mais toi, quand tu reviens et traverses mon rêve,
Tes bras sont plus frais que le jour qui se lève,
Tes yeux plus clairs.

A travers le passé ma mémoire t'embrasse.
Te voici. Tu descends en courant la terrasse
Odorante, et tes faibles pas s'embarrassent
Parmi les fleurs.

Par un après-midi de l'automne, au mirage
De ce tremble inconstant que varient les nuages,
Ah ! verrais-je encor se farder ton visage
D'ombre et de soleil ?

— Paul-Jean Toulet, *Souvenirs d'automne*

Comme vu dans le chapitre 1, le traitement d'images est un domaine industriel clé qui compte de nombreuses applications grand public. Les réalités augmentée et virtuelle ont été rendues accessibles au grand public par l'essor des *smartphones*, et les véhicules autonomes ainsi que les drones annoncent des changements sociétaux majeurs. De telles applications s'appuient en général sur des bibliothèques logicielles qui implémentent les différents opérateurs de traitement d'images et permettent de facilement les réutiliser grâce à des interfaces de programmation.

Ce chapitre effectue un survol des avancées actuelles en matière de traitement d'images dans la section 3.1, puis se focalise plus particulièrement sur une branche de ce domaine, la morphologie mathématique, décrite dans la section 3.2. Nous décrivons ensuite, dans la section 3.3, deux bibliothèques de traitement d'images sur lesquelles nous avons fondé les travaux de cette thèse.

3.1 Le traitement d'images aujourd'hui

Le traitement d'images consiste à extraire et transformer l'information d'une image, considérée comme un tableau de pixels en deux dimensions. Des dimensions supplémentaires peuvent toutefois intervenir : les IRM produisent ainsi des images tridimensionnelles, et les flux vidéos peuvent s'interpréter selon une troisième dimension temporelle. Quant à l'espace colorimétrique, il comporte lui aussi généralement trois dimensions : (R, G, B), (L, a, b), ou (X, Y, Z). Ces dimensions supplémentaires multiplient d'autant les calculs. Ainsi, les applications de traitement d'images mettent souvent en jeu un grand volume de données. Toutefois, les calculs sur les pixels étant relativement indépendants, ces calculs sont facilement parallélisables.

Parmi les sous-domaines du traitement d'images, la vision par ordinateur consiste à extraire la sémantique d'une image, en reconnaissant, par exemple, les objets qui la composent, comme illustré en figure 3.1. Différentes méthodes sont aujourd'hui en concurrence. La morphologie mathématique, que nous détaillerons plus bas, suit une approche algorithmique pure et géométrique afin de détecter les formes. L'apprentissage artificiel, particulièrement en vogue actuellement, se base, comme son nom l'indique, sur la recherche de caractéristiques clefs par apprentissage sur une base de données d'images.

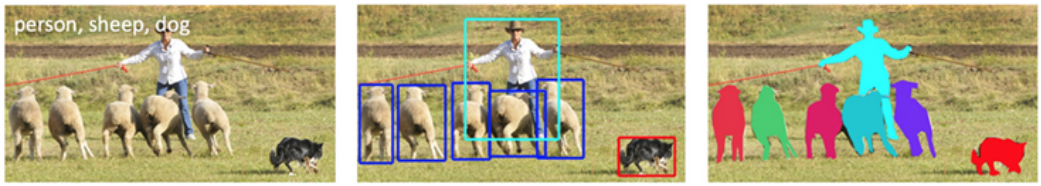


Figure 3.1 – Vision par ordinateur : reconnaissance d'objets dans une image avec détection, localisation et détermination de leurs contours par segmentation

Les produits de ces champs de recherche prennent la forme de bibliothèques logicielles qui rassemblent les implémentations de différents opérateurs sous une interface de programmation facilitant le développement d'applications. Dans le domaine de la vision par ordinateur, la plus connue de ces bibliothèques est OpenCV [95]. Celle-ci propose un large choix de filtres de traitement d'images, dont quelques opérateurs de base de la morphologie mathématique.

Les bibliothèques d'apprentissage artificiel s'appuient sur un principe différent : elles proposent de construire différentes structures apprenantes — arbres de décision, réseaux de neurones — qu'il faudra entraîner sur une base d'apprentissage, puis valider séparément sur une base de validation. Plusieurs bibliothèques d'apprentissage artificiel à sources ouvertes ont été développées par des laboratoires universitaires ; parmi eux Torch [30], Theano [12], Caffe [79] ou Scikit-learn [98, 22].

Depuis 2015, les grands industriels du monde numérique se sont emparés de ces questions et ont libéré les uns après les autres les sources de leurs bibliothèques. C'est d'abord Google, qui a ouvert les vannes avec Tensorflow [88, 1], sa bibliothèque dédiée aux réseaux de neurones profonds. Celle-ci permet de décrire via une interface de haut niveau en Python des graphes de neurones, qui seront ensuite générés en C++ ou en CUDA pour les processeurs graphiques. En janvier 2016, Microsoft a publié CNTK [2], son outil pour l'apprentissage profond, qu'il utilise pour la reconnaissance vocale. En mai de la même année, Amazon a rendu accessible DSSTNE [82], optimisé pour les calculs de matrices creuses et, notamment,

utilisé pour les recommandations sur la base d'articles de la boutique en ligne. Enfin, Facebook a dévoilé en août 2016 les sources de trois outils consacrés à la reconnaissance d'objets dans les images [48]. La mise à disposition de ces outils, outre la publicité positive qu'elle génère, permet de développer une communauté autour du logiciel, à même de créer des applications novatrices dans un domaine qui reste encore complexe.

Cette thèse ne s'intéresse cependant pas à l'apprentissage artificiel, sujet d'avenir s'il en est, mais plutôt à la branche algorithmique traditionnelle du traitement d'images, au travers de la morphologie mathématique.

3.2 La morphologie mathématique, une branche du traitement d'images

Parmi les différentes branches du traitement d'images, la morphologie mathématique est une théorie découverte et étudiée à l'École des mines de Paris depuis les années 1960 [108, 44]. Un centre de recherche de l'école est toujours dédié à étendre cette théorie ; il s'agit du Centre de morphologie mathématique (CMM).

(a) Image initiale (b) Image finale

FIGURE 3.2 – Exemple d'application de la morphologie mathématique : extraction des contours par gradient morphologique

La morphologie mathématique est fondée sur la théorie ensembliste des treillis. Les applications de cette théorie consistent notamment à détecter ou éliminer des structures géométriques dans des images ou bien à segmenter une image en régions. Les applications d'extraction de plaque d'immatriculation et de détection de dégénérescence maculaire présentées plus haut en figure 1.4, figure 1.5 et figure 1.6, ainsi que l'extraction des contours en figure 3.2, sont typiques de la morphologie mathématique. Pour faciliter l'usage des opérateurs de morphologie mathématique, plusieurs bibliothèques entièrement dédiées à cette discipline ont été développées par le CMM, la dernière d'entre elle étant SMIL [46, 47], dont nous reparlerons ci-après.

Les fonctions offertes par ces bibliothèques se décomposent généralement en un ensemble d'opérateurs de base agissant sur les pixels d'une image. Ces opérateurs sont généralement réguliers sur tous les pixels d'une image et sont facilement parallélisables. Ils peuvent se classer en trois grandes catégories.

Les opérateurs pixel-à-pixel effectuent des opérations identiques sur tous les pixels d'une ou plusieurs images de taille identique. Pour une image en entrée, chaque pixel de celle-ci n'est accédé qu'une seule fois. Un exemple est la soustraction de deux images, afin de déterminer les zones qui diffèrent.

Les opérateurs de réduction consistent à extraire des informations scalaires, comme le maximum, le minimum global ou la somme de tous les pixels, à partir d'une image entière.

Les opérateurs de voisinage sont un sur-ensemble des opérateurs pixel-à-pixel et consistent à également prendre en compte la valeur d'un certain nombre de pixels voisins du pixel courant. En comparaison des deux autres catégories, ils demandent plus de calculs par pixel.

Ces opérateurs de voisinage forment le cœur de la morphologie mathématique. Ils sont fondés sur la notion d'*élément structurant* servant à décrire un voisinage de pixels caractérisé par sa forme et sa taille. Sur le voisinage décrit par un élément structurant autour d'un pixel donné, la *dilatation* va produire un pixel dont la valeur est le maximum des pixels du voisinage et l'*érosion*, le minimum. Ainsi, si $\mathcal{V}_k(x)$ désigne le voisinage d'un pixel x déterminé par l'élément structurant k, alors les résultats x_{dil} et x_{er} de la dilatation et l'érosion du pixel x s'expriment respectivement par l'équation 3.1 et l'équation 3.2 :

$$x_{dil} = \max_{p \in \mathcal{V}_k(x)} p \tag{3.1}$$

$$x_{er} = \min_{p \in \mathcal{V}_k(x)} p \tag{3.2}$$

Les opérateurs complexes de la morphologie mathématique, comme la segmentation par ligne de partage des eaux [14], s'appuient notamment sur l'érosion et la dilatation.

3.3 Des bibliothèques pour le traitement d'images

Une implémentation efficace et optimisée des opérateurs de base de la morphologie mathématique sur le matériel moderne est critique dès que l'on cherche à atteindre une performance temps-réel. Les bibliothèques de traitement d'images prennent appui sur des modèles de programmation parallèles afin d'accélérer les calculs. Toutefois, ces bibliothèques sont généralement limitées aux architectures les plus communes, comme les processeurs centraux ou graphiques. D'autres systèmes sont nécessaires afin d'améliorer leur portabilité et leur performance, et c'est là le but du projet FREIA.

3.3.1 FREIA, un framework pour l'analyse d'images

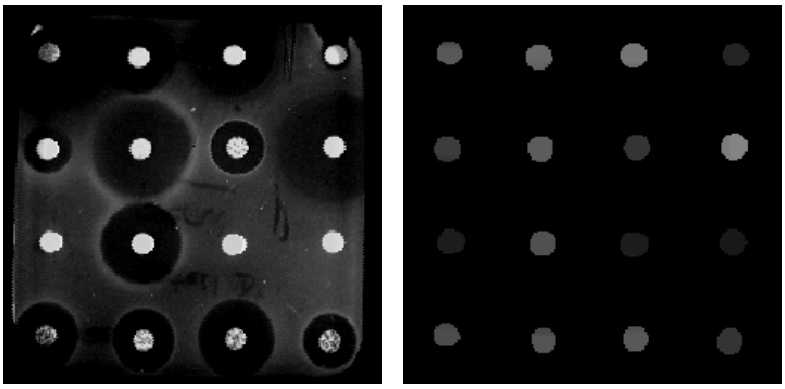
Le projet FREIA [15, 29], qui signifie « *FRamework for Embedded Image Applications* », consiste à fournir une interface de programmation commune à des accélérateurs matériels spécifiques au traitement d'images. Projet financé par l'Agence nationale de la recherche de 2008 à 2011, il réunit deux centres de recherche de l'École des mines de Paris, le Centre de recherche en informatique (CRI) et le Centre de morphologie mathématique (CMM), la grande école Télécom Bretagne et l'entreprise Thalès Research & Technology (TRT). FREIA visait à l'origine les accélérateurs SPoC [27], développé au CMM, et Terapix [20], de Thalès. Il a également gagné le support des processeurs graphiques grâce à l'utilisation du modèle de programmation OpenCL [60].

FREIA fournit une interface dans le langage C pour écrire des applications pouvant, par la suite, être exécutées sur ses cibles. Cette interface fournit des routines de création et de manipulation d'images, ainsi que des opérateurs de traitement d'images. Ces derniers se répartissent en deux sous-interfaces ; l'interface CIPO (« *Complex Image Processing Operators* ») est construite au-dessus de AIPO (« *Atomic Image Processing Operators* »), qui ne fournit que des opérateurs de base. FREIA permet d'écrire des applications de traitement d'images en utilisant uniquement des fonctions de son interface. Ainsi, FREIA s'apparente à un langage spécifique au domaine du traitement d'images, embarqué dans C. Un exemple d'utilisation de FREIA est disponible en extrait 3.1 : il s'agit ici de l'application d'extraction de plaques d'immatriculations décrite plus haut en figure 1.4.

Plusieurs autres applications FREIA ont servi de référence pour les implémentations de l'interface vers les différents accélérateurs cibles. Parmi celles-ci, on retrouve la détection de dégénérescence maculaire présentée en figure 1.5 et l'extraction de plaques d'immatriculations de la figure 1.4, respectivement sous le nom *retina* et *licensePlate*. Une liste des applications utilisées durant cette thèse, avec quelques unes de leurs caractéristiques (lignes de code, nombre et type d'opérateurs, taille d'image typique), est présentée dans le tableau 3.1. Les effets des applications *antibio*, *burner*, *deblocking* et *toggle* sont également représentés, respectivement, dans les figure 3.3, figure 3.4, figure 3.5 et figure 3.6.

Apps.	#LoC	#operateurs				taille d'image
		arith	morpho	red	Total	
anr999	88	1	30	2	33	288×224
antibio	200	276	288	267	831	256×256
burner	510	49	435	34	518	256×256
deblocking	162	26	9	2	37	512×512
licensePlate	202	4	65	0	69	640×383
retina	471	86	147	68	301	256×256
toggle	144	8	6	1	15	512×512

TABLE 3.1 – Liste des applications FREIA de référence



(a) Image initiale (b) Image finale

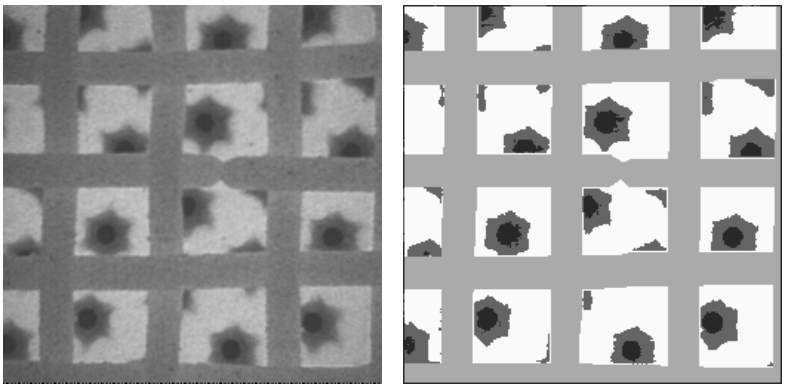
FIGURE 3.3 – Application *antibio* : sélection du meilleur antibiotique

```c
#include "freia.h"

#define N_OPEN 15
#define N_CLOSE 8
#define THR_LOW 50
#define THR_HIGH 150
#define N_CROSS 4

int main(int argc, char *argv[]) {
  freia_dataio fdin, fdout;
  freia_data2d *imin, *imopen, *imclose, *imout;
  const int32_t kernel1x3[9] = { 0, 0, 0, 1, 1, 1, 0, 0, 0 };
  const int32_t kernel3x1[9] = { 0, 1, 0, 0, 1, 0, 0, 1, 0 };
  freia_initialize(argc, argv);
  /* open images flow */
  freia_common_open_input(&fdin, 0);
  uint32_t w = fdin.framewidth, h = fdin.frameheight;
  freia_common_open_output(&fdout, 0, w, h, 8);
  /* images creation */
  imin = freia_common_create_data(16, w, h);
  imopen = freia_common_create_data(16, w, h);
  imclose = freia_common_create_data(16, w, h);
  imout = freia_common_create_data(16, w, h);
  freia_common_rx_image(imin, &fdin); /* read input image */

  /* computation */
  freia_cipo_open_generic_8c(imopen, imin, kernel1x3, N_OPEN);
  freia_cipo_close_generic_8c(imclose, imin, kernel1x3, N_CLOSE);
  freia_aipo_threshold(imopen, imopen, 1, THR_LOW, true);
  freia_aipo_threshold(imclose, imclose, THR_HIGH, 255, true);
  freia_aipo_and(imout, imopen, imclose);
  freia_cipo_open_generic_8c(imout, imout, kernel3x1, N_CROSS);
  freia_cipo_open_generic_8c(imout, imout, kernel1x3, N_CROSS);
  freia_cipo_dilate(imout, imout, 8, 3);
  freia_aipo_and(imout, imout, imin);

  freia_common_tx_image(imout, &fdout); /* save output image */
  /* images destruction */
  freia_common_destruct_data(imin);
  freia_common_destruct_data(imopen);
  freia_common_destruct_data(imclose);
  freia_common_destruct_data(imout);
  /* close images flow */
  freia_common_close_input(&fdin);
  freia_common_close_output(&fdout);
  freia_shutdown();
  return 0;
}
```

Extrait 3.1 – Application licensePlate en FREIA

Les opérateurs fournis par FREIA disposent de plusieurs implémentations ; leur liste est présentée dans le tableau 3.2. SPoC, Terapix, OpenCL et l'implémentation logicielle Fulguro ont été intégrés à FREIA avant le début de cette thèse ; SMIL, Sigma-C et MPPA en sont des produits directs. À noter que l'implémentation Sigma-C n'est pas utilisable d'emblée par les applications FREIA : Sigma-C ne s'interface pas facilement avec C.

Des optimisations génériques ou spécifiques à la cible visée peuvent être appliquées par le



(a) Image initiale (b) Image finale

FIGURE 3.4 – Application *burner* : segmentation d'image de four industriel

(a) Image initiale (b) Image finale

FIGURE 3.5 – Application *deblocking* : atténuation des macro-blocs JPEG

compilateur source-à-source PIPS [41, 75]. Dans une première phase, celui-ci va décomposer les appels vers l'interface CIPO de FREIA en suite d'appels AIPO, propager les constantes et dérouler les boucles. Une représentation intermédiaire sous forme de graphe orienté acyclique va également être générée. Ensuite, des optimisations vont avoir lieu sur cette représentation : élimination des variables inutiles et des sous-expressions communes, afin de réduire le nombre d'opérations exécutées. Dans une dernière étape, du code spécifique pour les différentes cibles (et notamment Sigma-C) peut être généré. Des optimisations spécifiques à chaque cible sont également réalisées durant cette phase : fission du graphe applicatif en sous-graphes, fusion d'opérateurs, etc. Toutes ces optimisations sont détaillées dans [29].

Une fois les applications compilées vers une cible, elles peuvent être exécutées sur les matériels compatibles. Les cibles SPoC et Terapix sont exécutées sur un FPGA Xilinx, tandis que la cible OpenCL a pu l'être sur plusieurs processeurs centraux et graphiques, ainsi que sur le processeur manycore MPPA de Kalray. Les cibles utilisées dans le projet FREIA et également durant cette thèse sont ainsi listées dans le tableau 3.3.

(a) Image initiale (b) Image finale

Figure 3.6 – Application *toggle* : amélioration de contraste

Cibles logicielles	Type de matériel
SPoC	FPGA
Terapix	FPGA
OpenCL	GPU, CPU
Fulguro	CPU
SMIL	CPU
Sigma-C	CPU, Manycore
MPPA	Manycore

Table 3.2 – Liste des cibles logicielles de FREIA et le type de matériel visé

Matériels	Type	Cibles logicielles	W
Xilinx Virtex-5	FPGA	SPoC, Terapix	26
Nvidia GeForce 8800 GTX	GPU	OpenCL	120
Nvidia Quadro 600	GPU	OpenCL	40
Nvidia Tesla C 2050	GPU	OpenCL	240
Intel dual-core	CPU	OpenCL	65
AMD quad-core	CPU	OpenCL	60
Intel Core i7-3820	CPU	SMIL, Fulguro, Sigma-C	130
Kalray MPPA-256	Manycore	Sigma-C, OpenCL, MPPA	10

Table 3.3 – Liste des accélérateurs matériels utilisés dans le projet FREIA et dans cette thèse

Parmi les cibles logicielles de FREIA, la bibliothèque SMIL fait figure de cas à part et mérite, de fait, sa propre sous-section.

3.3.2 SMIL, une bibliothèque moderne d'analyse d'images

SMIL [46, 47], acronyme de « *Simple (but efficient) Morphological Image Library* », est une bibliothèque logicielle de traitement d'images et, plus spécifiquement, de morphologie mathématique. Développée par le Centre de recherche en morphologie mathématique de

l'École des mines depuis 2011, elle vise à fournir des performances temps-réel avec des gros volumes de données et à traiter des images à la fois en deux et trois dimensions, par exemple IRM ou issues de microscopes électroniques, tout en étant facile à développer, à maintenir et à intégrer à des projets tiers.

Écrite dans le langage de programmation C++, cette bibliothèque s'appuie sur les dernières avancées des compilateurs modernes tels GCC ou LLVM, afin de s'exécuter rapidement sur les processeurs actuels. Ceux-ci permettent, en effet, de générer automatiquement des versions vectorielles optimisées de certaines boucles de calcul. Pour profiter du parallélisme offert par les architectures multicœurs, SMIL s'appuie également sur la spécification OpenMP [19], qui permet de paralléliser les boucles de calcul sur les différents cœurs. L'utilisation de C++ comme langage de base permet de garder le contrôle sur les ressources mémoire mobilisées et de fournir potentiellement des performances supérieures aux langages usuels (à l'exception du C et de l'assembleur). Au contraire du C dont il est issu, C++ offre plusieurs fonctionnalités de haut niveau comme la programmation orientée objet, la métaprogrammation au travers des templates, les fonctions lambda ou l'inférence de type qui permettent d'accélérer la production et de favoriser la maintenance du code sans perdre en performances. SMIL s'appuie ainsi sur ces fonctionnalités pour alléger le développement et factoriser les portions de code calculatoires, qui pourront plus facilement être optimisées.

Pour favoriser la programmation d'applications, SMIL utilise Swig [112] (« *Simple Wrapper and Interface Generator* »), afin de générer des interfaces pour des langages de plus haut niveau. En effet, C++ reste un langage statique et de bas niveau, malcommode à utiliser en comparaison d'autres langages dynamiques comme Python, qui possède une syntaxe plus simple et plus expressive. De plus, les développeurs d'applications de traitement d'images utilisent majoritairement des langages de haut niveau comme Python ou MATLAB, qui disposent chacun d'un écosystème vaste et pouvant facilement être intégré avec d'autres solutions. C'est ainsi que, grâce à Swig, SMIL propose des interfaces pour les langages Python, Ruby, GNU Octave et Java.

Pour aisément compiler SMIL, le logiciel CMake permet de détecter la présence des dépendances et d'activer automatiquement ou à la demande certaines fonctionnalités présentes dans la bibliothèque, comme le support d'OpenMP ou l'auto-vectorisation des boucles de calcul. CMake offre également le support de la compilation croisée, c'est-à-dire que l'on peut compiler SMIL pour un autre système, si la chaîne de compilation spécifique est présente.

SMIL fait aussi appel à d'autres outils pour faciliter son développement. Par exemple, le logiciel Git [52] est utilisé pour gérer le code source de SMIL de façon décentralisée. Quant au logiciel Doxygen [45], il permet de générer automatiquement de la documentation au format HTML ou LaTeX à partir de commentaires présents dans le code source.

Ces briques logicielles, sur lesquelles s'appuie SMIL, sont des outils éprouvés, à sources ouvertes, et qui jouent un grand rôle dans le développement d'applications dans le monde du logiciel libre. Un bref résumé du rôle de chacune d'entre elles est disponible dans le tableau 3.4. SMIL est également publiée sous licence ouverte, ce qui autorise la réutilisation libre de ses sources.

Les développements récents de SMIL intègrent les dernières avancées du traitement d'images par morphologie mathématique et, notamment, une implémentation distribuée de l'algorithme de segmentation par ligne de partage des eaux [23].

Un exemple d'utilisation de SMIL est présenté en extrait 3.2, ici à travers l'interface Python. Celui-ci reprend l'application d'extraction de plaque d'immatriculation présenté en figure 1.4. Par rapport à la version FREIA, dont le code est disponible via l'extrait 3.1, le présent code est de moitié plus court et profite des fonctionnalités de Python concernant la surcharge d'opérateur et la gestion mémoire par ramasse-miettes.

Logiciels	Classe	Bénéfices
C++	Performance	Programmation orientée objet bas niveau
OpenMP	Performance	Parallélisation des boucles de calculs
GCC/LLVM	Performance	Auto-vectorisation des boucles de calcul
CMake	Portabilité	Compilation conditionnelle, croisée
Swig	Programmabilité	Génération d'interfaces haut niveau
Doxygen	Programmabilité	Génération de documentation
Git	Maintenabilité	Gestion des sources décentralisé

Table 3.4 – Briques logicielles utilisées dans SMIL

```python
import smilPython as smil

N_OPEN = 15
N_CLOSE = 8
THR_LOW = 50
THR_HIGH = 150
N_CROSS = 4

imin = smil.Image()             # declare some image variables
imopen = smil.Image()
imclose = smil.Image()

imin.load("plate2.pgm")         # read input image

smil.open(imin, imopen, smil.HorizSE(N_OPEN))
smil.close(imin, imclose, smil.HorizSE(N_CLOSE))
smil.threshold(imopen, 1, THR_LOW, 255, 0, imopen)
smil.threshold(imclose, THR_HIGH, 255, 255, 0, imclose)
imand = imopen & imclose

smil.open(imand, imand, smil.VertSE(N_CROSS))
smil.open(imand, imand, smil.HorizSE(N_CROSS))
smil.dilate(imand, imand, smil.SquSE(3))
imout = imin & imand

imout.save("video_out_0/plate-out.pgm")  # save output image
```

Extrait 3.2 – Application licensePlate en SMIL Python

3.3.3 Des ponts entre SMIL et FREIA

Dans le cadre de cette thèse, deux ponts ont été créés entre SMIL et FREIA.

Le premier d'entre eux a consisté à faire de SMIL une cible de FREIA, de sorte à disposer d'une implémentation logicielle de référence pour FREIA, capable de tirer parti des processeurs modernes. Il a fallu rediriger tous les appels aux opérateurs FREIA vers leurs équivalents SMIL. Pour ce faire, il a fallu développer d'abord une interface C autour de SMIL, que nous avons appelée `smilc` [121]. En effet, un code C ne peut directement appeler des fonctions C++, car le compilateur C++ utilise des conventions de nommage différentes, ce qui pose problème lors de l'édition de liens. Pour contourner ce problème, il faut définir, en C++, une interface C autour des fonctions désirées [89]. L'utilisation dans cette interface de structures de données opaques permet de manipuler des objets C++ par des pointeurs C.

Grâce à ce premier pont, nous pouvons exécuter nos applications FREIA au-dessus de

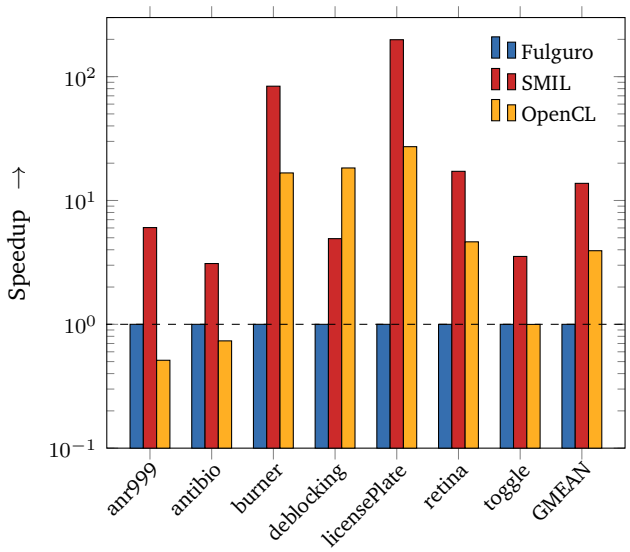


FIGURE 3.7 – Speedups (échelle logarithmique) des applications FREIA exécutées sur Intel Core i7-3820 selon les implémentations : (1) Fulguro (référence) ; (2) SMIL ; (3) OpenCL

SMIL. Nous avons ainsi comparé l'exécution de ces applications via SMIL, via la seconde implémentation logicielle, Fulguro, qui date de 2008, et via OpenCL. Les applications sont exécutées sur un processeur Intel Core i7-3820 à quatre cœurs et huit threads, avec extensions vectorielles AVX. Les rapports entre ces temps d'exécution, que nous exposons en figure 3.7, montrent un net avantage pour SMIL, qui se révèle de trois à cent fois plus rapide que son concurrent Fulguro et, en moyenne, trois fois plus rapide qu'OpenCL. Les applications qui disposent de longues chaînes d'opérateurs morphologiques de voisinage, comme *burner*, *licensePlate* et, dans une moindre mesure, *retina*, sont particulièrement avantagées. En outre, et contrairement à SMIL et OpenCL, Fulguro n'a pas été conçu pour supporter les processeurs multicœurs et s'exécute sur un unique cœur de calcul. L'implémentation OpenCL est, quant à elle, moins adaptée aux processeurs centraux qu'aux processeurs graphiques, mais se montre quatre fois plus rapide que Fulguro sur la même architecture.

Lier SMIL à FREIA de la sorte permet donc d'améliorer significativement les performances des applications FREIA sur les processeurs modernes, sans effort supplémentaire de portabilité, puisque le code des applications n'a pas été touché.

Le second de ces ponts fera l'objet du chapitre 4 et consistera à convertir une application SMIL en son équivalent FREIA, de sorte à pouvoir s'exécuter sur toutes ses cibles et, notamment, sur le processeur manycore MPPA de Kalray.

3.4 Conclusion

Ce chapitre se concentre sur le traitement d'images et, plus particulièrement, sur la branche de la morphologie mathématique. Sont introduites une bibliothèque logicielle, SMIL et une interface de programmation, FREIA, dédiées à ce domaine.

FREIA dispose de sept implémentations dont quatre sont antérieures à cette thèse — les trois autres en sont des contributions directes. Ces implémentations rendent les applications FREIA portables sur les matériels ciblés. L'utilisation d'optimisations source-à-source

permet d'améliorer les performances, par exemple en fusionnant certains opérateurs. Sept applications FREIA sont présentées ; elles serviront plusieurs fois par la suite.

SMIL est une bibliothèque de traitement d'images optimisée pour les processeurs multicœurs et vectoriels modernes. Cependant, SMIL ne peut pas tirer parti des processeurs graphiques. L'usage d'interfaces de haut niveau, par exemple dans le langage Python, facilite le développement d'applications SMIL.

Nous avons créé un pont entre FREIA et SMIL en dirigeant les appels vers les opérateurs de traitement d'images de FREIA vers leurs équivalents SMIL. Il a fallu pour cela concevoir une interface C autour de SMIL. Nos applications FREIA sont ainsi en mesure de s'exécuter efficacement sur les processeurs modernes.

Chapitre 4

Compilation d'un langage dynamique vers un langage statique

Quand nous en serons au temps des cerises,
Et gai rossignol et merle moqueur
Seront tous en fête.
Les belles auront la folie en tête
Et les amoureux du soleil au cœur.
Quand nous en serons au temps des cerises,
Sifflera bien mieux le merle moqueur.

Mais il est bien court, le temps des cerises,
Où l'on s'en va deux cueillir en rêvant
Des pendants d'oreilles.
Cerises d'amour aux robes pareilles
Tombant sous la feuille en gouttes de sang.
Mais il est bien court le temps des cerises,
Pendants de corail qu'on cueille en rêvant.

J'aimerai toujours le temps des cerises :
C'est de ce temps-là que je garde au coeur
Une plaie ouverte.

— Jean-Baptiste Clément, *Le temps des cerises (extr.)*

Ce chapitre est consacré à la question de la programmabilité et de la portabilité des applications, notamment dans le cadre du traitement d'images. Il a fait l'objet d'une présentation orale à l'édition 2016 de la conférence *Compilers for Parallel Computing* qui s'est déroulée en juillet 2016 à Valladolid en Espagne [123]. Nous verrons spécifiquement comment concilier les atouts des deux bibliothèques SMIL et FREIA, afin de profiter de l'interface de programmation de l'une et des cibles matérielles de la seconde, tout en conservant les performances des applications. De cette façon, nous améliorons la portabilité des applications écrites en SMIL : elle peuvent désormais s'exécuter sur l'ensemble des cibles matérielles de FREIA. Avec l'interface de haut niveau de SMIL comme point d'entrée de la chaîne de compilation FREIA, nous en améliorons également la programmabilité.

Nous rappelons, en section 4.1, les enjeux du domaine du traitement d'images, notamment en matière de programmabilité et de portabilité. Nous analysons plus particulièrement les interfaces de programmation de SMIL et de FREIA en section 4.2, puis nous présentons, dans la section 4.3, deux systèmes dédiés à compiler du code Python vers des langages de plus bas niveau. Nous introduisons un outil de conversion d'applications SMIL vers FREIA, `smiltofreia`, que nous décrivons dans la section 4.4. Nous traitons dans cet outil les difficultés inhérentes à la transformation d'un langage dynamique fondé sur Python en un langage statique, C. Nous évaluons, dans la section 4.5, les performances de notre outil sur notre ensemble de sept applications.

4.1 Concilier programmabilité et portabilité : le cas du traitement d'images

La vision par ordinateur est devenu un champ de recherche majeur suite à la conjonction de l'avènement des moteurs de recherche d'images sur Internet et de la révolution des smartphones à la fin des années 2000, qui ont mis une caméra dans chaque poche. Les applications grand public sont nombreuses et disruptives ; de la réalité augmentée aux voitures autonomes, la vision par ordinateur est l'apanage des grands groupes logiciels et est intimement liée aux évolutions de l'apprentissage artificiel. De nombreuses bibliothèques logicielles, telles OpenCV [95], GEGL [51], ImageMagick [71], SMIL [46] ou FREIA [15], permettent de facilement développer des applications dans ce domaine. Ces bibliothèques rivalisent en matière de fonctionnalités, de performance et de programmabilité, afin d'attirer et de conserver leurs utilisateurs.

Un langage de programmation de choix pour ces bibliothèques est Python ; généraliste et versatile, celui-ci dispose d'une syntaxe lisible et concise, d'un système de typage dynamique et d'un ramasse-miettes pour gérer la mémoire, peut facilement s'interfacer avec C ou C++ et supporte les paradigmes de programmation impératif, fonctionnel (dans une certaine mesure) et orienté objet. Toutefois, Python est limité par la performance de son interpréteur, qui induit un surcoût d'exécution et de mémoire occupée par rapport à un langage natif comme C ou C++ [77, 76]. De plus, l'interpréteur contrarie l'usage du parallélisme en limitant les threads à un seul cœur, pour plus de sécurité, au travers d'un système de verrouillage appelé Global Interpreter Lock [40]. Cependant, ces problèmes peuvent être en partie contournés par l'utilisation de Python en tant que surcouche, également appelée wrapper, d'une bibliothèque écrite en C ou C++, ce qui permet d'allier la performance des langages natifs à la programmabilité de Python.

Pourtant, dès lors qu'il s'agit de cibler une architecture matérielle exotique [111] où un interpréteur Python n'est pas forcément disponible, ce qui n'est pas rare dans le cas du traitement d'images, cette démarche est inopérante. La solution traditionnelle consiste à développer une nouvelle version optimisée vers l'architecture cible dans un langage de plus bas niveau. Les compilateurs modernes ne sont pas encore capables de cibler toutes les configurations matérielles existantes ; c'est aux développeurs de fournir une version optimisée de leurs applications ou bibliothèques pour un matériel spécifique. Les modèles de programmations comme OpenMP [19], pour les architectures à mémoire partagée, MPI [49], pour la mémoire distribuée, OpenCL [60], pour les plateformes hétérogènes, ou OpenACC [94], qui vise à la fois les processeurs multicœurs et les processeurs graphiques, peuvent faciliter le port vers une classe d'accélérateurs, mais les optimisations à apporter — vectorisation, parallélisation des boucles, fusion, fission et tuilage — sont souvent spécifiques au matériel.

Plusieurs projets logiciels tentent de combiner la programmabilité offerte par Python aux

performances des processeurs multicœurs, voire à l'efficacité des processeurs graphiques modernes. Par exemple, Cython [42], dont nous reparlons plus bas, permet d'interfacer facilement du code Python avec du C pour passer outre les limitations de l'interpréteur. Pythran [67] est un compilateur de Python vers C++ pour les programmes scientifiques et vise les processeurs multicœurs modernes disposant d'extensions pour le calcul vectoriel. Le code C++ généré peut être réutilisé directement par Python, une fois compilé sous forme de bibliothèque partagée. Numba [83] est un compilateur Just-in-Time de Python vers la représentation intermédiaire de LLVM [84]. Cela permet de bénéficier de l'ensemble des cibles matérielles du projet LLVM, mais nécessite toujours la présence d'un interpréteur. Similairement, Parakeet [107] permet de générer à partir de Python du code optimisé pour les CPUs, ainsi que les GPUs, via CUDA [37]. Les bibliothèques d'algèbre linéaire pour l'apprentissage profond Theano [12] et Tensorflow [1] proposent une interface Python, mais génèrent en interne du code C++ ou CUDA optimisé. Enfin, les compilateurs pour le traitement d'images Halide [105] et PolyMage [91] visent également à effectuer des optimisations à la fois spécifiques à leur domaine et aux architectures modernes. Ces compilateurs prennent en entrée des langages spécifiques au domaine simplifiés, afin d'améliorer la programmabilité. Tandis que les applications PolyMage sont encore limitées à une exécution sur CPU, Halide est en mesure de générer du code OpenCL et CUDA et vise ainsi une gamme plus large d'accélérateurs matériels. Tous ces projets ne ciblent généralement que des architectures répandues, à savoir des CPUs ou des GPUs.

4.2 Application à SMIL et FREIA

Les bibliothèques logicielles SMIL et FREIA sont dédiée au traitement d'images et, plus spécifiquement, à la morphologie mathématique. Développées toutes les deux à l'École des mines dans le Centre de morphologie mathématique, elles donnent accès aux mêmes primitives de traitement d'images. Cependant, elles offrent des fonctionnalités différentes en ce qui concerne la programmabilité et la portabilité.

4.2.1 SMIL, une bibliothèque avec une interface Python

Présentée plus en détail dans la sous-section 3.3.2, la bibliothèque de traitement d'images par morphologie mathématique SMIL est écrite en C++ et optimisée pour les processeurs multicœurs modernes via l'utilisation d'OpenMP et l'auto-vectorisation des boucles du compilateur GCC. L'utilisation du générateur de wrappers Swig [112] permet de fournir plusieurs interfaces dans des langages de programmation de plus haut niveau, comme Python, Java, Ruby et Octave. Parmi celles-ci, l'interface la plus utilisée pour développer des applications est Python. Un exemple basique de code SMIL en Python est disponible extrait 4.1. Ce dernier consiste à lire une image `imin`, allouer une image `imout` de même taille et effectuer une dilatation morphologique de la première vers la seconde, avant de la sauvegarder. Le paradigme orienté objet de Python, hérité de l'implémentation C++ sous-jacente, se remarque ici au travers de l'appel de la méthode `save()` de la classe `smil.Image`.

L'interface Python de SMIL surcharge les opérateurs standards tels que $+, -, \times, \div, <$ ou $>$ pour, par exemple, symboliser des opérateurs pixel à pixel entre deux images. Ceci apporte une interface plus naturelle pour les manipulations d'images, qui s'appuie sur le paradigme fonctionnel, où les fonctions ne modifient pas leurs arguments, mais retournent une nouvelle valeur en tant que résultat.

À travers l'usage d'interfaces vers des langages de programmation de haut niveau très utilisés dans le traitement d'images, comme le langage Python, SMIL peut facilement être

```python
import smilPython as smil

imin = smil.Image("input.png")  # read from disk
imout = smil.Image(imin)        # allocate imout
smil.dilate(imin, imout)        # morphological dilatation
imout.save("output.png")        # write to disk
```

Extrait 4.1 – Dilatation morphologique en SMIL

utilisée pour développer des applications de traitement d'images utilisant des opérateurs complexes, l'interface cachant les problématiques de bas niveau telle la gestion de l'allocation mémoire.

4.2.2 FREIA, un framework pour les accélérateurs matériels

FREIA, ainsi que présentée dans la sous-section 3.3.1, n'est à proprement parler pas une bibliothèque : plusieurs implémentations des mêmes opérateurs de traitement d'images coexistent, destinés à différentes cibles matérielles. Une interface commune, en C, fait le lien entre les applications et les différentes implémentations. Le but est de pouvoir développer une application une seule fois et qu'une recompilation suffise pour que celle-ci soit exécutable sur les différentes cibles de FREIA. Par ailleurs, le compilateur source-à-source PIPS peut être utilisé pour générer du code spécifique plus optimisé [29].

L'interface de FREIA est en C, langage de plus bas niveau que Python : pas de système de classes, pas de surcharge d'opérateurs et gestion manuelle de la mémoire. L'extrait 4.2 montre ainsi un exemple de dilatation morphologique écrite en FREIA : les images doivent être allouées dans la fonction principale et la mémoire doit être libérée à la fin de son utilisation.

```c
#include "freia.h"

int main(void) {
  /* initializations... */

  /* image allocations */
  freia_data2d *imin = freia_common_create_data(/*...*/);
  freia_data2d *imout = freia_common_create_data(/*...*/);
  /* read from disk */
  freia_common_rx_image(imin, /*...*/);
  /* morphological dilatation */
  freia_cipo_dilate(imout, imin, 8, 1);
  /* write to disk */
  freia_common_tx_image(imout, /*...*/);
  /* freeing memory */
  freia_common_destruct_data(imin);
  freia_common_destruct_data(imout);

  /* shutdown... */
}
```

Extrait 4.2 – Dilatation morphologique en FREIA

4.2.3 Comment combler le fossé ?

Grâce à l'utilisation du compilateur GCC, SMIL peut cibler différents processeurs. L'outil de compilation CMake [28] facilite l'utilisation de différents compilateurs permet de porter SMIL sur plusieurs autres plateformes — nous nous en servirons au chapitre 5 pour porter directement SMIL sur un cluster du processeur manycore MPPA. Toutefois, porter totalement cette bibliothèque sur un accélérateur matériel spécifique, comme un processeur graphique, est une lourde tâche. D'un autre côté, FREIA dispose d'un vaste ensemble de cibles matérielles et propose une architecture extensible, afin d'en rajouter de nouvelles. FREIA pêche cependant en comparaison par son interface C, qui ne permet pas d'écrire des applications aussi facilement que SMIL avec Python.

Plusieurs solutions existent afin de concilier la programmabilité de SMIL et les cibles matérielles de FREIA. Un port complet de SMIL sur chacune des cibles de FREIA est une tâche longue et complexe. Réutiliser le travail accompli dans FREIA semble une bonne idée pour obtenir rapidement des ports fonctionnels. Il est possible de ré-implémenter le langage SMIL en FREIA, c'est-à-dire de rediriger les appels aux opérateurs SMIL vers les équivalents FREIA au travers d'un wrapper. Cette solution ne permet cependant pas à PIPS d'effectuer ses optimisations, puisque ces dernières sont fondées explicitement sur l'interface FREIA C, qui ne serait dans ce cas pas exposée par les applications. Donner à PIPS la capacité d'analyser directement du code SMIL n'est pas à l'ordre du jour : PIPS ne supporte en effet que les langages C et Fortran, et développer de nouveaux analyseurs pour Python ou C++ est une procédure longue et fastidieuse. Une dernière solution consiste à développer un compilateur qui convertit des applications écrites en SMIL Python en FREIA. Ainsi, les applications SMIL peuvent être automatiquement portées sur toutes les cibles de FREIA et bénéficient, en outre, des optimisations de PIPS. C'est vers l'implémentation de cette solution que nous avons porté nos efforts.

4.3 Manipulation et accélération de code Python

Afin d'analyser et de convertir des applications SMIL Python en FREIA C, nous avons utilisé principalement deux outils : RedBaron, un framework pour le refactoring de code Python, et Cython, un compilateur de Python vers C.

La bibliothèque standard Python fournit, par ailleurs, des outils de manipulation de code Python, comme les modules *ast* [103] pour gérer des arbres syntaxiques et *inspect* [104] pour modifier un programme Python durant son exécution.

4.3.1 RedBaron, un outil pour le refactoring de code Python

Le *refactoring*, ou remaniement de code, est un processus de développement logiciel visant à modifier une portion significative de code source, souvent afin d'en améliorer la lisibilité ou la maintenabilité. Ce concept recouvre des notions allant du renommage de variables dans un ou plusieurs fichiers source au changement architectural majeur. Des aides au refactoring sont traditionnellement offertes par les environnements de développement logiciels, mais ces solutions sont rarement utilisables en dehors de ces environnements.

RedBaron [106, 99] est un outil facilitant les transformations de code Python spécialement adapté au refactoring. Pour effectuer ces transformations, RedBaron s'appuie sur une structure d'arbre syntaxique appelée Baron [9] et développée par le même auteur. Baron est un *Full Syntax Tree* (FST) ; contrairement à un AST classique, il conserve toutes les informations du code initial et, notamment, les commentaires et informations de formatage de code. Ceci permet de régénérer un code source identique à partir du FST :

```
fst_to_code(code_to_fst(source_code)) == source_code
```

Un exemple de FST est disponible en extrait 4.3. Celui-ci a été obtenu à partir de l'instruction `smil.dilate(imin, imout)`. Il est à noter qu'autour de chaque nœud de l'arbre, les informations de formatage, dont par exemple l'espace après la virgule, sont conservées.

```
{"type": "atomtrailers",
 "value": [
     {"type": "name", "value": "smil"},
     {"type": "dot", "first_formatting": [],
      "second_formatting": []},
     {"type": "name", "value": "dilate"},
     {"first_formatting": [], "third_formatting": [],
      "type": "call", "fourth_formatting": [],
      "second_formatting": [],
      "value": [
          {"type": "call_argument",
           "first_formatting": [],
           "second_formatting": [], "target": {},
           "value": {"type": "name", "value": "imin"}},
          {"type": "comma", "first_formatting": [],
           "second_formatting": [{"type": "space", "value": " "}]},
          {"type": "call_argument", "first_formatting": [],
           "second_formatting": [], "target": {},
           "value": {"type": "name", "value": "imout"}}
      ]}]}
```

Extrait 4.3 – FST de `smil.dilate(imin, imout)`

RedBaron offre une interface orientée objet efficace pour effectuer des requêtes et modifier le FST sous-jacent. Les nœuds peuvent être accédés comme des listes et modifiés directement par affectation. C'est ainsi que l'extrait 4.4 renomme toutes les occurrences de la variable `imin` en une version abrégée `in`.

```
from redbaron import RedBaron
red = RedBaron("smil.dilate(imin, imout)")
for node in red.find_all("NameNode", value="imin"):
    node.value = "in"
print(red.dumps())  # smil.dilate(in, imout)
```

Extrait 4.4 – Cas d'usage de RedBaron : renommage de variable

RedBaron, en fournissant une interface accessible pour manipuler du code, se démarque du module standard *ast*. Ce dernier, ne prenant pas en compte les informations de formatage et les commentaires, ne peut régénérer exactement à partir d'un arbre le code original. De plus, son interface de bas niveau rend les manipulations de nœuds plus complexes.

4.3.2 Cython, un compilateur de Python vers C

Cython [42, 11] est un outil permettant de lier des applications Python à des bibliothèques écrites en C afin de profiter des performances du langage compilé. Il permet,

— d'une part, de créer facilement des wrappers Python autour de bibliothèques C grâce à un langage inspiré de Python et,
— d'autre part, de compiler du code applicatif Python faisant usage de ces wrappers et ainsi générer du C ou directement des exécutables.

Cython permet de définir simplement une interface Python à une bibliothèque C dans une syntaxe proche de Python. Il s'agit, tout d'abord, de redéclarer les structures de données et les fonctions C dans un fichier spécifique d'extension `.pxd`, dans une syntaxe à mi-chemin du C et du Python. Par exemple, l'extrait 4.5 montre les déclarations de la structure de données `freia_data2d` et de six fonctions issues du header `freia.h`.

```
ctypedef unsigned int u32
ctypedef int i32
ctypedef i32 freia_status
ctypedef void * freia_ptr

cdef extern from "freia.h":

    ctypedef struct freia_data2d:
        freia_ptr raw
        freia_ptr *row
        u32 bpp
        u32 width
        u32 height
        u32 xStartWa
        u32 yStartWa
        u32 widthWa
        u32 heightWa

    freia_data2d * freia_common_create_data(u32, u32, u32)
    freia_status freia_common_destruct_data(freia_data2d *)

    freia_status freia_common_rx_image(freia_data2d *, freia_dataio *)
    freia_status freia_common_tx_image(freia_data2d *, freia_dataio *)

    freia_status freia_cipo_dilate(freia_data2d *, freia_data2d *,
                                   i32, u32)
    freia_status freia_cipo_erode(freia_data2d *, freia_data2d *,
                                  i32, u32)
```

EXTRAIT 4.5 – Redéfinition en Cython des déclarations du header `freia.h`

À partir de ces fichiers `.pxd`, il est possible de définir une interface « Pythonesque » au-dessus de ces appels vers C, cette fois dans des fichiers à l'extension `.pyx`. Ainsi l'extrait 4.6 présente l'implémentation d'une classe Python `Data2D` construite au-dessus des fonctions FREIA déclarées ci-dessus en extrait 4.5. Tout script ou programme Python peut ensuite appeler les fonctions définies dans un tel fichier. Les fichiers `.pyx` contiennent des informations de typage, habituellement absentes des sources Python, utiles au compilateur Cython pour générer du C.

Le compilateur Cython génère donc du code C à partir de fichiers `.pyx` construits ou non au-dessus de bibliothèques C via des déclarations `.pxd`. À partir de ce code C, il est possible de générer des bibliothèques statiques ou dynamiques que l'on peut appeler directement depuis Python. Le compilateur Cython peut également générer à la demande une fonction `int main(void)` dans le cas d'exécutables.

Pour accélérer du code Python et profiter de performances proches du C sans sacrifier la

```
cdef class Data2D:
    cdef freia_data2d * _c_data2d

    def __cinit__(self, u32 bpp, u32 width, u32 height) :
        self._c_data2d = freia_common_create_data(bpp, width, height)
    def __dealloc__(self) :
        freia_common_destruct_data(self._c_data2d)

    def rxImage(self, DataIO dataio) :
        return freia_common_rx_image(self._c_data2d, &dataio._c_dataio)
    def txImage(self, DataIO dataio) :
        return freia_common_tx_image(self._c_data2d, &dataio._c_dataio)

    def cipoDilate(self, Data2D imout, i32 connexity, u32 size) :
        return freia_cipo_dilate(imout._c_data2d, self._c_data2d,
                                 connexity, size)
    def cipoErode(self, Data2D imout, i32 connexity, u32 size) :
        return freia_cipo_erode(imout._c_data2d, self._c_data2d,
                                connexity, size)
```

Extrait 4.6 – Interface « Pythonesque » en Cython au-dessus de FREIA

programmabilité, il est suggérer de convertir les routines calculatoires en extension Cython (fichiers `.pyx`) en ajoutant des informations de typage, puis de compiler ceux-ci avec Cython pour pouvoir les ré-utiliser sous forme de bibliothèque dynamique dans le reste du code Python. Ainsi, il est possible de conserver un code « pythonesque » tout en obtenant des performances natives proches du C.

4.4 De SMIL à FREIA

Afin de convertir des applications SMIL Python en FREIA C, nous avons utilisé RedBaron, d'abord en conjonction avec Cython, puis seul dès lors que l'approche initiale n'a pas porté ses fruits.

4.4.1 Génération de code C avec Cython

Dans un premier temps, nous avons utilisé Cython dans l'optique de générer rapidement du C qui fait appel à FREIA à partir d'applications SMIL. Une première étape a consisté à développer un wrapper autour de l'API FREIA en Cython, afin de rapprocher l'API FREIA C de l'API SMIL Python. L'extrait 4.5 et l'extrait 4.6 présentés plus haut sont issus de ce wrapper. Ensuite, RedBaron a été utilisé pour convertir les applications SMIL Python vers ce wrapper, Cython se chargeant ensuite de générer du code C. Ce fonctionnement est résumé en figure 4.1.

Un exemple de code s'appuyant sur le wrapper Cython autour de FREIA est disponible extrait 4.7. Cet exemple de code Python, dans notre cas généré par RedBaron, est très proche de la version C présentée plus haut dans l'extrait 4.2, à l'exception de la libération de la mémoire allouée pour les images. Cette dernière est effectuée automatiquement à la destruction de l'objet `freia.Data2d` par le ramasse-miettes de l'interpréteur Python.

Avec ce wrapper, il devient possible d'écrire en Python des applications faisant appel à FREIA. Nous avons utilisé le FST fourni par RedBaron pour convertir des applications SMIL Python vers ce wrapper, en transformant les appels SMIL en appels FREIA, principalement

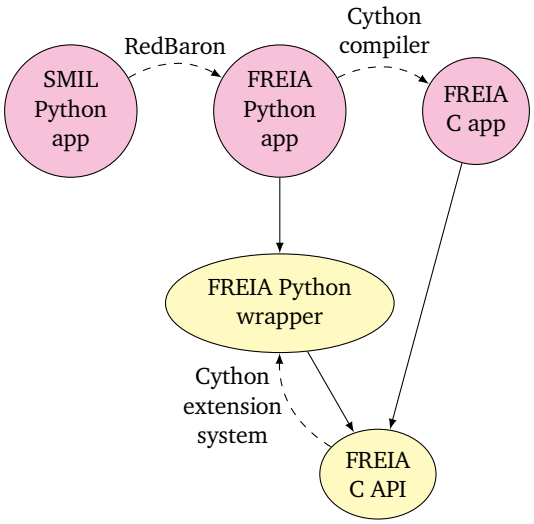


FIGURE 4.1 – Utilisation de Cython pour convertir des applications SMIL en FREIA

```
1  def main(*args):
2      # initializations...
3      freia_img_imin = freia.Data2D(fdin.framebpp, fdin.framewidth,
4                                    fdin.frameheight)
5      freia_img_imin.rxImage(fdin)
6      freia_img_imout = freia.Data2D(fdin.framebpp, fdin.framewidth,
7                                     fdin.frameheight)
8      freia_img_imin.cipoDilate(freia_img_imout, 8, 1)
9      freia_img_imout.txImage(fdout)
10     # shutdown...
```

EXTRAIT 4.7 – Conversion de extrait 4.1 en FREIA/Cython

par manipulation de chaînes de caractères. Enfin, nous avons utilisé le compilateur Cython pour générer du code C à partir de notre wrapper, ce qui au final revient à faire directement appel à FREIA. Un exemple de code généré par Cython est représenté en extrait 4.8. Cet extrait montre l'appel effectif à la fonction `freia_cipo_dilate`. Le code C généré par Cython conserve toutefois une dépendance à l'interpréteur CPython. Cet interpréteur n'ayant pas été porté sur toutes les cibles matérielles de FREIA, cela remet en question la portabilité du code généré par Cython. De plus, de nombreuses instructions supplémentaires et des structures de données opaques sont générées par Cython pour satisfaire cette dépendance. Ceci rend le code généré très complexe à analyser. Le compilateur source-à-source PIPS n'est ainsi pas en mesure d'appliquer toutes ses optimisations, afin de régénérer du code spécifique.

Utiliser Cython pour interfacer Python et C et, ainsi, accélérer des applications Python n'a pas fourni des résultats pouvant facilement être réutilisés en aval dans la chaîne de compilation FREIA.

```c
static PyObject *__pyx_pf_9smil_dilate_6Data2D_14cipoDilate(
    struct __pyx_obj_9smil_test_Data2D *__pyx_v_self,
    struct __pyx_obj_9smil_test_Data2D *__pyx_v_imout,
    __pyx_t_7pyfreia_int32_t __pyx_v_connexity,
    __pyx_t_7pyfreia_uint32_t __pyx_v_size) {
  PyObject *__pyx_r = NULL;
  __Pyx_RefNannyDeclarations PyObject *__pyx_t_1 = NULL;
  __Pyx_RefNannySetupContext("cipoDilate", 0);
  __Pyx_XDECREF(__pyx_r);
  __pyx_t_1 = PyInt_FromLong(
      freia_cipo_dilate(__pyx_v_imout->_c_data2d,
                        __pyx_v_self->_c_data2d,
                        __pyx_v_connexity, __pyx_v_size));
  __Pyx_GOTREF(__pyx_t_1);
  __pyx_r = __pyx_t_1;
  __pyx_t_1 = 0;
  __Pyx_XGIVEREF(__pyx_r);
  __Pyx_RefNannyFinishContext();
  return __pyx_r;
}
```

Extrait 4.8 – Appel effectif à la dilatation FREIA après génération de C par Cython

4.4.2 D'une API à l'autre

Plutôt que de se reposer sur Cython pour générer du C de bas niveau à partir de Python, nous avons profité de notre expérience sur RedBaron pour générer directement du code FREIA — donc du C — à partir d'applications SMIL Python. Pour cela, il a suffit de reprendre et d'étendre notre compilateur SMIL Python vers FREIA Cython pour qu'il puisse générer du C. Ce compilateur, nommé `smiltofreia` [126] et développé en Python au-dessus de RedBaron, parcourt le FST d'une application SMIL Python et transforme chaque nœud en l'instruction C correspondante.

Algorithme 4.1 : Description générale de l'exécution de `smiltofreia` sur une application SMIL Python

Entrées : `src` — code source SMIL Python
Sorties : `dst` — code FREIA C correspondant

```
  /* preprocessing: ensure there is a main function          */
1 src ← preprocess(src) ;
  /* get and process the RedBaron FST                        */
2 fst ← RedBaron.generate_fst(src) ;
3 dst ← generate_c(fst, Scope()) ;
4 dst.insert_freia_includes() ;
  /* apply clang-format to prettify the C output code        */
5 dst.clang_format() ;
6 return dst ;
```

L'algorithme 4.1 présente une vue générale du fonctionnement de notre compilateur. L'outil RedBaron est utilisé afin d'obtenir le *Full Syntax Tree* du code d'entrée. Notre compilateur opère sur les nœuds de ce FST pour typer les variables et générer du code C. Une passe de pré-traitement peut générer, si besoin, une déclaration de fonction `def main()`

autour du code d'entrée, afin d'obtenir un point d'entrée normalisé pour nos applications. Pour rendre notre code généré plus agréable à l'œil, il est possible d'appliquer sur celui-ci l'outil de formatage `clang-format` [25].

Algorithme 4.2 : Transformation en C d'un nœud de FST RedBaron

Entrées : `node` du FST et `scope` relatif
Sorties : `cc` — code FREIA C correspondant

```
   /* recursive bottom-up code generation and typing                    */
 1 def generate_c(node, scope) :
 2     cc ← Ccode() ;
 3     switch node.type() do
 4         case BlockNode do
 5             foreach subnode ∈ node do
 6                 cc.add(generate_c(subnode, scope)) ;
 7         case AssignNode do
 8             rcode ← generate_c(node.rightarg, scope) ;
 9             var, type ← node.leftarg, node.rightarg.rettype ;
10             if var ∈ scope then
11                 assert(scope.get_type(var) == type) ;
12             else
13                 scope.add_type(var, type) ;
14                 cc.add("%s %s ;", type, var) ;
15             cc.add("%s = %s ;", var, rcode) ;
16         case BinOpNode do
               /* in1 ⊗ in2                                                */
17             op, in1, in2 ← node.value, node.first, node.second ;
18             node.out, node.rettype ← gen_id(), get_rettype(op, in1, in2) ;
19             scope.add_type(node.out, node.rettype) ;
20             cc.add("%s %s ;", node.rettype, node.out) ;
21             cc.add("%s = %s ;", node.out, gen_init(node.out, node.rettype)) ;
22             cc.add("%s(%s,%s,%s) ;", freia_call(op), node.out, in1.out, in2.out) ;
23         [...] ;
24     return cc ;
```

Une description plus détaillée de la routine de traitement des nœuds RedBaron est disponible dans l'algorithme 4.2. Cet algorithm montre l'approche récursive suivie pour générer le code C correspondant à un nœud du FST. Chaque nœud est traité selon son type RedBaron, depuis les nœuds de plus haut niveau, qui représentent les construction visibles au niveau module Python, telles que les classes, les déclarations de fonctions ou les déclarations de variables globales, jusqu'aux nœuds de plus bas niveau : identificateurs, opérateurs, constantes ou informations de formatage. Par exemple, traiter un bloc d'instructions Python consiste à traiter chaque sous-nœud à la suite.

Cependant, cette approche, plus directe en comparaison de la précédente, nécessite de multiples précautions. D'un côté, Python est un langage dynamique disposant d'un ramasse-miettes qui gère automatiquement les allocations mémoire. C n'en dispose pas : les variables doivent être déclarées et typées ; la gestion de la mémoire est faite à la main ; et la mémoire

allouée sur le tas doit (normalement) être libérée à la fin de son utilisation. De plus, les APIs de SMIL et FREIA peuvent différer :

- des opérateurs de traitement d'images peuvent ne pas être disponibles dans les deux bibliothèques ;
- certaines structures de données peuvent être implémentées radicalement différemment de part et d'autre ;
- certaines fonctions de SMIL profitent du polymorphisme offert par Python et C++ pour proposer différentes signatures ;
- la surcharge d'opérateurs en Python et C++ permet à SMIL de proposer, au moins pour certains opérateurs, une API fonctionnelle ; à l'inverse, les opérateurs FREIA fonctionnent par passage de pointeurs et effets de bords : l'API est ici exclusivement impérative.

Notre compilateur `smiltofreia` doit résoudre chacun de ces problèmes afin de générer du code C valide qui respecte la sémantique initiale de l'application SMIL. En conséquence, les entrées de `smiltofreia` sont limitées à du code SMIL Python pur, sans utiliser d'autres modules Python.

Typage Afin de gérer la différence entre le typage dynamique de Python et celui, statique, de C, notre compilateur ne peut opérer que sur des applications dont le type des variables est statiquement inférable. Notre compilateur `smiltofreia` a été développé suivant une stratégie défensive, le but étant de placer les contraintes sur le code en entrée, afin d'assurer que les transformations utilisées produisent un code C bien typé et avec une gestion correcte de la mémoire. `smiltofreia` a connaissance des API SMIL et FREIA ainsi que de la correspondance entre la signature de leurs fonctions. Les variables sont typées à la première initialisation et ne peuvent changer de type. Dans le cas contraire, la compilation s'arrête avec un message d'erreur et fournit les coordonnées du nœud SMIL Python responsable. Les arguments des fonctions sont également typés et transformés avant d'être passés aux fonctions FREIA correspondantes.

Polymorphisme de fonction Python et C++ permettent le polymorphisme de fonctions, c'est-à-dire qu'une même fonction peut avoir plusieurs signatures avec différents arguments. L'interface FREIA C, au contraire, propose des fonctions avec une signature fixe, sans arguments optionnels. Plutôt que de restreindre cette fonctionnalité, nous avons préféré réécrire le code Python problématique avec RedBaron vers une forme canonique plus proche de l'appel FREIA correspondant, donc plus facile à manipuler. Par exemple, les deux instructions SMIL suivantes sont équivalentes et illustrent le polymorphisme de la fonction `smil.dilate()` par rapport à son dernier argument :

```
1  smil.dilate(imin, imout, 5)
2  smil.dilate(imin, imout, smil.SquSE(5))
```

À la ligne 1, le dernier paramètre est un entier représentant l'élément structurant par défaut de taille 5. À la ligne 2, le dernier paramètre est un élément structurant carré (l'élément structurant par défaut), dont la taille est passée explicitement au constructeur. Notre compilateur `smiltofreia` traite la première ligne en utilisant les capacités de RedBaron pour réécrire les nœuds et ainsi se ramener à la forme canonique de la seconde ligne.

Atomisation d'expressions d'images La bibliothèque SMIL pratique extensivement la surcharge d'opérateurs, ce qui offre une interface plus naturelle pour les opérateurs arithmétiques. Ainsi, une addition pixel à pixel entre deux images peut s'écrire de façon naturelle `o = i1 + i2` plutôt que `smil.add(i1, i2, o)`. La première formulation est fonctionnelle : le résultat de l'addition `i1 + i2` est alloué au vol par la fonction appelée et affecté à la variable `o`. De tels opérateurs peuvent être chaînés pour obtenir des codes plus expressifs, à la manière de

```
out = in0 * in1 + ((in2 - in4) | (in5 & in1)
```

C ne permet pas la surcharge d'opérateurs et prône un paradigme plus impératif : c'est l'appelant qui alloue et libère la mémoire pour passer un pointeur à l'appelé. Le calcul précédent correspond ainsi, en FREIA, à plusieurs appels imbriqués. Notre compilateur génère les instructions suivantes, en générant quatre variables intermédiaires :

```
freia_aipo_mul(tmp0, in0, in1);
freia_aipo_sub(tmp1, in2, in4);
freia_aipo_and(tmp2, in5, in1);
freia_aipo_or(tmp3, tmp1, tmp2);
freia_aipo_add(out, tmp0, tmp3);
```

Ces variables intermédiaires sont, par la suite, détectées et supprimées par les analyses de PIPS.

Différences entre les APIs Les bibliothèques SMIL et FREIA fournissent des interfaces relativement proches, mais les différences restantes doivent être prises en compte.

Éléments structurants Les éléments structurants, concepts centraux de la morphologie mathématique, sont implémentés différemment dans SMIL et FREIA. Dans FREIA, ils prennent la forme d'un tableau d'entiers de taille 9. Pour augmenter la taille du voisinage, il faut appliquer l'opérateur morphologique plusieurs fois de suite. Dans SMIL, ce sont des classes disposant d'un attribut de taille. Lors de la conversion d'un opérateur morphologique SMIL en FREIA, `smiltofreia` prend en compte la taille de l'élément structurant pour la passer à la fonction FREIA pertinente. Ainsi, la dilatation morphologique SMIL :

```
smil.dilate(imin, imout, smil.SquSE(5))
```

est convertie en le code FREIA suivant.

```
#define e0 SMILTOFREIA_SQUSE
#define e0_s 5
  freia_cipo_dilate_generic_8c(imout, imin, e0, e0_s);
```

Nous avons utilisé des macros préprocesseur afin d'imiter le comportement de la structure de données SMIL et garder trace des attributs. Cela permet également de faciliter la substitution des variables par PIPS. Nous avons défini les éléments structurants les plus communs dans un fichier d'en-tête à part.

Ajout de fonctions dans FREIA Durant le développement de `smiltofreia`, nous avons réalisé que certaines transformations pouvaient être facilitées en modifiant directement l'API FREIA. Par exemple, certains opérateurs morphologiques de FREIA présupposent d'utiliser un élément structurant par défaut, alors que l'équivalent SMIL accepte des éléments structurants arbitraires. Les fonctions FREIA avec une signature similaire à :

```
freia_status freia_cipo_dilate(freia_data2d *imout, freia_data2d *imin,
                               uint32_t size) {
  unsigned int i;
  int32_t square_strelt[9] = { 1, 1, 1, 1, 1, 1, 1, 1, 1};
  freia_aipo_dilate_8c(imout, imin, square_strelt);
  for (i = 1; i < size; i++)
    freia_aipo_dilate_8c(imout, imout, square_strelt);
  return FREIA_OK;
}
```

n'utilisent que des éléments structurants carrés, mais ceux-ci peuvent être de taille arbitraire : en interne, l'implémentation consiste en une boucle autour de l'appel atomique `freia_aipo_dilate`. Plutôt que d'ajouter des contraintes supplémentaires aux entrées de `smiltofreia` ou bien de générer du code FREIA complexe explicitant cette boucle, nous avons ajouté plusieurs fonctions à FREIA pour rapprocher son interface de celle de SMIL :

```
void freia_cipo_dilate_generic_8c(freia_data2d *imout,
                                  freia_data2d *imin,
                                  const int32_t *se,
                                  uint32_t size);
```

Ces nouvelles fonctions facilitent la génération de code FREIA à partir de SMIL. Un autre exemple de différence d'interface est celle des fonctions `smil.mask()`, absente de FREIA, et `freia_aipo_replace_const()`, absente de SMIL. Les deux fonctions ont une sémantique relativement proche, mais diffèrent tout de même. Il est possible d'implémenter chacune en utilisant la seconde en combinaison d'un opérateur de seuillage, mais cela implique l'allocation d'une image intermédiaire, aux dépens de l'empreinte mémoire et de la performance globale. Implémenter `smil.mask()` en FREIA résoud ce problème.

Sortie de `smiltofreia` En exécutant `smiltofreia` sur l'exemple de dilatation morphologique de l'extrait 4.1, nous obtenons le code FREIA de l'extrait 4.9. Au final, le code généré reste proche du code FREIA écrit à la main de l'extrait 4.2, les seuls changements visibles étant l'inclusion du fichier d'en-tête de compatibilité `smil-freia.h` et l'utilisation de macros préprocesseur pour la gestion de la taille des éléments structurants.

4.5 Évaluation de l'approche

Nous avons évalué l'intégration de notre compilateur `smiltofreia` dans la chaîne de compilation FREIA en reprenant sept applications FREIA : « anr999 », « antibio », « burner », « deblocking », « licensePlate », « retina » et « toggle ». Nous avons réécrit entièrement et à la main ces sept applications à la fois en SMIL C++ et en SMIL Python idiomatique, afin de pouvoir comparer la sortie de `smiltofreia` aux applications d'origine. Nos applications font, au final, toutes appel à la bibliothèque SMIL au travers de divers wrappers et sont exécutées sur le même processeur Intel Core i7-3820, ainsi que montré en figure 4.2.

```c
#include "freia.h"
#include "smil-freia.h"

int main(int argc, char *argv[]) {
  /* initializations... */
  freia_data2d *imin;
  imin = freia_common_create_data(/* */);
  freia_data2d *imout;
  imout = freia_common_create_data(/* */);
#define e0 SMILTOFREIA_SQUSE
#define e0_s 1
  freia_cipo_dilate_generic_8c(imout, imin, e0, e0_s);
  freia_common_tx_image(imout, &fdout);
  freia_common_destruct_data(imout);
  freia_common_destruct_data(imin);
  /* shutdown... */
}
```

EXTRAIT 4.9 – Sortie FREIA C de `smiltofreia` appliqué à la extrait 4.1

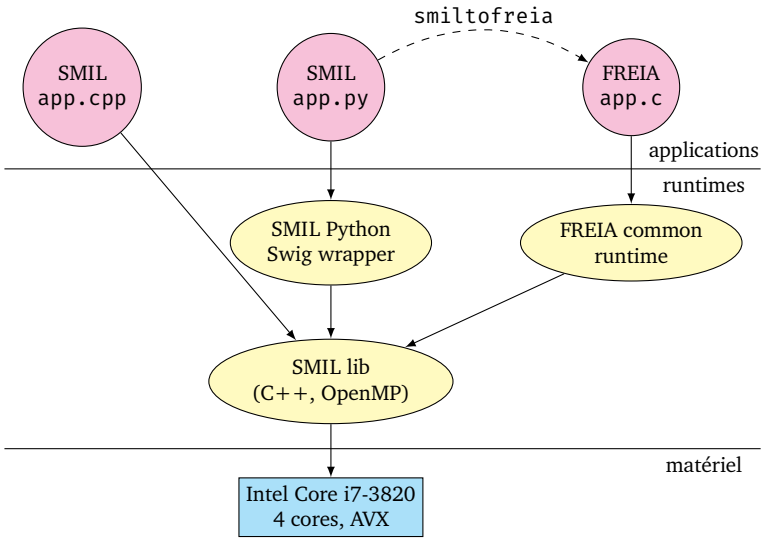


FIGURE 4.2 – Chaîne de compilation

Nous comparons, tout d'abord, dans le tableau 4.1 le nombre de lignes de code que comprend chaque application dans les deux langages SMIL Python et FREIA, ainsi que le gain offert par la réécriture en SMIL Python. À fonctionnalités identiques, les applications SMIL ont besoin de deux à dix fois moins de lignes de code. Ceci illustre l'expressivité supérieure du langage Python par rapport au C : les applications SMIL Python sont ainsi plus rapides à écrire et plus faciles à maintenir.

Il est possible d'utiliser ensuite PIPS sur les codes FREIA pour exécuter quelques passes d'optimisation telles que l'inlining des opérateurs composés, la substitution des variables et le déroulage des boucles. Les temps d'exécution des applications FREIA avec ou sans PIPS (« init » et « + opts ») peuvent ainsi être comparés. La figure 4.3 résume cette méthodologie d'évaluation.

Apps	#LoC		Gain
	SMIL	FREIA	
anr999	23	88	3.8
antibio	61	200	3.3
burner	55	510	9.3
deblocking	74	162	2.2
licensePlate	37	202	5.5
retina	40	471	11.8
toggle	40	144	3.6
GMEAN	44.4	213.0	4.8

Table 4.1 – Nombre de lignes de code SMIL et FREIA pour chaque application

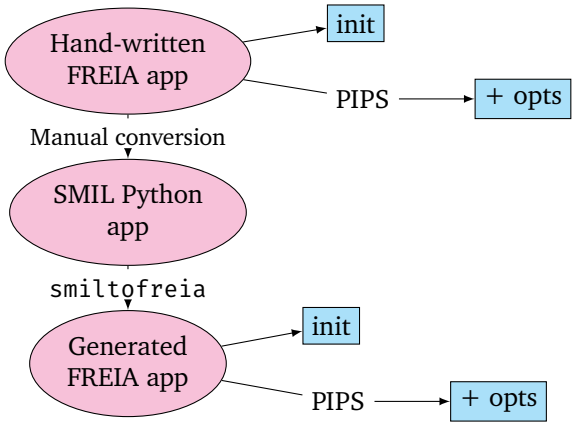


Figure 4.3 – Méthodologie d'évaluation : nous comparons les applications FREIA écrites à la main et les applications SMIL Python converties grâce à `smiltofreia` avec ou sans optimisations PIPS

Le tableau 4.2 présente les temps d'exécution de nos sept applications. Ceux-ci ont été mesurés en excluant les entrées/sorties des applications ; il s'agit donc des temps de calcul purs. La colonne « SMIL » représente les applications Python, « Hand-written », les applications originales et « Generated », les applications FREIA générées par `smiltofreia`. Les sous-colonnes « init » et « + opts » réfèrent à l'application ou non des optimisations de PIPS. Afin de rendre possible la comparaison, toutes ces applications font appel aux mêmes routines optimisées de la bibliothèque C++ SMIL, au travers du wrapper Python ou bien via FREIA. Les applications sont exécutées sur le même processeur hôte, l'Intel Core i7-3820 — 2012, 4 cœurs, AVX — présent dans la machine MPPA Developer.

Les trois graphiques présents en figure 4.4, figure 4.5 et figure 4.6 sont des représentations comparatives plus explicites de ces mêmes données. Ainsi, la figure 4.4 compare les exécutions de nos applications en SMIL C++ et SMIL Python. Nous voyons, de cette manière, que SMIL Python n'est en moyenne que 3% plus lent que SMIL C++, causé par l'utilisation du wrapper Python, qui a donc un impact minimal sur l'exécution globale de nos applications. Les applications SMIL écrites en Python sont donc très légèrement plus lentes que leurs équivalents C++, mais bénéficient de la syntaxe Python, plus simple et plus expressive.

Apps	SMIL		FREIA			
	C++	Python	Hand-written		Generated	
			init	+ opts	init	+ opts
anr999	0.7	0.7	0.7	0.6	0.7	0.6
antibio	20.5	20.5	25.0	24.5	25.2	25.0
burner	53.7	49.5	8.8	8.8	9.2	9.1
deblocking	28.8	29.8	30.3	30.0	31.3	30.7
licensePlate	2.4	2.7	2.3	2.1	2.4	1.9
retina	8.1	8.5	8.1	6.8	8.2	7.3
toggle	1.5	1.6	1.7	1.7	1.4	1.5

TABLE 4.2 – Temps d'exécution (ms) de sept applications FREIA et SMIL converties en FREIA

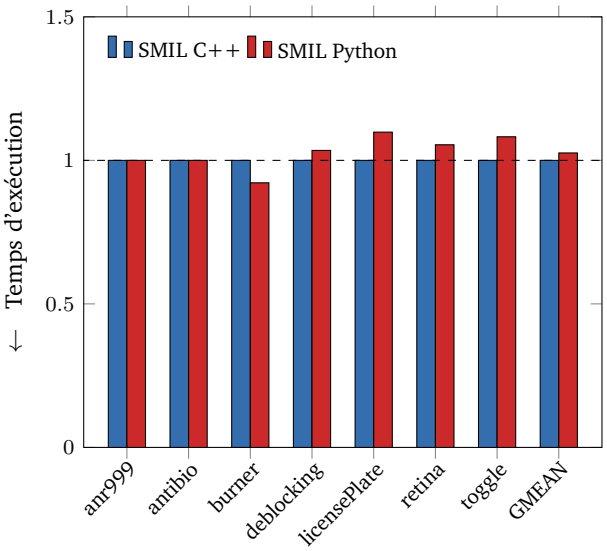


FIGURE 4.4 – Temps d'exécution relatifs d'applications de traitement d'images : (1) applications écrites en SMIL C++ ; (2) applications écrites en SMIL Python

Nous comparons ensuite les temps d'exécution de nos applications SMIL Python, avant et après conversion en FREIA par notre compilateur `smiltofreia`, puis optimisations par PIPS. Les résultats, disponibles en figure 4.5, montrent que la conversion en FREIA apporte généralement une meilleure performance, mais celle-ci est principalement due à des différences d'implémentation entre les opérateurs morphologiques des deux côtés. Ainsi, l'application « burner » est six fois plus rapide en FREIA qu'en SMIL. Cette application appelle l'opérateur morphologique appelé *reconstruction géodésique par fermeture*, implémenté dans SMIL avec des structures de données complexes, comme des queues hiérarchiques, qui sont adaptées à de gros volumes de données, mais qui se révèlent ici inefficaces avec nos tailles d'images réduites. En comparaison, la version FREIA de cet opérateur est plus simple et n'implique que des opérateurs de base, plus optimisés. Un extrait du code source de cet opérateur a déjà été commenté chapitre 6. En moyenne, le code généré de nos applications est 26% plus rapide que le code Python d'origine. Les optimisations de PIPS, appliquées au code FREIA généré, permettent un gain de performance supplémentaire, de l'ordre de 12%.

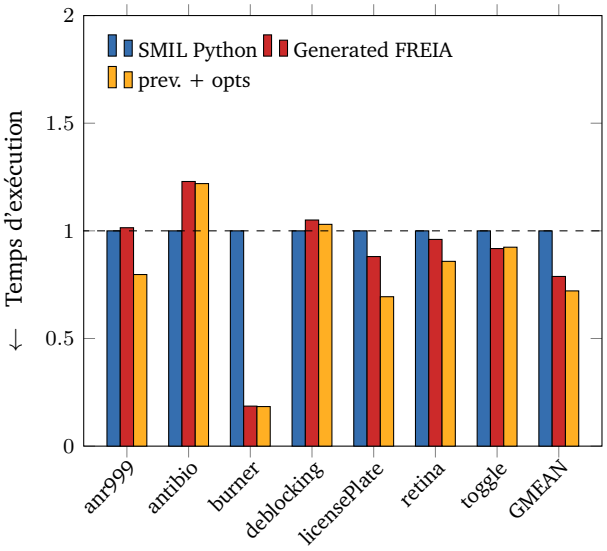


Figure 4.5 – Temps d'exécution relatifs d'applications de traitement d'images : (1) applications écrites en SMIL Python ; (2) puis converties en FREIA ; (3) avec optimisations de PIPS

La figure 4.6 compare les temps d'exécution de nos applications FREIA d'origine avec ceux de celles générées par `smiltofreia`, avec ou sans application de PIPS. Nous observons que le code généré est en moyenne aussi rapide que le code FREIA écrit à la main, et que les optimisations de PIPS ont des effets similaires sur les performances dans les deux cas. Dans le cas de « toggle », l'application générée bénéficie de l'introduction de la fonction `freia_aipo_mask()` dans l'API FREIA. L'application FREIA originale fait appel à la fonction `freia_aipo_replace_const()`, absente de SMIL et donc moins performante à l'exécution.

Ces résultats suggèrent que les applications SMIL peuvent bénéficier de la chaîne de compilation FREIA sans perte de performances par rapport au code FREIA écrit à la main. De plus, nos applications SMIL peuvent maintenant facilement être exécutées sur l'ensemble des cibles matérielles de FREIA (FPGAs, manycore et GPUs) sans modification de leur code.

4.6 Conclusion

Afin de profiter des cibles matérielles de FREIA, nous avons développé un compilateur de langage spécifique, `smiltofreia`, qui convertit une application SMIL écrite en Python vers FREIA. Ces travaux ont été réalisés en grande partie par Benoît Pin, ingénieur de recherche, dont nous avons sollicité la collaboration et qui est le principal auteur du logiciel. Dans ce cadre, notre contribution s'est apparentée à de la maîtrise d'ouvrage, puis, une fois les principales fonctionalités implémentés, à des tâches d'extension, de support et d'optimisation des performances. Nous avons également réalisé les diverses expérimentations qui ont nourri ce chapitre.

Grâce à ce compilateur, nous bénéficions directement des optimisations effectuées par le compilateur source-à-source PIPS sur les codes FREIA, visant à générer du code optimisé pour différentes architectures matérielles. Notre compilateur `smiltofreia` s'appuie sur des transformations d'arbre syntaxiques du code applicatif SMIL facilitées par le logiciel

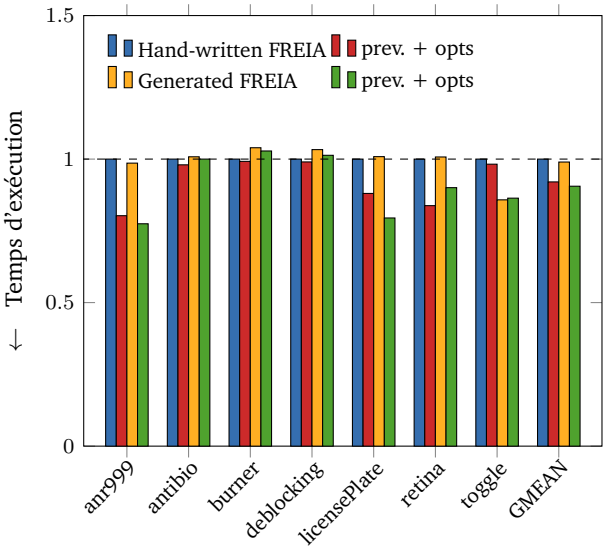


FIGURE 4.6 – Temps d'exécution relatifs d'applications de traitement d'images : (1) applications originales écrites en FREIA ; (2) avec optimisations de PIPS ; (3) applications réécrites en SMIL Python et converties en FREIA ; (4) avec optimisations de PIPS

RedBaron. Les appels vers les opérateurs de traitement d'images sont convertis vers leurs équivalents FREIA et les déclarations de variables sont générées au besoin. Puisque nous compilons depuis un langage dynamique (Python) vers un langage statique (C), les problèmes de typage et de polymorphisme ont du être résolus, parfois en ajoutant des contraintes sur le code d'entrée. D'autre part, les différences minimes entre les APIs des deux bibliothèques ont pu être comblées en ajoutant de nouvelles fonctions à FREIA, dont nous contrôlons le développement. Ces nouvelles fonctions ont, par ailleurs, eu pour conséquence d'améliorer les performances de certaines applications générées.

Nous avons évalué `smiltofreia` sur nos sept applications FREIA. Nous avons traduit à la main ces applications en SMIL Python, et nous montrons qu'elles comptent en moyenne plus de trois fois moins de lignes de code, illustrant la programmabilité supérieure de SMIL Python par rapport à FREIA. Grâce à `smiltofreia`, nous pouvons convertir à nouveau ces applications en FREIA et comparer la sortie de notre compilateur aux applications originales. Nous avons pu ainsi mesurer les performances de `smiltofreia` en exécutant nos applications sur le même processeur en appelant la même bibliothèque SMIL, au travers des wrappers Python et C. Les résultats illustrent la compétitivité du code FREIA généré avec le code Python initial, ainsi qu'avec le code FREIA original, et suggèrent qu'il peut se montrer parfois plus rapide. De plus, les optimisations offertes par PIPS permettent d'améliorer davantage les performances.

Au final, `smiltofreia` permet de porter des applications de traitement d'images écrites dans un langage haut niveau facilitant la programmation vers plusieurs accélérateurs matériels comme des processeurs manycore, des processeurs graphiques ou des FPGAs, via l'utilisation d'une chaîne de compilation intermédiaire fondée sur un langage de plus bas niveau. Nous parvenons ainsi à concilier la programmabilité offerte par l'interface Python de SMIL à la portabilité du framework FREIA, sans perdre en performance.

Chapitre 5

Parallélisme multiprocesseur à mémoire partagée

Moon river, wider than a mile
I'm crossin' you in style some day
Old dream maker, you heartbreaker
Wherever you're goin', I'm goin' your way

Two drifters, off to see the world
There's such a lot of world to see
We're after the same rainbow's end, waitin' 'round the bend
My huckleberry friend, Moon River, and me

— Johnny Mercer, *Moon River*

La diffusion des architectures à mémoire partagée dans les années 2000 a permis de nouveaux gains dans la puissance de calcul, au prix d'une parallélisation, souvent complexe à mettre en œuvre, du code des applications. Le modèle de programmation OpenMP [19] permet aux applications de facilement tirer parti de ces nouvelles architectures, grâce à des modifications à la marge de leur code source.

Ce chapitre se concentre sur le portage de la bibliothèque de traitement d'images SMIL [46, 47], qui utilise OpenMP, sur un des clusters de calcul du processeur MPPA, un nœud de calcul à mémoire partagée. À travers ce port, nous augmentons la portabilité des applications SMIL, qui peuvent désormais s'exécuter sur le MPPA. Contrairement au précédent chapitre 4, nos applications utiliseront cette fois-ci l'interface C++ de SMIL — aucun interpréteur Python n'a, à notre connaissance, été porté sur ce processeur. Cette interface de plus bas niveau ne profite cependant pas des facilités de programmation offertes par l'interface Python, qui reste la principale porte d'entrée pour écrire des applications SMIL.

Nous détaillons, en section 5.1, différentes méthodes permettant de tirer parti des architectures à mémoire partagée. Dans la section 5.2, nous présentons comment la bibliothèque de traitement d'images SMIL s'appuie sur ces outils afin d'optimiser ses performances. Enfin, la section 5.3 expose une méthodologie de portage de cette bibliothèque sur une architecture matérielle spécifique composée de seize cœurs de calcul, un cluster de calcul du MPPA, dont nous comparons les temps d'exécution et l'efficacité énergétique avec ceux d'un processeur standard.

5.1 Programmation parallèle sur architecture à mémoire partagée

Par définition, une architecture à mémoire partagée regroupe plusieurs unités de calcul accédant à une zone mémoire commune. Les différentes unités de calcul ont, ainsi, accès à l'ensemble de cette zone mémoire et, donc, aux données de leurs voisines sans nécessiter de transferts de données, coûteux en matière de performance.

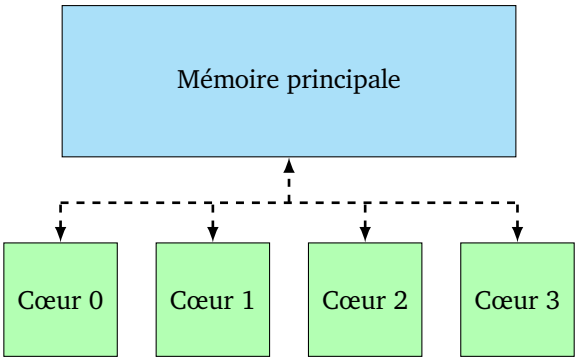


Figure 5.1 – Architecture à mémoire partagée : plusieurs unités de calcul accèdent à une même mémoire centrale

Ce type d'architecture facilite la parallélisation des calculs, notamment au travers du *parallélisme de données* : plusieurs unités de calcul réalisent une même suite d'opérations sur un ensemble de données. Les calculs sont « découpés » selon le nombre d'unités mises en jeu, qui se partagent les données. Ceci entraîne une diminution du temps d'exécution d'un facteur au mieux égal au nombre de ces unités travaillant en parallèle.

Pour tirer parti de telles architectures et ainsi améliorer les performances de leurs applications, les développeurs utilisent des interfaces de programmation qui sont, soit incluses dans la bibliothèque standard et l'environnement d'exécution des langages de programmation, soit fournies par le système d'exploitation lui-même. Celles-ci reposent généralement sur le concept de *thread*.

Moins liées au caractère partagé de la mémoire, les techniques de vectorisation des calculs ou de parallélisme d'instructions permettent également d'améliorer les performances des applications au prix d'une complexité accrue dans l'architecture des processeurs.

5.1.1 Les *threads*, briques de base du parallélisme de bas niveau

Les *threads*, ou *fils d'exécution*, représentent des sous-processus légers pouvant s'exécuter en concurrence, selon les ressources matérielles disponibles. Tous les threads créés lors de l'exécution d'un processus ont accès à la mémoire de ce processus. Pendant sa durée de vie, chaque thread dispose de sa propre pile d'instructions, indépendante de celle du processus maître. Pour éviter des conflits dans l'accès à cette mémoire, des primitives de synchronisation permettent de contraindre l'accès à certaines zones mémoire partagées.

Les programmes traditionnels s'exécutant séquentiellement ne comportent qu'un seul thread/processus tout au long de leur exécution. Puisqu'en général, un programme ne peut jamais être totalement parallélisé, un cas d'usage basique de threads consiste à accélérer certaines portions de code denses et intenses en calcul, comme des boucles. En pratique, au cours de son exécution, un programme multi-threadé va créer plusieurs threads qui vont

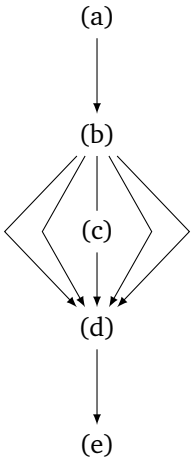


FIGURE 5.2 – Exemple d'un programme utilisant des threads : (a) le programme est lancé et il ne comporte qu'un seul thread ; (b) quatre threads sont démarrés par le thread principal ; (c) les cinq threads se partagent le travail ; (d) le thread principal attend la fin des autres threads ; (e) le programme se termine avec un seul thread

se partager les calculs du code ainsi parallélisé. À la fin de leur exécution, ils rendent, en général, la main au thread principal. Ce fonctionnement est illustré figure 5.2.

De nouvelles contraintes

Si la programmation parallèle par threads promet une réduction des temps de calcul en utilisant les cœurs disponibles, cela ne se fait pas sans contraintes. En effet, si, dans les programmes traditionnels, sans parallélisme, les instructions sont généralement exécutées dans l'ordre dans lequel elles apparaissent, ce n'est plus le cas dans les programmes parallèles. Ceci introduit plusieurs problèmes qui doivent être pris en compte par les concepteurs et développeurs logiciels. On peut notamment citer les *étreintes fatales* (*deadlocks*) et les *accès concurrents* (*race conditions*).

Accès concurrents Les accès concurrents interviennent lorsque deux ou plusieurs threads veulent modifier simultanément une donnée à une même adresse mémoire. Un thread peut alors mettre à jour la donnée sans que le ou les autres threads ne répercutent cette modification, ce qui conduit logiquement à des erreurs de calcul souvent très difficiles à détecter, puisqu'elles dépendent de l'ordre d'exécution des threads. Une solution consiste à utiliser un système de *verrou*, qui empêche plusieurs threads d'écrire simultanément à la même adresse mémoire, au prix d'une perte de performance.

Étreintes fatales Dans un système utilisant des verrous, des étreintes fatales interviennent si deux threads veulent chacun accéder à une ressource verrouillée par l'autre thread. Ces threads sont alors bloqués dans leur exécution ; le programme ne peut donc s'exécuter correctement. Pour éviter les étreintes fatales, il faut éviter de verrouiller deux données en même temps.

Outre ces nouvelles classes de bogues pouvant intervenir dès que plusieurs threads entrent en jeu, la programmation parallèle nécessite également de maîtriser de nouvelles interfaces de programmation permettant un contrôle explicite ou implicite des threads. Actuellement, deux interfaces standard pour la gestion des threads coexistent :

— *POSIX Threads* est une interface de programmation commune aux systèmes d'exploitation Unix et Linux qui fournit un ensemble de routines permettant de gérer explicitement les threads ;
— *OpenMP* [19] est une norme qui repose sur des directives préprocesseur offrant une interface de plus haut niveau.

Le standard POSIX

POSIX est un ensemble de normes pour des interfaces de programmation de systèmes d'exploitation mis en place à la fin des années 1980. Parmi celles-ci, la norme POSIX Threads [66], standardisée en 1995, décrit une interface explicite de gestion des threads. Cette interface fournit au langage de programmation de bas niveau C un ensemble de routines permettant de gérer finement un ou plusieurs threads pendant leur durée de vie.

```c
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 8

void *task(void *arg) {
  int thread_n = *(int *)arg;
  printf("Hello world from thread %d\n", thread_n);
  return NULL;
}

int main(void) {
  pthread_t threads[NTHREADS];
  int arg[NTHREADS];
  for (unsigned int i = 0; i < NTHREADS; i++) {
    arg[i] = i;
    pthread_create(&threads[i], NULL, task, &arg[i]);
  }

  for (unsigned int i = 0; i < NTHREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  return 0;
}
```

Extrait 5.1 – Exemple basique de programme C utilisant POSIX Threads : huit threads sont instanciés par le programme ; chaque thread affiche la chaîne de caractères « Hello world from thread "i" » avant de terminer son exécution

L'extrait 5.1 montre ainsi l'utilisation de threads au travers d'un exemple basique « *hello world* ». La fonction `void *task(void *arg);` est exécutée en parallèle par huit threads et, par conséquent, l'exécution du programme n'est plus déterministe : une autre exécution pourra montrer un autre ordre de complétion des threads. Dans cet exemple, les threads sont lancés par la routine

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

à laquelle on passe un pointeur vers la fonction `task`, ainsi qu'un pointeur générique vers une structure ou un tableau d'arguments. Un objet `pthread_t` est alors alloué et servira à

```
Hello world from thread 2
Hello world from thread 3
Hello world from thread 4
Hello world from thread 5
Hello world from thread 6
Hello world from thread 7
Hello world from thread 0
Hello world from thread 1
```

EXTRAIT 5.2 – Un résultat de l'exécution du programme extrait 5.1 : l'ordre des lignes n'est pas déterministe puisque tous les threads sont exécutés concurremment ; chaque ligne est en revanche complète car chaque appel à la fonction `int printf(const char *fmt, ...);` acquiert un verrou qui bloque l'accès à l'affichage aux autres threads

référencer et manipuler le thread. Le programme attend la fin de l'exécution des différents threads ainsi créés en appelant la routine

```
int pthread_join(pthread_t thread, void **retval);
```

Celle-ci permet également d'accéder aux structures renvoyées par l'exécution des threads, au travers de son argument `retval`.

Pour éviter les accès concurrents, la norme POSIX définit une structure de donnée appelée *mutex* (abrégé de *MUTual EXclusion*) permettant de verrouiller un ensemble d'instructions. Dans ce cas, un unique thread peut exécuter ces instructions à un instant donné, les autres restant bloqués dans leur exécution.

Outre la gestion des threads ainsi que les mutexes, l'interface POSIX Threads couvre également

— certaines propriétés des threads lors de leur exécution (adresse et taille de leur pile),
— des conditions d'exécution de certains threads dépendant de la valeur de données partagées et
— les barrières de synchronisation entre threads.

La richesse et l'exhaustivité de cette interface force néanmoins à une restructuration globale des programmes souhaitant l'utiliser pour des besoins non triviaux. Dans les cas où cette restructuration est trop coûteuse, l'utilisation de l'interface OpenMP peut s'avérer plus pertinente.

OpenMP, une norme fondée sur des directives préprocesseur

OpenMP [19] est une spécification de directives préprocesseur datant de 1997 et facilitant le parallélisme pour architectures à mémoire partagée. Elle est compatible avec les langages de programmation Fortran et C. Au départ se réduisant à la parallélisation des boucles de calcul `for`, elle s'est développée et supporte aujourd'hui le paradigme gestion de tâches, la vectorisation ainsi que le déport des calculs sur accélérateur dans les architectures hétérogènes.

L'utilisation d'OpenMP ne contraint pas le développeur à modifier de façon extensive le code à paralléliser. Si le programme s'y prête, la parallélisation via OpenMP consiste à décorer les boucles `for` par la directive préprocesseur

```
#pragma omp parallel for
```

Le code parallélisé sera alors automatiquement généré par le compilateur, qui va s'occuper d'effectuer un partitionnement de la boucle selon le nombre de threads disponibles, lancer et faire exécuter par chaque thread la portion de boucle correspondante. Évidemment, toutes les boucles `for` ne se parallélisent pas aussi simplement : par exemple, dès qu'une dépendance existe entre deux itérations, la parallélisation n'est pas possible.

```c
#include <stdio.h>
#include <omp.h>

int main(void) {

#pragma omp parallel
  {
    printf("Hello world from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

Extrait 5.3 – Exemple de code C utilisant OpenMP : parallélisation de l'affichage « Hello world from thread "i" »

Le programme présenté en extrait 5.3 est identique au programme POSIX Threads de l'extrait 5.1, à ceci près que, dans le cas présent, c'est l'environnement d'exécution d'OpenMP qui fixe le nombre de threads qui seront utilisés. La sortie d'une exécution de ce programme, est similaire à celle disponible en extrait 5.2 et mérite le même commentaire.

```c
void vecAdd(float *a, float *b, float *c, const unsigned int n) {
  unsigned int i;

#pragma omp parallel for
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Extrait 5.4 – Exemple de code C utilisant OpenMP : addition de deux vecteurs

L'extrait 5.4 est un exemple d'utilisation d'OpenMP afin de paralléliser des boucles de calcul, ici pour réaliser une addition de deux vecteurs. Les itérations vont être réparties sur les unités de calcul disponibles, divisant d'autant les temps d'exécution.

Des modèles de programmation alternatifs

Le parallélisme explicite offert par POSIX Threads et OpenMP s'est depuis quelque années effacé au profit d'autres modèles de programmation visant toujours les architectures à mémoire partagée.

Le modèle *gestion de tâches* disponible dans OpenMP, depuis la version 3.0 [96] datant de 2008, permet de paralléliser plus facilement des structures qu'on ne peut ramener à des boucles simples, par exemple des fonctions récursives. Ce modèle consiste à décrire un calcul sous la forme d'un graphe acyclique dirigé de tâches dépendantes, chaque tâche étant exécutée par un thread lorsque ses dépendances ont été elles-mêmes exécutées et que les ressources matérielles sont disponibles.

Le modèle de programmation flot de données, qui est étudié en détail au chapitre 6, est applicable aux architectures à mémoire partagée. OpenStream [101, 100] est ainsi une extension d'OpenMP offrant un parallélisme de type flot de données.

Depuis 2011, le support des architectures hétérogènes a été inclus dans la version 4.0 d'OpenMP. Une même application peut ainsi être exécutée automatiquement sur des accélérateurs matériels, si ceux-ci sont accessibles. Charge à l'environnement d'exécution OpenMP de déporter automatiquement les calculs.

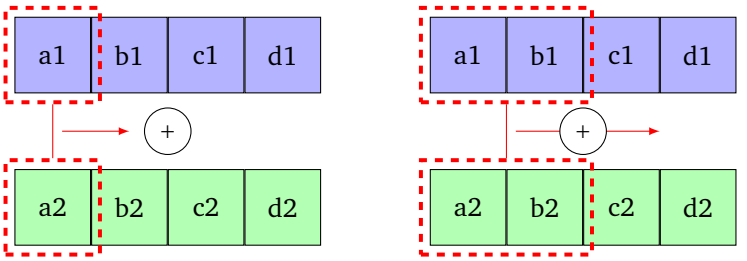
5.1.2 Le parallélisme dans les unités de calcul

En étant divisé en plusieurs cœurs, un processeur peut exécuter plusieurs processus en parallèle et ainsi gagner en performance. En jouant sur l'architecture des cœurs de calcul, d'autres types de parallélisme peuvent se révéler. Ainsi, la vectorisation permet à un processeur d'exécuter une instruction sur plusieurs données, tandis que le parallélisme d'instructions consiste à exécuter plusieurs instructions indépendantes en parallèle.

Un parallélisme faible grain : la vectorisation

Les processeurs exécutent des *instructions*, qui sont des opérations élémentaires appliquées à des données stockées dans des mémoires rapides appelées *registres*. L'arrivée des registres 128 bits et des jeux d'instructions vectorielles au début des années 2000 a permis d'améliorer significativement les performances des processeurs.

Une instruction vectorielle s'applique à une subdivision de registres appelée *vecteur*. Ainsi, avec des registres de 128 bits, une seule addition vectorielle peut s'appliquer sur quatre paires d'entiers 32 bits. La conséquence directe en est un gain de performance d'un facteur pouvant s'élever à la taille du vecteur. La vectorisation est, avant tout, utilisée pour accélérer les boucles de calcul, à condition de ne pas avoir de dépendance entre deux itérations successives.



(a) Pas de vectorisation : une addition par instruction, quatre instructions au total

(b) Vectorisation : deux calculs par instruction, deux instructions au total

FIGURE 5.3 – Effets de la vectorisation d'une addition de vecteurs sur le nombre d'instructions

Ainsi, l'addition de deux tableaux représentée en figure 5.3 peut être réalisée au moins deux fois plus rapidement en utilisant les instructions vectorielles adéquates.

Les instructions vectorielles ont été regroupées en extensions, que supportent ou non les processeurs au moment de leur parution. Les plus connues sont la série d'extensions SSE (« Streaming SIMD Extensions »), dont la première version date de 1999 et permet de réaliser quatre opérations 32 bits en une instruction, grâce à l'introduction de registres de 128 bits. Aujourd'hui, l'extension AVX-512 [72], datant de 2013 et supportée par les

processeurs manycore Intel Xeon Phi, permet des calculs vectoriels sur des registres de 512 bits.

Cependant, l'usage direct des instructions vectorielles est plutôt complexe : on leur préfère en général la vectorisation automatique prise en charge par le compilateur. En effet, en analysant les dépendances sur les boucles de calcul, les compilateurs modernes peuvent détecter les possibilités de vectorisation et générer les instructions correspondantes [8]. Dans les cas plus complexes, le modèle OpenMP dispose depuis sa version 4.0 [18] d'une directive `#pragma omp simd` qui génère une une version vectorisée des boucles ainsi annotées.

Les processeurs superscalaires et le parallélisme d'instructions

La manière dont un processeur exécute une séquence d'instructions peut également influer les performances des applications. Les instructions d'un processeur usuel peuvent être classées en plusieurs catégories, ordonnées selon leur complexité :

— les instructions arithmétiques ;
— les instructions pour calculs flottants ;
— les instructions de branchement ;
— les instructions de lecture *fetch* et *load* et d'écriture *store* vers la mémoire, ces dernières pouvant s'exécuter plusieurs centaines de fois plus lentement que les instructions arithmétiques.

Le parallélisme d'instructions consiste à exécuter différents types d'instructions en parallèle. Les processeurs superscalaires analysent, par exemple, dynamiquement les dépendances entre instructions, afin de les réordonnancer pour accélérer le flot d'instructions. Par exemple, deux instructions arithmétiques peuvent être interverties, si, d'une part, elles sont indépendantes et, d'autre part, les données de la seconde sont directement disponibles dans les registres du processeur. Un processeur superscalaire peut également effectuer un *fetch* et exécuter des instructions arithmétiques ou à calculs flottants pendant que les données sont transférées depuis la mémoire. Des systèmes de prédiction de branchement tentent de prédire la branche la plus probable, ce qui permet au processeur d'exécuter directement les instructions de cette branche et de vérifier *a posteriori* la validité de la prédiction.

Les processeurs « Very Long Instruction Word » ou VLIW disposent d'une architecture spécifique qui permet d'exécuter plusieurs instructions en même temps. Un nombre borné d'instructions indépendantes sont ainsi rassemblées dans un paquet ; les instructions d'un paquet sont exécutées en même temps. Dans ce cas, c'est au compilateur de générer les paquets de la façon la plus optimale possible. Contrairement aux processeurs superscalaires, l'ordonnancement des instructions est ici statique.

Le parallélisme d'instructions est, en général, directement supporté par le matériel et ne nécessite pas d'altérations de code pour en tirer parti.

5.2 SMIL, des traitements d'images nativement parallèles

La bibliothèque SMIL [46, 47] implémente certaines techniques décrites ci-dessus, afin de tirer au mieux parti des architectures à mémoire partagée actuelles. En effet, de par leur nature, les opérateurs de traitement d'images utilisés dans SMIL sont facilement parallélisables.

5.2.1 Le parallélisme dans le traitement d'images

En morphologie mathématique, théorie sur laquelle s'appuie SMIL, les opérateurs de traitement d'images fonctionnent pour la plupart sur le même principe : un parcours de

tous les pixels d'une ou plusieurs images d'entrée pour en extraire une image résultat de mêmes dimensions ou une unique valeur, dans le cas des réductions. Leur implémentation consiste ainsi en une boucle autour des pixels des images d'entrée, chaque itération étant généralement indépendante des autres. La parallélisation de ces opérateurs est donc aisée.

Le cas des réductions est plus complexe. Ces dernières consistent à extraire un unique scalaire d'une image, comme la valeur maximale ou minimale des pixels. La parallélisation de ces opérateurs se fait en deux étapes : l'opérateur est d'abord appliqué en parallèle sur une partition de l'image d'entrée, puis, dans un second temps, sur les valeurs obtenues à la première étape.

Le grain de la parallélisation est à choisir avec soin. Trop petit, il génère des surcoûts liés au changement de contexte des threads ; trop élevé, il peut consommer trop de mémoire. Dans le cadre du traitement d'images, la parallélisation s'effectue généralement au niveau des lignes d'images, voire au niveau d'un tuilage déterminé au préalable. C'est ainsi que procède SMIL pour paralléliser ses opérateurs.

5.2.2 SMIL, un parallélisme natif

La bibliothèque SMIL a été conçue dans le but de tirer parti des architectures modernes à mémoire partagée, notamment à travers l'utilisation du standard OpenMP et la vectorisation automatique des boucles de calcul.

OpenMP dans SMIL

SMIL fait un usage intensif d'OpenMP dans la plupart de ses opérateurs. Ainsi, dans chaque opérateur, les boucles autour des lignes d'une image sont décorées par la directive `#pragma omp for`. L'usage d'OpenMP dans SMIL est gardé par l'utilisation d'une instruction préprocesseur `#ifdef USE_OPEN_MP`, qui permet au besoin de se rendre totalement indépendant d'OpenMP, par exemple sur les plateformes ne disposant d'aucune implémentation.

Auto-vectorisation des boucles de calcul

La plupart des compilateurs modernes supportent l'auto-vectorisation des boucles de calcul. Cela évite d'écrire du code ciblant explicitement une extension vectorielle, qui peut ne pas être supportée par tous les processeurs. Cela laisse, en outre, au compilateur le soin d'optimiser le code sur toutes les cibles matérielles possibles et de rendre son code plus portable.

C'est depuis sa version 4.0 sortie en 2005, que le compilateur GCC a gagné le support de l'auto-vectorisation [8], au travers du flag *-ftree-vectorize*. Toutes les boucles ne peuvent être vectorisées par GCC ; les conditions suivantes doivent être notamment remplies :

— le nombre d'itérations doit pouvoir être facilement déterminé ;
— il faut éviter l'aliasing entre les données ;
— les appels de fonctions dans les itérations ne sont pas supportés ;
— cela ne peut concerner que la boucle interne d'un nid.

D'autres compilateurs, comme `icc` d'Intel, sont capables de vectoriser des boucles plus complexes.

Dans SMIL, l'auto-vectorisation permet d'accélérer les boucles parcourant les pixels d'une ligne d'image, chaque ligne étant traitée en parallèle via OpenMP.

5.3 Des applications SMIL parallèles sur un cluster de calcul du MPPA

Pour augmenter la portabilité de SMIL, nous avons réalisé un port de cette bibliothèque sur un cluster de calcul du processeur MPPA, un nœud de calcul à mémoire partagée. Ce port est facilité par la réutilisation du système de compilation de SMIL et par le support d'OpenMP par le processeur MPPA.

Afin de conserver la programmabilité offerte par SMIL, nous avons développé un environnement d'exécution pour transférer les données entre l'hôte et l'accélérateur qui abstrait les communications. Pour étudier les performances de ce port, nous avons traduit à la main nos sept applications FREIA présentées dans le chapitre 3 en SMIL C++. Nous comparons les exécutions de ces applications sur le MPPA et sur un processeur central conventionnel.

5.3.1 MPPA et OpenMP

Le processeur MPPA est composé de seize clusters de calcul composés chacun de seize cœurs de calcul organisés autour d'une mémoire centrale d'une taille de 2 Mo. Chacun de ces clusters est ainsi un nœud à mémoire partagée, les seize cœurs pouvant accéder indifféremment à cette mémoire centrale. Les outils de développement proposés par Kalray ciblant le MPPA Manycore supportent les spécifications POSIX Threads et OpenMP. Ainsi, des applications utilisant ces spécifications peuvent facilement tirer parti des seize cœurs d'un cluster de calcul.

Cependant, de telles applications sont restreintes à un unique nœud de calcul et ne peuvent donc utiliser le MPPA à son plein potentiel ; pour ce faire, des modifications substancielles du code source de ces applications est nécessaire. En effet, les clusters du MPPA communiquent entre eux via une interface de programmation par passage de messages spécifique ; les outils POSIX Threads et OpenMP sont inefficaces pour l'architecture à mémoire distribuée sous-jacente.

Pour construire des applications utilisant OpenMP sur l'accélérateur MPPA, il faut toutefois gérer les communications avec l'hôte au travers d'un des clusters d'entrée/sortie. L'usage de la bibliothèque de communication inter-clusters s'avère dans ce cas nécessaire.

5.3.2 Compilation croisée de SMIL vers un cluster de calcul

Pour porter SMIL sur le MPPA, il faut compiler son code source avec la chaîne de compilation fournie par Kalray. Puisque cette chaîne de compilation s'exécute sur la machine hôte et non sur la cible, il s'agit d'une compilation croisée. Le code source de SMIL est écrit en C++, mais permet également de générer des interfaces dans des langages de plus haut niveau, ainsi que décrit dans la sous-section 3.3.2. Toutefois, seuls C et C++ sont supportés par la chaîne de compilation Kalray pour le MPPA. Nous nous restreignons donc à compiler le cœur C++ de SMIL.

SMIL, ainsi qu'une grande partie des projets logiciels actuels se fondant sur des langages compilés, utilise un système de compilation. Ces outils fournissent des points d'entrée à la compilation de tout ou partie du code source, à l'installation des binaires obtenus ou bien au nettoyage du répertoire du projet, afin de supprimer les fichiers générés. Les plus connus de ces outils sont :

GNU Make, une surcouche à l'interface ligne de commande Bash, versatile mais rapidement limitée, souvent générée par les outils suivants [53] ;

les Autotools, une collection d'outils logiciels à sources ouvertes qui font toujours référence, malgré leur âge et leur complexité [5] ;

CMake, un système intégré plus simple et plus moderne, qui favorise la compilation croisée mais à la syntaxe critiquée [28] ;

Meson, projet très récent faisant la part belle à la rapidité d'exécution et à la facilité d'utilisation, avec une syntaxe inspirée de Python [114].

La bibliothèque SMIL s'appuie, quant à elle, sur le système CMake. Ce dernier propose plusieurs interfaces (texte, terminal, graphique) afin de modifier les options de compilation, ainsi qu'un langage de programmation pour automatiser certaines tâches. La bibliothèque SMIL s'appuie sur ce dernier outil afin de faciliter sa compilation. Dans ce cadre, CMake gère notamment la compilation conditionnelle et l'édition de liens vers des bibliothèques tierce-parties, fournissant, par exemple, des utilitaires d'encodage ou de décodage pour certains formats d'images (JPEG, PNG, TIFF), de téléchargement via HTTP (cURL) ou de boîtes à outils pour interfaces graphiques (qt). L'outil propose également la génération de la documentation, des interfaces vers d'autres langages de programmation.

En tirant parti des facilités qu'offre CMake pour la compilation croisée, nous avons compilé SMIL à destination d'un cluster de calcul du MPPA. La méthode suivie a consisté à écrire un fichier de configuration pour CMake, appelé « *toolchain file* », qui précise les emplacements des compilateur C et C++ à utiliser, ainsi que les options à leur passer. Nous avons préféré cette approche, qui valorise l'existant, plutôt que de réécrire tout ou partie des fichiers de configuration CMake de SMIL.

Une fois ce *toolchain file* écrit, nous pouvons configurer et compiler SMIL, sans oublier de désactiver les liens vers les bibliothèques tierces-parties inutiles pour un accélérateur matériel. Nous obtenons ainsi une version de SMIL compilée pour les clusters de calcul du MPPA. Le *toolchain file* utilisé dans ce cadre a été reproduit extrait 5.5. Il précise les chemins vers les compilateurs C et C++ de Kalray `k1-gcc` et `k1-g++`, ainsi que l'option de compilation pour cibler un cluster de calcul : `-mos=nodeos`. Nous définissons également une variable préprocesseur `__mppa__`, qui nous est utile, par la suite, afin d'isoler le code spécifique au MPPA que nous incluerons dans SMIL. Nous pouvons désormais compiler des applications SMIL pour les exécuter sur le MPPA.

5.3.3 Gestion des transferts d'images

Les clusters de calcul du MPPA n'implémentent pas de système de fichier, ni ne peuvent accéder au disque dur de la machine hôte. SMIL n'implémente pas les communications inter-clusters spécifiques à Kalray : les données (principalement des images) doivent être embarquées dans l'exécutable de nos applications, ce qui est peu pratique à l'usage. Afin de pallier cette limitation, nous avons élaboré puis implémenté un environnement d'exécution capable de transférer des images entre le disque dur de la machine hôte et le cluster de calcul exécutant nos applications.

Cet environnement d'exécution, illustré en figure 5.4, se compose de trois parties distinctes qui communiquent entre elles :

— sur le cluster de calcul du MPPA exécutant notre application, un environnement d'exécution inclus dans SMIL interceptant les lectures et écritures d'images et les transformant en commande à destination d'un cluster d'entrée/sortie ;
— sur ce cluster d'entrée/sortie, un exécutable chargé de relayer les commandes de lecture ou d'écriture et les données résultantes à la machine hôte ;
— sur la machine hôte, un exécutable utilisant SMIL pour charger et sauvegarder les images sur le disque dur et les transférer au MPPA à travers l'interface PCI-Express.

Les communications entre ces trois exécutables sont gérées par la bibliothèque `libmppaipc` fournie par Kalray. Cette dernière propose différentes structures et routines afin de

```
# Toolchain configuration for Kalray MPPA manycore chip

# the name of the target operating system
# use 'Generic' for embedded systems
SET (CMAKE_SYSTEM_NAME Generic)

SET (MPPA 1)

# which compilers to use for C and C++
SET (CMAKE_C_COMPILER $ENV{K1_TOOLCHAIN_DIR}/bin/k1-gcc)
SET (CMAKE_CXX_COMPILER $ENV{K1_TOOLCHAIN_DIR}/bin/k1-g++)

SET (CMAKE_C_FLAGS "-mos=nodeos -D__mppa__")
SET (CMAKE_C_FLAGS_MINSIZEREL "-Os -DNDEBUG")
SET (CMAKE_C_FLAGS_RELEASE "-O4 -DNDEBUG")
SET (CMAKE_C_FLAGS_RELWITHDEBINFO "-O2 -g")

SET (CMAKE_CXX_FLAGS "-mos=nodeos -D__mppa__")
SET (CMAKE_CXX_FLAGS_DEBUG "-g")
SET (CMAKE_CXX_FLAGS_MINSIZEREL "-Os -DNDEBUG")
SET (CMAKE_CXX_FLAGS_RELEASE "-O4 -DNDEBUG")
SET (CMAKE_CXX_FLAGS_RELWITHDEBINFO "-O2 -g")

SET (OpenMP_C_FLAGS "-fopenmp")
SET (OpenMP_CXX_FLAGS "-fopenmp")
```

Extrait 5.5 – Toolchain file vers les compilateur Kalray

faciliter les communications entre clusters via le *NoC* ou bien entre les clusters d'entrée/sortie et la machine hôte à travers l'interface PCI-Express.

Environnement d'exécution — machine hôte

L'exécutable hôte de notre environnement d'exécution accomplit principalement les deux tâches suivantes :

— au lancement de l'application, il va charger le code binaire ciblant le MPPA sur celui-ci et va faire exécuter par le cluster d'entrée/sortie l'environnement d'exécution lié ;
— pendant l'exécution de l'application, il va attendre des commandes de lecture ou d'écriture d'images envoyées par le cluster d'entrée/sortie, avant de les exécuter.

Pour faciliter le support des différents formats d'images, cet environnement d'exécution délègue à la version originale de SMIL sur l'hôte la gestion des entrées/sorties. Pour faciliter les appels à SMIL, nous avons écrit cet environnement d'exécution en C++.

Environnement d'exécution — cluster d'entrée/sortie

Notre environnement d'exécution sur le cluster d'entrée/sortie sert, tout d'abord, à lancer l'exécution de l'application courante sur le premier cluster de calcul du MPPA. Par la suite, il s'occupe principalement de relayer les commandes de lecture et d'écriture d'images provenant du cluster de calcul et à destination de la machine hôte. Pour cela, il s'appuie fortement sur la bibliothèque `libmppaipc`, afin de communiquer à travers l'interface PCI-Express et le *NoC*.

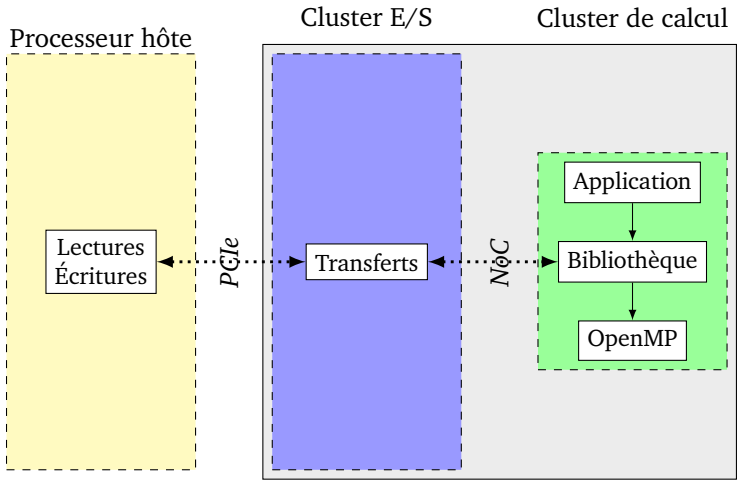


FIGURE 5.4 – Représentation de notre environnement d'exécution sur la machine hôte, le cluster d'entrée/sortie et le cluster de calcul : l'hôte et le cluster d'entrée/sortie sont esclaves du cluster de calcul

Environnement d'exécution — cluster de calcul

C'est sur le premier des seize clusters de calculs du MPPA que s'exécute l'application cible, qui embarque avec elle la bibliothèque SMIL compilée pour cette architecture. Un environnement d'exécution est là encore nécessaire pour :

— initialiser les communications avec le cluster d'entrée/sortie au lancement de l'application ;
— les finaliser à la terminaison ;
— intercepter les appels aux fonctions de lecture et d'écriture d'images de SMIL pour les transformer en communications avec le cluster d'entrée/sorties.

De façon à ne pas avoir à instrumenter le code des applications cibles, l'environnement d'exécution a été intégralement intégré dans SMIL. De cette manière, l'interception des commandes de lecture et d'écriture d'images est simplifiée et revient à trois lignes de code, ainsi que montré dans l'extrait 5.6 ; la fonction de détermination des formats d'images va toujours retourner le même gestionnaire d'images pour le MPPA, à condition que la variable préprocesseur `__mppa__` ait été définie. Cette variable garantit la portabilité des applications SMIL sur un processeur autre que le MPPA.

L'initialisation et la finalisation des communications avec le cluster d'entrée/sortie doivent intervenir respectivement le plus tôt et le plus tard possible durant l'exécution d'une application. Une solution simple consiste à instrumenter la fonction `main` des applications en y ajoutant des appels conditionnels vers des fonctions idoines. Cependant, cette approche nécessite d'altérer le code source des applications, nuisant ainsi à leur portabilité — altérations qui peuvent malgré tout facilement s'automatiser, mais au prix de dépendances logicielles supplémentaires.

SMIL contient une structure de données appelée `Core` qui implémente le patron de conception *singleton* : cette structure n'existe qu'en un seul exemplaire tout au long de l'exécution des applications qui l'utilisent. La structure en question étant appelée par la plupart des fonctions contenues dans SMIL, l'initialisation des communications avec le cluster d'entrée/sortie a été implémentée par extension du constructeur `smil::Core::Core()`, et,

```
namespace smil
{
  template <class T>
  ImageFileHandler<T> *getHandlerForFile(const char* filename) {
    string fileExt = getFileExtension(filename);

#ifdef __mppa__
    return new MPPAImageFileHandler<T>();
#endif // __mppa__

    if (fileExt=="BMP")
      return new BMPImageFileHandler<T>();

#ifdef USE_PNG
    else if (fileExt=="PNG")
      return new PNGImageFileHandler<T>();
#endif // USE_PNG

    // JPEG, TIFF and other image formats are managed here
  }
} // namespace smil
```

Extrait 5.6 – Interception des appels à la lecture et à l'écriture des données images

de même, leur finalisation par extension du destructeur `smil::Core::~Core()`. De cette façon, les canaux de communications entre le cluster de calcul et le cluster d'entrée/sortie sont automatiquement ouverts au début du programme et fermés à la fin.

Protocole de communication

Les exécutables sur l'hôte et sur le cluster d'entrée/sortie attendent des commandes envoyées par le cluster de calcul, qui exécute l'application cible. Ces commandes sont envoyées et reçues en utilisant la bibliothèque de communication inter-clusters de Kalray appelée `libmppaipc`. Cette bibliothèque propose des structures de données facilitant l'envoi de données entre différents clusters du MPPA.

Parmi ces structures, deux retiennent particulièrement l'attention :

`portal` permet à un cluster d'envoyer des données à plusieurs autres clusters ;

`sync` permet de facilement établir des barrières de synchronisation entre différents clusters.

À elles seules, ces deux structures permettent de bâtir des applications complexes utilisant tous les clusters du MPPA. Par exemple, considérons une application maître-esclave consistant à déporter un calcul sur plusieurs clusters du MPPA, le tout étant contrôlé par un cluster d'entrée/sortie. Dans ce cas, des objets `portal` peuvent êtres utilisés pour répartir des données d'entrée depuis la mémoire DDR attachée au cluster d'entrée/sortie vers les clusters de calcul correspondants et, également, pour effectuer les transferts et le regroupement des données résultats. Un objet `sync` sera dans ce cas utile pour signaler au cluster d'entrée/sortie que les clusters de calcul sont prêts à recevoir les données d'entrée.

Dans le cas de SMIL, ces structures servent à transmettre des commandes depuis le cluster de calcul exécutant l'application cible, ainsi que des données. Les commandes dont nous avons besoin sont de trois types :

read, pour transmettre une image depuis le disque dur de l'hôte au cluster de calcul en passant par le cluster d'entrée/sortie ;

write, pour écrire une image sur le disque dur depuis le cluster de calcul ;

quit, pour lancer la finalisation des communications et sortir des boucles pour arrêter l'exécution des binaires, d'abord sur le MPPA, puis sur le cluster d'entrée/sortie et enfin sur l'hôte.

Les commandes `read` et `write` sont envoyées par interception des fonctions SMIL idoines, comme détaillé extrait 5.6. La commande `quit` l'est, quant à elle, à la destruction de l'objet singleton `smil::Core`.

Dans les détails, les commandes utilisées correspondent à des structures de données dont l'un des champs caractérise le type de commande. Les champs supplémentaires servent à envoyer des informations pertinentes en sus de la commande, afin de limiter le nombre de communications. Par exemple, dans le cas de la commande `write`, les dimensions de l'image à écrire sont envoyées en même temps que la commande, de sorte que le cluster d'entrée/sortie puisse connaître la taille de l'image à recevoir. À l'aide de ces structures `portal` et `sync`, nous avons développé un protocole de transmission pour permettre un bon transfert des données images et une bonne exécution de ces commandes.

Pour mieux appréhender le fonctionnement de ce protocole de transmission, concentrons-nous sur le fonctionnement de la commande `read`.

1. (cluster de calcul) L'application veut lire une image sur le disque, caractérisée par son chemin dans le système de fichiers de l'hôte. Pour cela, elle utilise la primitive `smil::read(...);`.
2. (cluster de calcul) La bibliothèque SMIL répond à l'appel et le transmet au gestionnaire d'image pour le MPPA, qui court-circuite l'identification du format d'image.
3. (cluster de calcul) SMIL forme une commande `read` contenant la taille de la chaîne de caractère du chemin de l'image sur l'hôte et l'envoie au cluster d'entrée/sortie.
4. (cluster d'entrée/sortie) L'environnement d'exécution reçoit la commande, alloue une chaîne de caractères et prépare la réception du chemin.
5. (cluster de calcul) Le chemin de l'image est envoyé au cluster d'entrée/sortie.
6. (cluster d'entrée/sortie) Ce chemin est transféré à l'hôte.
7. (machine hôte) l'environnement d'exécution appelle la fonction non modifiée `smil::read(...);` avec le chemin de l'image en paramètre et récupère la structure de donnée image. Il transmet au cluster d'entrée/sortie ses dimensions, puis les données elles-mêmes.
8. (cluster d'entrée/sortie) Les dimensions de l'image fournies par l'hôte servent à allouer un tableau de pixels temporaire pour stocker les données reçues de l'hôte. Les dimensions sont alors retransmises au cluster de calcul, avant le tableau de pixels.
9. (cluster de calcul) Les dimensions servent à allouer un tableau de pixels qui recevra les données en provenance du cluster d'entrée/sortie.
10. (cluster de calcul) Ce tableau de pixels est converti en structure de données image de SMIL.
11. (cluster de calcul) L'application dispose désormais des données image dans la mémoire partagée.

La commande `write` est, quant à elle, de complexité moindre : la commande et les données suivent le même chemin, du cluster de calcul vers l'hôte.

Forces et limitations

L'approche suivie, qui consiste à réaliser une compilation croisée de la bibliothèque SMIL vers les clusters de calcul du processeur MPPA et de transférer les images à la demande via un environnement d'exécution ad hoc, garantit la portabilité des applications SMIL C++.

Via la compilation croisée, facilitée par CMake, la bibliothèque SMIL entière est directement portée sur l'accélérateur MPPA. De plus, la logique de communication inter-clusters étant cachée dans la bibliothèque SMIL, les applications sont réellement portable : aucune altération de leur code source n'est nécessaire pour les exécuter sur MPPA. En outre, grâce à OpenMP, les opérateurs SMIL peuvent tirer directement parti des seize cœurs du cluster de calcul ciblé. Enfin, cette approche peut être généralisée à toute autre bibliothèque logicielle, accédant ou non à un système de fichier.

Cependant, cette approche, si efficace soit-elle pour porter rapidement des applications SMIL sur le MPPA, souffre de plusieurs défauts. Elle ne fonctionne par défaut qu'avec des applications faisant exclusivement appel à SMIL : toute autre bibliothèque utilisée doit être au préalable compilée à destination du MPPA. D'autre part, la taille de l'exécutable obtenu après compilation, composé notamment de l'application, de SMIL, de l'environnement d'exécution OpenMP et de la bibliothèque standard C++, empiète sur l'espace mémoire réservé aux données dans le cluster de calcul, si bien que les images traitées sont de taille restreinte. Enfin, et non des moindres, notre système ne fonctionne pour l'instant que sur un seul cluster de calcul sur les seize disponibles. Il aurait théoriquement été possible d'exécuter en parallèle seize copies d'une application cible sur les seize clusters de calcul, à condition d'étendre notre environnement d'exécution sur le cluster d'entrée/sortie de sorte à ce qu'il effectue un tuilage des images, envoie à chaque cluster une portion d'image plutôt qu'une image entière lors de la lecture et recolle les images avant l'envoi à l'hôte pour écriture.

5.3.4 Tests de performance

Une fois ce port de la bibliothèque SMIL sur la puce MPPA achevé, il devient possible d'avoir accès à plusieurs métriques de performance. Pour cela, nous avons traduit en C++ SMIL les applications FREIA décrites dans la sous-section 3.3.1. À partir de ces applications, nous avons effectué quelques comparaisons pour ce qui concerne les temps d'exécution et la consommation d'énergie.

Scalabilité du cluster de calcul MPPA

Avec notre ensemble d'applications, il est possible de tester la scalabilité du système, c'est-à-dire sa réponse en matière de performances à une augmentation du parallélisme. En effet, l'environnement d'exécution OpenMP permet de préciser à l'exécution le nombre de threads qui doivent être lancés par le système. Nous avons ainsi testé la réponse du MPPA pour les temps d'exécution sur ces applications en faisant varier le nombre de threads de 1 à 16. Le temps d'exécution des calculs seul est mesuré ; le temps de transfert des images n'est pas pris en compte.

Les résultats sont disponibles dans la figure 5.5. Pour les applications considérées, l'utilisation d'OpenMP pour un faible nombre de threads est bénéfique au temps d'exécution, mais la taille faible laissée aux données ne permet pas d'exprimer le plein potentiel de la parallélisation. L'utilisation d'un grand nombre de threads engendre ici des problèmes de performance : la création des threads est, en effet, coûteuse et doit être contrebalancée par un grand nombre de calculs. Dans le cas présent, la taille des images et donc le nombre de calculs à effectuer par thread est trop faible.

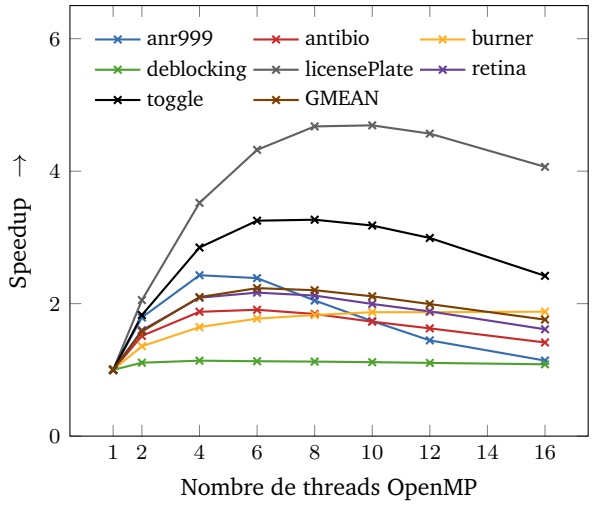


FIGURE 5.5 – Passage à l'échelle de SMIL/OpenMP sur cluster de calcul MPPA de 2e génération « Bostan » sur un échantillon de 7 applications de traitement d'images (référence : temps d'exécution sur 1 thread $= 1$)

Comparaison des deux générations de puces MPPA

Au cours de cette thèse, nous avons eu accès à deux générations de processeurs MPPA : la première, nom de code « Andey », sortie en 2012, et la seconde, nommée « Bostan », datant de fin 2015 [1]. Nous avons ainsi comparé les temps d'exécution des mêmes applications exécutées sur les deux générations de puces. Les résultats sont disponibles à travers le graphe de la figure 5.6.

Sur notre jeu d'applications, le saut de génération apporte un speedup d'environ $\times 6$ pour un thread, speedup qui se limite à $\times 2.5$ lorsque les seize threads du cluster de calcul cible sont utilisés.

Le cluster de calcul MPPA face à un processeur conventionnel

Enfin, nous avons comparé le MPPA au processeur de notre machine hôte, à savoir un processeur Intel Core i7-3820 CPU @ 3.60GHz de la génération Sandy Bridge datant de 2012. Ce processeur comporte quatre cœurs physiques et supporte l'hyperthreading, une technique qui permet à un cœur d'exécuter deux threads en parallèle. Ainsi, il peut exécuter jusqu'à huit threads en parallèle. OpenMP, là encore, permet de paralléliser directement les calculs de SMIL sur ces huit cœurs sans intervention de l'utilisateur.

Puisque nos applications SMIL sont portables, il suffit de les recompiler et de les exécuter sur l'hôte pour obtenir les temps d'exécution. Nous avons ainsi comparé les temps d'exécution de ces applications sur le MPPA de 2e génération et sur notre processeur hôte, avec 1 et 8 threads. Le graphique de la figure 5.7 montre ainsi qu'un cluster de calcul du MPPA est en moyenne entre cinq et six fois plus lent que le processeur Intel hôte.

Quant à l'énergie, la puce MPPA consomme au maximum environ une dizaine de Watts, tandis que l'enveloppe de dissipation thermique du processeur Intel s'élève à 130 W. Ce

1. Pour la petite histoire, ce sont les noms de montagnes dans les Alpes, non loin de Grenoble, où se situe le siège de Kalray.

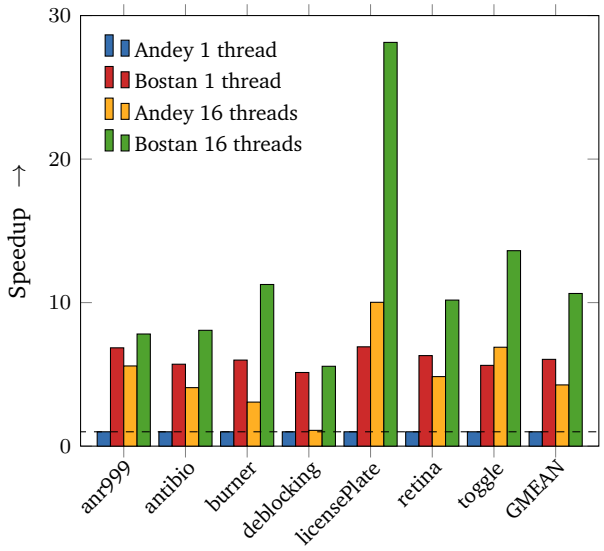


Figure 5.6 – Speedups de deux générations de clusters de calcul de la puce MPPA sur un échantillon de 7 applications SMIL : (1) Andey 1 thread (référence) ; (2) Bostan 1 thread ; (3) Andey 16 threads ; (4) Bostan 16 threads

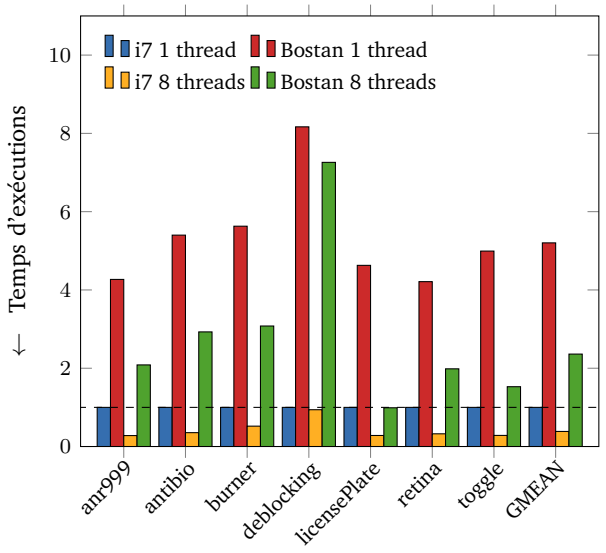
graphique compare également la consommation énergétique du MPPA et du processeur Intel sur notre jeu d'applications. En moyenne, le MPPA consomme donc au moins 30% de moins que le processeur Intel.

5.4 Conclusion

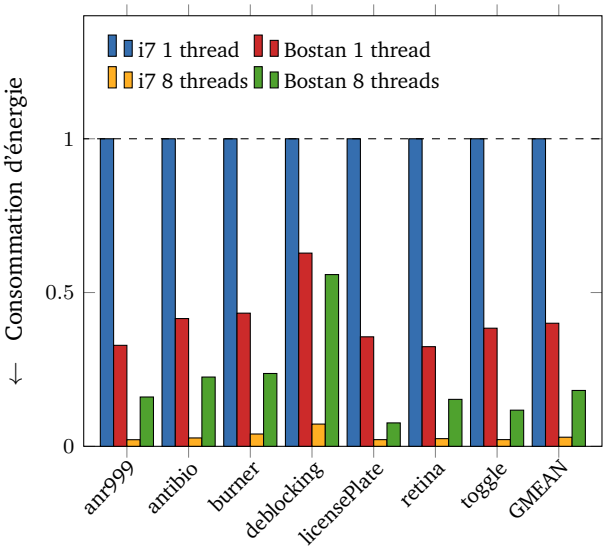
Dans ce chapitre, nous améliorons la portabilité de la bibliothèque SMIL en réalisant un port vers le nœud à mémoire partagée que constitue un cluster de calcul du processeur MPPA. Nous permettons ainsi d'exécuter des applications SMIL écrites en C++ sans modification de code source sur ce processeur, grâce à un environnement d'exécution qui abstrait les communications de données avec l'hôte. Nous facilitons, de cette façon, la programmabilité d'applications de traitement d'images sur le MPPA en cachant totalement la complexité de l'architecture sous-jacente grâce à notre environnement d'exécution.

Nous avons ainsi réalisé une compilation croisée de la bibliothèque SMIL à destination d'un cluster de calcul du MPPA en prenant appui sur CMake, l'outil de compilation utilisé par cette bibliothèque. Pour transférer les données entre l'hôte et le cluster de calcul cible, nous avons développé un environnement d'exécution en trois parties, dont l'une directement intégrée dans SMIL.

Puisque SMIL utilise OpenMP pour paralléliser certaines boucles de calcul et que le MPPA dispose d'un port d'OpenMP sur ses clusters de calcul, il est possible de profiter directement du parallélisme à mémoire partagée offert par ces nœuds de calcul. Nous avons réécrit en SMIL C++ les applications provenant de FREIA et listées dans la sous-section 3.3.1. Nous avons exécuté ces applications sur un processeur conventionnel et sur un cluster de calcul du MPPA. Nous avons mesuré les temps d'exécution de ces applications et déduit leur consommation énergétique. Selon cette dernière métrique, le MPPA se montre plus performant que le processeur central testé, malgré des temps d'exécution six fois plus lents.



(a) Temps d'exécution relatifs



(b) Consommation d'énergie relative

FIGURE 5.7 – Performance relative de « Bostan » face à Intel Core i7-3820, sur nos sept applications SMIL : (1) i7 1 thread (référence) ; (2) Bostan 1 thread ; (3) i7 8 threads ; (4) Bostan 8 threads

Chapitre 6

Le modèle flot de données

Tes pas, enfants de mon silence,
Saintement, lentement placés,
Vers le lit de ma vigilance
Procèdent muets et glacés.

Personne pure, ombre divine,
Qu'ils sont doux, tes pas retenus !
Dieux !… tous les dons que je devine
Viennent à moi sur ces pieds nus !

Si, de tes lèvres avancées,
Tu prépares pour l'apaiser,
A l'habitant de mes pensées
La nourriture d'un baiser,

Ne hâte pas cet acte tendre,
Douceur d'être et de n'être pas,
Car j'ai vécu de vous attendre,
Et mon cœur n'était que vos pas.

— Paul Valéry, *Les pas*

Le changement de paradigme associé à la fin de la course à la fréquence d'horloge des microprocesseurs et la multiplication des cœurs de calcul ont permis à d'anciens modèles de programmation de faire leur retour. Parmi ceux-ci, le modèle flot de données s'est imposé depuis quelques années comme particulièrement adapté pour représenter des algorithmes parallèles de traitement du signal. Ce modèle est présenté dans la section 6.1.

Pour aider à exprimer le parallélisme sur les 256 cœurs de sa puce MPPA, Kalray a proposé un langage de programmation flot de données appelé Sigma-C. Ce langage offre une couche d'abstraction des communications entre les différents clusters de calcul, d'entrée/sortie et la machine hôte, permettant ainsi de développer rapidement des prototypes d'applications fonctionnelles. Les principales caractéristiques de ce langage sont exposées dans la section 6.2.

Nous avons réalisé une chaîne de compilation qui, à partir d'applications utilisant l'interface de programmation FREIA, génère automatiquement le code Sigma-C correspondant.

À partir d'expériences sur les applications FREIA, des optimisations ont été testées. Les résultats ont fait l'objet d'un article à la conférence *Languages and Compilers for Parallel Computing* [122]. Nous décrivons les différents éléments de cette chaîne, ainsi que les résultats de ces expériences, dans la section 6.3.

Cette chaîne de compilation permet aux applications FREIA d'être automatiquement portées sur l'architecture complexe du processeur MPPA. L'interface de programmation de FREIA, bien qu'embarquée dans un langage de programmation de bas niveau, facilite alors l'écriture d'applications de traitement d'images à destination du processeur MPPA. Enfin, les optimisations réalisées sur la chaîne de compilation améliorent les temps d'exécution et la consommation énergétique des applications.

6.1 Le modèle

Le flot de données est un modèle de programmation de haut niveau qui s'éloigne du fonctionnement interne de la machine, à l'inverse du modèle impératif. Les applications flot de données, ainsi que décrit dans [116], sont principalement caractérisées par

— de larges quantités de données à traiter,
— des chaînes de calculs indépendants et
— une structure stable.

Les applications flot de données se décomposent en un graphe orienté dont les arcs représentent des transferts de « données » (au sens large) et les sommets des « actions » effectuées sur ces données. L'exécution d'une telle application s'apparente ainsi à la consommation et à la production de données par les sommets de ce graphe, d'où le nom « flot de données ». Les sommets sont également parfois appelés producteurs/consommateurs, acteurs, agents ou filtres. En figure 6.1 est ainsi schématisé un graphe flot de données correspondant à une addition-multiplication entre trois variables.

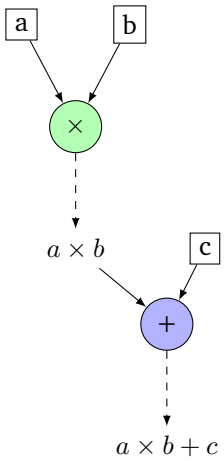


Figure 6.1 – Représentation du graphe flot de données de l'addition-multiplication des variables a, b et c

Les applications flot de données sont en général développées en deux étapes :

1. on commence d'abord par implémenter la logique interne des producteurs/consommateurs ;

2. on connecte différentes instances de ces producteurs/consommateurs sous la forme d'un graphe applicatif.

L'instanciation des producteurs/consommateurs sous la forme d'un graphe rappelle la programmation orientée objet, tandis que l'application d'une succession de filtres sur un ensemble de données sans état global rattache le modèle flot de données à la programmation fonctionnelle. En outre, la logique interne des producteurs/consommateurs est souvent codée avec un langage simili-impératif, afin de décrire plus précisément les transformations réalisées sur les données.

Les langages flot de données s'exécutent facilement sur les architectures multicœurs actuelles, ce qui a contribué à son renouveau. Chaque acteur peut être exécuté indépendamment des autres dès que le nombre de données en entrée est suffisant. Le parallélisme de données peut facilement s'exprimer à travers des constructions de type *split/join* qui séparent et fusionnent de grandes quantités de données qui seront traitées individuellement en parallèle. Une suite d'opérations sur un même ensemble de données peut, dans le modèle flot de données, se paralléliser en pipeline : plutôt que chaque opérateur s'exécute sur l'ensemble des données d'entrée, il est exécuté sur une partition, ce qui permet à l'opérateur suivant dans la chaîne d'accéder aux premières données traitées plus rapidement. Si plusieurs chaînes de calcul sont indépendantes, elles peuvent également être exécutées en parallèle dans le modèle flot de données.

C'est au milieu des années 1970 que Kahn [80] a pour la première fois décrit un réseau composé de producteurs/consommateurs de données agissant sur des files d'attentes de données. Le modèle a ensuite évolué vers, d'une part, le flot de données synchrone [86, 92] et, d'autre part, le flot de données cyclo-statique [16]. Le premier consiste à fixer le nombre de données consommées et produites par acteur, ce qui permet de déterminer statiquement un ordonnancement fini des processus, d'éviter ainsi les étreintes fatales et donc d'assurer la sûreté des programmes. Ainsi, les langages flot de données synchrones sont particulièrement populaires pour les applications industrielles. Parmi ceux-ci, on peux citer LUSTRE [70], Signal [85] et OpenStream [100]. Le modèle flot de données cyclo-statique est plus relâché, mais contraint, quant à lui, que le nombre de données consommées et produites par acteur suive un cycle. Le langage Sigma-C [56, 7] présenté plus bas entre dans cette catégorie.

D'autres langages ou frameworks se rattachent au modèle flot de données : DAGuE [21] et FlumeJava [24] permettent ainsi d'exprimer et d'optimiser des graphes de tâches pour des applications hautes-performances. Le flot de données est, en outre, utilisé dans le champ de l'apprentissage artificiel pour décrire des réseaux neuronaux profonds. On peut notamment citer Theano [10, 12], Tensorflow [88] et CNTK [2]. L'intérêt porté à ce modèle pour le traitement d'images commence d'ailleurs à dépasser le domaine de la recherche scientifique : le consortium industriel Khronos Group développe ainsi depuis 2014 OpenVX [62], une spécification flot de données dédiée à la vision par ordinateur.

6.2 Deux exemples de langages flot de données

Nous décrivons dans cette section, plus en détail, la syntaxe de deux langages flot de données : Streamit et Sigma-C. Ces deux langages, développés à dix ans d'intervalle, proposent des structures communes, afin de tirer plus efficacement parti des architectures multicœurs et distribuées modernes. Nous nous focaliserons davantage sur Sigma-C, base de nos travaux référencés dans ce chapitre.

6.2.1 StreamIt

Le langage flot de données StreamIt [116, 115] a été développé par le MIT depuis le début des années 2000. StreamIt repose sur la notion de *filters*, qui sont des blocs élémentaires de calcul. Ces filtres peuvent être connectés les uns aux autres en utilisant les structures spécifiques suivantes :

le *split/join*, qui sépare le flot de données pour les traiter en parallèle et joint les résultats pour obtenir un flot unique ;

le *pipeline*, qui connecte plusieurs filtres à la suite ;

la *feedback loop*, qui implémente une boucle de rétroaction.

Un exemple minimal de programme StreamIt, tiré de [110], est disponible dans l'extrait 6.1. Dans celui-ci, deux filtres `IntSource` et `IntPrinter` sont définis. Une structure pipeline `Minimal` est ensuite définie pour connecter les deux filtres l'un à la suite de l'autre. La fonction `work push{}` détermine le type, le nombre ainsi que l'origine des données produites par le filtre courant. De même, la fonction `work pop{}` consomme les données en entrée du filtre.

```
1  void->void pipeline Minimal {
2    add IntSource;
3    add IntPrinter ;
4  }
5
6  void->int filter IntSource {
7    int x;
8    init { x = 0; }
9    work push 1 { push(x++); }
10 }
11
12 int->void filter IntPrinter {
13   work pop 1 { print(pop()); }
14 }
```

Extrait 6.1 – Programme StreamIt minimal

Le compilateur StreamIt cible les architectures multicœurs ainsi que les clusters de calcul. Dans le cas d'un processeur multicœurs, plusieurs filtres sont, le cas échéant, assemblés dans un thread, le nombre total de threads équivalant alors au nombre de cœurs.

Diverses modifications de la structure des applications ont été implémentées dans le compilateur [55, 54], afin de rapprocher le graphe sous-jacent de l'architecture cible. Nous pouvons notamment citer la fusion de chaînes d'opérateurs connexes, visant à réduire le nombre de communications. Celle-ci est parfois suivie d'une passe d'insertion de structures split/join, afin d'augmenter le parallélisme de données. Nous avons suivi une approche similaire dans le cadre du langage Sigma-C sur la cible à 256 cœurs MPPA de Kalray.

6.2.2 Sigma-C

Sigma-C [56] est un langage de programmation flot de données cyclo-statique. Il a été développé par le laboratoire CEA LIST pour cibler des processeurs manycores, et notamment le processeur MPPA. Il vise à faciliter le portage d'algorithmes parallèles sur le MPPA grâce à une abstraction des cœurs cibles et des communications entre ces derniers. Il a été conçu à

la base comme représentation intermédiaire pouvant être générée à partir d'un langage de plus haut niveau, et c'est ainsi que nous nous en sommes servis.

Sigma-C est très proche du célèbre langage C, dont il reprend en grande partie la syntaxe. Plus précisément, il repose sur la norme C89 datant de 1989 — révisée notamment en 1999 et en 2011. Sigma-C ajoute deux nouvelles structures au C : les *agents*, qui correspondent aux acteurs du modèle flot de donnée, et les *subgraphs*, des graphes orientés d'agents.

Les agents

Les unités de base d'un programme Sigma-C sont appelés *agents*. Ce sont les producteurs/consommateurs du modèle flot de données. Un agent dispose de plusieurs canaux d'entrée et de sortie au travers desquels il consomme et produit des données.

L'exécution d'un agent peut se rapporter à une machine à états. À chaque état correspond la consommation et la production d'un nombre statiquement connu de données et fait intervenir un ou plusieurs canaux d'entrée ou de sortie. Pour chaque état, une fonction définit la transformation des données consommées pour obtenir les données produites.

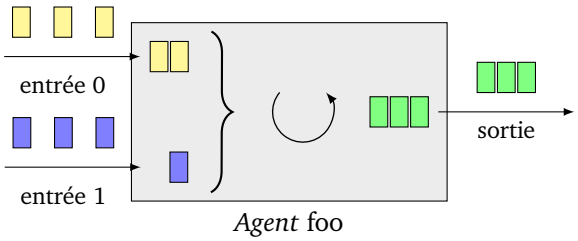


FIGURE 6.2 – Représentation schématique d'un agent à deux entrées et une sortie

Une représentation schématique d'un agent est disponible figure 6.2. Celui-ci dispose de deux canaux d'entrée et d'un canal de sortie. On peut imaginer que cet agent ne dispose que d'un seul état et que celui-ci consiste à prélever deux données (jaunes) sur sa première entrée et une donnée (bleue) sur la seconde pour les transformer en trois données (vertes), qui seront transmises via son canal de sortie.

Un code Sigma-C correspondant à l'agent schématisé figure 6.2 et décrit plus haut est présenté extrait 6.2. Ce code peut se décomposer en deux parties : le premier bloc, précédé du mot-clef `interface`, et le reste, composé de deux fonctions, `doit` et `start`, qui servent à manipuler concrètement les données.

Le bloc `interface` Afin de décrire plus précisément les échanges de données entre les agents, ceux-ci comportent un bloc `interface` dans lequel sont déclarés les différents canaux d'entrée/sortie, ainsi que le nombre de données consommées et produites à chaque état. Les déclarations des canaux d'entrée/sortie sont précédées par les mots-clefs `in` et `out` avec une indication du type de données échangées entre chevrons. Il est possible de déclarer des tableaux de canaux, à condition que leur nombre soit connu à la compilation. Le sous-bloc `spec` spécifie une liste d'états entre accolades, ces états correspondant à une liste de canaux d'entrée/sortie assortis d'un nombre lui aussi statiquement connu de données à prélever sur les canaux d'entrée et à produire sur les canaux de sortie. Pour comprendre plus aisément, reportons nous à la figure 6.2 : les couleurs jaune, bleue et verte des données schématisent le type de ces données. Dans l'extrait 6.2, toutes les données sont des entiers signés (`int`). De plus, la clause `spec` ne déclare qu'un seul état `{input0[2], input1, output[3]}`,

```
1  agent foo() {
2
3    /* describe agent interface */
4    interface {
5      /* define i/o channels */
6      in<int> input0, input1;  /* 2 input integer channels */
7      out<int> output;         /* 1 output integer channel */
8
9      /* declare flow scheduling */
10     spec {
11       {input0[2], input1, output[3]}
12     };
13   }
14
15   /* do something! */
16   void doit() exchange (input0 inp0[2],
17                         input1 inp1,
18                         output outp[3]) {
19     outp[0] = inp0[0];
20     outp[1] = inp1;
21     outp[2] = inp0[1];
22   }
23
24   /* infinite loop */
25   void start() {
26     doit();
27   }
28
29 }
```

Extrait 6.2 – Exemple d'agent en Sigma-C

ce qui correspond à consommer deux entiers sur la première entrée, consommer un entier sur la seconde et produire trois entiers sur la sortie. Il est quelquefois utile de répéter un état un nombre fixé de fois. La déclaration `spec{{input0}, (4){output}};` répète quatre fois la production d'une donnée sur le canal `output`, avant de consommer une donnée sur le canal `input0`. La transition d'un état à un autre étant sensiblement coûteuse, il faut veiller à traiter un grand nombre de données à chaque état, sous contrainte notamment de mémoire disponible. Sigma-C permet également certains usages plus avancés, comme la gestion d'un cache dans les canaux d'entrée et de sortie ou la copie implicite d'une entrée vers une sortie.

Les fonctions de manipulation des données Le bloc `interface` ne faisant que déclarer les données produites et consommées par l'agent correspondant, il faut, en Sigma-C, définir des fonctions qui vont accéder et manipuler ces données. Ces fonctions, sont en Sigma-C, caractérisées par le mot-clef `exchange` et correspondent directement aux états définis dans la clause `spec`. La fonction `doit` de l'extrait 6.2 en est un parfait exemple. Ces fonctions prennent deux listes d'arguments : une liste d'arguments « classiques » comme en C, ainsi qu'une liste de données tirées des canaux d'entrée/sortie déclarée après le mot-clef `exchange`. En accédant aux données des canaux, ces fonctions `exchange` permettent de les manipuler directement : la fonction `doit` insère la donnée du canal `input1` entre deux données du canal `input0`. La fonction `void start(void);` est obligatoire en Sigma-C ; elle permet de préciser l'ordre d'appel des fonctions `exchange`. Certaines fonctions

`exchange` peuvent correspondre au même état ; la fonction `start` permet de lever cette ambiguïté en appelant la bonne fonction au bon moment. L'exécution d'un agent consiste alors à effectuer une boucle infinie autour d'un appel à cette fonction `start`. Un bloc `init`, exécuté une seule fois au lancement de l'application, permet d'initialiser les variables internes de l'agent. Un agent, s'il représente une brique de base d'une application Sigma-C, doit comporter au minimum un canal d'entrée ou de sortie. En Sigma-C, une application est donc composée d'au moins deux agents qui se transfèrent des données. Afin de représenter ces échanges entre différents agents, Sigma-C fait intervenir une structure appelée *subgraphs*.

Les subgraphs

Un subgraph est un graphe orienté pouvant être cyclique dont les nœuds sont des agents et les arcs, un échange de données entre ces agents. Un subgraph peut disposer, comme un agent, d'une interface avec des canaux d'entrée/sortie, de manière à pouvoir le réutiliser dans d'autres subgraphs. Ainsi, un subgraph peut, en s'utilisant comme un agent « cacher », son fonctionnement et ses échanges de données internes et, par là, agir comme un *wrapper* autour de sa structure interne.

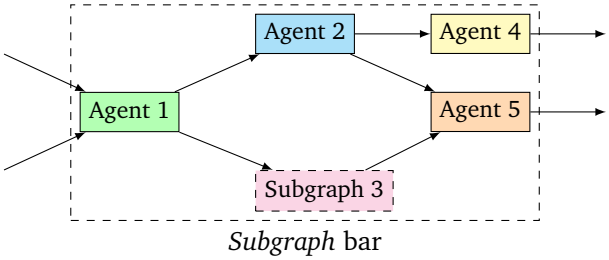


FIGURE 6.3 – Représentation schématique d'un subgraph à cinq agents

Une représentation schématique d'un subgraph de cinq agents est visible sur la figure 6.3. Ce subgraph dispose d'une interface comportant deux canaux d'entrée et deux canaux de sortie. L'un des agents interne est en fait un subgraph, ceci afin d'illustrer leur réutilisation.

L'extrait 6.3 est un exemple (incomplet) du code correspondant à la figure 6.3. On notera la présence d'un bloc `interface`, qui sert, par analogie avec les agents, à déclarer les canaux d'entrée/sortie. Une section `map` permet de déclarer les agents internes en utilisant le mot-clef `new` suivant une syntaxe rappelant C++ et Java, ainsi que les connexions entre l'ensemble des canaux d'entrée/sortie de ces agents à l'aide de la fonction `connect`. Les canaux de sortie des agents doivent être reliés à des canaux d'entrée, à l'exception des canaux du subgraph courant, qui doivent être reliés à des canaux internes du même type.

En Sigma-C, un subgraph spécifique nommé `root` joue le même rôle que la fonction `main` en C : il sert de point d'entrée à l'application. Le subgraph `root` ne dispose pas de canaux d'entrée/sortie ; tous les échanges de données doivent s'effectuer entre ses composants.

Sigma-C autorise différents subgraphs non connexes (sans interface). Cela permet, notamment, à plusieurs applications de s'exécuter au même moment sans s'échanger de données.

La bibliothèque standard

Pour accélérer le développement d'applications, Sigma-C fournit des agents utilitaires optimisés. Ceux-ci peuvent se classer dans deux catégories : les agents de gestion du flot et les agents d'entrée/sortie.

```
subgraph bar() {

  /* describe subgraph interface */
  interface {
    /* ... */
  }

  /* describe internal agents and connections */
  map {
    /* instanciate agents */
    agent a1 = new Agent1();
    agent a3 = new Subgraph3();
    /* ... */

     /* connect agents to subgraph interfaces */
    connect (input0, a1.input0);
    connect (a5.output, output1);
    /* ... */

    /* connect agents */
    connect (a1.output0, a2.input);
    connect (a3.output, a5.input1);
    /* ... */
  }

}
```

Extrait 6.3 – Exemple de subgraph en Sigma-C

Agents de gestion du flot Ces agents permettent de gérer plus finement le flot de données, par exemple en divisant un flot et en redirigeant les données qui y transitent.

Split L'agent Split a une entrée et plusieurs sorties. Chaque donnée reçue sur l'entrée est envoyée à une sortie différente.

Dup Similairement, l'agent Dup compte une entrée et plusieurs sortie. Chaque donnée reçue en entrée est envoyée à toutes les sorties.

Join Cet agent a plusieurs entrée et une unique sortie. Les données reçues sur chaque entrée sont recombinées vers la sortie dans l'ordre des entrées.

Sink Ne disposant que d'une seule entrée, le Sink permet de se débarrasser des données inutiles.

L'utilisation de ces agents permet de paralléliser un flot de données. Par exemple, une application comportant deux chaînes de calculs indépendantes mais agissant sur les mêmes données pourra tirer profit de l'agent Dup (parallélisme de tâches). De même, les agents Split et Join permettent de multiplier une chaîne de calculs, et ainsi tirer parti du parallélisme de données.

Agents d'entrée/sortie Ces agents écrivent ou lisent dans la mémoire associée.

StreamReader Cet agent ne dispose que d'une sortie. Il lit des données à une adresse fixée à l'avance dans la mémoire la plus proche et les transmet par paquets.

StreamWriter À l'inverse, cet agent ne compte qu'une entrée. Il reçoit les données par paquet et les écrit dans la mémoire à une adresse spécifique.

Le principal problème posé par ces agents est que l'adresse mémoire utilisée doit être connue à la compilation. Il faut donc précharger les données utiles en mémoire avant le lancement du programme et les décharger après sa fin. En combinant les agents de la bibliothèque standard et quelques agents écrits à la main dans un ou plusieurs subgraphs, on obtient une application Sigma-C complète.

Placement des agents

Conjointement développé avec le processeur MPPA, Sigma-C peut s'exécuter également sur architecture x86. Lorsque le MPPA est utilisé en tant qu'accélérateur matériel ou coprocesseur, les données ne sont, au départ, accessibles que depuis le processeur hôte et il faut les transférer aux clusters de calculs du MPPA pour effectuer les traitements.

Sigma-C permet de placer les agents sur l'une des trois cibles supportées, à savoir le processeur hôte, les clusters d'entrée/sortie et les clusters de calcul. Tout transfert entre l'hôte et les cœurs de calcul doit alors passer par un cluster d'entrée/sortie. L'avantage de Sigma-C est qu'il abstrait totalement, d'une part, les transferts PCI-Express entre l'hôte et le MPPA et, d'autre part, les transferts NoC inter-clusters.

Sur le MPPA, les agents sont placés à raison d'un par cœur de calcul. Si cela ne suffit pas, par exemple lorsqu'une application a plus de 256 agents, il est possible d'augmenter cette valeur à la compilation, au détriment des performances. Les cœurs des clusters d'entrée/sortie étant plus performants, plus d'agents peuvent y être placés, au risque de défavoriser les communications avec le processeur hôte.

Une application Sigma-C minimale consiste donc en un subgraph composé d'agents placés sur l'hôte, lisant ou écrivant les données sur le disque dur et d'agents de transfert sur les clusters d'entrée/sortie qui relayeront les données aux et depuis les clusters de calcul.

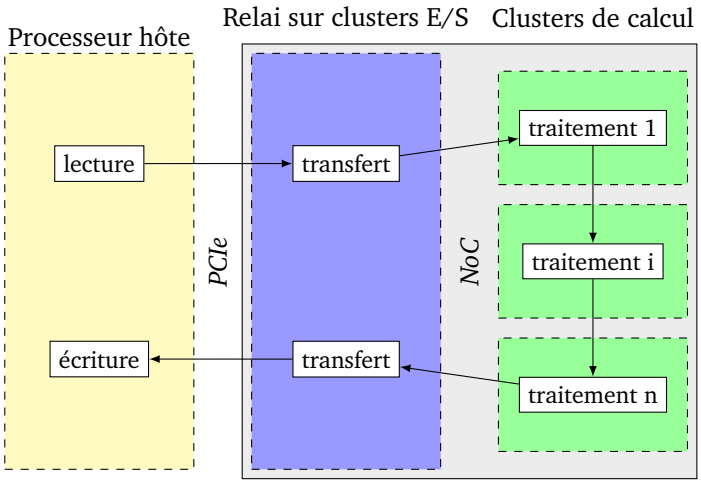


FIGURE 6.4 – Principe de fonctionnement d'une application minimale en Sigma-C utilisant le MPPA comme coprocesseur

Performances

Dans le cadre du modèle flot de données et, plus spécialement, des langages asynchrones comme le Sigma-C, la performance des applications dépend de celle de l'agent le plus lent,

qui va imposer son rythme aux autres agents. La détection et l'optimisation de ces goulots d'étranglement est généralement un processus clef pour améliorer ces performances. Cela signifie également qu'il est possible de « ralentir » légèrement les agents les plus rapides, sans impact sur les performances.

Les temps d'exécution dépendent également de la taille et de la fréquence des communications entre agents — les transitions entre états. De plus, deux agents liés peuvent se situer sur le même cluster, et dans ce cas les communications se réduisent à l'accès à la mémoire partagée, ou bien sur deux clusters différents, et les données sont transférées au travers du réseau sur puce.

Quelques défauts

Le développement d'applications pour le MPPA est en général grandement facilité par l'utilisation de Sigma-C. La gestion du parallélisme est inhérente au langage ou bien facilement accessible et exprimable à l'aide des agents Split et Join. Cependant, ce modèle souffre de plusieurs inconvénients.

D'une part, comme Sigma-C est un nouveau langage de programmation, il est nécessaire de réécrire complètement les applications que l'on veut porter sur le MPPA. En effet, traduire une application C classique en Sigma-C pour la paralléliser n'est pas chose simple, contrairement à l'utilisation de solutions à directives préprocesseur, comme OpenMP. Un effort préalable de portage est donc indispensable.

De plus, le langage souffre de sa trop grande staticité : les quantités de données transférées entre agents doivent être connues à la compilation. Cela créée une dépendance forte entre l'application et ses données et, parfois, impose de devoir recompiler à chaque changement du jeu de données d'entrée. Ceci implique, pour des applications de traitement d'images, de connaître à l'avance les dimensions des images, afin de correctement accéder aux voisins des lignes précédentes et suivantes et gérer les bords pour les opérateurs au voisinage. En outre, cela interdit les boucles dynamiques à l'échelle d'un subgraph.

Enfin, dernier inconvénient et non des moindres, le langage a cessé depuis mi-2015 d'être supporté par Kalray, qui a préféré se concentrer à la place sur le support d'OpenCL, plus demandé par sa clientèle. Il est ainsi impossible d'utiliser Sigma-C avec les dernières versions de la chaîne d'outils de Kalray et, notamment, de profiter des améliorations apportées par la puce MPPA de seconde génération « Bostan ».

6.3 Une nouvelle cible pour FREIA

Notre contribution principale a ici consisté à faire de Sigma-C une cible supplémentaire du framework FREIA de sorte que toute application écrite au-dessus de FREIA puisse être exécutée automatiquement sur la puce MPPA sans nécessiter de retoucher le code initial. Nous avons opéré en plusieurs étapes successives :

1. nous avons traduit en Sigma-C les opérateurs de base de traitement d'images de FREIA ;
2. nous avons étendu le compilateur source-à-source PIPS pour générer automatiquement des subgraphs Sigma-C en analysant le code source d'applications FREIA ;
3. nous avons mis en place un système de contrôle d'exécution pour gérer différents subgraphs simultanément et transférer automatiquement des images situées sur la machine hôte à la puce MPPA ;
4. nous avons enfin réalisé diverses optimisations, présentées plus bas, afin de tirer le maximum de performances du MPPA.

6.3.1 Opérateurs de traitement d'images en Sigma-C

Nous avons, en guise d'apprentissage du langage, implémenté les opérateurs de traitement d'images de FREIA en tant qu'agents Sigma-C. Nous avons testé leur conformité grâce à des subgraphs de test qui reproduisaient les applications FREIA.

Ces opérateurs, comme spécifié en section 3.2, sont classés en plusieurs catégories :

— les opérateurs arithmétiques réalisent des opérations élémentaires sur les pixels d'une ou plusieurs images ;
— les opérateurs de réduction extraient une unique valeur d'une image entière ;
— les opérateurs morphologiques sont des opérateurs à voisinage (pour opérer sur un pixel, ils doivent accéder aux valeurs des pixels voisins) ;
— enfin, des opérateurs complexes peuvent se décomposer en opérateurs des trois catégories précédentes.

Deux caractéristiques clefs de nos agents ont du être déterminées à la conception : le type de données pixels, ainsi que la granularité de nos agents.

Type de données pixels La plupart des images encodant leurs pixels sur des entiers 8 bits non signés, nous avons choisi d'effectuer nos traitements sur des entiers 16 bits signés afin de gérer sans peine les débordements. Les pixels 8 bits des images d'entrée sont alors directement transposés en pixels 16 bits durant la phase de traitement et sont convertis à nouveau en pixels 8 bits avant d'écrire les images résultats.

Granularité des agents Le nombre de données échangées entre deux agents influe fortement les performances des applications Sigma-C. Effectuer des calculs pixel par pixel est coûteux, puisque cela revient à multiplier les transitions d'état d'agents. De même, opérer sur une image entière n'est pas compatible avec le modèle flot de données, puisque, dans ce cas, les calculs ne peuvent être pipelinés et une ou plusieurs images complètes doivent cohabiter dans la mémoire d'un agent, ce qui augmente d'autant la pression mémorielle.

Une granularité intermédiaire est celle de la tuile de taille fixe, mais ce type de granularité rend complexe les implémentations des opérateurs au voisinage, notamment au niveau des bords des tuiles. De plus, les agents Split et Join de la bibliothèque standard Sigma-C ne gèrent pas les découpages avec recouvrement. Il est donc nécessaire d'implémenter des agents Split et Join spécifiques, qui ne seront pas optimisés par le compilateur Sigma-C, et qui vont demander plus de ressources de calcul. Le découpage en tuiles dépend également des dimensions d'une image afin de calculer les sauts d'adresses mémoires aux bords des tuiles. Dans ce cadre, le recouvrement imposé par les opérateurs morphologiques est complexe à gérer : chaque opération au voisinage va invalider les pixels situés sur le bord des tuiles, ce qui oblige soit à recoller les tuiles après chaque opérateur de ce type, soit à prévoir autant de recouvrement que d'appels à ces opérateurs à la suite. C'est pourquoi nous avons décidé d'utiliser une granularité ligne-à-ligne, qui implique certes que nos applications dépendront des dimensions des images d'entrée à la compilation, mais qui permet une implémentation simple des opérateurs de traitement d'images.

Dans le modèle flot de données, la performance d'une application est celle de l'agent le plus lent, qui va imposer son rythme au flot. L'exécution d'un agent consiste en plusieurs cycles composés d'une phase de calculs, qui dépend de la taille des données, et d'une phase de transition entre états, de coût fixe.

Nous avons représenté en figure 6.5 le temps d'exécution de plusieurs applications Sigma-C selon le nombre de pixels traités durant un état d'agent. Le temps de traitement par pixel diminue ainsi, quand le nombre de pixels augmente : les agents amortissent ainsi le temps passé dans les transitions. Cependant, opérer sur des quantités trop importantes de pixels

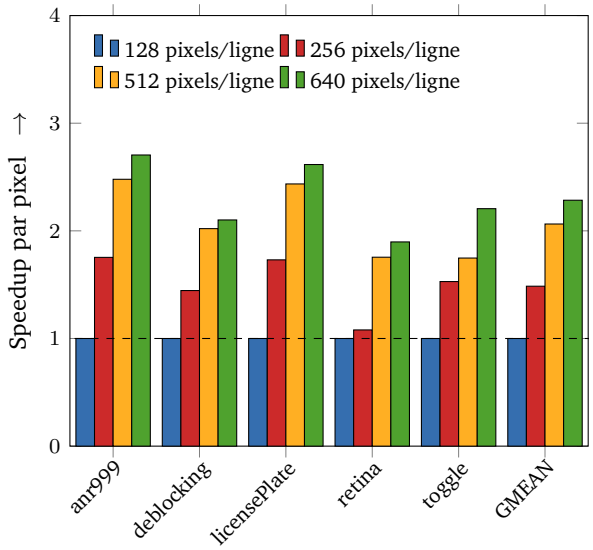


Figure 6.5 – Speedups par pixel de cinq applications selon le nombre de pixels traités par cycle d'agent : (1) lignes de 128 pixels (référence) ; (2) lignes de 256 pixels ; (3) lignes de 512 pixels ; (4) lignes de 640 pixels

en une seule fois peut augmenter la pression sur la mémoire partagée et ainsi occuper plus de clusters sur le MPPA. Opérer sur des lignes de calculs revient à dépendre des images d'entrée, mais permet d'éviter des complications dans l'implémentation des opérateurs au voisinage.

Opérateurs arithmétiques

Les opérateurs arithmétiques regroupent les opérations élémentaires qui transforment un pixel en un autre. Ces opérateurs sont généralement binaires, avec deux images en entrée. Par exemple, on peut sommer pixel à pixel deux images pour en obtenir une troisième. Une version unaire (avec une seule image en entrée) de la somme existe également : chaque pixel est sommé à une constante fixée, ce qui réduit l'empreinte mémoire et augmente la performance par rapport au cas d'une somme avec image constante.

La liste d'opérateurs dans cette catégorie est la suivante :

— somme, différence, multiplication, division ;
— somme et différence saturées ;
— maximum, minimum, test d'égalité ;
— opérateurs binaires (et, ou, non, non-et, ou exclusif) ;
— différence absolue, logarithme, seuillage, masquage.

Leur structure étant, pour la majorité d'entre eux, très proche, nous nous sommes appuyés sur le préprocesseur C pour les implémenter plus rapidement et ainsi éviter une duplication du code source, ce qui nuit à la maintenance. Par exemple, en extrait 6.4, se trouve la macro-définition utilisée pour générer les agents arithmétiques binaires. Les appels à cette macro peuvent se consulter en extrait 6.5. Les opérateurs unaires correspondants ont été définis de façon similaire.

Certains opérateurs, comme le seuillage, le masquage, le logarithme ou le non binaire, ne suivent pas ce schéma et ont du être définis à part.

```
#define AGENT_BINARY_OP(OPERATION, TYPE, EXPRESSION)                          \
  agent OPERATION(int width) {                                                \
    interface {                                                               \
      in<TYPE> input[2];                                                      \
      out<TYPE> output;                                                       \
      spec { input[0][width]; input[1][width]; output[width] };               \
    }                                                                         \
                                                                              \
    void start() exchange(input[0] inp1[width], input[1] inp2[width],         \
                          output outp[width]) {                               \
      int i;                                                                  \
      for (i = 0; i < width; i++)                                             \
        outp[i] = EXPRESSION;                                                 \
    }                                                                         \
  }
```

Extrait 6.4 – Macro-définition pour les agents arithmétiques binaires

```
AGENT_BINARY_OP(img_add_img_16, int16_t, inp1[i] + inp2[i])
AGENT_BINARY_OP(img_sub_img_16, int16_t, inp1[i] - inp2[i])
AGENT_BINARY_OP(img_mul_img_16, int16_t, inp1[i] * inp2[i])
AGENT_BINARY_OP(img_div_img_16, int16_t, inp1[i] / inp2[i])

AGENT_BINARY_OP(img_inf_img_16, int16_t,
                (inp1[i] < inp2[i] ? inp1[i] : inp2[i]))
AGENT_BINARY_OP(img_sup_img_16, int16_t,
                (inp1[i] > inp2[i] ? inp1[i] : inp2[i]))
AGENT_BINARY_OP(img_eq_img_16, int16_t,
                (inp1[i] == inp2[i] ? MAX : MIN))

AGENT_BINARY_OP(img_and_img_16, int16_t, inp1[i] & inp2[i])
AGENT_BINARY_OP(img_or_img_16, int16_t, inp1[i] | inp2[i])
AGENT_BINARY_OP(img_xor_img_16, int16_t, inp1[i] ^ inp2[i])
AGENT_BINARY_OP(img_nand_img_16, int16_t, ~(inp1[i] & inp2[i]))
AGENT_BINARY_OP(img_nor_img_16, int16_t, ~(inp1[i] | inp2[i]))

AGENT_BINARY_OP(img_absdiff_img_16, int16_t,
                (inp1[i] - inp2[i] > 0 ?
                 inp1[i] - inp2[i] : inp2[i] - inp1[i]))
AGENT_BINARY_OP(img_subsat_img_16, int16_t,
                (inp1[i] - inp2[i] <= MIN ? MIN : inp1[i] - inp2[i]))
AGENT_BINARY_OP(img_addsat_img_16, int16_t,
                (inp1[i] + inp2[i] >= MAX ? MAX : inp1[i] + inp2[i]))
```

Extrait 6.5 – Génération des définitions pour les agents arithmétiques binaires

Opérateurs de réduction

Les opérateurs de réductions réduisent une image entière à une simple valeur scalaire : le maximum de la valeur des pixels, le minimum ou bien la somme (aussi appelée *volume* d'une image). Des variantes des opérateurs de détermination d'*extrema* renvoient également des coordonnées d'un pixel qui les satisfait.

Leur implémentation en Sigma-C est plutôt directe : un agent reçoit et réduit les images ligne à ligne, le résultat étant stocké dans un accumulateur interne. Les agents doivent, cependant, avoir connaissance de la taille de l'image, afin de savoir à quel moment commu-

niquer le résultat et réinitialiser l'état de l'accumulateur.

Opérateurs morphologiques

Les opérateurs morphologiques fondent la base de la théorie de la morphologie mathématique. Leur utilisation permet notamment de discriminer des formes géométriques dans des images. Ce sont des opérateurs du voisinage : pour calculer la valeur d'un pixel de l'image sortante, les valeurs des pixels voisins du pixel d'entrée sont utilisées. Ceci rend ces opérateurs plutôt intenses en matière de calcul, en comparaison des autres classes d'opérateurs.

Afin de désigner les pixels voisins à utiliser, ces opérateurs font appel à un tableau booléen de taille 3×3 appelé *élément structurant*. Une réduction est ensuite opérée sur les voisins sélectionnées par cette matrice. Celle-ci varie selon l'opérateur :

l'érosion est le minimum des valeurs des pixels voisins sélectionnés ;

la dilatation en est le maximum ;

la convolution en est la moyenne arithmétique.

Puisque, pour chaque pixel, il est nécessaire de pouvoir accéder à ses voisins, il n'est plus possible de consommer et de produire une seule et même ligne de pixels en une seule fois. Nous devons avoir une fenêtre glissante de trois lignes qui avance une ligne à la fois.

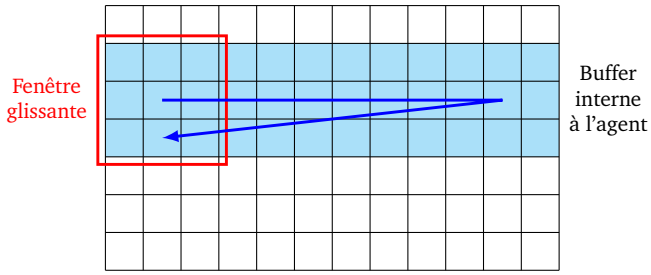


Figure 6.6 – Implémentation Sigma-C des opérateurs morphologiques : fenêtre glissante sur un buffer interne de trois lignes

De nombreux essais et optimisations ont permis d'obtenir la version actuelle des agents morphologiques. Nous pouvons citer notamment :

```
1  spec {
2    {input[width]} ;
3    (height - 1) {input[width]; output[width]} ;
4    {output[width]}
5  } ;
```

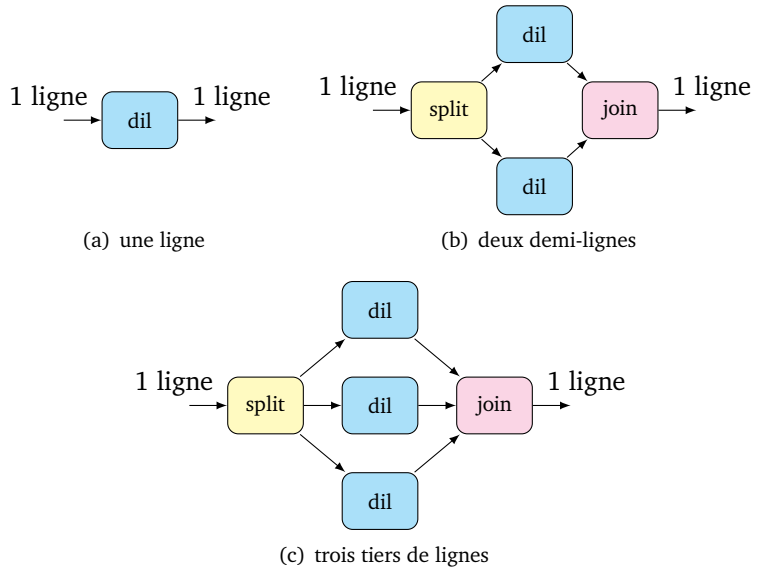
Extrait 6.6 – Spécification des agents morphologiques

— l'utilisation du cache pour les canaux d'entrée, tentative qui n'a pas abouti car les cas aux bords étaient trop complexes à gérer, et un bug du compilateur (corrigé depuis) sur-approximait la taille des données transmises (au final, nous nous sommes contentés d'une spécification simple de ligne à retard, représentée en extrait 6.6, où, à chaque ligne d'image reçue, c'est la précédente qui est produite) ;

— un buffer interne à l'agent pour stocker les trois lignes consécutives nécessaires au calcul, le buffer est de dimension 2 pour faciliter l'écriture des accès et légèrement plus grand que la taille d'une ligne, afin de gérer automatiquement les bords ;
— l'augmentation de la taille de la pile interne à l'agent afin de supporter ce buffer de trois lignes ;
— le remplissage du buffer interne par écrasement de la plus vieille ligne qui s'y trouve, ceci afin d'éviter des copies inutiles ;
— un cœur de boucle en assembleur, écrit et optimisé par les ingénieurs de Kalray tirant parti de l'architecture VLIW des cœurs de calcul, que nous comparons à l'implémentation naïve en C plus loin en figure 6.12.

Une tentative d'optimisation a consisté à paralléliser ces opérateurs. Ces opérateurs étant très calculatoires, nous avons cherché à les faire agir sur des portions de lignes plutôt que des lignes complètes, afin de pouvoir plus facilement répartir la charge de calcul sur les cœurs du MPPA. Des agents permettant de répartir les flux de données en entrée et de les agréger en sortie doivent être utilisés pour réaliser cette transformation. Une représentation schématique de celle-ci est visible en figure 6.7.

Puisque les opérateurs morphologiques sont des opérateurs au voisinage, il va falloir gérer un recouvrement lors de la division des lignes en entrée, ainsi qu'une élimination des données incorrectes pendant la phase de jointure des résultats. Pour ce faire, nous avons développé nos propres agents Sigma-C de division (*split*) et de jointure (*join*) de ligne. Nous nous sommes, dans un premier temps, restreints aux cas de division en 2 et 3 morceaux de lignes, cas représentés en figure 6.7.



(a) une ligne (b) deux demi-lignes

(c) trois tiers de lignes

FIGURE 6.7 – Cas traités pour la parallélisation des opérateurs morphologiques

Les résultats, disponibles en figure 6.8, montrent cependant une augmentation du temps d'exécution, soit une diminution de la performance globale de l'application, lorsque les calculs s'effectuent sur des portions de lignes. Le surcoût des communications ainsi qu'en occupation mémoire est prépondérant face aux bénéfices apportés par la parallélisation. L'implémentation actuelle des opérateurs morphologiques correspond au final à un seul agent par opérateur.

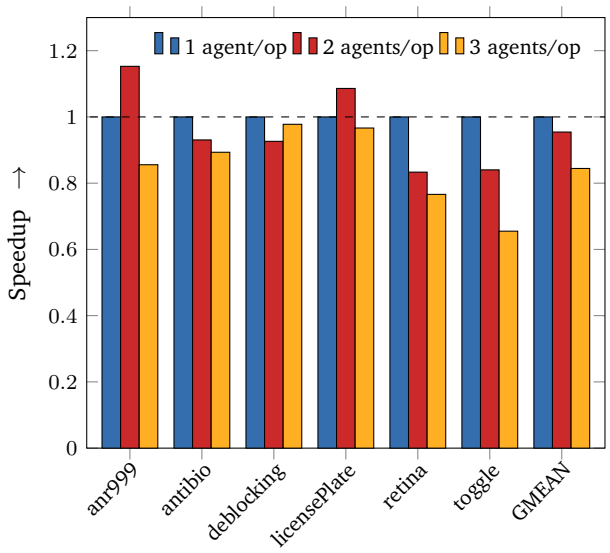


Figure 6.8 – Speedups de six applications pour les cas présentés en figure 6.7 : (1) un seul agent par opérateur morphologique (référence) ; (2) deux agents par opérateur morphologique (+ split/join) ; (3) trois agents par opérateur morphologique (+ split/join)

Opérateurs composés

La plupart des opérateurs de traitement d'images utilisés dans FREIA sont des combinaisons d'opérateurs des trois catégories précédentes. On peut notamment citer le *gradient morphologique*, qui soustrait le résultat d'une série d'érosions au résultat d'une série de dilatations.

```
1  void
2  freia_cipo_geodesic_reconstruct(freia_data2d *immarker,
3                                  freia_data2d *immask)
4  {
5    int32_t volcurrent, volprevious;
6    freia_global_vol(immarker, &volcurrent);
7
8    do {
9      volprevious = volcurrent;
10
11     freia_dilate(immarker, immarker);
12     freia_inf(immarker, immarker, immask);
13
14     freia_global_vol(immarker, &volcurrent);
15   } while(volcurrent != volprevious);
16
17 }
```

Extrait 6.7 – Extrait du code FREIA pour l'opérateur de reconstruction géodésique

La *reconstruction géodésique* itère sur une série de transformations, en comparant les volumes des images issues de deux itérations consécutives. La boucle termine lorsque les volumes sont égaux. Un extrait du code de cet opérateur peut se consulter extrait 6.7. La

suite de transformations effectuée à chaque itération — dilatation puis minimum — est convergente par nature : la boucle n'est pas infinie. Nous reviendrons plus loin sur les problèmes posés par cet opérateur et les solutions que nous y avons apportées.

De par leur nature, ces opérateurs composés s'implémentent en utilisant des subgraphs Sigma-C autour des opérateurs de base. L'utilisation astucieuse d'un compilateur source-à-source permet néanmoins d'éviter cette phase d'implémentation : c'est le compilateur qui les décomposera en opérateurs de base pour générer lui-même les subgraphs Sigma-C.

6.3.2 Génération source-à-source de *subgraphs* applicatifs

Après qu'un nombre suffisant d'opérateurs de traitement d'images ont été réimplémentés en agents Sigma-C, nous nous sommes concentrés sur la génération automatique par le compilateur source-à-source PIPS de subgraphs applicatifs par analyse de codes FREIA.

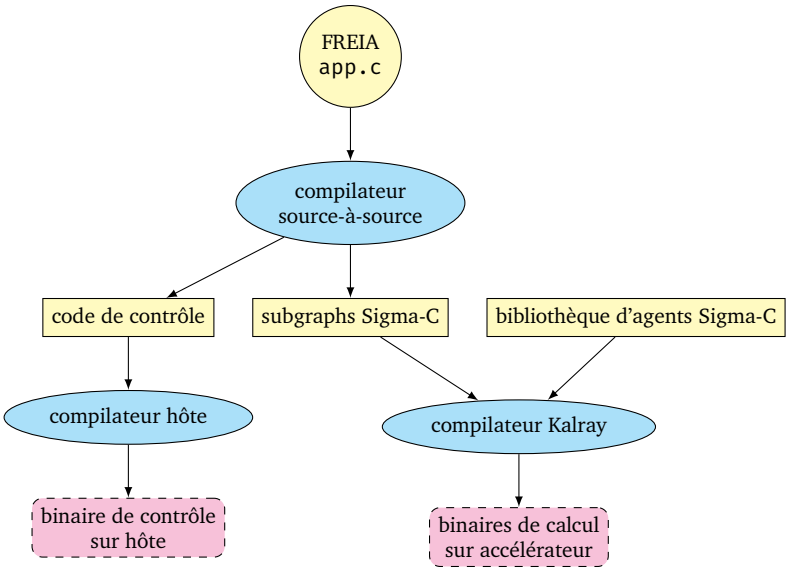


FIGURE 6.9 – Chaîne de compilation Sigma-C : une application FREIA est transformée en subgraph Sigma-C faisant appel à notre bibliothèque d'agents, dirigée par un code de contrôle sur l'hôte

Pour ce faire, nous avons repris les travaux précédemment réalisés dans le cadre du projet FREIA. Une fois cette génération fonctionnelle, nous avons testé l'impact de certains paramètres sur les temps d'exécution de sept applications FREIA.

Procédure générale

Cette génération automatique consiste, par exemple à partir des trois lignes de code présentées en extrait 6.8, à générer le code Sigma-C en extrait 6.9. Pour ce faire, il faut donc instancier les agents correspondant aux opérations appliquées, puis connecter les bons ports de sortie aux bons ports d'entrée.

De précédents travaux [29] ont permis au compilateur source-à-source PIPS [41, 75] d'analyser des applications FREIA pour en générer une représentation intermédiaire sous la forme d'une liste de graphes orientés acycliques ou DAGs (Directed Acyclic Graphs). Les sommets de ces DAGs sont ici les opérateurs élémentaires de traitement d'images de FREIA,

```
freia_aipo_erode_8c(im1, im0, kernel);   // morphological
freia_aipo_dilate_8c(im2, im1, kernel);  // morphological
freia_aipo_and(im3, im2, im0);           // arithmetic
```

Extrait 6.8 – Trois lignes de code FREIA

```
subgraph foo() {
  int16_t kernel[9] = {0,1,0, 0,1,0, 0,1,0};
  ...
  agent ero = new img_erode(kernel);
  agent dil = new img_dilate(kernel);
  agent and = new img_and_img();
  ...
  connect(ero.output, dil.input);
  connect(dil.output, and.input);
  ...
}
```

Extrait 6.9 – Génération automatique de code Sigma-C à partir du précédent extrait 6.8

les opérateurs composés ayant été inlinés. Les arcs représentent alors les données, à savoir des images ou des scalaires. Ces DAGs servent de base à la génération automatique de code visant les accélérateurs sur FPGA SPoC [27] et Terapix [20]. Une génération de code OpenCL permet de cibler également des accélérateurs graphiques. Nous sommes partis de ces DAGs pour les transposer efficacement en subgraphs Sigma-C.

Afin de transposer correctement une application FREIA en subgraph Sigma-C, nous devons au préalable nous assurer de l'absence de dépendance scalaire entre deux agents d'un même subgraph, c'est-à-dire qu'un agent ne consomme pas le produit d'une réduction effectuée dans le même subgraph pour l'appliquer à l'image. En effet, puisque les images sont traitées ligne par ligne, un scalaire résultant d'un opérateur de réduction ne peut être directement appliqué sur les lignes de la même image sans accumuler plusieurs lignes dans des canaux de communication inter-agents, ce qui ruine le parallélisme de pipeline et entraîne une chute de performances. Pour obtenir des subgraphs indépendants, une passe de séparation des DAGs sur les dépendances scalaires est appliquée en amont de notre générateur de code.

Nous encodons ensuite directement ces DAGs en subgraphs Sigma-C : chaque sommet mène à l'instanciation d'un agent, puis l'analyse des arcs permet de connecter le présent agent à ses prédécesseurs ou successeurs. Des différences minimes entre la représentation sous forme de DAG et la syntaxe Sigma-C ont dû être surmontées :

— le nombre de ports d'entrée et de sortie des agents Sigma-C étant fixé, il a fallu insérer des agents de réplication des données lorsque nécessaire ;
— les entrées et sorties d'un DAG doivent être traitées séparément ;
— les dépendances scalaires doivent être fournies aux agents les utilisant par un subgraph dédié, tout comme les résultats des réductions.

L'algorithme 6.1 représente ainsi de façon simplifiée — nous n'avons ici pas séparé la gestion des entrées/sorties du graphe — notre générateur de subgraphs Sigma-C à partir des DAGs fournis par le compilateur PIPS.

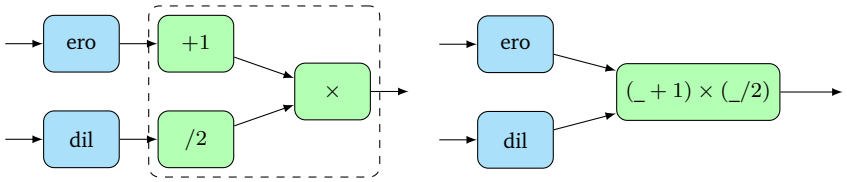
Algorithme 6.1 : Génération automatique de subgraphs Sigma-C

Entrées : L — une liste de nodes dans un DAG d'opérations image
Sorties : R — une liste d'instructions codant un subgraph Sigma-C

```
1  R ← () ; // empty instruction list
   /* generate subgraph declaration with run-time agents for
      transfering images and reduction results (using a Join
      agent)                                                    */
2  R.append(gen_subgraph_template(L)) ;
3  nb_red ← 0 ; // number of reduction results
   /* loop over every node, generate agent instance and
      connections                                               */
4  foreach v ∈ L do
       /* every node get an unique string id                   */
5      v.id ← get_node_id(v, L) ;
6      R.append("agent %s = new %s(%s)", v.id, v.op, v.args) ;
7      if is_reduction_op(v) then
           /* redirect output to Join agent                    */
8          R.append("connect(%s.output, jo.input[%d])", v.id, nb_red) ;
9          nb_red ← nb_red + 1 ;
10     if #Succs(v) > 1 then
           /* add an intermediary Dup agent                    */
11         R.append("agent dup%s = new Dup(%d)", v.id, #Succs(v)) ;
12         R.append("connect(%s.output, dup%s.input)", v.id, v.id) ;
13         i ← 0 ;
14         foreach succ ∈ Succs(v) do
15             R.append("connect(dup%s.output[%d], %s.input[%d])", v.id, i, succ.id,
                succ.get_pred_id(v)) ;
16             i ← i + 1 ;
17     else
18         foreach succ ∈ Succs(v) do
19             R.append("connect(%s.output, %s.input[%d])", v.id, succ.id,
                succ.get_pred_id(v)) ;
20 return R ;
```

Agrégation d'opérateurs arithmétiques

Dans le modèle de programmation flot de données, c'est la tâche la plus lente dans une chaîne de traitements qui a la plus grande influence sur l'exécution. Puisque les opérateurs arithmétiques réalisent peu d'opérations en comparaisons des opérateurs morphologiques, nous nous sommes intéressés à factoriser des agents arithmétiques connexes en des agents composés, ainsi que schématisé figure 6.10. Ceci diminue le nombre global d'agents par application et permet ainsi d'occuper moins de cœurs. Cependant, dans le cas où trop d'opérateurs sont agrégés, une diminution des performances est à craindre, puisque les agents agrégés peuvent devenir un nouveau goulet d'étranglement. Nous avons implémenté cette passe au-dessus de notre générateur de code Sigma-C.



(a) 3 agents arithmétiques connexes (b) 1 agent agrégé réalisant 3 opérations

Figure 6.10 – Agrégation d'agents arithmétiques

Algorithme 6.2 : Détection de composantes non triviales d'agents Sigma-C

Entrées : L — une liste de nodes dans un DAG d'opérations image
Sorties : R — un ensemble d'agents Sigma-C arithmétiques agrégés
Données : `max_thr` — poids des opérateurs dans les agents agrégés, ≥ 2

```
1  S_cc ← get_connected_components(L); // set of connected components
2  R ← ∅; // set of agent sets
   /* divide connected components into sub-components if too
      large                                                    */
3  foreach cc ∈ S_cc do
       /* number of nodes in component                        */
4      p ← # vtx_in_cc(cc);
       /* filter out trivial components                       */
5      if p ≥ 2 then
           /* number of sub-components of size ≤ max_thr      */
6          g ← ⌈p / max_thr⌉;
           /* balanced threshold for cutting current component */
7          thr ← ⌈p / g⌉;
           /* assign nodes to sub-components                  */
8          R ← R ∪ cut_in_sub_cc(cc, thr);
9  foreach aa ∈ R do
       /* generate agent declarations                         */
10     gen_aggregated_agent(aa));
11 return R;
```

L'algorithme 6.2 présente ainsi une version raccourcie de notre algorithme d'agrégation d'opérateurs arithmétiques. Nous déterminons, d'abord, les composantes connexes d'opérateurs arithmétiques, que nous subdivisons selon un seuil. Il faut ensuite générer la définition de l'agent agrégé, l'instancier dans le bon subgraph et le connecter à ses prédécesseurs et successeurs dans le graphe de l'application.

Nous avons testé cette optimisation sur plusieurs applications, et les résultats de la figure 6.11 montrent, comme prévu, que la fusion des agents arithmétiques a peu d'influence sur l'exécution globale, tout en libérant néanmoins des ressources de calcul.

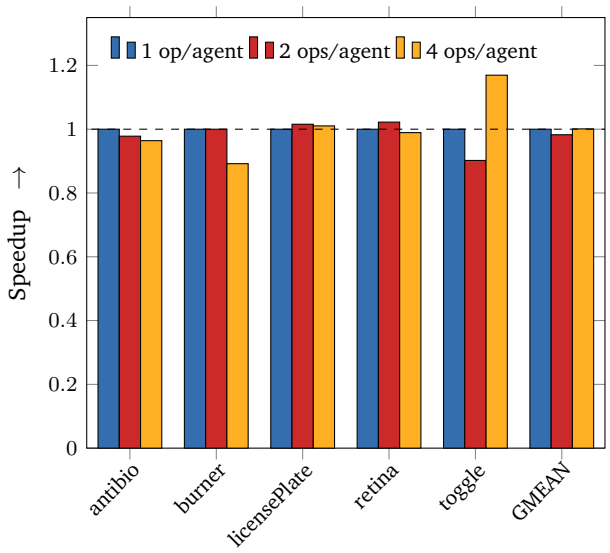


FIGURE 6.11 – Speedups sur cinq applications pour un nombre différent d'agents arithmétiques agrégés : (1) un opérateur par agent arithmétique (référence) ; (2) deux opérateurs arithmétiques agrégés ; (3) quatre opérateurs arithmétiques agrégés

Évaluation partielle pour les cœurs de boucle morphologiques

Les opérateurs morphologiques, étant les plus calculatoires, ont profité d'un cœur de boucle en assembleur spécialement développé par les ingénieurs de Kalray. Ces opérateurs dépendent d'un élément structurant, tableau booléen de taille 3×3 caractérisant le voisinage à prendre en compte dans le calcul. Cette structure étant généralement connue à la compilation, nous avons développé un générateur d'agents morphologiques Sigma-C grâce à son évaluation partielle : nous générons des agents morphologiques spécifiques à un élément structurant, qui, pour un pixel donné, accèdent directement aux voisins spécifiés sans condition sur l'élément structurant.

Nous avons comparé trois implémentations des agents morphologiques :

— une implémentation optimisée entièrement en Sigma-C ;
— l'implémentation avec le cœur de boucle en assembleur MPPA ;
— les agents morphologiques générés en Sigma-C et partiellement évalués.

Les résultats disponibles figure 6.12 montrent des gains allant jusqu'à 5 fois plus rapide que l'implémentation Sigma-C générique. L'évaluation partielle permet en outre d'obtenir des performances légèrement meilleures au cœur de boucle assembleur, et ce malgré la non-utilisation de l'assembleur.

6.3.3 Contrôle d'exécution

Le code Sigma-C généré par notre chaîne de compilation comporte généralement plusieurs subgraphs non connexes, qui sont tous placés sur les cœurs de calcul du MPPA. Afin de lancer le bon subgraph au bon moment et pour contrôler les entrées et sorties des applications, nous avons développé un environnement d'exécution en C. Ce dernier s'exécute sur la machine hôte, à côté de l'application Sigma-C générée, et communique via des *Unix named pipes*, ainsi que schématisé figure 6.13.

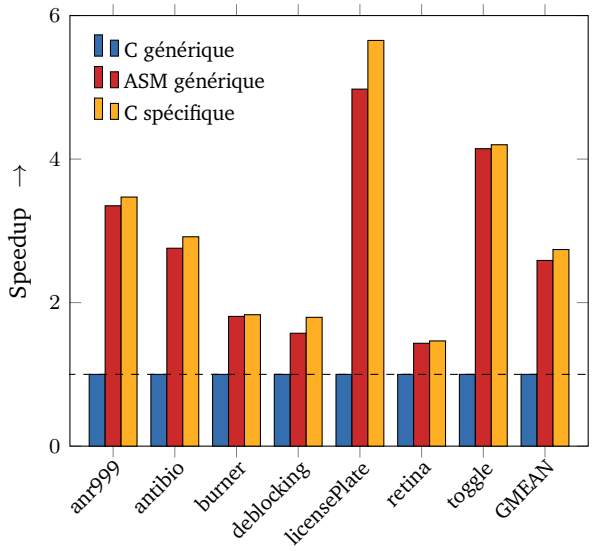


Figure 6.12 – Speedups pour différentes implémentations des cœurs de boucles des agents morphologiques : (1) implémentation C générique (référence) ; (2) implémentation générique en assembleur VLIW ; (3) implémentation C spécifique à l'élément structurant via évaluation partielle lors de la génération de code Sigma-C

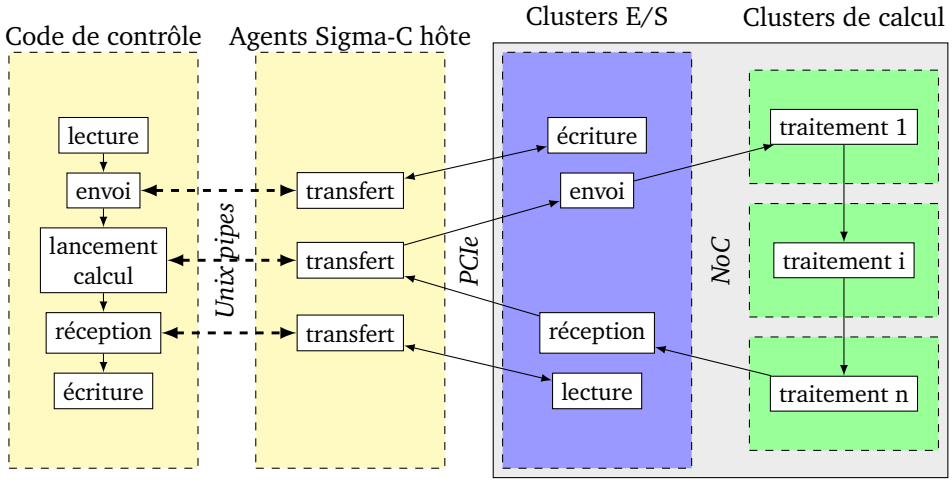


Figure 6.13 – Environnement d'exécution pour applications Sigma-C

Notre environnement d'exécution remplit les fonctions suivantes :
— allocation des images dans la mémoire DDR attachée au MPPA ;
— libération de la mémoire allouée dans la DDR attachée ;
— transfert d'une image depuis la mémoire de l'hôte vers la DDR attachée ;
— transfert d'une image depuis la DDR attachée vers la mémoire de l'hôte ;
— lancement d'un subgraph de calcul.
Pour chacune de ces fonctions, un subgraph dédié est exécuté sur l'hôte et un cluster

d'entrée/sortie, afin de relayer les commandes et les données à la puce. Notre environnement d'exécution gère également la lecture et l'écriture des différents formats d'images depuis le disque dur de la machine hôte. Il s'appuie, pour ce faire, sur une bibliothèque de traitement d'images purement logicielle appelée Fulguro [26], mais peut également utiliser SMIL à la place.

Optimisation des transferts de données

Afin d'améliorer les performances de notre environnement d'exécution, nous avons réalisé deux optimisations notables des transferts de données entre hôte et accélérateur.

Transferts directs depuis l'hôte La puce MPPA de 1re génération « Andey » souffre de temps d'accès élevés vers la mémoire DDR attachée aux clusters d'entrée/sortie. Nous avons donc altéré notre environnement d'exécution pour ne plus stocker les images dans la DDR, mais à la place les transférer directement depuis et vers l'hôte au début et à la fin de l'exécution de nos subgraphs de calcul. La figure 6.14 représente le speedup obtenu en transférant les images via PCI-Express plutôt que de les stocker temporairement dans la DDR. Les gains sont compris entre $\times 1.1$ et $\times 11$ pour une moyenne de $\times 2.5$. Ainsi, il est donc plus efficace de transférer directement les images depuis l'hôte que de les stocker dans la DDR attachée. La seconde génération de la puce MPPA, « Bostan », résout censément ce problème en diminuant les temps d'accès à la DDR attachée. Malheureusement, l'absence de support de Sigma-C sur cette nouvelle puce nous empêche de le confirmer.

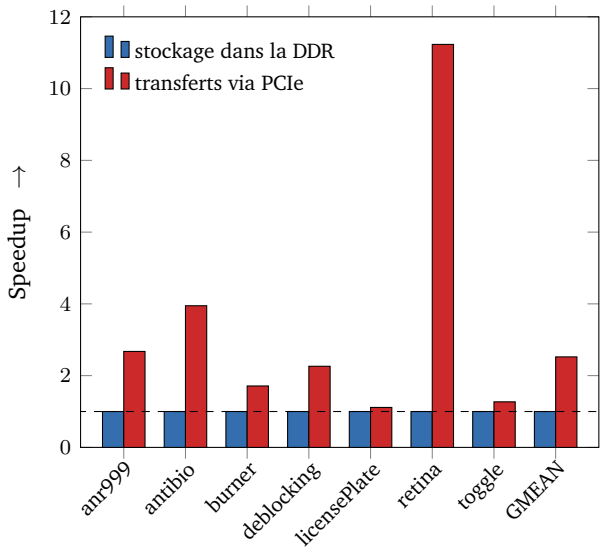


FIGURE 6.14 – Speedups pour le stockage temporaire des images : (1) utilisation de la mémoire globale attachée au processeur MPPA (référence) ; (2) transfert direct des images depuis l'hôte via l'interface PCI-Express

Déroulement de la reconstruction géodésique L'opérateur de reconstruction géodésique, déjà présenté en section 6.3.1, est souvent utilisé dans nos applications de traitement d'images. Il est caractérisé par une boucle `while`, illustrée en extrait 6.7, autour d'une transformation dont la suite est mathématiquement convergente. Comme la condition de

boucle dépend des données, il est impossible de déterminer le nombre d'itérations à la compilation, donc de dérouler totalement cette boucle. Le langage Sigma-C ne permet pas de gérer une telle construction. C'est notre environnement d'exécution qui s'en charge : chaque itération correspond à un subgraph, ce qui génère beaucoup de transferts de données depuis et vers l'hôte et diminue drastiquement les performances. Cependant, puisque la suite de transformations est convergente, réaliser des itérations supplémentaires n'a pas d'influence sur l'image résultante : il est donc possible de dérouler partiellement cette boucle sans en connaître le nombre d'itérations, ce qui diminue d'autant les transferts de données, tout en occupant plus de cœurs sur le MPPA. Nous avons ainsi réutilisé la passe de déroulage déjà implémentée pour le projet FREIA [29] dans notre compilateur source-à-source en amont de notre générateur de code Sigma-C. Nous représentons en figure 6.15 les speedups obtenus sur une collection d'applications utilisant l'opérateur de reconstruction géodésique, pour différents facteurs de déroulement. Dérouler d'un facteur 8 divise ainsi en moyenne les temps d'exécution d'un facteur $\times 3.5$. Au delà d'un facteur 8, le gain de performance n'est plus significatif : le nombre d'itérations supplémentaires tend à réduire le gain issu du déroulement, tandis que l'occupation d'un plus grand nombre de cœurs peut également avoir un impact négatif sur les performances.

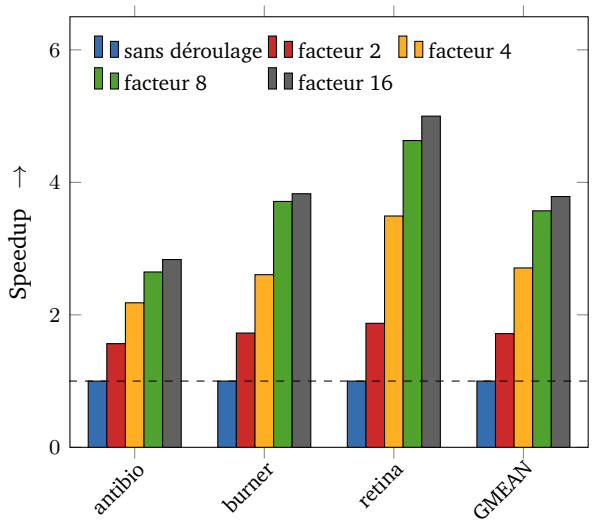


Figure 6.15 – Speedups pour différents facteurs de déroulage de la reconstruction géodésique : (1) pas de déroulage de boucle (référence) ; (2) déroulage d'un facteur 2 ; (3) déroulage d'un facteur 4 ; (4) déroulage d'un facteur 8 ; (5) déroulage d'un facteur 16

Exécution sur l'hôte

La chaîne de compilation Sigma-C de Kalray permet de cibler également l'architecture x86 de la machine hôte. Nous avons donc mesuré les temps d'exécution de nos applications de test sur le processeur hôte Intel Core i7-3820 disposant de huit cœurs logiques. Chaque agent Sigma-C étant compilé sous la forme d'un thread, le nombre limité de cœurs du processeur hôte ne permet pas d'obtenir des performances similaires à la puce MPPA, comme le montre la figure 6.16. Nos applications s'exécutent ainsi en moyenne trois fois plus rapidement sur l'accélérateur de Kalray que sur l'hôte. Les applications comportant de profonds pipelines d'opérateurs morphologiques tirent particulièrement parti du MPPA face au processeur Intel.

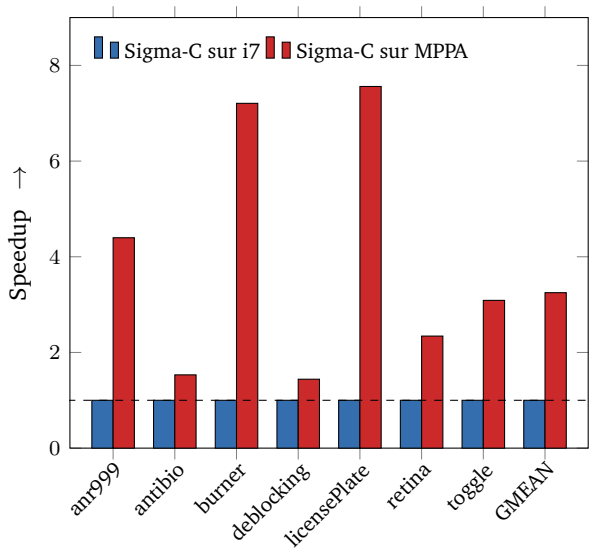


FIGURE 6.16 – Speedups de l'exécution d'applications Sigma-C sur différentes architectures : (1) exécution sur CPU hôte Intel Core i7-3820 (référence) ; (2) exécution sur MPPA

6.3.4 Limitations

En mettant bout-à-bout les trois pièces du puzzle — la bibliothèque d'agents de traitement d'images, le générateur de code Sigma-C et l'environnement d'exécution entre hôte et accélérateur —, il devient possible d'exécuter des applications FREIA directement sur la puce MPPA. Cependant, l'approche suivie souffre de plusieurs limitations, la plupart inhérentes à l'utilisation du langage Sigma-C.

En effet, Sigma-C a besoin de connaître les quantités de données produites et consommées entre deux agents à la compilation. Pour faciliter l'implémentation des opérateurs au voisinage, ainsi que pour diminuer le nombre de transitions entre états dans nos agents, nos agents échangent des lignes d'images. De plus, les agents morphologiques traitent séparément les première et dernière lignes d'une image. Ceci implique que les dimensions des images doivent être connues à la compilation des applications.

Le Sigma-C ne supporte pas les structures de contrôle dynamiques comme les boucles avec des conditions dépendantes des données. Il est également difficile d'activer différents subgraphs indépendants à la demande. Pour contourner ces limitations, nous avons introduit notre environnement d'exécution qui reproduit les structures de contrôle de l'application initiale non supportées par Sigma-C. Nous avons donc au final deux exécutables qui communiquent au travers de pipes Unix, ceci rajoutant une couche de complexité supplémentaire afin de pallier les carences de Sigma-C.

En outre, les agents Sigma-C, dont le nombre dépend de l'application compilée, sont placés à raison d'un ou deux par cœur de calcul sur les clusters du MPPA. Plus précisément, le nombre d'agents par cœurs est précisé au travers d'une option de compilation. L'occupation des différents cœurs dépend donc du nombre d'opérateurs de l'application exécutée. Dans l'idéal, une application doit donc comporter autour de 250 opérateurs, de sorte à occuper le maximum de cœurs disponibles. De plus, lorsqu'une application dispose de plusieurs subgraphs indépendants, ceux-ci sont placés en même temps sur le MPPA, ce qui réduit d'autant les ressources disponibles à un instant donné de l'exécution. Les applications les

plus performantes minimisent en effet le nombre de subgraphs utilisés.

Enfin, nous avons également été contraints de contourner les limitations du matériel, notamment au niveau de l'accès à la mémoire globale attachée aux clusters d'entrée/sortie. Nous espérons que la puce de 2e génération « Bostan » aura corrigé ce problème, bien que nous sommes dans l'incapacité de le déterminer.

6.4 Conclusion

Introduit dans les années 1970, le modèle flot de données facilite l'expression du parallélisme des applications destinées aux architectures multicœurs et manycore actuelles.

Le langage Sigma-C, développé pour tirer parti du processeur MPPA de Kalray, est un langage flot de données qui reprend la syntaxe du langage C. Sigma-C permet d'abstraire les communications entre l'hôte et l'accélérateur, ainsi qu'entre les différents nœuds de calcul qui composent le MPPA. Il permet ainsi de facilement tirer parti de l'architecture complexe du MPPA, au prix d'une réécriture complète des applications dans ce langage.

Nous avons créé un pont entre FREIA et Sigma-C, au travers d'une bibliothèque d'opérateurs de base de traitement d'images, d'un environnement d'exécution, qui gère les transferts entre hôte et accélérateur et les subgraphs Sigma-C, et d'un générateur automatique de code. Deux algorithmes issus de ce dernier sont ici détaillés. De la sorte, une application FREIA est automatiquement convertie en Sigma-C et peut s'exécuter sur le processeur MPPA. Nous augmentons ainsi le nombre de cibles logicielles et améliorons la portabilité des applications FREIA.

Cette nouvelle chaîne de compilation a permis de réaliser des expérimentations sur notre ensemble de test et, ainsi, de comparer le code Sigma-C exécuté sur MPPA ou sur un processeur conventionnel. Nous montrons que Sigma-C semble mieux exploiter le parallélisme des 256 cœurs du MPPA, particulièrement dans le cas des applications contenant des pipelines profonds de filtres morphologiques. Différentes optimisations nous ont permis d'améliorer les performances de ces applications ou de réduire leur empreinte mémoire. On peut notamment citer le déroulement des boucles convergentes, l'agrégation des agents arithmétiques ou bien l'évaluation partielle des cœurs de boucle morphologiques.

Chapitre 7

L'approche GPGPU : le framework OpenCL

How beautiful are your feet
In sandals, O prince's daughter
Your navel is a bowl
Well-rounded with no lack of wine
Your belly, a heap of wheat
Surrounded with lilies
Your breasts,
Clusters of grapes
Your breath,
Sweet-scented as apples
Nobody's gonna love you the way I love you.

— Song of Songs 7

L'introduction des accélérateurs graphiques dans les années 1990 a permis de décharger le processeur principal des calculs graphiques, simples et très répétitifs. Ces processeurs graphiques, aussi connus sous le nom de GPU (« *Graphics Processing Unit* »), ont conduit à des évolutions majeures dans le domaine de la 3D, au point que tous les ordinateurs modernes en contiennent.

La réserve de performance de ces processeurs spécialisés a longtemps intéressé les développeurs, qui souhaitaient y exécuter des calculs autres que graphiques. C'est ainsi que sont apparus les langages pour architectures hétérogènes permettant de déporter des calculs génériques sur accélérateurs graphiques, dont CUDA [37] et OpenCL [60] sont aujourd'hui les principaux représentants. Ce paradigme de programmation, appelé GPGPU pour « *General-Purpose Computing on Graphics Processing Units* », consiste à profiter des capacités des accélérateurs graphiques pour diminuer les coûts d'exécution de certains calculs.

Aujourd'hui, les processeurs graphiques sont concurrencés par l'émergence des processeurs manycore. Ces derniers disposent de moins d'unités de calcul — environ un ordre de grandeur en moins — qui compensent leur infériorité numérique par une complexité accrue et une meilleure efficacité énergétique globale. Le processeur MPPA de Kalray [43] supporte ainsi ce paradigme à travers la spécification OpenCL, de sorte à pouvoir exécuter directement des applications destinées à des accélérateurs matériels respectant la spécification. Le

projet FREIA [15] dispose, quant à lui, d'une cible logicielle OpenCL. Il a donc été possible, avec quelques modifications mineures, d'exécuter des applications FREIA sur le MPPA pour ainsi comparer l'efficacité de cet accélérateur face à des processeurs graphiques spécialisés.

Dans ce chapitre, nous décrivons dans les section 7.1 et section 7.2 les caractéristiques des processeurs graphiques, ainsi que quelques outils servant à les programmer. En section 7.3, nous examinons ensuite les détails de l'implémentation OpenCL de FREIA. Enfin, dans la section 7.4, nous listons les modifications nécessaires afin de l'adapter au processeur MPPA de Kalray, puis nous comparons ce dernier à plusieurs autres accélérateurs OpenCL, à code applicatif identique, pour ainsi tester les performances de notre accélérateur manycore.

7.1 Les processeurs graphiques, des puces spécialisées

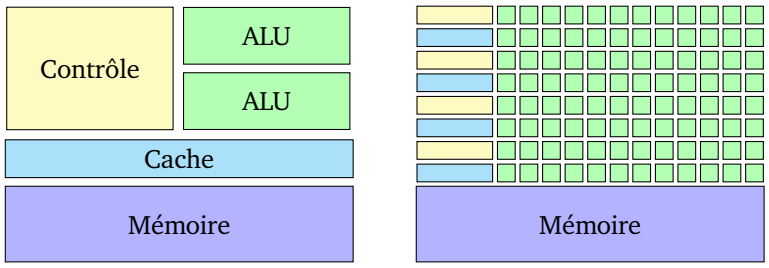
Démocratisées dans les années 1990, les cartes d'accélération graphiques améliorent significativement les calculs graphiques. Leur architecture diffère de celle des processeurs centraux : ils sont très adaptés aux calculs simples et répétitifs, par exemple des opérations entières sur les pixels d'une image 2D ou 3D. Avec le temps, ils ont acquis certaines capacités réservées aux processeurs centraux, comme les calculs sur nombres à virgule flottante double précision, ce qui en a fait des accélérateurs de choix pour les calculs scientifiques. Les processeurs graphiques ont notamment contribué à l'avènement des *architectures hétérogènes*, où le processeur principal déporte certains calculs spécifiques vers un ou plusieurs accélérateurs dédiés.

7.1.1 Un processeur central, un processeur graphique

Les processeurs centraux, également appelés CPU (*Central Processing Units*), sont caractérisés par leurs unités de calcul arithmétiques (ALU, pour *Arithmetic and Logic Unit*), en faible nombre mais capable d'effectuer des calculs complexes. Des unités de calcul vectorielles permettent également de réaliser un même calcul sur plusieurs données placées dans des registres de grande taille — allant désormais jusqu'à 512 bits [72], soit seize calculs flottants simple précision par cycle d'horloge. Les processeurs modernes consacrent également une partie de leur surface à optimiser le flot d'instructions des programmes exécutés. Ces unités de contrôle font, par exemple, de la prédiction de branchement ou de l'exécution d'instructions dans le désordre (« *out-of-order* ») et contribuent ainsi à augmenter le nombre d'instructions exécutées par cycle d'horloge du processeur.

Les processeurs graphiques sont, quant à eux, spécialisés dans les calculs arithmétiques. Ils disposent pour cela d'un grand nombre d'unités arithmétiques très efficaces pour effectuer des calculs entiers, moins pour des opérations à virgule flottante. Ces unités, ou *Processing Elements* (PE) dans la nomenclature OpenCL, sont regroupées en *Compute Units*, qui disposent d'une mémoire locale. De plus, une mémoire globale embarquée est accessible par tous les PEs, ce qui fait d'un processeur graphique une architecture à mémoire partagée à double niveau. Les PEs d'une *Compute Unit* exécutent tous la même suite d'instructions en parallèle ; l'architecture d'un processeur graphique suit ainsi le paradigme du parallélisme de données. La surface dédiée au contrôle du flot d'instructions étant bien plus faible que dans un processeur standard, les processeurs graphiques ne sont notamment pas optimisés pour exécuter des instructions conditionnelles complexes. Cependant, ils sont très efficaces dans les calculs graphiques, qui consistent à appliquer une suite d'opérations arithmétiques sur un ensemble de pixels.

La figure 7.1 illustre les différences d'architectures entre un processeur central et un processeur graphique, telles que décrites dans les paragraphes ci-dessus.



(a) Processeur central (b) Processeur graphique

FIGURE 7.1 – Comparaison des architectures d'un processeur central et d'un processeur graphique

7.1.2 L'évolution des GPUs : des cartes graphiques aux *System-on-Chip*

Les processeurs graphiques sont devenus courant par leur intégration dans les ordinateurs de bureau sous la forme de cartes d'extension appelées cartes graphiques. Depuis, de nouveaux usages ont vu le jour, notamment liés aux usages mobiles, avec les ordinateurs portables puis les *smartphones*, dont l'explosion du marché a entrainé une forte innovation dans le secteur des GPUs.

Les cartes graphiques dédiées ont ainsi subi la concurrence des puces graphiques intégrées aux processeurs centraux, moins consommatrices d'énergies mais moins performantes, à destination des ordinateurs portables. Cela a permis au constructeur AMD de concevoir une gamme de processeurs disposant de cœurs graphiques qui peuvent accéder directement à la mémoire centrale, unifiée entre le CPU et le GPU. L'amélioration des performances des cartes graphiques, notamment dans les calculs flottants double précision, leur architecture fortement parallèle ainsi que leur efficacité énergétique en ont également fait des acteurs de choix de l'informatique haute-performance pour les calculs scientifiques. Ainsi, les superordinateurs du TOP500 [117] comportent généralement des processeurs graphiques pour accélérer les calculs. Cependant, ils subissent désormais la concurrence des processeurs manycore : le superordinateur Tianhe-2, qui a occupé la première place du TOP500 de 2013 à 2016, dispose d'accélérateurs manycore Intel Xeon Phi [74].

Le développement des smartphones ces dernières années a mis également l'accent sur la consommation énergétique plutôt que sur la performance brute. C'est ainsi que des solutions basse consommation fondées sur des *System on Chip*, des agrégats d'accélérateurs spécifiques rassemblés sur une unique puce, ont vu le jour.

Tous ces usages rendent les processeurs graphiques malcommodes à programmer. Heureusement, des interfaces spécifiques ont vu le jour pour faciliter le travail des développeurs.

7.2 Modèles de programmation pour accélérateurs graphiques

Les processeurs graphiques, de nature très différents des processeurs centraux, ne se programment pas de la même façon. Dans le cas d'un processeur central, le système d'exploitation permet de lancer directement des exécutables binaires. Dans le cas d'un processeur graphique, c'est le processeur central qui envoie des commandes au processeur graphique par l'intermédiaire d'un programme appelé *driver* fourni par le concepteur du GPU ou par

rétro-ingénierie du matériel. Ces drivers implémentent différentes interfaces permettant d'exécuter des programmes sur GPU.

7.2.1 Au commencement, les interfaces de programmation graphiques

Les années 1990 ont vu fleurir deux interfaces de programmation destinées à normaliser les calculs graphiques sur GPU :

DirectX, de Microsoft [32], est une interface de programmation pour calculs graphiques réservée au système d'exploitation Microsoft Windows ;

OpenGL est une interface de programmation multiplateforme développée par un groupe d'industriels, le Khronos Group [61].

Ces deux interfaces exposent un *pipeline graphique*, une suite d'opérations graphiques de différents types servant à construire une scène 3D qui sera ensuite affichée sur un écran. Les différentes opérations sont alors fournies par l'interface, qui en propose un vaste catalogue. En tirant parti de cette chaîne d'opérations, il est possible, avec beaucoup d'efforts et par abus de l'interface, de faire exécuter et d'accélérer des calculs génériques.

Au début des années 2000, l'introduction des *shaders*, des fonctions graphiques programmables, a apporté une couche de programmabilité supplémentaire pour les développeurs voulant tirer parti des GPUs. Toutefois, il reste malcommode d'utiliser ces interfaces avant tout destinées aux calculs graphiques. Dans la seconde moitié des années 2000, des nouveaux modèles de programmation plus génériques ont vu le jour afin de combler ce manque : d'abord CUDA, puis OpenCL.

7.2.2 CUDA, le joyau propriétaire

CUDA [37] (« *Compute Unified Device Architecture* ») est le nom donné à un ensemble de technologies regroupées autour d'une interface de programmation destinée à faciliter la programmation d'applications plus généraliste sur processeurs graphiques. Cette interface de programmation a été développée par le fabricant de processeurs graphiques Nvidia. Initialement publié en 2007, CUDA est actuellement toujours supporté et activement développé, puisque l'outil est en version 7.5 au moment de l'écriture de ce manuscrit et qu'une version 8 est d'ores et déjà annoncée.

CUDA permet d'exploiter le parallélisme de données inhérent aux processeurs graphiques en exécutant, sur chaque unité de calcul, un même noyau de calcul. Cependant, les versions récentes de CUDA [33] supportent également l'exécution concurrente de différents noyaux de calculs, pavant la voie vers le parallélisme de tâches. De plus, durant l'exécution de ces noyaux de calcul, le processeur hôte reste capable d'effectuer des calculs complémentaires, de sorte à accélérer davantage l'exécution globale.

Déconstruction

CUDA comporte plusieurs briques jouant chacune un rôle spécifique :

plusieurs interfaces de programmation vers des langages de programmation génériques comme C ou Python (PyCUDA), mais également le langage Fortran, dédié aux calculs numériques ;

un modèle de programmation permettant de définir des noyaux de calculs (également appelés *kernels*) qui seront exécutés sur processeur graphique ;

un compilateur nommé `nvcc` qui génère à la fois le binaire exécuté par le processeur graphique et celui exécuté par l'hôte ;

un ensemble de bibliothèques permettant de déporter et d'accélérer des calculs d'algèbre linéaire (cuBLAS) aux réseaux de neurones pour l'apprentissage artificiel (cuDNN).

Le modèle de programmation sous-jacent consiste à déporter des calculs lourds mais peu complexes sur l'accélérateur graphique depuis le processeur central de l'hôte. Des *kernels*, ou noyaux de calcul, sont définis par le développeur pour être exécutés par chaque unité de calcul du processeur graphique. L'exécution d'une application CUDA suit généralement les étapes suivantes :

1. les données d'entrée, de préférence volumineuses, sont transférées depuis la mémoire vive du processeur central à la mémoire située sur l'accélérateur graphique ;
2. des *kernels* sont exécutés sur ces données par le processeur graphique ;
3. les données résultantes sont transférées depuis l'accélérateur graphique vers la mémoire du processeur central.

Exemple : addition de deux vecteurs

```
5  // CUDA kernel. Each thread takes care of one element of c
6  __global__ void vecAdd(double *a, double *b, double *c, int n) {
7    // Get our global thread ID
8    int id = blockIdx.x * blockDim.x + threadIdx.x;
9
10   // Make sure we do not go out of bounds
11   if (id < n)
12     c[id] = a[id] + b[id];
13 }
```

Extrait 7.1 – Noyau de calcul CUDA minimal

Un exemple d'application CUDA utilisant l'interface avec le langage C est présenté en extrait 7.1 et extrait 7.2. L'extrait 7.1 définit ainsi un kernel CUDA qui effectue une addition entre deux vecteurs de nombre flottants. Ce kernel, exécuté sur l'unité de calcul `int id` du processeur graphique, additionnera la composante `id` des vecteurs `double *a` et `double *b` et sauvegardera le résultat à la position `id` du tableau `double *c`. À l'exécution, ce kernel sera répliqué sur toutes les unités de calcul du processeur graphique considéré, additionnant de fait chaque composante des vecteurs `a` et `b`, dans la limite de leur taille donnée par l'entier `int n`.

En extrait 7.2 est présentée la portion congrue de notre application d'addition vectorielle servant à lancer le précédent kernel et illustrant le modèle de programmation présenté plus haut. Ainsi, la fonction

```
1    cudaError_t cudaMalloc(void **devPtr, size_t size);
```

alloue, ici, trois vecteurs de même taille dans la mémoire de l'accélérateur graphique. La fonction

```
1  cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
2                         enum cudaMemcpyKind kind);
```

```
54  // Allocate memory for each vector on GPU
55  cudaMalloc(&d_a, bytes);
56  cudaMalloc(&d_b, bytes);
57  cudaMalloc(&d_c, bytes);
58
59  // Copy host vectors to device
60  cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
61  cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
62
63  // Execute the kernel
64  vecAdd << <gridSize, blockSize>>> (d_a, d_b, d_c, n);
65
66  // Copy array back to host
67  cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);
```

Extrait 7.2 – Portion de code C pour exécuter le noyau CUDA extrait 7.1

sert, quant à elle, à copier `size_t count` données entre la mémoire de l'hôte et celle de l'accélérateur. Elle est, dans le cas présent, utilisée trois fois, pour transférer les tableaux `a` et `b` vers l'accélérateur et transférer le tableau résultat `c` vers l'hôte. Enfin, le kernel

```
1  __global__ void vecAdd(double *a, double *b, double *c, int n);
```

est appelé explicitement avec, comme arguments, les pointeurs vers les tableaux précédemment alloués sur l'accélérateur. Les arguments spéciaux `gridsize` et `blocksize` sont ici relatifs au nombre et à la géométrie des unités de calcul en jeu lors de l'exécution de ce kernel.

Une application CUDA typique mélange kernels exécutés sur l'accélérateur et code destiné à l'hôte ; le compilateur se charge ensuite de générer les binaires pour chaque cible. C'est ainsi qu'en pratique, les extrait 7.1 et extrait 7.2 sont issus du même fichier.

Un outil puissant mais limité

L'utilisation de CUDA facilite la programmation de calculs non spécifiques sur processeurs graphiques. En pratique, cet outil n'est compatible qu'avec les processeurs graphiques produits par le constructeur Nvidia. S'adressant à une gamme restreinte de cibles matérielles et développé par leur concepteur, CUDA est particulièrement performant sur ces plateformes et offre une programmabilité meilleure que celle de son rival, OpenCL, plus générique.

Cependant, malgré un écosystème foisonnant, CUDA reste limité par son support exclusif des processeurs graphiques de la marque Nvidia. La chaîne de compilation est, en effet, sous le contrôle de son créateur, les sources logicielles de celle-ci restant fermées, de sorte qu'aucun concurrent ne puisse profiter de cette interface. Ceci force les utilisateurs de CUDA à n'utiliser que du matériel Nvidia, au profit certes de meilleures performances une fois le code optimisé. Le standard OpenCL, disposant de sources ouvertes et du soutien d'un grand nombre d'acteurs industriels, s'érige comme le principal concurrent de CUDA, malgré sa programmabilité moindre.

7.2.3 OpenCL, l'alternative mal-aimée

OpenCL [60], abréviation de « *Open Compute Language* », est une spécification logicielle destinée à faciliter les calculs sur tout type d'accélérateurs et, notamment, des processeurs

centraux ou des processeurs graphiques. Il se démarque ainsi de CUDA, dont il reprend le modèle d'exécution, en étant moins limité quant aux cibles matérielles visées. OpenCL a fait son apparition en 2008 [59], soit un an après CUDA, et est depuis également activement soutenu et développé. La version courante d'OpenCL, dévoilée en avril 2016, est numérotée 2.2 [57].

Le projet OpenCL est géré par un consortium regroupant plusieurs industriels du monde des accélérateurs matériels. Ce consortium se nomme le Khronos Group et est, en outre, en charge d'un ensemble de technologies spécialisées ou non, comme les interfaces pour graphisme 3D OpenGL [61] et Vulkan [65]. Les spécifications développées par le Khronos Group sont libres d'accès et peuvent être implémentées sans contraintes par les fabricants désirant rendre leur matériel compatible.

OpenCL n'est pas limité qu'aux processeurs graphiques. Plusieurs implémentations pour FPGAs existent [3, 118]. En outre, OpenCL peut également cibler les processeurs multi-cœurs [73], en l'absence d'accélérateur graphique, ce qui permet à OpenCL de concurrencer le modèle de programmation OpenMP.

Architecture et modèle de programmation

OpenCL n'est, à proprement parler, pas un logiciel, mais une spécification : charge aux constructeurs souhaitant la respecter d'en fournir une implémentation viable. La spécification OpenCL peut se diviser en deux composantes :

deux langages de programmation appelés OpenCL C et OpenCL C++, respectivement sous-ensemble de C et de C++, afin d'écrire des noyaux de calcul ;

une interface de programmation pour les langages C et C++ servant à préciser le contexte d'exécution des noyaux de calcul.

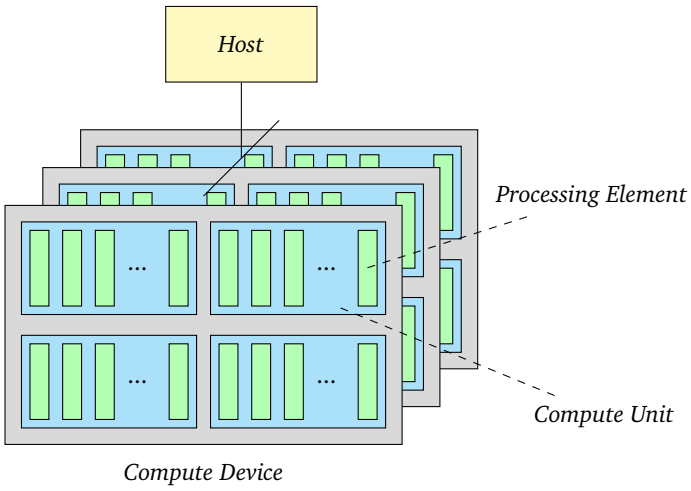


FIGURE 7.2 – Modèle d'architecture OpenCL

Le modèle de programmation d'OpenCL est similaire à celui de CUDA : un processeur hôte associé à une mémoire hôte déporte des calculs sur un ou plusieurs accélérateurs matériels, ainsi qu'illustré figure 7.2. Un accélérateur, aussi nommé *Compute Device* dans la terminologie OpenCL, se compose de plusieurs *Compute Units* (CU), elles-mêmes composées de *Processing Elements* (PE), ces derniers réalisant effectivement les calculs.

Les applications OpenCL ont une structure similaire à celle des applications CUDA : d'une part, des noyaux de calculs ou « *kernels* » exécutés par un accélérateur matériel et écrits en OpenCL C ou en OpenCL C++, d'autre part, un code application exécuté par l'hôte, faisant appel à l'interface de programmation C ou C++ et servant à allouer et libérer la mémoire sur l'accélérateur, transférer les données entre la mémoire de l'hôte et celle de l'accélérateur, ainsi que lancer les kernels sur les données ainsi transférées. Cependant, et au contraire de CUDA, les kernels OpenCL sont écrits dans un langage de programmation à part et doivent, par conséquent, être compilés séparément, par défaut au début de l'exécution du programme, par un compilateur spécifique à l'accélérateur cible, ceci assurant la portabilité du code sur différentes architectures. L'exécution d'une application OpenCL revient donc à effectuer la séquence d'actions suivantes :

1. compilation des noyaux de calcul vers l'accélérateur ;
2. transfert des données d'entrée depuis l'hôte vers l'accélérateur ;
3. lancement des calculs sur l'accélérateur ;
4. transfert des données résultat depuis l'accélérateur vers l'hôte.

Exemple : addition de deux vecteurs

```
1  __kernel void vecAdd(__global float *a,
2                       __global float *b,
3                       __global float *c,
4                       const unsigned int n) {
5
6    unsigned int id = get_global_id(0);
7
8    if (id < n)
9      c[id] = b[id] + a[id];
10 }
```

Extrait 7.3 – Noyau de calcul OpenCL minimal

Un exemple de kernel OpenCL C réalisant une addition vectorielle est représenté en extrait 7.3. Comme l'exemple présenté en sous-section 7.2.2, celui-ci consiste à additionner la composante `id` de deux vecteurs `a` et `b` et à stocker le résultat dans un troisième vecteur `c`. Ce kernel, instancié sur tous les PEs d'un accélérateur, additionnera toutes les composantes des deux vecteurs `a` et `b` dans la limite de leur taille `n`.

Une portion de code utilisant l'interface de programmation C pour OpenCL destinée à lancer le kernel précédent est représentée extrait 7.4. La mémoire sur l'accélérateur pour les vecteurs `a`, `b` et `c` est allouée en utilisant la fonction

```
1    cl_int clCreateBuffer();
```

puis les opérandes `a` et `b` sont envoyés vers l'accélérateur dans les buffers alloués à l'étape précédente. Le kernel est compilé grâce à la fonction

```
1    cl_int clBuildProgram();
```

et ses arguments sont passés à l'aide de la fonction

```
69  // Build the program executable
70  clBuildProgram(prgm, 0, NULL, NULL, NULL, NULL);
71
72  // Create the compute kernel in the program we wish to run
73  knl = clCreateKernel(prgm, "vecAdd", &err);
74
75  // Create the input and output arrays in device memory
76  d_a = clCreateBuffer(ctx, CL_MEM_READ_ONLY, bytes, NULL, NULL);
77  d_b = clCreateBuffer(ctx, CL_MEM_READ_ONLY, bytes, NULL, NULL);
78  d_c = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
79
80  // Write our data set into the input array in device memory
81  err = clEnqueueWriteBuffer(q, d_a, CL_TRUE, 0, bytes,
82                             h_a, 0, NULL, NULL);
83  err |= clEnqueueWriteBuffer(q, d_b, CL_TRUE, 0, bytes,
84                              h_b, 0, NULL, NULL);
85
86  // Set the arguments to our compute kernel
87  err |= clSetKernelArg(knl, 0, sizeof(cl_mem), &d_a);
88  err |= clSetKernelArg(knl, 1, sizeof(cl_mem), &d_b);
89  err |= clSetKernelArg(knl, 2, sizeof(cl_mem), &d_c);
90  err |= clSetKernelArg(knl, 3, sizeof(unsigned int), &n);
91
92  // Execute the kernel over the entire range of the data set
93  err = clEnqueueNDRangeKernel(q, knl, 1, NULL, &globalSize,
94                               &localSize, 0, NULL, NULL);
95
96  // Wait for the command queue
97  clFinish(q);
98
99  // Read the results from the device
100 clEnqueueReadBuffer(q, d_c, CL_TRUE, 0, bytes, h_c, 0, NULL, NULL);
```

EXTRAIT 7.4 – Code C minimal pour exécuter le kernel extrait 7.3

```
1  cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index,
2                        size_t arg_size, const void *arg_value);
```

appelée une fois pour chaque argument.

Enfin, la fonction

```
1  cl_int clEnqueueNDRangeKernel();
```

lance l'exécution du kernel sur l'accélérateur selon une topologie définie par les variables `globalSize` et `localSize`, avant le retour du vecteur résultat via l'appel à

```
1  cl_int clEnqueueReadBuffer();
```

En sus de ce mode de fonctionnement séquentiel, OpenCL permet l'exécution asynchrone des kernels au travers d'un système d'événements permettant de définir des dépendances entre les exécutions des kernels.

Comparaison avec CUDA

Déporter des calculs sur accélérateur graphique revient aujourd'hui à choisir entre CUDA et OpenCL. Le premier, réservé aux produits de la marque Nvidia, dispose d'une interface plus agréable pour les développeurs et de performances meilleures sur le matériel concerné. Quant à OpenCL, sa généricité rend son interface plus verbeuse : les portions de code présentées en extrait 7.2 et extrait 7.4 réalisent toutes deux le transfert de trois tableaux et l'exécution d'un kernel, mais la version OpenCL comporte plus de deux fois plus de lignes. L'utilisation de l'interface Python au travers du module PyOpenCL [81] peut, là, faciliter le travail des développeurs sans perdre en performance.

OpenCL étant un standard supporté par un nombre important de constructeurs, les applications l'utilisant sont portables : elles peuvent s'exécuter sans modifications sur tous les accélérateurs compatibles. Néanmoins, la spécification OpenCL autorise l'utilisation d'extensions propriétaires qui peuvent faciliter la programmabilité sur certaines plateformes aux dépends de la portabilité. En outre, puisque les cibles matérielles d'OpenCL sont parfois très différentes, il est difficile de conserver les performances en changeant de type d'accélérateur, obligeant parfois à réécrire certains kernels. Au contraire, les performances des applications CUDA sont relativement stables sur les GPUs de la marque Nvidia.

Enfin, CUDA dispose d'un vaste écosystème d'outils et de bibliothèques, telles cuBLAS [34], cuFFT [36] et cuDNN [35], qui favorisent le développement d'applications complexes. Les outils de développement CUDA simplifient ainsi le débogage d'applications, tâche d'autant plus complexe que les calculs sont déportés. OpenCL ne fournit en comparaison pas d'outils de ce type.

Évolutions futures

Le standard OpenCL, sous l'impulsion du Khronos Group, est en évolution constante. La prochaine version 2.2 voit ainsi l'arrivée du nouveau langage OpenCL-C++ [58] pour écrire les kernels, auparavant limités à OpenCL-C. Ce nouveau langage apporte des fonctionnalités de haut niveau issues du langage C++, comme les fonctions lambda, les templates ou la surcharge d'opérateurs.

Pour pallier le manque de performance entre les différentes plateformes cibles d'OpenCL, Khronos a développé un langage intermédiaire appelé SPIR-V [63] (pour « *Standard Portable Intermediate Representation* »). Celui-ci vise à devenir la représentation intermédiaire commune aux kernels OpenCL ainsi qu'aux shaders OpenGL et Vulkan. En outre, il profite des transformations et optimisations du compilateur LLVM. De cette façon, les pilotes des accélérateurs matériels n'auront, en entrée, qu'à supporter ce langage de bas niveau.

OpenCL évolue ainsi vers plus de simplicité pour les constructeurs voulant implémenter la spécification ou bien les développeurs voulant l'utiliser. Cependant, des interfaces de programmation de plus haut niveau, en chantier depuis quelques années, promettent de s'affranchir de la séparation entre code hôte et kernels déportés.

7.2.4 Vers des modèles haut niveau ?

Les principales critiques à l'encontre des interfaces de programmation CUDA et OpenCL se concentrent sur leur gestion explicite de la mémoire et des transferts de données. Le contrôle laissé au développeur par ces interfaces permet d'optimiser les performances aux dépends de la programmabilité. De plus, même si l'utilisation de ces interfaces assure la portabilité des applications au travers du respect des spécifications, les performances dépendent généralement du matériel. Afin de remédier à ces problèmes, des modèles de programmation de haut niveau ont été développés. Ceux-ci permettent, notamment, d'abstraire la gestion

de la localisation des données et fusionnent code hôte et kernels pour accélérer le développement. Charge au compilateur ou à l'environnement d'exécution de générer des kernels séparés et de gérer la mémoire de l'accélérateur et les transferts de données. La parallélisation automatique de code séquentiel peut également cibler les processeur graphiques [4].

Parmi ces nouveaux modèles de programmation, deux tendances s'affrontent :

— les modèles fondés sur des bibliothèques et utilisés via des interfaces de programmation ;
— les modèles fondés sur des directives préprocesseurs.

Des bibliothèques favorisant le déport des calculs

Plusieurs bibliothèques facilitent l'utilisation des processeurs graphiques. Parmi elles, citons :

C++ AMP, par Microsoft [31] mais limitée au système d'exploitation Microsoft Windows ;

Thrust, de Nvidia [39], une surcouche de CUDA ;

SYCL, développé par le Khronos Group [64] et compatible avec tous les accélérateurs OpenCL.

Ces bibliothèques permettent de fusionner code accéléré et code hôte, au prix de la dépendance à un constructeur, pour Thrust, ou à un système d'exploitation, pour C++ AMP. Quant à SYCL, il s'agit, comme OpenCL, d'une spécification proposée par le Khronos Group ; charge aux constructeurs d'en réaliser une implémentation. L'extrait 7.5 présente ainsi une addition vectorielle qui utilise SYCL. Le kernel est défini par la fonction lamdba passée en paramètres à `cl::sycl::kernel_functor()`, et l'exécution de celui-ci est réalisée par la fonction `cl::sycl::parallel_for()`.

Des directives préprocesseur pour adapter le code *legacy*

Des solutions fondées sur des directives préprocesseur ont récemment vues le jour, permettant de déporter ou non des blocs de code sans modification extensive de la source.

OpenACC [94], qui date de 2011, a été poussé par Nvidia, afin de faciliter le développement d'applications accélérées. De même que CUDA et Thrust, OpenACC ne fonctionne actuellement que sur un accélérateur Nvidia. Cependant, les avancées d'OpenACC ont en partie été fusionnées dans les versions 4.0 et 4.5 d'OpenMP [19], publiées respectivement en 2013 et 2015. OpenMP devient ainsi une solution d'accélération hétérogène qui couvre à la fois les processeurs multicœurs et les accélérateurs graphiques. OpenACC et OpenMP 4 abstraient tous deux les transferts de données entre hôte et accélérateur. OpenACC offre néanmoins un contrôle plus fin sur les données déportées et les calculs. Notre exemple d'addition entre deux vecteurs se réduit en les quelques lignes présentées dans les extrait 7.6 et extrait 7.7, identiques, à l'exception de la directive en tête de boucle.

Il devient alors aisé d'adapter du code *legacy* aux nouvelles architectures hétérogènes en utilisant de simples directives. Malheureusement, ces solutions à directives préprocesseur ont un support qui reste encore balbutiant dans les compilateurs standards. Ainsi, la dernière version de GCC supporte les deux spécifications, mais uniquement vers un nombre restreint de cibles matérielle : les GPUs Nvidia via OpenACC, certains co-processeurs Intel Xeon Phi avec OpenMP, et les derniers GPUs du constructeur AMD.

```
#include <CL/sycl.hpp>

#define LENGTH (1024) // Length of vectors a, b and c

int main() {
  std::vector h_a(LENGTH); // a vector
  std::vector h_b(LENGTH); // b vector
  std::vector h_c(LENGTH); // c vector (result)
  // Fill vectors a and b with random float values
  for (int i = 0; i < LENGTH; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
  }
  {
    // Device buffers
    cl::sycl::buffer d_a(h_a);
    cl::sycl::buffer d_b(h_b);
    cl::sycl::buffer d_c(h_c);
    cl::sycl::queue myQueue;
    cl::sycl::command_group(myQueue, [&]() {
      // Data accessors
      auto a = d_a.get_access<cl::sycl::access::read>();
      auto b = d_b.get_access<cl::sycl::access::read>();
      auto c = d_r.get_access<cl::sycl::access::write>();
      // Kernel
      cl::sycl::parallel_for(count,
                             cl::sycl::kernel_functor([=](cl::sycl::id<> item) {
                               int i = item.get_global(0);
                               c[i] = a[i] + b[i];
                             }));
    });
  }
}
```

Extrait 7.5 – Addition de vecteurs en SYCL

7.3 La cible OpenCL de FREIA

OpenCL est une des cibles du framework FREIA [15] préexistante à cette thèse. Elle permet ainsi à des applications FREIA d'être automatiquement exécutées sur des accélérateurs graphiques. Contrairement au langage flot de données Sigma-C présenté au chapitre 6, OpenCL dispose d'une interface de programmation en langage C. Ainsi, FREIA a pu directement être implémentée en OpenCL sans faire intervenir d'outil de compilation source-à-source. Toutefois, en parallèle à cette implémentation bibliothèque pure, une extension du compilateur source-à-source PIPS permet de générer automatiquement des kernels OpenCL optimisés notamment via fusion d'opérateurs [29].

Nous décrivons dans cette section 7.3 les éléments caractéristiques de ces deux implémentations, ainsi qu'une comparaison de leur efficacité.

7.3.1 La version bibliothèque

Au même titre que Fulguro [26], SPoC [27], Terapix [20] et, depuis le démarrage de la présente thèse, SMIL [46], OpenCL est une des cibles logicielles de la bibliothèque FREIA. Chaque opérateur de base de FREIA dispose ainsi d'une implémentation sous la forme d'un kernel écrit en OpenCL-C, factorisé grâce à l'utilisation de macros préprocesseur. Le

```c
#include <stdlib.h>

#define N 1000000000

int main(void) {
  float *a = malloc(N * sizeof(float));
  float *b = malloc(N * sizeof(float));
  float *c = malloc(N * sizeof(float));

  unsigned int i;
#pragma omp target
#pragma omp parallel for
  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];

  free(a);
  free(b);
  free(c);

  return 0;
}
```

Extrait 7.6 – Addition de vecteurs en OpenMP 4.0

```c
#include <stdlib.h>

#define N 1000000000

int main(void) {
  float *a = malloc(N * sizeof(float));
  float *b = malloc(N * sizeof(float));
  float *c = malloc(N * sizeof(float));

  unsigned int i;
#pragma acc kernels copyin(a[0 : N], b[0 : N]), copyout(c[0 : N])
  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];

  free(a);
  free(b);
  free(c);

  return 0;
}
```

Extrait 7.7 – Addition de vecteurs en OpenACC

développement de cette implémentation a été réalisé par Michel Bilodeau, ingénieur du Centre de morphologie mathématique de l'École des mines en 2011.

Les kernels

L'extrait 7.8 présente ainsi la macro utilisée afin de générer les opérateurs arithmétiques de traitement d'images binaires (pixel à pixel entre deux images). Ce kernel agit sur des tuiles de pixels calculées en fonction des dimensions des images sur l'hôte. Quelques spécifi-

cités du langage OpenCL-C ont été utilisées pour améliorer les performances et faciliter la programmabilité :

des variables vectorielles, comme ici `geom` et `ofs`, utilisées en remplacement de structures de données ;

des fonctions mathématiques intrinsèques à la bibliothèque OpenCL-C servent à spécialiser la macro.

```
18 KERNEL void BIOP(arith_, OPERATION_NAME, )(GLOBAL PIXEL *imOut,
19                                             GLOBAL PIXEL *imIn1,
20                                             GLOBAL PIXEL *imIn2, const int val1,
21                                             TGeom geom, TOfs ofs) {
22   // Generic pixel-wise operation on 2 inputs, one output
23   // imOut = imIn1 Op imIn2
24   int gid = (get_global_id(1)) * geom.y * geom.z + (get_global_id(0) * geom.x);
25   int i, line;
26   const int pitch = geom.y;
27   const int ofsOut = ofs.x, ofsIn1 = ofs.y, ofsIn2 = ofs.z;
28
29   for (line = 0; line < geom.z; line++) {
30     for (i = gid; i < (gid + geom.x); i++) {
31       imOut[i + ofsOut] = OPERATION(imIn1[i + ofsIn1],
32                                     imIn2[i + ofsIn2], val1);
33     }
34     gid += pitch;
35   }
36 }
```

Extrait 7.8 – Opérateurs arithmétiques binaires : implémentation OpenCL dans FREIA

Les opérateurs morphologiques disposent, quant à eux, de plusieurs implémentations qui dépendent de l'élément structurant passé en paramètre. Les calculs sont également vectorisés afin de profiter de cette autre forme de parallélisme.

À noter que ces kernels ne sont pas fortement couplés à l'interface FREIA : les utilisateurs de FREIA peuvent, si besoin, fournir leur propre implémentation OpenCL-C, si la version par défaut ne leur convient pas.

L'environnement d'exécution hôte

Côté hôte, l'implémentation OpenCL de FREIA réalise deux fonctions distinctes :

— elle initialise la plateforme OpenCL et les *compute devices* et elle s'occupe également de compiler les kernels ;

— elle lance l'exécution des différents kernels lors des appels aux fonctions FREIA correspondants ; cette partie est également responsable du tuilage des données pour les opérateurs au voisinage.

7.3.2 La version générée par le compilateur source-à-source PIPS

FREIA propose également une seconde implémentation OpenCL via l'utilisation du compilateur source-à-source PIPS. De façon analogue à la génération automatique de code Sigma-C décrite au chapitre 6, le code OpenCL est ici généré après analyse du code des applications.

Les applications résultantes profitent ainsi des optimisations inter-procédurales du compilateur PIPS : décomposition des fonctions complexes en suite d'opérateurs de base, élimination des opérations inutiles, fusion des branches de calcul redondantes et parfois fusion d'opérateurs. Les kernels sont ainsi réécrits sous une forme simplifiée, en évitant les calculs vectorisés et les intrinsèques OpenCL-C.

7.3.3 Comparaisons

Ces deux implémentations OpenCL ont été comparées sur sept applications FREIA dans [29]. Plusieurs accélérateurs ont ainsi été testé : deux processeurs centraux multicœurs et trois processeurs graphiques de la marque Nvidia. Les expériences ont porté sur les effets des optimisations réalisées par le compilateur source-à-source PIPS.

Les résultats montrent que ces optimisations génèrent un speedup moyen de $\times 2.7$ sur les processeurs multicœurs et $\times 18.1$ sur les accélérateurs graphiques testés. Ces expériences montrent l'intérêt de la génération automatique de noyaux de calcul et des optimisations réalisées par PIPS par rapport à une utilisation conventionnelle de la spécification OpenCL. Les kernels générés, bien que n'utilisant ni les intrinsèques OpenCL ni les structures de données vecteur, s'avèrent au final plus performants.

7.4 OpenCL adapté au manycore MPPA

Afin de répondre aux attentes de ses clients en ce qui concerne la programmabilité, Kalray a développé pour sa puce MPPA une implémentation d'OpenCL. Celle-ci vise à terme à remplacer le modèle de programmation flot-de-données discuté au chapitre 6 : fondé sur le langage Sigma-C maison, ce dernier oblige à repenser et réécrire entièrement les applications que l'on souhaite faire fonctionner sur MPPA.

Nous décrivons dans cette section les spécificités du MPPA en tant qu'accélérateur OpenCL et comment nous avons adapté la cible OpenCL de FREIA à cette puce.

7.4.1 Le modèle de programmation OpenCL et le MPPA

À mi-chemin entre un processeur classique et un processeur graphique étant donné le nombre de cœurs le composant, le MPPA est *a priori* adapté au modèle de programmation OpenCL. Les seize clusters de calcul forment ainsi seize Compute Units, tandis que les 256 cœurs de calcul sont autant de Processing Elements.

Cependant, les accès à la mémoire globale attachée n'étant pas possible directement depuis les seize clusters de calcul, Kalray a proposé une solution pour apporter directement les données depuis cette mémoire via un système de pagination depuis un des clusters d'entrée/sortie. Ce système appelé DSM (« *Distributed Shared Memory* ») est composé d'un serveur de pages situé sur le cluster d'entrée/sortie et de clients sur les clusters de calcul. Les clusters de calcul peuvent alors envoyer une requête au serveur, si besoin, pour qu'il transmette la page mémoire demandée à travers le NoC. Celle-ci sera alors stockée dans la mémoire partagée des clusters de calcul. La DSM permet un accès virtuellement direct à la mémoire globale, mais, puisque implémentée de façon logicielle, les performances peuvent en souffrir.

L'implémentation OpenCL de Kalray repose majoritairement sur le projet *pocl* (« Portable Computing Language ») [102, 78], qui vise à fournir une implémentation open-source d'OpenCL facilement adaptable à de nouvelles cibles matérielles. Ce dernier s'appuie sur le compilateur LLVM, qui dispose d'un vaste ensemble de cibles matérielles, afin de compiler les kernels OpenCL-C.

La chaîne de compilation AccessCore de Kalray, en version 2.2 lors de l'écriture de ce chapitre, supporte un sous-ensemble de la spécification OpenCL 1.2, sortie en 2012. Certaines fonctions OpenCL, faisant partie de cette spécification 1.2, ne sont ainsi pas implémentées, obligeant parfois à trouver des contournements. En particulier, le langage OpenCL-C servant à écrire les kernels n'est pas entièrement disponible : les calculs vectorisés et les intrinsèques mathématiques ne sont pas supportés. Ceci empêche la compilation des kernels OpenCL-C de FREIA, qui s'appuient sur ces fonctionnalités. La situation semble néanmoins évoluer : la prochaine version 2.5 de la chaîne de compilation AccessCore apportera enfin le support de ce langage.

De par son architecture particulière formée de seize clusters de seize cœurs chacun et contrairement à des processeurs graphiques standard, le MPPA impose la topologie d'exécution des kernels définie au travers de la fonction

```
1  cl_int clEnqueueNDRangeKernel();
```

Ainsi, le nombre de kernels par Compute Unit doit être fixé à 16, soit le nombre de Processing Elements par CU. Ces contraintes devront être respectées lors du port de FREIA OpenCL sur MPPA.

7.4.2 Adapter FREIA au MPPA

Les contraintes de l'implémentation OpenCL actuelle de Kalray et, notamment, l'absence des opérateurs vectoriels et des intrinsèques OpenCL-C empêchent la compilation des kernels OpenCL de FREIA. Les prochaines évolutions de la chaîne de compilation de Kalray pourront néanmoins suffire, afin de tester celle-ci.

Cependant, les kernels générés automatiquement par PIPS utilisent suffisamment peu ces fonctionnalités pour que de légères altérations permettent l'exécution sur MPPA sans perdre en portabilité. Les modifications que nous avons apportées consistent ainsi en

- la désactivation de la compilation des kernels issus de la bibliothèque FREIA, qui provoquaient des erreurs suite à l'utilisation d'opérateurs vectoriels et d'intrinsèques,
- le remplacement des intrinsèques OpenCL-C `max`, `min`, `sub_sat`, `add_sat` et `abs_diff` par des calculs explicites,
- le remplacement d'appels vers les fonctions de l'interface OpenCL `clWaitForEvents` et `clEnqueueWaitForEvents` non supportées ou non implémentées par *pocl* par des appels équivalents et fonctionnels à `clFinish` et
- l'adaptation de l'appel à `clEnqueueNDRangeKernel` aux contraintes évoquées plus haut de seize instances de kernels par Compute Unit.

Suite à ces adaptations, notre ensemble d'applications peut s'exécuter correctement sur le MPPA.

7.4.3 Le MPPA face à d'autres accélérateurs OpenCL

Grâce aux modifications apportées à l'implémentation OpenCL de FREIA et explicitées à la section précédente, nous avons pu tester les performances du MPPA sur notre jeu d'applications de traitement d'images. Nous avons ainsi comparé les performances des deux premières génération de processeur MPPA, puis confronter le MPPA aux processeurs décrits en sous-section 7.3.3.

Deux générations de MPPA

Ayant eu accès à deux générations de processeurs MPPA, nous avons pu comparer les temps d'exécution de nos sept applications sur les deux cibles.

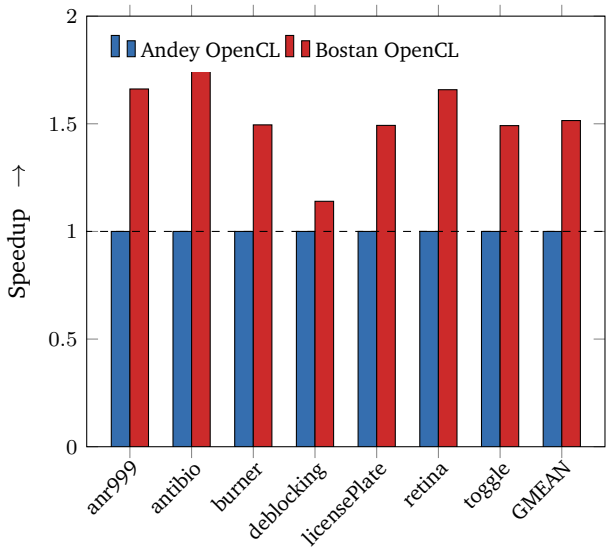


FIGURE 7.3 – Speedups de sept applications de traitement d'images sur deux générations de processeur MPPA avec le modèle de programmation OpenCL : (1) OpenCL sur Andey (référence) ; (2) OpenCL sur Bostan

L'histogramme en figure 7.3 présente ainsi le différentiel de performances entre la première génération, « Andey », et la seconde, « Bostan ». Le chapitre 5 compare en section 5.3.4 la performance d'OpenMP sur un cluster de calcul. Les résultats montraient alors un speedup allant jusqu'à $\times 6$ avec un seul thread. Dans le cas présent, les applications OpenCL utilisent les seize clusters de calcul sur les deux puces. Le speedup est ici moindre, avec une moyenne de $\times 1.5$, ce qui reste inférieur à l'évolution mesurée avec OpenMP. L'utilisation de la mémoire globale, à travers la DSM, ainsi que les communications inter-clusters semblent diminuer les apports de la seconde génération Bostan.

Les accélérateurs matériels

Après ces tests préliminaires, nous avons repris les chiffres de [29] concernant les deux processeurs graphiques Nvidia Quadro et Tesla afin de confronter le MPPA aux accélérateurs OpenCL déjà présentés plus haut. Nous avons également mesuré les temps d'exécution sur le CPU hôte Intel Core i7-3820. Nous figurons en tableau 7.1 les différents accélérateurs utilisés dans cette comparaison, ainsi que leur consommation énergétique moyenne.

Les applications comparées le sont à code égal : il s'agit du code généré par PIPS et non retouché par la suite. Ce sont donc les accélérateurs eux-mêmes et leurs implémentations OpenCL qui sont comparés.

Comparaison des performances

Nous avons d'abord comparé les temps d'exécution de nos applications sur les différents accélérateurs par rapport à la puce MPPA « Bostan ». Le résultat, présenté en figure 7.4,

Hardware	Type	Power
Kalray MPPA	Manycore	10 W
Intel Core i7-3820	CPU	130 W
Nvidia Quadro 600	GPU	40 W
Nvidia Tesla C 2050	GPU	240 W

Table 7.1 – Description des accélérateurs matériels OpenCL

montre que le MPPA, est en moyenne, de 3 à 20 fois plus lent que les autres cibles dans l'exécution du même code OpenCL. Il est à noter que les évolutions actuelles de Kalray sur sa pile logicielle OpenCL laissent présager des améliorations dans ce domaine. De plus, nous n'avons, contrairement au travail effectué au chapitre 6, pas implémenté d'optimisations spécifiques au MPPA dans FREIA.

L'application « deblocking » semble particulièrement plus lente sur le MPPA en comparaison des autres accélérateurs. Au contraire des autres applications, celle-ci dispose d'un graphe des tâches plus large que profond : plusieurs images intermédiaires coexistent simultanément. Cela entraîne de nombreux accès mémoire à la DDR qui grèvent particulièrement les performances.

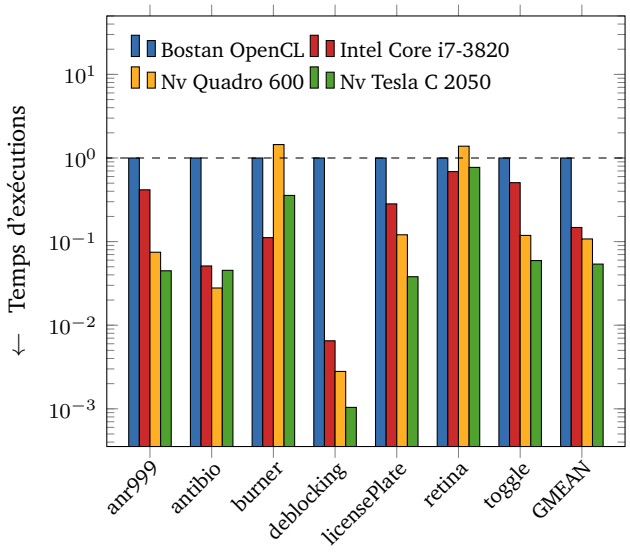
En ce qui concerne l'efficacité énergétique, le MPPA profite de sa faible consommation : seul le GPU Quadro 600 est, en moyenne, plus efficace. Ainsi, à code applicatif égal, le processeur MPPA offre, d'après ces mesures, un bon compromis entre performance et consommation énergétique.

7.5 Conclusion

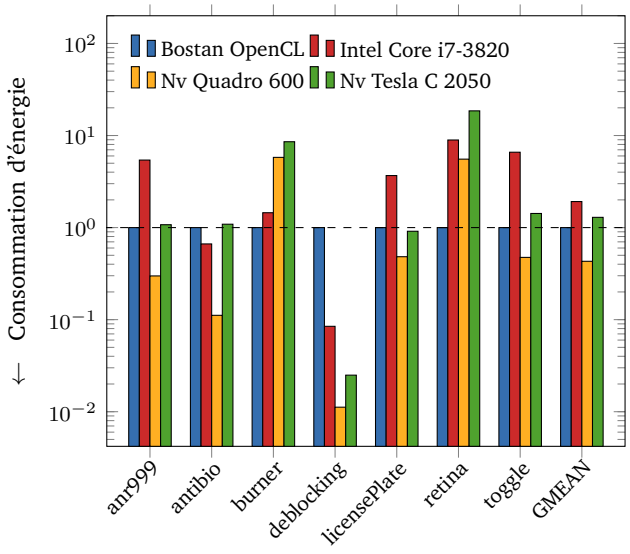
Les processeurs graphiques, de par leur architecture qui multiplie les unités de calcul, sont devenus des accélérateurs de choix pour les calculs scientifiques. Les interfaces traditionnelles de programmation graphique telles OpenGL et DirectX, trop spécifiques, ont pavé la voie à des *frameworks* dédiés aux calculs généralistes sur processeurs graphiques : CUDA et OpenCL en sont les principaux représentants, et les évolutions vont vers toujours plus de programmabilité, en fusionnant par exemple le code hôte et le code accéléré.

OpenCL est un modèle de programmation particulièrement portable : il peut s'adapter autant aux GPUs qu'aux processeurs multicœurs et manycore. En particulier, le processeur MPPA de Kalray implémente un sous-ensemble de l'interface OpenCL. Cela permet ainsi à ce processeur manycore d'exécuter des applications OpenCL sans ou avec peu de modifications. Dans ce chapitre, nous étudions la portabilité d'applications FREIA sur le MPPA via OpenCL et comparons les performances de cet accélérateur à plusieurs autres.

Nous avons étendu le framework FREIA pour rendre sa cible OpenCL compatible avec le MPPA, de sorte à améliorer davantage la portabilité des applications FREIA. En effet, le MPPA ne supporte qu'un sous-ensemble de la spécification : des modifications, bien que mineures, sont nécessaires. Suite à ces travaux, nous avons été en mesure de comparer les performances du MPPA à celles de plusieurs accélérateurs OpenCL. Les résultats montrent que, sur notre jeu d'applications FREIA, le MPPA est en-deçà de ses concurrents pour les temps d'exécution, mais se montre à la hauteur en matière de consommation énergétique. Les travaux actuels sur la pile logicielle OpenCL par Kalray laissent toutefois espérer une amélioration dans ce domaine.



(a) Temps d'exécution relatifs



(b) Consommation d'énergie relative

FIGURE 7.4 – Performance relative (échelle logarithmique) des plateformes OpenCL : (1) MPPA Bostan (référence) ; (2) CPU Intel Core i7-3820 ; (3) GPU Nvidia Quadro 600 ; (4) GPU Nvidia Tesla C 2050

Chapitre 8

Synthèse

I know when to go out
And when to stay in
Get things done

I catch a paper boy
But things don't really change
I'm standing in the wind
But I never wave bye-bye

But I try
I try

— David Bowie, *L'amour moderne (extr.)*

Les contributions décrites dans les chapitres précédents se focalisent chacune sur un modèle de programmation spécifique pour nos applications de traitement d'images. Ainsi, le chapitre 5 est dédié au modèle OpenMP sur un cluster de calcul du processeur MPPA, le chapitre 6, au modèle flot de données à travers le langage Sigma-C, et, enfin, le chapitre 7, au modèle OpenCL pour les accélérateurs matériels.

Les résultats des expérimentations réalisées au cours de ces travaux permettent de comparer les différents modèles de programmation supportés par les deux générations de processeur MPPA, ainsi que de comparer le MPPA à d'autres accélérateurs matériels. Nous effectuons une telle comparaison dans la section 8.1. L'agrégation des chaînes de compilation, détaillées dans ces chapitres, et du compilateur `smiltofreia`, décrit dans le chapitre 4, forme un environnement logiciel complet dédié au traitement d'images, qui cible différentes classes d'architectures matérielles et notamment le processeur MPPA, selon différents modèles de programmation. Cet environnement logiciel est décrit dans la section 8.2.

8.1 Une puce, différents modèles de programmation

Le processeur MPPA propose plusieurs niveaux de parallélisme : seize cœurs de calcul autour d'une mémoire partagée, seize clusters communiquant au travers d'un réseau sur puce et une puce se comportant comme un coprocesseur servant à accélérer certains cal-

culs d'un processeur hôte. Pour tirer parti d'une telle architecture, différents modèles de programmation peuvent être utilisés :

— OpenMP et POSIX Threads visent en particulier les architectures à mémoire partagée et sont supportés sur les clusters de calcul ;
— les transferts de données et synchronisations entre clusters utilisent une bibliothèque de communication à travers le réseau sur puce qui rappelle l'interface MPI ;
— le modèle flot de données abstrait le matériel aux dépens d'une réécriture totale des applications dans un nouveau paradigme ;
— OpenCL est une interface standard dédiée aux architectures hétérogènes, afin de décharger les calculs depuis un processeur hôte vers un accélérateur matériel disposant d'une mémoire séparée ;
— un dernier modèle, de plus bas niveau, permet de gérer finement les ressources d'un cluster de calcul, afin de construire des systèmes d'exploitations, que nous n'avons pas jusqu'à présent utilisé.

Nous avons, au cours de cette thèse, testé ces différents modèles de programmation — à l'exception toutefois du dernier, pas vraiment destiné au développement d'applications. Nous n'avons pas, à notre grand regret, été en mesure de finaliser l'examen du modèle par passage de message entre les différents clusters : une étude préliminaire est néanmoins présentée dans le chapitre 9.

8.1.1 OpenMP sur un cluster de calcul

Dans le chapitre 5, nous avons ainsi comparé les performances d'un cluster de calcul du MPPA à celles d'un processeur standard à plusieurs cœurs. Pour ce faire, nous avons effectué une compilation croisée de la bibliothèque de traitement d'images SMIL, qui utilise OpenMP. De cette façon, la même application SMIL peut s'exécuter à la fois sur un processeur standard et sur le MPPA, sans modification du code source. Cependant, la faible capacité de la mémoire partagée d'un cluster de calcul réserve cette application à des fins de recherche uniquement : un seul cluster sur seize est utilisé, et les images trop grandes ne sont pas découpées avant d'être transférées. Au niveau performances, le cluster de calcul se montre environ six fois plus lent que notre processeur standard de référence, à relativiser face à la réserve des quinze autres clusters inutilisés. La consommation énergétique, malgré la forte différence dans les temps d'exécution, se montre, comme attendu, en faveur du MPPA.

8.1.2 Sigma-C et le modèle flot de données

Le modèle flot de données est un paradigme de programmation particulièrement adapté aux architectures massivement parallèles. Le langage Sigma-C a été développé par le laboratoire List du CEA en collaboration avec Kalray, spécifiquement pour le processeur MPPA. Ainsi que décrit au chapitre 6, nous avons développé un compilateur source-à-source qui, à partir d'une application écrite dans un langage de haut niveau (FREIA), génère le code Sigma-C correspondant, qui s'appuie sur une bibliothèque légère d'opérateurs de traitement d'images. Les différents travaux d'optimisation qui ont suivi ont fait l'objet d'une publication dans la conférence *Languages and Compilers for Parallel Computing* [122]. Le langage Sigma-C n'est cependant plus supporté par le MPPA aujourd'hui, le modèle OpenCL ayant été préféré.

8.1.3 OpenCL pour les architectures hétérogènes

Le processeur MPPA supporte la spécification OpenCL, dédiée aux architectures hétérogènes. Le projet FREIA disposant d'une implémentation OpenCL, nous l'avons portée sur le MPPA et rapporté cette expérience au chapitre 7. Le MPPA ne supporte, en effet, qu'un sous-ensemble des fonctionnalités de la spécification, ce qui a donné lieu à quelques ajustements. Ici encore, des tests sur quelques applications de traitement d'images montrent que le MPPA est en moyenne plus lent qu'un ensemble d'architectures concurrentes, tandis que sa consommation énergétique globale reste en général inférieure. Les travaux d'optimisation des performances et d'implémentation de la spécification évoluent cependant en parallèle et laissent présager des améliorations futures des performances.

8.1.4 Performance et version du processeur MPPA

L'accélérateur MPPA a, au cours de cette thèse, subi une évolution majeure sous la forme d'une nouvelle version de cette puce. Le premier modèle, nom de code « Andey » et mis sur le marché début 2013, supportait les modèles de programmation flot de données et parallélisme explicite à base de POSIX Threads ou OpenMP avec communications inter-clusters. Le support du modèle de programmation OpenCL a été ajouté depuis, suite à des demandes de la part des clients de Kalray. Ce modèle n'est en effet pas restreint au MPPA, contrairement au langage flot de données Sigma-C, et le modèle parallèle explicite est trop proche de la cible matérielle pour être portable. Le second, surnommé « Bostan », garde l'architecture globale en seize clusters de seize cœurs de calculs, mais améliore les performances de ces derniers en augmentant notamment la fréquence d'horloge et la bande passante mémoire. Il abandonne cependant le support du langage Sigma-C pour se concentrer sur OpenCL et POSIX Threads/OpenMP.

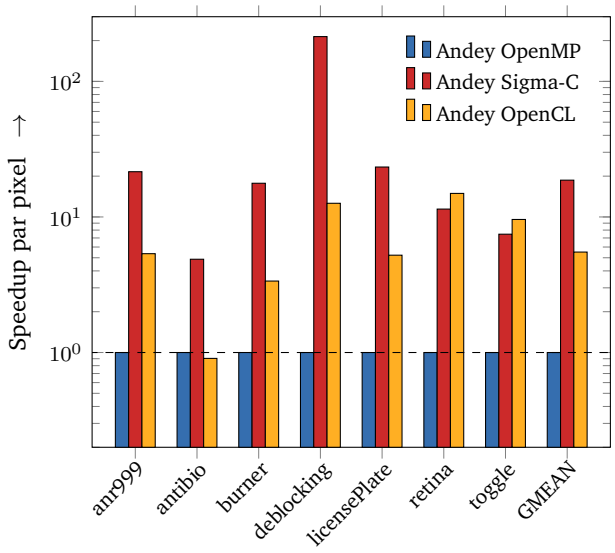


FIGURE 8.1 – Speedups normalisés par pixel de sept applications sur le MPPA « Andey » suivant différents modèles de programmation : (1) OpenMP sur un cluster Andey (référence) ; (2) Sigma-C sur Andey ; (3) OpenCL sur Andey

Nous avons comparé les temps d'exécution des sept mêmes applications issues du projet

FREIA selon les différents modèles de programmation sur les deux générations de puces MPPA. Le graphique de la figure 8.1 se concentre ainsi sur les trois modèles supportés par la première génération « Andey ». Nous comparons le modèle OpenMP exécuté sur un cluster de calcul à OpenCL et Sigma-C, qui tirent parti de plusieurs clusters. Dans tous les cas, Sigma-C se montre plus efficace qu'OpenMP, puisque non restreint à un seul cluster. De plus, l'OpenMP utilisé est restreint au parallélisme de données, tandis que Sigma-C gère plus efficacement les chaînes de traitement parallèles, qui caractérisent généralement les applications de traitement d'images. Le modèle OpenCL reste ici généralement plus performant qu'OpenMP, mais moins que Sigma-C. Les accès à la mémoire globale, qui ne peuvent s'effectuer qu'au travers du cluster d'entrée/sortie, qui devient ainsi un goulot d'étranglement, limitent les performances de ce modèle.

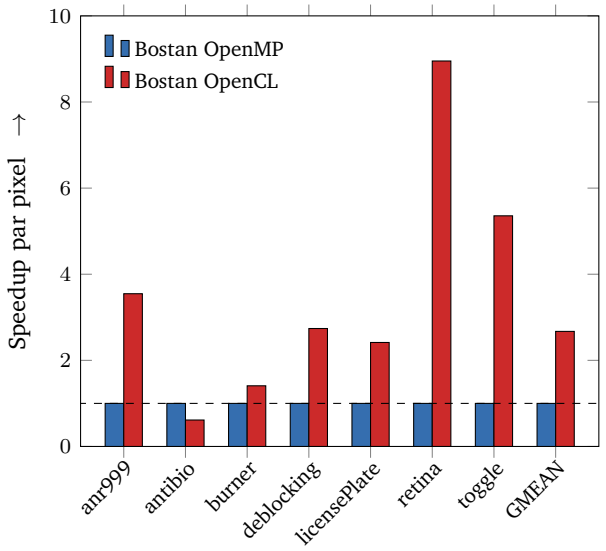
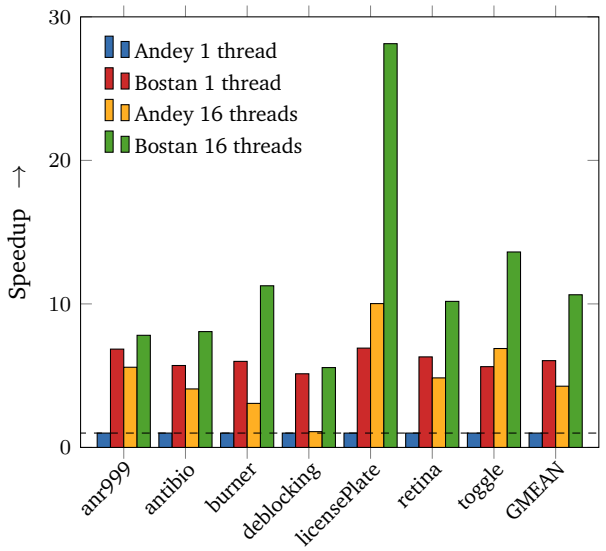


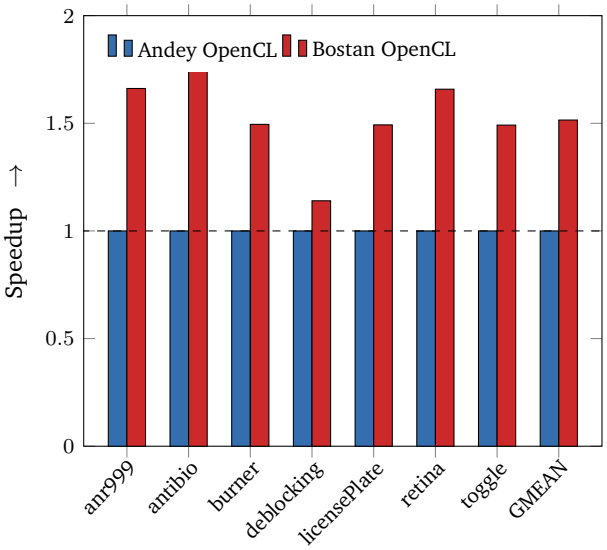
Figure 8.2 – Speedups normalisés par pixel de sept applications sur le MPPA « Bostan » suivant différents modèles de programmation : (1) OpenMP sur un cluster Bostan (référence) ; (2) OpenCL sur Bostan

La figure 8.2 présente la performance des deux modèles OpenMP sur un cluster et OpenCL pour le processeur MPPA de seconde génération, Bostan. Les variations entre OpenCL et OpenMP sont très semblables à celles de la génération précédente : OpenCL est plus performant, puisque n'étant pas limité à un unique cluster de calcul, soit seize cœurs sur les 256 qu'offre le MPPA. Cependant, une extension d'OpenMP à toute la puce pourrait se montrer plus efficace.

Enfin, la figure 8.3 — qui reproduit la figure 5.6 et la figure 7.3 — compare les deux générations de puces sur les modèles OpenMP et OpenCL. Nous constatons ici les avancées dans la micro-architectures des cœurs de calcul entre les deux générations, qui sont très marquées dans le cas du modèle OpenMP, avec des temps de calcul en moyenne trois fois inférieurs. Le constat est plus nuancé avec OpenCL : les traitements sont certes 50% plus rapides, ce qui s'explique par le rapport des fréquences d'horloge entre les deux générations de MPPA, mais l'évolution des performances reste limitée de ce côté. Nous espérons que les évolutions futures dans la pile logicielle OpenCL amélioreront ces performances.



(a) OpenMP (1 cluster de calcul)



(b) OpenCL

FIGURE 8.3 – Comparaison des deux générations de processeur MPPA « Andey » et « Bostan » sur les modèles OpenMP et OpenCL

8.1.5 Comparaison des modèles de programmation

Les modèles de programmations étudiés séparément dans les chapitre 5, chapitre 6 et chapitre 7 permettent chacun d'exploiter l'accélérateur MPPA. Les avantages et inconvénients de ces modèles de programmation ont été rassemblés dans le tableau 8.1. Nous y avons également fait figurer le modèle parallèle explicite avec communications inter-clusters, que nous décrivons plus en détail dans le chapitre 9.

Modèle	Avantages	Inconvénients
OpenMP	• répandu • portage rapide	• 16 cœurs sur 256 • mémoire : 2 Mo (code, données) • transferts depuis l'hôte
Sigma-C	• performant • communications	• trop statique • contrôle hôte • nouveau langage • abandonné
OpenCL	• répandu • portage rapide	• implémentation en cours • accès mémoire • performances ?
OpenMP + IPC	• 256 cœurs • performances ?	• tuilage • communications

Table 8.1 – Comparaison des différents modèles de programmation pour le MPPA étudiés durant cette thèse

Nous avons comparé ces différents modèles de programmation au travers des temps d'exécution de nos sept applications, avec comme référence le modèle OpenMP sur un cluster du MPPA Bostan. Les résultats de la figure 8.4 confirment l'efficacité du modèle flot de données et du langage Sigma-C, malheureusement abandonné, ainsi que le potentiel latent d'OpenMP, une fois étendu à tous les clusters.

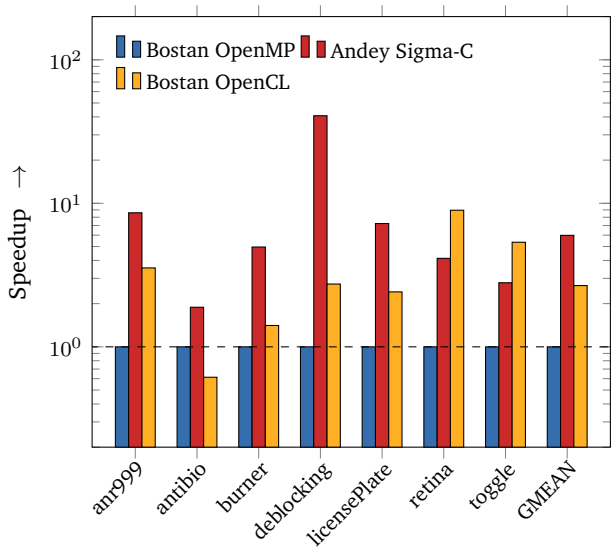


Figure 8.4 – Speedups par pixel de sept applications sur les MPPA « Andey » et « Bostan » suivant différents modèles de programmation : (1) OpenMP sur un cluster Bostan (référence) ; (2) Sigma-C sur Andey ; (3) OpenCL sur Bostan

Les performances du modèle flot de données sur le MPPA nous incitent à le comparer à d'autres cibles matérielles de FREIA. En figure 8.5, nous représentons ainsi la consommation

énergétique de nos applications de test exécutées en Sigma-C sur le MPPA Andey, sur CPU via l'implémentation SMIL, sur FPGA avec la cible SPoC de FREIA et en OpenCL sur deux GPUs. Sur ce jeu d'applications, le MPPA Andey se montre plus efficace énergétiquement que ses concurrents, à l'exception du FPGA SPoC, avec lequel il fait jeu égal. Dans l'impossibilité de tester Sigma-C sur la génération Bostan, nous ne pouvons que supposer ce dernier meilleur en matière d'efficacité énergétique.

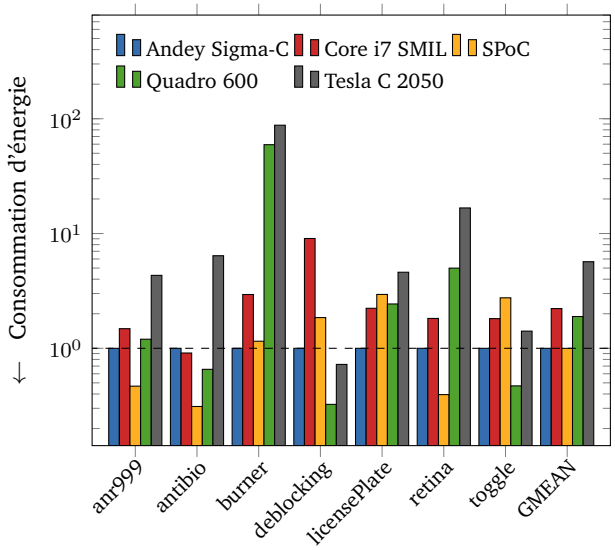


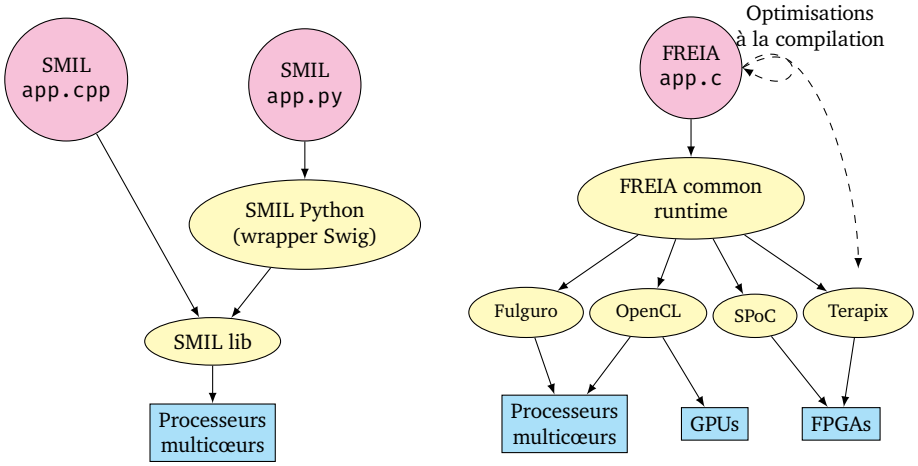
FIGURE 8.5 – Consommation énergétique relative de sept applications exécutées sur quatre accélérateurs : (1) Sigma-C sur le MPPA Andey (référence) ; (2) Intel Core i7-3820 via SMIL ; (3) accélérateur SPoC implémenté sur FPGA ; (4) OpenCL sur un GPU Nvidia Quadro 600 ; (5) OpenCL sur un GPU Nvidia Tesla C 2050

8.2 Un environnement logiciel complet pour le traitement d'images

Une des finalités de cette thèse a été de compléter la chaîne de compilation FREIA afin de cibler en aval le processeur MPPA. Cette chaîne a également été interfacée en amont avec la bibliothèque SMIL de façon à en simplifier le point d'entrée avec un langage applicatif de plus haut niveau.

Avant les travaux de cette thèse, les bibliothèques SMIL et FREIA étaient indépendantes, ainsi qu'illustré par la figure 8.6.

Le framework FREIA ne disposait alors que de quatre implémentations : Fulguro, l'implémentation logicielle, SPoC et Terapix pour les FPGAs et OpenCL pour les processeurs graphiques. Afin de développer la portabilité et la programmabilité de FREIA, plusieurs briques logicielles ont été développées au cours de cette thèse, au travers des différents modèles de programmation offerts par le processeur MPPA. Une liste exhaustive est disponible dans le tableau 8.2. L'état de la chaîne de compilation SMIL–FREIA à la fin de cette thèse est également résumée dans la figure 8.7 : `smiltofreia` a créé un pont entre les interfaces de programmations de SMIL et FREIA, et SMIL est devenue une cible logicielle de FREIA, liant ainsi les deux chaînes de compilation. L'ajout des cibles Sigma-C et l'adaptation d'OpenCL au MPPA permettent à FREIA de cibler cet accélérateur. Enfin, la bibliothèque SMIL peut se compiler directement à destination d'un cluster de calcul du MPPA.



(a) Pile logicielle SMIL (b) Pile logicielle FREIA

Figure 8.6 – État initial des piles logicielles SMIL et FREIA avant cette thèse

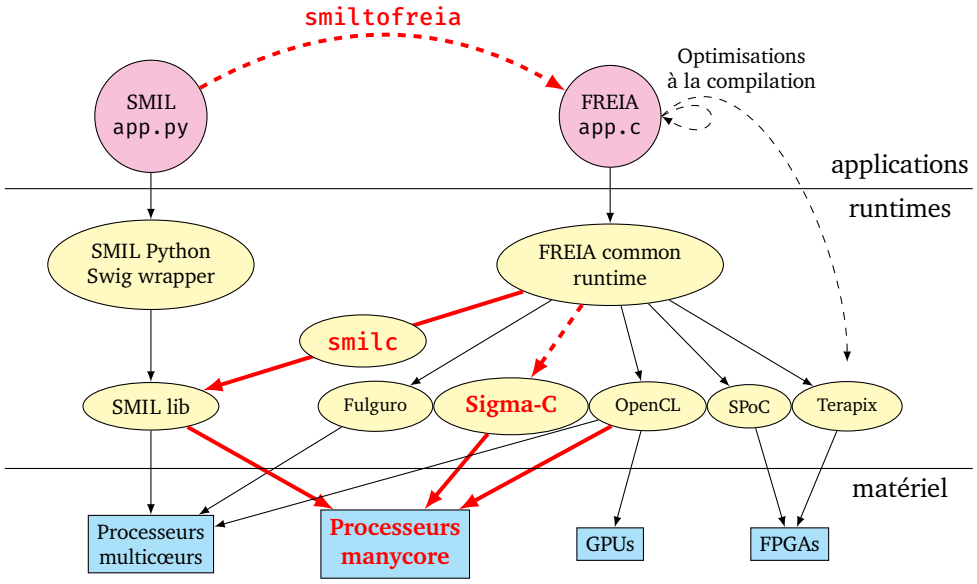


Figure 8.7 – État final des piles logicielles SMIL et FREIA a la fin de cette thèse : les éléments en rouge/gras ont été développés durant celle-ci

Le couplage entre SMIL et FREIA — nouvelle interface logicielle, d'un côté, et conversion automatique, de l'autre — améliore aussi les performances et la portabilité des applications SMIL, tout en augmentant la programmabilité des applications FREIA. C'est dans ce cadre que l'outil `smiltofreia` a été développé. Les détails de cet outil ont été présentés au chapitre 4 et ont été communiqués à la conférence *Compilers for Parallel Computing* [123].

Au final, nous obtenons une chaîne de compilation unique acceptant en entrée des applications écrites en SMIL Python ou en FREIA C et pouvant cibler de multiples façons

Logiciel	Description
FREIA–Sigma-C	Nouvelle implémentation flot de données
PIPS–Sigma-C	Génération automatique de subgraphs Sigma-C
`smilc`	Wrapper C autour de SMIL (C++)
FREIA–SMIL	Nouvelle implémentation vers CPU
SMIL–MPPA	SMIL sur un cluster de calcul
FREIA–OpenCL	Adaptation de FREIA au MPPA
`smiltofreia`	Conversion d'applications SMIL en FREIA
FREIA–MPPA	Nouvelle implémentation, parallélisme explicite

TABLE 8.2 – Logiciels développés pendant cette thèse

le processeur MPPA de Kalray, mais également d'autres accélérateurs, comme des GPUs avec OpenCL. Cette chaîne de compilation, que nous appelons `freiacc`, est représentée en figure 8.8. En fournissant une interface de haut niveau au travers du langage spécifique SMIL Python, `freiacc` facilite la programmation d'applications de traitement d'images. Grâce aux multiples cibles de FREIA et à l'utilisation de modèles de programmation largement adoptés comme OpenCL, `freiacc` assure la portabilité de ces applications sur une large gamme de matériels. Enfin, l'utilisation d'un modèle de programmation par type de cible, associé aux optimisations réalisées par le compilateur source-à-source PIPS, assurent de bonnes performances à nos applications. Concernant le processeur MPPA, le langage flot de donnéesSigma-C nous fournit des performances satisfaisantes, mais restreintes à la génération « Andey ». Nous espérons toutefois atteindre la performance crête sur « Bostan » en utilisant le modèle OpenMP avec communications inter-clusters.

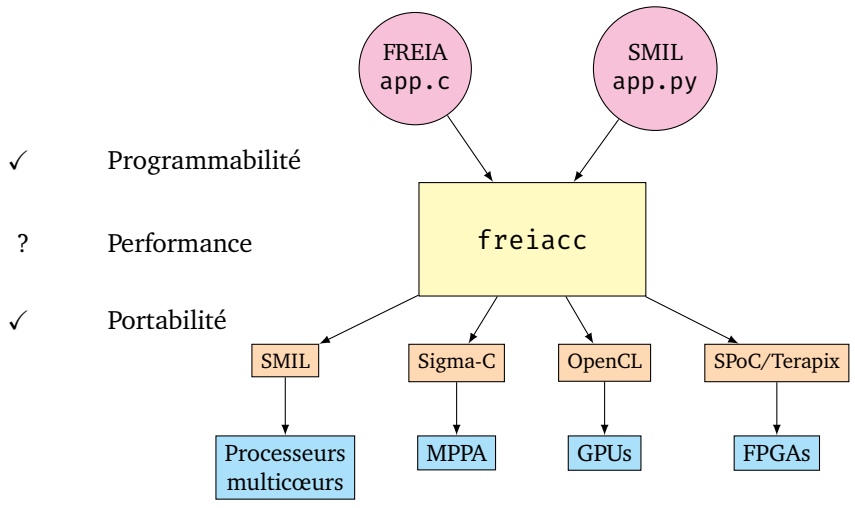


FIGURE 8.8 – Chaîne de compilation simplifiée : `freiacc` peut compiler des applications de traitement d'images écrites en FREIA ou en SMIL Python vers plusieurs catégories d'accélérateurs matériels, dont le MPPA

Cependant, l'ajout d'une cible dans cette chaîne de compilation est un travail long et coûteux : il s'agit de réimplémenter les opérateurs de traitement d'images, de développer l'environnement d'exécution associé et, enfin, de concevoir le générateur automatique de

code, tout en fournissant des performances acceptables. Notre approche a toutefois le mérite de conserver à l'identique le code source des applications, la portabilité étant assurée par la chaîne de compilation. Nous pouvons également profiter de certaines optimisations déjà implémentées et pouvant être partagées entre les cibles.

8.3 Conclusion

Au cours de cette thèse, différents outils ont été développés afin de réaliser des expérimentations sur les modèles de programmation supportés par le processeur MPPA. Ces outils et expériences sont indépendamment détaillés dans les précédents chapitres. Le présent chapitre agrège ces différentes contributions.

Nous comparons les modèles de programmation supportés par les MPPA « Andey » et « Bostan » et montrons que le modèle Sigma-C, bien qu'uniquement disponible sur Andey, se montre le plus efficace sur notre jeu d'applications. Nous comparons également la consommation énergétique de nos applications sur différents accélérateurs, dont Andey via Sigma-C. Nous montrons que le MPPA est très efficace dans ce domaine et avec ce langage flot de données, et les évolutions constatées avec OpenMP et OpenCL sous Bostan vont vers une amélioration de cette caractéristique.

Seconde contribution majeure, `freiacc` est la résultante de nos chaînes de compilation. Cet environnement logiciel compile des applications de traitement d'images écrites dans un langage spécifique de haut niveau vers différentes architectures suivant différents modèles de programmation, assurant à la fois programmabilité, portabilité et performances.

Chapitre 9

Conclusion

What was the start of all this ?
When did the cogs of fate begin to turn ?
Perhaps it is impossible to grasp that answer now,
From deep within the flow of time.

But for a certanty, back then
We loved so many yet hated so much
We hurt others and were hurt ourselves.

Yet even then we ran like the wind
Whilst our laughter echoed
Under cerulean skies…

— *Chrono Cross*

À mesure de l'évolution de l'informatique moderne, les unités de calcul ont gagné tant en performances qu'en complexité. Les processeurs modernes multiplient les unités de calcul, s'associent avec des coprocesseurs dédiés à certains types de calcul ou bien se regroupent dans des superordinateurs, reliés par des réseaux de communication. Les applications traditionnelles, développées dans un langage de programmation portable comme C ou C++, sont toujours capables de s'exécuter sur ces configurations — modulo recompilation —, mais ne peuvent généralement tirer parti que d'une unité de calcul à la fois.

De nouveaux modèles de programmation permettent de profiter des avantages des différentes catégories d'architectures matérielles, mais nécessitent généralement d'altérer de façon non triviale le code source des applications. Les langages spécifiques à un domaine, associés à des compilateurs permettant d'optimiser les codes applicatifs et de cibler des architectures matérielles non communes, facilitent également le travail des concepteurs d'applications.

Cette thèse s'est focalisée sur un problème d'optimisation de la programmabilité, de la portabilité et de la performance, restreint à domaine applicatif précis, celui du traitement d'images par morphologie mathématique, et à une architecture matérielle spécifique, celle du processeur MPPA, conçu par l'entreprise française Kalray. Ce processeur, qui cible les

marchés des systèmes embarqués ainsi que celui du calcul haute performance, présente différents niveaux de parallélisme qui rendent son utilisation efficace complexe.

9.1 Rappel des contributions

Plusieurs briques logicielles ont été développées au cours de cette thèse. Ces compilateurs, bibliothèques et environnements d'exécution ont permis de réaliser des expériences à partir d'applications de traitement d'images écrites dans un langage spécifique de haut niveau. Ces expériences ont montré que :

- les applications qui s'appuient sur le DSL de haut niveau SMIL Python ont des performances moindres que celles qui utilisent le DSL de plus bas niveau FREIA ;
- les applications SMIL Python s'écrivent en trois à cinq fois moins de lignes de code que les applications FREIA ;
- le compilateur `smiltofreia` permet de concilier la performance et la portabilité des applications FREIA à la programmabilité offerte par SMIL Python ;
- les nœuds à mémoire partagée du processeur MPPA peuvent être utilisés comme des processeurs conventionnels à forte efficacité énergétique, à condition d'abstraire les transferts de données ;
- ces nœuds sont toutefois limités par la capacité de la mémoire locale, qui oblige à limiter la taille du code applicatif au profit des données ;
- le modèle flot de données, au travers du langage Sigma-C, est trop statique : l'occupation des cœurs de calcul dépend du nombre d'appels aux opérateurs de l'application et la taille des données doit être connue à la compilation ;
- des optimisations simples permettent d'améliorer significativement les performances de Sigma-C, par exemple, en limitant les transferts de données ;
- le support et les performances d'OpenCL sur le MPPA restent aujourd'hui basiques, mais des évolutions sont prévues ;
- le MPPA de seconde génération, « Bostan », est bien plus rapide que son prédécesseur, « Andey », avec les modèles OpenMP et OpenCL ;

En combinant ces expériences, nous montrons, au chapitre 8, que le modèle flot de données est, parmi les modèles testés, le modèle le plus performant. Comparé à quatre autres accélérateurs matériels, ce modèle s'avère un des moins consommateurs d'énergie, sur notre jeu d'applications de traitement d'images. Le langage Sigma-C permet d'éviter les communications avec la mémoire DDR attachée, qui limitent les performances du modèle OpenCL. Toutefois, l'extension aux seize clusters de calcul du modèle OpenMP laisse présager de performances encore meilleures.

Les outils que nous avons développés s'intègrent pour former un environnement logiciel complet dédié au traitement d'images : `freiacc`. Cet environnement prend en entrée des applications écrites dans des langages spécifiques de haut niveau (SMIL Python ou FREIA) et cible différentes catégories d'accélérateurs matériels (CPUs, GPUs, FPGAS, processeurs manycore), grâce aux modèles de programmation supportés. Nous offrons ainsi à des spécialistes du traitement d'images la possibilité, sans coûts de développement supplémentaires, de porter leurs applications sur ces cibles matérielles et, notamment, le MPPA, qui s'avère des plus efficaces en matière de consommation énergétique. Nous prouvons ainsi par construction qu'il est possible de concilier programmabilité, portabilité et performance pour nos applications de traitement d'image sur cette cible manycore.

9.2 Perspectives : des progrès encore possibles

Quelques éléments resteront cependant inachevés, malgré trois années de labeur. Nous en décrirons ici deux : l'implémentation du modèle parallèle explicite, qui combine parallélisme de threads sur les clusters de calcul et communications explicites entre clusters, et qui vise à utiliser tous les cœurs du MPPA en même temps pour encore gagner en performance, ainsi que l'étude du portage d'un algorithme spécifique de traitement d'images, la segmentation par ligne de partage des eaux, qui fait partie d'une extension de la bibliothèque SMIL.

9.2.1 Gestion explicite du parallélisme sur MPPA

Dernier modèle de programmation offert pour développer des applications sur le MPPA, le modèle explicite propose de gérer finement le parallélisme intra-cluster, grâce à des interfaces standards, et les communications inter-clusters via une interface de programmation spécifique. Notre objectif est d'en faire une nouvelle cible logicielle de FREIA à destination du MPPA, en suivant la méthodologie détaillée au chapitre 6, à savoir :

1. implémenter les opérateurs de traitement d'images de FREIA dans ce modèle de programmation ;
2. développer un environnement d'exécution à même de transférer des images depuis et vers l'hôte ;
3. intégrer le tout dans FREIA ;
4. se servir de PIPS pour générer automatiquement du code optimisé.

Le but de cette nouvelle implémentation est d'utiliser au maximum le parallélisme offert par les 256 cœurs, par exemple, en divisant l'image, si elle est de taille suffisante, en tuiles qui seront traitées chacunes par un cluster de calcul. Dans ce cas, et contrairement à l'implémentation du chapitre 5, c'est un des clusters d'entrée/sortie qui exécute l'application et donc commande les autres clusters.

Une partie de ces travaux a déjà été réalisée et des pistes de réflexion ont été lancées quant au reste. Cette section expose les travaux déjà effectués.

Parallélisme intra-cluster

Au niveau d'un cluster de calcul, le processeur MPPA entre dans la catégorie des architectures à mémoire partagée. Ainsi que présenté au chapitre 5, plusieurs modèles de programmation existent pour ce type d'architecture et, notamment, POSIX Threads [66] et OpenMP [19], tous deux supportés par les outils de compilation à destination du MPPA. C'est avec ce dernier que nous avons décidé d'implémenter les opérateurs de traitement d'images de FREIA. En effet, la syntaxe à directives préprocesseurs d'OpenMP accélère de beaucoup le développement de ces opérateurs, la parallélisation effective étant laissée aux soins du compilateur.

Nous avons ainsi implémenté les opérateurs de FREIA en utilisant, comme dans le chapitre 6, des macro-définitions préprocesseur pour factoriser le code. Ces opérateurs sont ensuite compilés avec les outils de Kalray à destination des clusters de calcul du MPPA.

Communications inter-clusters

Afin de pouvoir transférer des informations et des commandes entre le cluster d'entrée/sortie maître et les clusters de calcul esclaves, nous avons réutilisé l'expérience acquise dans l'implémentation de l'environnement d'exécution pour SMIL du chapitre 5. Cette fois,

l'application sera exécutée par un des clusters d'entrée/sortie, qui se chargera de transférer des images entre les différents clusters et commandera l'exécution des noyaux de calcul sur ces derniers.

Notre environnement d'exécution sera, là encore, divisé en trois parties.

Le cluster d'entrée/sortie maître exécute l'application et se charge :
- de transférer les images entières depuis l'hôte vers la mémoire DDR attachée,
- de gérer le découpage de ces images en tuiles,
- de transférer les tuiles depuis et vers la mémoire des clusters de calcul et
- de transférer les commandes, afin d'exécuter les opérateurs de traitement d'images implémentés en OpenMP par les clusters de calcul sur les tuiles en mémoire.

L'hôte démarre le MPPA, lit et écrit les images locales via SMIL et les transfère vers le cluster d'entrée/sortie maître.

Les clusters de calcul reçoivent et renvoient les tuiles sur commande du cluster d'entrée/sortie et exécutent les différents opérateurs de traitement d'images.

Cet environnement d'exécution, déjà en partie fonctionnel, repose sur la bibliothèque de communication inter-clusters `libmppaipc` de Kalray. Toutefois, une seconde bibliothèque dédiée à la même tâche a été développée en 2016. Cette dernière, plus optimisée et à l'interface plus simple, s'appuie sur des transferts asynchrones, d'où son nom : `libmppa_async`. Nous proposons ainsi d'adapter notre environnement d'exécution avec les nouvelles routines proposées par cette bibliothèque.

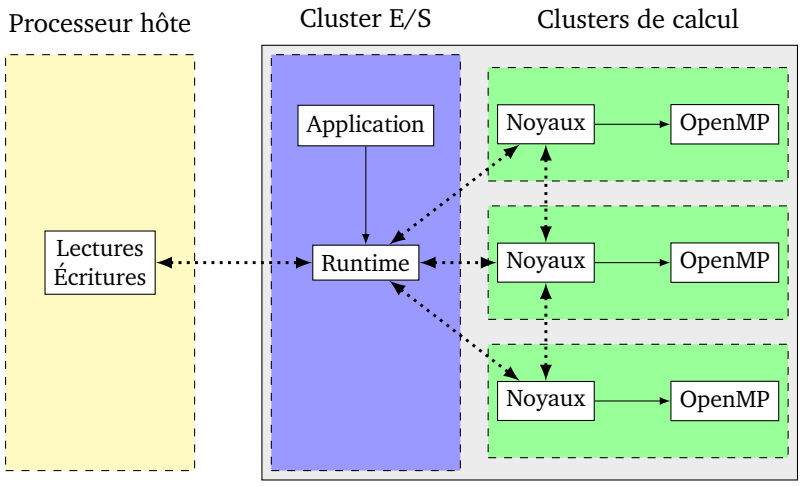


Figure 9.1 – Dernier modèle de programmation : OpenMP sur les clusters de calcul, tuilage des données depuis les clusters d'E/S, communications inter-clusters

Gestion des transferts de données

Au vu de la taille restreinte de la mémoire partagée des clusters de calcul, déterminer les dimensions des tuiles à communiquer est un problème critique. Différents opérateurs s'appliquent sur un nombre différent d'images, certaines images peuvent être consommées au cours des calculs, ce qui permet de réutiliser leur emplacement mémoire par la suite.

Dans le but de limiter au maximum les communications de données depuis le cluster d'entrée/sortie maître, les clusters de calcul exécutent des lots de commandes, qui s'appliquent sur plusieurs tuiles présentes en mémoire partagée. Nous avons ainsi entamé une réflexion sur un système qui, à partir de ce lot de commandes, effectue un pavage en deux dimensions des images en entrée, de sorte à ce que les tuiles nécessaires puissent cohabiter dans la mémoire partagée d'un cluster de calcul lors de l'exécution de ce lot.

Dans une première approche, les tailles des tuiles sont calculées en prenant en compte la superposition causée par une suite d'opérateurs morphologiques, qui sont des opérateurs au voisinage. Nous prévoyons, pour optimiser ces traitements, de réaliser des communications directement entre clusters de calcul, sans passer par le cluster d'entrée/sortie, afin de transmettre les bords des tuiles. Ces communications directes devrait permettre de contourner les accès à la mémoire DDR attachée, qui grèvent les performances du modèle OpenCL. Nous espérons, grâce à ce modèle, atteindre la performance crête du processeur MPPA.

Interfaçage avec FREIA et PIPS

À l'aide d'une interface de programmation bien pensée pour commander les clusters de calcul, nous pouvons faciliter, d'une part, l'intégration de cette nouvelle implémentation dans FREIA et, d'autre part, la génération de code spécifique par PIPS à partir d'applications FREIA.

Nous nous sommes ainsi inspirés de l'implémentation Terapix [20], qui fait entre autres usage d'un tableau de pointeurs vers les routines de traitement d'images, en coordination avec un type énuméré. Ceci peut également faciliter l'implémentation du générateur de code dans PIPS.

Nous avons réalisé une première intégration de cette implémentation spécifique dans FREIA et proposons de rapidement développer le système de compilation source-à-source associé, tout en entamant une réflexion quant aux optimisations pertinentes relatives à cette cible, à savoir, par exemple, l'optimisation des tailles de tuile et de la superposition pour trouver un compromis entre calculs et communications.

Récapitulatif et travaux futurs

Une partie majeure du travail d'implémentation a déjà été accomplie dans le but de réaliser cette nouvelle implémentation de FREIA à destination du MPPA. Un récapitulatif des travaux entrepris est ainsi disponible dans le tableau 9.1.

Tâche	Statut	Détails
Opérateurs de traitement d'images	Achevée	Réalisés en OpenMP
Environnement d'exécution	En cours	Tester avec `libmppa_async`
Intégration dans FREIA	En cours	À davantage optimiser
Génération automatique de code	À faire	Optimisations pertinentes ?

TABLE 9.1 – Statut de l'implémentation du modèle de programmation explicite dans FREIA

Les différentes tâches restant à accomplir consistent à
— tester le remplacement de la bibliothèque de communication `libmppaipc` par la plus récente `libmppa_async`,
— développer le générateur de code spécifique dans PIPS et
— améliorer les opérateurs aux voisinage, en communicant les bords de tuiles directement aux clusters qui en ont besoin.

9.2.2 Portage d'un algorithme de traitement d'images

Le projet FREIA repose sur la théorie de la morphologie mathématique et propose de construire des applications autour d'opérateurs issus de cette théorie. Ces opérateurs sont en général composés d'un ensemble restreint d'opérateurs de base. Optimiser ces derniers est donc critique afin d'obtenir une bonne performance.

Toutefois, la recherche évolue et de nouveaux algorithmes offrent de nouvelles propriétés aux opérateurs de traitement d'images de haut niveau. C'est le cas de la segmentation, dont un nouvel algorithme distribué est actuellement étudié et sera bientôt inclus dans la bibliothèque SMIL. Nous planifions de porter ce nouvel algorithme sur le processeur MPPA et d'étudier cette implémentation, par rapport à l'implémentation de référence destinées aux architectures à mémoire partagée.

L'opérateur de segmentation en morphologie mathématique

La segmentation est une opération classique du traitement d'images, qui consiste à réaliser une partition des pixels de celle-ci suivant des critères pré-établis. Généralement, elle est utilisée pour séparer, avant d'identifier, différents objets présents dans une image. La ligne de partage des eaux [14] est une méthode de segmentation issue de la morphologie mathématique. Elle consiste à « inonder » une image, considérée comme un relief topographique, et construire des barrages aux limites des bassins versants. Ce sont ces barrages qui formeront la segmentation.

La segmentation par ligne de partage des eaux s'effectue en trois étapes :

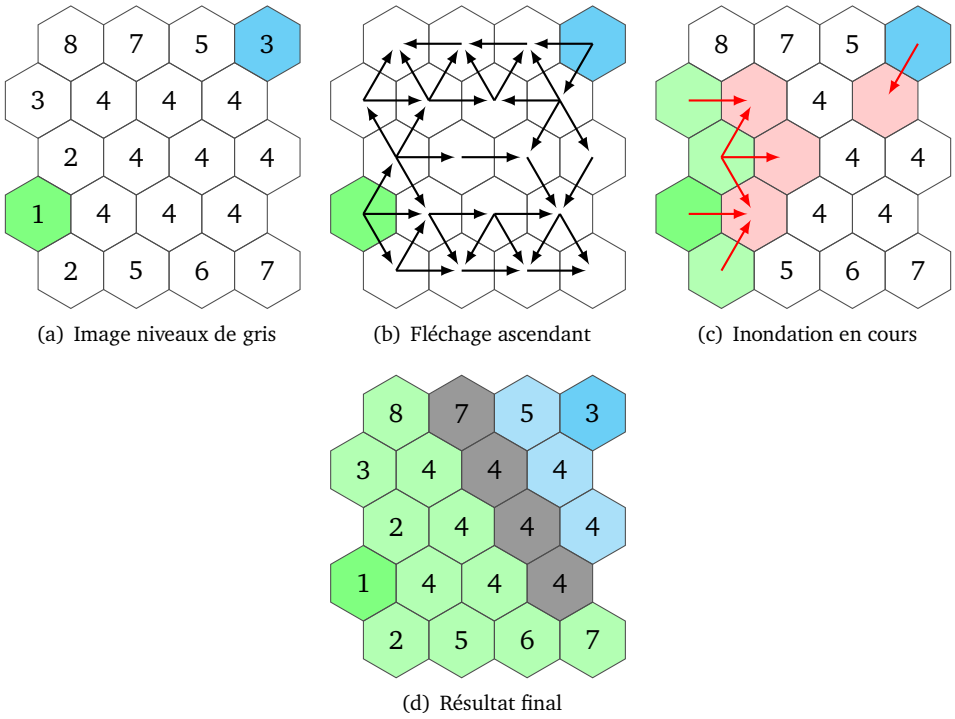
1. détermination des minima régionaux d'une image, ceux-ci formeront la base des bassins versants ;
2. labellisation des minima : on associe un label unique à chaque minima pour identifier chaque bassin versant ;
3. inondation à partir des minima et construction de la ligne de partage des eaux.

Première étape, la détermination des minima s'appuie, généralement, sur l'opérateur de reconstruction géodésique, dont nous avons déjà parlé dans le chapitre 6, ainsi que dans le chapitre 4. L'étape d'inondation est, quant à elle, critique en matière de performance. Dans SMIL, l'inondation, tout comme la reconstruction géodésique, fait appel à une file d'attente hiérarchique, afin de trier les pixels en attente d'être traités selon leurs valeurs. Cette approche est relativement efficace lorsque les images en entrées sont de taille importante mais la structure reste globale et, donc, difficile à paralléliser efficacement sur une architecture à mémoire distribuée comme le processeur MPPA. Il est certes possible de placer cette file d'attente dans la mémoire globale mais les communications nécessaires pour synchroniser cette structure de données entre tous les clusters de calcul augmenteront significativement les temps de calcul. Une nouvelle approche est donc nécessaire pour porter cet opérateur sur le MPPA.

Vers un algorithme de segmentation distribué

De récents travaux entamés au Centre de morphologie mathématique se sont concentrés sur un algorithme de segmentation par ligne de partage des eaux distribué [23]. Ce dernier fait appel à des outils développés dans les années 1990, comme les graphes de fléchage [13], que les avancées dans les capacités de calcul des processeurs ont récemment remis au goût du jour.

Un graphe de fléchage caractérise une relation d'ordre entre un pixel et ses voisins. Dans un graphe d'ascendance, un pixel va flécher un de ses voisins si, et seulement si, le voisin a



(a) Image niveaux de gris (b) Fléchage ascendant (c) Inondation en cours

(d) Résultat final

FIGURE 9.2 – Ligne de partage des eaux par graphes de fléchage (avec l'aimable autorisation de Théodore Chabardès)

une valeur supérieure au pixel courant. Le graphe de descendance est défini similairement, en prenant l'inégalité inverse.

Pour une image bidimensionnelle, un pixel a au maximum huit voisins. Un graphe de fléchage ne codant qu'une relation binaire entre un pixel et ses voisins, 8 bits par pixel suffisent donc pour représenter un tel graphe de fléchage. De cette manière, les graphes de fléchage peuvent se manipuler comme des structures de données images. Parcourir et opérer sur de tels graphes revient à effectuer des opérations bit-à-bit sur les pixels d'une image, ce qui est facilité par les capacités des processeurs modernes pour la vectorisation des calculs.

Ces graphes de fléchages facilitent, d'une part, la détermination des minima d'une image — des pixels qui ne sont fléchés par aucun voisins dans le graphe d'ascendance — mais permettent, d'autre part, d'accélérer l'étape d'inondation, qui s'effectuera en suivant le graphe d'ascendance, à partir des minima. Les premiers résultats confirment l'intérêt porté à ce nouvel algorithme, qui se montre d'un ordre de grandeur plus rapide que la méthode de référence, qui s'appuie sur ces fameuses files d'attente hiérarchiques. La figure 9.2 illustre ainsi les différentes étapes pour obtenir la segmentation d'une image en niveaux de gris, en suivant le graphe de fléchage ascendant.

À venir : implémentations sur le MPPA

Nous prévoyons de porter ce nouvel algorithme sur l'architecture distribuée du MPPA. Une interface vers les fonctions de fléchage — afin d'obtenir les graphes correspondants à partir d'une image —, de détermination des minima et de labellisation a été inclue dans

FREIA et liée aux fonctions correspondantes de SMIL. De plus, une première implémentation des opérateurs de fléchage a été réalisée et intégrée dans l'implémentation FREIA décrite en sous-section 9.2.1. L'implémentation des minima et de la labellisation suivra bientôt et la distribution de l'inondation peu après.

9.3 Traitement d'images et architectures manycore

Au vu des innovations technologiques actuelles et prochaines, il ne fait aucun doute que le traitement d'images, notamment au travers de la vision par ordinateur, est un domaine scientifique et industriel clef. La multiplication des capteurs optiques dans les objets quotidiens, avec d'abord les smartphones, puis les tablettes et maintenant l'Internet des objets, les drones et les véhicules automatiques, ouvre la voie vers des applications toujours plus novatrices, qui lorgnent désormais vers la réalité virtuelle ou augmentée. Pouvoir développer rapidement de telles applications, à l'aide d'interfaces de programmation de haut niveau, ainsi que proposé dans cette thèse, et ainsi cibler différentes architectures matérielles sans efforts, tout en garantissant de bonnes performances en temps de réponse et en autonomie, est un objectif primordial et un atout essentiel pour percer dans ce domaine.

Les processeurs manycore, en offrant plusieurs niveaux de parallélisme et une consommation énergétique faible, sont particulièrement adaptés à ce contexte, malgré la difficulté de les programmer. L'utilisation, encore confidentielle, de ces nouvelles architectures est vouée à se démocratiser et, sans doute, à concurrencer les processeurs graphiques dans les appareils embarqués. Développer des outils intégrés, de qualité industrielle et ciblant ces architectures s'avère nécessaire. Réécrire sans cesse les bibliothèques logicielles, pour cibler chaque nouvelle architecture, est un processus long et coûteux, qui ne permet que difficilement de tirer le maximum de performances d'une architecture donnée.

Des approches mixtes vont émerger, fondées sur des bibliothèques d'opérateurs de bas niveau et des compilateurs capables de réaliser des optimisations liées au domaines, par exemple en fusionnant certains opérateurs. L'utilisation de langages spécifiques à un domaine, en tant que porte d'entrée pour le développement d'applications, va également se développer. Cependant, la multiplication de ces langages risque, en proposant à chaque fois une nouvelle syntaxe à maîtriser, de diluer les efforts et de cantonner chaque langage spécifique à une niche. Pour éviter cet écueil, la normalisation de langages de référence peut fournir une base unique, sur laquelle différents compilateurs peuvent rivaliser avec leurs optimisations. Les langages spécifiques embarqués permettent de réutiliser la syntaxe d'un langage hôte et de combiner les langages spécifiques partageant le même hôte. La création de ponts vers des représentations standard pour les structures de données de haut niveau, comme les tableaux Numpy [93] en Python, sera nécessaire afin d'offrir une bonne interopérabilité entre ces langages.

Références

[1] M. Abadi et al. « TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems ». In : *ArXiv e-prints* (mar. 2016).

[2] Amit Agarwal et al. *An Introduction to Computational Networks and the Computational Network Toolkit*. Rapp. tech. MSR-TR-2014-112. Août 2014. url : http://research.microsoft.com/apps/pubs/default.aspx?id=226641.

[3] Altera. *Altera SDK for OpenCL*. url : https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[4] Medhi Amini. « Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators ». Thèse de doct. MINES ParisTech, déc. 2012.

[5] *An Introduction to the Autotools*. url : https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html#Autotools-Introduction.

[6] ARM. *big.LITTLE Technology*. url : https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[7] Pascal Aubry et al. « Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. » In : *ICCS*. Sous la dir. de Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra et Peter M. A. Sloot. T. 18. Procedia Computer Science. Elsevier, 2013, p. 1624–1633.

[8] *Auto-vectorization in GCC*. url : https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[9] *Baron : a Full Syntax Tree library for Python*. url : https://github.com/PyCQA/baron.

[10] Frédéric Bastien et al. *Theano : new features and speed improvements*. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. 2012.

[11] S. Behnel et al. « Cython : The Best of Both Worlds ». In : *Computing in Science Engineering* 13.2 (mar. 2011), p. 31–39.

[12] James Bergstra et al. « Theano : a CPU and GPU Math Expression Compiler ». In : *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Austin, TX, juin 2010.

[13] Serge Beucher. « Segmentation d'images et morphologie mathématique ». Thèse de doct. École Nationale Supérieure des Mines de Paris, juin 1990.

[14] Serge BEUCHER et Fernand MEYER. « The morphological approach to segmentation : the watershed transformation ». In : *OPTICAL ENGINEERING-NEW YORK-MARCEL DEKKER INCORPORATED-* 34 (1992), p. 433–433.

[15] Michel BILODEAU et al. *FREIA : FRamework for Embedded Image Applications*. French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne. 2008. URL : freia.enstb.org.

[16] G. BILSEN, M. ENGELS, R. LAUWEREINS et J. PEPERSTRAETE. « Cycle-static dataflow ». In : *IEEE Transactions on Signal Processing* 44.2 (fév. 1996), p. 397–408.

[17] *Bitcoin est un réseau de paiement novateur et une nouvelle forme d'argent*. URL : https://bitcoin.org/.

[18] OpenMP Architecture Review BOARD. *OpenMP Application Programming Interface*. Juil. 2013. URL : http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[19] OpenMP Architecture Review BOARD. *The OpenMP API specification for parallel programming*. URL : http://openmp.org/wp/.

[20] Philippe BONNOT et al. « Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA ». In : *Design Automation and Test in Europe*. IEEE, déc. 2008, p. 610–615.

[21] George BOSILCA et al. « DAGuE : A generic distributed DAG engine for High Performance Computing ». en. In : *Parallel Computing* 38.1-2 (jan. 2012), p. 37–51. URL : http://linkinghub.elsevier.com/retrieve/pii/S0167819111001347 (visité le 28/07/2014).

[22] Lars BUITINCK et al. « API design for machine learning software : experiences from the scikit-learn project ». In : *ECML PKDD Workshop : Languages for Data Mining and Machine Learning*. 2013, p. 108–122.

[23] Theodore CHABARDES, Petr DOKLADAL, Matthieu FAESSEL et Michel BILODEAU. « A parallel, O(n), algorithm for unbiased, thin watershed ». working paper or preprint. Fév. 2016. URL : https://hal.archives-ouvertes.fr/hal-01266889.

[24] Craig CHAMBERS et al. « FlumeJava : easy, efficient data-parallel pipelines ». en. In : ACM Press, 2010, p. 363. URL : http://portal.acm.org/citation.cfm?doid=1806596.1806638 (visité le 28/07/2014).

[25] *clang-format :A tool to format C/C++/Java/JavaScript/Objective-C/Protobuf code*. URL : http://clang.llvm.org/docs/ClangFormat.html.

[26] Christophe CLIENTI. *Fulguro image processing library*. Source Forge. 2008.

[27] Christophe CLIENTI, Serge BEUCHER et Michel BILODEAU. « A System On Chip Dedicated To Pipeline Neighborhood Processing For Mathematical Morphology ». In : *EUSIPCO : European Signal Processing Conference*. Août 2008.

[28] *CMake : Build, Test and Package Your Software*. URL : https://cmake.org/.

[29] Fabien COELHO et François IRIGOIN. « API compilation for image hardware accelerators ». In : *ACM Transactions on Architecture and Code Optimization* 9.4 (jan. 2013), p. 1–25. URL : http://dl.acm.org/citation.cfm?doid=2400682.2400708 (visité le 12/12/2013).

[30] Ronan COLLOBERT, Samy BENGIO et Johnny MARITHOZ. *Torch : A Modular Machine Learning Software Library*. 2002. URL : http://torch.ch/.

[31] Microsoft Corp. *C++ Accelerated Massive Parallelism*. URL : https://msdn.microsoft.com/en-us/en-en/library/hh265136.aspx.

[32] Microsoft Corp. *DirectX Graphics*. URL : https://msdn.microsoft.com/en-us/library/windows/desktop/hh309467.aspx.

[33] Nvidia Corp. *CUDA 7 Streams Simplify Concurrency*. URL : https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[34] Nvidia Corp. *NVIDIA CUDA Basic Linear Algebra Subroutines library*. URL : https://developer.nvidia.com/cublas.

[35] Nvidia Corp. *NVIDIA CUDA Deep Neural Network library*. URL : https://developer.nvidia.com/cudnn.

[36] Nvidia Corp. *NVIDIA CUDA Fast Fourier Transform library*. URL : https://developer.nvidia.com/cufft.

[37] Nvidia Corp. *NVIDIA CUDA Toolkit*. URL : https://developer.nvidia.com/cuda-toolkit.

[38] Nvidia Corp. *NVIDIA GPUs - The Engine of Deep Learning*. URL : https://developer.nvidia.com/deep-learning.

[39] Nvidia Corp. *NVIDIA Thrust*. URL : https://developer.nvidia.com/thrust.

[40] *CPython Global Interpreter Lock*. URL : https://wiki.python.org/moin/GlobalInterpreterLock.

[41] CRI, MINES ParisTech. *PIPS*. Open Source Research Compiler, under GPLv3. 1989. URL : pips4u.org.

[42] *Cython : C-Extensions for Python*. URL : http://cython.org/.

[43] B. D. de Dinechin, D. van Amstel, M. Poulhiès et G. Lager. « Time-critical computing on a single-chip massively parallel processor ». In : *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, p. 1–6.

[44] Edward Dougherty. *An introduction to morphological image processing*. Bellingham, Wash., USA : SPIE Optical Engineering Press, 1992.

[45] *Doxygen : generate documentation from source code*. URL : http://www.doxygen.org/.

[46] Matthieu Faessel. *SMIL : Simple (but efficient) Morphological Image Library*. 2011. URL : http://smil.cmm.mines-paristech.fr/doc/index.html.

[47] Matthieu Faessel et Michel Bilodeau. « SMIL : Simple Morphological Image Library ». In : *Séminaire Performance et Généricité, LRDE*. Villejuif, France, mar. 2013. URL : https://hal-mines-paristech.archives-ouvertes.fr/hal-00836117.

[48] Facebook AI Research (FAIR). *Learning to Segment*. URL : https://research.facebook.com/blog/learning-to-segment/.

[49] The MPI Forum. *The Message Passing Interface*. URL : http://www.mpi-forum.org/.

[50] Eclipse Foundation. *Eclipse Integrate Development Environment*. URL : https://eclipse.org/.

[51] *GEGL : GEneric Graphics Library*. URL : http://www.gegl.org/.

[52] *Git : free and open source distributed version control system*. URL : https://git-scm.com/.

[53] *GNU Make*. URL : https://www.gnu.org/software/make/.

[54] Michael I. GORDON. « Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures ». Thèse de doct. MIT.

[55] Michael I. GORDON, William THIES et Saman AMARASINGHE. « Exploiting coarse-grained task, data, and pipeline parallelism in stream programs ». en. In : *ACM SIGPLAN Notices* 41.11 (oct. 2006), p. 151. URL : http://portal.acm.org/citation.cfm?doid=1168918.1168877 (visité le 05/08/2014).

[56] Thierry GOUBIER, Renaud SIRDEY, Stéphane LOUISE et Vincent DAVID. « C : A Programming Model and Language for Embedded Manycores ». In : 2011.

[57] Khronos GROUP. *Khronos Releases OpenCL 2.2 Provisional Specification with OpenCL C++ Kernel Language for Parallel Programming*. Avr. 2016. URL : https://www.khronos.org/news/press/khronos-releases-opencl-2.2-provisional-spec-opencl-c-kernel-language.

[58] Khronos GROUP. *OpenCL : A State of the Union*. Avr. 2016. URL : https://www.khronos.org/assets/uploads/developers/library/2016-iwocl/IWOCL-Keynote_Apr16.pdf.

[59] Khronos GROUP. *OpenCL Computing Language v1.0*. Déc. 2008.

[60] Khronos GROUP. *OpenCL : The open standard for parallel programming of heterogeneous systems*. URL : https://www.khronos.org/opencl/.

[61] Khronos GROUP. *OpenGL : The Industry's Foundation for High Performance Graphics*. URL : https://www.khronos.org/opengl/.

[62] Khronos GROUP. *OpenVX : Portable, Power-efficient Vision Processing*. URL : https://www.khronos.org/openvx/.

[63] Khronos GROUP. *SPIR-V : The first open standard intermediate language for parallel compute and graphics*. URL : https://www.khronos.org/spir/.

[64] Khronos GROUP. *SYCL : C++ Single-source Heterogeneous Programming for OpenCL*. URL : https://www.khronos.org/sycl/.

[65] Khronos GROUP. *Vulkan - Industry Forged*. URL : https://www.khronos.org/vulkan/.

[66] Open GROUP. *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*. URL : http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html.

[67] Serge GUELTON et al. « Pythran : enabling static optimization of scientific Python programs ». In : *Computational Science & Discovery* (2015).

[68] Rachid HABEL. « Programmation haute performance pour architectures hybrides ». Thèse de doct. MINES ParisTech, nov. 2014.

[69] Rachid HABEL, Frédérique SILBER-CHAUSSUMIER, François IRIGOIN, Elisabeth BRUNET et François TRAHAY. « Combining Data and Computation Distribution Directives for Hybrid Parallel Programming : A Transformation System ». In : *International Journal of Parallel Programming* 44.6 (2016), p. 1268–1295. URL : http://dx.doi.org/10.1007/s10766-016-0428-3.

[70] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. « The synchronous dataflow programming language LUSTRE ». In : *Proceedings of the IEEE*. 1991, p. 1305–1320.

[71] *ImageMagick : Convert, Edit, Or Compose Bitmap Images*. URL : https://www.imagemagick.org/script/index.php.

[72] Intel. *Intel® AVX-512 instructions*. URL : https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[73] Intel. *Intel® SDK for OpenCL™ Applications*. URL : https://software.intel.com/en-us/intel-opencl.

[74] Intel. *Intel Xeon Phi*. URL : http://www.intel.fr/content/www/fr/fr/processors/xeon/xeon-phi-detail.html.

[75] François Irigoin, Pierre Jouvelot et Rémi Triolet. « Semantical interprocedural parallelization : an overview of the PIPS project ». en. In : *Proceedings of ICS 1991*. ACM Press, 1991, p. 244–251. URL : http://portal.acm.org/citation.cfm?doid=109025.109086 (visité le 21/05/2014).

[76] *Is C/C++ really faster than Python ?* URL : https://news.ycombinator.com/item?id=9753366.

[77] *Is Python faster and lighter than C++ ?* URL : https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c/.

[78] Pekka Jääskeläinen et al. « pocl : A Performance-Portable OpenCL Implementation ». In : *International Journal of Parallel Programming* 43.5 (2015), p. 752–785. URL : http://dx.doi.org/10.1007/s10766-014-0320-y.

[79] Yangqing Jia et al. « Caffe : Convolutional Architecture for Fast Feature Embedding ». In : *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA : ACM, 2014, p. 675–678. URL : http://doi.acm.org/10.1145/2647868.2654889.

[80] Gilles Kahn. « The semantics of a simple language for parallel programming ». In : 1974, p. 5.

[81] Andreas Klöckner. *PyOpenCL : OpenCL integration for Python*. URL : https://mathema.tician.de/software/pyopencl/.

[82] Amazon Labs. *Amazon DSSTNE : Deep Scalable Sparse Tensor Network Engine*. URL : https://github.com/amznlabs/amazon-dsstne.

[83] Siu Kwan Lam, Antoine Pitrou et Stanley Seibert. « Numba : A LLVM-based Python JIT Compiler ». In : *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas : ACM, 2015, 7 :1–7 :6. URL : http://doi.acm.org/10.1145/2833157.2833162.

[84] 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.

[85] P. Le Guernic, A. Benveniste, P. Bournai et T. Gautier. « Signal–A data flow-oriented language for signal processing ». In : *Acoustics, Speech and Signal Processing, IEEE Transactions on* 34.2 (avr. 1986), p. 362–374.

[86] Edward Ashford Lee et David G. Messerschmitt. « Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing ». In : *IEEE Trans. Comput.* 36.1 (jan. 1987), p. 24–35.

[87] Nelson LOSSING, Corinne ANCOURT et Francois IRIGOIN. « Automatic Code Generation of Distributed Parallel Tasks ». In : *Computational Science and Engineering (CSE), 2016 IEEE 19th International Conference on*. 2016.

[88] MARTIN ABADI et al. *TensorFlow : Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL : http://tensorflow.org/.

[89] MICROSOFT. *Using extern to Specify Linkage*. URL : https://msdn.microsoft.com/en-us/library/0603949d.aspx.

[90] Gordon E. MOORE. *Cramming more components onto integrated circuits*. McGraw-Hill New York, NY, USA, 1965. URL : http://web.eng.fiu.edu/npala/EEE6397ex/Gordon_Moore_1965_Article.pdf.

[91] Ravi Teja MULLAPUDI, Vinay VASISTA et Uday BONDHUGULA. « PolyMage : Automatic Optimization for Image Processing Pipelines ». en. In : ACM Press, 2015, p. 429–443. URL : http://dl.acm.org/citation.cfm?doid=2694344.2694364 (visité le 08/04/2015).

[92] Praveen K. MURTHY et Edward A. LEE. « Multidimensional Synchronous Dataflow ». In : *IEEE Transactions on Signal Processing* 50 (2002), p. 3306–3309.

[93] *NumPy : scientific computing with Python*. URL : http://www.numpy.org/.

[94] *OpenACC : Directives for Accelerators*. URL : http://www.openacc.org/.

[95] *OpenCV : Open Source Computer Vision*. URL : http://opencv.org/.

[96] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface, Version 3.0*. Mai 2008.

[97] *Partioned Global Address Space*. URL : http://www.pgas.org/.

[98] F. PEDREGOSA et al. « Scikit-learn : Machine Learning in Python ». In : *Journal of Machine Learning Research* 12 (2011), p. 2825–2830.

[99] Laurent PEUCH. *RedBaron, une approche bottom-up au refactoring en Python*. Oct. 2014.

[100] Antoniu POP. « Leveraging Streaming for Deterministic Parallelization - an Integrated Language, Compiler and Runtime Approach - ». Thèse de doct. MINES ParisTech, sept. 2011.

[101] Antoniu POP et Albert COHEN. « OpenStream : Expressiveness and Data-flow Compilation of OpenMP Streaming Programs ». In : *ACM Trans. Archit. Code Optim.* 9.4 (jan. 2013), 53 :1–53 :25. URL : http://doi.acm.org/10.1145/2400682.2400712.

[102] *Portable Computing Language*. URL : http://portablecl.org/.

[103] *Python ast — Abstract Syntax Trees*. URL : https://docs.python.org/3/library/ast.html.

[104] *Python inspect — Inspect live objects*. URL : https://docs.python.org/3/library/inspect.html.

[105] Jonathan RAGAN-KELLEY et al. « Halide : A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines ». In : *PLDI 2013* (juin 2013), p. 12.

[106] *Redbaron : Bottom-up approach to refactoring in python*. URL : http://github.com/PyCQA/redbaron.

[107] Alex RUBINSTEYN, Eric HIELSCHER, Nathaniel WEINMAN et Dennis SHASHA. « Parakeet : A Just-In-Time Parallel Accelerator for Python ». In : Berkeley, CA : USENIX, 2012.

[108] Jean SERRA. *Image analysis and mathematical morphology*. London New York : Academic Press, 1982.

[109] Richard M. STALLMAN et GCC DEVELOPERCOMMUNITY. *Using The Gnu Compiler Collection*. 2016.

[110] *StreamIt Cookbook*. Sept. 2006.

[111] Herb SUTTER. *Welcome to the Jungle*. 2011. URL : http://herbsutter.com/welcome-to-the-jungle/.

[112] *Swig : Simplified Wrapper and Interface Generator*. URL : http://www.swig.org/.

[113] Cisco SYSTEMS et Cortina SYSTEMS. *Interlaken Technology : New-Generation Packet Interconnect Protocol*. URL : https://www.cortina-systems.com/images/documents/400023_Interlaken_Technology_White_Paper.pdf.

[114] *The Meson Build System*. URL : http://mesonbuild.com/.

[115] *The StreamIt Language*. 2002. URL : http://www.cag.lcs.mit.edu/streamit/.

[116] William THIES, Michal KARCZMAREK et Saman P. AMARASINGHE. « StreamIt : A Language for Streaming Applications ». In : *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. London, UK, UK : Springer-Verlag, 2002, p. 179–196. URL : http://dl.acm.org/citation.cfm?id=647478.727935.

[117] *TOP 500*. URL : https://www.top500.org/.

[118] XILINX. *SDAccel Development Environment*. URL : http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

Bibliographie personnelle

[119] Pierre GUILLOU. *Compilation d'applications de traitement d'images sur architecture MPPA-Manycore*. Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014). Poster. Avr. 2014. URL : https://hal-mines-paristech.archives-ouvertes.fr/hal-01096993.

[120] Pierre Guillou. *Compiling Image Processing Applications for Many-Core Accelerators*. ACACES Summer School: Eleventh International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. Poster. July 2015. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01254412.

[121] Pierre Guillou. smilc*: A C wrapper for SMIL*. URL: https://scm.cri.ensmp.fr/git/smilc.git.

[122] Pierre Guillou, Fabien Coelho, and François Irigoin. "Automatic Streamization of Image Processing Applications". In: *The 27th International Workshop on Languages and Compilers for Parallel Computing*. Parasol. Hillsboro, United States, Sept. 2014. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01096735.

[123] Pierre Guillou, Benoît Pin, Fabien Coelho, and François Irigoin. "A Dynamic to Static DSL Compiler for Image Processing Applications". In: *19th Workshop on Compilers for Parallel Computing*. Valladolid, Spain, July 2016. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01352808.

[124] Nelson Lossing, Pierre Guillou, Mehdi Amini, and Francois Irigoin. "From Data to Effects Dependence Graphs: Source-to-Source Transformations for C". In: *18th International Workshop on Compilers for Parallel Computing (CPC 2015)* (Jan. 2015).

[125] Nelson Lossing, Pierre Guillou, and François Irigoin. "Effects Dependence Graph: A Key Data Concept for C Source-to-Source Compilers". In: *2016 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (Oct. 2016).

[126] Benoît Pin, Pierre Guillou, and Fabien Coelho. smiltofreia*: A SMIL Python to FREIA C compiler*. URL: https://scm.cri.ensmp.fr/git/smilpyc.git.

Index

Résumé

Nous assistons à une explosion du nombre d'appareils mobiles équipés de capteurs optiques : *smartphones*, tablettes, drones… préfigurent un Internet des objets imminent. De nouvelles applications de traitement d'images (filtres, compression, réalité augmentée) exploitent ces capteurs mais doivent répondre à des contraintes fortes de vitesse et d'efficacité énergétique. Les architectures modernes — processeurs manycore, GPUs,… — offrent un potentiel de performance, avec cependant une hausse sensible de la complexité de programmation.

L'ambition de cette thèse est de vérifier l'adéquation entre le domaine du traitement d'images et ces architectures modernes : concilier programmabilité, portabilité et performance reste encore aujourd'hui un défi. Le domaine du traitement d'images présente un fort parallélisme intrinsèque, qui peut potentiellement être exploité par les différents niveaux de parallélisme offerts par les architectures actuelles. Nous nous focalisons ici sur le domaine du traitement d'images par morphologie mathématique, et validons notre approche avec l'architecture manycore du processeur MPPA de la société Kalray.

Nous prouvons d'abord la faisabilité de chaînes de compilation intégrées, composées de compilateurs, bibliothèques et d'environnements d'exécution, qui à partir de langages de haut niveau tirent parti de différents accélérateurs matériels. Nous nous concentrons plus particulièrement sur les processeurs manycore, suivant les différents modèles de programmation : OpenMP ; langage flot de données ; OpenCL ; passage de messages. Trois chaînes de compilation sur quatre ont été réalisées, et sont accessibles à des applications écrites dans des langages spécifiques au domaine du traitement d'images intégrés à Python ou C. Elles améliorent grandement la portabilité de ces applications, désormais exécutables sur un plus large panel d'architectures cibles.

Ces chaînes de compilation nous ont ensuite permis de réaliser des expériences comparatives sur un jeu de sept applications de traitement d'images. Nous montrons que le processeur MPPA est en moyenne plus efficace énergétiquement qu'un ensemble d'accélérateurs matériels concurrents, et ceci particulièrement avec le modèle de programmation flot de données. Nous montrons que la compilation d'un langage spécifique intégré à Python vers un langage spécifique intégré à C permet d'augmenter la portabilité et d'améliorer les performances des applications écrites en Python.

Nos chaînes de compilation forment enfin un environnement logiciel complet dédié au développement d'applications de traitement d'images par morphologie mathématique, capable de cibler efficacement différentes architectures matérielles, dont le processeur MPPA, et proposant des interfaces dans des langages de haut niveau.

Mots Clés

Langages spécifiques à un domaine, Compilation, Processeurs manycore, Traitement d'image, Modèles de programmation

Abstract

Many mobile devices now integrate optic sensors; smartphones, tablets, drones… are foreshadowing an impending Internet of Things (IoT). New image processing applications (filters, compression, augmented reality) are taking advantage of these sensors under strong constraints of speed and energy efficiency. Modern architectures, such as manycore processors or GPUs, offer good performance, but are hard to program.

This thesis aims at checking the adequacy between the image processing domain and these modern architectures: conciliating programmability, portability and performance is still a challenge today. Typical image processing applications feature strong, inherent parallelism, which can potentially be exploited by the various levels of hardware parallelism inside current architectures. We focus here on image processing based on mathematical morphology, and validate our approach using the manycore architecture of the Kalray MPPA processor.

We first prove that integrated compilation chains, composed of compilers, libraries and run-time systems, allow to take advantage of various hardware accelerators from high-level languages. We especially focus on manycore processors, through various programming models: OpenMP, data-flow language, OpenCL, and message passing. Three out of four compilation chains have been developed, and are available to applications written in domain-specific languages (DSL) embedded in C or Python. They greatly improve the portability of applications, which can now be executed on a large panel of target architectures.

Then, these compilation chains have allowed us to perform comparative experiments on a set of seven image processing applications. We show that the MPPA processor is on average more energy-efficient than competing hardware accelerators, especially with the data-flow programming model. We show that compiling a DSL embedded in Python to a DSL embedded in C increases both the portability and the performance of Python-written applications. Thus, our compilation chains form a complete software environment dedicated to image processing application development. This environment is able to efficiently target several hardware architectures, among them the MPPA processor, and offers interfaces in high-level languages.

Keywords

Domain Specific Languages, Compilation, Manycore Processors, Image Processing, Programming Models