Méthodologie de placement d'algorithmes de traitement d'images sur architecture massivement parallèle

Florian Gouin

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à MINES ParisTech

Méthodologie de placement d'algorithmes de traitement d'images sur architecture massivement parallèle

Soutenue par
Florian GOUIN
Le 5 juillet 2019

École doctorale n°432
Ingénierie des Systèmes, Matériaux, Mécanique, Énergétique

Spécialité
Informatique temps-réel, robotique et automatique

Composition du jury :

Frédéric Magoulès Professeur, CentraleSupélec	*Président*
Lionel Lacassagne Professeur, Sorbonne Université	*Rapporteur*
David Defour Maître de conférence, Université de Perpignan	*Rapporteur*
Christophe Guettier Ingénieur, SAFRAN	*Examinateur*
François Irigoin Directeur de recherche, MINES ParisTech	*Directeur de thèse*
Corinne Ancourt Maître de recherche, MINES ParisTech	*Co-directrice de thèse*

MINES ParisTech | PSL

Pour mes enfants,
Raphaël *et* ***Alice***
qui ont partagé leur papa
avec l'écriture de ce manuscrit.

Remerciements

Je voudrais exprimer toute ma gratitude envers François Irigoin, mon directeur de thèse et directeur du Centre de Recherche en Informatique (CRI) de Fontainebleau, pour avoir accepté d'encadrer cette thèse ; Corinne Ancourt, ma co-directrice de thèse, pour la qualité des échanges techniques mais aussi la bienveillance qu'elle a su témoigner à mon égart au cours de ces années de thèse ; ainsi que Christophe Guettier et Marc Bousquet, à qui je souhaite rendre hommage, qui ont tous deux cru en mes capacités et m'ont apporté les moyens de réaliser cette thèse.

Je voudrais aussi témoigner ma reconnaissance envers mes rapporteurs, Lionnel Lacassagne et David Defour, pour toute l'attention qu'ils ont donnée à mes travaux de thèse ainsi que la qualité de leurs retours; Frédéric Magoules pour l'honneur qu'il m'a fait de présider ma défense de thèse, ainsi que François Courteille de chez Nvidia pour avoir accepté d'être membre de mon jury.

J'aimerais aussi, naturellement, remercier l'équipe du CRI de Fontainebleau ; Pierre Jouvelot, Fabien Coelho, Claude Tadonki, Olivier Hermant, Laurent Daverio, Emilio Gallego ou encore ma "colocataire de bureau", Claire Medrala, dont j'ai eu le plaisir d'apprécier, à de nombreuses reprises leur aide spontanée et leur bienveillance. Je souhaiterais aussi saluer amicalement les autres membres du CRI, anciens comme nouveaux, dont Pierre Guillou, Nelson Lossing, Vivien Maisonneuve, Olfa Haggui, Luc Perera, Bruno Sguerra, Lucas Sguerra ou encore Pierre Wargnier et remercier en particulier Benoît Pin, Adilla Susungi, Patryk Kiepas et Monika Rakoczy, pour toute l'amitié qu'ils m'ont témoignée et les moments forts que nous avons vécus ensemble.

J'aimerais aussi remercier mes collègues de Safran, tels que Philippe Foubert, Jean Marie Courteille, Philippe Craeye ou encore Pascal Gaden pour l'intérêt et l'attention qu'ils ont pu porter à mes travaux de thèse.

Enfin, je ne remercierai jamais assez mes parents pour toute l'aide, la motivation et la foi inébranlable en mes capacités qu'ils m'ont continuellement apportées. Je remercie aussi tendrement ma femme, Perrine, pour ce "travail d'équipe" que nous avons mené ensemble ainsi que mes enfants, Raphaël et Alice, qui je l'espère, seront eux aussi émerveillés par les sciences.

Abstract

In industries, the curse of image sensors for higher definitions increases the amount of data to be processed in the image processing domain. The concerned algorithms, applied to embedded solutions, also have to frequently accept real-time constraints. So, the main issues are to moderate power consumption, to attain high performance computings and high memory bandwidth for data delivery.

The massively parallel conception of Graphics Processing Unit (GPU)s is especially well adapted for this kind of tasks. However, this achitecture is complex to handle. Some reasons are its multiple memory and computation hierachical levels or the usage of this accelerator inside a global heterogeneous architecture. Therefore, mapping algorithms on GPUs, while exploiting high performance capacities of this architecture, aren't trivial operations.

In this thesis, we have developped a mapping methodology for sequential algorithms and designed it for GPUs. This methodology is made up of code analysis phases, mapping criteria verifications, code transformations and a final code generation phase. Part of the defined mapping criteria has been designed to assure the mapping legality, by considering GPU hardware specifities, whereas the other part are used to improve runtimes. In addition, we have studied GPU memories performances and the capacity of GPU to efficiently support coarse grain parallellism. This complementary work is a foundation for further improvments of GPU resources exploitation inside this mapping methodology.

Last, the experimental results have revealed the functional reliability of the codes mapped on GPU and a speedup on the runtime of many *C* and *C++* image processing applications used in industry.

Résumé

Dans le secteur industriel, la course à l'amélioration des définitions des capteurs vidéos se répercute directement dans le domaine du traitement d'images par une augmentation des quantités de données à traiter. Dans le cadre de l'embarqué, les mêmes algorithmes ont fréquemment pour contrainte supplémentaire de devoir supporter le temps réel. L'enjeu est alors de trouver une solution présentant une consommation énergétique modérée, une puissance calculatoire soutenue et une bande passante élevée pour l'acheminement des données.

Le GPU est une architecture adaptée pour ce genre de tâches notamment grâce à sa conception basée sur le parallélisme massif. Cependant, le fait qu'un accélérateur tel que le GPU prenne place dans une architecture globale hétérogène, ou encore ait de multiples niveaux hiérarchiques, complexifient sa mise en œuvre. Ainsi, les transformations de code visant à placer un algorithme sur GPU tout en optimisant l'exploitation des capacités de ce dernier, ne sont pas des opérations triviales.

Dans le cadre de cette thèse, nous avons développé une méthodologie permettant de porter des algorithmes sur GPU. Cette méthodologie est guidée par un ensemble de critères de transformations de programme. Certains d'entre-eux sont définis afin d'assurer la légalité du portage, tandis que d'autres sont utilisés pour améliorer les temps d'exécution sur cette architecture. En complément, nous avons étudié les performances des différentes mémoires ainsi que la gestion du parallélisme gros grain sur les architectures GPU Nvidia. Ces travaux sont une étape préalable à l'ajout de nouveaux critères dans notre méthodologie, visant à maximiser l'exploitation des capacités de ces GPUs.

Les résultats expérimentaux obtenus montrent non seulement la fiabilité du placement mais aussi une accélération des temps d'exécution sur plusieurs applications industrielles de traitement d'images écrites en langage *C* ou *C++*.

Table des matières

Introduction

Le GPU est une architecture initialement dédiée au calcul pour applications graphiques. Les nombres de publications scientifiques annuelles à son sujet, présentés dans la figure 1, montre une popularité en constante hausse ces dernières années. Son application au calcul scientifique généraliste intitulé, General-purpose Processing on Graphics Processing Units (GPGPU), a de même connu un fort essor jusqu'en 2014. Les publications de la communauté scientifique à ce sujet tendent depuis à diminuer lentement. Pourtant, le GPU est couramment cité pour les applications de calcul intensif, où il est présenté comme une architecure de choix du fait de sa forte puissance de calcul. La figure 2, publiée par Nvidia, met en avant le nombre d'unités de calcul des GPUs. Cette vue est cependant très simpliste et ne reflète aucunement la complexité de mise en œuvre de cette architecture que nous détaillons dans ce manuscrit.

Aujourd'hui, le GPU est couramment relié à d'autres sujets tels que l'intelligence artificielle, les réseaux de neurones, le *machine learning* ou encore le *deep learning*. Mais le GPU est aussi massivement utilisé dans le domaine de la finance et plus précisément sur un sujet d'actualité très médiatisé, celui des *crypto monnaies*. Cette devise virtuelle que l'on *«mine»*, nécessite une puissance de calcul élevée. La masse de calcul produite génère alors une valeur lucrative. Cependant, afin d'optimiser le rendement des ces opérations financières, l'efficience énergétique de l'architecture utilisée revêt un aspect important. Le GPU étant un bon candidat à ce sujet, il est fait état en 2018 d'une pénurie des GPUs liée à l'*eldorado* des *crypto monnaies*. Suivant le marché de l'offre et de la demande, une hausse des prix de ces processeurs est actuellement redoutée.

Dans le domaine industriel et plus particulièrement dans l'embarqué, nous retrouvons le GPU dans les équipements automobiles. Celles-ci sont développées par plusieurs grands constructeurs du monde automobile tels qu'Audi, Mercedes, Tesla, Toyota, Volvo ou encore Volkswagen. Tesla, en particulier, utilise un GPU pour analyser l'environnement du véhicule et offrir des fonctions de pilotage automatique voire de conduite totalement autonome. Cependant, pour arriver à un tel niveau d'avancée technologique et surtout un niveau de fiabilité indispensable, il est nécessaire de travailler sur une quantité et une qualité suffisante de données. Ce point explique le nombre élevé de capteurs inclus dans les voitures modernes. Par exemple, le *model S* chez Tesla propose de série une unique caméra. En option, la fonction de pilotage automatique requiert l'utilisation de trois caméras supplémentaires, tandis que la conduite autonome nécessite encore quatre caméras de plus, portant le total à huit caméras embarquées. À cet ensemble vient s'ajouter les autres senseurs non vidéo tels que les capteurs de type LIght Detection And Ranging (LIDAR). La figure 3 donne un aperçu de la répartition de ces senseurs et surtout de leur quantité sur ce véhicule. Perpendiculairement à cette notion de quantité vient s'ajouter celle de la qualité. Nous assistons à ce sujet à une course à la définition des senseurs de manière globale. Celle-ci est bénéfique dans le sens où elle améliore la qualité des résultats fournis par les différentes analyses. En contre-partie, l'augmentation des quantités de don-

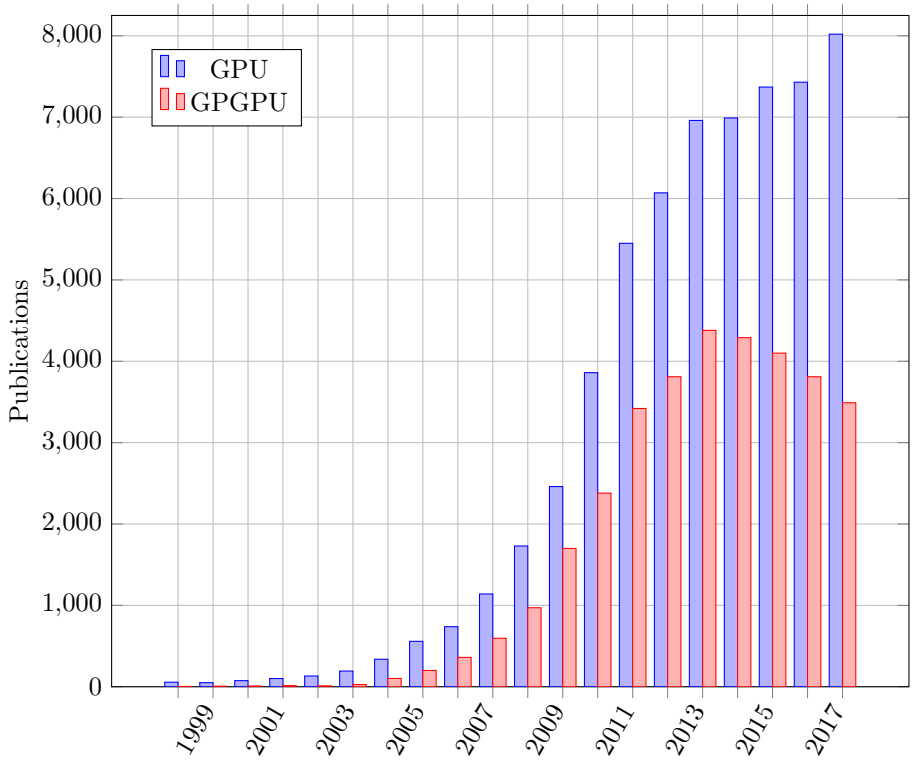


FIGURE 1 – Évolution du nombre de publications référencées par *Google Scholar* pour les mots clés *GPU* et *GPGPU*.

nées exploitées par les divers algorithmes contribue aux problématiques de saturation des bandes passantes mémoires. Dans le cas des capteurs vidéo, cette évolution est même de complexité quadratique de par l'exploitation d'une surface de capture à deux dimensions.

En parallèle, le temps limité de calcul imposé pour une utilisation temps réel ainsi que les contraintes de placement liées à l'embarqué, impliquent l'utilisation d'architectures aux caractéristiques spécifiques. En effet, celles-ci doivent nécessairement présenter un bon équilibre entre puissance de calcul, temps de transfert mémoire et consommation énergétique. Si le GPU présente un profil favorable à ces besoins, sa complexité architecturale rend aujourd'hui encore sa mise en œuvre non triviale, notamment pour les applications dont la complexité spatiale [1] des algorithmes demeure élevée.

Sujet de la thèse

Ainsi, dans le cadre de cette thèse, nous nous sommes intéressés à l'étude et à la définition d'une *«méthodologie de placement d'algorithmes de traitement d'images sur architecture massivement parallèle»*. Nous adressons dans ce sujet trois notions principales.

Tout d'abord, le domaine d'application ciblé est celui du traitement d'images. Cependant, comme nous le verrons dans ce manuscrit, notre approche de placement demeure généraliste et est ainsi adaptée à tout autre domaine d'application. À des fins d'applications industrielles, nos travaux ont cependant été évalués sur des algorithmes de traitement d'images.

Ensuite, la plateforme applicative ciblée est une architecture massivement parallèle. Notre choix s'est naturellement porté vers les GPUs. Celui-ci se justifie par l'adéquation de cette architecture vis-à-vis des caractéristiques propres aux algorithmes de traitement

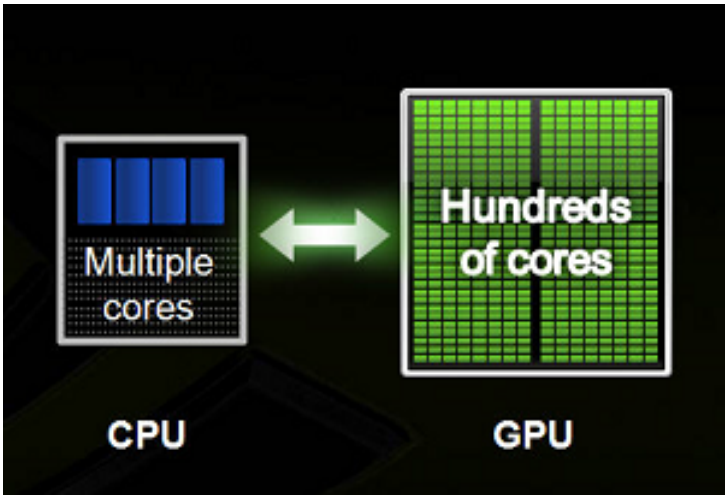


FIGURE 2 – Exemple de vulgarisation comparant les architectures CPU et GPU. *Source: Nvidia*

d'images de manière générale. De plus, les capacités actuelles du GPU additionnées à son implication nouvelle dans les applications embarquées en font une architecture d'intérêt. Nous reviendrons sur ces différents points dans la suite du manuscrit.

Enfin, la méthodologie est le guide vers l'objectif à atteindre. Elle constitue la clé de voûte permettant de relier entre elles, les deux premières notions.

Contributions

La méthodologie développée dans cette thèse constitue ma principale contribution. Elle permet de placer une application sur architecture hétérogène impliquant un processeur hôte et un GPU. J'ai pour cela spécifié un ensemble de critères qui permettent d'identifier les zones de codes pouvant être placées sur GPU. Ces critères sont alimentés par plusieurs phases intiales d'analyses de code statiques et dynamiques.

Afin de rendre cette méthodologie plus intuitive, j'ai en parallèle élaboré une représentation intermédiaire basée sur les graphes de flots de données. Cette représentation permet de visualiser le contenu d'une application dans sa globalité. Elle permet aussi de contrôler visuellement les transformations de code envisagées ainsi que les fragments de code placés sur GPU.

À ce sujet, j'ai sélectionné un ensemble de transformations de code permettant d'améliorer la quantité de placement sur GPU. Ces transformations ont pour rôle d'augmenter la quantité de fragments de code candidats à une exécution sur GPU. J'ai de plus défini des étapes optionnelles permettant de réduire les temps d'exécution sur GPU. Ces optimisations modifient notamment les localités spatiales et temporelles des différents fragments de code précédemment identifiés. Elles emploient aussi des modèles de performance tel que le calcul de l'*arithmetic intensity*, utilisé notamment pour le *roofline model* [164], afin de vérifier la qualité du placement vis-à-vis des capacités maximales de cette architecture.

Enfin, j'ai ajouté une étape de spécialisation pour les architectures Nvidia. Celle-ci permet de se rapprocher de la *peak performance* des GPUs du fabricant. Mon approche porte notamment sur le choix des espaces mémoires ou encore sur la concurrence intra et inter GPU.

Durant la conception de cette méthodologie, j'ai pu effectuer des expériences sur deux

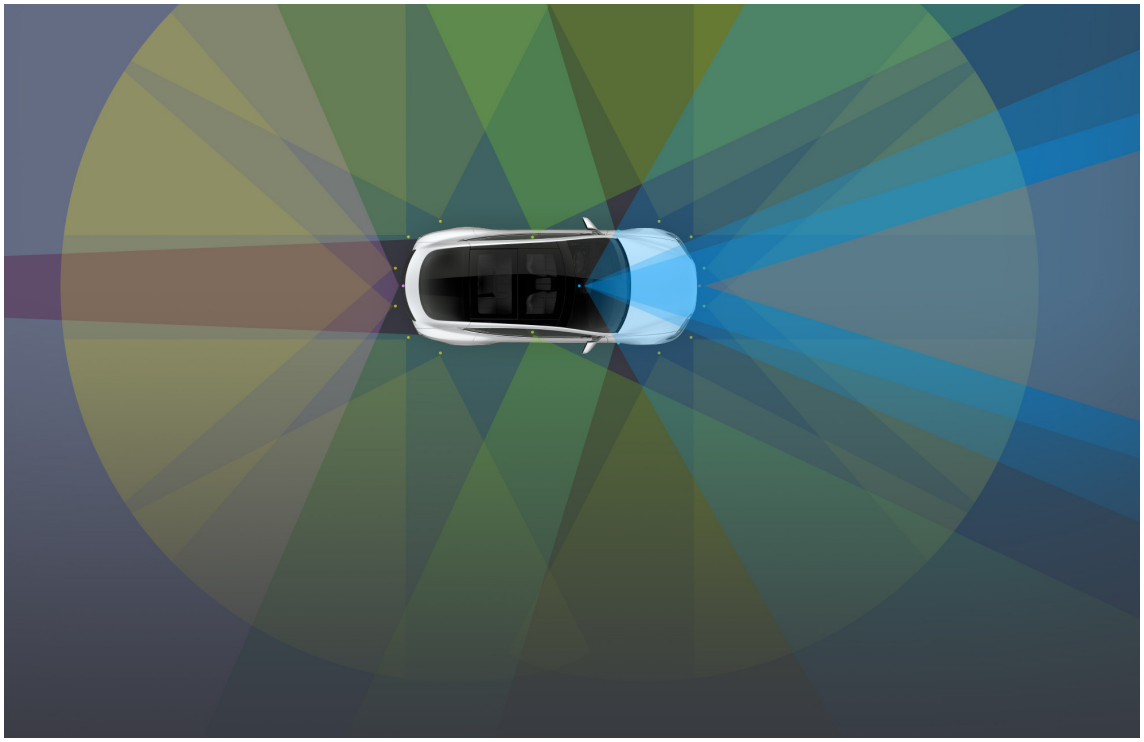


FIGURE 3 – Répartition des capteurs embarqués sur la voiture *model S*. *Source: Tesla*

cas d'application industriels. Les algorithmes Threewise [64, 63] et Simpleflow [150] ont ainsi été portés sur les GPUs Quadro K2000 et TX1 de Nvidia avec, au passage, une réduction significative des temps d'exécution.

Structure du manuscrit

Ce manuscrit de thèse se compose de cinq principaux chapitres.

Dans le **chapitre 1**, nous introduisons l'architecture GPU de manière générale. Nous débutons avec un court historique expliquant la création de cette architecture en rupture et faisons un tour d'horizon des différents acteurs présents sur le marché. Nous décrivons ensuite le principe de fonctionnement de cette architecture afin de mieux appréhender ses spécificités. Comme pour tout processeur, le GPU est contrôlable par un langage dédié, son Instruction Set Architecture (ISA). Ce langage spécifique à chaque processeur est cependant rarement utilisé directement par les développeurs à cause de sa complexité et surtout de sa spécificité. Un langage intermédiaire est alors proposé afin de simplifier le dévelopement des applications et d'améliorer la compatibilité de ces dernières pour différentes générations d'architecture. Nous parlons ainsi dans ce même chapitre de ces Application Programming Interface (API) permettant de mettre en œuvre les GPUs. Enfin, nous concluons ce chapitre en évoquant l'implication du GPU, dans le domaine du traitement d'images.

Après cette mise en contexte, nous détaillons, dans le cadre du **chapitre 2**, l'état de l'art du placement d'algorithmes sur GPU. Ce précieux héritage, construit durant ces dix dernières années par la communauté scientifique et plus particulièrement par la communauté de la compilation, a permis d'améliorer la prise en main de cette architecture puissante mais dont l'exploitation efficiente reste néanmoins complexe. Nous détaillons particulièrement les différentes méthodes utilisées pour le placement d'algorithmes sur

GPU mais aussi l'optimisation de ce placement.

Avec la connaissance de l'architecture des GPU et des méthodes de placement issues de l'état de l'art, nous développons dans le courant du **chapitre 3** notre propre méthodologie de portage. Celle-ci est régie par un ensemble de critères propres à l'architecture des GPU.

Dans le **chapitre 4** sont détaillées les différentes expérimentations réalisées validant notre méthodologie.

En complément, nous avons mené, dans le **chapitre 5**, deux études portant sur les critères de performance des différentes mémoires ainsi que la gestion du parallélisme à gros grain pour deux architectures GPU de Nvidia.

En raison du volume important de données représentées, les résultats d'analyse de notre méthodologie, liés à la représentation graphique que nous avons développée, ont été délégués en **annexe** de ce manuscrit de thèse.

Chapitre 1

Contexte

Sommaire

Dans ce chapitre nous présentons les causes de la naissance d'une architecture en rupture : le GPU. Nous détaillons ses moyens de mise en œuvre, son fonctionnement architectural mais aussi son implication dans le domaine du traitement d'images. Cette mise en contexte a pour but de mieux appréhender les chapitres suivants.

1.1 L'héritage des GPUs

Une *«étrange maladie»* a touché nos processeurs au début du 21$^{\text{ème}}$ siècle. Alors que leurs performances étaient jusque là sans cesse améliorées, un effet de plafonnement est venu opérer. La structure même des Central Processing Unit (CPU)s ne permettait alors plus d'augmenter leurs fréquences de fonctionnement. La solution pour contourner ce problème a été de multiplier les cœurs de calculs au sein des processeurs. La scalabilité de ces derniers était alors rendue possible par l'exploitation du parallélisme. Cependant, alors que nous étions aveuglés par des performances calculatoires toujours plus élevées, les performances des transferts de données n'ont pas connu la même évolution. Hennessy et Patterson dans leur ouvrage de référence [75] font référence à ce constat. Afin de palier à cette contrainte, de nouveaux ensembles d'instructions tels que MultiMedia eXtension (MMX), Streaming SIMD Extensions (SSE) puis Advanced Vector Extensions (AVX) ont été ajoutés aux jeux d'instructions des CPUs. Ces instructions, à l'origine prévues pour les applications multimédia, permettent d'exploiter des unités de calculs vectoriels dont le but est de regrouper une ou plusieurs opérations sur un même bloc de données consécutives. Au sein de la taxynomie de Flynn [60], cette approche architecturale est de type Single Instruction on Multiple Data (SIMD).

Dans le domaine du rendu graphique, l'application récurrente de certains patterns, ainsi qu'une augmentation sans fin des volumes de données à traiter, ont engendré la création de processeurs dédiés à l'accélération graphique. C'était le début des GPUs. Leur ambition était d'accélérer les temps de rendu pour atteindre le temps réel, mais aussi d'améliorer la qualité de rendu des scènes tri–dimensionelles pour le domaine des jeux vidéo. Ce marché très porteur a permis aux industriels de faire progresser le concept des GPUs basé sur une approche vectorielle pour arriver à une architecture massivement parallèle de type Single Instruction Multiple Thread (SIMT). Ces capacités calculatoires se comptent en *teraflops* pour plusieurs centaines, voire milliers, de cœurs de calcul tandis que la bande passante mémoire se compte en centaines de giga-octets par seconde. La figure 1.1 nous montre l'évolutions de ces performances. La position dominante du GPU comparée aux autres architectures explique non seulement la popularité de cette architecture mais aussi son origine.

Le détournement des GPUs pour effectuer des calculs plus généralistes a été facilité par la libération en 2007 du pipeline de rendu graphique. Celui-ci auparavant considéré comme une boîte noire, devenait alors programmable par les développeurs pour permettre d'utiliser la puissance de calcul des GPUs pour toutes sortes de calculs volumineux.

L'ère du GPGPU débutait.

1.2 Les différents acteurs

En 2018, trois principaux acteurs sont présents dans le domaine des GPUs. Intel, Nvidia et Advanced Micro Devices (AMD) se partagent l'héritage d'un passé où les GPUs n'étaient utilisés que dans l'unique but d'accélérer les calculs de rendu de scènes en trois dimensions. Avec l'avènement des GPUs intégrés dans les architectures embarquées telles

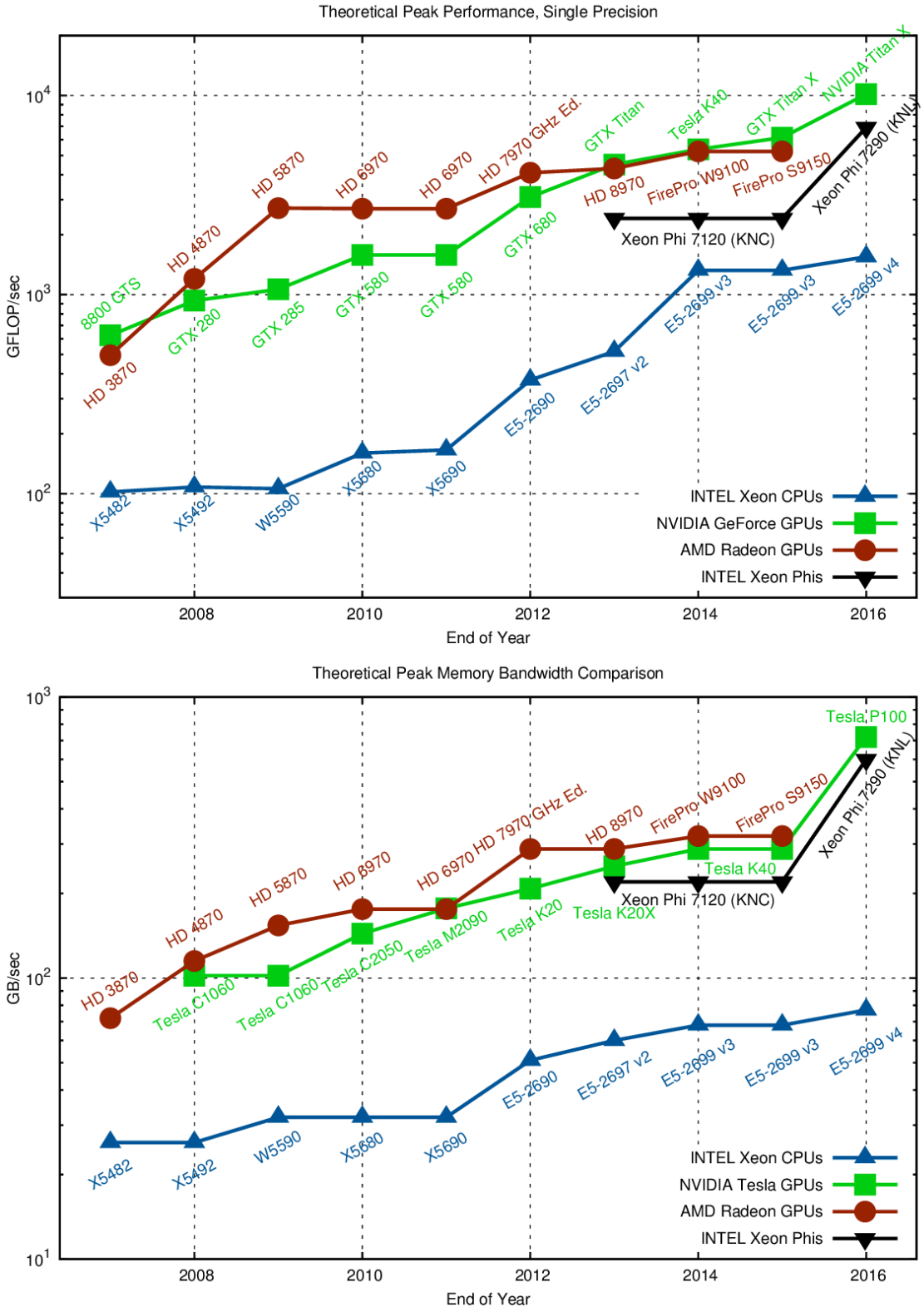


FIGURE 1.1 – Évolution des performances maximales de différentes architectures au cours du temps. Le graphique du haut représente les performances calculatoires, celui du bas la bande passante mémoire. *Source: Karl Rupp*

que les *smartphones* et les tablettes tactiles, d'autres acteurs sont venus s'ajouter dans le cadre de ce marché de niche exclusivement. Dans le domaine du High Performance Computing (HPC), en novembre 2017, la gamme *Tesla* de Nvidia se retrouve en concurrence avec le *Xeon Phi* d'Intel, dans les dix premiers super-calculateurs du classement *top500*[10].

1.2.1 Intel

L'entrée d'Intel dans le domaine des GPUs remonte à 1998 avec son processeur graphique Intel740. S'en suivront une seconde génération intitulée *Extreme Graphics* au tout début des années 2000 puis plusieurs générations liées à l'architecture Intel Graphics Media Accelerator (Intel GMA). Il faudra cependant attendre la génération *Intel HD Graphics*, en 2013, pour avoir une utilisation de type GPGPU. Cependant, la stratégie d'Intel est d'intégrer dans le même System On Chip (SOC) que ses CPU [1] un GPU servant à la fois pour les rendus graphiques mais aussi pour effectuer du calcul vectoriel en partageant un même espace mémoire. Ce choix stratégique explique le *leadership* d'Intel en terme de GPUs vendus. Cependant, ces unités, appelées Integrated Graphics Processor (IGP), sont optimisées pour une faible consommation énergétique au prix d'une puissance de calcul modérée. Afin de palier ce problème, la série de GPUs *Iris Graphics* est venue renforcer l'offre d'Intel. Il s'agit de la même base architecturale que la série *Intel HD* mais paramétrée pour apporter une puissance de calcul supérieure. La série *Iris Pro Graphics* embarque en supplément une mémoire interne dont la taille est de l'ordre de 100MB. Enfin, en 2017, avec l'architecture *Coffee Lake*, la dénomination des IGP *Intel HD* a été modifiée par *Ultra HD*. Avec au plus un peu moins de 600 cœurs de calcul et une mémoire interne de 100MB, l'ensemble de ces solutions ne permet pas d'atteindre à génération égale le niveau de performance des GPUs dédiés de Nvidia ou d'AMD. Ce constat s'explique par le choix d'Intel d'utiliser une approche architecturale fondamentalement différente. Ce choix puise ses origines dans l'architecture *Larrabee* qui s'avèrera être un échec à cause de sa complexité de mise en œuvre. Elle permettra cependant de poser les bases de l'architecture *Xeon Phi* dédiée au calcul vectoriel haute performance. Cette architecture en concurrence avec les GPGPUs est en réalité une architecture *manycores* de type *x86* basée sur une topologie en anneau et utilisant massivement des instructions de type SIMD comme l'AVX 512. Celle-ci est donc, en toute logique, fortement impactée par la problématique de coalescence dans ses accès aux données mémoire.

1.2.2 AMD / ATI

Initialement spécialisées dans les CPUs à architecture x86, les solutions graphiques proposées par AMD sont issues de l'héritage d'Array Technologies Incorporated (ATI), suite à son rachat en 2006. On retrouve principalement les solutions GPUs d'AMD dans le domaine des jeux vidéos, notamment avec les consoles de jeux *Playstation 4* de Sony et *Xbox One* de Microsoft. Nintendo a longtemps employé les solutions graphiques d'AMD dans ses consoles de jeux vidéos. Cependant, en 2017, l'entreprise a décidé de changer de stratégie en utilisant l'architecture Tegra X1 de Nvidia pour sa console *Switch*.

L'architecture *TeraScale* est la première de type GPGPU grâce à la libération du pipeline graphique fixe. Trois générations se succéderont. L'approche architecturale est basée sur un jeu d'instruction de type Very Long Instruction Word (VLIW) et SIMD.

L'architecture Graphics Core Next (GCN) qui a succédé en 2011 à l'architecture TeraScale, connaîtra en 2019 sa $6^{\text{ième}}$ génération intitulée *Navi*. GCN se différencie de son

1. À l'exception de certains processeurs des séries XEON destinées aux serveurs.

aînée par l'utilisation d'un jeu d'instructions de type Reduced Instruction Set Computer (RISC) et SIMD ce qui la classifie au final comme une architecture SIMT et la rapproche ainsi des architectures de son concurrent Nvidia.

Pour le domaine du HPC, AMD propose une gamme *Fire Pro* comparable à la gamme *Tesla* chez Nvidia. Concernant le domaine de l'embarqué, AMD propose un Accelerated Processing Unit (APU) intégrant un CPU et un GPU basé sur les architectures *TeraScale* puis GCN dans un unique SOC. Les deux processeurs partagent alors le même espace mémoire. C'est ce modèle de processeur qui est utilisé dans les solutions pour consoles de jeux proposées par AMD.

1.2.3 Nvidia

La première génération de GPGPU est apparue chez Nvidia avec l'architecture Tesla [2] plus connue sous l'appellation G80. Se succéderont par ordre chronologique les architectures Fermi, Kepler, Maxwell, Pascal et enfin Volta en 2017.

Nvidia est un acteur très actif dans le domaine du calcul scientifique avec sa gamme de produit *Tesla*. Pour le domaine de l'embarqué, la gamme *Tegra* similaire à l'AMD APU, intègre un GPU classique de la marque avec un CPU basse consommation sur un même SOC additionnée d'une mémoire partagée entre les deux entités. Plus de détails sur l'architecture Nvidia seront donnés dans la section 1.3.

1.2.4 Autres acteurs

Matrox a été un acteur actif dans le domaine des GPUs. Cependant, l'entreprise n'a pas pris le virage du GPGPU et s'est spécialisée aujourd'hui dans le domaine de l'affichage et du rendu graphique.

On retrouve enfin plusieurs acteurs dans le marché de niche de l'embarqué parmi lesquels nous pouvons citer *Mali*, *Adreno*, *Vivante*, *VideoCore* ou *PowerVR*. Les performances calculatoires de ces GPUs sont cependant très limitées comparées à des solutions telles que *Nvidia Tegra* ou *AMD APU*.

1.3 Architecture générale des GPUs

La figure 1.2 représente l'agencement de l'architecture GP104 de génération Pascal qui est utilisée dans le GPU Nvidia GTX 1080 [133]. Nous nous appuyons sur cette architecture populaire en 2018 pour décrire le fonctionnement des GPUs. Les autres générations de GPU ont cependant une structure architecturale et un fonctionnement similaire. Le GPU est considéré comme un accélérateur ne pouvant fontionner de façon autonome. Ainsi dans la figure 1.2, l'interface d'entrée permettant de mettre en œuvre le GPU est le port PCI Express 3.0 servant d'interface de communication avec le processeur hôte mais aussi avec d'autres GPUs, s'il y en a.

L'architecture des GPUs, comme beaucoup d'autres architectures, peut être décomposée selon 3 dimensions. Ainsi, le flot d'instructions, le flot de données et le flot calculatoire sont trois axes dimensionnels pouvant constituer pour chacun d'entre eux un facteur limitant dans l'exécution d'un algorithme. On parle alors de *bottleneck* lorsque la saturation de l'une de ces trois dimensions limite la capacité globale.

2. Chez Nvidia, l'architecture G80 Tesla aujourd'hui obsolète est à distinguer de la gamme de produit Tesla dédiée au calcul haute performance pour serveurs de calcul.

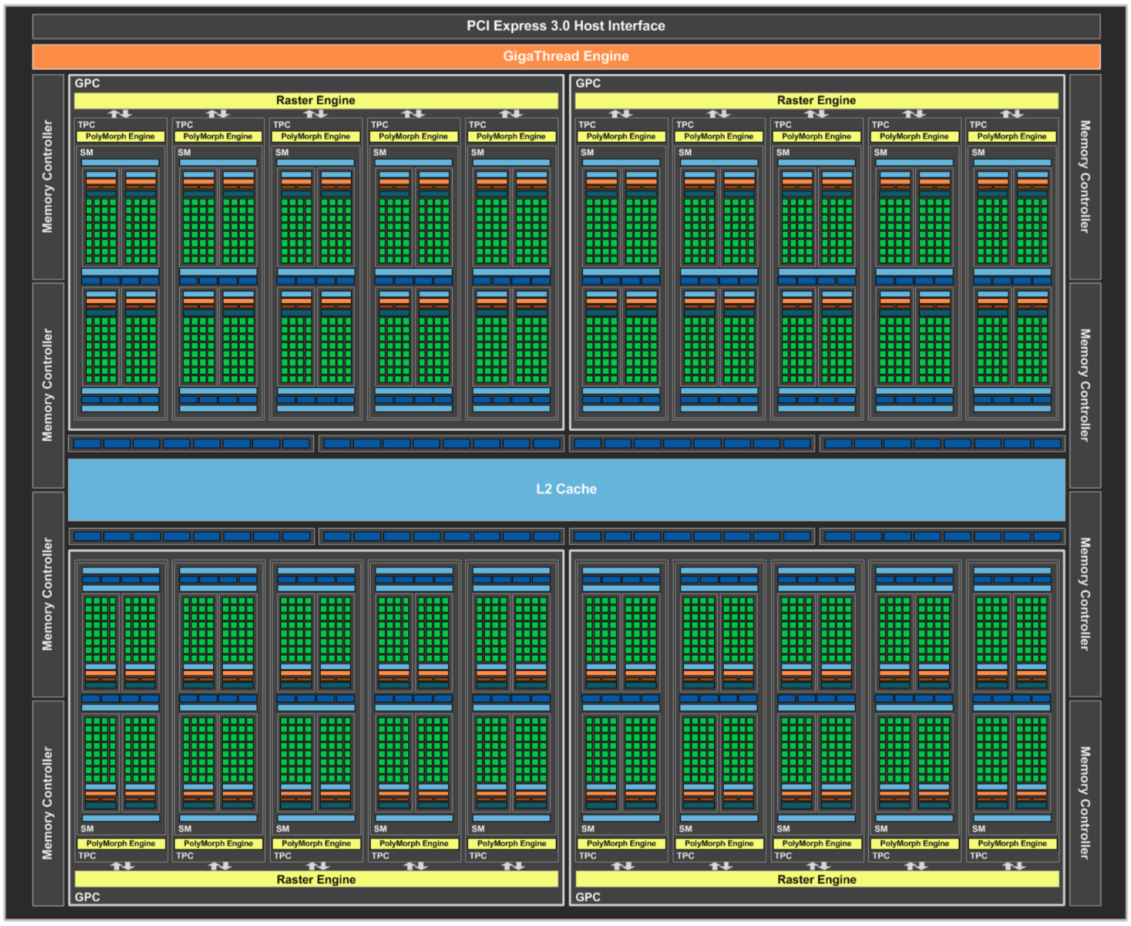


FIGURE 1.2 – Vue globale de l'architecture Nvidia Pascal - GP104 utilisée pour les GTX 1080

1.3.1 Le flot calculatoire

Le GPU est une architecture complexe à hiérarchies multiples. Au plus haut niveau, le SOC du GPU contient plusieurs clusters appelés Graphics Processing Cluster (GPC) dans la figure 1.2. Ils sont au nombre de quatre pour la GTX 1080 mais peuvent être réduits à un unique cluster comme pour les GPUs embarqués de la gamme Tegra, par exemple. Cette clusterisation paramétrable apporte ainsi une forte scalabilité aux architectures GPU. Chacun de ces clusters incorpore plusieurs Texture Processor Cluster (TPC) composés de deux Streaming Multiprocessor (SM) chacun et d'un *Polymorph engine*. Ce dernier, ainsi que les *raster engine*, ne présentent pas d'intérêt pour l'HPC, car leur fonction est de gérer l'exécution des *shaders* dans le cadre des rendus graphiques 3D. Nous reviendrons sur ce sujet dans la section 1.4. En revanche, la vue détaillée d'un SM nous montre que chacun d'entre eux incorpore des unités de calcul intitulées *core*. Celles-ci sont utilisées pour le calcul simple précision des nombres entiers et flottants. Le calcul en double précision est effectué par les *DP Units*. Les Special Function Unit (SFU) permettent d'effectuer des opérations complexes telles qu'un calcul de racine carré ou encore de cosinus. Enfin, les *load/store units* sont utilisées pour le traitement des données. Nous reviendrons sur ces derniers dans la section 1.3.2.

Pour la génération Pascal, chaque SM est composé de 64 *Compute Unified Device Architecture (CUDA) cores*, 32 *DP units*, 16 *load/store units* et 16 *SFU*. Cependant, d'une

FIGURE 1.3 – Vue d'un multi–processeur SM de l'architecture Nvidia Pascal

génération à l'autre ces valeurs peuvent varier. On notera tout de même une stabilisation à 64 *cores* dédiés aux calcul de flottants simple précision, depuis plusieurs générations chez Nvidia, mais aussi chez AMD avec l'architecture GCN.

1.3.2 Le flot de données

Sur la figure 1.2, de part et d'autre du processeur, nous retrouvons les huit contrôleurs mémoire permettant d'intéragir avec la mémoire globale du GPU située à l'extérieur du SOC. Toute transaction avec celle-ci se fait au travers de la mémoire cache L2, unifiée pour tout le processeur. Ces transactions se déroulent par ligne de 128 octets chacune. De plus, tout transfert de données entre l'espace mémoire du processeur hôte et la mémoire globale du GPU transite tout d'abord par le port PCI Express, ensuite par le même cache L2 et enfin par les contrôleurs mémoire avant d'atteindre la mémoire "globale". Le cheminement est identique pour les échanges de données entre GPUs. À Ce niveau, l'architecture est de type Symmetric MultiProcessing (SMP).

À plus bas niveau dans la hiérarchie du GPU (cf. figure 1.3), nous retrouvons au sein des SM plusieurs branches permettant de transporter les données provenant de la mémoire globale jusqu'aux unités de calcul. Dans le cas le plus simple, la donnée transite alors du cache L2 vers le cache L1, pour ensuite atteindre les *load/store units* qui mettront finalement à disposition des unités de calculs, les données solicitées. La portée du cache L1 est exclusivement limitée à son SM et les lignes de cache transférées font 32 octets. Ainsi, les caches L1 et L2 permettent dans le cas d'une localité temporelle efficace d'améliorer la bande passante d'accès aux données grâce aux *cache hits*. À cela s'ajoute la localité spatiale des données qui est un second facteur optimisant la bande passante mémoire. Ce

gain est obtenu par l'effet réducteur sur le nombre de requêtes effectuées par les *load/store units* dans un SM.

Il existe de plus deux autres moyens d'accéder aux données présentes dans la mémoire globale. Dans le premier cas, la *constant memory* est en réalité un fragment de la mémoire globale réservé aux données utilisées en lecture seule par le GPU. Un cache dont le comportement est étudié plus en détails au chapitre 5.1 lui est dédié. Dans le second cas, la *texture memory* représente de même une sous-partie de la mémoire globale que le GPU ne peut accéder qu'en lecture seule. Cependant, l'utilisation de la *surface memory* permet d'accéder au même espace mémoire en lecture et écriture. Dans le cadre de la *texture memory*, les unités de calcul de texture intitulées *tex* dans la figure 1.3 apportent le calcul d'interpolation entre valeurs contigües, la réplication de données en dehors des espaces de données alloués ou encore la normalisation de la dynamique des données entre 0 et 1. Enfin, la *texture memory* dispose d'un cache partagé avec le cache L1 mais dont l'utilisation est optimisée pour la localité spatiale dans le cadre d'appels bidimensionels aux espaces de données.

Les variables scalaires initialisées lors de l'exécution d'un kernel sur GPU, sont stockées dans l'espace des registres propres au SM concerné. Cet espace intitulé *register file* dans la figure 1.3 est présent deux fois par SM. Enfin, dans le cas où un tableau serait initialisé, celui-ci serait positionné dans la mémoire locale et non dans l'espace des registres. La *mémoire locale* est en réalité un fragment de la *mémoire globale* dont l'utilisation repose sur un fonctionnement classique au moyen des caches L1 et L2.

Enfin, il existe un dernier espace mémoire appelé *shared memory* chez Nvidia. Cet espace mémoire est segmenté dans chaque SM ce qui rapproche ce niveau hiérarchique d'une architecture à mémoire distribuée. Cet espace mémoire, par sa localité proche des unités de calcul, présente une bande passante supérieure à celle de la *mémoire globale*. D'autres types d'architectures utilisent pour le même concept l'appellation *scratchpad memory*. En restant dans le domaine des GPUs, AMD au moyen du langage Open Computing Language (OpenCL) emploie le terme *mémoire locale*. Cette dernière est à distinguer de la *mémoire locale* précédemment décrite.

1.3.3 Le flot d'instructions

Le GPU est un accélérateur. Il doit être interfacé avec un processeur hôte. De ce fait, au plus haut niveau, c'est le CPU qui déclenche l'exécution de *kernels*, avec un fonctionnement asynchrone entre les deux entités. Seize *pipelines* d'instructions sont disponibles pour permettre au CPU de gérer les kernels ainsi que les unités de transfert mémoire au travers du bus *PCI Express* du GPU.

Au sein du GPU, les GPC ont un comportement de type Multiple Instructions on Multiple Data (MIMD). Chaque *cluster* reçoit ses instructions à exécuter dans le cadre d'un kernel et aucune synchronisation entre *clusters* n'est possible. Au niveau des SM en revanche, il existe des instructions de synchronisation qui apporteront une approche SIMD à ce niveau hiérarchique. Chaque kernel lors de son appel par l'hôte reçoit ainsi un nombre global d'instances à exécuter appelé *grille*. Cette dernière est composée d'un certain nombre de *blocks* correspondant aux itérations à placer sur les GPC. Ces *blocks* sont eux-mêmes composés de plusieurs *threads* représentant les itérations à placer sur les SM. De ce fait le découpage multi-hiérarchiques des itérations se fait de la façon suivante :

$$taille\ de\ grille = nombre\ de\ blocks \times nombre\ de\ threads\ par\ block \tag{1.1}$$

Les instructions provenant d'un kernel sont placées dans le *cache d'instructions* de chaque SM. Ces instructions sont alors dispatchées dans les *buffers d'instructions* et les *blocks* de

threads sont pour leur part sous-découpés en *warps* de trente deux *threads* par le *warp scheduler*. Les *dispatch units* viennent alors alimenter en instructions les différentes unités du SM.

1.4 Interfaces de programmation pour GPU

Les API présentées dans cette section utilisent un modèle de calcul par *streams*. Celui-ci considère comme *stream* un flot de données constituant la source d'alimentation d'une application. Sur chacun des éléments constituant ce flot est alors appliqué une série d'opérations regroupées au sein de fonctions intitulées *kernels*. Celles-ci, dans un souci de rendement calculatoire, sont exécutées au travers d'un ou plusieurs *pipelines* afin de masquer les temps de latence. Le principe du calcul par *stream* est donc fortement centré sur les données et est de ce fait souvent associé à l'utilisation de graphes de flot de données. On le retrouve ainsi employé pour adresser d'autres types d'architectures de type Field-Programmable Gate Array (FPGA) ou Digital Signal Processor (DSP). De plus, ce modèle est particulièrement indiqué pour les algorithmes présentant une intensité arithmétique forte, un parallélisme de données évident et une bonne localité spatiale des données.

1.4.1 OpenGL

Open Graphics Library (OpenGL) dont les débuts remontent à 1992, est une librairie de rendu graphique, produite par le consortium industriel Khronos. Celle-ci spécifie une API haut niveau en langage C, dont différentes implémentations sont fournies par les concepteurs de GPU. Jusqu'au début des années 2010, cette librairie a été une des rares solutions pour effectuer du calcul parallèle sur le *pipeline* fixe des GPUs. Ce *pipeline* avait pour défaut de contraindre les possibilités calculatoires aux besoins exclusifs du rendu graphique. La libération du *pipeline* graphique dans la version 2.0 d'OpenGL a permis de le rendre programmable au moyen du langage openGL Shading Language (GLSL). Il était alors possible de détourner l'utilisation première du GPU en utilisant les *fragment shaders* ainsi que les *vertex shaders* pour effectuer du calcul généraliste haute performance. L'ère du GPGPU commençait.

Il est à noter qu'une version spécifique pour les systèmes embarqués a vu le jour sous l'appellation OpenGL for Embedded System (OpenGL ES). Celle-ci fournit les spécifications d'une API plus adaptée aux architectures présentant des ressources mémoire et des processeurs plus limités. En complément, OpenGL for Safety Critical applications (OpenGL SC) est une version dérivée d'OpenGL ES. Elle permet de répondre aux besoins des applications critiques grâce à son adéquation avec les normes *DO-178 Level A* pour l'avionique ou encore *ISO26262 ASIL D* pour l'automotive.

Pour en revenir à OpenGL, sa version 4.0 a rendu programmable d'autres étages du pipeline graphique tels que les *tessellations shaders*, permettant ainsi d'exploiter le parallélisme dynamique au sein des GPUs. De plus, face au besoin grandissant de calcul parallèle généraliste, la version 4.3 a introduit l'utilisation des *compute shaders*. Ceux-ci ont une approche conceptuelle similaire à celle d'OpenCL ou de CUDA.

Enfin, Vulkan, introduit dans la section 1.4.8, est appelé à remplacer OpenGL. Cette dernière est maintenue dans un unique but de rétro-compatibilité vis-à-vis des anciennes applications.

1.4.2 Direct Compute / Direct3D

Direct3D est une librairie de rendu graphique développée par Microsoft pour ses propres Operating System (OS). Elle constitue la principale alternative à OpenGL et en a suivi les mêmes cheminements liés aux évolutions des architectures GPU. Nous retrouvons ainsi en commun l'utilisation de *shader models* au moyen d'un langage de programmation intitulé High Level Shader Language (HLSL) et ayant pour particularité d'être compatible avec GLSL. De plus, Microsoft a introduit avec la version 11 de *Direct3D* le principe des *compute shaders* au moyen de leur API *Direct Compute*. Cette dernière permet alors d'adresser le calcul parallèle généraliste et la rapproche de solutions telles que CUDA ou encore OpenCL.

1.4.3 Cuda

CUDA [123] est à la fois le langage et l'API officiel de NVIDIA créé en 2007 pour mettre en œuvre les GPUs de la même marque. L'API est décomposée en deux niveaux. Le premier, intitulé *CUDA Driver API* [125] est une API C de bas niveau permettant un niveau de contrôle à très faible granularité des GPUs NVIDIA à partir du processeur hôte. En tant que second niveau, la *CUDA Runtime API* [126] est une surcouche de la première, simplifiant la gestion mémoire ainsi que la configuration et le lancement des threads sur le GPU.

NVidia Cuda Compiler (NVCC) [124] est le compilateur officiel pour le langage CUDA. Celui-ci repose sur la librairie NVidia Parallel Thread eXecution (NVPTX) développée par NVIDIA et intégrée au sein du framework de compilation LLVM [86]. NVCC permet de générer un code binaire exécutable sur GPU à partir d'un code C ou C++. Ce processus de compilation peut aussi utiliser l'ISA de Nvidia intitulée Parallel Thread eXecution (PTX) [127]. Deux approches de compilation sont disponibles : la compilation *offline* et la compilation de type Just In Time (JIT).

Dans le cadre de la **compilation *offline***, la première étape du processus de compilation passe par la séparation des portions de code pour hôte et des portions de code pour le GPU toutes deux intégrées dans le même fichier de code source. Le code à destination du GPU est alors utilisé dans deux étapes complémentaires. La première permet de transformer le code source en pseudo-code assembleur de type PTX. La seconde, permet de générer le *cubin* qui correspond au code binaire pour GPU. C'est durant cette étape que l'ISA spécifique à une génération de GPU est utilisée. Enfin, la portion de code hôte est modifiée pour pouvoir communiquer avec le GPU au moyen de l'une des deux API citées. La portion de code pour l'hôte modifiée est alors transférée à un compilateur CPU tierce pour générer le binaire exécutable. La compilation *offline* est le mode de compilation par défaut.

La **compilation JIT** se différencie en finalisant la compilation du pseudo-code assembleur PTX au moment de l'exécution du code hôte. Le processus de compilation est alors effectué par le *driver* du GPU ce qui engendre un surcoût en temps d'exécution au lancement de l'application générée. Un mécanisme de sauvegarde des binaires compilés, complété par un système de *versioning*, permet de limiter ce surcoût. La compilation JIT aurait pu avoir un intrérêt pour adapter dynamiquement le code d'un kernel en fonction des portions de codes à contrôle dynamique. Cependant, contrairement à OpenCL, qui utilise exclusivement un format de compilation JIT sur un code source, la manipulation de code pseudo-assembleur par NVCC complexifie cette approche dynamique.

La principale critique formulée à l'encontre de CUDA reste sa limitation aux architectures Nvidia contrairement à son concurrent principal, OpenCL. En revanche, cette

spécialisation permet à CUDA d'être d'une part plus proche de la logique de fonctionnement de leurs GPUs, et, d'autre part, plus dynamique quant à la prise en compte des évolutions liées aux différentes générations de GPUs. Ces éléments facilitent ainsi l'optimisation du placement sur les architectures Nvidia, ce qui explique la popularité de CUDA aujourd'hui.

1.4.4 OpenCL

OpenCL [148] est à la fois une API ainsi qu'un langage de programmation dont Apple a été l'initiateur en 2009. Comme pour OpenGL ou Vulkan, ses spécifications sont définies par le consortium Khronos et son implémentation reste à la charge des concepteurs d'architectures. OpenCL connaît une forte popularité car contrairement à CUDA, il permet d'adresser de nombreuses architectures hétérogènes parmi lesquelles nous pouvons citer les CPUs et les GPUs. Il adresse ainsi non seulement le parallélisme de données mais aussi le parallélisme de tâches. Son mode de fonctionnement est cependant figé à un modèle de compilation de type JIT dont l'API impose la manipulation de chaînes de caractères en guise de code source. Ce mode de fonctionnement apporte un gain potentiel pour la parallélisation de code à contrôle dynamique en permettant de manipuler et d'adapter le code source en fonction du contexte et des contraintes d'exécution. Plusieurs solutions présentées dans le chapitre 2 mettent en avant ce concept. Cependant, ce modèle de compilation souffre d'une complexité de mise en œuvre supérieure liée à la gestion du code source dédié à OpenCL. Cela se traduit par un transfert de la charge de travail automatisée du compilateur vers celle du développeur. Ce point est souvent reproché à OpenCL, notamment dans le cadre d'application sur de faibles portions de code. Cependant, *SYCL* [9, 82] du même consortium Khronos, utilisé conjointement avec OpenCL, permet de se rapprocher du modèle de *compilation offline* de CUDA.

1.4.5 BrookGPU

Brook [31] est un langage de programmation ainsi qu'un compilateur créé par Stanford University dont le concept est basé sur un modèle de calcul par *streams*. *BrookGPU* [30] qui est une version de Brook spécialisée pour GPU, est reconnu comme étant un des premiers acteurs ayant favorisé l'emmergence du GPGPU par le détournement du pipeline de rendu graphique pour effectuer du calcul généraliste sur GPU. Il permet notamment de générer du code pour les librairies basées sur la programmation par *shaders* telles qu'OpenGL ou directX. Il a de ce fait servi de base pour plusieurs API de calcul sur GPU dont celles officielles de Nvidia et d'AMD.

1.4.6 ATI Stream / CTM

ATI Stream est initialement connu sous le nom de Close To Metal (CTM). Il s'agit d'une librairie utilisant le modèle de calcul par *streams* de Brook à partir d'une version dérivée de celui-ci intitulée Brook+. Son objectif initial était de rendre disponible les applications GPGPU pour les produits vendus à l'époque par ATI. Suite au rachat de ce dernier par AMD, cette librairie a été remplacée par AMD Mantle.

1.4.7 AMD Mantle

Mantle [97] est une API bas niveau spécifiquement définie pour les GPUs de son initiateur, AMD. Elle a été définie pour fournir une alternative à Direct 3D et OpenGL. Comparé à CTM dont il succède, Mantle apporte une réduction de la charge de travail du

CPU pour le pilotage du GPU. Cette amélioration permet de limiter le phénomène où les performances globales du GPU sont limitées par le CPU qui n'arriverait pas à suivre.

Mantle n'est aujourd'hui plus supporté. Il a cependant servi de base pour la définition de Vulkan qui légitimement devient ainsi son remplaçant.

1.4.8 Vulkan

Dérivé d'AMD Mantle, Vulkan est présenté par Khronos comme le successeur d'OpenGL. Basé sur le même principe que les autres solutions du consortium, Vulkan est une API générique dont les différentes implémentations existantes sont spécialisées pour une architecture de GPU donnée. Vulkan se veut ainsi multi-plateformes. On retrouve ainsi une implémentation pour les GPUs de Nvidia, AMD ou encore Intel. Son principe est de fournir au sein d'une nouvelle et unique API pour GPU des fonctions de rendu graphique comparables à OpenGL ajoutées de fonctions pour le calcul parallèle comme OpenCL. Vulkan hérite aussi des avancées de Mantle pour mieux équilibrer la charge de travail du CPU et du GPU. Le contexte Vulkan devient notamment *thread-safe*. Ainsi l'utilisation de plusieurs cœurs CPU est rendu possible. Enfin l'API de Vulkan est orientée objet.

1.4.9 Apple Metal

Apple Metal, développé par Apple pour son système d'exploitation IOS, est un concurrent direct de Vulkan. On retrouve ainsi la même volonté d'avoir une API bas niveau, alliant non seulement des fonctions de rendu graphique mais aussi de calcul parallèle. Son API est orientée objet, son driver supporte le *multi-threading* et une attention particulière est apportée à la charge de travail du CPU. Enfin, Metal est compatible avec les GPU de Nvidia, AMD et Intel.

1.4.10 Conclusion

Les architectures Nvidia ont une présence forte dans le secteur industriel. Leurs solutions Tegra, dédiées pour l'embarqué, connaissent notamment une grande popularité. Cependant, la programmation de ces solutions se limite exclusivement aux langages Vulkan, OpenGL et CUDA.

Notre choix, dans le cadre ce cette thèse, se porte sur le langage CUDA pour être en cohérence avec la demande du marché industriel pour les produits Nvidia. De plus, dans le cadre de notre méthodologie, ce choix nous permet de mieux exploiter les performances des GPUs Nvidia, grâce à la spécialisation plus avancée offert par ce langage.

1.5 Le GPU en traitement d'images

Dans le domaine spécifique du traitement d'images, les applications typiques correspondent à l'application d'un pattern algorithmique récurrent sur une portion ou la totalité d'images en entrée. Ainsi, chaque pixel constituant une image est une source potentielle de parallélisme de données. De plus, avec l'amélioration constante de la résolution des capteurs d'images, la quantité de données à traiter pour chaque algorithme de traitement d'images subit globalement la même croissance. En conséquence, la quantité de calculs à effectuer, ajoutée à l'augmentation du nombre de données à transférer entre les différents espaces mémoire, ont tendance à provoquer de manière générale en traitement d'images, une augmentation des temps d'exécution. Grâce à ses bandes passantes mémoires élevées

et son architecture de type SIMT adaptée au parallélisme de données, le GPU est naturellement devenu une solution adéquate dans ce domaine. Cette popularité se justifie par l'ensemble des Domain Specific Language (DSL) et des librairies de traitement d'image spécialisées pour GPU disponibles.

1.5.1 OpenCV

Open Computer Vision (OpenCV) est une librairie *open source* de traitement graphique couramment employée dans le milieu industriel. Elle inclut nativement le support des GPUs au moyen d'OpenCL grâce à la surcharge de ses fonctions originales. En parallèle, il existe un module officiel au sein de la librairie intitulé *GPU Module* et permettant d'employer les GPUs. Cependant, en raison de sa spécificité pour les architectures Nvidia, celui-ci sera renommé *CUDA Module* à partir de la version 3.0 d'OpenCV. L'ensemble des fonctions originales surchargées pour OpenCL n'est cependant pas identique à celui des fonctions contenues dans le *CUDA Module*. En d'autres termes, certaines fonctions sont spécifiques à l'une ou l'autre des deux solutions.

1.5.2 GpuCV

En revenant aux origines de l'ère du GPGPU, GpuCV [58, 15] est une surcouche d'OpenCV. Cette librairie offre une surdéfinition de certaines fonctions standards d'OpenCV pour pouvoir s'interfacer avec des GPUs au moyen d'OpenGL avec GLSL ou de CUDA. Les transferts de données entre la mémoire du CPU et du GPU sont notamment gérés automatiquement. Le projet ne semble cependant plus actif, ce qui s'explique par l'intégration du support des GPUs dans OpenCV.

1.5.3 CUDA NPP

Nvidia propose aussi sa propre librairie intitulée *CUDA Nvidia Performance Primitive (NPP)*. Celle-ci fournit un ensemble de fonctions et de primitives pour le traitement du signal mais aussi pour le traitement d'images avec notamment des méthodes de transformation linéaire, des opérations morphologiques, des fonctions de filtrage, de statistique ou encore de changement d'espace colorimétrique.

1.5.4 ArrayFire

ArrayFire est une libraire *open source* spécialisée pour GPU fournissant des fonctions mathématiques, de traitement du signal et de traitement d'images. Elle permet notamment de programmer les GPUs au moyen d'OpenCL et de CUDA. Cette librairie est à l'initiative de la Defense Advanced Research Projects Agency (DARPA) ce qui explique son utilisation dans le milieu de la défense américaine.

1.5.5 Intel IPL

Intel Image Processing Library (Intel IPL) [78] est une librairie développée par Intel. Celle-ci a servi de base pour le développement d'OpenCV. On retrouve notamment de nombreuses fonctions au sein d'OpenCV dont le nom commence par *IPL*. Intel IPL fournissait à l'origine des primitives adaptées au traitement d'images. Aujourd'hui, Intel Integrated Performance Primitive (Intel IPP) [4] succède à Intel IPL. Cette transformation a été adressée dans le but d'étendre les domaines d'applications d'Intel IPL mais aussi pour améliorer ses performances calculatoires en employant des instructions vectorielles telles

que les instructions AVX 512. Cette librairie est couramment utilisée avec l'architecture Xeon Phi d'Intel.

1.5.6 CLIPP

OpenCL Image Processing Primitives (OpenCLIPP) [13] est une librairie basée sur OpenCL auquel sont ajoutées des primitives de traitement d'images sur le même principe qu'Intel IPP ou CUDA NPP.

1.5.7 Matlab Parallel Computing Toolbox

Du côté de Matlab, la *Parallel Computing Toolbox*[8] est une librairie commerciale officielle developpée par Mathworks. Elle permet d'utiliser dans Matlab, les GPU Nvidia à partir de la génération *Fermi*. Les flottants double précision sont supportés ainsi que la gamme Nvidia Tesla spécialisée pour le calcul haute performance. De plus, les clusters de calculs ainsi que les architectures à GPU multiples sont adressables. Une série de fonctions accélérée par le GPU est proposée par cette librairie. Outre les fonctionnalités proposées spécifiques à l'analyse du signal, nous y retrouvons des fonctions applicables au traitement d'images tels que les filtres, les transformées de Fourrier ou encore des applications linéaires. La parallélisation se fait sur les données des tableaux passés en paramètre des fonctions concernées. Enfin cette librairie Matlab accepte les noyaux de calculs au format PTX générés par le compilateur CUDA.

1.5.8 DSLs de traitement d'images

Dans le domaine des DSL, PolyMage [107, 80] ou encore Freia [26, 32, 70] permettent d'utiliser les GPUs au moyen d'un langage dédié. Cette approche nécessite un apprentissage supplémentaire mais en contrepartie permet d'obtenir une meilleure qualité de portage sur GPU.

1.5.9 OpenVX

OpenVX [6] est un standard d'implémentation défini par le consortium Khronos, comme OpenGL ou OpenCL. Celui-ci est adapté au traitement d'images sur architecture accélératrice dont le GPU fait partie. On retrouve ainsi plusieurs implémentations en fonction des fabricants comme VisionWorks chez Nvidia. L'API d'OpenVX fournit un ensemble de fonctions et de primitives répondant à de nombreux besoins du traitement d'images. Ces fonctions ont pour avantage d'être optimisées dans le cadre de chaque implémentation spécifique à une architecture donnée. Celles-ci sont représentées par des nœuds au sein d'un graphe dont les arrêtes représentent les flots de données. Cette approche de programmation par graphe de flot de données permet de simplifier la conception d'algorithmes de traitement d'images à forte complexité spatiale pour se concentrer principalement sur l'agencement des différents modules. Enfin, l'objectif d'OpenVX est d'obtenir une portabilité de la mise en œuvre des algorithmes mais aussi de leurs performances entre différentes architectures. Il est à noter qu'une version *safety critical* sous la dénomination OpenVX SC est disponible pour les systèmes critiques utilisés notamment dans le secteur de l'aérospatial. L'ensemble de ces caractéristiques explique l'intérêt grandissant pour OpenVX dans le milieu industriel aujourd'hui. Cependant, nous noterons que les phases d'optimisations dans le graphe de flot de données demeurent manuelles et restent donc à la charge de l'utilisateur.

1.5.10 Conclusion

La plupart des algorithmes de traitement d'images, que nous avons pu étudier, utilisaient OpenCV ou Matlab. Notre choix, dans le cadre de cette thèse se porte sur l'utilisation de la librairie OpenCV.

Notre décision a été influencée par plusieurs points. Tout d'abord, Matlab a une approche plus orientée "prototypage". Notre volonté dans le cadre de cette méthodologie est d'avoir un code déployable sur des solutions GPU variées et notamment pour un usage embarqué. Les solutions Tegra de Nvidia, intègre justement le support natif de la librairie OpenCV pour les applications de traitement d'images. OpenCV étant *open source* et utilisant les langages C et C++, son intégration sur la plupart des plateformes de calculs s'en retrouve facilitée. Enfin, OpenCV est une librairie plus spécialisée pour le traitement d'images.

1.6 Conclusion

L'architecture des GPGPUs a fortement évolué durant ces dix dernières années. D'une simple approche vectorielle de type SIMD, celle-ci a évolué vers une architecture à plusieurs niveaux hierarchiques de type SIMT. L'arrivée de nouveaux éléments architecturaux tels que les mémoires caches, les transferts asynchrones, l'adressage unifié des espaces mémoires ou encore le parallélisme dynamique sont quelques exemples qui ont permis d'augmenter considérablement les performances des GPUs. En contre-partie, ces gains tendent à complexifier la mise en œuvre de cette architecture.

Actuellement, l'acteur le plus dynamique au niveau industriel reste Nvidia. En comparaison, les solutions GPU d'Intel ne fournissent pas une capacité calculatoire suffisante. AMD pour sa part présente une gamme de produits APU intéressante pour de l'embarqué. Cependant, celle-ci reste globalement trop orientée vers le marché grand publique du jeu vidéo. Nvidia, à contrario, adresse aujourd'hui de nombreux marchés. Sa gamme de produits Tegra par exemple vise clairement le marché de l'embarqué. Le Nvidia Drive, dérivé du Tegra, présente même un niveau d'industrialisation fort en adressant en particulier le marché de l'automotive et ses normes afférentes. Le domaine du calcul scientifique est de son côté addressé via les produits Tesla que l'on retrouve parmi plusieurs des meilleurs calculateurs du Top500 [10]. En novembre 2017, les solutions RadeonPro et FirePro d'AMD ne connaissent pas la même popularité dans ce classement.

Dans l'optique d'appliquer les résultats de cette thèse dans le domaine industriel, la suite de ce manuscrit de thèse est donc spécialisée pour les solutions de Nvidia.

Chapitre 2

État de l'art : placement sur GPU

Sommaire

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

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

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

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

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

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

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

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

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

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

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

2.1 Transformation par annotation de directives

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

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

2.1.1 HMPP

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

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

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

2.1.2 hiCUDA

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

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

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

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

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

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

2.1.3 OpenMP

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

2.1.4 "OpenMP C to CUDA"

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

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

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

2.1.5 OpenMPC

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

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

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

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

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

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

2.1.6 Mint

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

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

mint parallel pour la délimitation de régions de code parallèles,

mint for pour la parallélisation de nids de boucles,

mint barrier pour la synchronisation des *threads* parallélisés,

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

mint copy pour les transferts mémoires.

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

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

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

2.1.7 GPSME

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

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

2.1.8 OpenACC

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

2.1.9 PGI Accelerator

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

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

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

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

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

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

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

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

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

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

2.2 Transformation automatique de code

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

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

2.2.1 C-to-CUDA

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

2.2.2 PIPS et Par4All

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

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

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

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

2.2.3 PPCG

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

La méthodologie de PPCG se décompose en cinq étapes :

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

2.2.4 R-Stream

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

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

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

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

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

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

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

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

2.2.5 Togpu

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

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

2.3 Squelettes algorithmiques

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

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

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

2.3.1 SkePU/SkePU2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.2 SkelCL

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

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

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

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

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

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

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

2.3.3 Thrust

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

Sorting pour classer des données selon un ordonnanceur donné,

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

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

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

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

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

2.3.4 Bones

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Optimiseurs GPU

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

2.4.1 CUDA-Lite

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

uniquement que la mémoire globale au sein des GPUs. *CUDA-Lite* optimise alors les accès mémoire dans le but d'augmenter la bande passante et de réduire la latence d'accès aux données. Pour cela, une attention particulière est portée à la coalescence des appels mémoires. Le *tiling*, en tant que transformation de boucles, est alors employé pour améliorer la localité des accès mémoire des appels en écriture et en lecture. De même, une transformation des accès en mémoire globale vers des *shared memories* est utilisée pour réduire la pénalité des accès non coalescents. Cette dernière transformation reste pertinente, même s'il n'y a pas de réutilisation des données au sein du programme CUDA du fait des temps d'accès réduits de la *shared memory*.

Il est à noter que l'architecture des GPUs a évolué depuis cette étude avec notamment l'adjonction de mémoires caches sur la mémoire globale comme décrit dans la section 1.3. Les accès mémoire non coalescents ont ainsi un impact beaucoup plus limité aujourd'hui sur les GPUs. Enfin, *CUDA-Lite*, utilisant en entrée un code déjà porté en CUDA, est plus un outil d'optimisation pouvant être incorporé dans un compilateur qu'un système de placement sur GPU.

2.4.2 Optimiseur de placement de code GPU

Yang *et al.*[168] ont élaboré un compilateur sans nom, améliorant la prise en compte des hiérarchies mémoire et la gestion du parallélisme des GPUs. Leur compilateur traite un code CUDA non optimisé en entrée. Cette caractéristique se traduit par des accès exclusivement en mémoire globale ainsi qu'une distribution simplifiée des *threads* avec un unique *thread* par *block*. Leur compilateur présente de nombreuses similitudes fonctionnelles avec CUDA-Lite décrit dans la section 2.4.1. Cependant, dans leur cas, l'utilisation de directives annotées dans le code source original demeure optionnelle. Enfin, nous noterons que leur méthode de recherche d'une solution optimale de placement présente la particularité de ne pas utiliser de modèle polyédrique.

Le processus de compilation passe par les étapes suivantes :

1. vectorisation des accès mémoire contiguës au sein d'un *thread* pour améliorer la bande passante mémoire,
2. analyse de la coalescence des accès en mémoire globale pour chaque *block* de *threads*,
3. placement des accès mémoire redondants et non-coalescents en *shared memory*,
4. analyse des dépendances et du partage de données entre *threads* dans un *block*,
5. transformation par fusion de *blocks*[6] et de *threads*[7] pour améliorer le parallélisme et la réutilisation des données partagées,
6. réordonnancement des instructions d'un *kernel* pour améliorer le masquage des temps de communications pendant les temps de calcul,
7. et enfin, élimination des conflits d'accès aux zones mémoire identiques entre *blocks* de *threads*.

Dans le cadre de l'optimisation de code CUDA, la solution de Yang *et al.* effectue aussi de la spécialisation pour une architecture donnée en prenant en considération la taille des *shared memory*, la quantité de registres disponibles, ainsi que le nombre d'unités de calcul par cluster. Enfin, la validation de leur framework de compilation a été effectuée par comparaison des performances du temps d'exécution de leurs résultats avec ceux de CUBLAS2.2 pour un ensemble de 10 algorithmes. Leurs performances sont au final similaires voire supérieures.

6. La fusion de *block* revient à faire du *tiling* de boucles
7. La fusion de *threads* revient à faire de l'*unrolling* de boucles

2.4.3 GPUCC

GPUCC [167] est un compilateur open source basé sur LLVM [86]. Clang [85] assure le support de code C++ et C. GPUCC se présente comme une alternative intégralement open source par rapport à NVCC. Wu *et al.* s'intéressent aux problématiques de compilations séparées entre le code CUDA et le code C ou C++ de l'hôte. Ils ont notamment étudié une solution fondée sur les représentations intermédiaires de LLVM et y apportent une étape d'optimisations avant la génération du binaire. Ces optimisations correspondent au déroulage de boucles, à l'*inlining* de fonctions, à l'inférence d'espace mémoire, à l'analyse d'alias mémoire ou encore à l'optimisation des calculs arithmétiques de scalaires par factorisation des éléments communs d'équations mathématiques.

GPUCC ne permet pas de faire du placement assisté sur GPU, car il s'agit d'un compilateur source vers cible. De plus, la génération du code CUDA reste à la charge du développeur. Enfin, les optimisations citées sont effectuées au niveau de l'utilisation de l'ISA PTX pour l'architecture ciblée.

2.5 Conclusion

Le tableau 2.1 donne, pour chaque solution présentée dans cet état de l'art, les entrées et sorties ainsi que les *frameworks* utilisés pour le placement sur GPU. Nous avons ainsi détaillé trois approches de placement. Celle des transformations par directives annotées dans le code source au cours de la section 2.1, celle des transformations automatiques de code par compilateur dans la section 2.2 et enfin l'utilisation des squelettes algorithmiques spécialisés pour GPU dans la section 2.3. Enfin, nous avons abordé dans la section 2.4 les optimiseurs dédiés à l'amélioration du placement sur GPU.

Notons qu'un ensemble varié de *frameworks* peuvent être adaptés à l'architecture spécifique des GPUs. Dans le cadre des transformations automatiques de code, les solutions de placement impliquent globalement l'utilisation d'une représentation intermédiaire du code source original. Celle-ci prend traditionnellement la forme d'Internal Representation (IR) pouvant être complétée par une abstraction polyédrique pour l'analyse des tableaux. L'utilisation de représentations polyédriques a d'ailleurs pour avantage de pouvoir traiter un nombre de contraintes élevé définissant l'ensemble dans lequel une solution de placement optimale sera recherchée. Cette solution maximise alors l'adéquation des contraintes d'une architecture ciblée avec l'algorithme spécifié. C'est la raison pour laquelle on retrouve le modèle polyédrique sur la plupart des optimiseurs de placement. Cependant, comme nous avons pu le voir, ce modèle souffre de limitations d'usage pour certains algorithmes.

Les solutions de placement sur GPU sont ainsi presque toujours limitées à des applications intra-procédurales, pour des SCOPs à contraintes affines, identifiés manuellement par le développeur.

Enfin, l'architecture matérielle des GPUs a fortement évolué depuis les prémices des solutions de cet état de l'art. Ainsi, l'apparition de niveaux de caches mémoire, par exemple, rend obsolète certaines optimisations misant sur l'utilisation des *shared memories*. Concernant la cohabitation des architectures CPU et GPU, certaines solutions de minimisation des transferts mémoire peuvent aussi être remises en cause. Les transferts asynchrones de données en sont la motivation, tout comme l'unification dans un espace unique et partagé des mémoires dédiées pour certaines architectures basse consommation. De plus, l'utilisation de composants tels que les canaux de transfert de données pour *textures* et *surfaces* ou encore la concurrence de *kernels* semblent être inexploités par tous les outils de cet état de l'art. Ces spécificités architecturales ont pourtant été ajoutées dans le but d'améliorer

les performances des GPUs. Il est donc essentiel de les prendre en compte au sein des méthodes de placement pour se rapprocher de la *peak performance* des GPUs.

	entrée	framework	sortie
HMPP	Fortran/C	HMPP	CUDA/OpenCL
hiCUDA	C/C++	Open64	CUDA
OpenMP	Fortran/C/C++		binaire CUDA
OpenMP C to Cuda	OpenMP	OMPi	CUDA
OpenMPC	OpenMP	Cetus	CUDA
Mint	C	ROSE	CUDA
GPSME	C/C++	ROSE	CUDA/OpenCL
OpenACC	Fortran/C/C++		binaire CUDA
PGI Accelerator	Fortran/C/C++	PGI	CUDA
C-to-CUDA	C	Pluto	CUDA
PIPS / Par4All	Fortran/C	PIPS	CUDA/OpenCL
PPCG	C	PET+ISL	CUDA/OpenCL
R-Stream	Fortran/C		CUDA/OpenCL
Togpu	C++	Clang/LLVM	CUDA
SkePU/SkePU2	C/C++	Clang/LLVM	CUDA/OpenCL
SkelCL	C		OpenCL
Thrust	C/C++		CUDA
Bones	C	CAST	CUDA/OpenCL
CUDA-Lite	CUDA		CUDA
Yang *et al.*[168]	CUDA		CUDA
gpucc	CUDA	Clang/LLVM	binaire CUDA

TABLE 2.1 – Tableau récapitulatif des solutions de placement pour GPU présentées dans ce chapitre. Pour certaines solutions, les informations et capacités relatives à l'utilisation d'architectures autres que les GPUs ne sont pas incluses dans ce tableau récapitulatif.

Chapitre 3

Méthodologie de placement sur processeur graphique

Sommaire

L'ensemble des solutions de placement sur GPU présentées dans le chapitre 2 démontre la complexité de la programmation des architectures GPU. Cette complexité est, d'une part, liée à l'aspect non trivial du parallélisme et, d'autre part, à la spécificité architecturale des GPUs, employant massivement le parallélisme selon une classification hybride MIMD/SIMD. Parmi les quatre grandes approches abordés dans l'état de l'art, l'exploitation de trois d'entre elles à partir d'un algorithme séquentiel n'est pas immédiate.

L'utilisation des DSLs nécessite une compréhension avancée de l'algorithme étudié afin que celui-ci puisse être entièrement réécrit selon les contraintes d'une nouvelle architecture. En contrepartie, le placement est supposé être qualitatif.

Pour les squelettes algorithmiques, la première problématique à lever, avant toute utilisation, est l'identification des classes algorithmiques employées en fonction de la collection de squelettes considérée. Si l'aspect fonctionnel de l'application ciblée est moins étudié ici, il est tout de même nécessaire de définir la corrélation idéale entre un ensemble de *patterns* de programmation spécifiques à un domaine et leurs squelettes adaptés.

Enfin, dans le cas des transformations par directives, une analyse du code source est nécessaire afin de définir non seulement la légalité des transformations appliquées mais aussi la sélection des directives les plus adaptées.

Le quatrième type d'approche correspond aux transformations automatiques de code par un compilateur. Cette démarche est globalement plus directe en requérant pour entrée, uniquement le code source original de l'application concernée. Les étapes d'analyses et de transformations sont alors effectuées par le compilateur qui génère en sortie le code résultant. Nous noterons tout de même que certaines solutions comme PPCG nécessitent une phase d'analyse minimale de la part du programmeur, afin de déterminer les portions de code étudiées par le compilateur. De plus, la variété de langages existants, les paradigmes de programmation, l'arithmétique des pointeurs, les analyses interprocédurales ou encore les limitations aux transformations affines, sont quelques écueils couramment rencontrés qui complexifient grandement le travail à accomplir non seulement par les compilateurs mais surtout par leurs concepteurs s'ils souhaitent qu'un code efficace soit produit.

La démarche présentée dans ce chapitre s'appuie sur les diverses solutions employées dans l'état de l'art pour définir une méthodologie globale de portage d'algorithmes séquentiels sur GPU. Toutes les étapes de la méthodologie présentée pourraient être intégrées dans un compilateur et optimiseur pour GPU.

Ce chapitre est ainsi consacré à la problématique du portage d'algorithmes sur GPU. Pour cela une succession de quatre principales étapes, visibles dans la figure 3.1, a été définie en s'appuyant sur l'approche des transformations automatisées de code par compilateur.

En premier lieu, les phases d'analyses de code sont développées au sein des sections 3.1 et 3.2. L'ensemble des informations à collecter pour l'orientation des choix méthodologiques y sont détaillées.

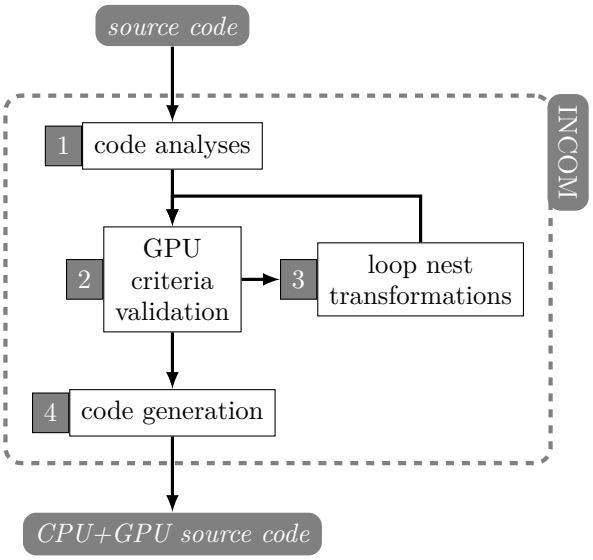


FIGURE 3.1 – Vue macroscopique de la méthodologie de placement d'algorithmes sur architecture hybride CPU et GPU

Dans la section 3.3, l'identification des portions de code adaptées à un portage sur GPU est abordée. Pour illustrer notre méthode, les critères de placement spécifiés pour les architectures Nvidia sont utilisés. Les solutions GPU du fabricant AMD étant conceptuellement assez proches, les différences devraient être tout au plus limitées au paramétrage de ces mêmes critères. Concernant les portions de code non retenues pour le placement, un ensemble de transformations de code a été défini dans le but d'améliorer la quantité de code légalement plaçable sur GPU. Ce sujet est abordé dans la section 3.4

La section 3.5 aborde les aspects de spécialisation de code, indispensables au placement sur GPU. Le processus de génération de code est présenté dans la section 3.6.

Enfin, la dernière étape de cette méthodologie, confirmant la solution de placement, est présentée dans la section 3.7.

À l'issue de ce chapitre, les portions de code placées sur GPU sont assurées d'être fonctionnelles. Cependant, le code obtenu ne représente qu'un cas sur l'ensemble des placements possibles. La «quête du graal» débute alors ici avec la recherche du placement optimisant un ou plusieurs objectifs tel que le temps d'exécution globale de l'application. Les optimisations et spécialisations de code, abordées dans le chapitre 5, ont été introduites dans ce sens.

3.1 Analyses de code statique

La première étape consiste à effectuer un ensemble d'analyses sur le code source choisi, afin de recueillir les informations permettant d'orienter la stratégie de placement. Ces analyses sont regroupées en deux catégories : les analyses statiques et dynamiques. Dans le premier cas, l'analyse porte uniquement sur le code source de l'application et permet d'extraire des informations qui sont toujours vraies quelque soit les données en entrée du programme. Dans le second cas, détaillé dans la section 3.2, l'analyse porte sur une, voire plusieurs, instances d'exécution du programme. Cette dernière complète les informations collectées par l'analyse statique avec des informations connues uniquement lors de l'exécution du programme.

L'analyse de code statique prend place après les phases d'analyse lexicale puis syntaxique. De ce fait, nous n'avons pas de présupposé quant au langage de programmation utilisé, du moment que celui-ci reste compatible avec les API de mise en œuvre pour GPU énumérées au chapitre 1. Ces deux phases pourront être réalisées automatiquement par l'utilisation d'un *parser* de code tel que ceux de PIPS ou de LLVM [86, 85] par exemple. Nous utiliserons cependant, dans la suite du manuscrit, les langages C ou C++ à des fins d'illustration. La structure du code étant supposée connue, nous pouvons alors procéder aux diverses analyses statiques.

La figure 3.2 donne un aperçu de l'ensemble des analyses statiques appliquées sur le code source de l'application. Dans un premier temps sont réalisées les analyses portant sur l'identification des boucles, des appels de fonction, des branchements ou encore des accès mémoire. Ensuite vient l'identification des blocs de base correspondant aux portions de code restantes. Enfin, les analyses des itérations de boucles et des accès aux espaces mémoire permettent de calculer les dépendances de données. Celles-ci donnent lieu à une classification des boucles de l'application.

Nous utilisons comme illustration le listing 3.1, qui est un extrait original de code provenant de l'algorithme *simpleflow*[1]. Ce dernier est issu du dépôt des contributions [2] à la librairie OpenCV et a servi de base d'étude pour l'élaboration de cette méthodologie car il présente de nombreuses caractéristiques intéressantes détaillées en section 4.2. Les résultats de l'étude complète du portage de cet algorithme sont disponibles dans le chapitre 4.

```
61 inline static float dist(const Vec2f& p1, const Vec2f& p2) {
62         return (p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] - p2[1]) * (p1[1] - p2[1]);
63 }
```

...

```
70 static void removeOcclusions(const Mat& flow, const Mat& flow_inv,
71                 float occ_thr, Mat& confidence) {
72         const int rows = flow.rows;
73         const int cols = flow.cols;
74         if (!confidence.data) {
75                 confidence = Mat::zeros(rows, cols, CV_32F);
76         }
77         for (int r = 0; r < rows; ++r) {
78                 for (int c = 0; c < cols; ++c) {
79                         if (dist(flow.at<Vec2f>(r, c), -flow_inv.at<Vec2f>(r, c))
80                                         > occ_thr) {
81                                 confidence.at<float>(r, c) = 0;
82                         } else {
83                                 confidence.at<float>(r, c) = 1;
84                         }
85                 }
86         }
87 }
```

Listing 3.1 – Extrait de code provenant de l'algorithme *simpleflow*

1. L'intégralité du code original est présent en annexe A

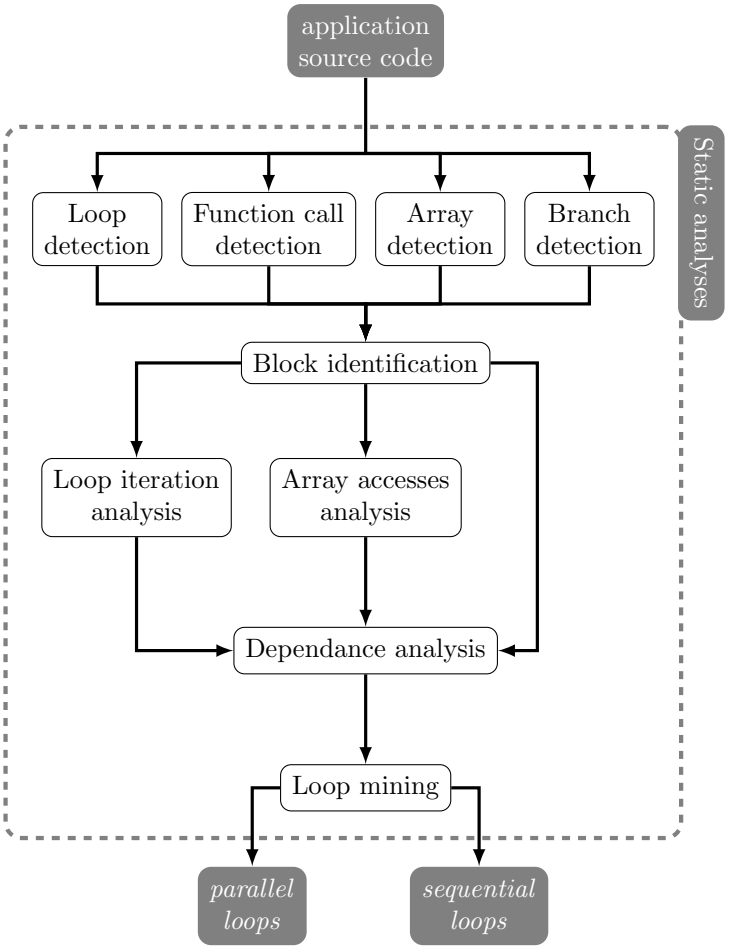


FIGURE 3.2 – Détails de la phase d'analyse de code statique

3.1.1 Identification des appels de fonction

Un programme se limite rarement à une seule et unique fonction. Il est souvent composé de fonctions multiples dès que sa taille devient conséquente. Ce découpage permet d'améliorer d'une part la compréhension du programme, mais aussi de réduire la complexité spatiale du code en factorisant au sein de fonctions certains *patterns* redondants. Du fait que notre méthodologie considère des programmes dans leur globalité et de manière interprocédurale, nous nous intéressons aux fonctions et à leurs appels qui doivent être identifiés. Notre objectif est de prendre en compte les aspects interprocéduraux au sein de nos analyses, notamment pour le calcul des dépendances ou encore pour une identification en profondeur des nids de boucles du programme.

Bien que le principe de l'*inlining* eut été une solution envisageable, elle n'a pas été retenue. Cette opération étant la contraposée du découpage par fonction, le choix de ne pas l'employer se justifie à plusieurs titres. D'une part, la taille du code source aurait dans certains cas tendance à augmenter de façon exponentielle [2]. D'autre part, la compréhension globale du programme s'en retrouverait réduite et pourrait aller à l'encontre d'un placement sur GPU par fonction choisi par le programmeur. Enfin, à partir de cette analyse des appels de fonctions, il est possible d'obtenir le graphe des appels, ou *Call Graph*, de notre programme avec le nombre d'appels effectués pour chaque fonction. L'analyse dynamique de la section 3.2 permet entre autres de traiter les cas de contrôle dynamique.

Dans le cadre du listing 3.1, nous observons à la ligne 79 l'appel à la fonction *dist* faisant référence à la fonction de même nom et de même signature détaillée en lignes 61 à 63. Les appels répétés à la fonction *at* aux lignes 79, 81 et 83 en revanche font référence à une méthode d'objet provenant de la librairie OpenCV. Enfin, la fonction *zeros* à la ligne 75 présente des caractéristiques similaires à celles que nous venons de décrire pour la fonction *at*. Nous reviendrons sur ces cas particuliers dans la section 3.1.3.

3.1.2 Identification des boucles

L'identification des boucles revêt une forte importance dans cette méthodologie car l'architecture des GPUs est particulièrement adaptée au parallélisme de données. Ce type de parallélisme est très courant dans les applications de traitement d'images.

En fonction du langage de programmation utilisé, mais aussi au sein d'un même langage, les paradigmes de boucles peuvent avoir des structures syntaxiques variées telles que *for*, *while* ou encore *do/while*. Cependant un état d'amorce, une condition de fin et un paramètre d'incrément sont les trois éléments caractéristiques de toute boucle, déterminant l'ensemble d'itérations. Ce sujet sera traité plus en détails dans la section 3.1.7.

Les fonctions récursives peuvent être considérées comme des boucles potentielles à condition que les trois éléments caractéristiques précédents puissent être déterminés. Le choix d'écarter l'*inlining* évoqué dans la section 3.1.1 prend ici tout son sens. En effet, dans le cas où ces paramètres ne pourraient être définis, la profondeur d'*inlining* serait aussi indéterminée.

Dans le cadre de notre méthodologie, chaque entête de boucle est affectée d'un identifiant unique de la forme l_n. Ici l signifie qu'il s'agit d'une boucle [3] et n en est son numéro unique incrémenté selon l'ordre lexicographique et l'analyse interprocédurale du code source analysé.

Dans le listing 3.1, l'exemple est composé de deux boucles *for* parfaitement imbriquées (lignes 77 et 78) et recevant respectivement les identifiants l_0 et l_1. Si la fonction *dist*

2. Notre méthodologie n'interdit pas les appels de fonctions récursives dans certaines parties du code.
3. Le choix de la lettre l vient du terme anglais *loop*

intégrait une boucle, celle-ci recevrait l'identifiant l_3. Du fait qu'une fonction puisse être appelée en plusieurs endroits du code, une boucle peut ainsi recevoir plusieurs identifiants. En revanche un identifiant ne peut être affecté qu'à une unique boucle. Ce processus d'attribution des identifiants à un ensemble de boucles est donc surjectif.

3.1.3 Identification des accès aux espaces mémoire

Si les boucles précédemment identifiées permettent de parcourir les espaces d'itérations, les accès mémoire décrivent pour leur part la méthode d'accès aux données.

Ces espaces mémoire peuvent être définis de deux manières. Dans le cas de l'allocation statique, l'espace mémoire est défini au lancement de l'application et ne peut être modifié. Dans le cas de l'allocation dynamique, l'espace mémoire est au contraire défini lors de l'exécution du programme et peut-être modifié, voire libéré.

De plus les espaces mémoires sont exploitables de différentes façons. Dans le cas du langage C, par exemple, cet accès s'effectue par l'utilisation de tableaux ou au moyen de pointeurs. Pour les langages basés sur le paradigme de programmation orientée objet, tels que java ou C++, un espace mémoire peut-être encapsulé voire masqué en tant que donnée privée au sein d'un objet. C'est le cas par exemple avec la librairie OpenCV basée sur *C++* qui met à disposition l'objet de type *Mat*. Dans le listing 3.1, l'accès se fait alors par l'utilisation d'une méthode *at* interne à l'objet *Mat*. Cette méthode prend en paramètre un ensemble de coordonnées et retourne l'élément correspondant au sein de son espace mémoire. Trois exemples sont illustrés aux lignes 79, 81 et 83 pour l'instance d'objet *confidence*. L'allocation et l'initialisation de celui-ci sont visibles à la ligne 75 au moyen du constructeur *zeros* qui a pour fonction supplémentaire d'initialiser le tableau avec la valeur 0. Ce genre de constructeur est problématique pour les analyseurs de codes qui ont tendance à ne pas pouvoir identifier l'espace mémoire encapsulé dans ce genre d'objet. Une méthode couramment employée est la réécriture du code source au moyen de tableaux standards afin de pouvoir obtenir des informations pertinentes de la part des analyseurs de code. L'autre solution consiste à prendre en compte la syntaxe de la librairie utilisée. Dans le cadre du listing 3.1, cela revient à analyser sémantiquement le contenu de la librairie OpenCV et ainsi considérer lors de l'analyse l'objet *Mat* comme un tableau.

3.1.4 Identification des branchements

Un branchement constitue une rupture dans la séquence du flot d'exécution d'un code source. Plusieurs chemins peuvent être empruntés de façon prédictible ou non, et l'on parle alors respectivement de code à contrôle statique ou dynamique. Ces ruptures dans le déroulement du programme ont pour effet d'induire des modifications sur le comportement des boucles, sur l'effet des dépendances mais aussi sur les accès aux espaces mémoire. Parmi les instructions de branchement des langages C et C++, on trouve les cas classiques $if/then/else$ et $switch/case$. Intégré dans une boucle, l'instruction de branchement de type *break* aura pour effet immédiat de stopper l'exécution de la boucle alors que l'instruction *continue* ignorera l'exécution de l'instance de boucle courante. Enfin, l'instruction de branchement *return* illustrée dans le listing 3.1 aura pour effet de sortir immédiatement du corps de la fonction qui l'intègre.

Dans notre méthodologie, les conditionnelles de branchement sont annotées par un identifieur de la forme c_n. Ici, c représente le type d'élément et n est son numéro unique incrémenté selon l'ordre lexicographique du code analysé de manière interprocédurale. Comme pour l'identification des boucles, ce processus de labellisation est surjectif.

Dans le listing 3.1, l'exemple contient deux cas de branchement. Le premier à la ligne 74 correspond à l'identifiant c_0. Celui-ci autorise deux chemins d'exécution l'un passant à la ligne 77, l'autre passant préalablement par la ligne 75. Le second à la ligne 79 correspond à l'identifiant c_1. Ce dernier autorise deux chemins d'exécution, permettant d'exécuter respectivement les instructions de la ligne 81 ou de la ligne 83.

3.1.5 Identification des blocs de base

Une fois les appels de fonctions, les boucles et les branchements identifiés, les fragments de code résultants sont considérés comme des blocs de base. Ceux-ci seront annotés au moyen de l'identifiant b_n avec b correspondant à l'identifiant de bloc de base et n au numéro unique du bloc selon l'ordre lexicographique du code analysé de manière interprocédurale. Ici encore, le processus de labellisation est surjectif.

3.1.6 Construction de la *représentation spinale* du programme

L'ensemble des représentations isolées (AST, graphe d'appels des fonction, "*interprocedural control flow graph*") que nous avons pu évaluer pour cette méthodologie étaient inexploitables dés lors que la complexité spatiale du programme devenait trop importante. La problématique repose sur la masse importante d'informations à traiter et à représenter. Celles-ci viennent dans le meilleur cas surcharger la représentation attendue et nuit ainsi à l'interprétation de cette dernière. Dans le pire cas, la simplification de la représentation ne permet pas de prendre de décision pertinente. Nous avons alors défini une nouvelle forme de représentation graphique de programmes afin de remédier à ce problème.

Les objectifs de cette nouvelle représentation sont :

1. d'avoir une vue hiérarchique du programme permettant
   (a) d'identifier l'adéquation entre les *patterns* de boucles et les contraintes architecturales de l'accélérateur,
   (b) d'identifier les transformations adaptées,
2. d'avoir une vue interprocédurale du programme, permettant de passer au-delà des frontières de procédures et d'affecter des transformations plus transverses telle que la fusion interprocédurale de nids de boucles,
3. d'y associer le flot de données et les dépendances afin
   (a) d'ajuster la granularité des calculs, le placement en mémoire de données,
   (b) d'identifier et d'optimiser les communications entre hôte et accélérateur,
   (c) d'identifier les associations entre les domaines d'itérations et les espaces de données considérés (régions de tableaux),
   (d) d'identifier les données importées et exportées par une fonction ou un ensemble de boucles,
   (e) d'identifier d'éventuelles mises à jour de variables inutiles,
   (f) d'identifier le placement concurrentiel de nids de boucles ou de parties de code indépendants.

Après que toutes les zones de code de l'algorithme aient été annotées, il est possible de construire cette représentation hiérarchique de notre programme fondée sur un graphe de contrôle interprocédural auquel est associé le flot d'exécution du programme. Nous appelons cette représentation graphique, *représentation spinale*, en référence à la représentation des connections nerveuses de la moelle spinale dans la colonne vertébrale

humaine. La figure 3.3 donne un aperçu de la représentation spinale simplifiée de la fonction *removeOcclusions*, détaillée dans le listing 3.1. Cette représentation est considérée comme simplifiée et sera complétée dans la section 3.1.11 avec les résultats obtenus lors des analyses restantes.

Dans cette représentation, l'axe horizontal représente le déroulement allant de la gauche vers la droite de l'algorithme étudié. Il respecte l'ordre lexicographique basé sur la trace d'exécution des éléments précédemment identifiés. L'axe vertical correspond aux différents niveaux de profondeur de notre représentation. Ainsi, la première ligne de niveau 0 représente le corps de la fonction principale. Chaque boucle, ainsi que chaque cas de branchement, implique un degré de profondeur supplémentaire correspondant respectivement au contenu du corps de la boucle et de chaque cas de branchement.

Le déroulement de l'algorithme est représenté par une ligne continue parcourant les boucles, les branchements et les blocs de base précédemment identifiés. L'ensemble des éléments parcourus représentent ainsi les nœuds de la représentation tandis que les lignes continues constituent l'ensemble des traces possibles d'exécution. Lorsque cette ligne se termine par un nœud final, le déroulement de l'algorithme reprend au niveau de l'entête de la boucle situé à un niveau de profondeur inférieur. Cela correspond ainsi aux cas où le corps de boucles a été entièrement déroulé ou qu'une instruction de branchement telle que *break* ou *return* a été atteinte. Au final, l'entête permettra soit d'itérer de nouveau sur le corps de boucle, soit de redescendre d'un niveau supplémentaire au sein du nid de boucles. Dans le cas où le niveau de profondeur est 0, la fin de ligne correspond à la fin du programme ou de la partie de code étudiée. Pour le cas particulier des branchements, les différents traces possibles sont reliées à la conditionnelle d'origine par une ligne pointillée. Dans un souci de lisibilité, chaque branche est alors représentée sur un unique niveau de profondeur supérieur comparé à celui de sa conditionnelle. Dans notre exemple, la boucle d'entête l_1 (*correspondant à la ligne 78 dans le listing 3.1*) se termine soit par le bloc b_3 (*l81*) soit par le bloc b_4 (*l83*) à cause du branchement de c_1 (*l79*). De plus les boucles l_0 et l_1 sont parfaitement imbriquées du fait que le corps de la boucle l_0 est uniquement composé de la boucle l_1. l_0 se termine donc de la même façon que l_1 et la suite de l'application continuerait à partir du nœud l_0. Enfin, le branchement c_0 (*l74*) permet d'atteindre le bloc b_1(*l75*) ou au contraire l'ignore. Dans les deux cas, l_0 sera atteint.

Certains entêtes de boucles ou de branchement peuvent être complexes. Dans ce cas de figure, l'entête est complété par un bloc de base. La connexité forte entre ces deux éléments est alors symbolisé par un double trait au sein de la représentation. Ainsi, dans notre exemple, la conditionnelle c_1 fait notamment appel à la fonction *dist*. Ce déroulement a été délégué au sein du bloc b_2.

Nous complétons notre représentation grâce aux informations fournies par l'analyse interprocédurale. Les différentes fonctions parcourues sont symbolisées par un trait orange sous lequel est inscrit le nom de la fonction. Cette partie représente ainsi le graphe d'appel de l'application et la superposition des traits oranges permet à tout moment de connaître la profondeur du graphe d'appel au sein de la représentation.

3.1.7 Analyse des boucles

L'analyse présentée dans cette section permet de déterminer les caractéristiques des boucles précédemment identifiées. Nous introduisons ci-dessous un formalisme qui nous permet de considérer les boucles complexes et non normalisées classiquement rencontrées dans le cadre de programmes *C++*. Ainsi pour toute boucle l_ϵ correspond une application $u_\epsilon : \mathbb{N} \to L_\epsilon$ telle que :

— ϵ représente l'identifiant unique d'une des boucles définissant l'espace d'itération,

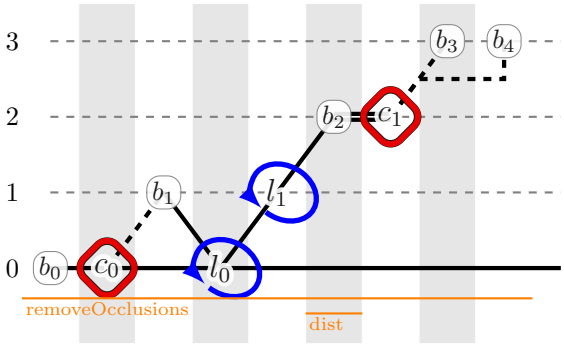


FIGURE 3.3 – Représentation spinale de la fonction *removeOcclusions*. *(version simplifiée)*

— u_ϵ est une suite numérique spécifiant l'ordre de parcours du domaine d'itération de la boucle l_ϵ,
— l'ordre de parcours est donné par la fonction itérative $\mathcal{D}_\epsilon$ telle que $u_\epsilon(n+1) = \mathcal{D}_\epsilon(u_\epsilon(n))$,
— $\forall n \in \mathbb{N}$, $u_\epsilon(n)$ est un terme de cette suite correspondant ainsi à la $n^{ième}$ itération de la boucle l_ϵ,
— $u_\epsilon(0)$ représente la première itération de la boucle,
— L_ϵ représente l'ensemble d'arrivée dans $\mathbb{N}$ ou plus généralement dans $\mathbb{Z}$, correspondant à l'ensemble des itérations de la boucle l_ϵ.

Dans le cas d'une **boucle finie**, l'application est alors spécialisée en $u_\epsilon : I_\epsilon \to L_\epsilon$ telle que :

— $I_\epsilon \subset \mathbb{N}$ correspond à l'ensemble des valeurs du compteur de la boucle ϵ,
— $|I_\epsilon|$ représente le cardinal de l'ensemble de départ, ce qui correspond au nombre d'itérations de la boucle ϵ,
— u_ϵ est bornée.

Une **boucle normalisée** présente pour sa part les caractéristiques d'une boucle finie auxquelles viennent s'ajouter :

— $I_\epsilon = L_\epsilon$,
— $u_\epsilon(0) = 0$
— $u_\epsilon(n+1) = n+1$
— $u_\epsilon = Id_I$
— u_ϵ est une application bijective,

En appliquant cette analyse à l'exemple du listing 3.1, les boucles l_0 et l_1 sont normalisées et présentent les caractéristiques suivantes :

— $u_0(0) = 0$, $u_1(0) = 0$
— $|I_0| = rows$, $|I_1| = cols$
— $u_0(n+1) = u_0(n) + 1$, $u_1(n+1) = u_1(n) + 1$

Dans le cas où les paramètres d'une **boucle imbriquée** l_m dépendent au sein d'un même nid des valeurs d'itérations d'une boucle l_q de plus faible profondeur, alors le domaine d'itération est donné par leur composée.

Enfin, une **boucle parallèle** a pour spécificité d'être libérée de toute contrainte quant-à son ordre de parcours. L'analyse de dépendances permettra à ce titre de déterminer si cette propriété est vérifiée. Ainsi, afin de pouvoir déterminer de manière indépendante $u_\epsilon(n)$ pour toute itération d'une boucle parallèle ϵ, il est nécessaire de déterminer l'application $\mathcal{I}$ telle que $u_\epsilon(n) = \mathcal{I}(u_\epsilon(0))$.

Au final, dans notre méthodologie, l'analyse de boucles donne les bornes ainsi que la taille $|I_\epsilon|$ du domaine d'itération généré par toute boucle l_ϵ. La relation résultante est

donnée par la formule 3.1.

$$I_\epsilon \xrightarrow{\mathcal{I}} L_\epsilon \tag{3.1}$$

Dans le cas où ces paramètres peuvent être définis à partir du seul code source, la boucle est alors considérée à contrôle statique et $|I_\epsilon|$ est connu. Dans le cas inverse, l'analyse dynamique présentée en section 3.2 peut apporter des informations supplémentaires. Une analyse statique pourra être employée sinon par sur-approximation au pire cas :

— en considérant la boucle comme non finie soit $I_\epsilon \in \mathbb{N}$ ou
— en contraignant $|I_\epsilon|$ à une ou plusieurs valeurs significatives données [4], spécialisant ainsi l'algorithme.

3.1.8 Analyse des fonctions d'accès mémoire

L'analyse présentée dans cette section s'applique à chaque accès d'espace mémoire précédemment identifié.

Les fonctions d'accès mémoire sont de type $\mathcal{A} : L^p \rightarrow \mathbb{N}$. L^p représente par abus de langage, le domaine d'itération généré par l'ensemble des p boucles imbriquées tel que $L^p_\epsilon = L_\epsilon \times L_{\epsilon+1} \times ... \times L_{\epsilon+p-1}$. Ces dernières constituent les dimensions utilisées pour le parcours d'un espace mémoire donné. On distingue parmi l'ensemble des fonctions d'accès mémoire, celles procédant à un accès en lecture $\mathcal{A}_R$ de celles en écriture $\mathcal{A}_W$. Cette classification est employée pour déterminer les éléments communs entre ces deux espaces lors de l'analyse des dépendances de la section 3.1.9. L'ensemble d'arrivée est quant à lui contraint à l'ensemble $\mathbb{N}$, correspondant au modèle d'adressage mono-dimensionnel conventionnellement mis à disposition pour les unités mémoire.

Cependant, certains langages comme le C ou le C++, permettent l'emploi d'accès mémoire multidimensionnels au moyen de tableaux à dimensions multiples. Il en est de même pour *opencv* grâce à l'objet *Mat*. On considère alors une fonction supplémentaire $\mathcal{L}$ de linéarisation utilisée dans la formule 3.2. L'application $\mathcal{A}$ précédemment décrite permet alors d'obtenir un accès mémoire à d dimensions à partir de l'espace d'itérations défini par l'ensemble des p boucles concernées. Cet accès mémoire à d dimensions sera ensuite transformé par linéarisation en accès mémoire mono-dimensionnel par la transformation $\mathcal{L}$.

$$L^p \xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} \mathbb{N} \tag{3.2}$$

En pratique, ces unités mémoire ayant une capacité de stockage finie, le domaine d'arrivée est majoré par la taille de la zone mémoire utilisée, elle-même inférieure à la capacité mémoire globale.

Nous introduisons maintenant les deux ensembles :

— $\mathcal{R}(S, l, i)$ qui regroupe l'ensemble des accès en lecture effectués par l'instruction S pour une itération i de la boucle l et
— $\mathcal{W}(S, l, i)$ pour les accès en écriture.

Les méthodes d'analyse de régions de tableaux [142, 37] permettent de recueillir ce type d'information. En plus des accès en lecture et en écriture, Creusillet et Irigoin [37] introduisent les notions de régions *IN* et OUT permettant de synthétiser pour un bloc d'instructions :

— les régions mémoire importées par ce bloc, qui sont accédées en lecture avant d'avoir été écrite, pour les régions *IN*,

4. Pour une application de traitement d'images, il est possible de contraindre les domaines d'itération de boucles liés au parcours d'une image selon un format standard de résolution d'image

— les régions mémoire exportées, qui sont accédées en écriture, pour les régions OUT.

Dans les cas les plus complexes, correspondants notamment à la présence de branchement au sein des nids de boucles ou encore à l'utilisation d'espaces non denses, les analyses de régions de tableaux permettent d'approximer l'espace mémoire considéré, en donnant l'enveloppe convexe de ces éléments.

Au final, l'ensemble des espaces abordé dans le cadre de cette méthodologie, est synthétisé par la formule 3.3 selon la nomenclature précédemment décrite.

$$\underbrace{I^p \xrightarrow{\mathcal{I}} L^p}_{\text{boucles}} \overbrace{\xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} M}^{\text{accès mémoire}} \text{, tel que } I^p \subset \mathbb{N}^p \text{ et } M \subset \mathbb{N}. \tag{3.3}$$

3.1.9 Analyse des dépendances

Cette section porte sur l'étape critique d'analyse des dépendances de données. Celle-ci permet notamment d'identifier les contraintes qui doivent être respectées sur l'ordre d'exécution des instructions du code et aussi de distinguer les boucles parallèles des boucles séquentielles.

Si l'analyse des dépendances sur les scalaires reste simple, ce n'est pas le cas des dépendances portant sur les tableaux.

Considérons $S1$ et $S2$, deux instructions telles que $S1$ est exécutée selon l'ordre lexicographique avant $S2$. Celles-ci ne présentent pas de dépendance si elles satisfont les conditions de Bernstein étendues aux boucles.

Il existe plusieurs méthodes permettant de vérifier les conditions de Bernstein parmi lesquelles nous pouvons citer :

— le test du Plus Grand Commun Diviseur (PGCD) décrit dans [12],
— le test de Banerji [22] spécifique au domaine d'application affine,
— le test de Fourier-Motzkin [40],
— l'Omega test [137, 138], plus précis que Fourier-Motzkin sur les solutions entières,
— la recherche par simplexe [41, 106] ou encore
— la programmation linéaire en nombres entiers [59].

Pour des questions d'efficacité et parce qu'elles sont plus appropriées à notre problème, nous utilisons pour le calcul des dépendances des approximations et les ensembles $\mathcal{R}(S, l, i)$ et $\mathcal{W}(S, l, i)$ définis précédemment.

Nous définissons les ensembles 3.4, 3.6 et les conditions 3.5.

$$\begin{cases} S_1 \prec S_2, & \\ D_a(S_1, S_2, l_\epsilon) = \cup_{i_1 \in L_\epsilon, i_2 \in L_\epsilon, i_1 < i_2} \mathcal{R}(S_1, l_\epsilon, i_1) \cap \mathcal{W}(S_2, l_\epsilon, i_2) & \text{données avec antidépendances} \\ D_f(S_1, S_2, l_\epsilon) = \cup_{i_1 \in L_\epsilon, i_2 \in L_\epsilon, i_1 < i_2} \mathcal{W}(S_1, l_\epsilon, i_1) \cap \mathcal{R}(S_2, l_\epsilon, i_2) & \text{données avec dépendances de flot} \\ D_s(S_1, S_2, l_\epsilon) = \cup_{i_1 \in L_\epsilon, i_2 \in L_\epsilon, i_1 < i_2} \mathcal{W}(S_1, l_\epsilon, i_1) \cap \mathcal{W}(S_2, l_\epsilon, i_2) & \text{données avec dépendances de sortie} \\ D(S_1, S_2, l_\epsilon) = D_a(S_1, S_2, l_\epsilon) \cup D_f(S_1, S_2, l_\epsilon) \cup D_s(S_1, S_2, l_\epsilon) & \\ DL(l_\epsilon) = \cup_{S_1, S_2 \in Body(l_\epsilon), S_1 \neq S_2} D(S_1, S_2, l_\epsilon) & \end{cases} \tag{3.4}$$

$$\begin{cases} D(S_1, S_2, l_\epsilon) \neq \varnothing & \text{si } S_1 \text{ et } S_2 \text{ présentent des dépendances portées par la boucle } l_\epsilon \\ D(S_1, S_2, l_\epsilon) = \varnothing & \text{si } S_1 \text{ et } S_2 \text{ n'ont pas de dépendantes portées par la boucle } l_\epsilon \end{cases} \tag{3.5}$$

Si $DL(l_\epsilon) = \varnothing$, alors aucune des instructions du corps de la boucle l_ϵ ne génère de dépendance portée par l_ϵ. La boucle l_ϵ peut être exécutée en parallèle.

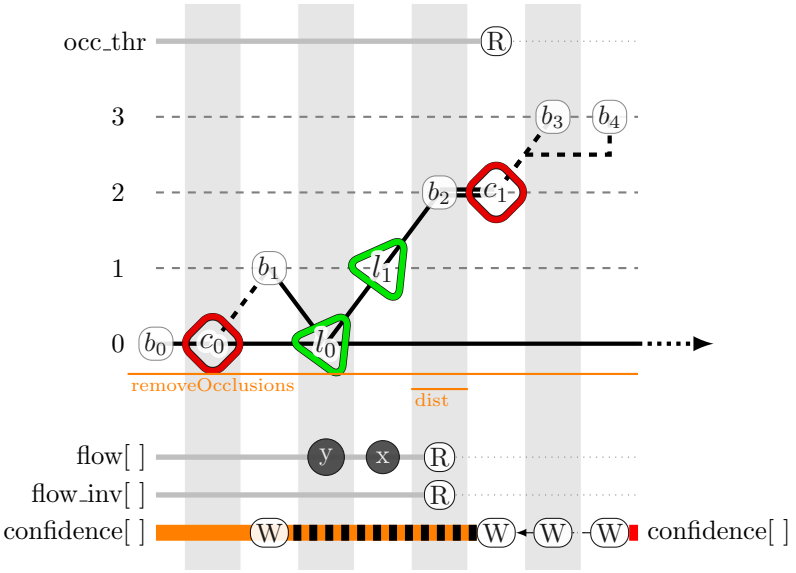


FIGURE 3.4 – Représentation spinale de la fonction *removeOcclusions*. *(version enrichie)*

Nous élargissons cette notation à un nid de boucles de profondeur p dont la première boucle est l_ϵ :

$$DN(l_\epsilon^p) = \cup_{k=0}^{p-1} DL(l_{\epsilon+k}) \tag{3.6}$$

Si $DN(l_\epsilon^p) = \varnothing$, alors les p boucles $l_\epsilon, l_{\epsilon+1}, .., l_{\epsilon+p-1}$ peuvent être exécutées en parallèle.

3.1.10 Catégorisation des boucles

À partir des résultats issus de l'analyse des dépendances décrite dans la section 3.1.9, l'ensemble des boucles est segmenté en deux catégories. On considère ainsi les boucles possédant une dépendance entre deux itérations du nid de boucles comme des **boucles séquentielles**. Tandis que les boucles ne présentant aucune dépendance entre deux itérations sont classées comme **boucles parallèles**.

3.1.11 Complétion de la *représentation spinale*

Les résultats des analyses de boucles, d'accès mémoire et de dépendances sont utilisés dans la représentation initiée dans la section 3.1.6. La figure 3.4 donne un aperçu de la représentation spinale complète. Les variables scalaires, telles que *occ_thr* dans notre exemple, sont situées dans la partie haute de la représentation tandis que les accès mémoire tels que $flow[\ ]$, $flow\_inv[\ ]$ et $confidence[\ ]$ sont représentés dans la partie basse. La représentation des dépendances est alors définie selon la convention suivante :

- Les dépendances de flot sont représentées par un trait de liaison rouge,
- les dépendances de sortie par un trait de liaison noir,
- les antidépendances par un trait de liaison orange.
- Enfin, les données accédées successivement en lecture seule sont représentées par un trait gris plus fin.

La présence de branchement au sein du code implique des variations dans les effets de dépendances. Dans l'exemple de la figure 3.4, c_0 engendre deux branchements possibles :

1. La branche contenant b_0 est parcourue avant de revenir sur la branche principale et exécuter l_0. Ce cas engendre une dépendance de sortie.

2. Le programme reste sur la branche principale et passe directement à l_0 sans exécuter b_0 avec une antidépendance héritée du code amont.

. Les deux cas de dépendance sont alors représentés par un hachurement selon le code couleur des dépendances respectives (noir et orange).

Lorsqu'une boucle permet de parcourir une des dimensions d'un espace mémoire A^d, ce dernier est marqué de son axe parcouru au niveau de la boucle en question. Dans l'exemple de la figure 3.4, l_0 et l_1 permettent de parcourir respectivement les axes y et x de l'espace mémoire $flow[\,]$.

En complément, les boucles identifiées parallèles sont à présent représentées par un triangle vert pointé vers le niveau supérieur. Les boucles séquentielles sont au contraire représentées par un triangle bleu orienté vers la droite du niveau courant.

3.2 Analyses de code dynamique

Contrairement à l'analyse de code statique, l'analyse dynamique s'appuie sur l'exécution du programme pour collecter des informations complémentaires conditionnées par les entrées. Ces dernières ont pour particularité d'être difficilement prédictibles ou encore indéterminables à la seule analyse du code source. C'est pourquoi, l'analyse de code dynamique nécessite comme pré-requis, la compilation du code source étudié afin de le rendre exécutable.

Cette étape de la méthodologie est illustrée dans la figure 3.5. Une des métriques qui nous intéresse est le temps d'exécution de certaines portions critiques du code source. Ce type d'analyses est aussi utile pour lever l'ambiguïté propre aux zones de code contenant des branchements identifiés dans la section 3.1.4 ainsi que pour déterminer précisément les domaines d'itérations des boucles à bornes dynamiques (section 3.1.7).

L'utilisation de *profilers* tels que GProf [3] ou encore Vtune [5] de la suite Intel Parallel Studio permettent de récupérer ce genre d'information. Cependant, une autre méthode consiste à instrumenter le code source au moyen de *patterns* de code particuliers, dans le but de mesurer le temps d'exécution des différentes portions de code étudiées. Ainsi, l'utilisation des compteurs de boucles permet de récupérer le nombre d'itérations des boucles à bornes dynamiques. En complément, l'usage de variables booléennes ou de compteurs de branche en guise de détecteurs de branchement permet d'identifier les portions de codes atteintes.

Cependant, sans analyse approfondie et complémentaire, les résultats obtenus sont à considérer avec prudence car, d'un jeu de données en entrée à un autre, ils peuvent être sensiblement différents, remettant ainsi en cause la qualité de la solution générée. En traitement d'images, par exemple, le contenu de deux images distinctes peut engendrer des branchements ou des domaines d'itérations de boucles différents lors de l'exécution du même programme. L'utilisation de formats et de résolutions d'images variés peuvent modifier la taille du domaine d'itérations, influençant ainsi directement le temps d'exécution de l'application.

Comme le montre la figure 3.5, nous nous intéressons au temps nécessaire à l'exécution globale de l'application. En complément de cette mesure, viennent s'ajouter les temps de parcours des boucles et des fonctions précédemment détectées dans les sections 3.1.2 et 3.1.1 lors des analyses statiques. Ces informations seront notamment utilisées plus tard dans le déroulement de la méthodologie pour alimenter la procédure de validation/invalidation de *kernels* décrite au chapitre 3.7.

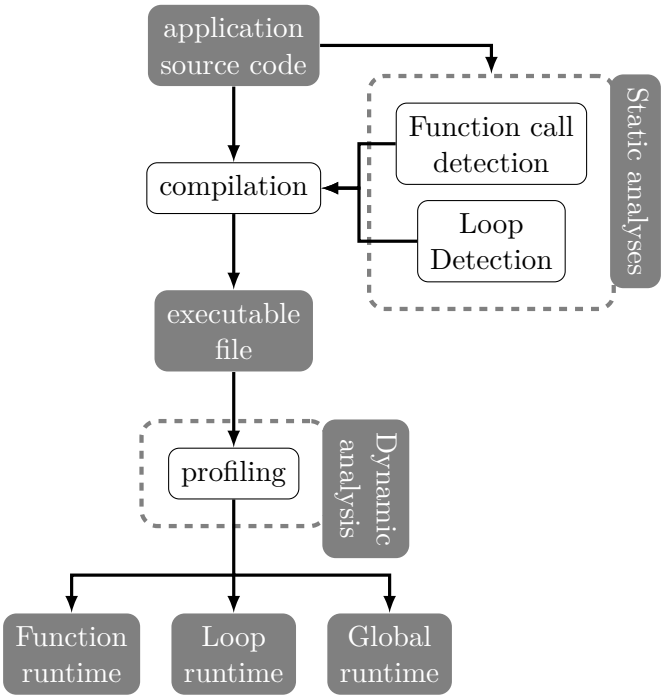


FIGURE 3.5 – Détails de la phase d'analyse de code dynamique

3.3 Conditions nécessaires au placement sur GPU

Après avoir complété l'ensemble des analyses précédemment décrites, l'objectif de cette nouvelle étape consiste à déterminer les portions de code adaptées aux spécifications architecturales des GPUs. Ces derniers étant particulièrement efficaces pour exploiter le parallélisme de données, l'approche utilisée ici va porter sur l'étude des nids de boucles et de leurs accès mémoire associés. Pour cela, trois critères, visibles dans l'encart *GPU criteria* de la figure 3.6, ont été définis afin de déterminer les portions de code adaptées pour un placement sur GPU. Ces trois critères correspondent au *loop pattern*, *loop size* et *memory size* dans la figure 3.6. Ils portent respectivement sur la structure, le domaine d'itérations et l'empreinte mémoire de chaque nid de boucle évalué. La vérification de ces critères se fait de manière séquentielle, chaque étape devant être validée avant de passer à la suivante.

Les nids de boucles ne pouvant répondre favorablement à l'ensemble de ces trois critères sont alors ignorés et maintenus sur CPU. C'est pourquoi, un ensemble de transformations de boucles a été identifié afin d'augmenter le nombre de boucles plaçables sur GPU. Nous abordons spécifiquement ce sujet dans la section 3.4.

À l'issue de ce processus de vérification, chaque ensemble de boucles répondant favorablement à la totalité de ces critères est transformé en un *kernel*. La notion de *kernel*, que nous utilisons dans la suite de ce document, correspond à un programme de taille réduite comparé à l'algorithme initial et prenant la forme d'une fonction. Son exécution par l'architecture ciblée (le GPU dans notre cas) est ordonné par le processeur hôte (le CPU) au moyen d'un appel de fonction associé à un nombre d'instances d'exécution défini. Il existe donc un lien fort entre le nombre d'itérations des boucles étudiées et le nombre d'instances d'exécution du *kernel* généré.

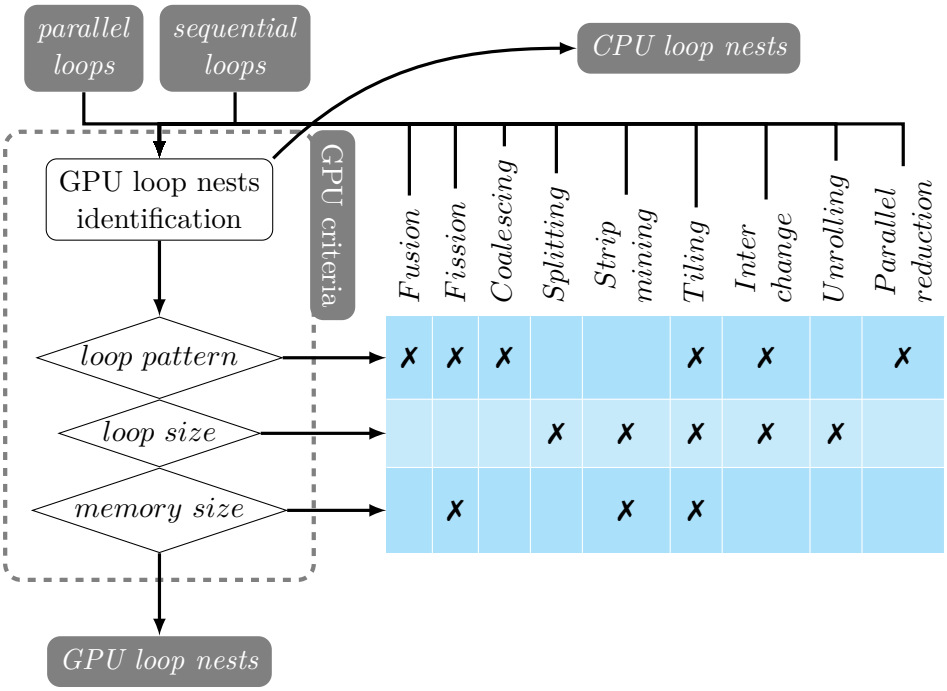


FIGURE 3.6 – Transformations de nid de boucles pour architectures SIMT

3.3.1 Critère 1 : Structure et profondeur du nid de boucles d'un *kernel*

Le premier critère de placement est constitué par l'étape *loop pattern* de la figure 3.6. Afin d'être compatible avec l'architecture spécifique des GPUs, un nid de boucles candidat doit nécessairement répondre à des caractéristiques structurelles particulières. Ainsi, en conséquence de l'architecture à double niveau décrite dans la section 1.3, l'exemple de nid de boucles pour GPU illustré dans la figure 3.7 est composé de deux ensembles distincts de boucles imbriquées.

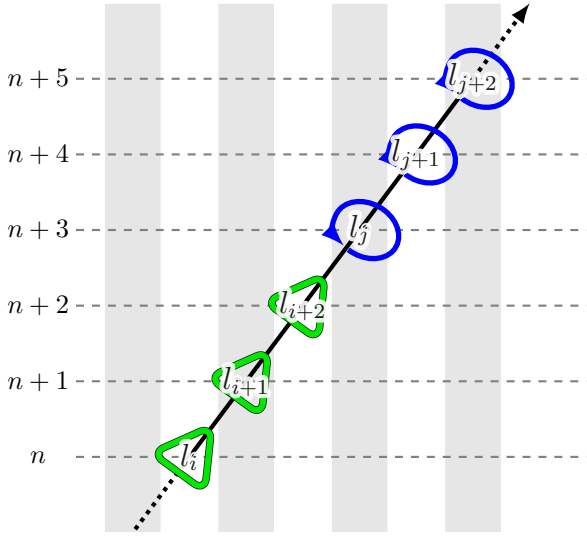


FIGURE 3.7 – Exemple de pattern de nid de boucles pour GPU

Le premier ensemble de boucles est composé d'une à trois boucles imbriquées parmi les plus externes du nid. Cet ensemble b de boucles correspond aux instances de blocs de

la grille de calcul globale qui sont exécutés par les *clusters GPC* du GPU. Ces derniers étant totalement indépendants, il n'existe pas d'instruction de synchronisation à ce niveau permettant de contraindre l'ordonnancement temporel de l'exécution des blocs. En conséquence, les boucles propres à cet ensemble doivent obligatoirement avoir été identifiées comme parallèles par l'analyse des dépendances.

Directement imbriqué dans le précédent ensemble, le second ensemble b de boucles est de même composé d'une à trois boucles imbriquées. Cet ensemble correspond aux différentes instances de *threads* inclus dans chaque bloc et sont exécutés par les *Cuda cores*. À la différence du premier, l'existence d'instructions de synchronisation telles que *___syncthreads* en Cuda permet non seulement de contraindre l'ordre d'exécution mais aussi d'échanger des données entre *threads*. Ainsi, cet ensemble n'a pas de contrainte quant-aux dépendances embarquées et permet de considérer tous types de boucle, séquentielles comme parallèles.

La figure 3.7, qui utilise uniquement les boucles de notre représentation de code, montre l'ensemble maximal de boucles portables sur GPU. Les boucles sont ici parfaitement imbriquées et correspondent à l'objectif. Cependant, à ce niveau de la méthodologie, il est concevable que les boucles imbriquées soient séparés par des blocs ou encore des branchements. La seule contrainte est alors d'étudier des boucles consécutives dont les niveaux de profondeurs se succèdent.

Nous introduisons à présent les ensembles permettant de définir les kernels qui répondent à ce premier critère de placement.

Nous notons $\mathcal{ND}(l_{i_0}, p)$ l'ensemble des nids de boucles de profondeur p, dont la première boucle est l_{i_0}.

Nous définissons, en (3.7), $K_0(l_{i_0}, B, T)$ l'ensemble des nids de boucles nd de profondeur $B + T$, dont la première boucle est l_{i_0}, ayant B boucles externes et T boucles internes et tels que les B boucles externes peuvent être exécutées en parallèle.

$$
\begin{aligned}
K_0(l_{i_0}, B, T) = \{ & nd \in \mathcal{ND}(l_{i_0}, B+T), B \in [1..3], T \in [1..3] \mid \\
& \exists b \in \mathbb{N}, t \in \mathbb{N} \;\; t.q. \\
& i_0 \leqslant b < i_0 + B, && \text{Boucles externes.} \\
& j_0 = i_0 + B, && \text{Boucles parfaitement imbriquées} \\
& j_0 \leqslant t < j_0 + T, && \text{Boucles internes.} \\
& \cup_{k=i_0}^{B-1} DL(l_b) = \varnothing && \text{Boucles externes parallèles.} \\
& \}
\end{aligned}
\tag{3.7}
$$

L'objectif est de maximiser le nombre de dimensions de l'espace d'itérations défini par B et T afin d'augmenter le taux d'occupation des grilles de calcul du GPU. Nous définissons, en (3.8), $K_{max}(l_{i_0}, B, T)$ l'ensemble des kernels de $K(l_{i_0}, B, T)$ qui maximise le nombre de boucles plaçables sur les composantes architecturales du GPU.

$$
\begin{aligned}
K_{max}(l_{i_0}, B, T) = \{ & nd_1 \in K_0(l_{i_0}, B, T), B \in [1..3], T \in [1..3] \mid \\
& \forall nd_2 \in K_0(l_{i_0}, B', T'), B' \in [1..3], T' \in [1..3] \\
& B + T \geq B' + T' \\
& \}
\end{aligned}
\tag{3.8}
$$

Les boucles externes des kernels $K_{max}(l_{i_0}, B, T)$ sont parallèles, tandis que celles qui sont internes peuvent supporter des dépendances. Nous cherchons désormais parmi l'ensemble des nids de boucles de $K_{max}(l_{i_0}, B, T)$ ceux qui ont un minimum de boucles internes

dépendantes. Cet ensemble correspond à $K_{min}(l_{i_0}, B, T)$, défini en (3.9). L'objectif est de réduire l'emploi d'instructions de synchronisation ou de barrières mémoire ayant un impact négatif sur les performances.

$$
\begin{aligned}
K_{min}(l_{i_0}, B, T) = \{\, & nd_1 \in K_{max}(l_{i_0}, B, T), B \in [1..3], T \in [1..3] \;\; | \\
& l_{t_1} \in nd_1, i_0 + B \leqslant t_1 \leqslant i_0 + B + T, && l_{t_1} \text{ boucles internes de } nd_1 \\
& DL(l_{t_1}) \neq \varnothing, && l_{t_1} \text{ non parallèles} \\
& \forall nd_2 \in K_{max}(l_{i_0}, B, T), \\
& l_{t_2} \in nd_2, i_0 + B \leqslant t_2 \leqslant i_0 + B + T, && l_{t_2} \text{ boucles internes de } nd_2 \\
& DL(l_{t_2}) \neq \varnothing, && l_{t_2} \text{ non parallèles} \\
& |t_1| \leqslant |t_2|. && \text{Min. \# boucles internes non parallèles} \\
& \}
\end{aligned}
\tag{3.9}
$$

Nous définissons en (3.10) l'ensemble $KC1(nd, l_\epsilon)$ des nids boucles dont les sous-nids débutant en l_{i_0} sont bien imbriquées et respectent les contraintes du critère 1. Les nids de boucles nd_0 peuvent résultés de transformations de boucles préalablement appliquées à nd afin d'augmenter le nombre de kernels vérifiant les critères (Section 3.4).

$$
\begin{aligned}
KC1(nd, l_\epsilon) = \{(nd_0, l_{i_0}), nd_0 \in T(nd, l_\epsilon), l_{i_0} \in nd_0 | \\
& \exists\, l_{i_0} \in nd_0, \\
& B \in [1..3], T \in [1..3], \\
& K_{min}(l_{i_0}, B, T) \neq \varnothing \\
& \}
\end{aligned}
\tag{3.10}
$$

Nous avons identifié six transformations de boucle, ayant un impact bénéfique possible sur ce premier critère. L'emploi de ces transformations dans le cadre du critère 1 est décrit en section 3.4 et plus spécifiquement dans les numéros de sections spécifiées entre parenthèses dans la liste suivante :

— la fission (3.4.2),
— la fusion (3.4.1),
— le *tiling* (3.4.6),
— l'échange de boucles (3.4.7),
— le *coalescing* (3.4.3) et
— les réductions parallèles (3.4.9).

L'ensemble des nids de boucles résultant de ces transformations, appliquées à un nid de boucles nd commençant par l_ϵ, est noté $T_{Pattern}(nd, l_\epsilon)$. C'est un sous ensemble de $T(nd, l_\epsilon)$ (Section 3.4, équation 3.15).

3.3.2 Critère 2 : Taille des domaines d'itération

Le second critère intitulé *loop size* dans la figure 3.6 porte sur les contraintes de taille liées aux domaines d'itérations des nids de boucles précédemment définis. L'existence de ces contraintes repose sur le fait que l'architecture des GPUs ne peut exécuter qu'un ensemble fini d'itérations pour chaque exécution de *kernel*. Ainsi, l'espace d'itérations engendré par l'ensemble des boucles de chaque *kernel* doit être borné par des paramètres dépendants de l'architecture cible.

En reprenant les ensembles définis, pour le critère 1, dans la section 3.3.1, nous définissons, en (3.11), l'ensemble $KC2(nd, l_\epsilon)$ des nids de boucles qui respectent le critère 2 et qui appartiennent au nid de boucles englobant nd, commençant en l_ϵ.

$$
\begin{aligned}
KC2(nd, l_\epsilon) = \{(nd_0, l_{i_0}), nd_0 \in T(nd, l_\epsilon), l_{i_0} \in nd_0 \mid \\
& (nd_0, l_{i_0}) \in KC1(nd, l_\epsilon), \quad nd_0 \text{ respecte critère 1} \\
& |I_{i_0}| \leqslant 2\ 147\ 483\ 647, \\
& |I_{i_0+1}| \leqslant 65\ 535, && \text{pour } B > 1 \\
& |I_{i_0+2}| \leqslant 65\ 535, && \text{pour } B > 2 \\
& j_0 = i_0 + B, \\
& |I_{j_0}| \leqslant 1024, \\
& |I_{j_0+1}| \leqslant 1024, && \text{pour } T > 1 \\
& |I_{j_0+2}| \leqslant 64, && \text{pour } T > 2 \\
& \prod_{p=0}^{T} |I_{j_0+p}| \leqslant 1024, \\
& \prod_{p=0}^{T} |I_{j_0+p}| \geqslant 4 \times 32, \\
& (\prod_{p=0}^{T} |I_{j_0+p}|) \% 32 = 0 \\
& \}
\end{aligned}
\tag{3.11}
$$

Les paramètres utilisés pour ce second critère proviennent du guide technique de Nvidia [123] et sont donc adaptés aux GPUs de ce constructeur. Les contraintes exercées sur le domaine d'itération de I_{i_0}, I_{i_0+1} et I_{i_0+2} portent sur l'ensemble b de boucles externes du critère 1. Les boucles I_{j_0}, I_{j_0+1} et I_{j_0+2} correspondent au contraire à l'ensemble t de boucles internes. Pour les deux ensembles, $|I_\epsilon|$ représente le cardinal de l'ensemble de départ I de la boucle d'indice ϵ tel que défini dans la section 3.1.7. Ces contraintes portent sur l'ensemble de départ I des boucles car l'application $\mathcal{I} : I \to L$ n'est pas obligatoirement bijective.

Les *clusters* des GPUs Nvidia ne peuvent exécuter plus de 1024 instances de *threads* par *bloc*. Ce critère implique la contrainte sur le produit des tailles des domaines d'itération du seconde ensemble de boucles tel que $\prod_{p=0}^{T} |I_{j_0+p}| \leqslant 1024$. Pour les GPUs d'AMD, l'architecture GCN porte ce paramètre à 2048.

Les deux dernières contraintes de couleur orange, ne sont pas strictes. La première porte sur la quantité minimale de *threads* par *bloc*. La seconde considère l'équilibrage du nombre de *threads* pour l'ensemble des *blocs* considérés. Leur violation ne remet pas en cause la légalité du placement. Cependant, dans un souci d'efficacité d'exécution de chaque *kernel*, il est conseillé de les prendre en considération.

Pour comprendre la raison d'être de ces contraintes, il est nécessaire de comprendre le fonctionnement des architectures GPU décrit dans la section 1.3.3. Chez Nvidia, lors de l'exécution des *kernels*, les *warp schedulers* sont chargés de découper les *blocs* de *threads* en *warps* de 32 *threads* chacun. Les *dispatch units*, au nombre de quatre par *cluster*, ont alors pour fonction de lancer chaque *warp* pour seize *cuda cores*. Ainsi, afin de tirer le maximum des capacités de chaque *cluster*, il est préférable de constituer un *bloc* avec un minimum de quatre *warps* soit 4×32 *threads* ou 128 itérations pour le second ensemble de boucles.

La pertinence de ces contraintes est spécifiquement abordée dans les expérimentations de la section 5.2.1.

La quantité globale d'itérations pour le second ensemble de boucles de *kernel* est donné par le cardinal du produit des domaines d'itérations soit, $\prod_{p=0}^{T}|I_{j_0+p}|$. En considérant le processus de découpage en *warps* décrit pour la précédente contrainte, nous cherchons à ce que chacun des *warps* générés soit entièrement chargé de *threads*. Chez Nvidia, chaque *warp* peut contenir jusqu'à 32 *threads*. Notre approche consiste ainsi à utiliser pour le second ensemble de boucles, un domaine d'itération qui soit un multiple de 32. Ce paramètre garantit alors que tout *warp* généré sera pleinement exploité. Chez AMD en revanche, les GPUs utilisent des *wavefronts*[5] constitués de 64 *work items*[6].

Pour adapter les boucles retenues à ce second critère, quatre transformations de boucle ont été retenues :

— le *tiling* (3.4.6),
— le *splitting* (3.4.4),
— le *strip mining* (3.4.5) et
— l'*unrolling* (3.4.8).

L'ensemble des nids de boucles résultant de ces transformations, appliquées à un nid de boucles nd commençant par l_ϵ, est noté $T_{Bounds}(nd, l_\epsilon)$. C'est un sous ensemble de $T(nd, l_\epsilon)$ (Section 3.4, équation 3.15).

3.3.3 Critère 3 : Empreinte mémoire

Les GPUs disposent d'unités mémoire pouvant être soit dédiées, soit partagées avec le CPU. Dans les deux cas de figure, la quantité finie de mémoire disponible justifie notre troisième critère de placement nommé *GPU memory size* dans la figure 3.6. Cette limite correspond alors, pour un contexte donné, à la quantité d'espace mémoire rendu exploitable par le driver du GPU. En pratique, cette quantité est inférieure à la quantité physiquement disponible car le GPU en exploite au moins une petite quantité [7] pour son fonctionnement propre, à laquelle s'ajoute la quantité de mémoire utilisée [8] pour l'exécution de la session graphique de l'OS. Les GPUs en 2018 disposent d'espaces mémoire pouvant atteindre 24Go dans le cas de la *Tesla P40* chez Nvidia.

Nous définissons, en (3.12), l'ensemble $KC3(nd, l_\epsilon)$ des nids de boucles qui respectent le critère 3 et qui appartiennent au nid de boucles englobant nd, commençant en l_ϵ. Ces nids de boucles doivent respectés avant tout le critère 2. Soit $M_{k_{i_0}}$ l'empreinte mémoire du kernel k_{i_0} composé du nid de boucles ayant l_{i_0} en première boucle externe.

$$\begin{aligned} KC3(nd, l_\epsilon) = \{(nd_0, l_{i_0}), nd_0 \in T(nd, l_\epsilon), l_{i_0} \in nd_0 \mid \\ (nd_0, l_{i_0}) \in KC2(nd, l_\epsilon), \\ |(M_{k_{i_0}}| < |M^{acc/global}| \\ \} \end{aligned} \tag{3.12}$$

L'empreinte mémoire $M_{k_{i_0}}$ de ce *kernel* est alors donné par la relation (3.13) où $A_{k_i i_0}$ correspond à l'ensemble des accès mémoire effectués au sein du *kernel*. Elle dépend de L_i, quelque soit la taille $B + T$ des dimensions placées sur les composantes du GPU.

5. Équivalent des *warps* de Nvidia
6. Équivalent des *threads* de Nvidia
7. La mise en œuvre des GPUs Nvidia requiert environ 50Mo dans la mémoire globale
8. La session graphique prend environ 500Mo dans la mémoire globale des GPUs Nvidia

$$L_i \xrightarrow{\mathcal{A}} A_{k_{i_0}} \xrightarrow{\mathcal{L}} M_{k_{i_0}} \tag{3.13}$$

Le critère 3 permet ainsi de vérifier que la taille de l'empreinte mémoire des *kernels* étudiés reste inférieure à l'espace mémoire principal du GPU $M^{acc/global}$.

Ce troisième critère peut être étendu en considérant les autres espaces mémoire des GPUs. Nous avons réalisé, dans ce but, une étude de ces différents espaces mémoire dans la section 5.1.

Enfin, dans le cas où ce critère ne serait pas rempli, trois transformations de boucles décrites dans la section 3.4 ont été identifiées pour améliorer cette situation :

— la fission (3.4.2),
— le *tiling* (3.4.6) et
— le *strip mining* (3.4.5).

Ces transformations ont pour caractéristique intéressante de réduire l'empreinte mémoire des boucles concernées.

L'ensemble des nids de boucles résultant de ces transformations, appliquées à un nid de boucles nd commençant par l_ϵ, est noté $T_{Memory}(nd, l_\epsilon)$. C'est également un sous ensemble de $T(nd, l_\epsilon)$ (Section 3.4, équation 3.15).

3.3.4 Sélection d'un *kernel*

Les trois critères de placement, abordés dans les sections 3.3.1, 3.3.2 et 3.3.3, retournent un ensemble de solutions valides de placement sur GPU.

Parmi toutes les solutions possibles, nous choisissons, en (3.14), les nids de boucles ayant pour première boucle externe celle de plus haut niveau.

$$\begin{aligned} KC123_{opt}(nd, l_\epsilon) = \{(nd_0, l_{i_0}) \in KC3(nd, l_\epsilon) \mid \\ \forall (nd_1, l_{i_1}) \in KC3(nd, l_\epsilon) \\ i_0 \leqslant i_1 \\ \} \end{aligned} \tag{3.14}$$

Nous introduisons, et utiliserons dans la suite de ce document, $k_i^{B,T}$, un *kernel* solution de $KC123_{opt}(nd, l_\epsilon)$. Il s'agit du sous-nid de boucles (n_i, l_i), composé des $B + T$ boucles et dont l'identifiant unique i correspond à l'indice de la boucle la plus externe l_i.

3.4 Amélioration de la quantité de code placé sur GPU par transformations de boucles

Nous venons de définir au cours de la section 3.3, trois critères permettant de déterminer l'adéquation d'un ensemble de boucles aux contraintes architecturales d'un GPU. Cependant, ces critères ont un fonctionnement binaire permettant uniquement de discriminer les ensembles de boucles compatibles et incompatibles. Ainsi, nous avons identifié un ensemble de transformations de code pouvant être exécutées de manière légale, car préservant l'aspect fonctionnel de l'algorithme étudié. Nous considérons dans cette section les ensembles de boucles incompatibles avec l'architecture des GPUs et tentons de les rendre "compatibles".

Notre méthodologie permet ainsi d'appliquer, sur ces ensembles de boucles, des transformations de code afin d'augmenter le nombre de kernels potentiellement exécutables sur GPU. Nous notons $T(nd, l_\epsilon)$, défini en (3.15), l'ensemble des versions d'un nid de boucles nd commençant par l_ϵ après application de transformations de programme légales de $\mathcal{T}_{legal}$.

$$T(nd, l_\epsilon) = \{nd' \in \mathcal{ND}(l_\epsilon, .)|\ \exists \texttt{Transformation} \in \mathcal{T}_{\texttt{legal}}, \texttt{Transformation(nd)} = \texttt{nd}' \quad \} \tag{3.15}$$

Chaque application d'une de ces transformations engendre un cycle dans la méthodologie. Il faut ainsi effectuer un retour vers le processus de vérification des critères de placement. Si malgré les cycles de transformations, un ensemble de boucles ne parvient pas à répondre aux trois critères précédemment définis, alors celui-ci sera définitivement rejeté et demeurera sur CPU. Nous abordons dans cette section l'apport, vis-à-vis du placement de code sur GPU, des *parallel reductions* ainsi que des transformations de boucles suivantes : la *fusion*, la *fission*, le *tiling*, le *strip mining*, l'*interchange*, le *splitting*, le *coalescing* et l'*unrolling*. Chacune de ces transformations est introduite par une description de son usage dans notre méthodologie. L'aspect légal de la transformation est ensuite défini. Un exemple d'application illustre cette description. Enfin un bref récapitulatif argumentant l'impact vis-à vis des autres transformations vient clore chaque transformation afin de favoriser la convergence vers une solution de placement.

Nous utiliserons un extrait de la fonction *crossBilateralFilter* pour illustrer ces transformations de code. Cette fonction est issue de l'algorithme *simpleFlow* [150] dont l'intégralité figure en annexe A. Nous nous référons pour la suite de ce chapitre à la représentation spinale de cet extrait de code, représenté dans la figure 3.7.

Afin d'être utilisable pour l'ensemble des transformations, nous avons préalablement privatisé le tableau *weights* dans le corps de la boucle l_7, afin de rendre les boucles $l_{6,7}$ parallèles. Nous considérons son allocation possible dans l'espace mémoire du GPU.

3.4.1 Fusion de boucles

La fusion est utilisée pour valider le critère 1.

Description de la transformation

La fusion est une transformation de boucles permettant de "fusionner" le contenu de deux boucles adjacentes et de caractéristiques similaires, en une unique boucle. Pour deux boucles l_ϵ et $l_{\epsilon'}$ de même niveau de profondeur, l'application de la fusion permet de fusionner le corps de ϵ et ϵ' dans ϵ''. Nous avons la relation :

$$fusion(I_\epsilon, I_{\epsilon'}) = I'_{\epsilon''} \text{ tel que } I_\epsilon = I_{\epsilon'} = I'_{\epsilon''}$$

La fusion de boucles est couramment employée pour :
— augmenter la granularité des boucles concernées,
— améliorer la localité temporelle des données et ainsi améliorer l'impact des mémoires caches,
— réduire les synchronisations entre boucles,
— réduire la surcharge des entêtes de boucles,
— favoriser l'utilisation des registres.

Kennedy et McKinley[81] ont notamment étudié son utilisation dans le but commun de :
— maximiser le parallélisme des boucles,
— minimiser les dépendances de données,
— maximiser la localité des données.

Cette transformation joue un rôle majeur dans de nombreux cas d'optimisation de code, comme le prouve les nombreuses publications à son sujet [101, 102, 144, 42, 61, 157, 139, 16].

Critère d'applicabilité de la fusion

La fusion et la fission (3.4.2), utilisées conjointement, permettent de redistribuer les boucles entre nids et de converger vers un ensemble de nids de boucles parfaitement imbriquées [42]. Nous considérons ainsi la fusion de boucles, afin de transformer un nid de boucles irrégulier en une imbrication de boucles régulières répondant au critère 1 de la section 3.3.1. À l'opposé de l'imbrication parfaite de boucles de la figure 3.7, il arrive couramment que plusieurs sous-ensembles de boucles distincts soient intégrés dans un corps de boucle. La figure 3.7 présente un cas concret provenant d'un extrait de la fonction $crossBilateralFilter$. Dans cet exemple, l'ensemble des boucles $l_{6,7}$ est un bon candidat pour un placement sur les *blocs* d'un GPU. Cependant, dans le but de maximiser le nombre d'instances de *kernel* générés, il est possible d'exploiter les trois dimensions d'instances de *threads* du GPU qui restent inexploitées. Ceci nécessite l'intégration de boucles plus profondes au sein du nid. À ce titre, nous trouvons justement dans notre exemple un ensemble de boucles candidates, $l_{[\![8;24]\!]}$. Cependant, cet ensemble n'est pas intégralement imbriqué, la profondeur du nid étant de 5 niveaux pour un total de 19 boucles. Plusieurs boucles se retrouvent ainsi à des niveaux de profondeur identiques au sein du nid. La boucle l_7, notamment, contient plusieurs sous-ensembles de boucles portés par les boucles $l_{8,10,12,14,16,18,20}$. C'est la raison pour laquelle l'ensemble de ces boucles ne peut être intégré dans l'espace des instances de *kernel*. Pour le moment, le *kernel* est de type $k_6^{2,0}$ selon la nomenclature $k_i^{B,T}$. Cependant, la condition portant sur le domaine d'instance des *threads* du critère 1 n'est pas remplie du fait que $T = 0$.

Légalité de la transformation

La fusion reste légale tant que le sens des dépendances embarquées n'est pas modifié par la transformation et que le nombre d'itérations des boucles concernées est identique [9]. Nous nous référerons à ce sujet aux publications citées ci-dessus.

Cependant l'architecture des GPUs fonctionne à partir d'instances de *threads* réparties en *blocs* dont l'ordre d'exécution est considéré comme aléatoire. La fusion est donc applicable dans notre contexte pour l'ensemble des *blocs* lorsque les boucles concernées ne présentent pas de dépendance, ce qui est en cohérence avec le critère 1. Pour chaque *bloc*, l'ordre d'exécution des *threads* sur les SM sera contraint au moyen d'instructions de synchronisation afin de préserver lorsqu'elles existent, les dépendances embarquées.

Dans les deux cas, les boucles concernées devront avoir le même niveau de profondeur ainsi qu'un nombre d'itérations identique, tel que :

$$I_{l_\epsilon} = I_{l_{\epsilon'}}$$

Dans le cadre des GPUs, l'application de la fusion pour des domaines d'itérations de tailles différentes est cependant décrite en section 3.5.2.

Exemple de fusion de boucle

Les domaines d'itération des huit couples de boucles imbriquées $l_{8,9}$, $l_{10,11}$, $l_{12,13}$, $l_{14,15}$, $l_{16,17}$, $l_{18,19}$, $l_{21,22}$ et $l_{23,24}$ sont identiques. Il est donc envisageable d'effectuer 8 fusions de boucles réduisant ainsi le nid à 5 boucles imbriquées pour 5 niveaux de profondeurs distincts. Le *kernel* $k_6^{2,0}$ dans son état initial est composé des boucles $l_{8,10,12,14,16,18,20}$. Seuls les six premiers sous-ensembles de boucles sont fusionables, car les deux derniers

9. Sous condition que les domaines d'itérations de boucles soient compatibles

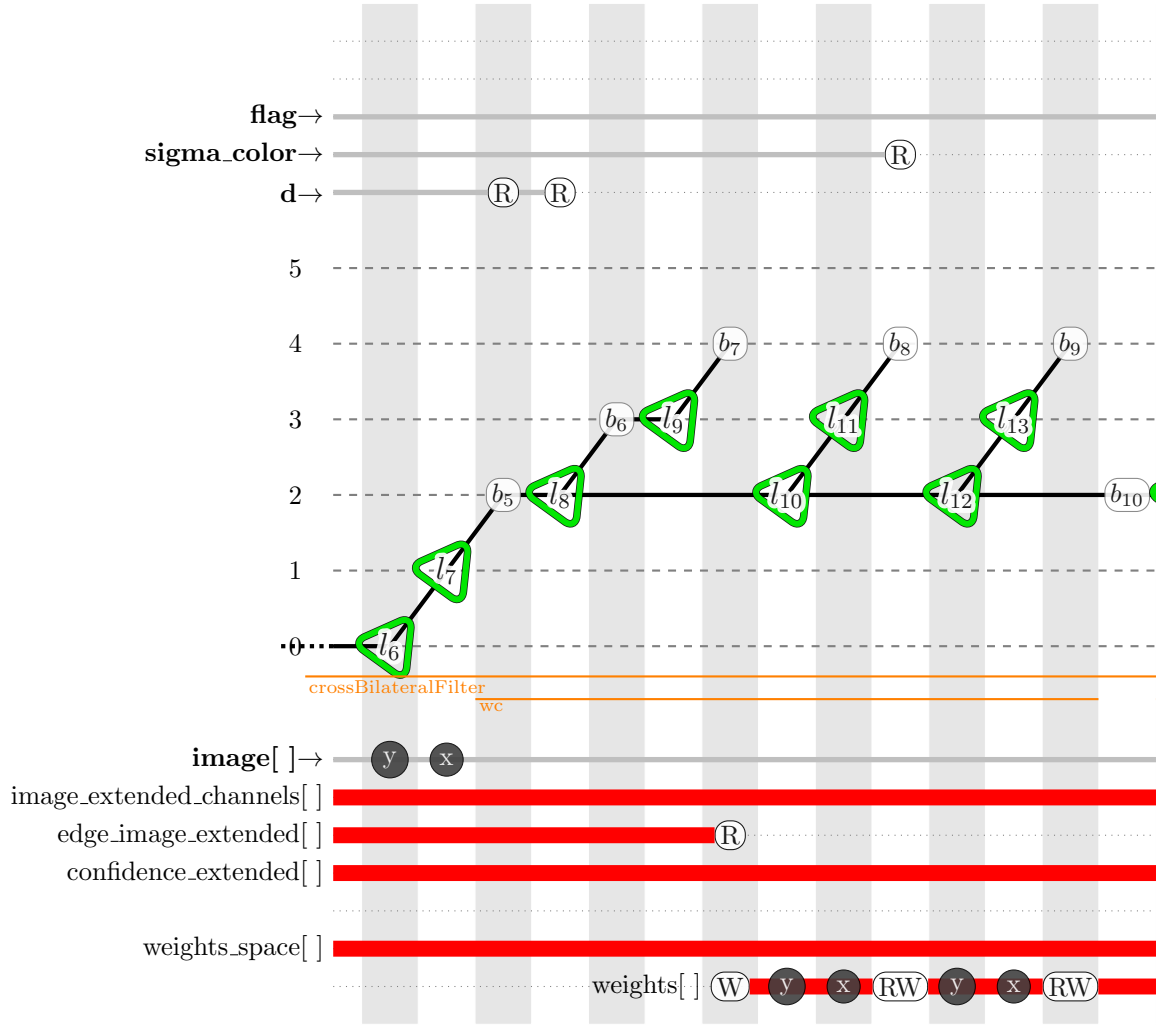


FIGURE 3.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (1/2)

sous-ensembles de boucles $l_{21,22}$ et $l_{23,24}$ sont intégrés dans la boucle l_{20} dont le nombre d'itérations diffère de celui des précédentes.

Critère de convergence

En appliquant récursivement la fusion, la boucle résultante intègre les corps des boucles présentant un même niveau de profondeur. Dans le cadre de notre méthodologie, il s'agit alors d'une propagation horizontale par rapport à la représentation spinale. Cette application récursive est utilisable pour chaque niveaux de profondeur des boucles de *kernel*, en allant de la boucle la plus externe vers la plus interne. La propagation est alors verticale.

La fusion n'est cependant plus utile :

1. lorsqu'il n'existe plus de sous-ensemble de boucles de même niveau de profondeur que les boucles constitutives du *kernel* (pour la propagation horizontale),
2. lorsque la propagation horizontale de la fusion a été appliquée pour l'ensemble des boucles du *kernel* (pour la propagation verticale).

Pour le premier point, prolonger la propagation verticale au-delà des boucles du *kernel* permet d'améliorer les performances d'exécution de ce dernier. Il s'agit cependant d'un cas d'optimisation différent de notre cas d'application propre au placement sur GPU.

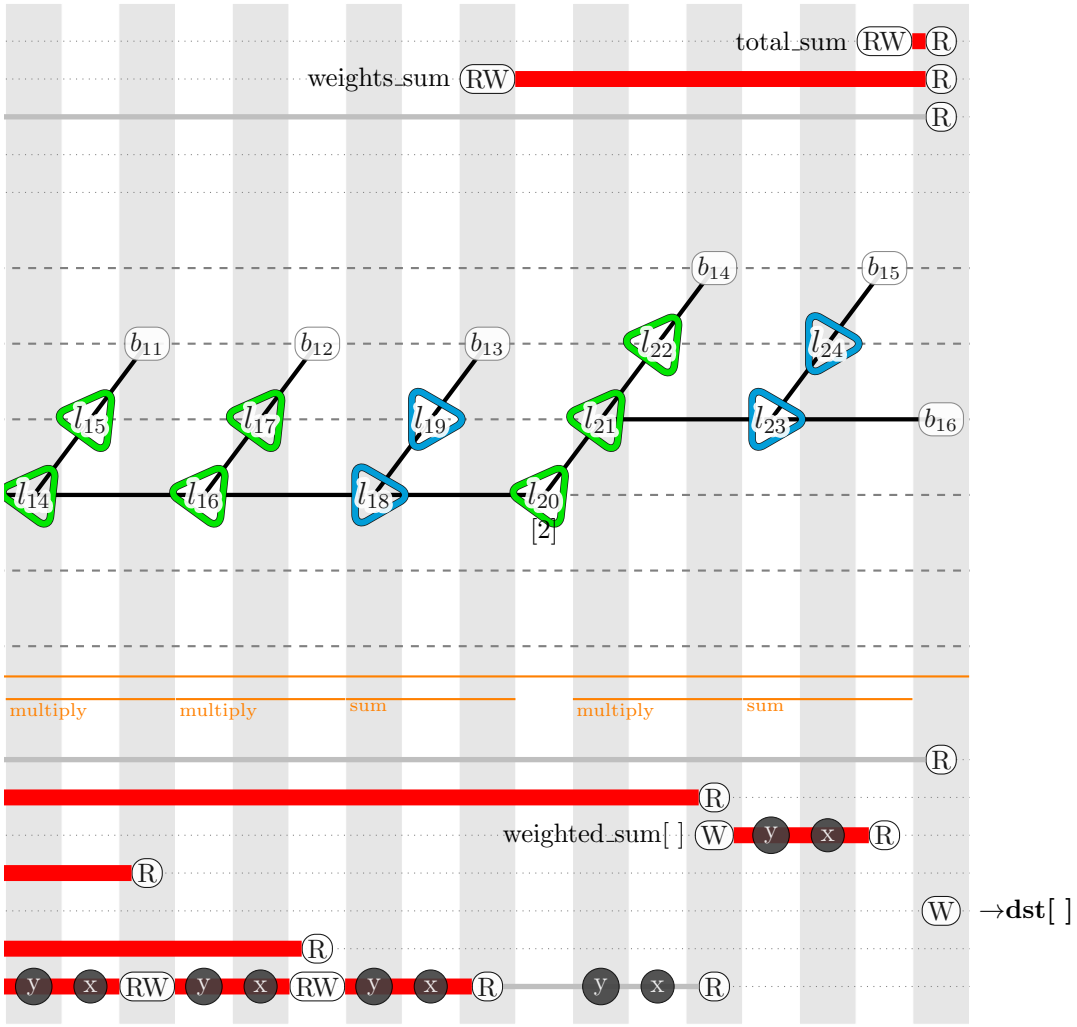


FIGURE 3.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (2/2)

La fusion n'est pas envisagée lorsque :

— son application avec une boucle séquentielle engendre une dépendance sur les *blocs* du GPU, le critère 1 n'est alors plus rempli,

— l'empreinte mémoire à l'issue de la transformation devient trop importante pour les espaces mémoire du GPU, le critère 3 n'est alors plus vérifié.

Pour ces deux cas, la fission détaillée dans la section 3.4.2 pourra être envisagée en remplacement.

3.4.2 Fission ou distribution de boucles

La fission, aussi appelée distribution de boucles, est utilisée pour valider les critères 1 et 3.

Description de la transformation

La fission fractionne le contenu d'un corps de boucle afin de répartir chacun des fragments résultants, dans une boucle indépendante et de caractéristiques identiques à la boucle initiale. Cette transformation est illustrée en 3.16. Son application sur une boucle

l_ϵ engendre deux boucles l_ϵ et $l_{\epsilon'}$. Les domaines d'itérations I_ϵ, I'_ϵ et $I'_{\epsilon'}$ de ces trois boucles sont identiques. La profondeur du nid concerné est également inchangée.

Nous avons la relation :

$$I_\epsilon \rightarrow I'_\epsilon, I'_{\epsilon'} \text{ tel que } I_\epsilon = I'_\epsilon = I'_{\epsilon'} \tag{3.16}$$

Cette transformation a pour effet de réduire la granularité des boucles et de réduire la localité temporelle des données [10].

Cette transformation a un effet inverse de celui de la fusion (3.4.1). Elle permet ainsi d'annuler l'effet d'une fusion, afin de retrouver l'espace initial.

La fission est couramment utilisée pour isoler les portions de code à l'origine d'une dépendance embarquée, dans le corps d'une boucle. Si des portions de codes parallèles existent, son application permet alors d'extraire de la boucle initialement séquentielle, une ou plusieurs boucles parallèles.

Critère d'applicabilité

La fission est employée pour valider les critères 1 et 3.

Dans le cadre du critère 1, elle répond à la problématique portant sur la parfaite imbrication des boucles constituant un *kernel*. Ce sujet a déjà été abordé précédemment pour la fusion. À l'inverse de cette dernière, la fission fragmente chaque sous-ensemble de boucles en un *kernel* distinct. Pour les données communes aux différents *kernels*, cela se traduit par un transfert des données utilisées en registres, en communications avec la mémoire globale. Cette dernière présentant un temps de latence beaucoup plus élevé, la fission sera utilisée lorsque les critères d'application de la fusion ne peuvent pas être satisfaits.

Appliquée récursivement jusqu'à la boucle externe du *kernel*, la fission permet de multiplier les *kernels*. Ainsi dans le cadre du critère 3, celle-ci sera employée lorsque la contrainte portant sur la mémoire GPU est dépassée. La fission permet ainsi de répartir les accès mémoire entre les différents *kernels* générés, réduisant alors l'empreinte mémoire du *kernel* initial. Par extension, nous employons de la même façon cette transformation pour réduire la pression sur les registres.

Légalité de la transformation

Kennedy et McKinley [81] définissent la fission de boucle comme légale, si toutes les instructions impliquées dans un cycle de dépendance sont conservées dans une unique boucle.

Dans le cadre de son application pour les GPUs, cette transformation reste légale tant qu'elle n'engendre pas de cycle de dépendances entre les *kernels* générés.

Exemple de fission de boucle

Dans l'exemple de la figure 3.7, la boucle l_{20} ne peut être fusionnée avec les boucles $l_{8,10,12,14,16,18}$ car son domaine d'itérations est incompatible. Une solution possible consiste à appliquer une fission de boucle au niveau de la boucle l_{20}. La transformation génère ainsi une boucle $l_{7'}$ de caractéristiques identiques à l_7. Cette transformation est appliquée récursivement jusqu'à la boucle externe du *kernel*, soit la boucle l_6. Cette application récursive génère au final un second *kernel* incluant les boucles $l_{6',7',20}$.

10. Ce qui a un effet positif sur l'utilisation du cache

La fission est couramment utilisée comme une étape initiale aux transformations et optimisations automatisées de boucles. Son application génère plusieurs nids parfaitement imbriqués. L'application des transformations de boucles suivantes s'en trouve simplifiée. Ce cas est abordé notamment par Darte [42]. Les combinaisons de boucles sont ensuite recomposées par fusion selon le modèle d'optimisation désiré.

Critère de convergence

Dans le cadre de notre méthodologie, l'objectif premier est d'obtenir un ensemble de boucles parfaitement imbriquées afin de les placer sur GPU. La fission sera employée si la fusion n'est pas applicable et que l'empreinte mémoire du *kernel* reste supérieure au critère 3 ou que la pression des registres reste trop forte. Cependant, ce dernier point ne remet pas en cause la légalité du placement. Il est considéré en tant qu'objectif secondaire pour optimiser le temps d'exécution du *kernel*. Un équilibre doit être trouvé entre la pression des registres et la pénalité des temps d'accès mémoire induits par la fission.

3.4.3 Coalescing

Le *coalescing* est utilisé pour valider le critère 1.

Description de la transformation

L'objectif du *coalescing* est, comme pour la fusion, de combiner deux boucles en une seule et unique. Cependant à la différence de cette dernière, le *coalescing* exploite des boucles imbriquées et regroupe des itérations et non pas des corps de boucles. Dans la représentation spinale, l'effet est une translation verticale car elle porte sur la profondeur du nid de boucles, comme décrit en 3.17. Elle réduit un domaine d'itérations I à p dimensions en un nouveau domaine I' composé de p' dimensions, tel que $p > p'$. La taille du domaine d'itérations global reste inchangée du fait que $\prod_i^p |I_i| = \prod_i^{p'} |I'_i|$.

$$I^p \rightarrow I'^{p'} \text{ tel que } \begin{cases} p > p' \\ \prod_i^p |I_i| = \prod_i^{p'} |I'_i| \end{cases} \tag{3.17}$$

Les fonctions d'accès mémoire qui font référence aux domaines d'itérations L^p doivent être modifiées pour prendre en compte le réduction du nombre de dimensions d'itérations.

Cette transformation est bénéfique pour lutter contre la surcharge liée aux entêtes de boucles multiples. Cependant, le GPU employant des instances de *threads* à la place des boucles, cet apport reste limité.

Critère d'applicabilité

Le *coalescing* est employé dans notre méthodologie afin de maximiser l'emploi des instances de *threads* et de *blocs* disponibles sur le GPU. Elle permet d'augmenter le nombre de boucles imbriquées placées sur les six dimensions du domaine d'instances du *kernel*.

L'emploi du *coalescing* est déclenché par l'existence de boucles dans le corps du *kernel* alors que les six dimensions d'instances de *kernel* sont déjà exploitées. Ces six dimensions ne doivent cependant pas être saturées, conformément aux contraintes du critère 2 portant sur les domaines d'itérations. Cette transformation a ainsi un impact sur la structure de boucles validé par le critère 1.

Légalité de la transformation

Par définition, le *coalescing* ne modifie pas l'ordre des itérations du nid de boucles concerné et par conséquence l'ordre des accès mémoire. La seule restriction porte sur l'application à un ensemble de boucles parfaitement imbriquées.

Exemple de *coalescing*

Nous considérons dans le strict cadre de cet exemple que l'architecture cible est restreinte à 2 dimensions d'instances pour les *threads* ainsi que pour les *blocs*. Dans l'exemple de la fonction $crossBilateralFilter$ (figure 3.7), nous considérons après *fission* le nid de boucles imbriquées $l_{6,7,20,21,22}$. Les boucles imbriquées $l_{6,7}$ sont placées sur les *blocs* et viennent ainsi occuper les deux dimensions d'instances de blocs disponibles. Les boucles imbriquées $l_{20,21,22}$ constituant trois dimensions d'itérations ne peuvent tenir sur les deux dimensions du domaine d'instances de *threads* spécifié dans cet exemple. Deux stratégies sont alors possibles. Le *coalescing* peut être appliqué sur :

— l_7 et l_{20},
— l_{20} et l_{21}.

Dans le premier cas, le domaine d'instances des *blocs* est renforcé. Si l_{20} était une boucle séquentielle, cette solution ne serait pas possible car les *blocs* doivent être indépendants (critère 1). Cependant l_{20} étant une boucle parallèle, cette option reste envisageable.

Dans le second cas, c'est au contraire le domaine d'instances des *threads* qui est renforcé. Le critère 2 montre des contraintes plus strictes sur le nombre d'instances des *threads* que pour l'ensemble des *blocs*. La saturation des *threads* est donc d'autant plus rapide. Cette limitation pourra être contournée par l'utilisation d'autres transformations réduisant les domaines d'itérations.

Au final, la boucle l_{20} étant parallèle, le choix dépend de la taille des domaines d'itérations I_{21} et I_{22}, et leur impact sur le critère 2.

Critère de convergence

Dans le cadre de notre méthodologie, l'utilisation du *coalescing* dépend de :

— la saturation de l'un des domaines d'instances définis par le critère 2,
— la profondeur du nid de boucles imbriquées,
— l'existence de boucles séquentielles entourant l'ensemble des *blocs* du *kernel*,
— l'existence de sous-ensembles de boucles compatibles avec l'imbrication du nid de boucles.

Enfin, nous notons que cette transformation n'a pas d'impact sur l'empreinte mémoire validée par le critère 3, car l'ensemble des accès mémoire est maintenu dans le corps du *kernel*.

3.4.4 Index set splitting

L'*index set splitting*, que nous appellerons *splitting*, est essentiellement utilisé pour valider le critère 2.

Description de la transformation

Le *splitting* est une transformation qui, à partir d'une boucle, génère une partition de son espace d'itérations en plusieurs sous-ensembles de boucles, de même niveau de

profondeur dans le nid. Elle diffère de la fission (3.4.2), en modifiant le domaine d'itérations et non pas les instructions contenues dans le corps de la boucle originale.

Les caractéristiques du *splitting* sur le domaine d'itérations sont synthétisées dans la formule 3.18. Son application $\mathcal{S}$ induit ainsi une réduction du domaine d'itération original I en fragmentant ce dernier en n domaines d'itérations contigus. Le nombre global d'itérations reste inchangé.

$$I = \bigcup_i I'_i,\ \forall i \forall j\ I'_i \cap I'_j = \varnothing \tag{3.18}$$

Les fonctions d'accès mémoire qui font références à ce domaine d'itérations doivent être modifiées en conséquence.

Critère d'applicabilité

Cette transformation est utilisée afin de réduire un domaine d'itérations trop conséquent par rapport aux contraintes du critère 2. Plusieurs sous-domaines d'itérations sont alors générés conformément aux contraintes sur le nombre d'instances de *blocs* et de *threads* du GPU. La difficulté repose alors sur la recherche du partitionnement maximisant les performances de l'architecture.

Appliqué sur la boucle de *kernel* la plus externe, le *splitting* permet de fragmenter l'appel à ce *kernel* en plusieurs appels contenant moins d'instances. L'empreinte mémoire pour chaque appel est alors plus faible qu'initialement. En conséquence, le *splitting*, dans ce cas précis, a un impact sur le critère 3.

Légalité de la transformation

Le *splitting* ne modifie pas l'ordre d'itérations des boucles et préserve ainsi les dépendances existantes. De ce fait, il peut être légalement appliqué aussi bien sur les boucles parallèles que séquentielles. Cependant dans le cadre du placement sur GPU, l'application du *splitting* sur les boucles du *kernel*[11] engendre des sous-ensembles de boucles non parfaitement imbriquées, incompatibles avec les contraintes d'imbrication du critère 1. Il convient dans ce cas d'appliquer successivement au *splitting*, une autre transformation agissant sur l'imbrication des ensembles de boucles telle que la fusion ou la fission.

Exemple d'*index set splitting*

Dans l'exemple de la fonction $crossBilateralFilter$ (figure 3.7), nous considérons le cas où l'espace mémoire *weights* défini dans A^2 est tel que $|A_1| = |A_2| = 65$. En se focalisant sur l'ensemble de boucles $l_{6,7,18,19}$, le domaine d'itérations des boucles $l_{18,19}$ permet le parcours de A_1 et A_2. L'ensemble $l_{6,7}$ étant placé sur les *blocs* et $l_{18,19}$ sur les *threads*, le critère 2 n'est alors pas respecté du fait que $|I_{18}| \times |I_{19}| > 1024$. L'application du *splitting* permet alors de fragmenter le domaine d'itérations initial et de générer des boucles compatibles avec les contraintes du critère 2.

Pour respecter nos critères, l'utilisation des instances de *threads* est maximisée en segmentant l_{18} en boucles de $\lfloor \frac{1024}{|I_{19}|} \rfloor = 15$ itérations. Le *splitting* de la boucle l_{18} génère $\lceil \frac{|I_{18}|}{15} \rceil = 5$ boucles. La dernière de ces boucles est composée de $|I_{18}|\%15 = 5$ itérations seulement.

Cette solution permet de remplir l'ensemble des contraintes du critère 2 tout en minimisant le nombre de boucles générées.

11. À l'exception de la boucle la plus externe.

Une solution plus optimale encore, serait d'appliquer un *coalescing* sur $l_{18,19}$ générant ainsi un domaine d'itération global mono-dimensionnel. Un *splitting* est alors appliqué, surchargeant ainsi les *threads* avec 1024 instances d'une part et les *warps* avec 32 *threads* d'autre part. Seule la dernière boucle ne serait pas maximisée avec 129 *threads*.

Si les dépendances le permettent, il est possible ensuite d'appliquer une fusion pour l'ensemble des boucles générées afin de répondre aux contraintes du critère 1. Dans le cas où les registres viendraient à être saturés, nous appliquerions à la place une fission sur l'ensemble $l_{6,7}$. Cette solution, du fait de la dépendance sur $l_{18,19}$, engendrerait en revanche une hausse des communications mémoire.

Critère de convergence

Dans le cadre de notre méthodologie, le *splitting* est appliqué afin de converger vers une solution conforme au critère 2. Une fois cet objectif atteint, son application n'est plus nécessaire. Cependant, cette transformation n'est pas suffisante pour converger vers une solution de placement. Son application a tendance à générer des sous-ensembles de boucles allant à l'encontre du critère 1. L'utilisation de la fusion ou de la fission permet de répondre à cette problématique.

3.4.5 Strip mining

Le *strip mining* de boucles est utilisé pour valider les critères 2 et 3.

Description de la transformation

Le *strip mining* permet de segmenter le domaine d'itérations d'une boucle en blocs [12] **égaux** composés d'itérations contiguës. Cette transformation implique l'utilisation de deux boucles imbriquées afin de couvrir le domaine d'itérations initial. La *boucle externe* de cette transformation parcourt l'ensemble des blocs d'itérations tandis que la *boucle interne* parcourt les itérations pour chaque bloc. De ce fait, le *strip mining* a un impact sur la profondeur du nid de boucles généré.

Cette transformation est communément utilisée pour la vectorisation. Elle permet alors de générer à partir du domaine d'itération initial, un sous-espace dont la taille est compatible avec la famille d'instructions vectorielles utilisée. En fonction du contenu de la boucle interne, celle-ci peut être remplacée ultérieurement par une instruction vectorielle dédiée.

Concernant les niveaux de boucles, le *strip mining* a un effet opposé à celui du *coalescing* (3.4.3). Pour un ensemble de boucles imbriquées, le premier consomme des itérations afin de constituer de nouvelles boucles. Celles-ci viennent augmenter la profondeur du nid. À l'inverse, le second consomme des boucles imbriquées afin d'expanser le domaine d'itérations des autres boucles. La profondeur du nid est alors compressée.

Les caractéristiques nous intéressant, pour cette transformation au niveau du nid de boucles, sont illustrées dans la formule 3.19. L'application du *strip mining* permet d'augmenter la profondeur p d'un ensemble de boucles tel que $p' = p + 1$.

12. Nous noterons la différence entre les blocs d'itérations pour un ensemble de boucles et les *blocs* d'instances spécifiques aux GPUs. Le sujet porte sur les blocs d'itérations ici.

$$strip\_mining(I^p, k) = I'^{p'} \text{ tel que } \begin{cases} p' = p + 1 \\ \prod_i^p |I_i| = \prod_i^{p'} |I'_i| \\ \exists \epsilon, |I_\epsilon| = |I'_{\epsilon_1}| \times |I'_{\epsilon_2}| \\ |I'_{\epsilon_1}| = \frac{|I_\epsilon|}{k} \\ |I'_{\epsilon_2}| = k \end{cases} \tag{3.19}$$

Critère d'applicabilité

Le *strip mining* est employé lorsqu'au moins un des domaines d'itérations des boucles du *kernel* a une taille trop élevée pour le critère 2. La transformation est alors appliquée pour générer une boucle interne dont le domaine d'itérations est compatible avec le domaine d'instances du GPU. L'utilisation conjointe de l'*interchange* (section 3.4.7) permet de positionner la boucle externe du *strip mining* parmi l'ensemble des boucles du *kernel*[13], si celle-ci est compatible. Dans le cas contraire, on essaie de la placer à l'extérieur de ce dernier.

Le *strip mining* est aussi une solution, lorsque l'empreinte mémoire d'un *kernel* est trop forte et ne respect par le critère 3. Celui-ci est alors employé pour générer une boucle interne compatible avec la disponibilité mémoire du GPU. La boucle externe de cette transformation sera obligatoirement déplacée vers l'extérieur du *kernel* grâce à l'*interchange*. Cette transformation génère ainsi une boucle externe effectuant plusieurs appels d'un même *kernel* dont l'empreinte mémoire a été rendue compatible avec les caractéristiques du GPU.

Légalité de la transformation

Le *strip mining* augmente le nombre de dimensions du domaine d'itérations utilisé sans avoir de conséquence sur l'ordre d'accès aux données. Les dépendances sont ainsi préservées. L'emploi de cette transformation est possible pour toute boucle, séquentielle comme parallèle. Son application est donc admise pour les instances de *blocs* ou de *threads* du GPU.

Un code de contrôle sur les instances du GPU est ajouté au début du corps du *kernel*, lorsque :

— la taille du domaine d'itérations initial est un nombre premier ou que
— le nombre d'itérations k de la boucle interne n'est pas un diviseur entier du domaine d'itérations de la boucle initiale.

Ce code de contrôle préserve ainsi un code correct et évite tout débordement mémoire.

Exemple de transformation

Nous utilisons l'exemple de l'*index set splitting* (3.4.4), mais appliqué à l'ensemble de boucles parallèles $l_{6,7,16,17}$. Le raisonnement est alors similaire, cependant nous appliquons le *strip mining* sur la boucle l_{16} générant ainsi deux boucles imbriquées. La boucle interne $l_{16'}$ correspond alors aux 15 instances de *threads*. La boucle externe l_{16} étant parallèle, celle-ci est modifiée pour correspondre à 5 instances de *blocs* sur le GPU. Enfin, un code de contrôle est ajouté dans le *kernel* afin de vérifier que $|l_{16} \times l_{16'}| = 65$. Comparé au *splitting*, cette transformation a pour avantage d'augmenter l'utilisation des *blocs* du GPU.

13. le plus souvent parmi les instances de *blocs* du GPU.

Critère de convergence

Dans le cadre de notre méthodologie, le *strip mining* peut être appliqué à chacune des boucles du *kernel* ne répondant pas aux critères 2 et 3. En paramétrant correctement la taille des blocs d'itérations et tenant compte des contraintes des critères 2 et 3, il n'est théoriquement pas nécessaire d'employer cette transformation plus d'une fois par boucle.

Dans le cadre d'une utilisation conjointe avec l'*interchange* (section 3.4.7), la légalité d'application de cette dernière transformation, conditionnera l'emploi initial du *strip mining*.

3.4.6 Tiling

Le *Tiling* de boucles est utilisé pour valider simultanément les critères 1,2 et 3.

Description de la transformation

Comme le *strip mining* (3.4.5), le *tiling*,qui est plus général, augmente la profondeur d'un nid de boucles en décomposant le domaine d'itérations initial en blocs d'itérations multidimensionnelles appelés *tuiles*. Cette segmentation implique une augmentation du nombre de dimensions d'itérations employées pour le parcours des espaces mémoire. Cependant, à la différence du *strip mining*, l'application $\mathcal{T}$ du *tiling* considère un ensemble de p boucles imbriquées où p est en général supérieur ou égal à 2.

Nous avons la relation :

$$\begin{cases} 2 \leqslant p < p' \\ \prod_i^p |I_i| = \prod_i^{p'} |I'_i| \end{cases} \tag{3.20}$$

Dans sa forme classique, le *tiling* correspond à une généralisation du *strip mining* sur n dimensions du domaine d'itération. Dans le cas d'un espace à deux dimensions, il génère des tuiles, classiquement de forme rectangulaire, voire carrée. L'objectif commun aux deux transformations est :

— de réduire la taille des domaines d'itérations,
— d'augmenter la localité des données,
— de minimiser l'empreinte mémoire pour les boucles internes et
— de contrôler la granularité du parallélisme.

L'utilisation parallèle de transformations telle que le *loop skewing* permet de générer des formes de tuile plus complexes à partir d'un *tiling* rectangulaire. L'objectif est alors de découper le domaine global d'itérations tout en mettant en évidence le plus de parallélisme potentiel possible. Parmi les formes de tuiles les plus couramment utilisées on retrouve :

— le triangle [165],
— le parallélogramme [66, 165],
— le diamant [67, 27],
— l'hexagone [67, 66].

La forme des blocs influence la catégorisation des boucles externes générées en recréant du parallélisme à partir de boucles initialement séquentielles.

Cette transformation est couramment utilisée pour le traitement d'images, car les espaces de données manipulés sont bidimensionnels et s'y prêtent bien. Les données utilisées pour le traitement du signal étant généralement mono-dimensionnel, l'emploi du *strip mining* est souvent suffisant.

Critère d'applicabilité

Le *tiling* est une transformation que l'on qualifiera de transverse car elle impacte les trois critères de placement.

Nous ne détaillerons pas ici son implication sur les critères 2 et 3, ce sujet ayant déjà été abordé dans le cadre du *strip mining*.

En revanche, le *tiling* apporte un autre avantage, sur le critère 1 : le cas où une boucle séquentielle présente trop d'itérations pour être placée sur les instances de *threads* du GPU. Dans ce cas, avec des paramètres de tuilage prenant en considération les dépendances de données, le *tiling* peut dégager une boucle externe parallèle ayant pour vocation d'être placée sur les instances de *blocks* du GPU.

Légalité de la transformation

Si le *strip mining* ne modifiait par l'ordre de parcours du domaine global d'itérations, le *tiling* en revanche privilégie le parcours des itérations par blocs. De ce fait, l'application de cette transformation n'est légale que si les dépendances ne sont pas modifiées. Ainsi dans notre cadre d'application sur GPU, les boucles externes, correspondantes aux instances de *blocks*, doivent toutes être parallèles, ce qui implique l'indépendance des calculs effectués par chaque tuile. Le parcours de ces mêmes tuiles peut alors se faire dans un ordre aléatoire et les boucles externes sont plaçables sur les instances de *blocs* du GPU. Si une boucle externe parallèle ne peut être obtenue, alors cette transformation ne peut être appliquée pour un placement sur GPU.

La méthode hyperplane [84] est souvent utilisée pour définir un paramétrage de *tiling*. Elle permet de dégager une boucle externe séquentielle portant les dépendances et des boucles internes parallèles. L'association de l'*interchange* s'il est légal est donc nécessaire.

Le paramétrage du *tiling* ayant un impact direct sur la taille des tuiles, les paramètres doivent être calculés soigneusement pour respecter les contraintes portant sur la taille des domaines d'itérations de *threads* du GPU (critère 2) ainsi que sur l'empreinte mémoire maximale (critère 3).

Enfin comme pour le *strip mining*, un code de contrôle sur les instances est mis en place dans le cas où le découpage des domaines d'itérations initiaux ne se fait pas selon un multiple de celui-ci. L'objectif est alors d'éviter tout débordement des accès mémoire.

Exemple de transformation

En reprenant l'exemple du *strip mining*, le *tiling* permettrait de redécouper le domaine d'instances des *threads* selon les boucles parallèles $l_{16,17}$. Par induction, quatre boucles parallèles seraient alors générées :

— deux boucles externes imbriquées $l_{16e,17e}$ placées sur les instances de *blocs*,
— deux boucles internes imbriquées $l_{16i,17i}$ placées sur les instances de *threads* telles que $|l_{16i}| \times |l_{17i}| = 1024$.

Quatre boucles ($l_{16e,17e}$ et $l_{6,7}$) sont alors candidates pour être placées sur les *blocs* alors que trois dimensions d'instances seulement sont disponibles. On pourrait de ce fait appliquer un *coalescing* sur $l_{16e,17e}$ ou $l_{6,7}$ afin de réduire l'ensemble à trois boucles candidates.

Ces transformations ne peuvent cependant pas s'appliquer pour l'ensemble de boucles $l_{6,7,18,19}$. Les boucles $l_{18,19}$ sont en effet séquentielles et l'étude approfondie des dépendances indique qu'il n'est pas possible de dégager de boucle parallèle. Les boucles $l_{18e,19e}$ générées seraient alors séquentielles et ne pourraient être placées sur les instances de *blocs*.

Critère de convergence

Dans notre méthodologie, le *tiling* est employé uniquement si un ensemble de boucles parallèles externes peut être extrait d'un ensemble de boucles initiales.

Cette transformation est paramétrée afin que la taille des tuiles résultantes soit conforme à l'ensemble des trois critères de placement. Ainsi, il n'est pas nécessaire d'appliquer consécutivement plusieurs fois un *tiling*.

3.4.7 Interchange

Le *loop interchange*, ou échange de boucles, est utilisé pour valider les critères 1 et 2.Nous utiliserons l'appellation *interchange* pour désigner cette transformation de code dans la suite de ce manuscrit.

Description de la transformation

L'*interchange* permet d'échanger l'ordre de deux boucles dans un nid de boucles imbriquées. Son utilisation revient à modifier l'ordre de parcours des dimensions de ce domaine d'itérations.

Cette transformation est principalement utilisée pour améliorer la localité des données. Elle dépend du *layout* des données, spécifique à chaque langage de programmation [14].

Enfin, l'*interchange* est couramment utilisé pour déplacer :

— les boucles parallèles vers l'intérieur du nid de boucles et

— les boucles séquentielles vers l'extérieur du nid.

L'*interchange* est ainsi bénéfique pour la vectorisation de données et plus généralement pour le parallélisme à grain fin.

Critère d'applicabilité

Dans cette méthodologie, l'*interchange* est employé afin de déplacer, lorsque c'est possible, les boucles séquentielles placées sur les *blocks* du GPU. Les instances de ces derniers étant indépendantes, la présence de dépendances embarquées remet alors en cause le critère 1.

Deux approches sont possibles. Les boucles séquentielles peuvent être déplacées vers l'extérieur du *kernel* ou au contraire au contraire ramenées à l'intérieur du *kernel*, soit dans le corps de ce dernier, soit dans l'ensemble des instances de *threads*.

L'objectif est alors dual :

— constituer un ensemble de trois boucles parallèles correspondant aux *blocks*,

— constituer un second ensemble de trois boucles présentant une localité forte afin de les placer sur les *threads*.

Légalité de la transformation

L'*interchange* modifie l'ordre de parcours des domaines d'itérations. Celui-ci reste légal tant que les dépendances ne sont pas modifiées. On distingue alors les trois cas suivants pour deux boucles imbriquées contiguës :

— L'échange de **deux boucles parallèles** est légal car il n'y a pas de dépendance.

— L'échange d'**une boucle parallèle** avec **une boucle séquentielle** ou de

— **deux boucles séquentielles** est légal lorsque le sens des dépendances est préservé.

14. Les tableaux, en langage *Fortran*, sont stockés par colonne, mais par ligne en *C*.

Critère de convergence

Dans notre méthodologie, l'*interchange* peut être appliqué récursivement sur un nid jusqu'à ce que les contraintes du critère 1 soient atteintes. Son champs d'application est cependant limité par la profondeur du nid de boucles. Celui-ci est de plus restreint pour les boucles séquentielles en fonction de l'interaction entre leurs dépendances respectives.

L'*interchange* modifiant l'ordre des boucles, il conviendra de vérifier que le critère 2 est toujours vérifié. Enfin, cette transformation n'a pas d'impact sur le critère 3, du fait que les accès mémoires contenus dans le corps du *kernel* ne sont pas modifiés.

3.4.8 Unrolling

L'*unrolling*, ou déroulage de boucles, est utilisé pour le critère 2.

Description de la transformation

L'*unrolling* a pour effet de dérouler un certain nombre d'itérations au sein de la boucle concernée. Cette transformation est couramment exploitée afin de réduire le poids des instructions de contrôle de boucle, augmenter la quantité d'instructions embarquées dans le corps de la boucle impliquée et exploiter au mieux le nombre de registres de données disponibles. Le GPU exploitant des instances de *kernels*, celui-ci n'est pas concerné par la problématique des instructions de contrôle de boucle.

L'application de l'*unrolling* engendre entre autre une modification du domaine d'itérations initial I en fonction du facteur de déroulement k tel que :

$$|I'| = \lfloor |I|/k \rfloor$$

Lorsque k n'est pas un multiple du domaine d'itérations initial I, il est possible d'ajouter un nid de boucles parcourant les itérations supplémentaires I'' tel que

$$|I''| = |I| \mod k$$

Dans le cadre des GPU, nous préférons arrondir par excès, selon la valeur de k, le domaine d'instances et ajouter un code de contrôle pour chacune des instances du *kernel*. Le nombre d'itération résultant correspond à :

$$|I'| = \lceil |I|/k \rceil$$

Ce contrôle permet ainsi de maintenir l'ensemble des accès mémoire A^d cohérent en évitant tout débordement mémoire. Cette solution épargne le coût d'un second lancement du même *kernel* pour une quantité réduite de données. L'unique branchement, liée au contrôle ajouté, évite le phénomène de divergence propre au GPU.

La détermination du facteur de déroulage k optimal nécessite une connaissance précise du fonctionnement de l'architecture ciblée. Le choix et l'ordonnancement des instructions employées dans l'ISA ainsi que le nombre de registres disponibles ont un impact sur la détermination de k. Ce sujet est abordé notamment par Sarkar [143].

Dans le cadre des GPUs, on distingue deux cas d'application :

1. l'*unrolling* de la boucle la plus interne du *kernel*
2. l'*unrolling* des autres boucles du *kernel*

Le premier cas modifie la granularité du *kernel*, tandis que le second génère des sous-ensembles de boucles de même profondeur. Ce second cas remet ainsi en cause l'imbrication parfaite de l'ensemble de boucles du *kernel* et de ce fait le critère 1. Il nécessite donc l'utilisation de transformations supplémentaires que nous allons définir.

Critère d'applicabilité

Employé pour améliorer le critère 2, l'*unrolling* permet de réduire la pression sur les différentes dimensions du domaine d'instances des *kernels*. Son application est donc sollicitée lorsque le domaine d'itérations de l'une des boucles d'un *kernel* est supérieur aux contraintes des domaines d'instances des *blocs* et des *threads* du GPU.

De plus, lorsque le *kernel* présente une granularité très fine, l'utilisation de l'*unrolling* permet de mieux masquer les temps de latence élevés de certaines instructions, telles que les accès mémoire, au moyen de l'Instruction Level Parallelism (ILP). L'augmentation de la granularité du *kernel* améliore le temps d'exécution global de ce dernier. Néanmoins cette transformation s'applique au détriment de la quantité de parallélisme du *kernel*. Nous l'appliquons donc en veillant à maintenir, un certain niveau de saturation des instances de *threads*. Cette action est réalisée par le transfert d'itérations des boucles des *blocks* vers celles des *threads*.

Enfin, l'*unrolling* ayant un impact sur le domaine d'itérations d'une boucle, celui-ci peut permettre d'appliquer une fusion (3.4.1) de deux boucles l_a et l_b dont les domaines d'itérations initiaux I_a et I_b diffèrent. Le critère d'applicabilité dans ce cas correspond à $|I_b| = k \times |I_a|$ où k représente le facteur de déroulage de la boucle l_b.

Afin de remédier à la problématique des sous-ensembles de boucles pouvant être générés après l'*unrolling*, l'utilisation conjointe de la *fusion* (section 3.4.1) ou la *fission* (section 3.4.2) sont deux solutions possibles. Celles-ci permettent de maintenir la compatibilité du *kernel* avec le critère 1.

Légalité de la transformation

Par définition, l'*unrolling* ne modifie pas l'ordre d'exécution des instructions et par extension les dépendances existantes. Cette transformation est de ce fait toujours légale, même pour une boucle séquentielle. Elle est donc applicable pour les ensembles d'instances de *blocs* ou de *threads* du GPU.

En revanche, l'utilisation de l'*unrolling* sur les boucles du *kernel* engendre [15] l'apparition de sous-ensembles de boucles incompatibles avec le critère 1. La fusion ou la fission de boucles doivent alors être appliquées afin d'y remédier.

Enfin, le facteur de déroulage k est une variable à considérer pour la légalité d'application de l'*unrolling*. Dans son cadre le plus strict, k doit être un diviseur entier de la taille du domaine d'itération initial.

Exemple de transformation

Dans la figure 3.7, nous considérons le nid de boucles $l_{6,7,8,9}$. Les boucles $l_{6,7}$ sont placées sur les *blocs* du GPU tandis que les boucles $l_{8,9}$ sont placées sur les *threads*. Nous considérons maintenant le cas où le domaine d'itérations des boucles $l_{8,9}$ ne répond plus au critère 2. $|I_8| = 64$ et $|I_9| = 64$ est un exemple engendrant $|I_8| \times |I_9| = 4096$. L'application de l'*unrolling* avec pour paramètre $k = 2$ réduit le domaine d'itérations de ces deux boucles tel que $|I'_8| = 32$, $|I'_9| = 32$ et $|I'_8| \times |I'_9| = 1024$. Ces nouveaux domaines sont alors compatibles avec le critère 2.

Autre exemple, l'application de l'*unrolling* avec pour paramètre $k = 2$ sur la boucle l_{20} permet d'obtenir des domaines d'itérations identiques $l_{18,19}$, $l_{20,21,22}$ mais aussi $l_{20,23,24}$. La boucle l_{20} étant composée de deux itérations seulement, celle-ci disparaît en étant

15. À l'exception de la boucle la plus interne

intégralement déroulée. La fusion de ces trois ensembles de boucles selon le critère de légalité défini dans la section 3.4.1 est alors rendu possible au bénéfice du critère 1.

Critère de convergence

Dans notre méthodologie, l'*unrolling* est appliqué afin de vérifier le critère 2. Cependant, cette transformation ne sera pas appliquée si les registres de chaque cluster viennent à être surchargés. De même, l'apparition de *cache-miss* liés à l'augmentation de la quantité de mémoire sollicitée constitue une limitation à l'application de l'*unrolling*. Les *cache-miss* pourront être vérifié au moyen du *profiler* CUDA *nvprof*. L'utilisation des registres sera obtenu au moment de la compilation avec NVCC, en utilisant l'option $-Xptxas = "-v"$.

3.4.9 Les réductions parallèles

$$I^p \xrightarrow{\mathcal{I}} L^p \xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} M$$

La détection des réductions parallèles est utilisée pour valider le critère 1.

Description de la transformation

Les réductions parallèles exploitent la commutativité et plus particulièrement l'associativité de certaines opérations arithmétiques afin de lever la contrainte propre à l'ordre d'exécution de boucles séquentielles. Ces dernières sont de ce fait assimilables à des boucles parallèles lorsque la dépendance embarquée est uniquement induite par la réduction. Deux types d'applications sont alors possibles : le *pattern* de type *prefix* ou les opérations atomiques.

Le premier est une généralisation du *pattern* de programmation *prefix sum* [76]. Cette approche permet d'extraire du parallélisme à partir d'une boucle séquentielle en décomposant son domaine d'itérations en blocs. Chacun de ces blocs peut être calculé de manière concurrente avec les autres. Cette transformation est utilisée pour générer une boucle interne séquentielle, une boucle intermédiaire parallèle sur les blocs, et une boucle externe séquentielle pour accumuler la réduction. Son application dans le cadre des GPUs Nvidia est notamment décrite par Kirk et Hwu [83] ainsi que Harris et al. [74].

Le second type d'application utilise un ensemble d'instructions spécifiques permettant d'éviter le phénomène de *data race* entre les instances de *threads* utilisant des données communes en mémoire. Ces instructions qualifiées d'*atomic* effectuent une opération mémoire inintéruptible combinant *lecture*, *modification* et *écriture*. La cohérence du résultat est alors assurée par l'exécution sérialisée de ces instructions pour chaque *thread* exploitant le même espace mémoire en dépendance. De plus, la propriété associative de ces opérations permet d'envisager toutes les combinaisons possibles [16] portant sur l'ordre d'exécution des instances de *threads*. Les dépendances associées sont alors levées et l'exécution des *threads* peut être considérée comme parallèle.

En complément des opérations atomiques viennent les instructions *shuffle*, spécifiques à Cuda. Celles-ci permettent de contraindre l'ordre d'exécution des *threads* composant chaque *warp* au moyen de masques d'exécution. En imposant l'ordre d'exécution des *threads*, il est alors possible :

— d'assurer le sens d'une dépendance,
— de limiter les collisions des opérations atomiques concurrentes.

16. On ne considère pas ici la problématique portant sur la précision des calculs flottants.

Cependant, le périmètre d'applicabilité de ces instructions se limite aux *warps* soit 32 *threads*.

Les GPUs exploitant massivement le parallélisme, les boucles séquentielles constituent un écueil pour cette architecture. Ces instructions sont de ce fait importantes pour les GPUs. C'est pourquoi, Nvidia a créé une librairie traitant spécifiquement ce sujet : Cuda UnBound (CUB) [7].

Critère d'applicabilité

Les réductions parallèles sont envisagées lorsqu'au moins une des dimensions propres aux instances de *blocs* présente une dépendance embarquée, rendant ainsi leurs exécutions séquentielles. Par définition, ce cas ne respecte pas le critère 1 de placement sur GPU. Les opérations compatibles avec cette transformation sont identifiées au moyen d'une analyse plus fine, portant sur le contenu des blocs de base b_n identifiés dans la section 3.1.5. Lorsque la dépendance d'une boucle séquentielle est exclusivement induite par une opération associative, l'utilisation des réductions parallèles permet alors de dégager du parallélisme sur l'exécution globale pour les instances de *threads*.

L'application du *pattern prefix* décompose les instances d'un *kernel* en plusieurs appels successifs si la dépendance entre les *threads* est liée à une opération associative. La boucle séquentielle est alors placée à l'extérieur du *kernel* tandis que la boucle parallèle est placée sur le GPU. Chaque appel de *kernel* effectué par la boucle externe porte alors sur un nombre d'instances réduit ce qui implique par rapport à la boucle initiale :

— une empreinte mémoire restreinte,
— un domaine de dépendances restreint.

Les *patterns prefix* ont pour impact une décomposition du *kernel*.

Les opérations atomiques sont aussi employées pour porter sur GPU une boucle initialement séquentielle. Cependant elles se limitent à remplacer les instructions dans le corps d'une boucle par des instructions équivalentes de type atomique [17]. La structure du nid de boucles est de ce fait conservée et l'exécution sérialisée des instructions atomiques permet de maintenir la cohérence des données.

Légalité de la transformation

Les réductions parallèles sont exclusivement appliquées dans le corps d'un *kernel*, pour les instructions correspondant à une opération associative et sollicitant des données mémoire en dépendances au niveau des instances de *threads*. Ce sont généralement des accumulations ou des calculs itératifs.

L'utilisation de réductions parallèles peut nécessiter l'emploi de barrières de synchronisation afin d'assurer la cohérence des données partagées entre les différentes instances de *kernel* exécutées sur le GPU.

Les données communes aux différents appels de *kernels* sont implicitement maintenues cohérentes, car l'exécution d'un *kernel* engendre une barrière, synchronisant la grille de *threads* le composant. Ainsi dans le cadre du *pattern prefix*, l'emploi d'instructions de synchronisation pour les différents appels de *kernel* est inutile. Leur utilisation est rendu nécessaire lorsque les différents appels de *kernel* sont distribués sur plusieurs *pipelines* d'exécution [18] de CUDA. Ces derniers sont utilisés pour la concurrence de *kernels* au sein

17. La liste des opérations atomiques supportées est définie dans la documentation officielle de Cuda [123].

18. CUDA utilise jusque 16 *pipelines* d'exécution simultanément afin de gérer les communications asynchrones ainsi que les appels concurrents de *kernels*

d'un même GPU ou pour la distribution de *kernels* sur de multiples GPUs.

En revanche pour les opérations atomiques, l'emploi d'instructions de synchronisation est nécessaire si une instruction utilise le résultat d'une précédente opération atomique dans un même *thread*. La synchronisation devra alors être placée entre ces deux instructions et la dépendance concernée devra couvrir au plus un *block* de *threads*. Dans le cas où cette dernière se limiterait à la couverture d'un *warp*, l'usage d'instructions *shuffle* sera possible.

Exemple de transformation

Dans le cadre de la fonction $crossBilateralFilter$, nous considérons les boucles imbriquées $l_{18,19}$ et $l_{23,24}$. Ces boucles effectuent une réduction de la totalité des données référencées par $weights[\ ]$ et $weighted\_sum[\ ]$. Chaque élément est additionné et accumulé et le résultat est stocké dans les variables scalaires respectives $weights\_sum$ et $total\_sum$.

L'utilisation d'une opération *atomicAdd* portant sur les instances de *threads*, permet de traiter les boucles $l_{18,19}$ et $l_{23,24}$ comme parallèles. Cependant, l'intégralité de ces opérations atomiques utilise une unique variable scalaire, sérialisant ainsi l'exécution de cette instruction. En conséquence les performances du GPU seront dégradées. L'utilisation du *pattern prefix sum* permet au contraire de limiter ce phénomène.

Nous avons spécifiquement étudié ce sujet et l'avons appliqué au calcul de variance locale [64, 63]. Ces applications de type *Analysis Of Variance (ANOVA)*, sont couramment utilisées en analyse de données. Dans l'article [64], je montre que la méthode *prefix* permet d'augmenter le parallélisme du calcul de variance tout en réduisant le nombre d'opérations arithmétiques et d'accès mémoire.

Critère de convergence

L'emploi des réductions parallèles est déclenché lorsqu'une variable scalaire sert d'accumulateur sur la totalité du domaine d'itération d'une boucle, comme pour l'exemple de la fonction $crossBilateralFilter$.

Les opérations atomiques en concurrence sur une donnée sont exécutées en série. De ce fait, le temps d'exécution du *kernel* peut se retrouver fortement dégradé. L'usage du *pattern prefix* est donc privilégié car celui-ci dégage naturellement du parallélisme de boucle.

3.4.10 Conclusion

Nous avons abordé un ensemble de transformations de code permettant, dans le cadre de notre méthodologie, d'augmenter le nombre de nids de boucles transformables en *kernels*. Notre méthodologie considère en entrée un code source sur lequel est appliqué une série de transformations afin de produire en sortie un nouveau code source optimisé. Chaque application d'une transformation entraîne un cycle de validation des trois critères de placement décrit dans la section 3.3. Cette vérification permet d'améliorer la convergence vers une solution de placement pour les nids de boucles incompatibles.

Afin de faciliter le choix des transformations à appliquer, nous avons synthétisé dans la figure 3.6, l'amélioration pouvant être apportée par chacune des transformations, sur les trois critères de placement.

Nous considérons, en complément, d'autres transformations classiquement utilisées en compilation, telles que **l'expansion** et la **privatisation** de **scalaires** ou de **tableaux**. Celles-ci ne sont pas détaillées mais sont aussi utiles pour déclarer certaines boucles parallèles. C'est le cas notamment pour la fonction $crossBilateralFilter$ qui a servi d'exemple

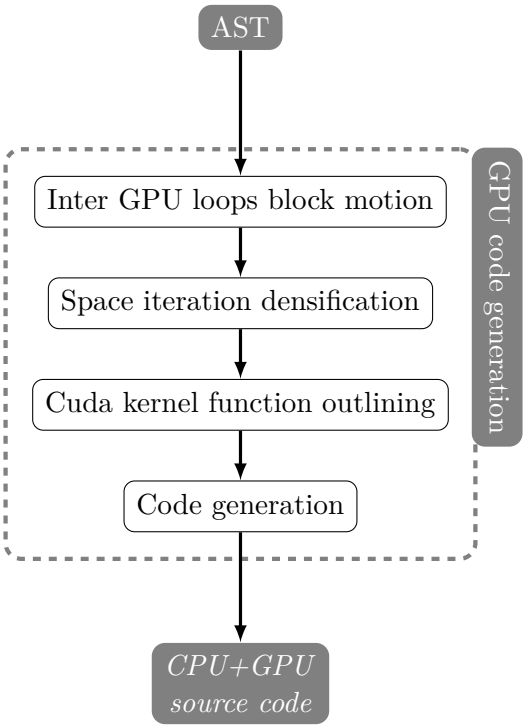


FIGURE 3.8 – Génération de code source pour hôte et accélérateur de type GPU

pour l'ensemble des transformations. Pour rappel, les boucles $l_{6,7}$ ont été préalablement déclarées parallèles en privatisant le tableau *weights*.

Enfin, la **représentation spinale**, introduite dans ce chapitre, permet de visualiser très rapidement l'ensemble des infos nécessaires pour les choix de transformation à appliquer. Son approche interprocédurale, notamment, apporte une aide pour l'application des transformations au-delà des frontières de procédures. L'utilisation de l'***inlining*** peut être nécessaire dans ce cas.

En conclusion, cette étape importante de notre méthodologie permet d'**améliorer le taux d'occupation du GPU**.

3.5 Préparation avant la génération de code

Les nids de boucles adaptées aux contraintes des GPUs ont été identifiées au cours de la section 3.3. Leur nombre a été augmenté au moyen de transformations de boucles décrites dans la section 3.4.

La figure 3.8 représente le processus final de génération de code dans notre méthodologie. Lors de cette étape, les nids de boucles ayant été identifiés comme portables sur GPU sont transformés en *kernels*. Ce processus de génération de code est décomposé en plusieurs étapes. L'ordonnancement des *threads* est abordé dans la section 3.5.1, le déplacement des blocs situés entre les boucles de *kernel* dans la section 3.5.2 et la densification des espaces d'itération dans la section 3.5.3. Enfin l'*outlining* des *kernel*s ainsi que la génération des appels de *kernel* sont traités spécifiquement dans la section 3.6.

Nous générons à la fin de ce processus un nouveau code source compilable, ayant la même fonctionnalité que le code original. Ce code source est dérivé de la représentation spinale, après l'ensemble des transformations décrites dans ce chapitre.

3.5.1 Ordonnancement des instances de *threads*

Le critère 1 portant sur la structure des boucles du *kernel* permet le portage de boucles séquentielles sur les instances de *thread*. Les *threads* pour chaque *block* peuvent être interdépendants. Ce cas n'est pas problématique, car l'ordre d'exécution des instances de *threads* sur les unités SM peut-être contraint. Ainsi l'association de branchements spécifiques à l'identifiant de chaque *thread* permet, grâce à une instruction de synchronisation, d'honorer les dépendances. Enfin, lorsque les dépendances sont limitées à un *warp*, les instructions *shuffle* peuvent être employées.

Ces sujets ont déjà été abordés pour les réductions parallèles dans la section 3.4.9.

3.5.2 Déplacement de blocs inter-boucles GPU

L'ensemble des critères définis dans la section 3.3 permet la définition de *kernels*. Selon le *critère 1* défini dans la section 3.3.1, un *kernel* correspond à un espace d'itérations à six dimensions généré à partir de d boucles imbriquées avec $1 < d \leq 6$. Cependant pour améliorer la fréquence de ces ensembles de boucles, nous avons autorisé la présence de blocs et de branchements entre chacune des boucles. La figure 3.9 illustre un cas concret où plusieurs blocs, b_1 et b_2 en l'occurrence, sont intercalés entre des boucles. La représentation spinale présentée correspond à la fonction *calcIrregularityMat*, qui provient du même algorithme que celui de la figure 3.4. Nous remarquons que l'ensemble de boucles $l_{0,1,2,3}$ est un nid répondant au *critère 1* de portage sur GPU tel que $l_{0,1}$ soit placé sur les blocs du GPU et $l_{2,3}$ soit placé sur les *threads*. Nous supposons pour la suite, les critères 2 et 3 sont vérifiés.

Nous considérons alors la problématique portant sur l'imbrication parfaite des boucles. Dans ce cas de figure, le corps de chaque boucle est composé d'une unique autre boucle. Ce pattern se répète ainsi pour l'ensemble du nid de boucles et la boucle la plus interne, l_3, contient les instructions à exécuter. Dans le processus de portage, l'ensemble de boucles considéré pour le GPU va être transformé en entête de *kernel*. Or, à l'exécution, le GPU génère un ensemble d'instances correspondant au nombre d'exécutions concurrentielles pour ce même *kernel*. Le nombre d'instances est défini par la taille du domaine d'itérations du nid de boucles porté sur GPU [19], I_p dans la formule 3.3. La taille de ce domaine est calculable grâce à la formule 3.21. Rappelons que $|I_i|$ correspond à la taille du domaine d'itérations de la boucle ayant pour identifiant i. p correspond à la profondeur du nid répondant au *critère 1*. Le nombre d'instances résultant est alors valable pour tout élément situé dans le corps de la boucle de *kernel* la plus interne l_{i+p} avec $p = B + T - 1$.

$$\prod_{p=0}^{B+T-1} |I_{i+p}| \tag{3.21}$$

Cependant, les portions de code b_1 et b_2, intercalées entre ces boucles, sont supposés être exécutés pour un nombre plus faible d'itérations, tel que défini par la formule 3.22. Soit d' le niveau de profondeur d'un bloc encastré relativement au *kernel*. Nous avons $d' < B + T - 1$ et $B + T - 1 = 4$. Pour b_2, $d' = 2$. Pour b_1, $d' = 1$. Afin de générer un code correct, nous proposons deux solutions exploitant les transformations de boucles de la section 3.4. La première utilise la fission de boucles pour une extraction des blocs en dehors du *kernel*. La seconde au contraire intègre ces blocs dans le *kernel* par une fusion de boucles.

19. Dans le cadre du critère 1, l'espace d'itérations des boucles concernées est supposé régulier et convexe.

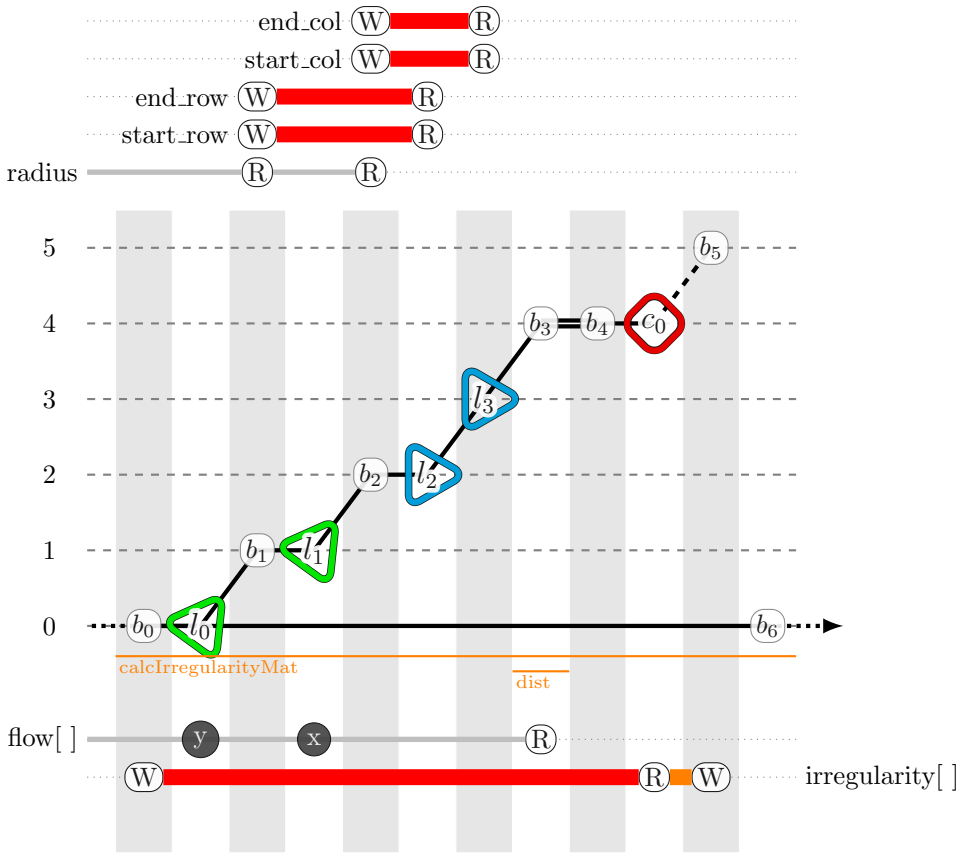


FIGURE 3.9 – Représentation spinale de la fonction *calcIrregularityMat* où les blocs b_1 et b_2 ne permettent pas d'avoir des boucles parfaitement imbriquées

$$\forall d' < B + T - 1, \quad \prod_{p'=0}^{d'} |I_{i+p'}| < \prod_{p=0}^{B+T-1} |I_{i+p}| \tag{3.22}$$

Extraction des blocs du kernel

Il s'agit de la solution la plus simple à mettre en œuvre. Celle-ci emploie la *distribution de boucle*, déjà abordée en section 3.4.2. Cette transformation est applicable à toute portion de code encastrée entre deux boucles et permet de l'extraire du *kernel*. La figure 3.10 présente un exemple d'application pour la fonction *calcIrregularityMat*. Les blocs b_1 et b_2 sont alors exclus du kernel pour être intégrés aux boucles $l_{0.1}$ pour le premier et $l_{0.2}$ et $l_{1.1}$ pour le second. $l_{0.1}$ et $l_{0.2}$ présentent des caractéristiques identiques à l_0. Il en va de même pour l_1 et $l_{1.1}$.

Concernant la légalité d'application de cette transformation, il faut considérer deux cas d'application liés au *critère 1*. Pour rappel, ce dernier spécifie, pour un *kernel*, deux ensembles de boucles distincts. Le premier, plus externe, est composé de boucles parallèles. Le second, plus interne, est composé de boucles quelconques.

Kennedy et McKinley [81] considèrent la distribution de boucle légale, si tous les éléments impliqués dans un cycle de dépendance sont préservés au sein de la même boucle. Ainsi, il n'y a aucune contre-indication à extraire un bloc encastré dans l'une des boucles

parallèles du premier ensemble défini par le critère 1 [20]. L'exemple de la figure 3.10, correspond à ce cas de figure.

Pour les blocs situés entre les boucles du second ensemble [21], leur extraction sera légale si :

— comme dans le cas précédent, la boucle contenant le bloc est parallèle,
— la boucle contenant le bloc est séquentielle mais ce dernier n'est pas impliqué dans les dépendances.

Si cette approche permet de rétablir l'imbrication parfaite des boucles, elle a tendance en contrepartie à augmenter la quantité générale de données transitant en mémoire car, pour préserver le parallélisme, il est nécessaire d'appliquer, quand c'est possible, une expansion de scalaire ou de tableau sur certaines variables. Nous le constatons sur l'exemple de la figure 3.10. Les variables scalaires *start_row*, *end_row*, *start_col* et *end_col* devront être "étendues" en variables multidimensionnelles de la même taille que l'espace d'itérations des boucles englobantes, soit la boucle l_0 pour les deux premières variables et $l_0 \times l_1$ pour les deux dernières. Cette solution est donc à réserver pour les cas où l'exécution du *kernel* est saturée par le flot de calculs, sous peine de dégrader les performances d'exécution du *kernel*.

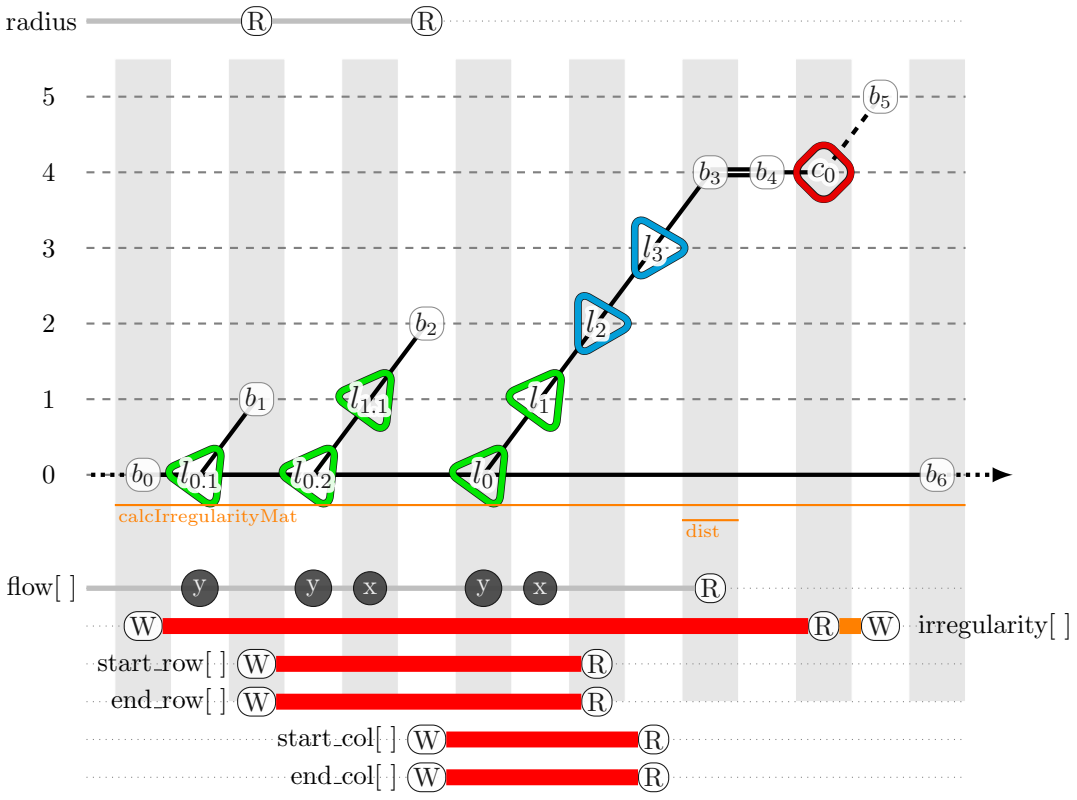


FIGURE 3.10 – Déplacement de blocs encastrés pour la fonction *calcIrregularityMat*. *(Méthode par exclusion)*

20. Cet ensemble de boucles correspond au placement sur les blocs du GPU
21. Cet ensemble de boucles correspond au placement sur les *CUDA cores* du GPU

Inclusion des blocs dans le *kernel*

L'inclusion de codes dans le *kernel* correspond à l'emploi sous condition particulière de la *fusion*. Darte [42] rappelle notamment que la fusion de boucles est légale si le sens des dépendances n'est pas modifié et Kennedy et McKinley [81] considèrent l'application de cette transformation pour des boucles dont le domaine d'itérations présente la même cardinalité.

Dans le cadre de l'inclusion de code dans le *kernel*, nous considérons la fusion de boucles pour des tailles de domaines d'itérations variées. Par ce moyen, il est possible de ramener les portions de code placées entre les boucles définies par le *critère 1*, à l'intérieur du *kernel*. Cette transformation est liée à :

— l'utilisation de barrière de synchronisation, au moyen de l'instruction CUDA *___syncthreads()*,
— l'utilisation de branches de tests, exécutées en fonction des valeurs d'itérations,
— la redondance d'exécution d'instructions.

Si la portion de code ne contient pas de dépendance cyclique, alors il est possible de l'inclure dans le *kernel*, lorsque le sens des dépendances n'est pas modifié. La portion de code sera alors exécutée par chaque *thread* avec, pour conséquence, l'augmentation du nombre initial de communications mémoire, d'instructions et d'opérations arithmétiques exécutées.

Pour limiter cet effet, il est possible de restreindre l'exécution de cette portion de code à un sous-ensemble de *threads* pour chaque *block*. Pour cela, il est possible de définir un masque d'exécution portant exclusivement sur le second ensemble de boucles du *critère 1*. Un branchement conditionnel vérifie alors l'identifiant de chaque *thread*. La diffusion du résultat pour le même ensemble de boucles se fait alors par l'utilisation d'une barrière de synchronisation définie par l'instruction *___syncthreads()* en CUDA. Cette instruction permet ainsi de préserver les dépendances entre *threads*.

La figure 3.11 présente un exemple particulier d'inclusion de blocs pour la fonction *calcIrregularityMat*. Dans la figure 3.9, les blocs b_1 et b_2 présentent une dépendance pour les variables *start_row*, *end_row*, *start_col* et *end_col* au niveau des entêtes de boucles l_2 et l_3. L'inclusion des blocs b_1 et b_2 implique donc une inversion du sens des dépendances pour ces variables. Dans ce cas de figure, il est possible d'approximer le domaine d'itération par une enveloppe convexe et régulière. Le domaine d'itération global est alors régulé par le branchement conditionnel c_{id}, dépendant des blocs b_1 et b_2.

L'approche par inclusion dans le *kernel* peut-être une solution efficace notamment pour les calcul intermédiaires d'indices de tableaux. Cette approche engendre cependant une augmentation du nombre global d'instructions exécutées.

De plus, cette approche a tendance à augmenter les branchements. Ces derniers favorisent alors les divergences de codes qui ont un impact négatif sur l'architecture de type SIMT des GPUs. De plus, l'utilisation de barrières de synchronisation a un impact non négligeable sur l'efficacité du pipeline d'instructions.

3.5.3 Normalisation des espaces d'itération

Pour l'exécution d'un *kernel*, le GPU génère jusque six vecteurs d'instances. Chacun de ces vecteurs contient un nombre n_k fini d'instances, avec $1 \leq k \leq 6$, ayant un identifiant unique dans le domaine $[0; n_k - 1]$. Cette caractéristique correspond au domaine d'itérations d'une boucle normalisée telle que définie dans la section 3.1.7. Or toute boucle utilisée dans un algorithme n'est pas nécessairement normalisée. Il convient de définir l'espace normalisé I_p pour l'ensemble des p boucles portées sur GPU selon les critères 1 et 2 de la section 3.3.

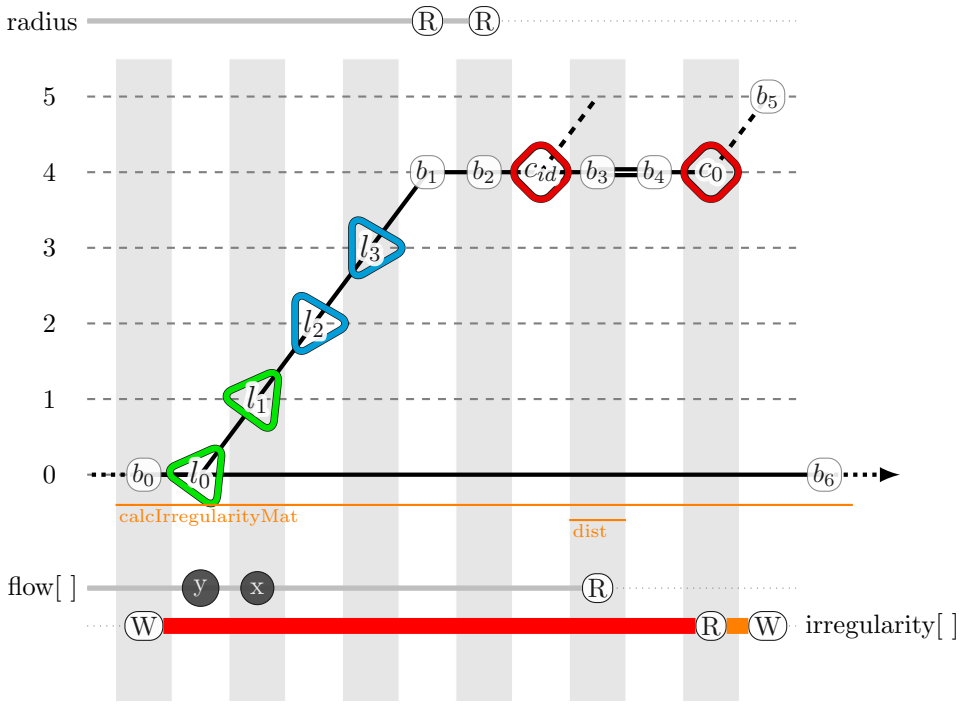


FIGURE 3.11 – Déplacement de blocs encastrés pour la fonction *calcIrregularityMat*. *(Méthode par inclusion)*

On emploie pour ce processus une méthode de normalisation telle que celle de De Michiel et *al.* [48]. Afin de respecter l'aspect fonctionnel de l'algorithme initial, la fonction d'itération $\mathcal{I}$ doit aussi être intégrée au *kernel* pour les accès mémoires.

3.5.4 Linéarisation des accès mémoire

$$I^p \xrightarrow{\mathcal{I}} L^p \xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} M$$

Dans la section 3.1.8, portant sur l'analyse des accès mémoire, nous avons décrit tout accès mémoire comme résultant de la transformation composée $\mathcal{L}(\mathcal{A}) : L^p \xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} \mathbb{N}$, définie selon la formule 3.2. Pour rappel, L^p représente l'espace d'itérations engendré par les p boucles imbriquées. L'ensemble d'arrivée dans $\mathbb{N}$ résulte, quant-à lui, du système d'adressage mono-dimensionnel des unités mémoire. Entre deux se trouve l'ensemble A^d des fonctions d'accès mémoire multidimensionnel du code source.

Les accès multidimensionnels aux données dans les langages *C* et *C++* peuvent être réalisés au moyen de l'indirection et du déréférencement de pointeurs. Le langage CUDA, en tant que surcouche étendant ces deux langages, permet la manipulation de pointeurs au sein d'un *kernel* et ainsi d'employer le déréférencement de pointeurs pour les accès multi-dimensionnels. Cependant, le déréférencement a un coût qui impacte le temps d'exécution des *kernels*. Nvidia recommande donc de linéariser les accès mémoire au sein des *kernels*. L'impact de la linéarisation sur les performances du GPU a été étudié par Amini [16].

Le principe de la linéarisation est de traduire une référence multidimensionnelle en une référence mono-dimensionnelle. Cette traduction dépend de l'allocation de la structure initiale, dans le cas des tableaux, de leur "*layout*". Cette fonction correspond à l'application $\mathcal{L}$ dans notre méthodologie.

Certaines librairies, telle qu'OpenCV au moyen de l'objet *Mat*, offrent une API qui masque le processus de linéarisation, au moyen d'un adressage multidimensionnel. Dans

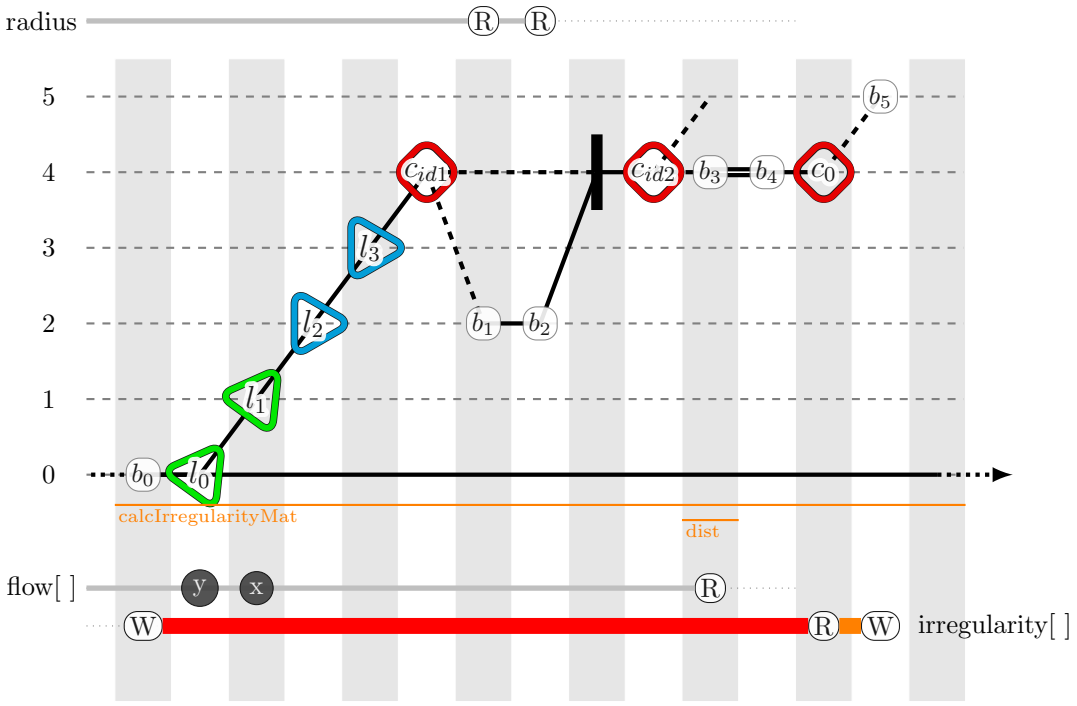


FIGURE 3.12 – Déplacement de blocs interboucles pour la fonction *calcIrregularityMat*. *(Méthode par inclusion et synchronisation)*

notre exemple, la fonction *removeOcclusions*, du listing 3.1, utilise les objets *flow*, *flow_inv* et *confidence*, tous de type *Mat* et représentant des images accédées selon leurs lignes et leurs colonnes. L'accès à leurs données se fait au moyen de la méthode *at* et des variables de boucle r et c, comme coordonnées bidimensionnelles. La manipulation d'images dans ce repère apporte un avantage certain quant-à la lisibilité du code source et plus particulièrement pour les calculs de voisinage. Les images ayant une taille fixe, la fonction de linéarisation est définie par la fonction $\mathcal{L}(r, c) : r \times cols + c$ où $cols$ est le nombre total de colonnes de l'image.

Il existe d'autres méthodes propres à CUDA pour la gestion des accès multidimensionnels. Nous reviendrons sur ce sujet en abordant les optimisations et spécialisations pour GPU, décrites dans la section 5.1.

3.6 Génération de code pour GPU

$$I^p \xrightarrow{\mathcal{I}} L^p \xrightarrow{\mathcal{A}} A^d \xrightarrow{\mathcal{L}} M$$

Guelton et *al.* [68] ont défini une méthode pour la génération automatique de code sur architecture hétérogène en considérant notamment l'architecture GPU. Leur procédure passe par quatre étapes. La première utilise une fonction d'estimation du temps d'exécution des boucles afin de déterminer celles pouvant être accélérées sur GPU. Ensuite vient le processus d'*outlining*, extrayant le *kernel* de l'application et segmentant les données locales, des données externes du *kernel*. La troisième étape porte sur la génération des transferts pour les données externes au *kernel*. Enfin la dernière étape s'assure en appliquant un *symbolic tiling* que le code généré est compatible avec la taille de l'espace mémoire disponible sur l'architecture cible. Leur approche est applicable pour un nid de boucles parfaitement imbriqué. L'absence de conditions pour l'application du *tiling* laisse

supposer que les boucles considérées sont exclusivement parallèles. Enfin, leur exemple sur GPU ne traite que la boucle la plus profonde, qui est parallèle. L'*outlining* est alors appliqué pour le corps de cette boucle.

Notre procédure de génération de code pour GPU est décomposée en quatre étapes. Elle suit logiquement les mêmes grandes lignes mais présente quelques différences. Notre méthodologie considère un nid de boucles sans supposition sur sa structure (domaine d'itérations, imbrication, dépendances embarquées). Nous n'utilisons pas d'estimation du temps d'exécution mais une analyse dynamique plus précise décrite en section 3.2. De plus, un processus visant à détecter les boucles dont la performance est dégradée sur GPU est exécuté comme ultime étape de la méthodologie. Nous considérons cette étape comme externe au processus de génération de code et sa description est faite en section 3.7. Notre critère 3 décrit au cours de la section 3.3.3 vérifie déjà en amont dans la méthodologie, la capacité des espaces mémoire à supporter l'empreinte mémoire des *kernels*.

La première étape de notre processus de génération de code utilise l'*outlining* pour la **création des *kernels*** et l'**identification des données** utilisées. Elle est décrite en section 3.6.1.

Le GPU est un accélérateur et nécessite un processeur hôte pour sa mise en œuvre. La mémoire de l'hôte et celle de l'accélérateur sont la plupart du temps distinctes. Dans le cas de CUDA, l'Unified Virtual Addressing (UVA) profite des capacités étendues de l'adressage 64 *bits* pour considérer comme un unique espace d'adressage les mémoires de l'hôte et de l'accélérateur. De même, certains SOCs intègrent un CPU, un GPU et une mémoire partagée. C'est le cas de la majorité des CPUs Intel (1.2.1), les APUs chez AMD (1.2.2) et la gamme des processeurs Tegra chez Nvidia (1.2.3). Dans les deux cas (mémoire partagée/distribuée), l'espace mémoire GPU du *context* CUDA est distinct de celui du reste de l'application. De plus, le système de partage de données *sans copie* fourni par Nvidia désactive les mémoires cache de l'hôte et de l'accélérateur car il n'existe pas de mécanisme de cohérence entre elles. En pratique la bande passante mémoire est significativement réduite. Nous n'avons donc pas considéré cette solution dans notre méthodologie. Cela nous oblige à toujours considérer l'**allocation des espaces mémoire pour l'accélérateur** dans la section 3.6.2 et la **création des communications hôte/accélérateur** dans la section 3.6.3. Ces deux procédures constituent respectivement les étapes 2 et 3 de notre processus de génération de code. En reprenant la formule 3.3, nous utilisons, pour ces deux parties, les domaines d'accès mémoire M définis en section 3.1.8 avec les références des p dimensions propres au domaine d'instances de chaque *kernel*.

Enfin, la dernière étape, développée dans la section 3.6.4, porte sur la **génération des appels de *kernels*** dans le code de l'hôte.

3.6.1 *Outlining* des *kernels* cuda

Le processus d'*outlining* permet d'encapsuler une partie de code source identifié au sein d'une fonction. Afin de définir les paramètres de la fonction, il faut considérer l'ensemble des accès mémoire et des variables non privées à cet extrait défini par (3.23) selon Guelton et *al.* [68].

$$\begin{aligned} ExternalVars(S) = ReferencedVars(S) - \\ (DeclaredVars(S) \cup PrivateVars(S)) \end{aligned} \tag{3.23}$$

Dans cette formule :
— S correspond à un *statement*,

— $ReferencedVars$ représente l'ensemble des variables référencées par S,
— $DeclaredVars$ représente l'ensemble des variables déclarées localement dans S,
— $PrivateVars$ représente l'ensemble des variables privatisées dan S et
— $ExternalVars$ correspond à l'ensemble des paramètres formels de la fonction générée.

Liao et *al.*[96] ont élaboré un processus d'*outlining* pour les langages *C* et *C++* dans le cadre du compilateur *Rose*. Ils abordent ainsi la classification des régions mémoires impactées ainsi que l'élimination des déréférencements de pointeurs inutiles.

La programmation sur GPU est réalisée au moyen de *kernels*. De ce fait, l'*outlining* constitue une étape incontournable dans le processus de portage depuis un code CPU. Dans notre méthodologie, la phase d'*outlining* exploite les nids de boucles identifiés comme légalement plaçables sur GPU au cours des sections 3.3 puis 3.4. En sections 3.5.3 et 3.5.2, les espaces d'itération de ces nids ont été normalisés et rendus parfaitement imbriqués. À cette étape de la méthodologie, les nids sont considérés comme transformables en *kernel*. Le processus d'*outlining* permet alors :

— d'encapsuler dans un *kernel* le corps de la boucle GPU la plus interne,
— de générer les paramètres de ce *kernel* définis par (3.23),
— de remplacer les entêtes de boucles GPU par des paramètres d'instances.

3.6.2 Allocation des tableaux

Les données échangées entre l'hôte et l'accélérateur ont été précédemment identifiées lors du processus d'*outlining* en section 3.6.1. Les espaces mémoire M_{acc} devant être explicitement alloués correspondent aux régions des données $allocation(k)$ définies par (3.24).

$$allocation(k) = referenced(k) - declared(k) \tag{3.24}$$

On considère :

— $referenced(k)$ comme l'ensemble des tableaux accédés dans le *kernel* k et
— $declared(k)$ comme l'ensemble des tableaux déclarés localement dans k.

Pour les GPU, les tableaux privatisés doivent être déclarés dans le *kernel* et font donc partie de $declared(k)$. Les variables scalaires ne nécessitent pas de déclaration explicite particulière. Enfin, l'allocation des tableaux est effectuée avant toute exécution de *kernel*.

Ces espaces mémoires sont libérés :

— après l'exécution du *kernel* pour les régions accédées exclusivement en lecture,
— après le transfert des données $M_{acc} \xrightarrow{\mathcal{E}^{h \leftarrow d}} M_{host}$ (section 3.6.3) pour les régions accédées en écriture.

Le respect du critère 3, pour chaque *kernel*, garantit un espace mémoire suffisant.

L'allocation de données ayant un coût non négligeable au niveau du temps d'exécution, nous considérons, en complément, l'optimisation suivante. Pour deux *kernels* k_a et k_b, exécutés successivement sur un même GPU tel que $k_a \prec k_b$, si l'union de leurs empreintes mémoire respectives M_{k_a} et M_{k_b} est plus petite que la taille de la mémoire considérée selon (3.25), alors :

— la libération des données mémoires de k_a est omise et
— l'allocation des données communes à k_a et k_b n'est de ce fait pas nécessaire.

$$|M_{k_a} \cup M_{k_b}| < |M_{acc}| \tag{3.25}$$

Par application récursive pour l'ensemble des *kernels*, cette optimisation permet de minimiser le nombre d'allocations mémoire tout en maximisant l'usage de la mémoire exploitée. En conséquence, le temps d'exécution du programme est réduit.

3.6.3 Création des communications hôte/accélérateur

La fonction $\mathcal{E}^{h\to d}$, décrite en (3.26), correspond au transfert de données depuis l'espace mémoire M_{host} de l'hôte (le CPU) vers l'espace mémoire M_{acc} de l'accélérateur (le GPU). La fonction $\mathcal{E}^{h\leftarrow d}$ décrite en (3.27) correspond au transfert de données inverse.

$$M_{host} \xrightarrow{\mathcal{E}^{h\to d}} M_{acc} \tag{3.26}$$

$$M_{acc} \xrightarrow{\mathcal{E}^{h\leftarrow d}} M_{host} \tag{3.27}$$

Le sujet des communications de données hôte/accélérateur a été étudié par Amini [16]. Celui-ci exploite en particulier l'analyse des régions de tableaux développée par Creusillet et Irigoin [37]. L'analyse qu'il a développé permet de réduire la quantité de communications de données hôte/accélérateur.

Les communications induites de (3.28) sont générées :

— de l'hôte vers l'accélérateur [22] ($\mathcal{E}^{h\to d}$) pour les données utilisées en lecture dans le *kernel* et correspondant aux régions IN,

— de l'accélérateur vers l'hôte ($\mathcal{E}^{h\leftarrow d}$) pour les données utilisées en écriture dans le *kernel* et correspondant aux régions OUT.

$$exchange(k) = referenced(k) - declared(k) \tag{3.28}$$

Les transferts de données de type $\mathcal{E}^{h\to d}$ sont effectués entre l'allocation de l'espace mémoire concerné et le lancement du *kernel*. Les transferts de données de type $\mathcal{E}^{h\leftarrow d}$ sont effectués entre l'exécution du *kernel* et la libération de l'espace mémoire concerné.

Comme pour l'allocation des espaces mémoires (3.6.2), nous considérons l'optimisation visant à réduire les transferts de données entre l'hôte et l'accélérateur afin d'améliorer le temps d'exécution du programme. Ainsi, pour tout *kernel* k_i ($i \in \mathbb{N}$), nous générerons :

— un transfert mémoire $\mathcal{E}^{h\leftarrow d}$ lorsque des données correspondant à une région OUT du kernel k_i sont utilisées par une instruction exécutée ensuite par le processeur hôte.

— un transfert mémoire $\mathcal{E}^{h\to d}$ lorsque des données correspondant à une région IN du kernel k_i ont été précédemment allouées ou modifiées par une instruction exécutée par le processeur hôte.

Cette optimisation permet ainsi de minimiser les transferts de données tout en respectant la cohérence entre les deux espaces mémoires.

3.6.4 Génération des appels de *kernel*

La dernière étape du processus de génération de code consiste à générer les appels de *kernel*. Les paramètres sont alors spécifiés en fonction des données placées sur le GPU. L'exécution d'un *kernel* remplaçant un nid de boucles, il reste à paramétrer la répartition des instances d'exécution pour chaque *kernel* généré. Cette information est donnée par les résultats du processus de normalisation des espaces d'itérations de la section 3.5.3.

3.7 Mécanisme de validation/invalidation de *kernels*

Dans la section 3.6, l'ensemble des *kernels* ainsi que leurs appels ont été générés. Comme le GPU est un accélérateur, l'initialisation des espaces mémoire a été effectué

22. *device* selon la nomenclature de CUDA.

ainsi que les communications correspondant aux échanges de données entre l'hôte et l'accélérateur.

L'application une fois compilée, prendra place sur une architecture globale qualifiée d'hétérogène, car utilisant un ou plusieurs GPUs orchestrés par un processeur hôte, le CPU. La conception des deux architectures étant différente, le portage sur GPU d'un algorithme initialement défini pour CPU, engendre inévitablement des temps d'exécution différents. Cependant, le résultat demeure fonctionnellement identique. De plus, l'utilisation d'une architecture globale hétérogène implique des coûts de communication et de synchronisation non négligeables, notamment dans le cas où la mémoire est distincte entre l'hôte et l'accélérateur.

Pour ces raisons et pour éviter une dégradation des performances, cette méthodologie passe par une ultime étape de validation/invalidation de *kernels* GPU. Celle-ci succède à l'analyse de code dynamique de la section 3.2 en comparant les métriques précédemment collectées avec le temps d'exécution de chaque *kernel*. Pour mémoire, dans la figure 3.5, nous avions obtenu le temps d'exécution de chaque fonction, de chaque boucle exécutée ainsi que de l'application dans sa globalité. Ainsi, tout *kernel* dont le temps d'exécution est supérieur au temps initialement collecté donnera lieux au remplacement du code GPU concerné au profit du code CPU d'origine. Les données concernant le temps d'exécution des *kernels* peuvent être obtenues après compilation en utilisant le *profiler* Nvidia *nvprof* ou en instrumentant la partie de code CPU au moyen d'instructions *cudaEvent*. Ces instructions permettent aussi de collecter la date à laquelle s'est passée un évènement sur le GPU.

3.8 Conclusion

Nous avons décrit dans ce chapitre, l'ensemble des étapes de la méthodologie que nous avons définie pour porter un algorithme séquentiel sur une cible hétérogène : CPU + GPU. Cette méthodologie prend en entrée un code source séquentiel, et génère en sortie, un code source fonctionnellement identique et utilisant la programmation par *kernel* du langage CUDA.

Notre méthodologie débute par une phase d'**analyse statique** du code source, afin d'en extraire ses caractéristiques. L'algorithme est alors labellisé en blocs, boucles et branchements. Les accès mémoire ainsi que les dépendances sont identifiés du code original. Les appels de fonctions sont identifiés afin de donner une dimension **interprocédurale** à notre méthodologie. Le code source étudié est alors transformé, de manière automatique, en une représentation intermédiaire originale que nous avons nommé **représentation spinale**. Celle-ci combine l'AST, le graphe d'appels des fonctions, le graphe de dépendances et le graphe des appels. Cette représentation graphique facilite l'utilisation de cette méthodologie pour des programmes complexes.

Une **analyse dynamique** vient compléter la phase d'analyse statique et collecte des temps d'exécution. Cette analyse permet de vérifier que les performances des portions de code portées sur GPU sont bien améliorées. Cette solution nous affranchit d'un modèle de prédiction devant évoluer avec les générations successives de GPU.

Les nids de boucles du code source initial sont ensuite confrontes à **trois critères de placement** que nous avons spécifiés. Ceux-ci portent sur la structure, la profondeur et le parallélisme potentiel des nids de boucles, le nombre d'itérations de chacune des boucles concernées et l'empreinte mémoire des nids de boucles. Ces critères permettent de décider de la compatibilité des nids de boucles avec le modèle de fonctionnement par *threads* des GPUs.

Les nids de boucles incompatibles avec ces trois critères subissent un processus itératif

de **transformations** afin de remplir ces critères. Ce processus permet ainsi d'**augmenter le taux d'utilisation du GPU**. Suite à l'application de ces transformations, les nids de boucles ne pouvant toujours pas valider les critères de placement sur GPU sont maintenus sur CPU.

Une fois les transformations appliquées et les critères de placement validés, les nids de boucles GPU sont modifiés avant la génération de code. Les contraintes d'ordonnancement sont appliquées pour les boucles correspondant aux instances de *threads*, les espaces d'itérations sont normalisés et les boucles sont rendues parfaitement imbriquées.

La phase de génération de code utilise un processus d'***outlining*** pour chaque nid de boucles GPU afin de générer les *kernels*. Cette phase est complétée par la **génération des allocations mémoire** sur l'accélérateur et des **communications de données hôte/accélérateur**. Enfin, la **génération des appels de fonctions** dans le code hôte conclut cette partie.

En dernier lieu, un processus de **validation des *kernels*** est utilisé. Son rôle consiste à vérifier que l'exécution de chaque *kernel* apporte une **réduction du temps d'exécution** par rapport à l'exécution sur CPU.

Dans le chapitre 5 nous complétons cette méthodologie en spécialisant davantage le code et en utilisant des optimisations avancées. Nous considérons, en particulier, certaines fonctionnalités fournies par CUDA afin de maximiser l'utilisation des capacités des GPUs Nvidia.

Enfin, dans le chapitre 4, nous validons notre méthodologie sur l'intégralité de l'algorithme *simpleFlow*.

À ma connaissance, la méthodologie détaillée que nous avons proposée dans ce chapitre est la première remplissant les critères suivants :

— elle est utilisable pour des langages classiques y compris *C* et *C++*,
— elle traite des noyaux pas nécessairement réguliers, au-delà des codes polyédriques,
— elle propose un processus de transformations adaptées et optimisées au GPU,
— elle utilise une représentation intermédiaire originale et efficace pour l'intégralité du programme.

Cette méthodologie a permis de placer sur GPU, 4 *kernels* sur 6 nids de boucles candidats initialement. Pour les architectures utilisées, nous avons obtenu un *speedup* de 5.0 et de 25.0 (chapitre 4).

Chapitre 4

Évaluation de la méthodologie de placement sur GPU

Sommaire

Nous avons détaillé dans le chapitre 3 notre méthodologie de portage d'algorithmes sur accélérateurs de type GPU. Celle-ci permet, à partir d'un algorithme séquentiel, de transformer certains nids de boucles en *kernels*. Pour cela, nous avons défini trois critères de placement portant sur la structure des nids de boucles, la taille de leur domaine d'itérations ainsi que leurs empreintes mémoire. En complément, nous avons défini un ensemble de transformations de code visant à améliorer le nombre de nids de boucles candidats mais aussi la qualité du placement sur GPU. Afin d'alimenter ce processus de placement, plusieurs analyses statiques sont utilisées. Elles permettent de générer une représentation graphique du code facilitant le déroulement de cette méthodologie. Enfin, les analyses dynamiques permettent de vérifier que chaque *kernel* généré engendre bien un gain en terme de temps d'exécution.

Dans ce chapitre, nous évaluons cette méthodologie de placement sur GPU. Dans un premier temps, nous introduisons dans la section 4.1 les architectures utilisées. Nous présentons ensuite dans la section 4.2, les deux applications ayant servi à l'évaluation de notre méthodologie. Enfin, nous détaillons les résultats pour ces deux applications dans les sections respectives 4.3 et 4.4. Nous évaluons en particulier les analyses statique et dynamique, les critères de placement ainsi que les transformations de code.

4.1 Architectures expérimentales utilisées

Nous avons utilisé deux architectures GPU Nvidia pour les expérimentations. Ce choix est justifié, dans le chapitre 1, par la présence forte de Nvidia dans le secteur industriel. Ce concepteur offre des solutions pour une large gamme d'application telles que l'embarqué ou le calcul scientifique. L'évaluation de notre méthodologie compare les résultats obtenus pour une architecture GPU "classique", sur une plateforme de type *workstation* (Endicott), avec ceux d'une architecture GPU basse consommation, sur une plateforme embarquable (Jetson TX1).

Les caractéristiques de la plateforme **Endicott** sont présentées dans la section 4.1.1 et celles de la **Jetson TX1** dans la section 4.1.2.

4.1.1 Endicott

Endicott est une *workstation*, équipée d'un processeur Intel $64\,bits$ de génération *Haswell*. Il s'agit d'un **Core i7 4770s** composé de 4 cœurs physiques et de 8 cœurs logiques grâce à la technologie d'*HyperThreading*. Sa fréquence de fonctionnement nominale est de $3.1\,GHz$ et peut être augmentée grâce à la technologie *Turbo Boost* à $3.9\,GHz$ sur des périodes de temps réduites. Sa puissance de calcul maximale pour des opérations flottantes simple précision est de $\mathbf{499,2\,GFlops}$. Son Thermal Design Power (TDP) est de $\mathbf{65\,W}$. Enfin sa bande passante mémoire théorique est de $\mathbf{25,6\,GB.s^{-1}}$. La puissance de ce processeur hôte permet d'exploiter le GPU de cette plateforme en minimisant le risque de sous-alimentation des *pipelines* de communication entre l'hôte et l'accélérateur GPU.

Cette station de travail est aussi équipée d'un GPU **Quadro K2000** développé par Nvidia. Cette carte est basée sur une architecture Kepler [129] GK107. La K2000 intègre deux unités Next Generation Streaming Multiprocessor (SMX) (visibles dans la figure 4.1) de 192 cœurs portant ainsi le total à 384 cœurs Cuda par processeur. La puissance de calcul maximale pour les opérations flottantes simple précision est de $\mathbf{732,67\,GFlops}$. La K2000 embarque $2\,GB$ de mémoire dédiée dont la bande passante maximale est de $\mathbf{64\,GB.s^{-1}}$ pour un bus mémoire de $128\,bits$. Enfin, le TDP de ce GPU est de **51W**.

4.1.2 Jetson TX1

La Jetson TX1 est une plateforme d'évaluation conçue par Nvidia pour des applications embarquées. Elle intégre un processeur basse consommation Tegra X1 [121] basé sur le SOC T210 de la marque. Ce dernier embarque quatre cœurs **ARM A57** cadencés à $1.9\,GHz$. Afin de réduire la consommation énergétique globale du SOC T210, les cœurs A57 peuvent être désactivés pour utiliser en remplacement quatre cœurs ARM A53 cadencés à $1.3\,GHz$. La sélection des cœurs A57 ou A53 sera déclenchée, à l'initiative exclusive du processeur, en fonction du taux d'occupation de ces ressources. La puissance de calcul maximale est alors de $\mathbf{60,8GFlops}$.

Sur ce même SOC est aussi incorporé un GPU basé sur l'architecture *Maxwell* [130] **GM20B**. La figure 4.2 illustre la configuration de l'un des Maxwell Streaming Multi-

FIGURE 4.1 – Vue d'un cluster SMX de l'architecture Nvidia Kepler de première génération utilisée pour les Quadro K2000

processor (SMM) spécifique à cette architecture. Chacun d'entre-eux est composé de 128 cœurs CUDA dont la fréquence de fonctionnement est de $1GHz$. Deux unités SMM sont présentes dans le processeur de la Tegra X1 pour un total de 256 cœurs CUDA. La puissance de calcul maximale pour la partie GPU est de **512 GFlops** pour des opérations flottantes simple précision.

En complément, une unique **mémoire unifiée** de $4\,GB$ est partagée par l'ensemble des processeurs précedemment cités. Sa bande passante est de $\mathbf{25.6\,GB.s^{-1}}$ pour un bus mémoire de $64\,bits$.

Le **TDP global** du processeur Tegra X1 est de **15 W**.

4.1.3 Comparaison des architectures

Le tableau 4.1 résume les principales caractéristiques pour les deux plateformes précédemment décrites. Celles-ci présentent certaines caractéristiques intéressantes pour nos expérimentations.

Le domaine d'application, notamment, n'est pas le même pour les deux plateformes. La Jetson TX1 répond à des problématiques de consommation énergétique réduite propre à l'embarqué. Endicott, au contraire, est une plateforme de type *workstation* favorisant les performances calculatoires.

Au niveau de la puissance des processeurs hôte, l'ARM A57 du SOC T210 a une

FIGURE 4.2 – Vue d'un cluster SMM de l'architecture Nvidia Maxwel de seconde génération utilisée pour la Tegra X1

puissance de calcul plus faible(60 $GFlops$) que le Core i7 d'Intel(500 $GFlops$). Le pilotage du GPU par l'A57 est donc plus délicat avec un risque de sous alimentation des pipelines de communication hôte/accélérateur.

Concernant les GPUs, l'architecture *Maxwell* de la Tegra X1 est plus récente que l'architecture *Kepler* de la Quadro K2000. Cela se traduit par quelques différences notamment au niveau de la disposition des mémoires cache. Dans le cas de l'architecture *Kepler*, le cache L1 est partagé avec la *shared memory*. Sur l'architecture *Maxwell*, le même cache est communalisé avec le cache de la *texture memory*. Le nombre de canaux d'accès, pour ce dernier cache, est aussi moins élevé sur l'architecture *Maxwell*. Cependant la taille globale pour chaque SMM ne varie pas. Nous évaluons en particulier les conséquences sur les temps d'accès aux différents espaces mémoires dans la section 5.1.

Les deux GPUs intègrent deux unités SM composées chacune de quatre *warp schedulers* pour huit *instruction dispatch units*. Les fréquences de fonctionnement étant similaires et le cache d'instructions restant identique, le débit de placement des *warps* ne devrait ainsi pas évoluer. L'utilisation des *warp schedulers* et des *instruction dispatch units* pour du

	Endicott		Jetson TX1		
	cpu	gpu	cpu	cpu	gpu
Concepteur	Intel	Nvidia	ARM	ARM	Nvidia
Architecture	Haswell	GK107	v8-A	v8-A	GM20B
Catégorie	Core i7	Quadro	Cortex	Cortex	Tegra
Modèle	4770s	K2000	A53	A57	T210
Compute Capability		3.0			5.3
Nb. cœurs	4	384	4	4	256
Nb. threads	8	4096	4	4	4096
Fréquence Nom.(GHz)	3.1	0.954	1.3	1.9	1.0
Fréquence Max.(GHz)	3.9				
Puissance calc. SP Max.(GFlops)	**499.2**	**732.67**	**41.6**	**60.8**	**512**
L1I cache(KB)	32	16	48		16
L1D cache(KB)	32	16/32/48	32		24
L2 cache(MB)	1	0.256	2		0.256
Qté. mémoire(GB)	8	2	4		
Type	DDR3	GDDR5	LPDDR4		
Fréquence(GHz)	0.8	1.0	1.6		
Data Rate	2	4	2		
Bus mémoire(bits)	128	128	64		
Bande passante(GB/s)	**25.6**	**64**	**25.6**		
Gravure(nm)	22	28	20		
TDP max(W)	65	51	15		

TABLE 4.1 – Tableau récapitulatif des architectures expérimentales utilisées.

parallèlisme *coarse grain* est étudié en particulier dans la section 5.2.

Comparativement à l'architecture *Maxwell*, l'architecture *Kepler* embarque 64 *CUDA cores* supplémentaires par unité SM. Nous noterons aussi que le nombre de *texture units* est moins élevé sur l'architecture *Maxwell* ce qui peut se traduire par une bande passante plus faible pour la *texture memory*, lors de calculs d'interpolation ou de réplication de données.

Les performances générales de ces GPUs sont plutôt limitées, quand on les compare, à génération identique, aux plus puissants GPUs tels que les Quadro K6000 et M6000. Les effets d'un mauvais placement sont ainsi amplifiés par une bande passante mémoire et une fréquence de fonctionnement plus faibles.

La Jetson TX1 présente une mémoire homogène unifiée, réduisant les coûts de transfert mémoire. Endicott au contraire présente une mémoire hétérogène. Les transferts mémoire transitent, dans ce dernier cas, par le port Peripheral Component Interconnect express (PCIe). Ce dernier peut constituer un facteur limitatif.

Enfin, pour plus de détails sur l'architecture des deux GPUs, nous avons ajouté les caractéristiques du SMX de la K2000 et du SMM de la TX1, dans le tableau 4.2. La principale différence entre ces deux processeurs porte sur la quantité de registres utilisables et sur les caractéristiques de la *shared memory*.

Pour les deux architectures :

— chaque *warp* est constitué au maximum de 32 *threads* et

— chaque *block* est constitué au maximum de 1024 *threads*.

Ces caractéristiques sont identiques aux paramètres utilisés pour le second critère de pla-

Compute Capability	3.0	5.3
SM Version	sm_30	sm_53
Threads / Warp	32	32
Warps / Multiprocessor	64	64
Threads / Multiprocessor	2048	2048
Thread Blocks / Multiprocessor	16	32
Shared Memory / Multiprocessor (bytes)	49152	65536
Max Shared Memory / Block (bytes)	49152	49152
Register File Size / Multiprocessor (32-bit registers)	65536	65536
Max Registers / Block	65536	32768
Register Allocation Unit Size	256	256
Register Allocation Granularity	warp	warp
Max Registers / Thread	63	255
Shared Memory Allocation Unit Size	256	256
Warp Allocation Granularity	4	4
Max Thread Block Size	1024	1024
Concurrent kernel execution	16	16
Shared Memory Size Configurations (bytes)	49152 (32768) (16384)	65536
Warp register allocation granularities	256	256

TABLE 4.2 – Caractéristiques du SM 3.0 et du SM 5.3 *Source: Nivida [119]*

cement dans la section 3.3.2.

De même, pour le premier critère de placement (section 3.3.2), les 3 dimensions utilisées pour la répartition des *blocks* et des instances de *threads* sont adaptés à ces deux architectures. De ce fait, nous considérons pour ce critère $B = 3$ et $T = 3$.

Enfin, pour le troisième critère de placement, nous considèrons pour les deux architectures $M^{acc/global} = 1.75\,GB$ comme quantité de mémoire allouable. La mémoire globale ($4\,GB$) de la Jetson TX1 étant partagée entre l'hôte et l'accélérateur, nous avons retenu une répartition équitable ($2\,GB/2\,GB$) de celle-ci. La mémoire de la Quadro K2000 est de $2\,GB$. Pour les deux plateformes, nous préservons $250\,MB$ pour la session graphique.

4.2 Applications étudiées

Deux algorithmes ont été utilisés pour l'évaluation de notre méthodologie. L'algorithme de flot optique *simpleflow* présenté en section 4.2.1 et l'algorithme de variance locale présenté en section 4.2.2.

4.2.1 Algorithme de flot optique

Nous avons utilisé pour nos expérimentations l'algorithme du *simpleFlow* de Tao et al.[150]. L'intégralité du code source, ayant servi de référence, est contenu dans l'annexe A. Celui-ci provient du dépot officiel des contributions [2] à la bibliothèque de traitement d'images OpenCV. Cet algorithme effectue, à partir de deux images distinctes de même taille, le calcul du flot optique dense. Le résultat est alors représenté dans une table contenant pour chaque pixel, son déplacement horizontale et verticale entre les deux images en entrée, dans les coordonnées pixeliques de ces dernières.

Cet algorithme a été retenu car il présente de nombreuses caractéristiques intéressantes pour l'évaluation de notre méthodologie :

- Nous sommes étranger au développement de cet algorithme et n'avons de ce fait eu aucune influence sur sa conception.
- Son développement n'a pas été conçu pour du *benchmarking*. C'est un cas concret d'application de traitement d'image en C++. Le C++ est de plus en plus utilisé pour les applications industrielles.
- Ses 600 lignes de codes représentent un algorithme de complexité importante.
- Son découpage en multiples sous-fonctions permet d'étudier le comportement interprocédural de notre méthodologie.
- Les multiples nids de boucles ayant en moyenne 6 niveaux de profondeur, représentent une source variée de *kernels* potentiels.
- Les domaines d'itérations des boucles de l'algorithme ont des tailles variées.
- Le code source présente du contrôle dynamique avec :
  - des bornes de boucles variables et
  - des branchements non prédictibles.
- Son temps d'exécution, pouvant facilement atteindre plusieurs dizaines de secondes, présente un grand intérêt à être accéléré.
- Sa grande quantité de communications mémoire permet de vérifier la bonne prise en compte des temps de transfert hôte/accélérateur dans le cadre de cette méthodologie.

Enfin, les algorithmes de flot optique présentent de manière globale un fort potentiel, en témoigne les nombreuses publications à leur sujet. La grande quantité de données, mais aussi de calculs, à traiter a donné lieu à de nombreuses études, telles que [149, 134] utilisant le GPU ou encore [135, 136], démontrant les capacités du CPU à supporter une telle charge. Une nouvelle approche émerge, à l'heure actuelle, portant sur l'utilisation conjointe, ou en remplacement, de réseaux de neurones. Nous citerons [51, 77] à titre d'exemple et de manière non exhaustive.

On retrouve, couramment, l'utilisation d'algorithmes de flot optique, notamment pour des applications de Simultaneous Localization And Mapping (SLAM), de stabilisation d'images ou encore de détection d'objets mobiles. Ils sont aujourd'hui employés, pour ce type d'applications, dans plusieurs systèmes optroniques embarqués chez Safran ou encore pour la conduite autonome de véhicules chez Tesla.

4.2.2 Algorithme de calcul de variance locale

L'algorithme de calcul de variance locale [64, 63] est une application spatiale de traitement d'images. Son principe est de calculer, pour chaque *pixel* d'une image, la variance de son voisinage. On le retrouve dans de nombreuses applications telles que la détection d'anomalies en statistiques ou encore pour l'amélioration de contrastes en traitement d'images chez Safran.

Contrairement à un algorithme de convolution, considérant classiquement des fenêtres de 3×3 ou 5×5 éléments, la taille du voisinage est ici bien plus importante, en considérant des voisinages de plusieurs centaines d'éléments. De plus, la tendance à l'augmentation quadratique des résolutions de capteurs d'images a un impact direct sur la taille de ce voisinage en traitement d'images.

4.3 Évaluation de la méthodologie sur l'algorithme de flot optique

L'algorithme de flot optique, présenté dans la section 4.2.1, est utilisé pour évaluer les résultats de la méthodologie de portage sur GPU décrite dans le chapitre 3. Dans un premier temps, nous décrivons, dans la section 4.3.1, les paramètres expérimentaux de cette évaluation. Nous abordons ensuite les résultats des analyses statiques et dynamiques dans la section 4.3.2. Nous évaluons, en section 4.3.3, les critères de placement de la section 3.4. Enfin, en section 4.3.4, nous étudions l'intérêt des transformations de code de la section 3.4.

4.3.1 Protocole expérimental

Dans le cadre de cette évaluation, les mesures sont collectées par instrumentation du code source. Afin de minimiser l'influence de cette instrumentation de code sur le temps d'exécution de l'algorithme, seules les dates des différents évènements sont enregistrées. Les temps d'exécution sont calculés à postériori. Le programme est compilé au moyen de NVCC avec un niveau d'optimisation $-O3$.

L'appel à la fonction principale de cet algorithme est donné dans le listing 4.1. Deux images **img1** et **img2**, de type OpenCV Mat, sont utilisées comme donnée d'entrée pour l'algorithme de flot optique. Celles-ci sont de type High Definition (HD) (1920×1080 pixels) et chaque pixel est codé sur trois composantes couleurs de $8\,bits$. En sortie, le résultat du flot optique est stocké dans l'objet OpenCV Mat dénommé **flow**. Pour les paramètres restants, nous avons utilisé :

- **3** niveaux de sous-échantillonage [1],
- un rayon de **2** pixels pour la fenêtre de recherche du flot optique et
- une distance maximale de flot optique de **4** pixels pour chaque échelle.

Ces paramètres correspondent à plusieurs cas d'application rencontrés pour cet algorithme. Les autres paramètres utilisés sont ceux définis par défaut par l'algorithme. Le jeu d'images en entrée ainsi que les paramètres spécifiés sont conservés à l'identique pour chacun des tests de cette évaluation.

```
calcOpticalFlowSF(img1,img2,flow,3,2,4);
```

Listing 4.1 – Paramètres d'appel de la fonction *simpleflow*

4.3.2 Analyses préliminaires

Nous effectuons, dans un premier temps, les analyses statique et dynamique permettant de construire la représentation spinale de l'algorithme et de déterminer les temps d'exécution des nids de boucles originaux et des différentes fonctions. Le but de ces analyses est d'identifier les nids de boucles naturellement plaçables sur GPU.

Analyse statique – Identification des portions de code portables sur GPU

Notre méthodologie utilise la représentation spinale décrite dans les sections 3.1.6 et 3.1.11, afin de modéliser les caractéristiques de l'algorithme *simpleFlow*. Le résultat de

1. Ce paramètre permet notamment de définir le nombre d'itérations de la boucle l_{26}

cette représentation, générée de manière automatique, est donné en annexe B. Le calcul des dépendances reste pour le moment manuel.

Les résultats mettent en évidence quatre fonctions intégrant un nid de boucles compatible avec les critères de placement sur GPU. Nous retrouvons ainsi dans la représentation spinale les nids de boucles correspondant aux différents appels de fonctions :

— *removeOcclusions* pour $l_{22,23}$, $l_{24,25}$, $l_{105,106}$ et $l_{107,108}$,
— *wd* pour $l_{2,3}$, $l_{12,13}$, $l_{55,56}$, $l_{62,63}$, $l_{73,74}$, $l_{87,88}$ et $l_{109,110}$,
— *calcIrregularityMat* pour $l_{27,28}$ et $l_{41,42}$,
— *calcConfidence* pour $l_{69,70}$ et $l_{83,84}$.

Cependant, une étude attentive de la représentation spinale montre que la fonction *calcConfidence* correspond à une zone de code mort. En effet, les régions exactes pour les tableaux *confidence* et *confidence_inv*, accédées en écriture dans les nids $l_{69,70}$ et $l_{83,84}$, sont totalement modifiées en aval par les accès en écriture des nids $l_{105,106}$ et $l_{107,108}$ de la fonction *removeOcclusions*. De plus, aucun accès en lecture pour l'ensemble des tableaux n'est effectué entre les deux entités. La fonction *calcConfidence* a donc été supprimée de l'algorithme original dans le cadre du portage sur GPU. Le résultat fonctionnel de l'algorithme reste identique après cette suppression.

Le placement initial sur GPU porte donc sur les nids de boucles des trois fonctions restantes : *removeOcclusions*, *wd* et *calcIrregularityMat*.

Modèle de représentation des résultats expérimentaux

Les figures 4.3 à 4.10 représentent les résultats exprimentaux concernant les temps d'exécution sur les plateformes Jetson TX1 (4.1.2) et Endicott (4.1.1). Ce modèle de représentation illustre pour chaque fonction et chaque nid de boucles, leurs dates respectives de démarrage et de fin, relatives à celles de l'application globale. En conséquence, la zone représentée entre ces deux dates correspond au temps d'exécution de l'élément évalué. Les temps d'exécution sont représentés :

— en **gris** pour les **fonctions**,
— en **bleu** pour les **nids de boucles**,
— en **vert** pour les ***kernels*** et
— en **orange** pour les **communications mémoire** hôte/accélérateur.

Les différents temps d'exécution ont été répartis sur six niveaux hiérarchiques, numérotés de 0 à 5. Le premier correspond au cercle le plus interne et le dernier au plus externe. La fonction principale *calcOpticalFlowSF* de l'algorithme *simpleFlow* représente la totalité du niveau 0. Son temps d'exécution correspond à la durée d'exécution globale de l'évaluation. Le temps d'exécution pour chaque fonction et chaque nid de boucles, identifiés au chapitre 3, est systématiquement placé au niveau hiérarchique supérieur dans cette représentation. Le temps d'exécution des *kernels* suit cette même règle.

Le temps évolue dans le sens opposé de celui des aiguilles d'une montre. L'expérimentation débute à la position *'3h'* et se termine à la position *'4h'*. Cette plage fixe correspond au temps d'exécution global de l'algorithme étudié. La taille figée de cette représentation permet :

— de visualiser la répartition des temps d'exécution entre les différents éléments mesurés,
— de mieux identifier les éléments les plus consommateurs en temps d'exécution,
— de favoriser la comparaison entre plusieurs représentations,
— de mieux apprécier les accélérations, pour ces éléments, suite aux transformations appliquées et
— de conserver une vue d'ensemble adaptée au format de ce manuscrit (*A4 portrait*).

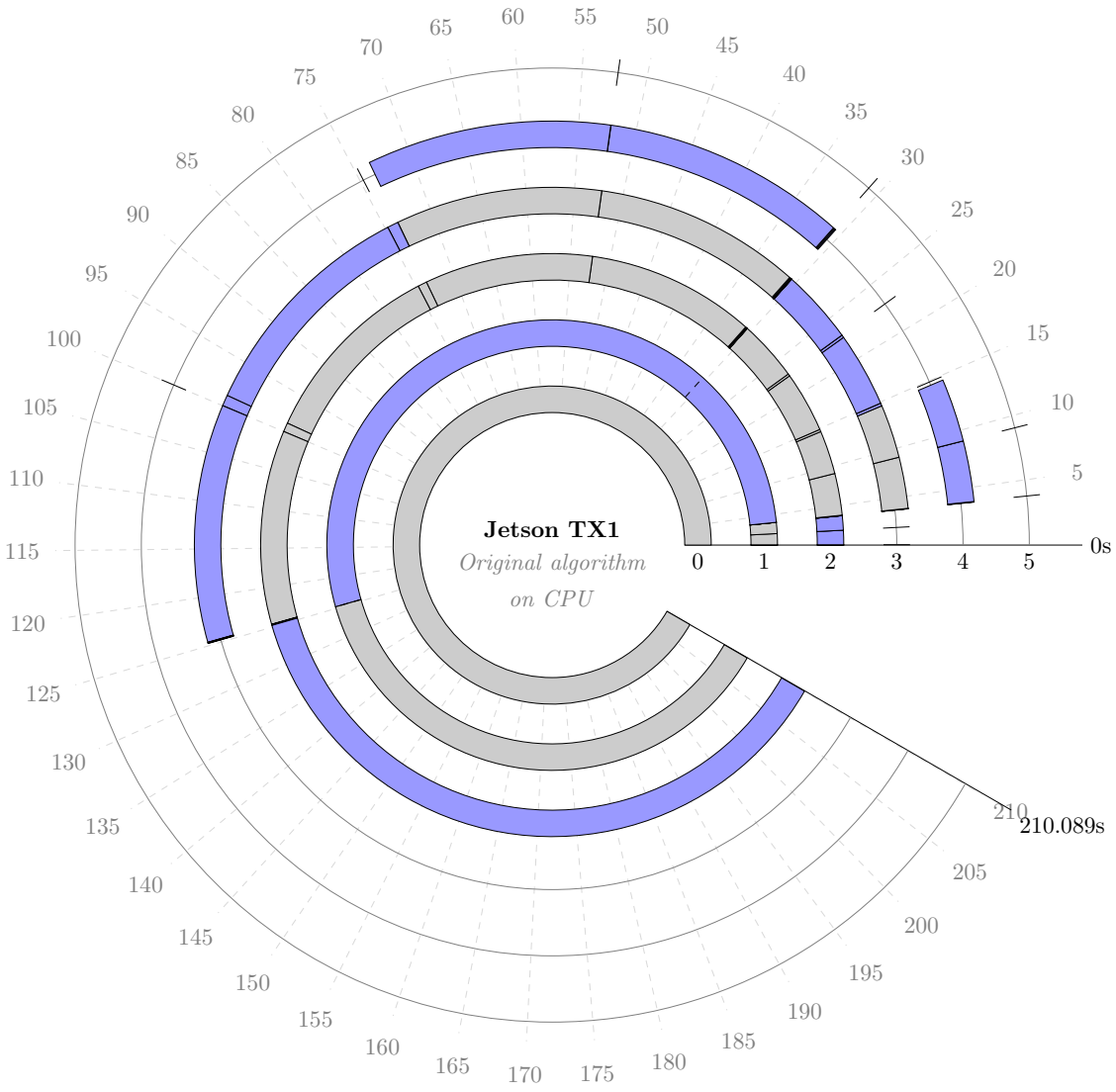


FIGURE 4.3 – Exécution de l'algorithme original *simpleFlow* sur le CPU de la Jetson TX1.

Les éléments les plus consommateurs en temps d'exécution sont plus prioritaires dans le processus de portage afin d'améliorer le temps d'exécution global.

La forme en disques de cette représentation permet d'accentuer les détails des niveaux hiérarchiques les plus élevés, là où la granularité est la plus faible et où les temps d'exécutions ont tendance à être les plus courts. La représentation hiérarchique permet de mieux appréhender les résultats selon une approche inter-procédurale. Enfin, le positionnement des différents temps d'exécution au sein de l'exécution globale de l'application permet de visualiser l'enchaînement des différentes entités exécutées sur l'hôte et sur l'accélérateur.

Pour plus de précision, l'intégralité des données utilisées pour ces représentations est détaillée dans l'annexe D. Les données concernant la plateforme Jetson TX1 sont regroupées dans la section D.1. Celles d'Endicott ont été placées dans la section D.2.

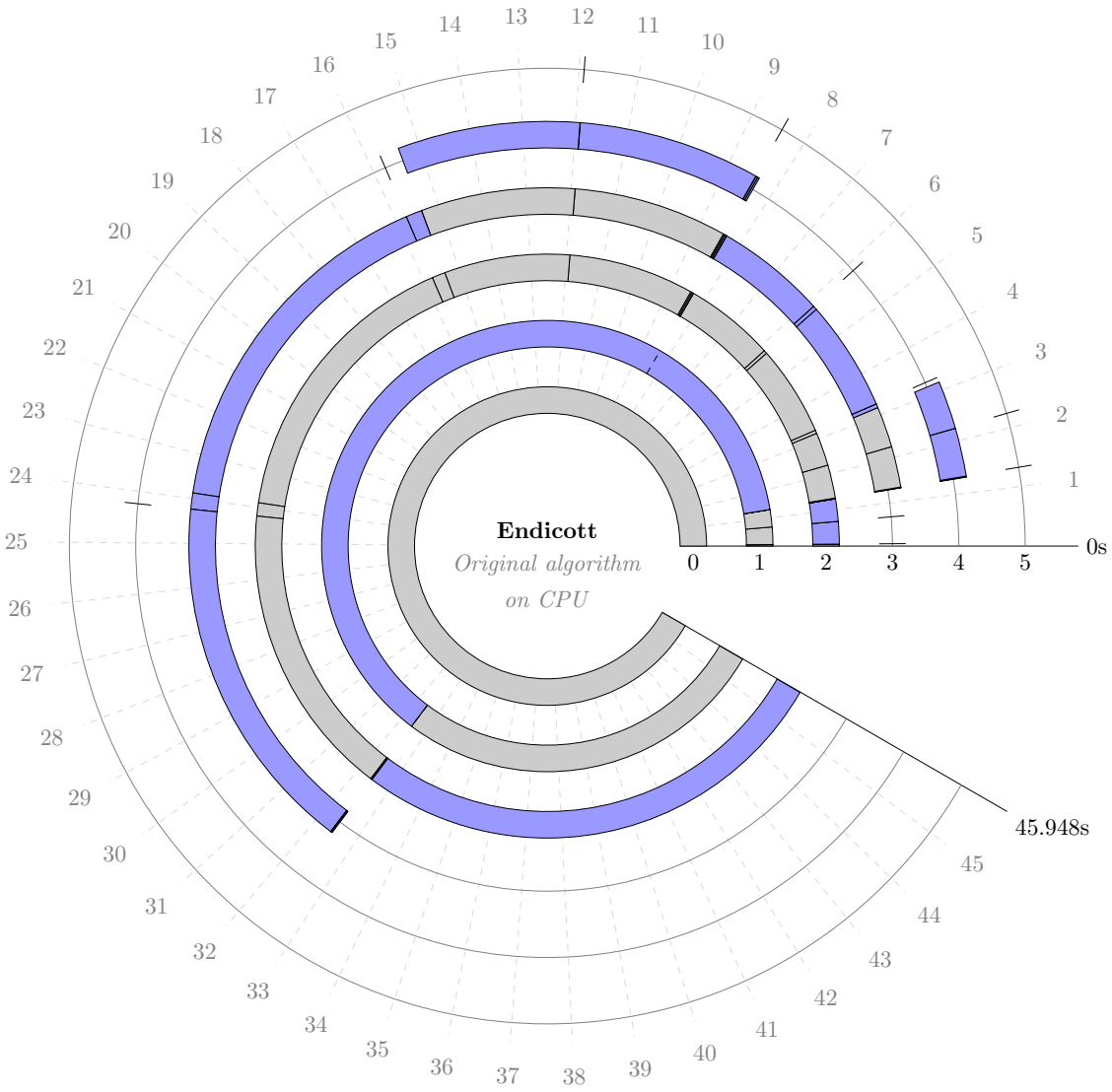


FIGURE 4.4 – Exécution de l'algorithme original *simpleFlow* sur le CPU d'Endicott.

Analyse dynamique – Interprétation de l'exécution de l'algorithme original

L'algorithme original a été écrit pour une exécution séquentielle, correspondant à l'usage d'un unique *thread* placé sur CPU. Celui-ci n'est pas optimisé pour exploiter le maximum de performances d'une architecture CPU donnée. Les temps d'exécution, présentés dans la suite de cette évaluation :

- correspondent donc à l'état original de l'algorithme sans optimisation pour la partie CPU,
- portent exclusivement sur le gain apporté par la méthodologie de portage sur GPU depuis une version non optimisée,
- ne concernent pas les méthodes d'optimisation pour CPU qui sortent du cadre de cette thèse et
- ne permettent pas de déterminer la plateforme la plus adaptée à l'algorithme *simpleflow*.

Les plateformes Endicott et Jetson TX1 présentent pour la partie CPU, une bande

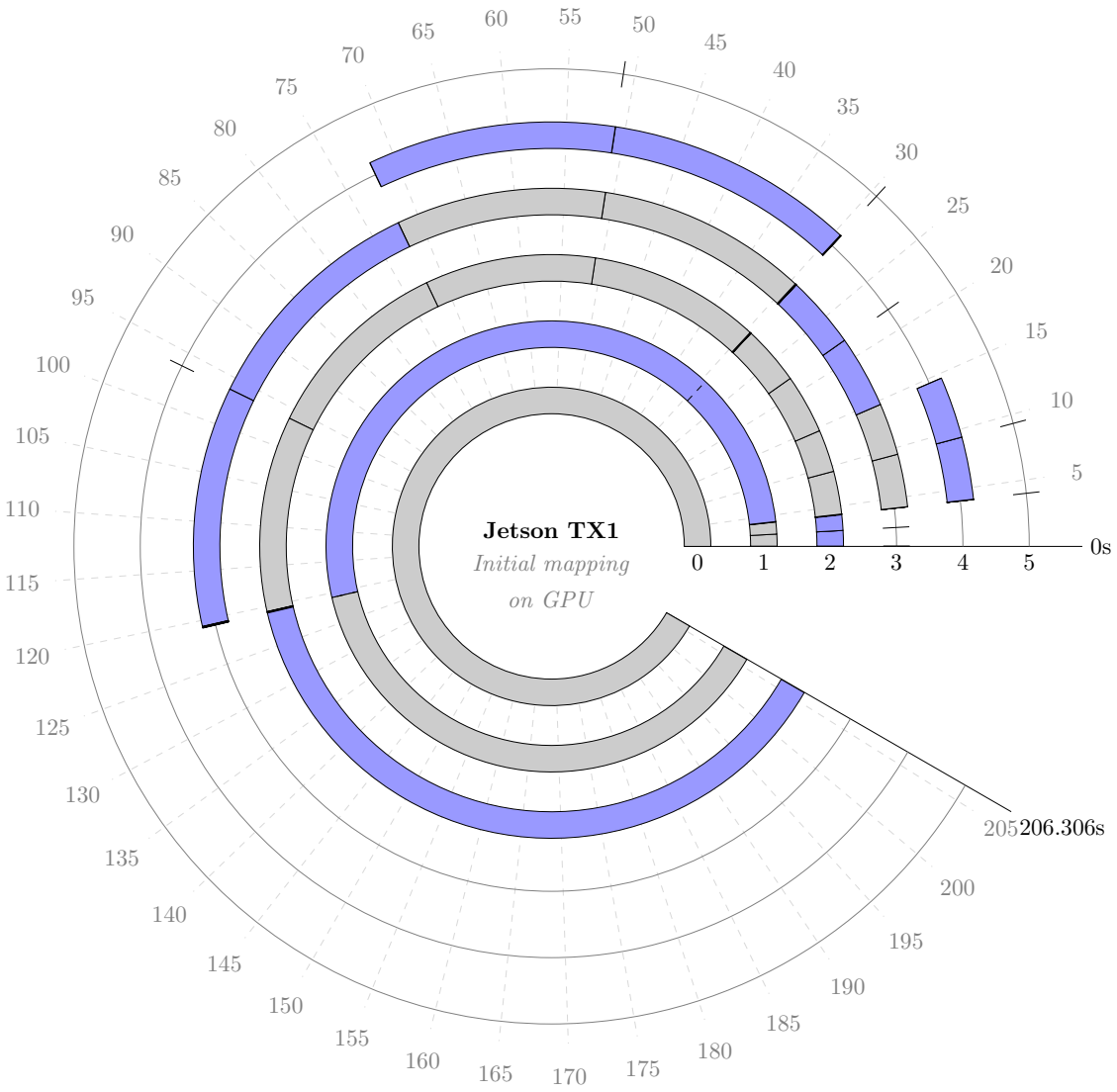


FIGURE 4.5 – Placement initial de l'algorithme Simpleflow sur le GPU de la Jetson TX1

passante mémoire identique de $25.6\,GB.s^{-1}$. Cependant la fréquence de fonctionnement du CPU d'Endicott ($3.9\,Ghz$) est supérieure à celui de la Jetson TX1 ($1.9\,Ghz$). Pour un unique cœur de calcul, le rapport des puissances de calcul simple précision est de 4.1 en faveur d'Endicott. Sans surprise, l'application est exécutée en $45.9\,s$ sur Endicott et en $210.1\,s$ sur la Jetson TX1. Le rapport est alors de 4.57, ce qui reste dans l'ordre de grandeur du rapport théorique exprimé.

Les éléments les plus consommateurs en temps d'exécution sont les fonctions *crossBilateralFilter* et *calcOpticalFlowSingleScaleSF* dont la totalité des appels représente $66,08\%$ et $32,28\%$ du temps global d'exécution sur la Jetson TX1 et 48.49% et 49.05% sur Endicott. La répartition des temps d'exécution n'est donc pas identique entre les deux architectures.

Chaque appel à la fonction *wd* engendre un temps d'exécution extrêment faible ($< 1\,ms$). Son placement sur GPU présente, de ce fait, peu d'intérêt et nous ignorons son portage.

Enfin, la fonction *calcConfidence* représente 1.36% et 1.99% du temps d'exécution global sur la Jetson TX1 et Endicott. Son abandon suppose donc un *speedup* de 1.01 et 1.02.

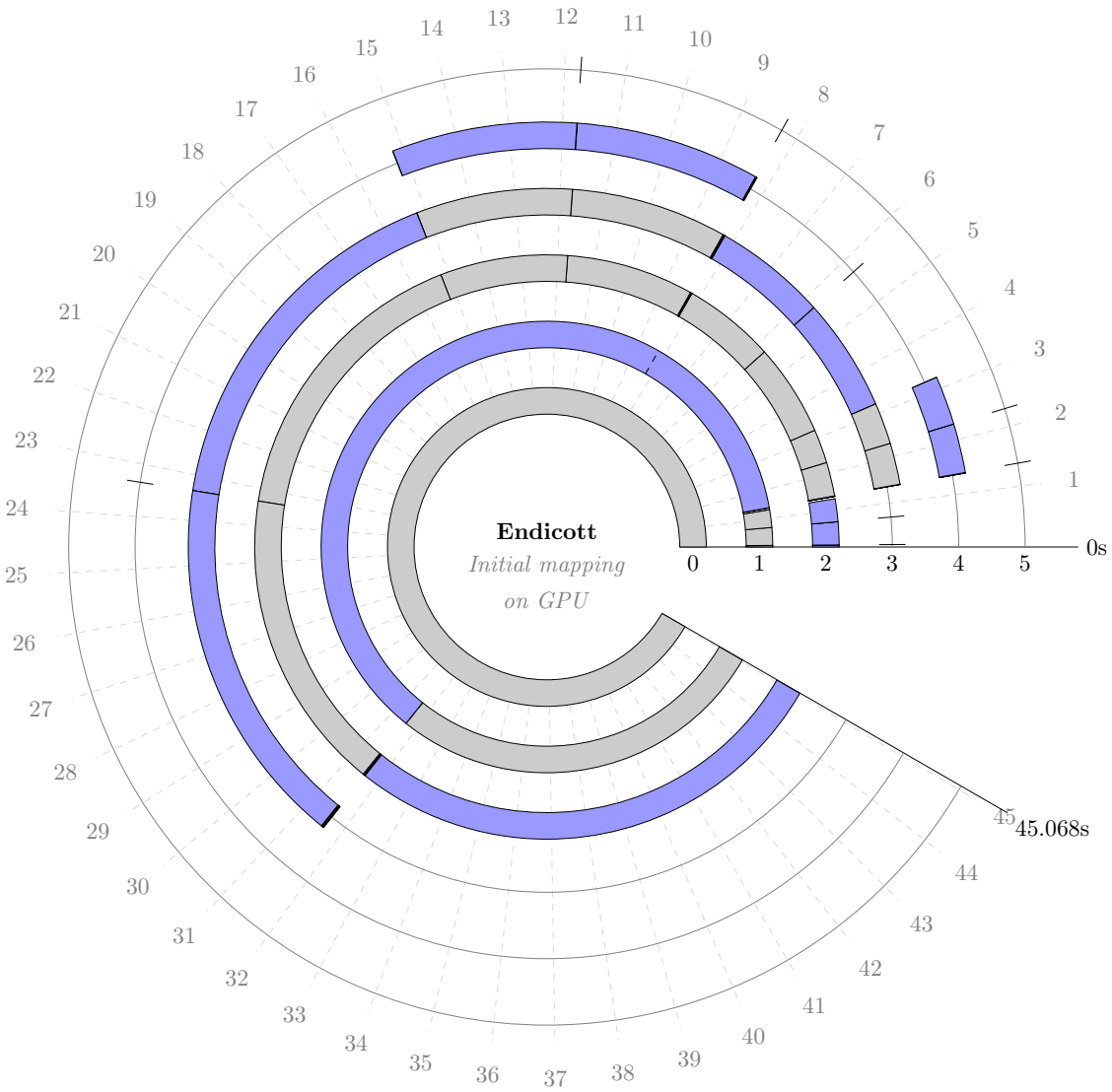


FIGURE 4.6 – Placement initial de l'algorithme Simpleflow sur le GPU d'Endicott

4.3.3 Phase de placement sur GPU

Dans cette partie de l'évaluation, nous utilisons la méthodologie de portage du chapitre 3 sur l'algorithme original. Cependant, nous n'utilisons pas le processus de transformations de code de la section 3.4 qui sera évalué dans la section 4.3.4. Seul le *tiling* de boucles est utilisé pour répartir les itérations des boucles parallèles sur les instances de *blocks* et de *threads* du GPU. Cette approche se rapproche ainsi de la vectorisation de boucles, classiquement utilisé pour le portage sur GPU [16].

Suite à l'analyse préliminaire de la section 4.3.2, seuls deux nids de boucles issus des fonctions *removeOcclusions* et *calcIrregularityMat*, sont portés sur GPU. Les *kernels removeOcclusions_kernel* (listing C.5) et *calcIrregularityMat_kernel* (listing C.1) sont alors générés ainsi que leurs communications mémoires CPU/GPU. Conformément à l'algorithme original, *removeOcclusions_kernel* est exécuté six fois dans la globalité de l'algorithme et *calcIrregularityMat_kernel* quatre fois.

Pour rappel, les fonctions *wd* et *calcConfidence* ont été écartées du processus de portage.

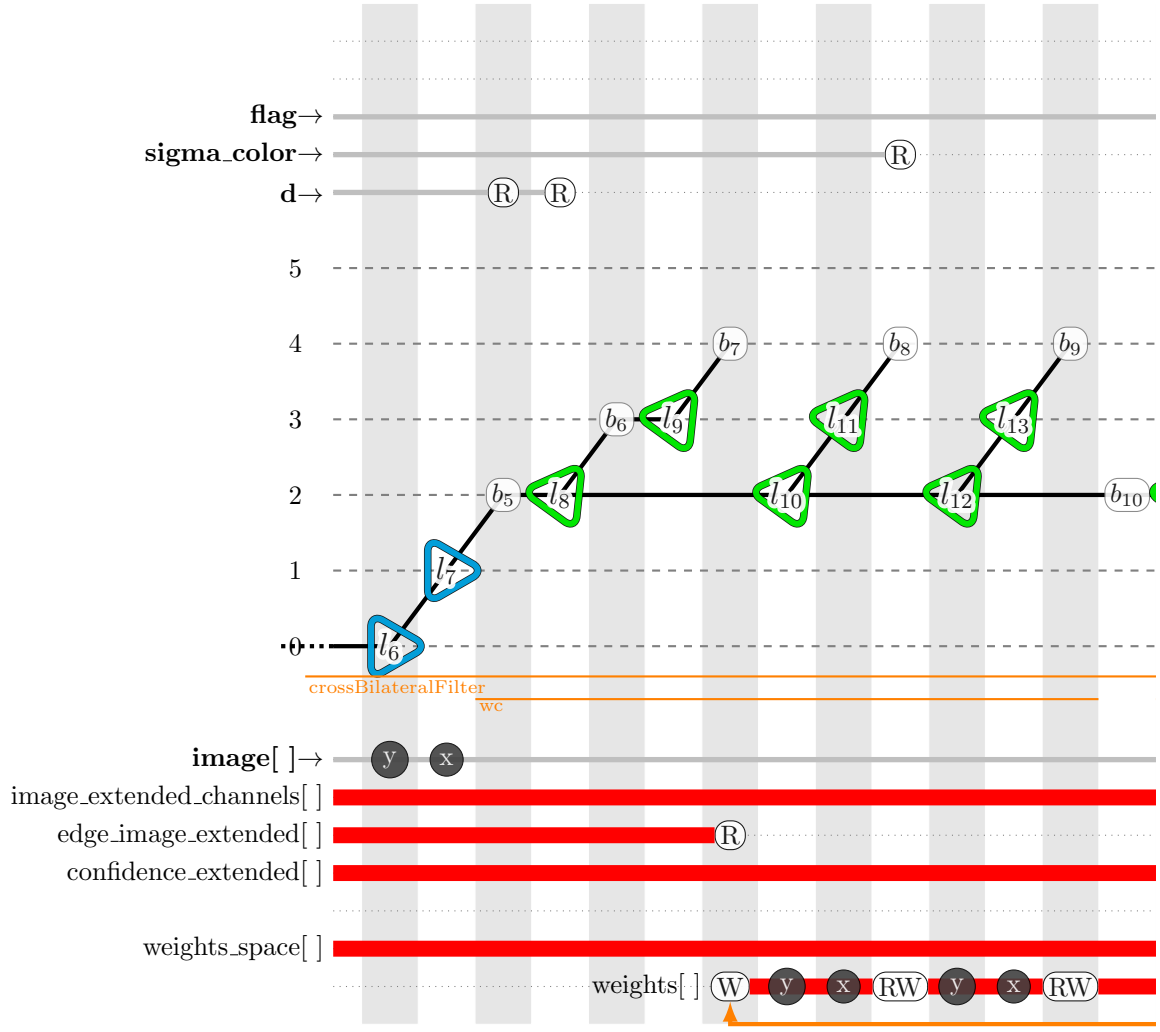


FIGURE 4.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (1/2)

Le temps d'exécution global visible dans la figure 4.5 passe ainsi de 210.1 *s* à 206.3 *s* sur la Jetson TX1. Pour Endicott, le temps d'exécution de la figure 4.6 passe de 45.9 *s* à 45.06 *s*. Le *speedup* dans les deux cas est de 1.02.

Dans la mesure où les deux fonctions concernées par le portage sur GPU représentent une part extrêment faible du temps d'exécution global, nous constatons sans surprise que le *speedup* reste proche de 1.

La fonction ***removeOcclusions*** subit globalement une dégradation de son temps d'exécution, suite à son placement sur GPU. Cette dégradation est imputable aux communications mémoire CPU/GPU. Le *speedup* moyen sur TX1 est de 0.3 et celui sur Endicott de 0.2. La pénalité globale de ce placement correspond à 0.125 *s* sur TX1 et 0.096 *s* sur Endicott

En revanche, ***calcIrregularityMat*** bénéficie globalement d'une accélération suite à son placement sur GPU. Sur TX1, le *speedup* moyen est de 3.9 et sur Endicott, celui-ci est de 4.1. Le gain de ce placement est de 0.185 *s* sur TX1 et 0.06 *s* sur Endicott

Nous remarquons tout de même que la majorité du *speedup* global, pour l'application, est lié à l'abandon de *calcConfidence*. L'ensemble des exécutions pour cette fonction représente dans le cadre de l'algorithme initial, 2.868*s* sur TX1 et 0.914*s* sur Endicott. Cela correspond respectivement à 76% et 103.8% de l'accélération du temps d'exécution global

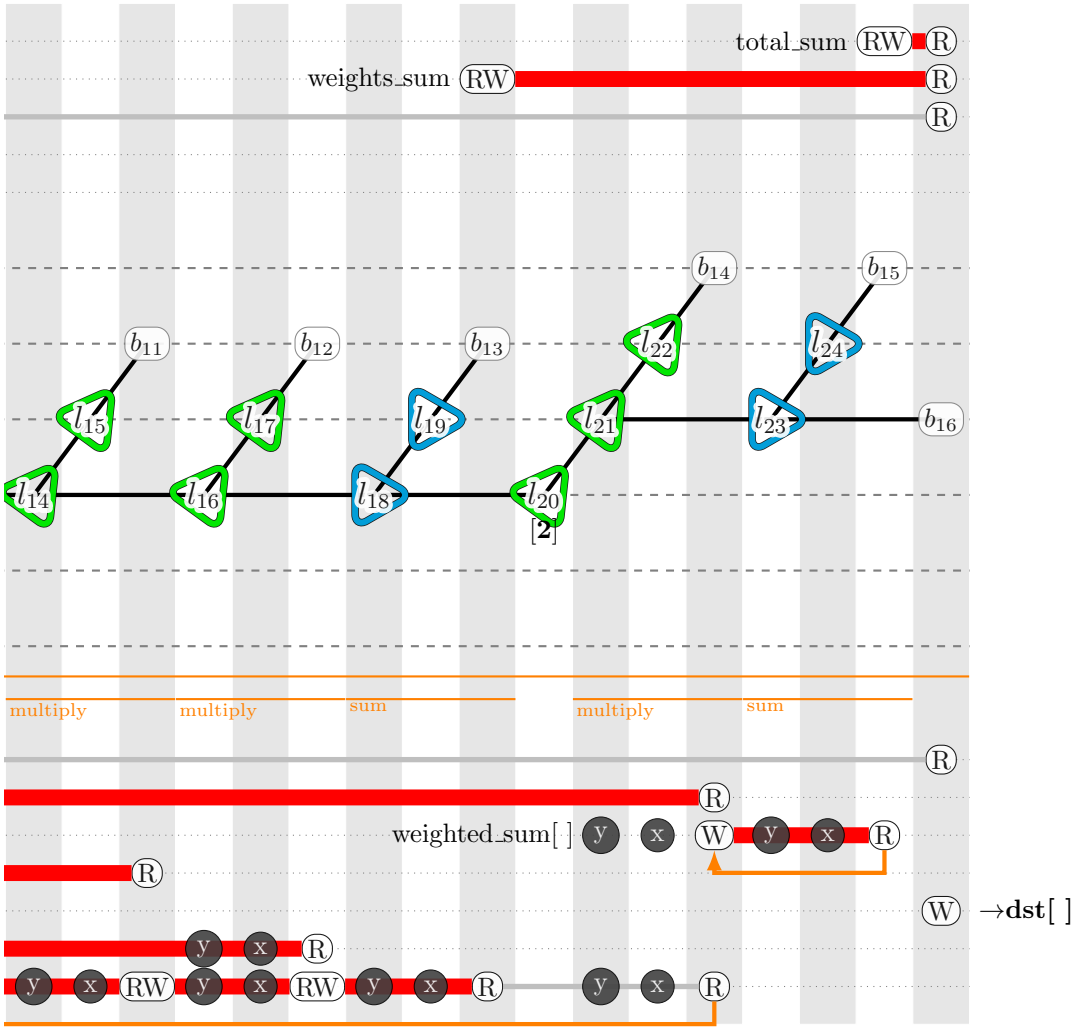


FIGURE 4.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (2/2)

obtenu, en considérant le placement pénalisant de *removeOcclusions*.

Suite à cette première phase de placement, nous concluons :

— qu'aucun nid de boucles, parmi ceux ayant les temps d'exécution les plus importants, n'est plaçable sur GPU et
— que l'utilisation exclusive de critères de placement, tels que ceux développés dans la section 3.3, ne permettent pas de considérer suffisamment de nids de boucles pour engendrer une accélération significative.

Ce constat n'est pas imputable à l'imbrication parfaite ou non des nids de boucles et aux espaces d'itération normés des nids de boucles concernés. Nos critères de placement ne considèrent pas ces paramètres. La normalisation des espaces d'itérations de la section 3.5.3 ainsi que le déplacement de blocs inter-boucles GPU de la section 3.5.2 assurent les critères du placement.

Les transformations de code de la section 3.4 sont donc nécessaires, afin d'améliorer la quantité de placement du GPU.

4.3.4 Amélioration de la quantité de code placé sur GPU

Nous appliquons à présent le processus de transformations de code introduit dans la section 3.4.

À l'issu de ce processus, plusieurs nids de boucles deviennent compatibles avec les critères de placement :

— $l_{[4,11]}$, $l_{[14,21]}$, $l_{[75,82]}$ et $l_{[89,96]}$ pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* et

— $l_{[57,61]}$, $l_{[64,72]}$ et $l_{[111,115]}$ pour la fonction *crossBilateralFilter*.

crossBilateralFilter

La fonction *crossBilateralFilter* a déjà été utilisée, pour illustrer les transformations de code, dans la section 3.4. L'extrait de sa représentation spinale est repris dans la figure 4.7. Afin de simplifier les explications, nous désignons les différents éléments, sujets aux transformations de code, dans le référentiel de cette représentation. Ainsi, l'ensemble des transformations du nid de boucles $l_{[6,24]}$ sont applicables aux nids de boucles $l_{[57,61]}$, $l_{[64,72]}$ et $l_{[111,115]}$ de la représentation spinale globale, située en annexe B.

Les boucles $l_{6,7}$ présentent une dépendance embarquée, symbolisée par une flèche générant un cycle, sur les tableaux *weights* et *weighted_sum*. Cette dépendance empêche, selon le critère 1 (section 3.3.1), le placement de ces boucles sur les instances de *block* du GPU et donc la création d'un *kernel* à ce niveau. L'analyse des dépendances indique que l'intégralité des données, pour ces deux tableaux, présentent une antidépendance [2], entre les itérations de ces deux boucles. Afin de rendre les boucles $l_{6,7}$ parallèles, il est possible d'appliquer au niveau du corps de la boucle l_7 :

— une expansion ou

— une privatisation de ces tableaux.

En s'appuyant sur les analyses statique et dynamique, nous vérifions la validité de ces solutions. Les tableaux *weights* et *weighted_sum* sont alloués de manière dynamique. Leur taille est similaire et dépend du paramètre d. Celle-ci correspond à :

$$|M_{weights}| = |M_{weighted\_sum}| = (2 \times d + 1)^2 * 4\,Bytes$$

L'analyse dynamique nous donne $d = 18$ pour cette évaluation. L'analyse statique nous permet de retrouver, dans la représentation spinale globale, le lien entre d et les paramètres de l'application :

— *upscale_averaging_radius* pour $l_{[57,61]}$ et $l_{[64,72]}$ et

— *postprocess_window* pour $l_{[111,115]}$

Ces paramètres ont pour valeur par défaut 18 mais sont modifiables par l'utilisateur.

La privatisation de *weights* implique alors l'allocation en *heap memory* de $5.35\,KB$ de données pour chacune des itérations des boucles $l_{6,7}$. La privatisation de *weighted_sum* engendre la même quantité de données allouée.

L'analyse dynamique donne pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* :

— 1080, 540 et 270 itérations pour la boucle l_6 et

— 1920, 960 et 480 itérations pour la boucle l_7.

Ces valeurs correspondent à la taille des images d'entrées (1920×1080 pixels) ainsi que leurs différents niveaux de sous-échantillonage selon un rapport de décimation par deux pour chaque dimension. Le nombre de niveaux d'échantillonage dépend du paramètre *layers*

2. symbolisée par la couleur orange de la flèche

défini lors de l'appel à la fonction principale. Celui-ci a été fixé à 3 pour cette évaluation (listing 4.1).

La quantité globale de données, allouées pour le nid de boucles $l_{[6,24]}$, augmenterait au maximum de $2 \times 5.35\,KB \times 1920 \times 1080 = 21.15\,GB$ pour cette évaluation. Le critère 3 (section 3.3.3), dans ce cas, ne serait plus respecté pour les deux plateformes utilisées. En conséquence, l'expansion de tableau ne permet pas de répondre au critère 3 de placement.

À la différence de l'expansion de tableau, la quantité de mémoire requise par la privatisation de tableau dépend du nombre de *threads* exécutés simultanément sur le GPU. Par défaut, $8\,MB$ peuvent être alloués, au maximum, en *heap memory* par *kernel*. La table 4.2 indique l'exécution possible de 2048 *threads* par SM pour les deux plateformes. Ces dernières disposent chacune de deux unités SM, ce qui représente 4096 *threads* exécutés simultanément. Cela représente $4096 \times 5.35KB = 42.8\,MB$ de données utilisées en *heap memory*. Cette valeur est supérieure à la limite [3] par défaut du GPU.

L'allocation dynamique de mémoire étant coûteuse, nous proposons, parmi l'ensemble des solutions de placement possible, une privatisation de ces tableaux à un niveau de profondeur plus élevé dans le nid de boucles. Cette solution correspond au listing C.3 dans l'annexe C. Les boucles $l_{8,9}$, $l_{10,11}$, $l_{12,13}$, $l_{14,15}$, $l_{16,17}$, $l_{18,19}$, $l_{21,22}$ et $l_{23,24}$ ont le même nombre d'itérations. Ces boucles sont parallèles à l'exception de $l_{18,19}$ et $l_{23,24}$ qui présentent un cas de réduction.

La boucle l_{20} ayant un nombre connu d'itérations (deux itérations), nous procédons à l'**unrolling** complet de celle-ci, afin de ramener $l_{21,22}$ et $l_{23,24}$ au même niveau que les autres boucles. L'analyse des dépendances nous permet d'effectuer la **fusion** de ces boucles dans les boucles $l_{8,9}$.

Nous procédons enfin à la privatisation, dans le corps de la boucle l_9, des tableaux :
- *weights* sous la forme de la variable scalaire *weight* et
- *weighted_sum* qui n'utilise plus qu'un registre temporaire lors de la réduction des données dans le scalaire *total_sum*.

Suite à ces deux privatisations :
- il n'y a plus de tableau alloué dynamiquement dans la mémoire de l'accélérateur,
- les boucles $l_{6,7}$ deviennent parallèles et
- les trois critères de placement sont satisfaits et permettent le placement de $l_{6,7}$ sur les instances de *blocks* du GPU.

Cependant, suite à la *fusion*, les boucles $l_{8,9}$ deviennent séquentielles.

Dans le cas où $d < 15$, il est possible selon le critère 2 (section 3.3.2) de placer les boucles $l_{8,9}$ sur les instances de *threads* du GPU, car :

$$\forall d \in \mathbb{N},\ (2 \times d + 1)^2 \leq 1024 \Rightarrow d \leq 15$$

Il est dans ce cas nécessaire d'employer une **réduction parallèle** ou une instruction *atomicAdd*, afin de préserver les dépendances embarquées sur les scalaires *total_sum* et *weights_sum*.

Cependant, dans le cadre de cette évaluation, nous sommes dans le cas $d > 15$. Nous avons choisi, afin d'obtenir une solution de placement plus générique, de répartir les itérations des boucles $l_{6,7}$ sur les instances de *blocks* et de *threads* du GPU, en appliquant un **tiling**.

Suite à son placement sur GPU, la fonction *crossBilateralFilter* connait une accélération moyenne de 58.47 sur TX1 (figure 4.9) et 8.77 sur Endicott (figure 4.10). Ce gain nous permet de justifier l'intérêt des transformations de code dans notre méthodologie.

3. Cette limite peut être étendue au moyen du paramètre *cudaLimitMallocHeapSize*

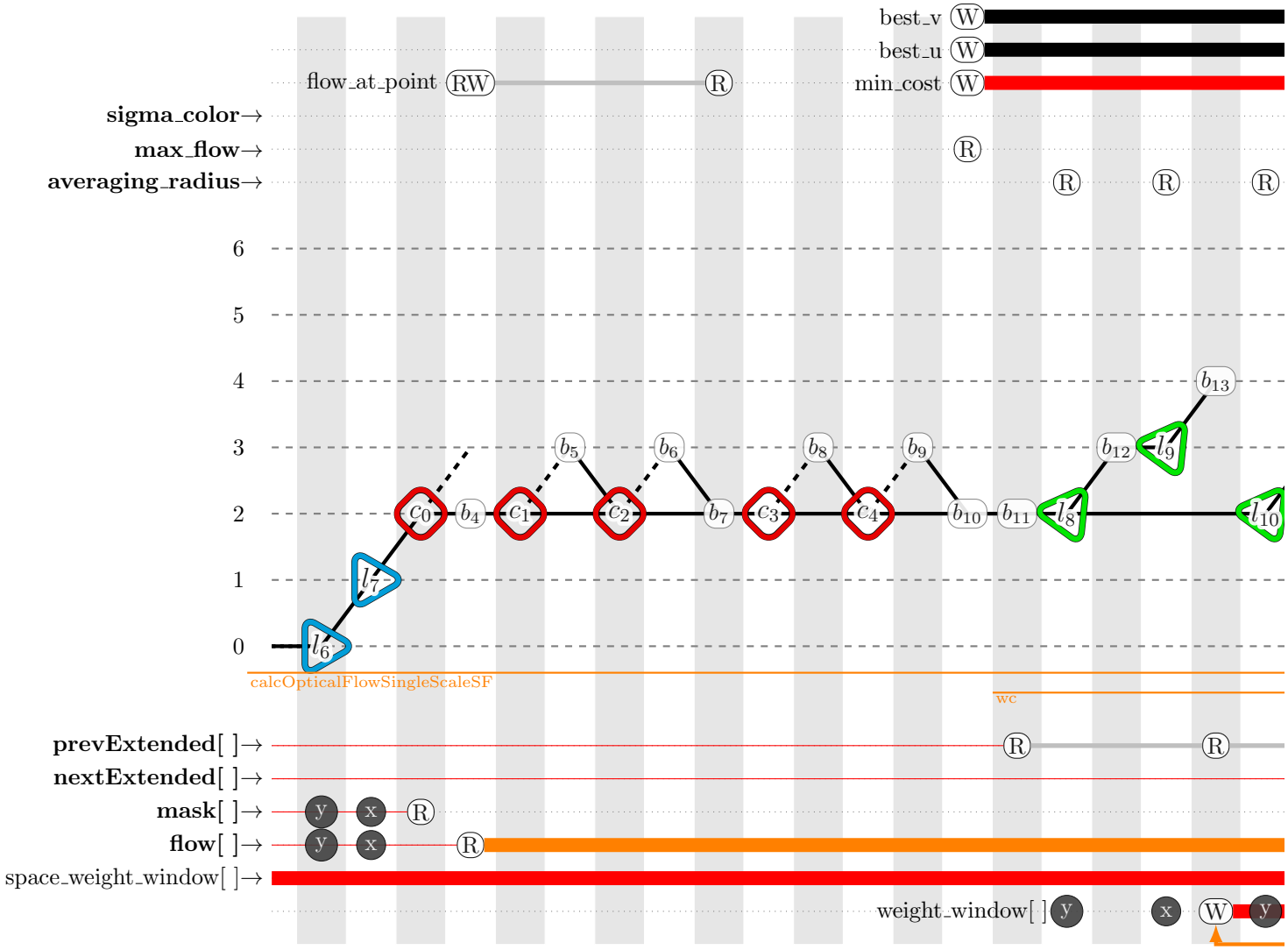


FIGURE 4.8 – Extrait de représentation spinale pour la fonction *calcOpticalFlowSinglsScaleSF* (1/2)

calcOpticalFlowSingleScaleSF

Nous étudions à présent la fonction *calcOpticalFlowSingleScaleSF*. L'extrait de sa représentation spinale est visible dans la figure 4.8. Comme pour la fonction *crossBilateralFilter*, nous nous référons aux différents éléments selon le référentiel de cette dernière. Les différentes transformations effectuées sur le nid de boucles $l_{[6,19]}$ sont transposables aux boucles $l_{4,11}$, $l_{14,21}$, $l_{75,82}$ et $l_{89,96}$ de la représentation spinale globale, située dans l'annexe B.

Nous remarquons que :

— les boucles $l_{6,7}$ présentent une antidépendance sur le tableau *weight_window*,
— les boucles $l_{16,17}$ présentent :
  — une dépendance de flot sur le scalaire *min_cost*,
  — une dépendance de sortie sur le scalaire *best_u* et
  — une dépendance de sortie sur le scalaire *best_v*,
— les boucles $l_{18,19}$ présentent une dépendance de flot sur le scalaire *cost*,
— les boucles $l_{8,9}$, $l_{10,11}$, $l_{12,13}$,$l_{14,15}$ sont parallèles.

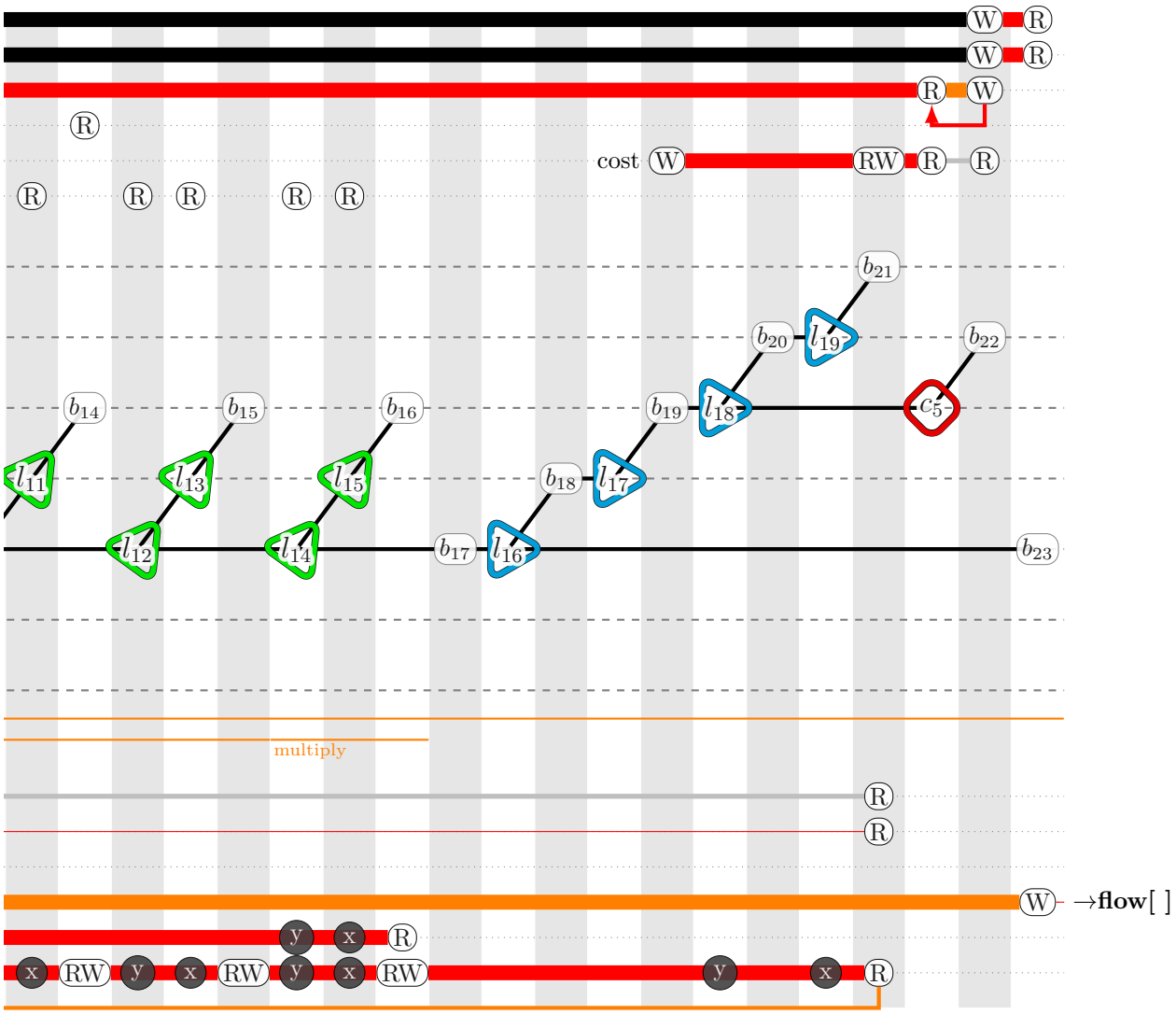


FIGURE 4.8 – Extrait de représentation spinale pour la fonction *calcOpticalFlowSinglsScaleSF* (2/2)

L'analyse dynamique donne pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* :

— 1080, 540 et 270 itérations pour la boucle l_6 et
— 1920, 960 et 480 itérations pour la boucle l_7.

Comme pour la fonction *crossBilateralFilter*, ces valeurs correspondent à la taille des images d'entrée (1920×1080 pixels) ainsi que leurs différents niveaux de sous-échantillonage. Le nombre d'itérations pour ces deux boucles dépend des données en entrée et ne peut être déduit par une simple analyse statique. Enfin, le branchement conditionnel c_0 dépendant des données du tableau *mask*, nous considérons ces domaines d'itération selon une enveloppe dense et convexe pour chaque niveau.

Les boucles $l_{18,19}$, ainsi que $l_{8,9}$, $l_{10,11}$, $l_{12,13}$,$l_{14,15}$, dépendent du paramètre *averaging_radius* de l'application. Le nombre d'itérations pour ces boucles correspond à $2 \times averaging\_radius + 1$. Ce paramètre a été affecté à 2 dans le cadre de cette expérimentation (listing 4.1), ce qui donne 5 itérations pour chacune de ces boucles.

Enfin, les boucles l_{16} et l_{17} dépendent indirectement du paramètre *max_flow*. Le

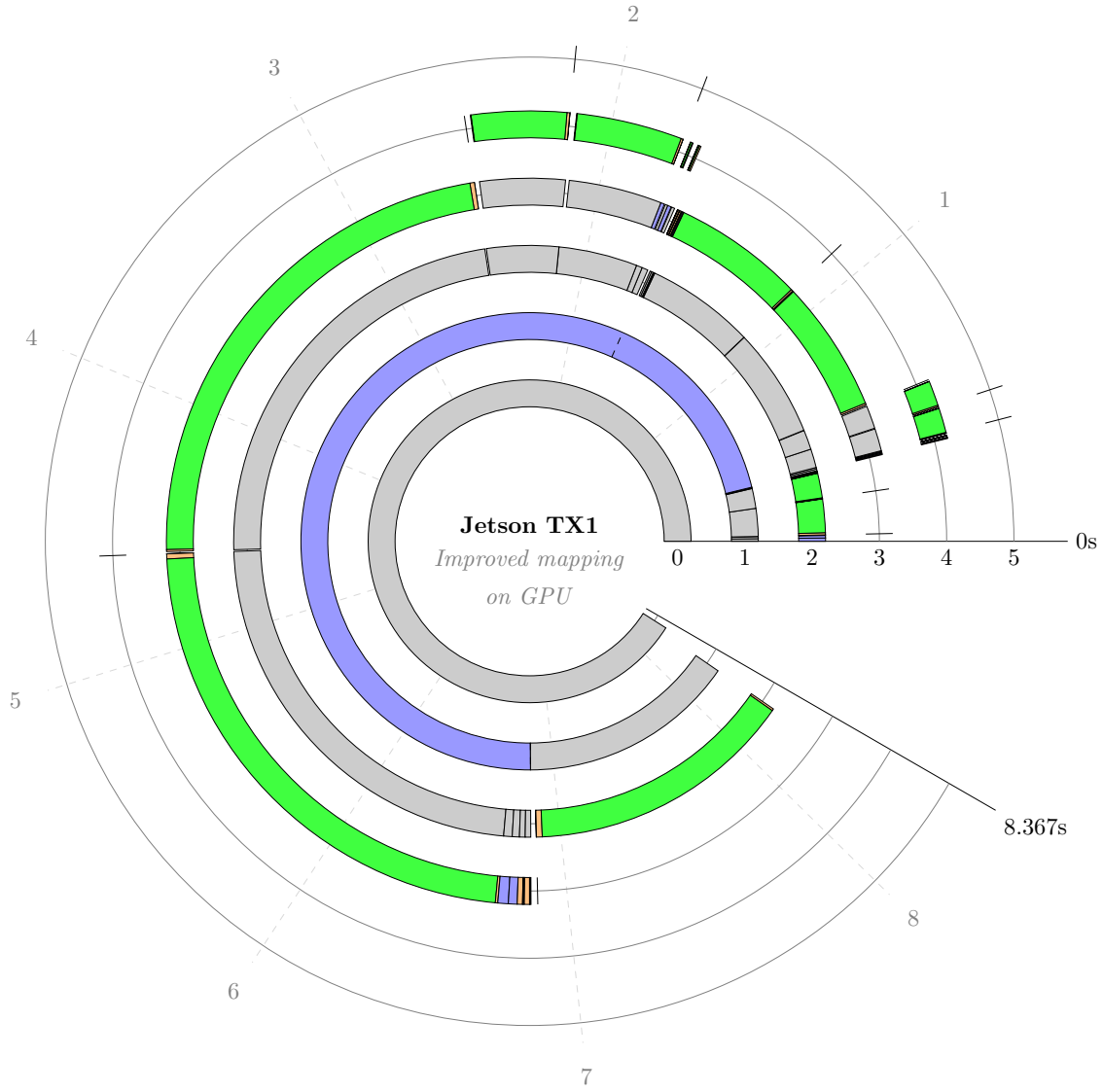


FIGURE 4.9 – Amélioration de la quantité de placement sur le GPU de la Jetson TX1

nombre d'itérations est bien plus complexe à définir. Nous utiliserons de ce fait une enveloppe convexe d'itération correspondant à $2 \times max\_flow + 1$. Ce paramètre a été défini avec une valeur de 4 dans le cadre de cette expérimentation (listing 4.1), ce qui donne 9 itérations pour chacune des deux boucles.

Les bornes pour l'ensemble des boucles de la fonction *calcOpticalFlowSingleScaleSF* sont donc dynamiques et ne peuvent être déterminées par la simple utilisation des analyses statiques.

Notre solution de placement, correspondant au listing C.2 dans l'annexe C, considère les transformations de code suivantes. La problématique est similaire à celle de la fonction *crossBilateralFilter* pour les boucles $l_{6,7}$. Le critère 1 (section 3.3.1) de placement n'est pas respecté, les boucles $l_{6,7}$ étant séquentielles. L'expansion de tableau ne permet pas de repecter le critère 3 (section 3.3.3) de placement. La privatisation, sous forme d'un scalaire, du tableau *weight_window* permet :

— de lever la dépendance embarquée pour ces deux boucles,
— de placer les itérations de ces deux boucles sur les instances de *blocks* du GPU.

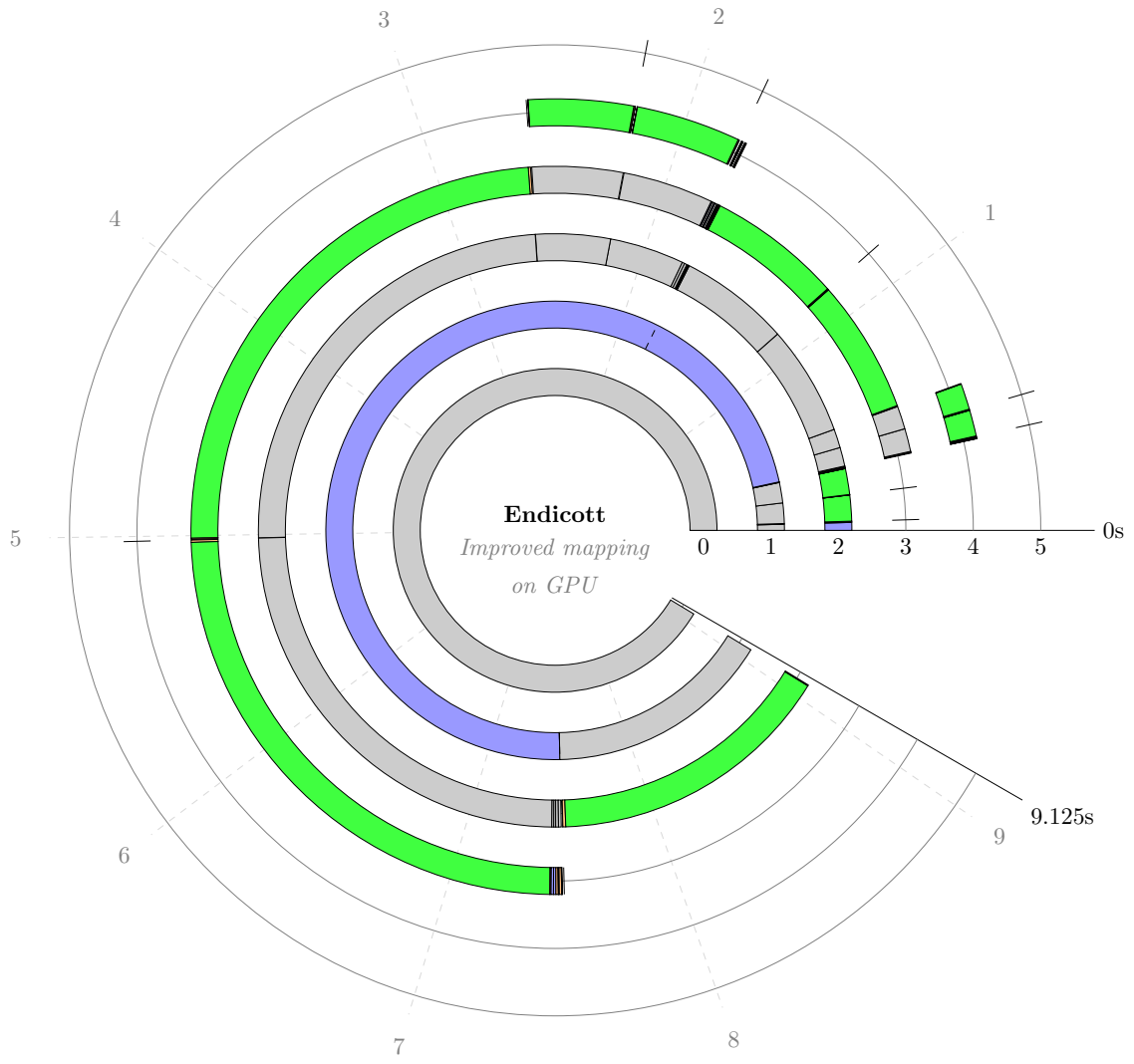


FIGURE 4.10 – Amélioration de la quantité de placement sur le GPU d'Endicott

La **fusion** des boucles $l_{10,11}$, $l_{12,13}$,$l_{14,15}$ dans la boucle $l_{8,9}$ ne pose pas de problème particulier du fait du typage parallèle de ces boucles. Cette transformation améliore au passage la localité temporelle des données. Cependant, le niveau de profondeur des boucles $l_{18,19}$ diffère de celui des boucles $l_{8,9}$. Après analyse des dépendances, nous fusionnons l'ensemble de ces boucles dans les boucles $l_{18,19}$. Cette solution a pour conséquence d'augmenter le nombre d'opérations et de communications mémoire des boucles $l_{[8,15]}$ par un facteur correspondant au nombre d'itérations des boucles $l_{16,17}$. En revanche, suite à cette transformation :

- le tableau *weight_window*, n'est plus alloué dynamiquement dans la mémoire de l'accélérateur,
- le tableau *weight_window* est privatisé sous forme la forme de la variable scalaire *weight* dans la boucle l_19,
- les boucles $l_{6,7}$ deviennent parallèles et
- les trois critères de placement sont satisfaits et permettent le placement de $l_{6,7}$ sur les instances de *blocks* du GPU.

Enfin, nous appliquons un ***tiling*** sur les boucles $l_{6,7}$. En conséquence de l'augmentation du nombre d'opérations et de communications mémoires, la pression sur les registres

augmente. La taille des tuiles est alors réduite, chaque *block* étant composé de 640 *threads*, afin que le nombre d'unités registre utilisées restent dans les limites des caractéristiques du GPU (table 4.2).

Cette solution nous permet de bénéficier d'une accélération moyenne de 12.56 sur TX1 et de 3.96 sur Endicott. Le gain est ici plus limité que celui de la fonction *crossBilateralFilter*. Cela s'explique par l'augmentation du nombre d'opérations et de communications mémoire ainsi que la taille réduite du *tiling*, réduisant le nombre de *threads* par *block*.

Conclusion sur les transformations de code

Après les transformations de code, nous obtenons un *speedup* global sensiblement plus important, avec 24.66 sur TX1 et 4.94 sur Endicott. La différence s'explique par l'écart entre les puissances de calcul des CPUs de la TX1 et d'Endicott. Le gain significatif obtenu provient de l'importance de ces deux fonctions dans le temps d'exécution global.

Point intéressant, suite aux transformations, le temps global d'exécution de l'application est similaire pour les deux plateformes. Cependant, la tendance s'inverse et la TX1 devient plus rapide ($8.367\,s$) qu'Endicott ($9.125\,s$). L'architecture basse consommation de la TX1 nous apporte, au final, les meilleures performances en temps d'exécution pour cette évaluation.

En conclusion, notre processus de transformations permet d'améliorer la quantité de code placé sur GPU. Cela se traduit par une amélioration significative du temps d'exécution global. De plus, dans le cadre de l'architecture embarquée basse consommation du TX1, notre évaluation met en évidence la nécessité du placement de code sur la partie GPU du SOC T210, afin de sensiblement réduire le temps d'exécution cette architecture.

4.3.5 Conclusion sur l'évaluation de la méthodologie

Nous avons montré, avec cette évaluation, l'importance de chacune des phases de la méthodologie dans le processus de placement sur GPU et en particulier leurs impacts sur les temps d'exécution.

La phase d'**analyse statique** nous a servi, dans la section 4.3.2 :

- à constituer la représentation spinale de l'algorithme étudié, regroupant l'ensemble des informations nécessaires au placement sur GPU et
- à identifier le code mort des opérations effectuées par la fonction *calcConfidence* dans l'algorithme *simpleflow*.

La phase d'**analyse dynamique** nous a permis, dans la section 4.3.2 :

- d'évaluer les nids de boucles les plus consommateurs en temps d'exécution et ainsi de prioriser leur portage sur GPU,
- d'identifier les fonctions, telle que *wd*, ayant un temps d'exécution trop faible ($< 1\,ms$) pour un placement sur GPU et
- de reconnaître les fonctions, telle que *removeOcclusions*, dont les performances, suite à leur portage sur GPU, sont dégradées par les échanges mémoire hôte/accélérateur.

Les **critères de placement**, développés dans la section 3.3, ont permis d'obtenir un placement fonctionnel sur GPU, dans les sections 4.3.3 et 4.3.4. Cependant, la simple identification des nids de boucles compatibles avec l'architecture GPU, n'était pas suffisante pour obtenir une accélération significative du temps d'exécution global.

Le processus de **transformations de code**, décrit dans la section 3.4, a apporté (dans la section 4.3.4) une augmentation de la quantité de nids de boucles adaptés aux critères de placement. En conséquence, le temps d'exécution global a bénéficié, grâce à ce

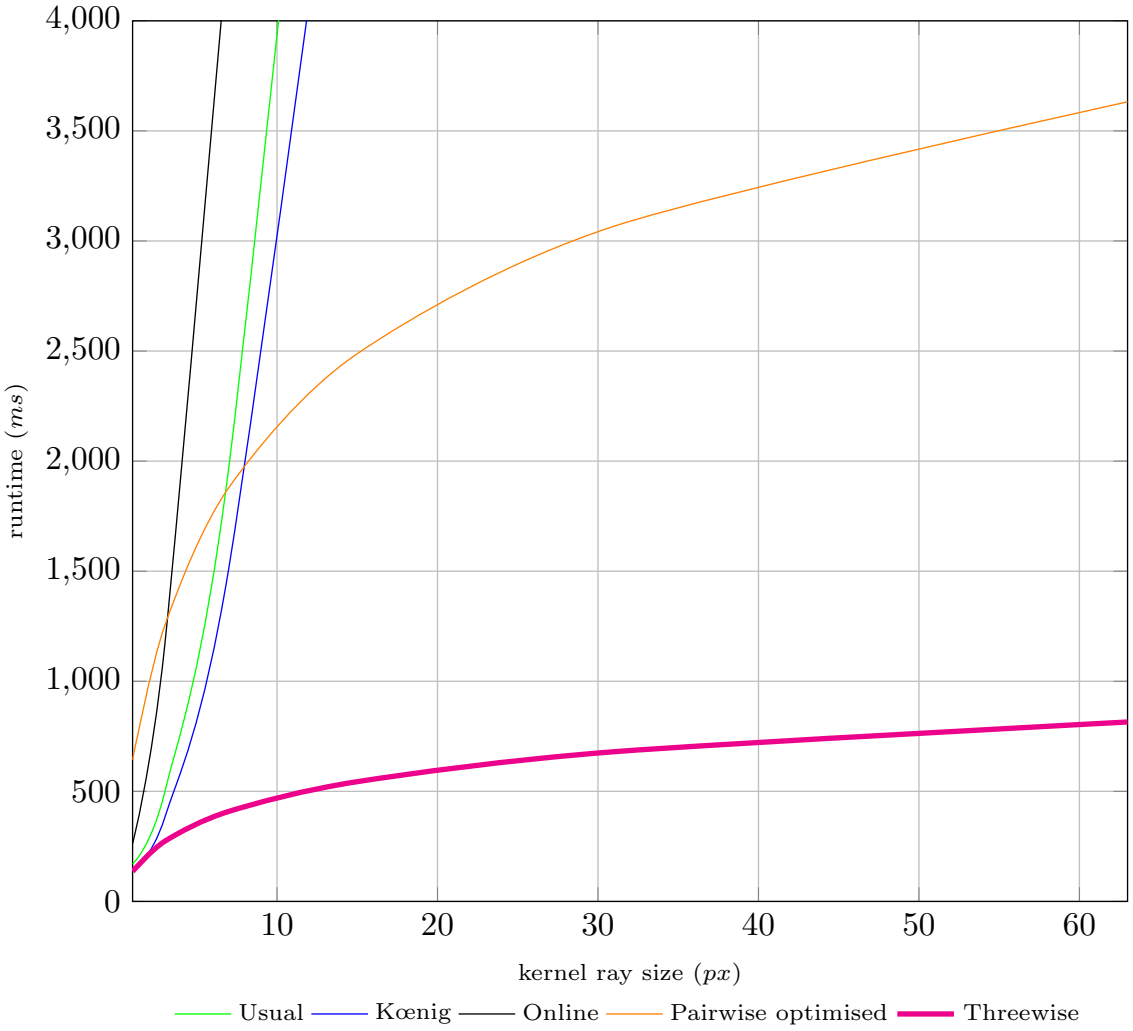


FIGURE 4.11 – Temps d'exécution de l'algorithme de variance locale en fonction de la taille du voisinage

processus, d'une accélération significative de 24.66× et 4.94× respectivement sur les deux architectures évaluées.

Enfin, nous avons constaté que le CPU ARM de la Jetson TX1 dispose d'une puissance de calcul plus limitée. Notre méthodologie de placement sur GPU a apporté, pour cette plateforme basse consommation, un gain sensible de 24.66× sur le temps d'exécution global. Nous en déduisons que le placement sur GPU est indispensable pour le SOC *T210*, afin de réduire le temps d'exécution.

4.4 Évaluation des transformations de code sur l'algorithme de variance locale

Dans le cadre de notre méthodologie, nous avons défini un ensemble de transformations (section 3.4) permettant d'améliorer la quantité et la qualité des placements sur GPU. Dans cette section, nous abordons spécifiquement le cas des réductions parallèles. Deux publications [63, 64] présentent nos travaux et nous les résumons dans cette section.

4.4.1 Description du sujet d'expérience

Nous avons étudié l'utilisation des réductions parallèles sur le cas concret du calcul de variance locale. Cette application calcule, pour chaque élément d'un ensemble, la variance de son voisinage spatial. Sa complexité dépend ainsi de la taille :

— des données utilisées en entrée,
— du voisinage considéré.

La complexité résultante est donc de type $N \times M$. Notre état de l'art rapporte quatre méthodes pour le calcul de variance : la forme usuelle, la méthode de Kœnig, l'algorithme *online* et l'algorithme *pairwise*.

En appliquant le processus de transformations de code de notre méthodologie, nous avons obtenu, par l'utilisation des réductions parallèles et de *loop interchange*, une nouvelle variante de l'algorithme *pairwise*, plus adaptée aux GPUs. En sélectionnant des paramètres de transformations adéquats, notre approche, nommée *threewise*, réduit la redondance des calculs et des communications mémoire, et réduit ainsi considérablement le temps d'exécution de l'application.

4.4.2 Protocole expérimental

La compilation pour chaque algorithme a été effectuée au moyen de NVCC pour CUDA 7.0. L'évaluation des différents algorithmes a été effectuée sur la plateforme Endicott. Nous avons utilisé, comme donnée d'entrée, une image de la National Aeronautics and Space Administration (NASA), constituée de 80 millions de *pixels* (9688×8262 *pixels*), dont la dynamique en niveau de gris est encodée sur $8\,bits$. Notre évaluation porte sur la relation entre le temps d'exécution et la variation de la taille du voisinage considéré, le paramètre M.

4.4.3 Analyse et interprétation des résultats

L'ensemble des résultats est représenté dans le figure 4.11. La forme usuelle, la méthode de Kœnig et l'algorithme *online* présentent une évolution quadratique de leurs temps d'exécution respectifs. Ce résultat est en cohérence avec la complexité de type $N \times M$ de l'algorithme.

La transformation de l'algorithme *pairwise* permet de réduire la complexité de l'algorithme à une forme de type $N \log M$. Son utilisation apporte un intérêt à partir d'un voisinage de plus de 8 *pixels*. L'algorithme *threewise* conserve la même forme de complexité. Cependant, son temps d'exécution est optimal pour l'ensemble des tailles de voisinnage considérées.

4.4.4 Conclusion

Nous avons montré dans cette expérimentation, que les transformations de codes, introduites dans la section 3.4, permettent d'améliorer la qualité du placement sur GPU. L'utilisation des réductions parallèles et du *loop interchange* nous a permis, dans ce cas précis, de réduire la complexité de type $N \times M$ d'un algorithme en $N \log M$ et de réduire la quantité de calculs et de communications mémoires redondantes. En conséquence, le temps d'exécution de l'algorithme est minimal par rapport aux autres algorithmes.

4.5 Conclusion

Nous avons évalué, dans ce chapitre, notre méthodologie de placement d'algorithmes sur GPU avec deux applications de traitement d'images.

Nous avons employé deux architectures Nvidia dédiées à des domaines d'application distincts. Nous considérons ainsi dans cette évaluation les GPUs basse consommation propres au domaine de l'embarqué et les GPUs "classiques" intégrés dans les plateformes de type *workstation*.

La première application, le *simpleflow*, présente une complexité spatiale de code intéressante. Ses nombreux appels de fonctions nous ont permis d'évaluer l'aspect interprocédurale de notre méthodologie. De plus cette application comporte de nombreux nids de boucles dont la plupart possèdent des bornes variables. Enfin, l'utilisation de la librairie OpenCV, dans le code original, nous a permis de tester notre méthodologie avec le langage C++ et plus particulièrement le paradigme de programmation orienté objet.

Concernant notre méthodologie, la représentation spinale de cet algorithme a pu être générée à partir de l'analyse de code statique. L'analyse de code dynamique a permis de déceler la dégradation du temps d'exécution d'un nid de boucles lors de son placement sur GPU. L'aspect fonctionnel de l'algorithme original et la légalité du placement ont été préservés grâce à nos trois critères de placement développés dans les sections 3.3.1, 3.3.2 et 3.3.3. Cependant, la stricte application de ces critères n'a pas permis d'améliorer significativement le temps d'exécution global de l'algorithme. Pour cela, les transformations de code sont essentielles pour améliorer le nombre de *kernels* plaçables sur GPU. Le temps d'exécution global de l'algorithme a ainsi été sensiblement réduit avec un *speed-up* de 24.66× pour le GPU basse consommation et de 4.94× pour le GPU de notre *workstation*. Sur Jetson TX1, nous observons que le placement sur la partie GPU du SOC est une condition nécessaire à l'obtention de temps d'exécution réduits, le processeur hôte ayant des performances calculatoires plus limitées.

La seconde application, le calcul de *variance locale*, est une application beaucoup plus courte et monofonction. Cependant, sa complexité quadratique impacte fortement son temps d'exécution en traitement d'images, car une image HD contient approximativement deux millions de pixels à traiter.

Les transformations de code de notre méthodologie nous ont permis de définir une nouvelle forme algorithmique (l'algorithme *threewise*). Celle-ci remplace la complexité initiale avec une nouvelle complexité en $N \log N$, engendrant ainsi un temps d'exécution inférieur pour l'ensemble des cas. Par cette expérimentation, nous avons ainsi mis en évidence la capacité de notre ensemble de transformations de code à améliorer de manière qualitative le placement sur GPU.

Au final, notre méthodologie de placement nous a permis de porter avec succès, sur GPU, une application C++ de traitement d'image, le *simpleflow*, en s'appuyant sur notre représentation spinale. Les phases d'analyses statiques ont montrés leur capacité à s'adapter aux contraintes du langage C++ et de la librairie OpenCV, tandis que les analyses dynamiques ont assuré une amélioration des performances en fonction de l'architecture utilisée. Enfin les résultats obtenus pour le *simpleflow* et l'algorithme *threewise* soulignent l'importance de la phase de transformation de code dans notre processus de portage.

Afin d'améliorer encore les performances des *kernels*, il est nécessaire de les spécialiser davantage pour les architectures ciblées. Nous abordons ce sujet avec les optimisations et les spécialisations de code du chapitre 5.

Chapitre 5

Étude des temps d'accès aux diverses mémoires et de l'exploitation du parallélisme gros grain sur GPU Nvidia

Sommaire

Dans le chapitre 3, nous avons défini une méthodologie de portage d'algorithmes sur GPU. Celle-ci considère les domaines d'itérations des nids de boucles pour les adapter aux domaines d'instances des GPUs. Cette méthodologie, validée au cours du chapitre 4, est décomposée en plusieurs étapes :

— les analyses statiques et dynamiques du code source (sections 3.1 et 3.2),
— la vérification des trois critères de placement sur GPU (section 3.3),
— l'application de différentes transformations de code pour améliorer quantitativement et qualitativement le placement sur GPU(section 3.4),
— la génération de code pour GPU (sections 3.5 et 3.6),
— la validation des *kernels* au moyen des analyses dynamiques (section 3.7).

Au final, ces différentes procédures suffisent pour générer un placement valide sur GPU à partir d'un algorithme séquentiel. Certains points d'optimisation ont de plus été introduits tels que :

— pour le critère 1 de placement (section 3.3.1) :

  - la minimisation des dépendances à l'intérieur d'un *block* et
  - la maximisation du nombre de dimensions d'instances de *threads*
- pour le critère 2 de placement (section 3.3.2) :
  - la saturation des unités SM en utilisant une quantité minimale d'instances de *thread* par *block* et
  - la saturation des *warps* en utilisant un multiple de 32 *threads* pour la taille des *blocks*
- la linéarisation des accès mémoire (section 3.5.4) lors de la préparation avant génération de code,
- l'élimination des allocations et libérations d'instances mémoire redondantes sur l'accélérateur (section 3.6.2)
- et pour la génération des communications mémoire hôte/accélérateur (section 3.6.3) :
  - l'élimination des communications redondantes et
  - la réduction des transferts aux espaces de données modifiés.

Ces diverses optimisations contribuent à l'identification d'une solution de placement qualitative, réduisant le temps d'exécution de l'algorithme résultant. Mais l'optimisation de code est un sujet très vaste. Le flot de calculs, le flot d'instructions et le flot de données, abordés dans la section 1.3, sont respectivement exploités, au sein des GPUs, par des ressources dédiées telles que :

- les *CUDA cores*,
- les *instruction dispatch units* et
- les *load/store units*.

Leur saturation a alors une influence directe sur le temps d'exécution d'un *kernel*. On retrouve notamment, parmis les nombreuses sources possibles de sous-exploitation de ces ressources :

- le nombre élevé de cycles nécessaire pour l'exécution de certaines opérations,
- le phénomène de "bulles" dans les *pipelines* d'instructions ou encore
- le temps de latence et la bande passante des différentes mémoires.

Nous nous intéressons, dans notre méthodologie, à la minimisation du temps d'exécution des algorithmes placés sur GPU. Nous abordons ainsi, dans ce chapitre, deux ensembles de mesures expérimentales, préalables à de futures optimisations fines de code pour CUDA, dans le cadre de notre méthodologie, qui améliorent le placement sur les GPUs Nvidia.

Cette thèse s'intéressant en particulier aux applications de traitement d'image, nous retrouvons dans ce domaine trois facteurs dont la prise en compte présente une source potentielle d'accélération du temps d'exécution :

1. En raison du volume important de données à traiter, la bande passante mémoire constitue souvent le facteur limitant sur GPU.

2. Il arrive aussi, dans certains traitements complexes, que ce volume de données soit fragmenté en plusieurs sous-ensembles, traités dans des *kernels* distincts. La quantité de données à traiter peut alors devenir insuffisante pour exploiter les capacités du GPU.

3. Enfin, l'utilisation de sources de données multiples [1], engendre des *pipelines* d'applications de traitement d'image portant sur des ensembles distincts de données. Ces *pipelines* sont ainsi compatibles avec une exécution concurrente, lorsque les portions du programme ne présentant pas de dépendance.

1. tel que l'ensemble des capteurs du véhicule *model S* de Tesla cité dans le chapitre 1

Nous évaluons ainsi, dans la section 5.1, plusieurs performances des différents espaces mémoire sur GPU pour le premier facteur. Pour les deux derniers, nous étudions, dans la section 5.2, le comportement du GPU lors d'un placement exploitant un parallélisme à gros grain (*coarse grained parallelism*).

Les expérimentations décrites dans ce chapitre sont préliminaires à l'intégration de ces deux sujets dans notre méthodologie.

5.1 Étude des espaces mémoire sur GPU

Notre méthodologie de placement sur GPU (chapitre 3) spécifie, lors du processus de génération de code (section 3.6), le placement des données sur la *global memory*. Dans cette section, nous étendons les capacités de notre méthodologie à l'usage des autres espaces mémoires mis à disposition par CUDA.

Concrètement, trois catégories de mémoire physique sont présentes sur toutes les architectures GPU :

***Device memory* :** Physiquement située à l'extérieur du SOC, la *device memory* est partagée par l'ensemble des unités SM. Elle est accessible par l'ensemble des *blocks* de *threads* et est utilisée pour l'échange de données avec le processeur hôte.

***Shared memory* :** Présente dans chaque unité SM, la *shared memory* est exclusivement accessible aux CUDA cores de l'unité SM concernée. Elle est utilisable par l'ensemble des *threads* de chaque *block* et permet l'échange de données entre ces *threads*.

Unités registres : Comme pour la *shared memory*, les unités registres sont présentes dans chaque unité SM et sont dédiées aux CUDA cores. Chaque allocation d'une unité registre est dédiée à un *thread* et l'échange de données n'est de ce fait pas possible.

Ces trois mémoires physique sont exploitées, au travers du langage de programmation CUDA, au moyen de sept espaces mémoires. La différence entre ces espaces provient essentiellement des différentes méthodes d'accès aux données et plus particuliérement des mémoires caches dédiées. En reprenant la nomenclature issue de la documentation officielle de CUDA [123], ces espaces correspondent à :

— la *Global memory*,
— la *Local memory*,
— la *Constant memory*,
— la *Texture memory*,
— la *Surface memory*,
— la *Shared memory* et
— les registres.

Nous faisons ainsi la distinction entre les unités de mémoires physiquement présentes sur le GPU des espaces mémoires logiques fournis par CUDA. En complément, CUDA appelle "*Host memory*", l'espace mémoire du processeur hôte mis à disposition pour le contexte CUDA.

L'ensemble des solutions présentées dans l'état de l'art (chapitre 2) propose le placement des données en *global memory*. La *local memory* et les unités registres sont utilisées de manière implicite par le compilateur NVCC.

Nous considérons alors les solutions permettant d'identifier le placement de données sur les autres espaces mémoire. Parmi ces solutions, le placement de données en *shared memory* est géré de manière automatique par :

— PPCG [160, 159] (section 2.2.3),
— *C-to-CUDA* [23] (section 2.2.1),
— R-Stream [103] (section 2.2.4) et
— *Cuda-Lite* [153] (section 2.4.1).

Pour la *constant memory*, seul *C-to-CUDA* semble l'exploiter. Le placement est réalisé par ce dernier, lorsque les accès sur un tableau sont en lecture seule et sont communs pour l'ensemble des *threads* de chaque *block*.

Enfin, nous ne connaissons aucun compilateur source-à-source exploitant la *texture memory* ou la *surface memory*.

De manière générale, les informations sur les modèles de placement mémoire utilisés par les compilateurs sont assez rares et peu détaillées. Nous ne connaissons, de plus, aucune publication portant sur la détermination automatisé de l'espace mémoire le plus adapté à chaque espace de données d'un algorithme donné.

Ce constat peut s'expliquer par les modifications architecturales, spécifiques aux mémoires, apportées à chaque nouvelle génération de GPU chez Nvidia [132, 131, 130, 129, 120]. Les optimisations dans ce domaine sont donc spécifiques à chaque génération. Ces modifications permanentes soulignent le besoin actuel d'améliorer les bandes passantes d'accès aux données, abordé en introduction du chapitre 1.

Nous nous intéressons ainsi à la problématique portant sur l'identification de l'espace mémoire minimisant le temps d'accès aux données dans un algorithme. Pour cela, nous synthétisons, dans la section 5.1.1, les caractéristiques de chacun de ces espaces mémoire. Nous établissons ensuite l'objectif de cette expérimentation dans la section 5.1.2 et détaillons notre protocole de test dans la section 5.1.3. Enfin, dans la section 5.1.4, nous analysons les performances des différents espaces mémoire afin d'établir des critères de sélection.

Cette étude est réalisée avec le GPU Quadro K2000 d'Endicott et le SOC T210 de la Jetson TX1. Ces deux plateformes sont décrites dans la section 4.1.

5.1.1 Descriptions des espaces mémoire CUDA

Nous présentons, dans cette section, les spécificités des différents espaces mémoires disponibles dans CUDA. L'ensemble des informations utilisées ont été extraites des documentations officielles CUDA [122, 123].

Global memory

La *global memory* est l'espace d'échange principal entre le processeur hôte et le GPU. Il prend place sur la majeure partie de la *device memory*.

Son utilisation a déjà été décrite dans la section 3.6.2. Les espaces de données sont alloués au moyen de l'instuction *cudaMalloc*, qui garantit un alignement des données sur 256 $Bytes$. Leur libération se fait au moyen de la fonction *cudaFree*. Les données non libérées ont une persistance correspondant à la durée de vie du contexte CUDA utilisé.

Les communications de données entre l'hôte et l'accélérateur ont été abordées dans la section 3.6.3.

Depuis la génération *Fermi* (*compute capability* $\geq 2.x$), les communications de données avec la *global memory* passent par un processus de cache. Pour la génération *Fermi*, il s'agit d'un cache à double niveaux (L1 + L2). Cependant, depuis la génération *Kepler* (*compute capability* $\geq 3.x$), les données ne transitent plus que par le cache L2. Les lignes de cache L1 font 128 $Bytes$ tandis que celles du cache L2 font 32 $Bytes$.

L'utilisation de mémoire cache apporte une accélération du temps d'accès aux données de la *global memory* en cas de *cache hit*. L'alignement des données et l'augmentation du *stride* lors des accès ont donc un impact direct sur le temps d'accès global aux données.

Local memory

La *local memory* n'est pas utilisable de manière explicite au moyen de CUDA. Dans le processus de compilation de NVCC, son utilisation ne peut se faire qu'au moyen des instructions *ld.local* et *st.local* de l'ISA PTX de Nvidia [127]. Chaque *thread* dispose d'un espace privé de $512\,KB$, résidant dans la *device memory*.

De manière générale, la *local memory* est utilisée par le compilateur NVCC, pour les tableaux déclarés de manière statique dans le *kernel*. Elle permet de limiter la surcharge des unités registres, en appliquant le principe du *register spilling*. Dans le cadre de notre méthodologie, la privatisation de tableaux dans un *kernel*, invoquée dans les transformations de code de la section 3.4, a un impact sur l'utilisation de la *local memory*.

Les communications de données transitent par les mémoires cache L1 et L2, décrites pour la *global memory*. L'utilisation de ces deux niveaux de cache permet de limiter la pénalité liée à la bande passante plus faible de la *device memory*, en comparaison à celle des unités registre.

L'emploi de la *local memory*, associée à la privatisation de tableaux, sera cependant évitée autant que possible. Nous lui préférons au contraire la privatisation de scalaire, qui maintient les données dans les unités registre. Ce sujet a été abordé dans la section 4.3.4.

Texture/Surface memory

La *texture memory* est un espace mémoire accessible en lecture seule. Concrètement, elle exploite, comme la *global memory*, une partie de la *device memory*, externe au SOC. Elle se différencie cependant par un canal d'accès distinct, composé d'un cache dédié (le *texture cache*) et d'unités de calcul spécialisées (les *texture units*).

Le **texture cache**, en association avec l'exploitation des *CUDA arrays*, conçoit la localité spatiale des données selon deux dimensions. Il est ainsi optimisé pour les accès bi-dimensionnels en accélérant, dans ce cas précis, le temps d'accès aux données, comparativement aux classiques méthodes d'accès mono-dimensionnelles. De plus, la *texture memory* étant accessible en lecture seule, l'absence de vérification pour la cohérence des données, entre les différents modules de *texture cache*, se traduit par un gain supplémentaire sur le temps d'accès aux données placées en cache.

Enfin, les **texture units** sont des unités de calculs spécialisées, dont le rôle est d'effectuer :

— l'interpolation spatiale de données selon une loi linéaire et
— la réplication dynamique de données lors d'accès en dehors des bornes de définition des dimensions de l'espace mémoire alloué.

Ces unités permettent ainsi de réduire la quantité de calculs effectués par les *CUDA cores* en formattant de manière dynamique les données acheminées. En pratique, cet espace mémoire remplace judicieusement les fonctions OpenCV :

— *copyMakeBorder* avec la réplication dynamique des données et
— *resize* avec l'interpolation spatiale des données.

De nombreux exemples d'utilisation de ces fonctions sont visibles dans l'algorithme *simpleFlow* de l'annexe A.

La *texture memory* étant prévue pour un accès en lecture seule, l'accès en écriture pour cet espace de données a été introduit avec la *surface memory* (*compute capability*

$> 2.x$). Cette dernière permet ainsi de s'affranchir du transit des données par la *global memory* (accessible en écriture), lorsque l'on souhaite procéder à une modification par le GPU des données déclarées en *texture memory*. L'usage de la *global memory* a alors pour désavantage de générer des transferts de données internes au GPU avec la *texture memory*. Bien que la *surface memory* soit aussi accessible en lecture, son principal intérêt reste l'accès en écriture des données exploitées par le *texture memory*.

Shared memory

À la différence de la *device memory*, la *shared memory* est un espace mémoire physiquement intégré dans le SOC du GPU. Sa proximité avec les SMs offre ainsi une bande passante supérieure et un temps de latence plus faible. Son utilisation est exclusive à chaque SM et le processeur hôte ne peut, de ce fait, y accéder. La persistance des données instanciées est limitée à la durée d'exécution de chaque *block* de *threads* dans le SM concerné. Au maximum 1024 *threads*, peuvent ainsi s'échanger des données au moyen de la *shared memory*. La cohérence des données entre *threads* est alors préservée en utilisant l'instruction de synchronisation *___syncthreads()*.

La *shared memory* est répartie en 32 modules appelés *memory banks*. Ces derniers présentent l'avantage, pour les architectures parallèles, de pouvoir travailler de manière simultanée, permettant ainsi d'améliorer la bande passante globale d'accès aux données. En revanche, l'inconvénient de ce type de disposition mémoire est la génération de conflits lorsque des accès concurrents se font sur une même *memory bank*. La résolution de cette problématique passe alors par une sérialisation des accès, dégradant ainsi le temps d'accès aux données. Cependant, dans un *warp*, les accès concurrents à une même *memory bank* ne sont pas en conflit lorsque la requête porte sur la même donnée. Cette dernière est alors diffusée à l'ensemble des *threads* concernés. Depuis la génération *Fermi* (*compute capability* $\geq 2.x$), 32 *memory banks* sont utilisées pour une répartition cyclique en mots de $32\,bits$.

Pour chaque *block*, nous identifions les conflits sur les *memory banks* au moyen de 5.1.

$$\forall x \in [0, 1023],\ \exists ? y \in [0, 1023]\ tq \begin{cases} y \neq x \\ a(y) \neq a(x) \\ a(y) \mod 32 = a(x) \mod 32 \end{cases} \tag{5.1}$$

L'ensemble y des solutions représente alors les identifiants de *threads* dont l'accès à la *shared memory* présente un conflit de *memory bank*. La variable x représente l'identifiant d'un *thread* dans un *block* et a est une fonction d'accès en *shared memory*.

Nous considérons l'emploi de la *shared memory* lorsque les différents *threads* d'un *block* présentent une réutilisation de données. Cet espace mémoire est alors considéré comme une *scratchpad memory*, gérée manuellement dans le *kernel*. Il permet notamment de résoudre la problématique de coalescence des données lorsque les accès ne présentent pas un *stride* unitaire. En conséquence, la localité spatiale s'en retrouve améliorée, ce qui se traduit par un temps d'accès plus faible.

Constant memory

Sur un principe similaire à celui de la *texture memory*, l'usage de la *constant memory* correspond à l'exploitation d'une partie dédiée de la *device memory* du GPU. Cette mémoire dispose cependant d'une mémoire cache distincte (le *constant cache*) pour chaque SM. Comme pour le *texture cache*, le *constant cache* profite de l'accès restreint en lecture

seule pour s'affranchir du processus de vérification de cohérence des données. Cependant, pour chaque *warp*, les accès aux données au moyen de la *constant memory* ne sont pas réalisés sous forme de lignes de cache mais sont au contraire linéarisés. Le temps d'accès aux données est donc proportionel au nombre d'adresses distinctes utilisées dans chaque *warp*. Enfin, la persistance des données instanciées en *constant memory* correspond à la durée de vie du contexte CUDA correspondant. Les données allouées ne peuvent être libérées par le programme du fait qu'il n'y a aucune instruction au sein de l'ISA permettant de le faire.

Afin d'être placés en *constant memory*, les espaces de données doivent impérativement respecter les trois critères suivants :

1. Les accès devront exclusivement être en lecture à l'intérieur des kernels concernés.
2. La somme des empreintes mémoires correspondantes, pour la globalité de l'application, devra être inférieure à la capacité mémoire de la *constant memory*. Chez Nvidia, cet espace mémoire représente 65Ko de données librement disponibles pour le programmeur et 65Ko de données supplémentaires utilisables par le compilateur NVCC.
3. L'allocation de l'espace mémoire devra être statique et défini au moment de la compilation. Dans le cas contraire, le calcul d'une enveloppe convexe maximale au moyen d'une analyse de région permettra d'utiliser une approche dynamique.

Si l'ensemble de ces critères est rempli, le placement est alors considéré comme légal. Cependant, afin de maximiser la bande passante de cet espace mémoire, on ajoutera comme critère d'optimisation la maximisation des accès communs entre *warps*.

5.1.2 Description du sujet d'expérience

Dans la section 5.1.1, nous avons détaillé les spécificités des cinq espaces mémoire pour GPU mis à disposition par CUDA : la *Global memory*, la *Constant memory*, la *Texture memory*, la *Shared memory* et les registres. Nous cherchons maintenant à déterminer les paramètres favorisant la selection d'un de ces espaces mémoire en fonction des caractéristiques de l'algorithme étudié. Ce sujet est vaste et nous avons spécialisé notre étude, dans le cadre de cette thèse, pour un unique accès en lecture par *thread*. Nous nous intéressons, dans ce cadre, à la corrélation entre les différentes méthodes d'accès aux données avec le temps d'exécution d'un *kernel* type. Ces travaux initiaux seront complétés dans de futures publications.

5.1.3 Protocole expérimental

Nous avons défini un *kernel* spécifique à l'évaluation de chaque espace mémoire. Chacun de ces *kernels* est exécuté dix fois avec les même paramètres, afin d'évaluer la variabilité des temps d'exécution.

Les temps d'exécutions sont collectés au moyen de *CUDA Events*. Cette instruction renvoie au processeur hôte, sa propre date d'exécution sur l'accélérateur. En encadrant, dans le code hôte, l'appel d'un *kernel* par deux de ces instructions, nous obtenons, par la différence des deux dates résultantes, le temps d'exécution du *kernel*. L'instruction *CUDA Event* correspondant à la fin d'exécution du *kernel* est associée à l'instruction *cudaEventSynchronize* afin de maintenir une synchronisation avec le processeur hôte.

Les transferts de données entre l'hôte et l'accélérateur transitent par l'unité de *cache L2*, globale au GPU. En conséquence, la persistance des données dans cette mémoire cache vient "polluer", dans le cadre de notre expérimentation, la mesure des temps d'accès

aux différents espaces mémoires. Nous avons de ce fait ajouté, avant chaque exécution d'un *kernel*, une étape "d'empoisonement" du cache L2. N'ayant pas d'information sur la méthode de mise en cache employée, nous transférons $100\,MB$ de données vers la *global memory* afin de s'assurer de la saturation du *cache L2*. Pour rappel, la taille de cette mémoire cache est de $256\,KB$ sur les GPUs d'Endicott et de la Jetson TX1. Les tests que nous avons menés ont montré que l'empoisonnement préalable du cache L2 implique une augmentation du temps d'exécution de chaque *kernel*, justifiant ainsi son utilisation. Enfin, le *cache L1* est nativement invalidé entre chaque *kernel* exécuté, afin de garantir la cohérence des données. De ce fait, aucune action n'a été prise à son sujet.

L'ensemble des évaluations a été compilé avec NVCC 8.0.33 en utilisant l'option $-O0$. Cette précaution nous assure qu'aucune optimisation sur les accès aux données n'est effectuée par le compilateur.

Chaque *kernel* est exécuté pour 2025 *blocks*, saturés par 1024 instances de *threads*. Ces chiffres ont été choisis car ils correspondent à la quantité de données d'une image de résolution standard *full HD*. En adaptant le nombre global d'instances de *threads* à la quantité de données d'une image, nous nous plaçons ainsi dans le cas du parallélisme de données, typique au traitement d'images.

L'ensemble des *kernels* a été spécifié pour que chaque *thread* effectue un accès en lecture et un accès en écriture pour un entier codé sur $8\,bits$. L'accès en écriture sur la *global memory*, correspondant à la sortie du *kernel*, est identique pour l'ensemble des *kernels*. Notre évaluation porte donc sur l'évaluation et la comparaison des performances en lecture des différents canaux d'accès mémoire. Le flot de communication des données est le facteur limitatif pour l'exécution de ces *kernels*. Le flot d'instructions et le flot d'opérations arithmétiques sont réduits au strict minimum. Le temps d'excution des *kernels* dépend donc des temps d'accès aux données.

Chaque *kernel* présente deux variantes se distinguant par une fonction distincte d'accès en lecture :

— $\mathcal{R}_1$, définie en (5.2), distribue les accès distincts de manière cyclique [2] et
— $\mathcal{R}_2$, définie en (5.3), effectue, au contraire, une distribution par blocs regroupant les accès identiques.

$$\mathcal{R}_1 = (threadId \mod s) \times c \tag{5.2}$$

$$\mathcal{R}_2 = \lfloor \frac{threadId}{s} \rfloor \times c \tag{5.3}$$

La distinction entre $\mathcal{R}_1$ et $\mathcal{R}_2$ permet d'évaluer l'impact de la contiguité des accès communs. Pour ces deux fonctions, $threadId$ représente l'identifiant du *thread* courant, tel que $threadId \in [0; 1023]$ pour chaque *block*. Le paramètre s définit le nombre d'accès distincts, tel que $s \in [1; 1024]$. Enfin, le coefficient c modifie le *stride*, correspondant à l'écart spatial entre les accès aux données. Ce coefficient permet d'augmenter la quantité de *cache miss* ce qui engendre une augmentation des communications avec la mémoire physique, les transferts se faisant par lignes de données contiguës. À ce sujet, le bus mémoire de la TX1 est de 8 *Bytes* et celui de la K2000 d'Endicott de 16 *Bytes*. Nous avons contraint cette expérimentation aux cas d'étude $c \in \{1, 16\}$ tel que :

c = 1 correspond à un cas de parfaite coalescence des données et

c = 16 correspond à la plus grande largeur de bus mémoire pour les deux plateformes considérées.

2. Méthode de type Round-robin

Enfin, nous ne considérons pas dans les résultats de cette évaluation les temps de transfert pour l'échange de données entre le processeur hôte et le GPU.

Nous détaillons à présent les méthodes d'évaluation des différents espaces mémoire.

Évaluation des registres

Pour l'ensemble des figures, l'évaluation du temps d'accès des registres correspond à :

1. l'initialisation d'une variable scalaire dans l'espace des registres,
2. l'écriture de la valeur de cette variable, dans la *global memory* en sortie.

Chaque unité registre est dédiée à un unique *thread*. Le temps d'exécution correspond alors nécessairement à 1024 accès distincts dans le référentiel d'un *block* et à 32 accès distincts dans le référentiel d'un *warp*. De ce fait, $\mathcal{R}_1$ et $\mathcal{R}_2$ ne peuvent s'appliquer pour ce cas.

Évaluation de la *shared memory*

Lorsque les données sont utilisées au-delà du context d'un *block* (au sein d'une unité SM), la *shared memory* ne peut être employée de manière exclusive du fait que :

— le processeur hôte ne peut y accéder et
— la cohérence des données est limitée à un unique *block*.

Ainsi, notre évaluation de la *shared memory* est réalisée en faisant préalablement transiter les données par la *global memory*. Les performances de la *global memory* ont donc une influence certaine sur les résultats obtenus pour la *shared memory*. Ce binôme nous permet de plus d'observer le comportement de la *shared memory* comme l'équivalent, pour la *global memory*, d'une mémoire cache L1 gérée manuellement par le développeur.

Le *kernel* servant à l'évaluation de la *shared memory* est spécifié de la façon suivante :

1. Chaque instance du *kernel* débute en effectuant un unique accès en lecture à la *global memory* selon les fonctions $\mathcal{R}_1$ ou $\mathcal{R}_2$.
2. Un branchement conditionnel vérifie auparavant que l'accès aux données se fait de manière unique pour chaque *block* limitant ainsi la redondance de communications.
3. Pour les *threads* concernés, chaque donnée récupérée est alors transférée dans la *shared memory*.
4. Afin de nous assurer de la cohérence de ces données pour l'intégralité de chaque *block*, nous effectuons une synchronisation des *threads* au moyen de l'instruction *___syncthreads()*.
5. Chaque instance du *kernel* procède ensuite à un accès en lecture à la *shared memory* afin d'écrire, en sortie de *kernel*, la valeur récupérée dans la *global memory*.

Évaluation des autres espaces mémoire

L'évaluation des autres espaces mémoire se fait selon l'implémentation, dans chaque *kernel*, du *pattern* algorithmique commun suivant :

1. lecture des données dans l'espace mémoire évalué selon les fonctions $\mathcal{R}_1$ ou $\mathcal{R}_2$ et
2. écriture de la valeur lue dans la *global memory*.

5.1.4 Analyse et interprétation des résultats

Nous abordons dans un premier temps le formalisme employé pour la représentation des résultats. Dans un second temps, nous procédons pour chaque plateforme à l'analyse de ces résultats, représentés dans les figures 5.1 à 5.4 pour *Endicott* puis dans les figures 5.5 à 5.8 pour la *Jetson TX1*.

Représentation des résultats

Afin de pouvoir comparer les résultats des fonctions d'accès $\mathcal{R}_1$ et $\mathcal{R}_2$, nous avons défini comme métrique commune le nombre d'accès distincts dans le référentiel d'un *block* de 1024 *threads*. Cette métrique correspond à :

— la fonction $N1_{block}$ en (5.4) pour $\mathcal{R}_1$,
— la fonction $N2_{block}$ en (5.5) pour $\mathcal{R}_2$.

$$\forall s \in [1;1024],\ N1_{block}(s) = s \tag{5.4}$$

$$\forall s \in [1;1024],\ N2_{block}(s) = \lceil \frac{1024}{s} \rceil \tag{5.5}$$

La même démarche a été effectuée en utilisant pour référentiel un *warp* de 32 *threads*. Nous avons alors adapté la représentation des résultats dans ce référentiel en modifiant :

— la fonction (5.4) en (5.6) afin d'obtenir $N1_{warp}$,
— la fonction (5.5) en (5.7) pour $N2_{warp}$.

$$\forall s \in [1;32],\ N1_{warp}(s) = s \tag{5.6}$$

$$\forall s \in [1;32],\ N2_{warp}(s) = \frac{1024}{s \times 32} = \frac{32}{s} \tag{5.7}$$

Dans le cas de $N1_{warp}$, les données sont redondantes entre *warps*. Pour $N2_{warp}$ les données sont au contraires distinctes. Ces deux fonctions nous permettent d'évaluer dans une unité SM l'effet lié à la réutilisation de données entre *warps*.

La **moyenne des temps d'exécutions**, pour les dix exécutions de chaque *kernel*, est représentée par un trait hachuré pour chaque espace mémoire. Nous avons ajouté en trait continu l'**écart-type par rapport à la moyenne** pour l'ensemble de ces courbes. Plus l'écart entre ces traits est important, plus la variation des temps mesurés est forte. Un unique trait continu indique au contraire un écart-type proche de zéro. Ce dernier cas correspond à une excellente stabilité des temps d'exécution ayant pour conséquence une meilleure reproductibilité des temps d'exécution.

Nous avons volontairement figé les échelles des figure 5.1 à 5.8, afin d'en faciliter leurs comparaisons.

Enfin, le temps d'exécution du *kernel* évaluant les registres a été "extrapolé" à l'ensemble des valeurs d'accès distincts. Les unités registres offrant le temps d'accès le plus faible et en considérant ce temps comme négligeable, cette courbe permet d'apprécier :

— le temps d'exécution minimal pouvant être atteint,
— le temps d'écriture en sortie, commun à l'ensemble des *kernels* et
— le temps d'accès en lecture des différents espaces mémoire [3].

Résultats pour Endicott - Quadro K2000

Dans le **référentiel d'un *block***, les temps d'exécution pour les différents espaces mémoire sont représentés :

— dans la figure 5.1 pour la **distribution cyclique** des accès et
— dans la figure 5.2 pour la **distribution par bloc**.

Dans le **référentiel d'un *warp***, les résultats sont représentés :

— dans la figure 5.3 pour la **distribution cyclique** et
— dans la figure 5.4 pour la **distribution par bloc**.

3. en évaluant la différence entre la courbe des registres et les courbes des différents espaces mémoire.

Pour la ***constant memory***, les temps d'accès augmentent globalement avec le nombre d'accès distincts pour l'ensemble des représentations.

Pour la distribution cyclique (figure 5.1a), le temps d'accès maximal est atteint pour 32 accès distincts et reste constant au-delà, révélant ainsi un phénomène de saturation. Pour la distribution par bloc (figure 5.2a), le temps d'accès reste au contraire minimal et constant jusqu'à 32 accès distincts et augmente au-delà. Le maximum est atteint pour 1024 accès distincts. Cette différence de résultat révèle l'influence du modèle de distribution employé sur les temps d'accès aux données. Nous constatons que le point de rupture est commun dans les deux cas, soit 32 accès distincts et que la saturation a lieu :

— au-delà du point de rupture pour la distribution cyclique et
— avant le point de rupture pour la distribution par bloc.

Dans les deux cas, ce point correspond au cas où chacun des *threads* composant un *warp* procède à un accès distinct. Ces observations nous permettent de confirmer, dans ce cas de figure, que :

— l'utilisation d'accès communs entre *warps* ne permet pas d'accélérer le temps d'accès global à la constant memory,
— le temps d'accès global à la *constant memory* évolue avec le nombre d'accès distincts dans un *warp*.

Ce dernier point se vérifie dans les figures 5.3a et 5.4a où, dans le référentiel d'un *warp*, les courbes correspondant au temps d'accès à la *constant memory* sont similaires.

La pression sur les échanges de données en mémoire, exercée par le paramètre $c = 16$, implique un comportement distinct dans l'ensemble des cas. Dans le référentiel d'un *block*, nous observons dans les figures 5.1b et 5.2b le même phénomène de dégradation des temps d'exécution au-delà de 128 accès distincts. En revanche, dans le cadre d'un *warp* :

— la distribution cyclique de la figure 5.3b ne révèle pas de changement notable tandis que
— la distribution par blocs de la figure 5.4b connait un taux de croissance plus prononcé, au delà de 4 accès distincts.

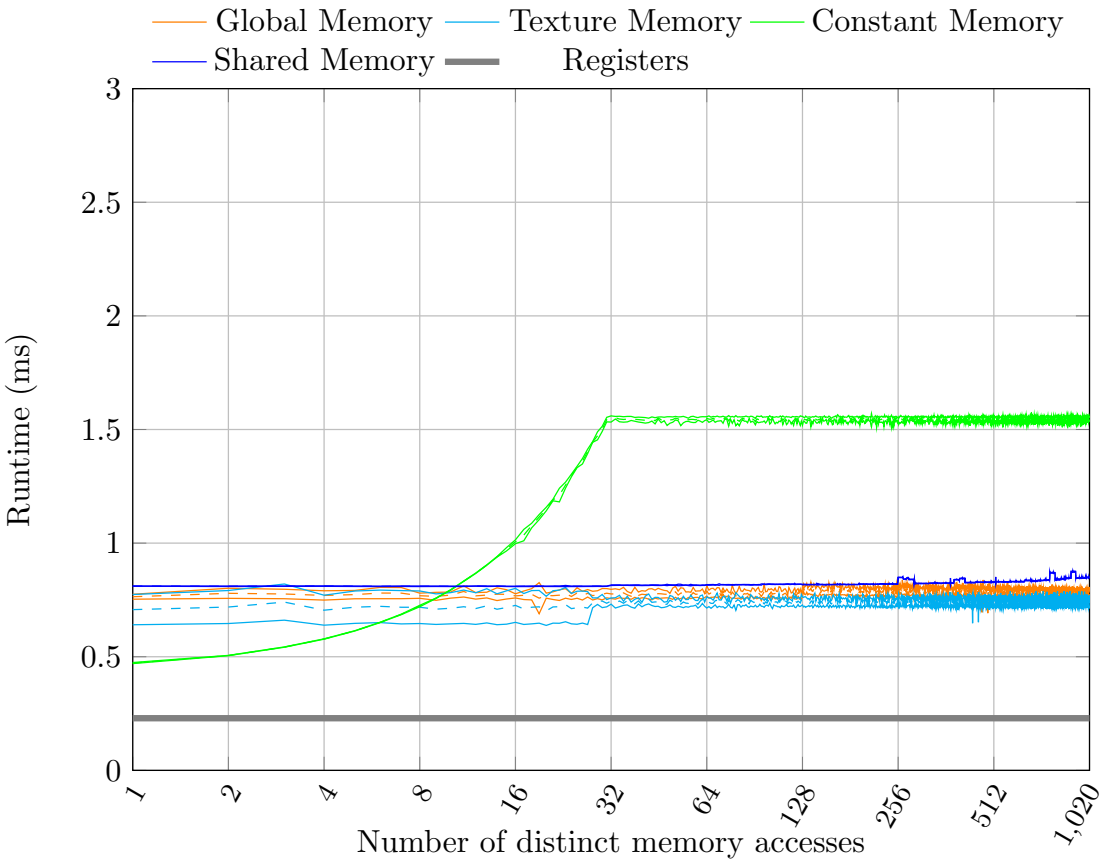
Dans le premier cas, le nombre d'accès distincts reste inférieur ou égal à 32 dans le référentiel d'un *block*. Dans le second cas, la rupture observée à partir de 4 accès distincts correpond à 128 accès distincts dans le référentiel d'un *block*. Nous en déduisons que seul le nombre d'accès distincts, global à un *block*, semble avoir un lien avec l'augmentation des temps d'accès globaux à la *constant memory*. Ces résultats nous révèlent l'implication d'une mémoire cache, pouvant potentiellement correspondre au cache L2 de la *device memory*, dans l'utilisation de la *constant memory*.

Enfin, nous constatons que la *constant memory* présente les meilleures performances d'accès en lecture (à l'exclusion des unités registre) pour moins de 8 accès distincts par *warp*. Cette valeur est cependant influençable par la valeur du *stride* des accès aux données comme le montre la figure 5.4b.

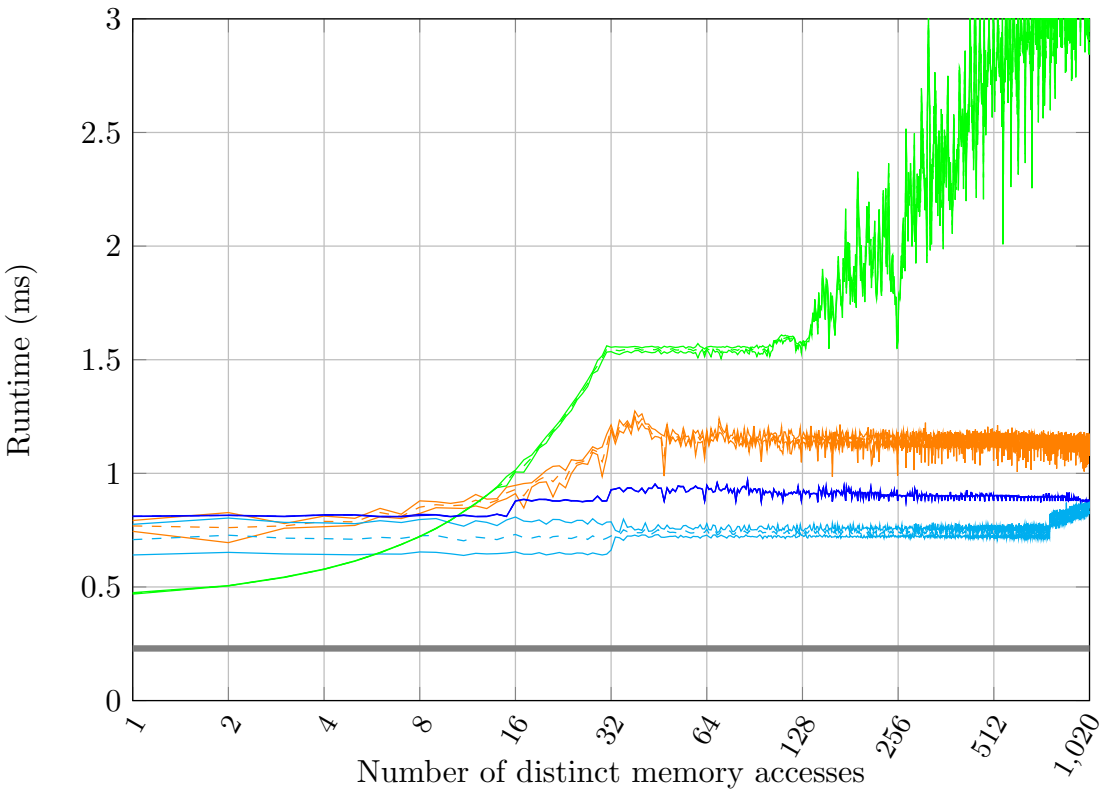
Pour la ***texture memory***, nous observons que la variance des temps d'exécution est plus importante :

— en-dessous de 32 accès distincts pour la distribution cyclique (figure 5.1a) et
— au-delà de 32 accès distincts pour la distribution par blocs (figure 5.2a).

Le nombre d'accès distincts par *warp* semble donc avoir un effet négatif sur la stabilité des temps d'accès en lecture pour la *texture memory*. Les figures 5.3a et 5.4a renforcent ce constat, avec une variance plus élevée sur l'ensemble de la courbe. Cependant, dans ce cas de figure, l'augmentation de la variance ne vient pas dégrader les temps d'accès. Les accès communs dans un *warp* permettent ici de réduire, de façon non prédictible, les temps d'accès en lecture à la *texture memory*.

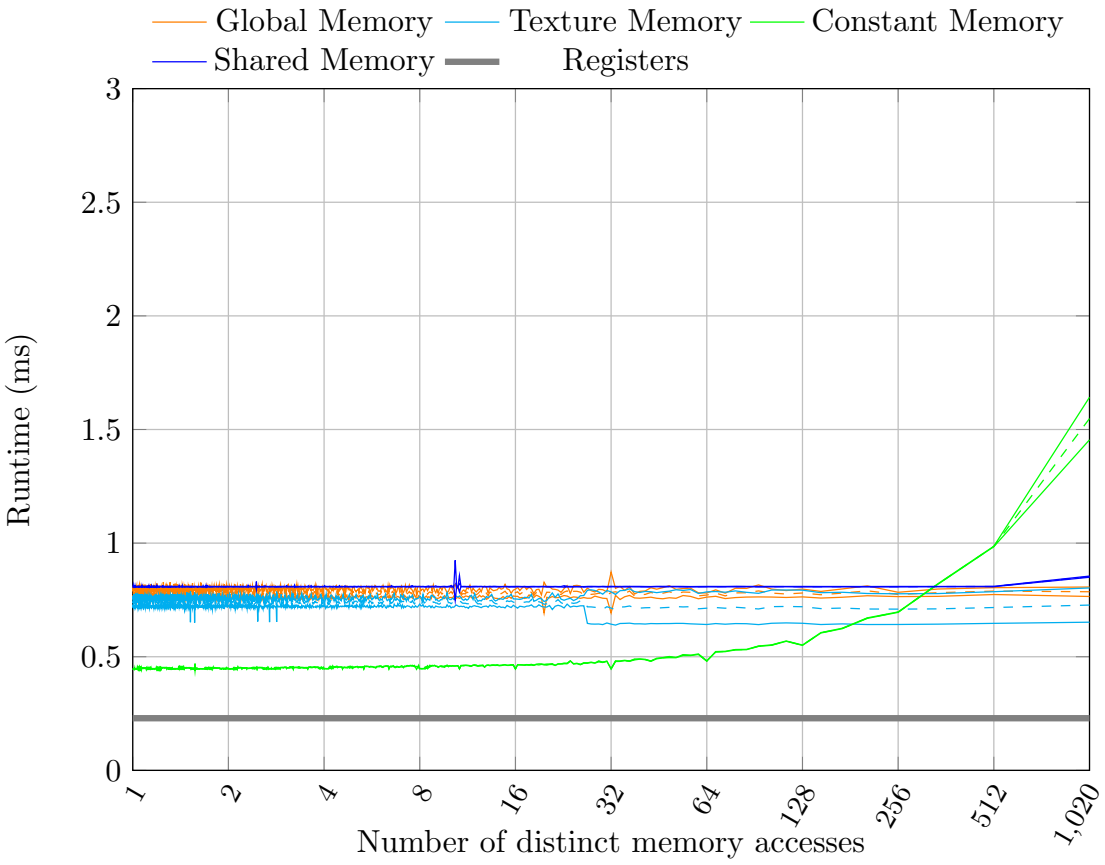


(a) Accès mémoire contigues. $c = 1$

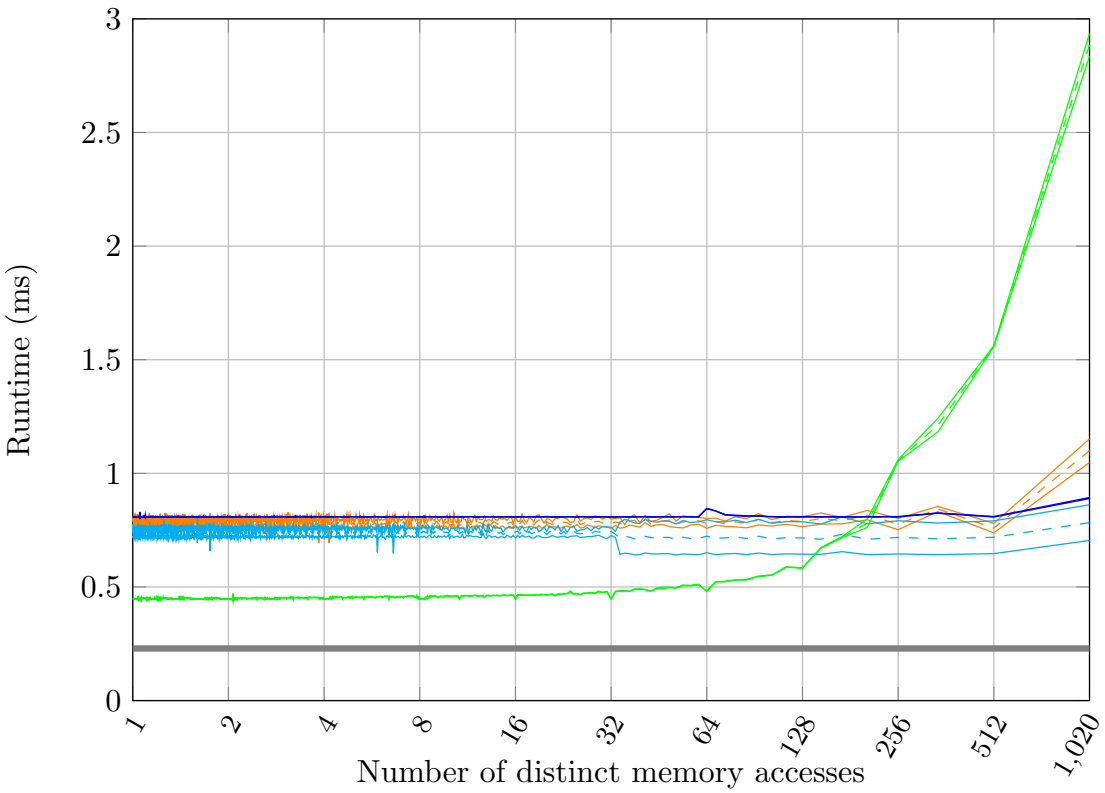


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.1 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Block*

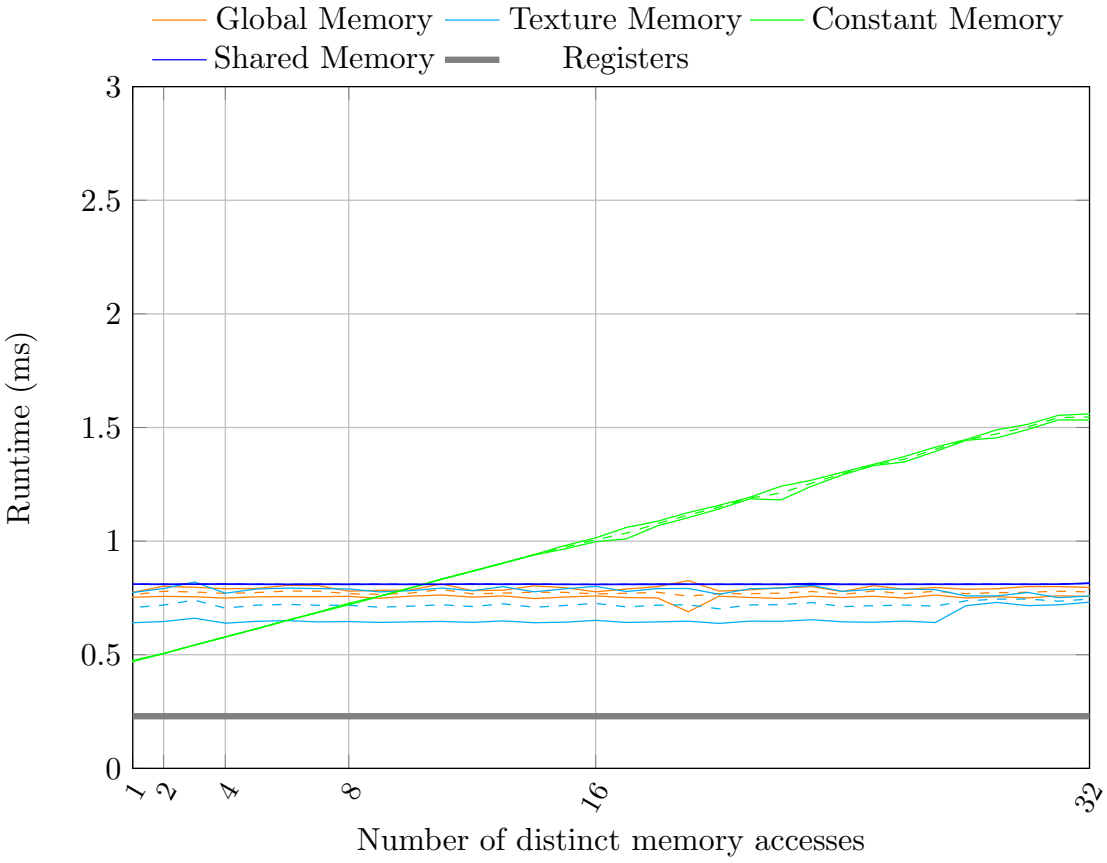


(a) Accès mémoire contigues. $c = 1$

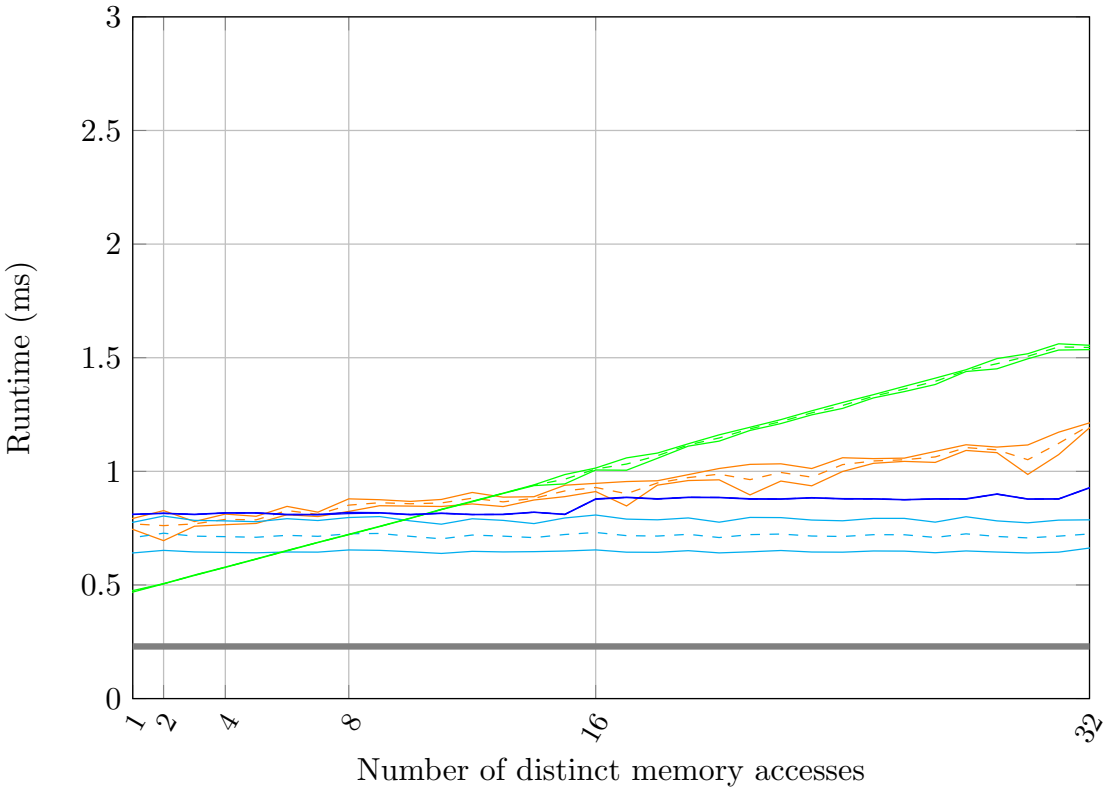


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.2 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Block*

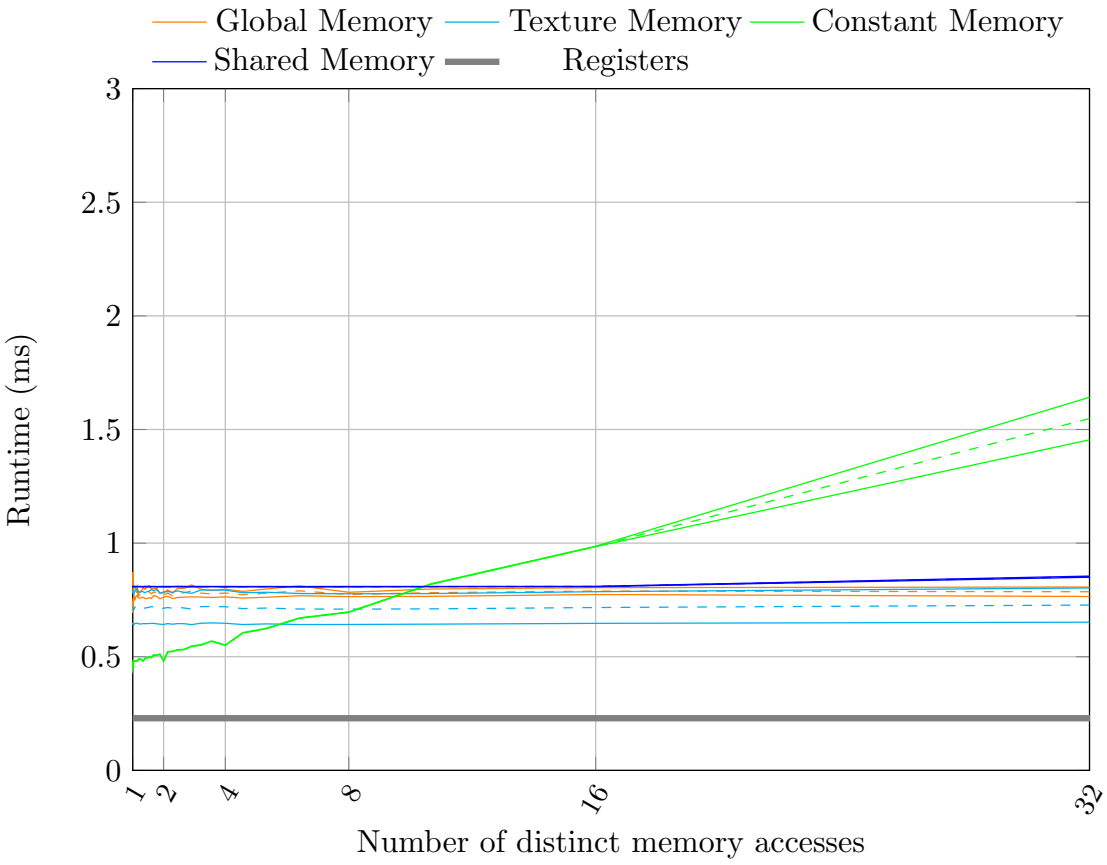


(a) Accès mémoire contigues. $c = 1$

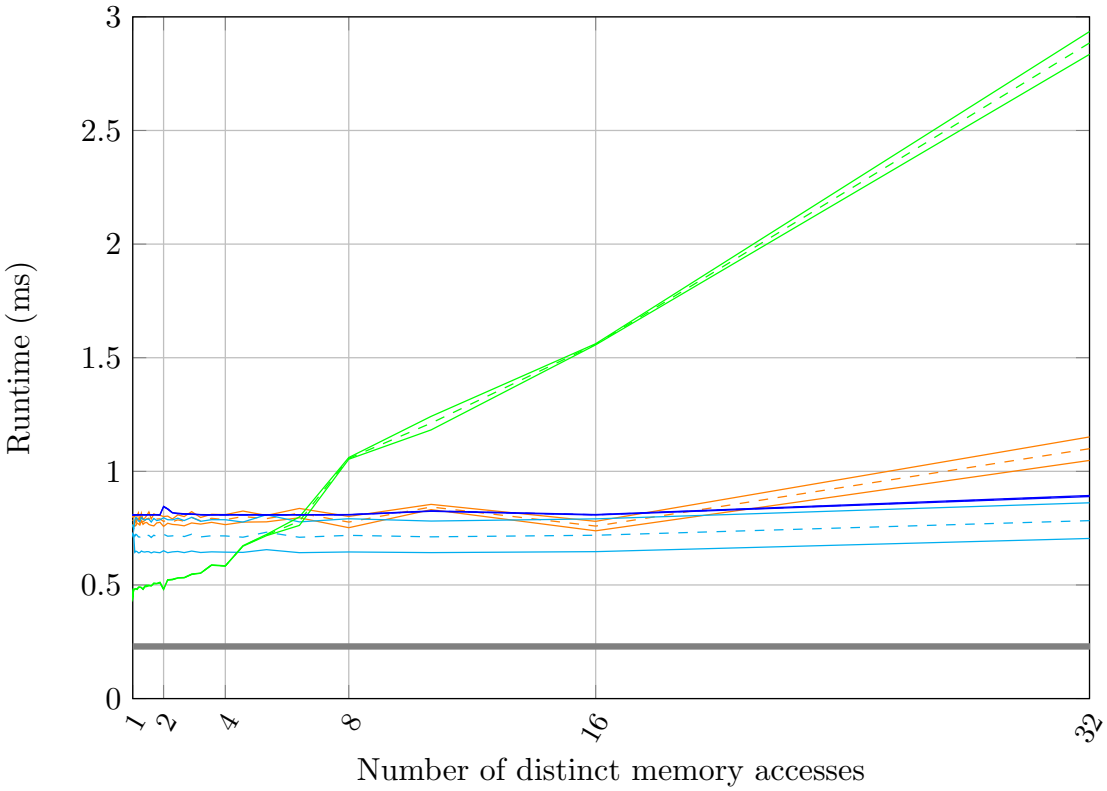


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.3 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Warp*



(a) Accès mémoire contigues. $c = 1$



(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.4 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Warp*

Le *stride* pour l'accès aux données semble ne pas avoir d'effet notable sur les temps d'accès. Pour un *stride* de 16, nous ne constatons, dans les figures 5.1b et 5.2b, qu'une légère augmentation, lorsque le nombre d'accès distincts dans un *block* est proche de 1024. Le cache de la *texture memory* semble donc moins affecté par l'augmentation du stride dans les accès aux données. Ce constat reste cependant à vérifier avec l'augmentation du *stride* à des valeurs plus élevées.

Enfin, nous noterons que pour l'ensemble des cas, les temps d'accès à la *texture memory* restent inférieurs à ceux de la *global memory*.

L'évaluation de la ***global memory*** laisse apparaitre, dans le cas d'accès coalescents ($c = 1$), un temps d'accès stable et une faible variance pour les deux modèles de distribution (figures 5.1a et 5.2a). Cependant l'augmentation du *stride* laisse apparaître un impact direct sur les temps d'exécution. Dans les figures 5.1b et 5.3b, nous observons pour la distribution cyclique une augmentation linéaire des temps d'exécution jusqu'à environ 32 accès distincts. Au-delà de ce point, nous observons dans la figure 5.1b, un phénomène de saturation des temps d'accès. De plus, la figure 5.2b met en évidence l'aspect bénéfique de la localité temporelle, apportée par la distribution par blocs des accès, en repoussant l'augmentation des temps d'exécution au-delà de 128 accès distincts. Les accés commun entre *warps* ont donc un effet limité sur les temps d'accès.

Nous en concluons que :

— la coalescence des données permet d'améliorer les temps d'accès à la *global memory*,
— les accès communs dans un *warp* permettent de limiter la pénalité des accès non coalescents.

La ***shared memory*** présente pour l'ensemble des résultats une stabilité supérieure et une faible variance des temps d'exécution. Pour rappel, nous évaluons cet espace mémoire conjointement à la *global memory*. Les données, provenant de cette dernière, transitent exclusivement par le cache L2 tandis que le cache L1 est dédié à la *local memory*. Nous observons, dans les figures 5.1a, 5.2a, 5.3a et 5.4a, que la *shared memory* ne permet pas d'obtenir des temps d'accès inférieurs à ceux de la *global memory* lors d'accès coalescents et redondants. Le cache L2 permet ainsi d'atteindre un niveau de performance équivalent dans ce cas. En revanche, lors d'accès non coalescents (figures 5.1b, 5.2b, 5.3b et 5.4b), la *shared memory* permet de limiter la pénalité des *cache-miss* sur la *global memory* :

— en effectuant le préchargement des données de la *global memory* vers la *shared memory*,
— en chargeant les données sur la *shared memory* de manière à recréer une coalescence des accès en lecture, améliorant ainsi la localité spatiale.

Le gain apporté par la *shared memory* est particulièrement visible dans la figure 5.1b. la distribution cyclique ayant un impact sur la localité temporelle des données, la *shared memory* permet de limiter le phénomène de *cache-miss* par une gestion manuelle de la persistence des données.

Nous en déduisons que :

— l'usage systématique de la *shared memory* n'est pas fondé du fait qu'elle n'apporte pas un gain systématique par rapport au cache L2,
— la *shared memory* permet de limiter la dégradation des temps d'accès à la *global memory* notamment lors d'accès distincts et non coalescents dans les *warps*.

Cependant, notre utilisation de la *shared memory* évite les phénomènes de conflit sur les accès concurrents en lecture aux *memory banks* de la *shared memory*. Dans le cas contraire, les performances de cet espace mémoire auraient été dégradées.

En **conclusion** : Les résultats présentés concernent le GPU *Quadro K2000*, basé sur une architecture *Kepler*.

Pour notre cas d'évaluation, les temps d'accès en lecture [4] sont minimisés en utilisant :
— la *constant memory* lorsque le nombre d'accès distincts par *warp* reste inférieur à huit et
— la *texture memory* au-delà.

Ce point de transition est cependant influencé par :
— le nombre d'accès distincts entre *warps* et
— la taille du *stride* pour l'accès aux données.

Le domaine de supériorité de la *constant memory* est alors réduit au profit de la *texture memory*.

Au sujet de la stabilité des temps d'accès en lecture, la *shared memory* a montré une variance minimale pour l'ensemble des cas testés. Ainsi, pour de multiples exécutions, cet espace mémoire offre la meilleure récurrence des temps d'accès.

Résultats pour la Jetson TX1 - T210

Nous abordons à présents les résultats obtenus pour le GPU de la *TX1*. Notre analyse se focalise en particulier sur les différences observables, par rapport aux résultats du GPU *Quadro K2000* d'Endicott.

Dans le **référentiel d'un *block***, les temps d'exécution pour les différents espaces mémoire sont représentés :
— dans la figure 5.5 pour la **distribution cyclique** des accès et
— dans la figure 5.6 pour la **distribution par bloc**.

Dans le **référentiel d'un *warp***, les résultats sont représentés :
— dans la figure 5.7 pour la **distribution cyclique** et
— dans la figure 5.8 pour la **distribution par bloc**.

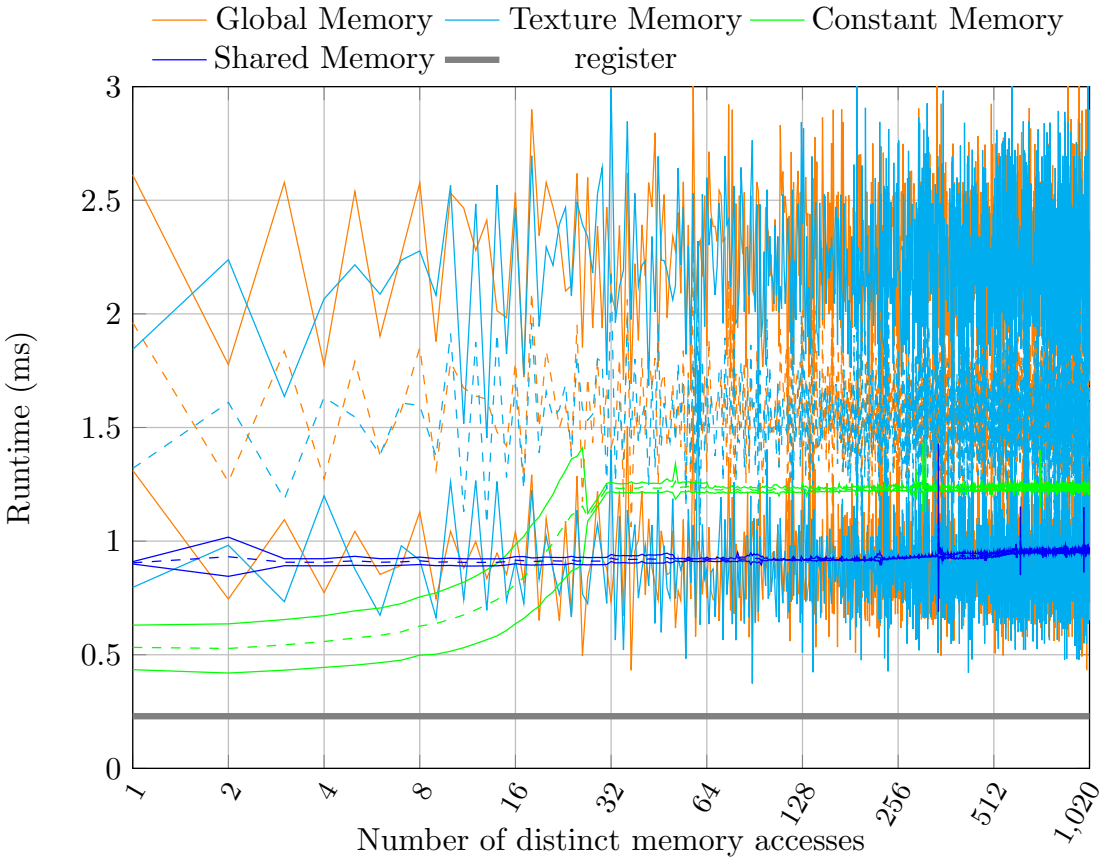
Comparée à la *Quadro K2000*, nous relevons, pour l'ensemble des représentations, une **nette augmentation de la variance** des temps d'exécution. Le format basse consommation des mémoires *LPDDR4* de la *TX1* semble être une raison légitime. Afin de vérifier ce point, il serait nécessaire, en complément de cette étude, d'effectuer la même évaluation sur une architecture comparable à la *Quadro K2000*, soit :
— de type *Maxwell* et
— employant des unités mémoire de type *GDDR5*.

Nous constatons, cependant, que la ***shared memory*** fait exception avec une variance nettement plus faible, comparable à celle de la Quadro K2000. Pour l'ensemble des cas, elle permet d'améliorer sensiblement le temps d'accès aux données de la *global memory*.

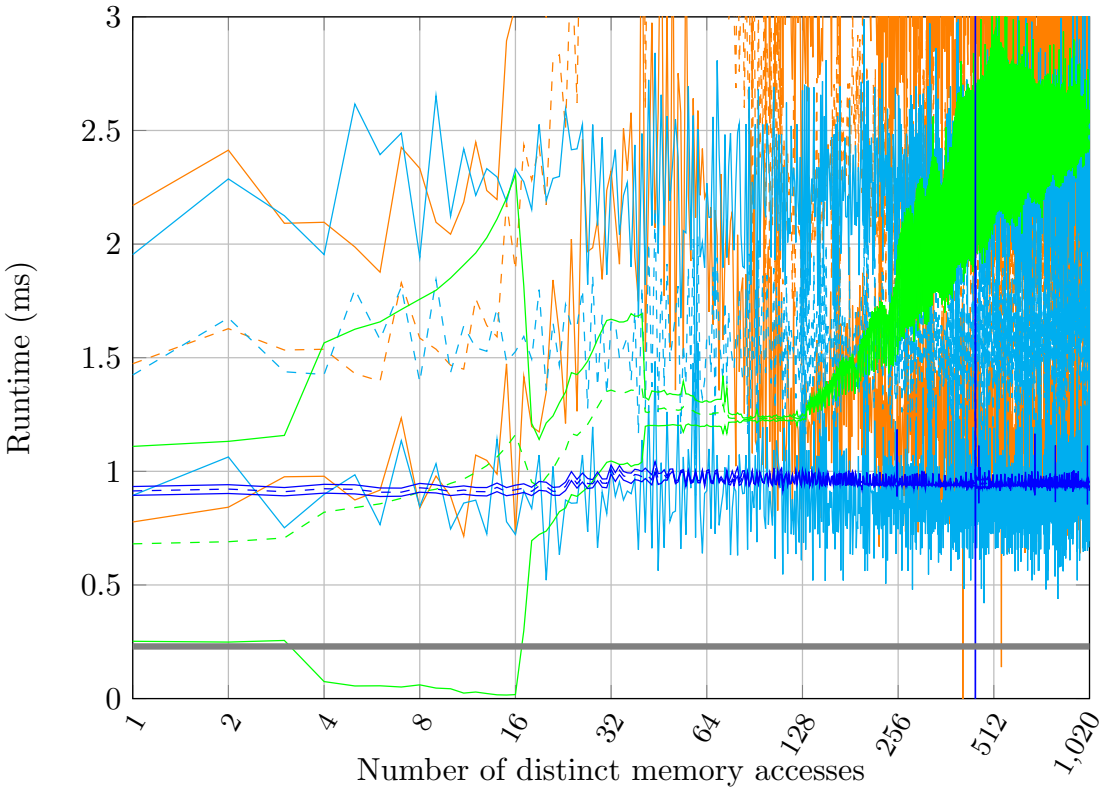
Pour la ***constant memory***, en ne considérant pas la variation plus prononcée des mesures, les résultats sont similaires à ceux d'Endicott. Nous retrouvons ainsi le même comportement, incitant à maximiser le nombre d'accès communs entre les *threads* d'un même *warp*. Au niveau de la variance, nous notons en particulier une forte augmentation, lorsqu'il existe une réutilisation des données dans un *warp*. Dans le cas contraire, la variance reste modérée au prix de l'augmentation attendue des temps d'accès. Les accès à la constant memory étant linéarisés, l'absence de réutilisation des données engendre une augmentation des *cache-misses*. Ces derniers se traduisent par une pénalité systématique des temps d'accès, caractérisée par une faible variance des résultats. La *constant memory* permet ainsi, pour cette architecture, d'améliorer les temps d'accès en échange d'une dégradation de leur stabilité.

À la différence de la quadro K2000, la ***texture memory*** et la ***global memory*** présentent des performances similaires pour leur temps d'accès moyen. Cependant, la *global*

4. à l'exception des accès aux registres

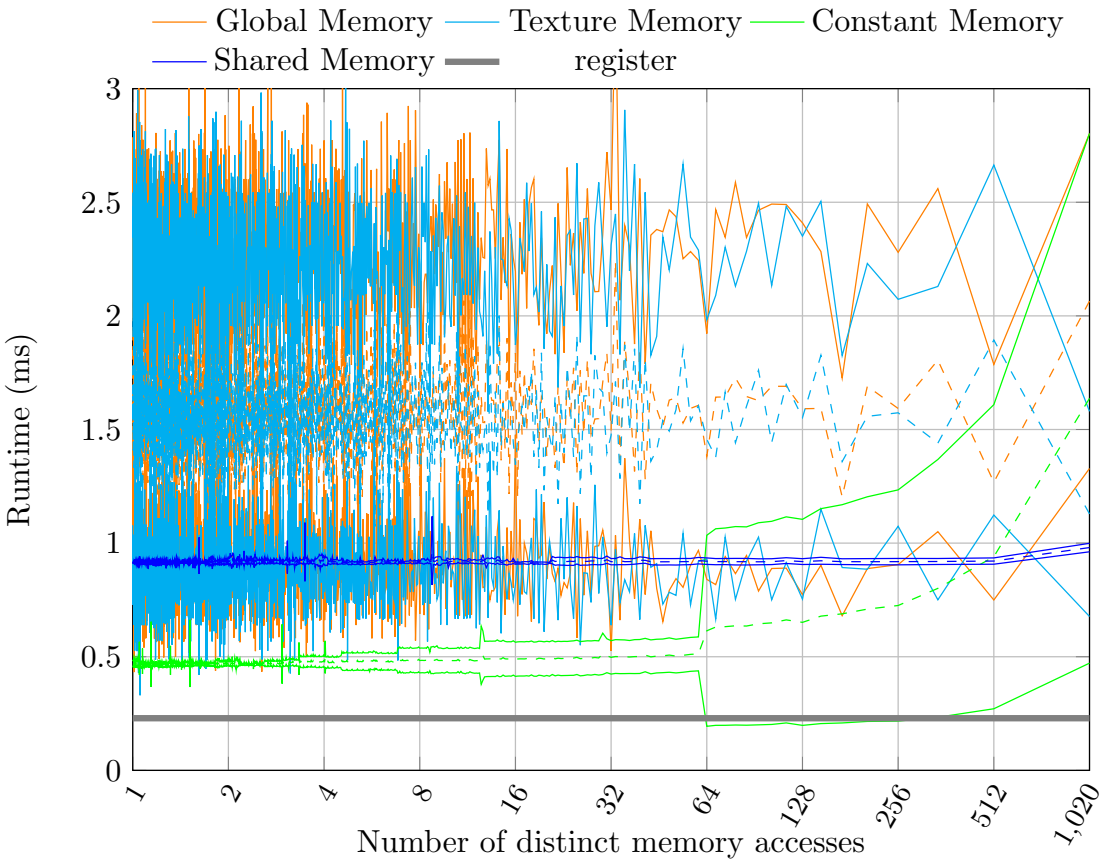


(a) Accès mémoire contigues. $c = 1$

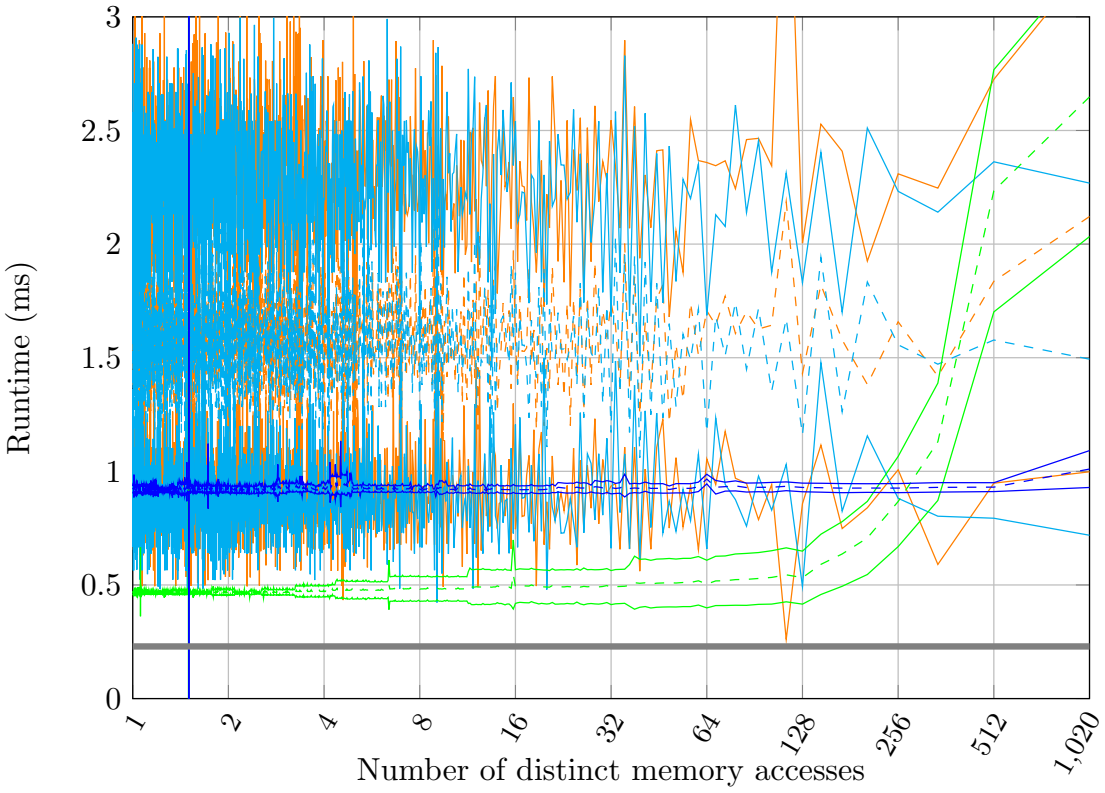


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.5 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Block*

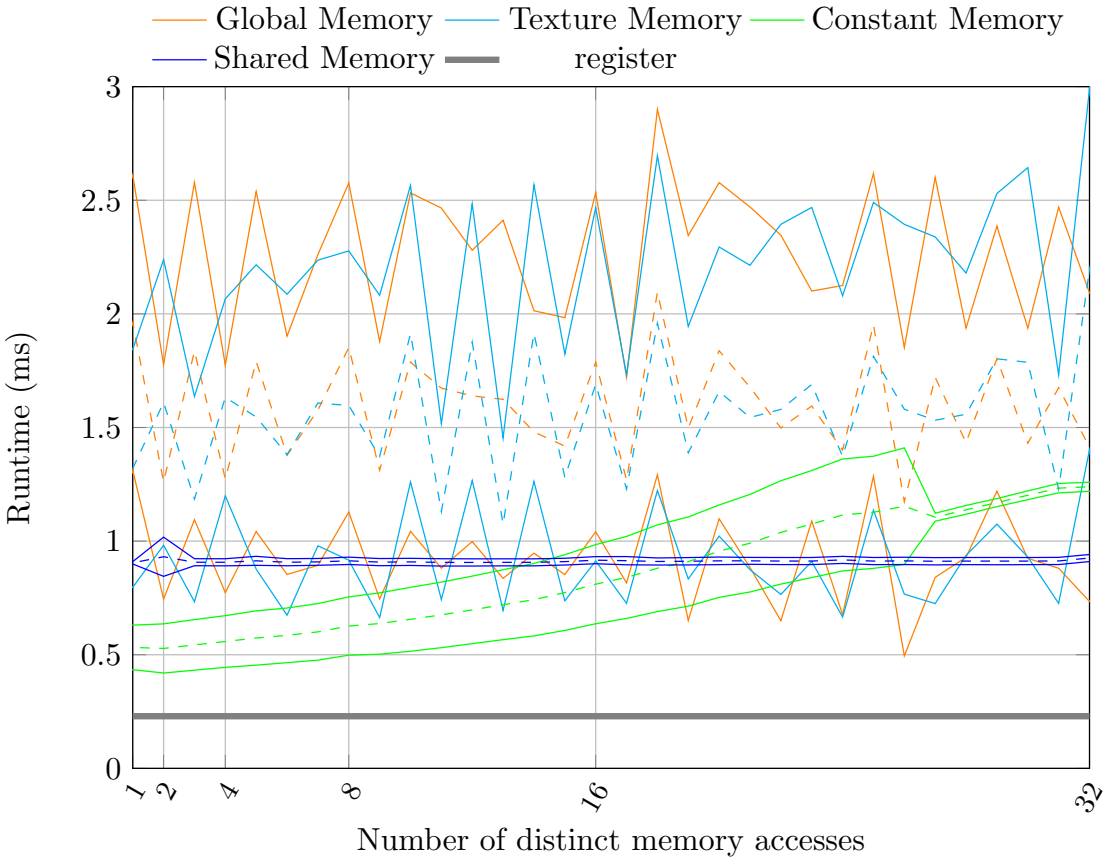


(a) Accès mémoire contigues. $c = 1$

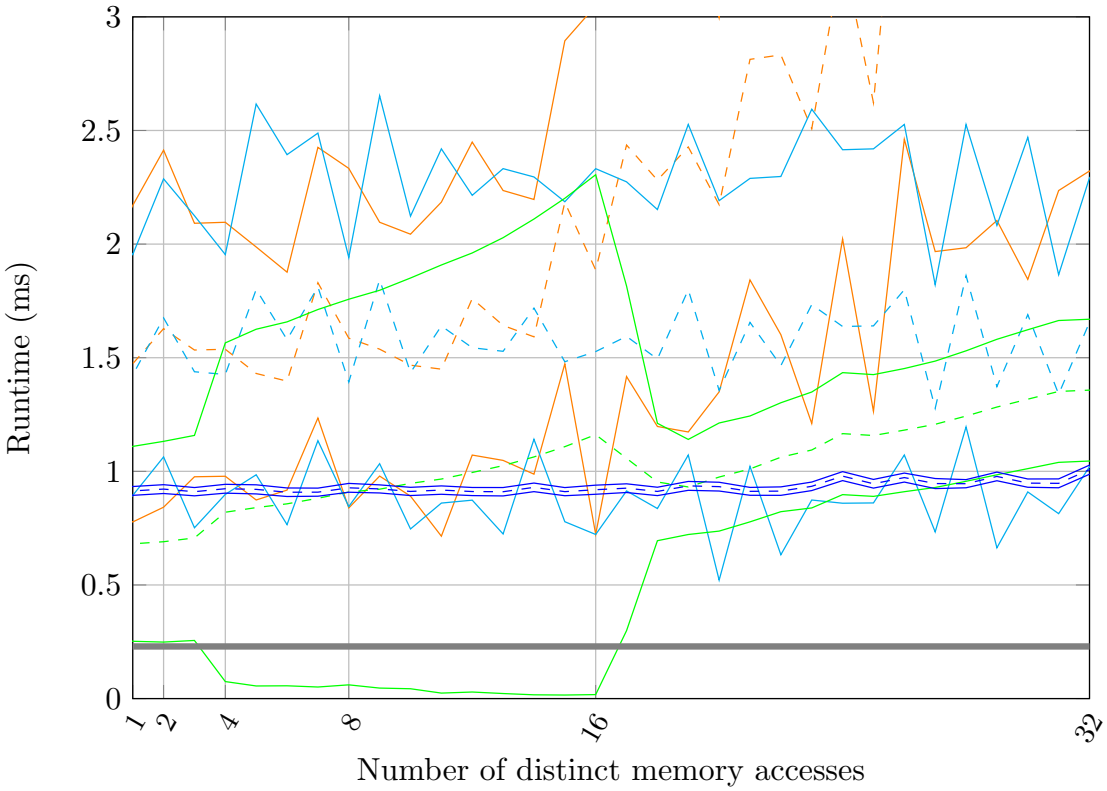


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.6 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Block*

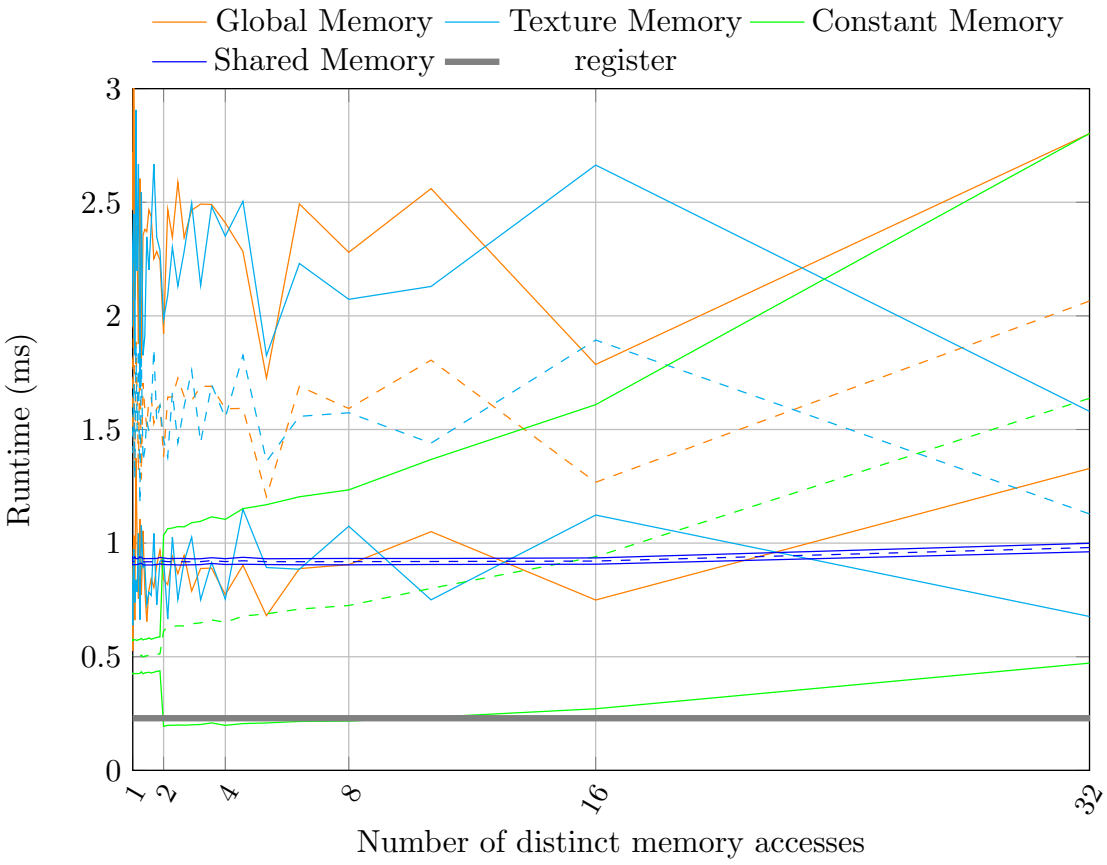


(a) Accès mémoire contigues. $c = 1$

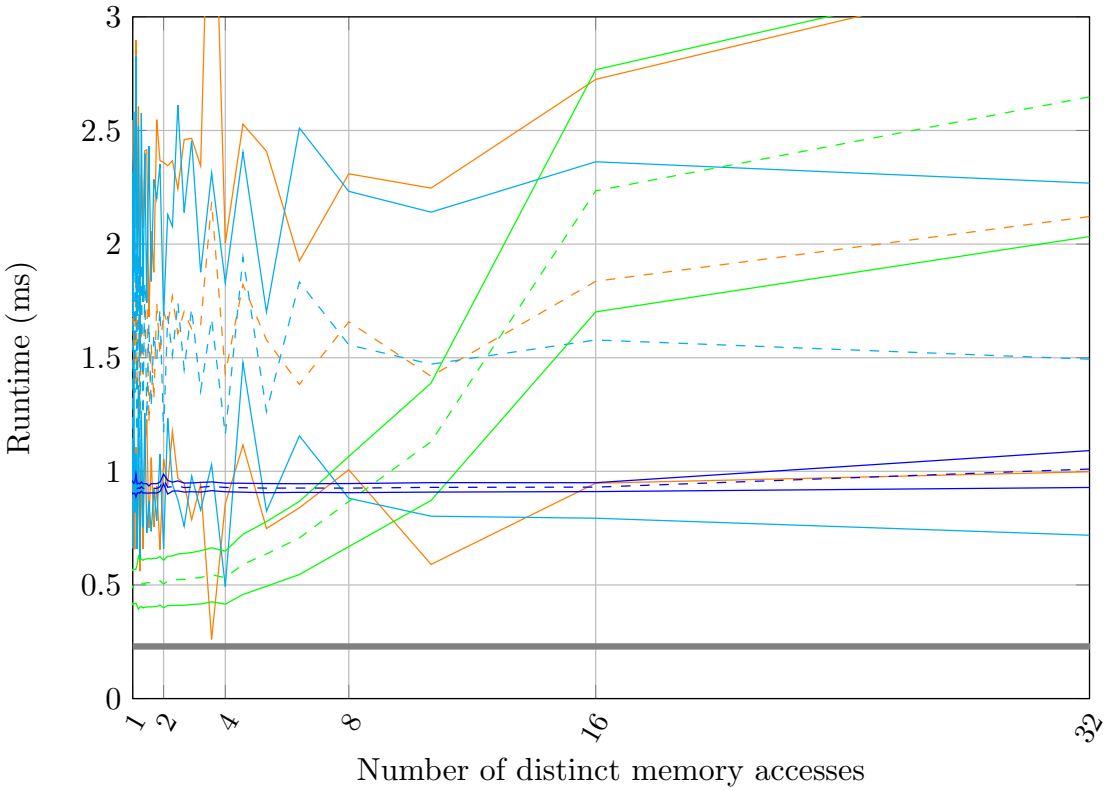


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.7 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Warp*



(a) Accès mémoire contigues. $c = 1$



(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.8 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Warp*

memory reste plus sensible à l'augmentation de la taille du *stride* lors des accès aux données. Ce phénomène est particulièrement observable dans la figure 5.7b, où l'augmentation du nombre d'accès distincts, affecté d'un stride de 16 $Bytes$, engendre une augmentation du temps d'accès global.

En **conclusion** : Jusqu'à 16 accès distincts par *warp*, l'usage de la *constant memory* permet de minimiser les temps d'accès, lorsque ceux-là sont contigus. Au-delà, le placement des données communes en *shared memory* donne les meilleurs résultats. Ce point de transition est cependant influencé par le *stride* d'accès aux données. Enfin, même si la *texture memory* présente des résultats similaires, comparé à la *global memory*, sa meilleure résistance au stride la rend préférable.

Pour la variance des résultats, l'augmentation relevée nous laisse supposer l'usage d'un modèle de *mapping* des mémoires cache basé sur un algorithme aléatoire de remplacement des lignes de cache. Ce type d'algorithme, du fait de sa simplicité de mise en œuvre, permet de répondre au besoin d'économie en énergie. Ce dernier point est cohérent avec les applications embarquées visées par la TX1. Ce principe de renouvellement du cache est d'ailleurs employé pour les architectures CPU basse consommation du fabricant *ARM* et notamment pour l'*A57* présent dans la *TX1*.

5.1.5 Conclusion

Nous avons étudié les performances en lecture de quatre espaces mémoire : la *global memory*, la *texture memory*, la *constant memory* et la *shared memory*. Cette analyse s'est focalisée en particulier sur l'influence des temps d'accès mémoire en fonction de trois principaux paramètres :

— le nombre d'accès distincts dans chaque *block*,
— la distribution des accès distincts et
— le *stride* des accès aux données.

Les communications mémoire constituent ici le facteur influençant directement le temps d'exécution des *kernels*.

Les expérimentations ont été menées sur deux architectures GPU distinctes : La *Quadro K2000* et la *TX1*. La première est de génération *Kepler* et dispose d'une mémoire dédiée de type *GDDR5*. La seconde, adaptée aux contraintes énergétique d'un usage embarqué, est de génération *Maxwell* et partage sa mémoire de type *LPDDR4* avec le processeur hôte.

Entre les deux architectures, nous retrouvons des similitudes sur le comportement des différents espaces mémoire. Ainsi, lorsque les accès sont coalescents, nous avons observé sur la *Quadro K2000* les temps d'accès minimaux pour :

— la *constant memory* de 1 à 8 accès communs par *warp* et
— la *texture memory* au-delà de 8 accès communs.

Pour la *TX1*, les temps d'accès sont minimaux pour :

— la *constant memory* de 1 à 16 accès communs par *warp* et
— la *shared memory* au-delà de 16 accès communs.

Ce point de transition varie cependant, au profit de la *shared memory*, en augmentant la valeur du *stride* lors des accès mémoire.

Cependant, la Jetson TX1 présente une variance des temps d'accès plus prononcée. Ce constat peut s'expliquer par :

— la mémoire basse consomation de type *LPDDR4* utilisée et
— l'utilisation d'une méthode de remplacement aléatoire des lignes de cache qui est plus économe en consommation énergétique.

Seule la *shared memoy* a présenté une faible variance sur ses temps d'accès.

Afin de compléter ces résultats, il conviendrait d'étudier en complément :

— les performances des accès en écriture,
— la variation de la taille du *stride*,
— la variation de la taille des données utilisées,
— la variation de la quantité globale de données exploitées,
— la variation du nombre d'accès communs entre *blocs*, afin d'évaluer le comportements entre les SMs,
— les cas où la bande passante des communications de données ne limite pas le temps d'exécution du *kernel* et
— le phénomène de conflit sur les banques de données de la *shared memory*.

L'idée serait alors de pouvoir déterminer selon une méthode heuristique l'espace mémoire adapté à chaque espace de données utilisé dans un algorithme. Il serait alors possible d'inclure la sélection des espaces mémoire dans notre méthodologie. Face au nombre de paramètres à prendre en compte, des résultats différents pour chaque architecture GPU et des fonctions de coût considérées (temps d'accès, consommation energétique, ...), il serait intéressant d'étudier les résultats que pourrait apporter un réseau de neuronnes, entraîné à partir de l'extension du modèle de *benchmark* abordé dans cette section.

5.2 Exploitation du parallélisme *coarse grain* sur GPUs Nvidia

Dans la précédente section, nous avons abordé l'analyse des performances des différents espaces mémoire mis à disposition par CUDA. Nous nous intéressons à présent à la capacité et aux performances du GPU pour exploiter le parallélisme à gros grain (*coarse grain*).

En programmation, le parallélisme est applicable :

— aux instructions (ILP),
— aux itérations de boucles,
— aux sections parallèles ou encore
— aux programmes informatiques.

Ce classement, par ordre croissant de granularité, est corrélé à la quantité et à la taille des tâches pouvant être réalisées de manière concurrentielle. La principale difficulté est alors de trouver le bon niveau de segmentation en fonction des capacité de traitement parallèle de l'architecture matérielle ciblée.

Le parallélisme à gros grain se distingue du parallélisme à grain fin (*fine grain*) par la mise en concurrence de tâches :

— plus importantes (souvent assimilées à un programme informatique) et
— dont le temps d'exécution est irrégulier.

L'ensemble des solutions de transformation automatique de code, listées dans la section 2.2, exploitent principalement, pour le placement sur GPU, le parallélisme à grain fin. Chaque itération d'un nid de boucles est alors transformé en un *thread*, exécuté par un *CUDA core*. Cette méthode est particulièrement adaptée à la plupart des applications de traitement d'images, présentant des *patterns* algorithmiques réguliers sur des ensembles importants de données.

Cependant, il arrive, même en traitement d'images, que les portions parallèles de code présentent :

— des quantités irrégulières d'opérations à effectuer et/ou
— un faible nombre d'instances parallèles.

Le parallélisme à gros grain est alors adapté à ces caractéristiques.

Dans le cas de l'algorithme *simpleFlow* (section 4.2.1), les fonctions *selectPointsToRecalcFlow* et *extrapolateFlow* sont deux exemples où :

- les boucles externes du nid sont séquentielles et
- les boucles internes :
  - sont parallèles,
  - présentent un faible nombre d'itérations et
  - ont des bornes dynamiques.

Dans ces deux cas, le placement des boucles internes sur GPU ne permet pas de saturer les capacités calculatoires de cette architecture massivement parallèle. De plus, leur maintien sur le processeur hôte nécessite de mettre en place des transferts mémoire entre l'hôte et l'accélérateur afin de maintenir la cohérence des données partagées. Chaque *kernel* constituant pour le GPU un programme exécutable, nous envisageons, dans ce cas de figure, l'exploitation du parallélisme *coarse grain*. Nous étudions ainsi, dans cette section, la capacité ainsi que les performances du GPU pour exploiter cette forme de parallélisme.

À ce sujet, Amini [16] a évalué deux algorithmes de détection de parallélisme pour le placement de code sur GPU. Le premier, basé sur l'algorithme de parallélisation d'Allen et Kennedy [43], maximise la distribution des boucles selon un parallélisme à granularité fine et est particulièrement adapté aux architectures vectorielles. Le second [79], utilisant les analyses de régions de tableau [37], permet une parallélisation de type *coarse grain* sur les données. Amini a conclu que l'algorithme d'Allen et Kennedy présentait des performances supérieures avec un temps d'exécution quatre fois plus faible sur architecture *Kepler*. Cependant, contrairement à notre évaluation, l'approche *coarse grain* a été utilisée par Amini :

- pour du parallélisme de donnée,
- pour un nid présentant un nombre élevé d'itérations indépendantes (les boucles parallèles i et j présentent un total de $\sum_{i=1}^{2999} \sum_{j=i}^{2999} j = 4\ 498\ 500$ itérations) et
- pour un pattern de calcul parfaitement régulier.

Nous détaillons brièvement, dans la section 5.2.1, l'état de l'art ainsi que les caractéristiques des techniques des GPUs permettant l'exécution d'opérations concurrentes. Nous décrivons ensuite, dans la section 5.2.2, l'objectif de cette expérimentation. Le protocole expérimental est défini dans la section 5.2.3. Enfin, les résultats sont analysés et interprétés dans la section 5.2.4.

5.2.1 Description du parallélisme *coarse grain* pour les GPUs

Nous abordons, dans cette expérimentation, le parallélisme de tâches. Nous étudions ainsi, selon la taxonomie de Flynn, l'aspect MIMD des GPUs.

À partir de la génération *Kepler*, les GPUs Nvidia ont la capacité d'exploiter de manière concurrentielle plusieurs *pipelines* d'instructions, alimentés par le processeur hôte. Ces *pipelines*, portant le nom de *CUDA stream*, permettent d'exploiter le parallélisme *coarse grain*, au niveau des *kernels*, sur GPU. Ils sont notamment utilisés pour :

- la concurrence entre le processeur hôte (CPU) et l'accélérateur GPU,
- la concurrence entre accélérateurs GPU,
- les transferts mémoire asynchrones entre l'hôte et l'accélérateur et
- la concurrence entre *kernels* dans un même GPU.

Les instructions *CUDA event* fournissent alors les informations permettant la synchronisation entre *CUDA streams*. Par défaut, un unique *CUDA stream*, d'identifiant 0, est utilisé. L'exploitation de *CUDA streams* supplémentaires requière une déclaration explicite de la part du développeur.

Nous abordons à présent chacune des approches permettant l'exploitation du parallélisme *coarse grain* sur GPU.

Concurrence CPU/GPU

Cette forme de concurrence exploite un unique *CUDA stream*. Elle repose sur le fonctionnement nativement asynchrone du GPU pour l'exécution des *kernels*. Le processeur hôte empile ainsi, dans le *CUDA stream* choisi, les ordres d'exécution de *kernel* sans attendre de retour. Son exploitation ne requière donc aucune action de la part du développeur. Cette solution permet :

— de masquer l'*overhead* lié au lancement des *kernels* et
— d'exécuter des tâches concurrentes sur le processeur hôte.

Les communications synchrones de données sont souvent utilisées pour assurer la synchronisation entre le processeur hôte et l'accélérateur.

La principale problématique réside dans le bon équilibrage des charges entre l'hôte et l'accélérateur. À ce sujet, un état de l'art assez complet a été réalisé par Mittal et Vetter [104].

Concurrence inter-GPUs

Dans le cadre du parallélisme de tâches, la concurrence inter-GPUs permet de distribuer l'exécution de *kernels* indépendants sur différents GPUs. Le processeur hôte communique alors avec chaque GPU au moyen d'un *CUDA stream* dédié et leur synchronisation est réalisée au moyen d'instructions *CUDA event*. La sélection d'un GPU est effectué au moyen de la fonction *cudaSetDevice*. Chaque GPU constitue un nœud de type Non Uniform Memory Access (NUMA) dans l'architecture globale.

Les problématiques liées à l'usage de plusieurs GPUs sont :

— la répartition des données sur les GPUs,
— la minimisation des données échangées et
— la minimisation des dépendances entre les *kernels* distribués.

Dans l'état de l'art, StarPU [21] et R-Stream [94] sont capables de distribuer les *kernels* et leurs données associées sur plusieurs GPU.

Concurrence des tranferts de données

En complément de la concurrence portant sur l'exécution de *kernels*, il est possible d'effectuer les transferts de données de manière asynchrone au moyen d'un *CUDA stream* dédié. Cette forme de concurrence permet de masquer le temps de transfert des données en exécutant sur le GPU un ou plusieurs *kernels* concurrents. Le transfert est alors assuré par un *copy engine*, capable d'effectuer des requêtes Direct Memory Access (DMA) au travers du bus PCIe. De ce fait, les données dans l'espace mémoire du processeur hôte doivent être déclarées comme *page locked memory* (appelée *pinned memory* par CUDA). Enfin, l'achèvement d'un transfert est signalé au moyen d'instructions *CUDA event*.

Concurrence intra-GPU

Pour ce dernier cas, plusieurs *kernels* sont exécutés en concurrence dans un unique GPU. À la différence de la concurrence inter-GPUs, les données résident dans la mémoire globale du GPU, qui est unifiée selon un modèle d'architecture SMP. Ce sujet a été abordé par Guevara *et al.* [69]. Cependant leur approche est antérieure à l'existence des *CUDA streams*. Plus récemment Cruz *et al.* [39] ont aussi abordés ce sujet.

Nous retrouvons en complément le parallélisme dynamique chez Nvidia. Celui-ci permet à une instance de *kernel* d'ordonner l'exécution d'un nouvel ensemble d'instances de *kernel*.

5.2.2 Description du sujet d'expérience

Nous nous intéressons à la concurence de *kernels* intra-GPU afin de trouver une solution aux problématiques de granularité irrégulière et de sous-utilisation des ressources calculatoires du GPU. L'enjeux est alors de réduire le taux d'activité du CPU au profit du GPU, ce qui a pour conséquence de limiter les transferts de données entre CPU et GPU.

L'objectif de cette expérimentation est double :

1. Nous cherchons à définir la quantité minimale d'instances de *threads* pour surcharger les unités SM.
2. Nous souhaitons étudier le comportement d'un GPU pour un placement parallèle de type *coarse grain*.

L'architecture des GPUs, décrite dans le chapitre 1, est basée sur un modèle hiérarchique à deux niveaux, composé de processeurs SM intégrant des Cuda *cores*. Lors de l'exécution d'un *kernel* :

— la grille des instances de *threads* est décomposée en *blocs*[5], distribués par le GPU sur les différents SMs,
— ces *blocs* sont eux-même décomposés en *warps* de 32 *threads* et placés sur les *CUDA cores* par les *warp schedulers* et les *dispatch units*.

Selon ce modèle architectural, les GPUs sont couramment cités pour leur pertinence à exploiter le parallélisme de données selon une granularité fine. L'usage courant est de sur-alimenter en *blocs* les *sm*s et en *threads* les *CUDA core*s, afin d'optimiser la bande passante des *pipelines* d'intructions. Cette sur-alimentation permet notamment de réduire le phénomène de "bulles" dans les *pipelines* selon le principe du Simultaneous multithreading (SMT). Les chiffres de 64, 128 ou encore 256 *threads* par *block* sont souvent avancés, sur les forums spécialisés, comme étant le point d'équilibre entre sous-alimentation et sur-alimentation des unités SM. Nous cherchons à déterminer la valeur exacte pour une architecture GPU donnée.

Le problème de sous-alimentation se pose lorsque le nombre défini de *blocs* et de *threads* est inférieur respectivement au nombre de SMs ou de *CUDA core*s disponibles. Ce type d'utilisation est moins performant, comparé à une exécution sur CPU, du fait de la frèquence de fonctionnement plus faible, du cache d'instructions réduit et du modèle simplifié d'exécution des instructions sur GPU. Nous souhaitons ainsi étudier, dans ce cas de figure, le comportement d'une architecture GPU selectionée, pour une exécution concurrentielle de *kernels*.

5.2.3 Protocole expérimental

Cette expérimentation a été exclusivement réalisée avec le GPU *Quadro K2000* de la plateforme *Endicott*. L'ensemble des caractéristiques de cette architecture *Kepler* sont détaillées dans la section 4.1. Selon le tableau 4.2 et la documentation CUDA [123], cette architecture GPU permet d'exécuter de manière simultanée jusque 16 *kernels* concurrents.

Notre objectif étant d'évaluer la concurrence des *kernels*, les durées présentées dans cette expérimentation correspondent uniquement au temps d'exécution des *kernels*. Les durées propres aux communications de données entre l'hôte et l'accélérateur ne sont pas prises en compte.

5. selon le paramètrage défini lors de l'appel au *kernel* par le développeur

Pour cette expérimentation, nous avons utilisé le modèle de *kernel* spécifié dans le listing 5.1. Nous avons distribué celui-ci sur 32 *CUDA streams*. Ce *kernel* effectue un traitement encapsulé dans une boucle *do/while*. Pour chaque itération de cette boucle, son entête exécute une nouvelle itération jusqu'à ce que le temps d'exécution de l'instance du *kernel* dépasse la valeur de la variable *clock_count*. Pour cette expérimentation nous avons fixé cette variable à $10\,ms$. Nous noterons que la durée d'exécution d'une instance de *kernel* ne pourra faire précisémment $10\,ms$. Nous considérons, cependant, cette différence négligeable.

Nous lançons l'exécution d'un *kernel* sur chacun des 32 *CUDA streams* et étudions la durée d'exécution globale pour l'ensemble de ces instances de *kernels*. Cette opération est réalisée pour plusieurs configurations de test, en faisant varier le nombre d'instances de *threads* et de *blocs*. Pour chaque configuration de test, le même paramètrage (*blocks/threads*) est utilisé pour l'ensemble des *kernels*. En conséquence de la combinatoire importante de paramètres (*blocks/threads*) à évaluer, l'expérimentation a été entièrement automatisée par l'utilisation d'instructions *cudaDeviceSynchronize* entre chaque configuration de test.

Les résultats ont été récupérés au moyen de *nvprof*, le *profiler* de Nvidia.

```
83 __global__ void concurrencyKernel(uint1* output, const unsigned int length, const clock_t clock_count){
84     const unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
85 
86     if(id<length){
87         const clock_t start_clock = clock();
88 
89         clock_t clock_offset;
90         do{
91             //Specialize kernel here with a distinct computation
92             //...
93             clock_offset = clock() - start_clock;
94         }while (clock_offset < clock_count);
95 
96         output[id] = (uint1) clock_offset;
97     }
```

Listing 5.1 – Modèle de *kernel* utilisé pour l'évaluation du parallélisme *coarse grain* sur GPU

5.2.4 Analyse et interprétation des résultats

L'ensemble des résultats obtenus sont visibles dans la figure 5.9. Ces données sont représentées en utilisant pour l'axe des *threads* :

— une échelle linéaire dans la figure 5.9a et

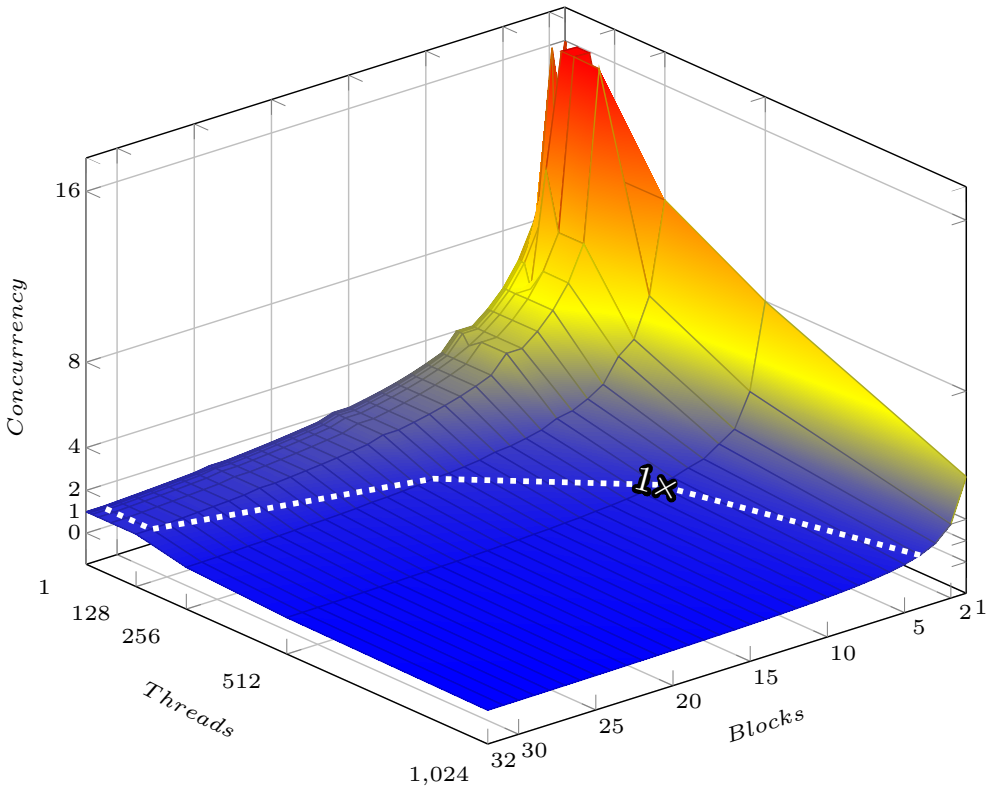
— une échelle logarithmique dans la figure 5.9b.

Pour représenter les résultats de cette expérimentation, nous avons utilisé la métrique *concurrency*, définie dans la formule 5.8.

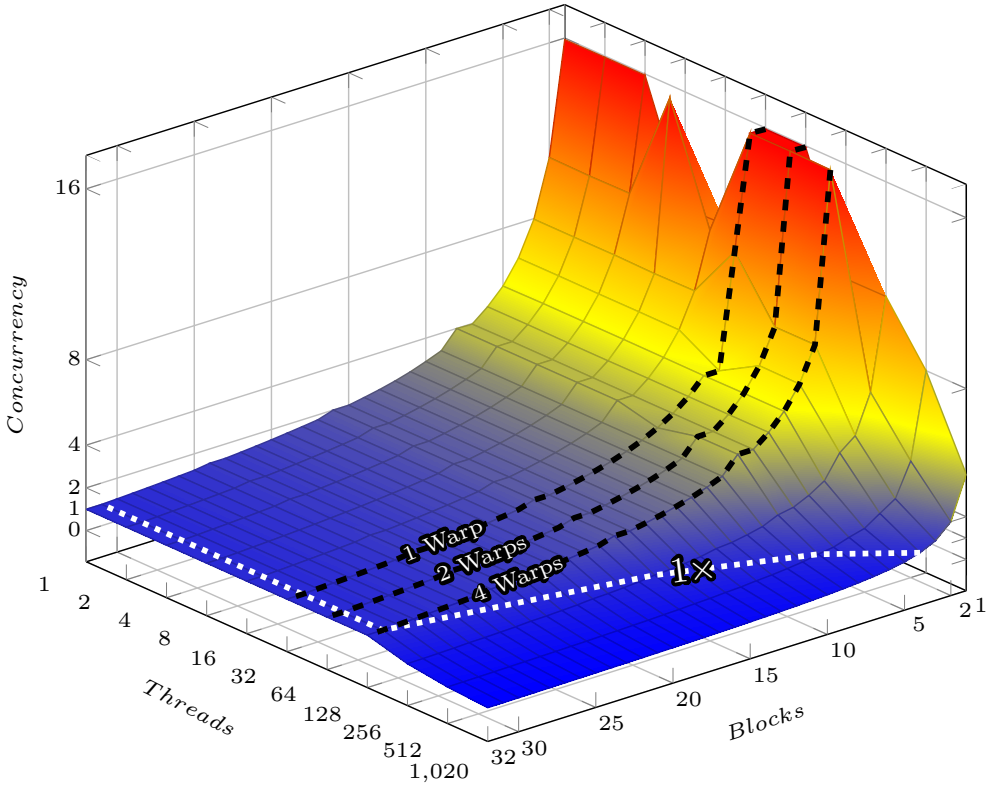
$$concurrency = \frac{\Delta t_{theor.}}{t_{max} - t_{min}} \tag{5.8}$$

Dans cette formule, pour chaque configuration de test (*blocks/threads*) :

— t_{max} correspond à la date la plus récente de fin d'exécution de *kernel*,



(a) Échelle de *threads* linéaire



(b) Échelle de *threads* logarithmique

FIGURE 5.9 – Analyse de la concurrence de kernels intra-GPU sur architecture Nvidia Kepler

— t_{min} à la date la plus ancienne de début d'exécution de *kernel* et
— $\Delta t_{theor.}$ aux temps global théorique donné par la formule 5.9. Pour ce dernier :
  — $nb_{kernels}$ correspond au nombre de *kernels* utilisés par configuration de test.
  — $t_{theor.}$ correspond au temps d'exécution théorique d'une instance de *kernel*.

$$\Delta t_{theor.} = nb_{kernels} \times t_{theor.} = 32 \times 10\,ms = 320\,ms \tag{5.9}$$

La métrique *concurrency* nous donne donc le rapport entre :
— le temps d'exécution théorique global des 32 *kernels*, pour une unique instance par *kernel*
— et le temps d'exécution global mesuré.

Nous considérons ainsi les cas :

concurrency > 1 : lorsque le GPU est sous-utilisé par un unique *kernel*. Dans le cadre d'un placement de type *coarse grain*, les unités de calcul inutilisées sont alors exploitées pour exécuter de manière concurrentielle les *kernels* des différents *CUDA streams*. La valeur de *concurrency* correspond dans ce cas au nombre moyen de *kernels* exécutés simultanément sur le GPU.

concurrency < 1 : lorsque la quantité d'instances de *threads* à traiter pour un même *kernel* vient surcharger la capacité de traitement des unités SM. Le temps d'exécution de ce *kernel* s'allonge et devient alors supérieur à $10\,ms$. L'exécution concurrente de *kernels* n'est alors plus adaptée et le placement converge vers l'exploitation d'un parallélisme à grain fin. La surcharge des SMs a pour bénéfice de masquer les bulles au sein des *pipelines* d'instructions selon un principe similaire à l'*hyperthreading* sur les CPUs *Intel*.

concurrency $= 1$: lorsque la capacité de traitement des unités SM est tout juste atteinte. Ce cas correspond au point d'équilibre entre les deux cas précédents.

En fonction des configurations de test (*blocks*/*threads*), le détail des données acquises, correspondant à la métrique *concurrency*, est fourni dans la table 5.1. Le code couleur utilisé dans cette table correspond à :

rouge : la zone de données où *concurrency* > 1.0.

vert : la zone de données où *concurrency* < 1.0.

bleu : la zone de données où *concurrency* ≈ 1.0.

Lors de leur représentation, l'intensité de chacune de ces couleurs est corrélée respectivement à la quantité de *kernels* exécutés en concurrence (rouge), à la saturation du GPU (vert) et à la proximité du point d'équilibre (bleu).

En complément, nous avons ajouté deux représentations graphiques de ces données (figure 5.9) afin d'améliorer la visualisation et l'interprétation du phénomène de concurrence sur GPU. Nous étudions à présent la surcharge des unités SMX et l'exécution concurrentielle de *kernels*.

Saturation des unités SMX

Nous avons tracé sur les figures 5.9a et 5.9b l'évolution du point d'équilibre (*concurrency* $= 1.0$) en fonction du nombre de *blocks* et de *threads* utilisés. Les valeurs intermédaires ont été extrapolées en utilisant une interpolation linéaire. La courbe résultante, représentée par des tirets blancs, est identifiable par la mention "1×". Cette courbe délimite, entre autres, la zone de données correspondant à la saturation des unités SMX (*concurrency* < 1.0).

concurrency		*blocks*															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
threads (warps)	1 (1)	15.93	10.63	7.97	6.38	5.30	4.56	3.97	3.99	3.18	2.90	2.65	2.45	2.28	2.13	2.00	2.00
	2 (1)	15.94	10.61	7.97	6.39	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.45	2.28	2.13	1.99	2.00
	4 (1)	15.94	10.63	7.98	6.38	5.32	4.55	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	1.99
	8 (1)	10.55	15.94	7.95	6.39	5.32	4.55	3.99	3.99	3.19	2.90	2.66	2.45	2.28	2.13	1.99	2.00
	16 (1)	10.63	10.58	7.98	6.38	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	32 (1)	15.94	15.98	10.65	5.31	5.31	4.56	3.99	3.55	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	64 (2)	15.94	15.98	7.97	6.39	5.32	4.56	3.99	3.98	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	128 (4)	10.62	15.94	7.98	6.37	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	1.99
	256 (8)	10.63	6.37	4.56	3.99	2.91	2.46	2.13	2.00	1.68	1.52	1.39	1.33	1.18	1.10	1.03	1.00
	512 (16)	7.99	3.99	2.46	2.00	1.52	1.33	1.10	1.00	0.86	0.78	0.71	0.67	0.60	0.57	0.52	0.50
	1024 (32)	3.99	2.00	1.33	1.00	0.80	0.67	0.57	0.50	0.44	0.40	0.36	0.33	0.31	0.29	0.27	0.25

		blocks															
		17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
threads (warps)	1 (1)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	2 (1)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	4 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	8 (1)	1.78	1.68	1.59	1.52	1.45	1.39	1.33	1.28	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	16 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	32 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	64 (2)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	128 (4)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	256 (8)	0.91	0.86	0.82	0.80	0.74	0.71	0.68	0.67	0.63	0.60	0.58	0.57	0.54	0.52	0.51	0.50
	512 (16)	0.46	0.44	0.42	0.40	0.38	0.36	0.34	0.33	0.32	0.31	0.29	0.29	0.27	0.27	0.26	0.25
	1024 (32)	0.24	0.22	0.21	0.20	0.19	0.18	0.17	0.17	0.16	0.15	0.15	0.14	0.14	0.13	0.13	0.12

TABLE 5.1 – Résultats de l'expérimentation sur la concurrence de *threads*

La figure 5.9b met en évidence, pour ce seuil de saturation, un phénomène de rupture se produisant à partir de 4 *warps* (128 *threads*). Il s'en dégage deux tendances distinctes dans la courbe :

1. une première partie évoluant de manière constante jusque 4 *warps* puis
2. au-delà de 128 *threads*, une seconde partie dont l'évolution est fonction du nombre de *blocks* utilisés.

L'évolution globale du seuil de saturation correspond alors à la formule 5.10 où :
— **warps** correspond au découpage en groupe de *threads* de chaque *block* selon la formule 5.11 et
— **blocks** correspond au nombre de *blocks* utilisés.

$$\begin{cases} warps \times blocks = 128 & warps \geq 4 \\ blocks = 32 & warps < 4 \end{cases} \tag{5.10}$$

$$warps = \lceil \frac{threads}{32} \rceil \ tq \ threads \in [1, 1024] \tag{5.11}$$

Nous en déduisons ainsi, que :
— les SMX sont saturés à partir de 128 *threads* par *blocks* et
— les SMX sont sous-exploités en dessous de 128 *threads* par *blocks*.

Afin de maximiser le taux d'occupation des SMX, nous avons ainsi défini dans le second critère de placement, en section 3.3.2, le paramètre d'optimisation :

$$\prod_{p=0}^{T} |I_{j+p}| \geqslant 4 \times 32$$

Exécution concurrente de *kernels*

L'architecture des unités SMX de la *Quadro K2000* intègre quatre *warp schedulers*, visibles dans la figure 4.1. Ces derniers sont ainsi capables de gérer quatre *warps* en concurrence. Chaque *warp scheduler* est associé à deux *intruction dispatch units* permettant ainsi d'exécuter en concurrence deux instructions indépendantes pour un même *warp*. Chaque SMX a ainsi la capacité d'exécuter jusque 8 instructions distinctes de manière simultanée. La *Quadro K2000* intégrant deux SMX, le total est ainsi porté à 16 instructions distinctes sur cette plateforme.

Dans la figure 5.9, le parallélisme de tâches sur GPU correspond à la zone de données où $concurrency > 1.0$ et dont la couleur de représentation évolue du bleu (faible concurrence) vers le rouge (forte concurrence). Pour cette zone, nous pouvons observer que la valeur de *concurrency* évolue en fonction du nombre de *blocks* et de *threads* affectés à chaque *kernel*. Lorsque l'une de ces deux dimensions augmente, la *concurrency* diminue, ce qui implique que les *kernels* peuvent être exécutés en concurrence non seulement entre les SMs mais aussi entre les *warp schedulers*. Enfin, les résultats de *concurrency* sont plafonnés à 16 *kernels* exécutés simultanément ce qui correspond aux spécifications du *compute capability 3.0* de Nvidia. Ce maximum est cependant atteint de manière irrégulière. L'alimentation de 32 *CUDA streams* nous apparait délicate dans cette expérimentation du fait que :
— les *CUDA streams* soient alimentés à partir d'un unique *thread* sur le CPU et
— le temps d'exécution de chaque *kernel*, lorsque la valeur de *concurrency* est supérieure à 1.0, reste assez court ($10ms$).

Le temps d'exécution des *kernels* et la puissance du CPU sont donc deux critères a prendre en considération pour l'exploitation du parallélisme de tâches sur GPU.

Le lien entre les caractéristiques de la *Quadro K2000* et les résultats obtenus ne sont pas évidents. En prenant, le cas où chaque *kernel* est exécuté pour 2 *blocks* de 128 *threads* (soit 4 *warps*) chacun, nous obtenons une *concurrency* de 16.0, dans le tableau 5.1. Nous avons donc 16 *kernels* qui s'exécutent en concurrence, ce qui correspond à un total de $16 \times 128 \times 2 = 4096$ *threads* soit 128 *warps* exécutés simultanément sur cette architecture. Le rapport est alors de

— 16 en comparaison avec les 8 *warp schedulers* disponibles et de
— $10,7$ en comparaison avec les 384 *CUDA cores* disponibles.

Ce rapport non unitaire s'explique de deux façons :

1. l'ILP employé au sein des SMs [161, 162] permet de :
   (a) masquer le temps de latence des instructions entre les exécutions des anglwarps d'un même *block*,
   (b) exécuter simultanément deux instructions indépendantes, pour un même *warp*, au moyen des deux *instruction dispatch units* de chaque *warp scheduler*,
2. la borne de la boucle utilisée à l'intérieur de chaque *kernel* dépend du temps d'exécution de ce dernier et non d'un nombre d'itération fixe.

Pour le premier point, l'ILP permet de paralléliser l'exécution des *warps* sur un même *warp scheduler*. Ainsi, pour le second point, la boucle interne au *kernel* itère jusqu'à ce que la durée fixée soit atteinte pour l'ensemble des *warps* d'un *block*.

5.2.5 Conclusion sur l'exploitation du parallélisme de tâches

Nous avons étudié dans cette section la capacité du GPU a gérer le parallélisme de tâches. Dans certaines conditions, le GPU peut ainsi présenter les caractéristiques d'une architecture de type MIMD, adaptée au parallélisme *coarse grain*. Les résultats obtenus concerne la *Quadro K2000* de génération *Kepler* qui dispose de deux unités SMX.

De cette expérimentation, nous en avons dégagé quatre principaux constats :

— les unités SMX sont surchargées à partir de 4 *warps*,
— au maximum 16 *kernels* concurrents peuvent être exécutés simultanéments,
— la concurrence entre *threads* repose sur le fonctionnement des *warp schedulers* et
— l'ILP permet d'augmenter le nombre de *warps* traités simultanément.

En complément, il serait intéressant de comparer ces résultats avec le comportement de GPUs intégrant plus d'unités SMX. Ces analyses nous permettent de mieux appréhender la complexité du fonctionnement des architectures GPUs. Nous espérons ainsi, dans le futur, pouvoir ajouter l'exploitation du parallélisme *coarse grain* dans notre méthodologie.

5.3 Conclusion sur les expériences

Nous avons abordé dans ce chapitre deux aspects de l'optimisation du placement sur GPU, dans le but de réduire le temps d'exécution des *kernels*. L'architecture des GPUs étant complexe, leur optimisation est d'une part complexe et d'autre part concerne de nombreux paramètres. Nous avons ainsi privilégié, dans nos travaux, l'étude des espaces mémoires et de la concurrence entre *kernels*. Nous considérons ces deux sujets particulièrement sensibles pour les applications de traitement d'image. Celles-ci présentent, en règle général, une forte quantité de données à :

1. acheminer depuis la mémoire principale de l'accélérateur,
2. traiter par les *CUDA cores* puis

3. stocker sur la mémoire principalei de l'accélérateur.

En conséquence, les accès mémoire constituent souvent le chemin critique pour le temps d'exécution global des *kernels*, notamment pour l'exploitation du parallélisme de type *fine grain* sur GPU.

Cependant, certaines applications de traitement d'images exploitent leurs données en parallèle selon des flots :

— de faible quantité,
— de quantité irrégulière,
— de quantité non prédictible et
— dont le temps de taitement est, en conséquence, disparate.

Ce type d'application répondent alors aux caractéristiques du parallélisme *coarse grain*.

Au sujet de l'**étude des espaces mémoires**, nous avons étudié leur comportement respectif, sur deux plateformes de génération différente (*Kepler* et *Maxwell*). Pour déterminer les performances des temps d'accès en lecture de chacun d'entre-eux, nous avons considéré en particulier la variation de trois paramètres :

— la redondance des accès dans chaque *block*,
— la distribution des accès dans chaque *block* et
— l'écart spatial entre les accès (le *stride*).

Il ressort de ces premières mesures que la *constant memory* présente unanimement le meilleur temps d'accès lorsque le nombre d'accès commun entre les *threads* d'un même *warp* est important. Au-delà de 8 accès communs pour la *Quadro K2000* et de 16 accès communs pour la *TX1*, la *texture memory* pour la première et la *shared memory* pour la seconde présente une meilleure performance. Ces deux points de transition sont cependant influencés par le *stride* existant entre les différents accès. Celui-ci vient en effet augmenter la pression exercé sur les différentes mémoires cache. Enfin, nous avons relevé que l'architecture de type embarquée de la *TX1* sacrifie les performances de ces différents espaces mémoire au profit d'une consomation énergétique réduite. Notamment, la variance des temps d'accès augmente sensiblement, ce qui nuit globalement à la reproductibilité des résultats. Seule la *shared memory* est épargnée par ce point. Le placement des données sur cet espace mémoire est donc recommandable pour cette architecture GPU.

Enfin, pour l'étude portant sur la **concurrence des *threads***, notre objectif est de déterminer les critères permettant d'exploiter le parallélisme *coarse grain* sur GPU. Nous avons ainsi pu vérifier que le GPU était capable d'exécuter simultanément jusque 16 *kernels* distincts. En complément, nous avons pu remarquer que les ressources calculatoires du GPU étaient saturées à partir de 4 *warps* par *blocks*.

Ces expériences ont été réalisées sur des algorithmes de test spécifiques. Notre objectif, à présent, est de généraliser les résultats de ces travaux afin de les intégrer dans notre méthodologie.

Conclusion

Depuis l'architecture vectorielle dédiée au domaine du rendu graphique, le GPU a fortement évolué, en passant notamment par un point clé de son existence, au tout début des années 2010 : le calcul généraliste, engendré par l'avènement du GPGPU. Le succès rapide qu'a connu ce dernier vient incontestablement du haut niveau de performance calculatoire (HPC) que les GPUs peuvent atteindre grâce à l'emploi massif du parallélisme. Or, à la même époque, l'évolution des performances des CPUs, suivant la célèbre loi de Moore, était stoppée nette par le phénomène du mur des fréquences.

Cependant, à l'origine, la consommation énergétique était un facteur secondaire dans la spécification des architectures GPU. La bande passante calculatoire retenait alors toute l'attention avec des limites repoussées continuellement vers de nouvelles frontières. De nos jours, le GPU est clairement considéré comme la solution la plus efficace pour porter le marché vidéo-ludique. De même, on le retrouve souvent employé dans les super-calculateurs du *Top500* et son utilisation courante pour des sujets en plein essort tels que les crypto-monnaies ou encore les réseaux de neurones en font une architecture très populaire.

Face à cette masse de puissance calculatoire, l'architecture pourtant déjà assez complexe des GPUs a dû prendre en compte les dissipations thermiques puis son efficience énergétique. Actuellement, un des plus gros enjeux pour le GPU porte notamment sur son développement dans le domaine de l'HPC embarqué permettant ainsi une faible consommation énergétique. Or, de nouveaux acteurs tel que le Multi-Purpose Processor Array (MPPA) de Kalray aborde aussi cette problématique. Si leur solution actuelle, *MPPA2 Bostan*, est basée sur le principe d'une carte accélératrice, l'entreprise prévoit aussi pour la prochaine génération *MPPA3 Coolidge* de rendre pleinement autonome leur processeur. Pour sa part, AMD avec son architecture APU semble se maintenir à l'écart de cette course dans le cadre des applications industrielles. Intel, qui avait évoqué une version embarquée de son Xeon Phi, paraît cependant être assez peu actif sur ce sujet pour le moment. Nvidia de son côté considère l'embarqué grand public au moyen de son architecture Tegra. De plus, celle-ci est aussi déclinée pour les applications industrielles de l'automotive avec son architecture Drive. Leur solution prend une forme nouvelle, tournée vers l'autonomie. Au sein d'un unique SOC sont intégrés un GPU et un CPU basse consommation. Ce dernier a la charge d'exécuter l'OS et de piloter la partie GPU. De plus, la mémoire globale est communalisée entre les deux composants, offrant ainsi une alternative à la bande passante limitative du bus PCIe.

Comme pour toute architecture à faible consommation énergétique, la réduction de l'empreinte énergétique des GPUs a des conséquences sur leurs performances. Il est de ce fait indispensable de les prendre en considération, au moyen de phases d'optimisation et de spécialisation, lors du processus de portage d'algorithmes sur GPU. L'objectif est ici d'atteindre un niveau d'efficacité le plus élevé possible vis-à-vis des capacités de l'architecture.

Synthèse des contributions

Dans le cadre de cette thèse, nous abordons la problématique du portage sur GPU d'algorithmes séquentiels dont la structure de code C++ est de complexité importante. Pour cela, nous avons développé une méthodologie [62, 65] complète permettant de définir le placement d'un algorithme séquentiel sur architecture hétérogène, composée d'un processeur hôte, le CPU, ainsi que d'un accélérateur, le GPU. À ce sujet, nous avons étudié l'état de l'art du placement sur GPU dans le *chapitre 2*. Notre analyse fait état de quatre grands courants qui sont :

— les annotations ou directives de transformation dans le code,
— la transformation automatique de code au moyen de compilateurs source à source,
— la programmation par squelettes algorithmiques et
— la programmation par DSLs.

Nous avons listé les avantages et les défauts de chaque solution et avons décidé de fonder notre approche sur celle des compilateurs source à source en exploitant des processus d'analyses et de transformations de code automatisées. Notre méthodologie est alors décomposée en quatre principales étapes décrites dans ce manuscrit :

1. les analyses statiques et dynamiques,
2. l'identification de code,
3. les transformations de code et
4. la génération de code.

Nous avons ainsi défini dans la première partie du *chapitre 3* un ensemble d'analyses statiques et dynamiques permettant d'extraire les caractéristiques du code source étudié. Ces analyses sont basées sur une approche interprocédurale afin de considérer une application dans sa globalité. À partir de ces données, nous avons conçu une représentation graphique originale que nous avons nommé représentation spinale. Celle-ci fusionne les caractéristiques du graphe d'appels, du graphe de flots de données ainsi que du graphe de dépendances. Nous considérons, au travers de notre représentation spinale, la problématique de la juste quantité de données représentées selon une disposition la plus ergonomique possible. Nous apportons une solution dont l'objectif est d'atteindre un équilibre entre :

— une vue trop synthétique ne permettant pas de prendre des décisions bas niveau,
— une vue trop complexe affichant à l'opposé un volume d'informations trop élevé pour être concrètement exploitable
— et la fragmentation des données sur de multiples vues, complexifiant alors le traitement global des informations.

Nous considérons ainsi dans notre approche, non seulement la quantité mais aussi la qualité des informations représentées.

Nous avons ensuite formalisé un ensemble de critères permettant d'identifier les portions de code adaptées aux architectures GPUs. Cependant, afin de raffiner ces solutions de placements identifiés, nous avons aussi défini un ensemble de transformations de code ayant un impact sur chacun des critères de placement. Il en résulte alors une augmentation de la quantité et de la taille des portions de code plaçables sur cette architecture.

Enfin, nous avons déterminé dans la dernière partie du *chapitre 3* un ensemble minimal de spécialisations indispensables pour une mise en œuvre des GPUs. Cette spécialisation est définie pour les architectures Nvidia au travers de Cuda. Au cours de cette étape, nous avons aussi mis en place, à partir des résultats de l'analyse de code dynamique, un mécanisme permettant d'identifier les portions de code dont les performances sur GPU sont dégradées comparées à leur version originale. Les placements concernés sont alors ignorés pour être remplacés par le code d'origine sur CPU.

Nous avons évalué cette méthodologie, dans le chapitre 4, à la fois sur une architecture GPU classique mais aussi sur l'architecture embarquée basse consommation *Tegra X1*. De plus, ces expérimentations ont permis d'améliorer les performances en temps d'exécution de l'algorithme de flot optique *simpleflow* écrit en *C++* et présent dans les contributions OpenCV. De même, nos travaux ont abouti à la définition d'un algorithme de calcul de variance sur voisinage optimisé pour GPU, l'algorithme *Threewise* [63, 64]. Celui-ci est notamment utilisé en traitement d'images mais aussi dans d'autres domaines exploitant l'analyse de données tels que le calcul financier, les calculs de simulation ou encore les systèmes de surveillance d'anomalies.

Enfin, dans le chapitre 5, nous avons étudié, pour les GPUs Nvidia :

— le comportement des différents espaces mémoire et

— l'exploitation du placement à grain grossier.

La première étude nous a permis de déterminer différents critères influençant le temps de communication des données en fonction de l'espace mémoire choisi. Dans la seconde étude, nous avons observé différents cas où les architectures GPU Nvidia présentaient un comportement de type MIMD, adapté notamment aux algorithmes présentant un faible niveau de parallélisme et au parallélisme de tâches. Ces deux études constituent une étape préalable à l'ajout d'optimisations fines, visant à maximiser l'exploitation des capacités des GPUs.

Perspectives

Afin d'améliorer encore l'efficacité de notre méthodologie, plusieurs points mériteraient d'être développés davantage.

Automatisation et compilateurs

En premier vient l'intégration de notre méthodologie dans un environnement de programmation. Le fait de se passer d'une intervention humaine permettrait d'accélérer le temps pris pour le processus de portage sur GPU. Cependant, dans le cadre de notre démarche d'optimisation et de spécialisation, un nombre important de paramètres interdépendants sont à prendre en compte. L'utilisation de transformations au sein d'un modèle de représentation polyédrique est une solution couramment exploitée pour réaliser ce genre de tâche. On le retrouve notamment employé dans plusieurs solutions telles que PPCG, C-to-Cuda ou encore R-Stream que nous avons présentées dans la section 2.2. L'utilisation de ce modèle comportant plusieurs dimensions issues des itérations de boucle ou encore des accès mémoire permet théoriquement de converger vers une solution prenant en considération les spécificités architecturales de la cible, ce qui répond à notre besoin. Ainsi, le travail à fournir ici consisterait à modéliser les transformations et les optimisations de code utilisées dans ce manuscrit. Cependant, le modèle polyédrique souffre de limitations connues comme les contraintes de transformations réduites à un modèle affine, l'utilisation d'espaces d'itération convexes ou encore l'application à des portions de code à contrôle statique. C'est pour ces raisons, que nous avons défini notre méthodologie sans la spécialiser vis-à-vis des contraintes du modèle polyédrique.

Représentation dynamique

En considérant cette méthodologie comme un outil de portage sur GPU, un autre axe de développement serait de rendre dynamique notre représentation spinale spécifiée, dans la section 3.1.6. La modification par l'utilisateur de cette représentation graphique

se traduirait alors de manière dynamique par les transformations de code adaptées. Les divers critères d'applicabilité impactés pourraient être mis en évidence afin de vérifier la légalité de la transformation envisagée. Par exemple, l'attention serait attirée sur l'effet lié aux dépendances de données lors d'une transformation. De même, lors d'un placement sur de multiples GPUs, les communications mémoire engendrées seraient mises en avant.

Placement sur CPU

Le placement de code sur GPU engendré par notre méthodologie est conditionné par le taux d'accélération constaté. Le mécanisme décrit dans la section 3.7 permet d'invalider un placement dont l'efficacité en terme de temps d'exécution est moins prononcé sur GPU que sur CPU. Les portions de code concernées sont alors maintenues sur CPU. Il serait cependant intéressant d'optimiser le placement sur CPU de l'ensemble des portions de codes n'ayant pas été maintenues sur le GPU. De même en poussant plus loin ce raisonnement, il serait pertinent de définir un ensemble de critères de placement pour le CPU et de les confronter dans le cadre de la méthodologie à ceux que nous avons décrits pour le GPU. Appliquée à un algorithme, la stratégie de placement alors définie serait adaptée à l'architecture hétérogène globale et non à la seule architecture GPU.

Empreinte énergétique

Dans le cadre de nos travaux, nous avons étudié le comportement de notre méthodologie sur deux architectures distinctes décrites dans le chapitre 4.1. L'une est une station de calcul classique intégrant une carte accélératrice GPU *Quadro K2000* de chez Nvidia. L'autre est une unité de calcul embarquée basse consommation intégrant un CPU et un GPU sur un unique SOC, le *Tegra X1*. Nos résultats montrent une certaine scalabilité entre les caractéristiques des diverses architectures GPU et les temps d'exécutions mesurés. Notre méthodologie est ainsi principalement focalisée sur les temps de calculs au moyen de processus d'optimisation et de spécialisation adéquats. Cependant, elle ne considère pas les problématiques de coûts énergétiques. Ainsi il serait intéressant d'intégrer, au sein de notre méthodologie, l'empreinte énergétique des différentes transformations appliquées. Leur emploi serait essentiellement profitable aux applications industrielles embarquées disposant d'une source d'énergie autonome. Cependant, sujet nécessite une instrumentation poussée ainsi que des connaissances en électronique avancées afin d'effectuer des mesures précises et ainsi de générer un modèle de consommation énergétique fiable. Plusieurs travaux [33, 93, 108, 105] sont déjà consacrés à ce sujet.

Annexe A

Code source de l'algorithme *simpleFlow*

```
/*M///////////////////////////////////////////////////////////////////////////////////////
 //
 //  IMPORTANT: READ BEFORE DOWNLOADING, COPYING, INSTALLING OR USING.
 //
 //  By downloading, copying, installing or using the software you agree to this license.
 //  If you do not agree to this license, do not download, install,
 //  copy or use the software.
 //
 //
 //                          License Agreement
 //               For Open Source Computer Vision Library
 //
 // Copyright (C) 2000-2008, Intel Corporation, all rights reserved.
 // Copyright (C) 2009, Willow Garage Inc., all rights reserved.
 // Third party copyrights are property of their respective owners.
 //
 // Redistribution and use in source and binary forms, with or without modification,
 // are permitted provided that the following conditions are met:
 //
 //   * Redistribution's of source code must retain the above copyright notice,
 //     this list of conditions and the following disclaimer.
 //
 //   * Redistribution's in binary form must reproduce the above copyright notice,
 //     this list of conditions and the following disclaimer in the documentation
 //     and/or other materials provided with the distribution.
 //
 //   * The name of the copyright holders may not be used to endorse or promote products
 //     derived from this software without specific prior written permission.
 //
 // This software is provided by the copyright holders and contributors "as is" and
 // any express or implied warranties, including, but not limited to, the implied
 // warranties of merchantability and fitness for a particular purpose are disclaimed.
 // In no event shall the Intel Corporation or contributors be liable for any direct,
 // indirect, incidental, special, exemplary, or consequential damages
 // (including, but not limited to, procurement of substitute goods or services;
 // loss of use, data, or profits; or business interruption) however caused
 // and on any theory of liability, whether in contract, strict liability,
 // or tort (including negligence or otherwise) arising in any way out of
 // the use of this software, even if advised of the possibility of such damage.
 //
```

```
41  //M*/
42
43  #include "simpleFlow.h"
44
45  //
46  // 2D dense optical flow algorithm from the following paper:
47  // Michael Tao, Jiamin Bai, Pushmeet Kohli, and Sylvain Paris.
48  // "SimpleFlow: A Non-iterative, Sublinear Optical Flow Algorithm"
49  // Computer Graphics Forum (Eurographics 2012)
50  // http://graphics.berkeley.edu/papers/Tao-SAN-2012-05/
51  //
52
53  static const uchar MASK_TRUE_VALUE = (uchar) 255;
54
55  inline static float dist(const Vec3b& p1, const Vec3b& p2) {
56          return (float) ((p1[0] - p2[0]) * (p1[0] - p2[0])
57                          + (p1[1] - p2[1]) * (p1[1] - p2[1])
58                          + (p1[2] - p2[2]) * (p1[2] - p2[2]));
59  }
60
61  inline static float dist(const Vec2f& p1, const Vec2f& p2) {
62          return (p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] - p2[1]) * (p1[1] - p2[1]);
63  }
64
65  template<class T>
66  inline static T min(T t1, T t2, T t3) {
67          return (t1 <= t2 && t1 <= t3) ? t1 : min(t2, t3);
68  }
69
70  static void removeOcclusions(const Mat& flow, const Mat& flow_inv,
71                  float occ_thr, Mat& confidence) {
72          const int rows = flow.rows;
73          const int cols = flow.cols;
74          if (!confidence.data) {
75                  confidence = Mat::zeros(rows, cols, CV_32F);
76          }
77          //l22, l24, l105, l107
78          for (int r = 0; r < rows; ++r) {
79                  //l23, l25, l106, l108
80                  for (int c = 0; c < cols; ++c) {
81                          if (dist(flow.at<Vec2f>(r, c), -flow_inv.at<Vec2f>(r, c))
82                                          > occ_thr) {
83                                  confidence.at<float>(r, c) = 0;
84                          } else {
85                                  confidence.at<float>(r, c) = 1;
86                          }
87                  }
88          }
89  }
90
91  static void wd(Mat& d, int top_shift, int bottom_shift, int left_shift,
92                  int right_shift, float sigma) {
93          //l2, l12, l55, l62, l73, l87, l109
94          for (int dr = -top_shift, r = 0; dr <= bottom_shift; ++dr, ++r) {
95                  //l3, l13, l56, l63, l74, l88, l110
96                  for (int dc = -left_shift, c = 0; dc <= right_shift; ++dc, ++c) {
97                          d.at<float>(r, c) = (float) -(dr * dr + dc * dc);
98                  }
99          }
```

```
100         d *= 1.0 / (2.0 * sigma * sigma);
101         exp(d, d);
102 }
103
104 static void wc(const Mat& image, Mat& d, int r0, int c0, int top_shift,
105                 int bottom_shift, int left_shift, int right_shift, float sigma) {
106         const Vec3b centeral_point = image.at<Vec3b>(r0, c0);
107         int left_border = c0 - left_shift, right_border = c0 + right_shift;
108         //l6, l16, l59, l66, l77, l91, l113
109         for (int dr = r0 - top_shift, r = 0; dr <= r0 + bottom_shift; ++dr, ++r) {
110                 const Vec3b *row = image.ptr<Vec3b>(dr);
111                 float *d_row = d.ptr<float>(r);
112                 //l7, l17, l60, l67, l78, l92, l114
113                 for (int dc = left_border, c = 0; dc <= right_border; ++dc, ++c) {
114                         d_row[c] = -dist(centeral_point, row[dc]);
115                 }
116         }
117         d *= 1.0 / (2.0 * sigma * sigma);
118         exp(d, d);
119 }
120
121 static void crossBilateralFilter(const Mat& image, const Mat& edge_image,
122                 const Mat confidence, Mat& dst, int d, float sigma_color,
123                 float sigma_space, bool flag = false) {
124         const int rows = image.rows;
125         const int cols = image.cols;
126         Mat image_extended, edge_image_extended, confidence_extended;
127         copyMakeBorder(image, image_extended, d, d, d, d, BORDER_DEFAULT);
128         copyMakeBorder(edge_image, edge_image_extended, d, d, d, d, BORDER_DEFAULT);
129         copyMakeBorder(confidence, confidence_extended, d, d, d, d, BORDER_CONSTANT,
130                         Scalar(0));
131         Mat weights_space(2 * d + 1, 2 * d + 1, CV_32F);
132         wd(weights_space, d, d, d, d, sigma_space);
133         Mat weights(2 * d + 1, 2 * d + 1, CV_32F);
134         Mat weighted_sum(2 * d + 1, 2 * d + 1, CV_32F);
135
136         std::vector<Mat> image_extended_channels;
137         split(image_extended, image_extended_channels);
138
139         //l57, l64, l111
140         for (int row = 0; row < rows; ++row) {
141                 //l58, l65, l112
142                 for (int col = 0; col < cols; ++col) {
143                         wc(edge_image_extended, weights, row + d, col + d, d, d, d, d,
144                                         sigma_color);
145
146                         Range window_rows(row, row + 2 * d + 1);
147                         Range window_cols(col, col + 2 * d + 1);
148
149                         multiply(weights, confidence_extended(window_rows, window_cols),
150                                         weights);
151                         multiply(weights, weights_space, weights);
152                         float weights_sum = (float) sum(weights)[0];
153
154                         //l61, l68, l113
155                         for (int ch = 0; ch < 2; ++ch) {
156                                 multiply(weights,
157                                                 image_extended_channels[ch](window_rows,
                                                 ↪  window_cols),
```

```
158                                             weighted_sum);
159                                 float total_sum = (float) sum(weighted_sum)[0];
160
161                                 dst.at<Vec2f>(row, col)[ch] =
162                                             (flag && fabs(weights_sum) < 1e-9) ?
163                                                         image.at<Vec2f>(row, col)[ch] :
164                                                         total_sum / weights_sum;
165                         }
166                 }
167         }
168 }
169
170 static void calcConfidence(const Mat& prev, const Mat& next, const Mat& flow,
171                 Mat& confidence, int max_flow) {
172         const int rows = prev.rows;
173         const int cols = prev.cols;
174         confidence = Mat::zeros(rows, cols, CV_32F);
175
176         //l69, l83
177         for (int r0 = 0; r0 < rows; ++r0) {
178                 //l70, l84
179                 for (int c0 = 0; c0 < cols; ++c0) {
180                         Vec2f flow_at_point = flow.at<Vec2f>(r0, c0);
181                         int u0 = cvRound(flow_at_point[0]);
182                         if (r0 + u0 < 0) {
183                                 u0 = -r0;
184                         }
185                         if (r0 + u0 >= rows) {
186                                 u0 = rows - 1 - r0;
187                         }
188                         int v0 = cvRound(flow_at_point[1]);
189                         if (c0 + v0 < 0) {
190                                 v0 = -c0;
191                         }
192                         if (c0 + v0 >= cols) {
193                                 v0 = cols - 1 - c0;
194                         }
195
196                         const int top_row_shift = -std::min(r0 + u0, max_flow);
197                         const int bottom_row_shift = std::min(rows - 1 - (r0 + u0),
198                                         max_flow);
199                         const int left_col_shift = -std::min(c0 + v0, max_flow);
200                         const int right_col_shift = std::min(cols - 1 - (c0 + v0),
201                                         max_flow);
202
203                         bool first_flow_iteration = true;
204                         float sum_e = 0, min_e = 0;
205
206                         //l71, l85
207                         for (int u = top_row_shift; u <= bottom_row_shift; ++u) {
208                                 //l72, l86
209                                 for (int v = left_col_shift; v <= right_col_shift; ++v) {
210                                         float e = dist(prev.at<Vec3b>(r0, c0),
211                                                         next.at<Vec3b>(r0 + u0 + u, c0 + v0 + v));
212                                         if (first_flow_iteration) {
213                                                 sum_e = e;
214                                                 min_e = e;
215                                                 first_flow_iteration = false;
216                                         } else {
```

```
217                                             sum_e += e;
218                                             min_e = std::min(min_e, e);
219                                     }
220                             }
221                     }
222                     int windows_square = (bottom_row_shift - top_row_shift + 1)
223                                     * (right_col_shift - left_col_shift + 1);
224                     confidence.at<float>(r0, c0) =
225                                     (windows_square == 0) ? 0 : sum_e / windows_square - min_e;
226                     CV_Assert(confidence.at<float>(r0, c0) >= 0);
227             }
228     }
229 }
230
231 static void calcOpticalFlowSingleScaleSF(const Mat& prev_extended,
232                 const Mat& next_extended, const Mat& mask, Mat& flow,
233                 int averaging_radius, int max_flow, float sigma_dist,
234                 float sigma_color) {
235         const int averaging_radius_2 = averaging_radius << 1;
236         const int rows = prev_extended.rows - averaging_radius_2;
237         const int cols = prev_extended.cols - averaging_radius_2;
238
239         Mat weight_window(averaging_radius_2 + 1, averaging_radius_2 + 1, CV_32F);
240         Mat space_weight_window(averaging_radius_2 + 1, averaging_radius_2 + 1,
241                         CV_32F);
242
243         wd(space_weight_window, averaging_radius, averaging_radius,
244                 averaging_radius, averaging_radius, sigma_dist);
245
246         //l4, l14, l75, l89
247         for (int r0 = 0; r0 < rows; ++r0) {
248                 //l5, l15, l76, l90
249                 for (int c0 = 0; c0 < cols; ++c0) {
250                         if (!mask.at<uchar>(r0, c0)) {
251                                 continue;
252                         }
253
254                         // TODO: do smth with this creepy staff
255                         Vec2f flow_at_point = flow.at<Vec2f>(r0, c0);
256                         int u0 = cvRound(flow_at_point[0]);
257                         if (r0 + u0 < 0) {
258                                 u0 = -r0;
259                         }
260                         if (r0 + u0 >= rows) {
261                                 u0 = rows - 1 - r0;
262                         }
263                         int v0 = cvRound(flow_at_point[1]);
264                         if (c0 + v0 < 0) {
265                                 v0 = -c0;
266                         }
267                         if (c0 + v0 >= cols) {
268                                 v0 = cols - 1 - c0;
269                         }
270
271                         const int top_row_shift = -std::min(r0 + u0, max_flow);
272                         const int bottom_row_shift = std::min(rows - 1 - (r0 + u0),
273                                 max_flow);
274                         const int left_col_shift = -std::min(c0 + v0, max_flow);
275                         const int right_col_shift = std::min(cols - 1 - (c0 + v0),
```

```
276                             max_flow);
277
278                     float min_cost = FLT_MAX, best_u = (float) u0, best_v = (float) v0;
279
280                     wc(prev_extended, weight_window, r0 + averaging_radius,
281                             c0 + averaging_radius, averaging_radius, averaging_radius,
282                             averaging_radius, averaging_radius, sigma_color);
283                     multiply(weight_window, space_weight_window, weight_window);
284
285                     const int prev_extended_top_window_row = r0;
286                     const int prev_extended_left_window_col = c0;
287
288                     //l8, l18, l79, l93
289                     for (int u = top_row_shift; u <= bottom_row_shift; ++u) {
290                             const int next_extended_top_window_row = r0 + u0 + u;
291                             //l9, l19, l80, l94
292                             for (int v = left_col_shift; v <= right_col_shift; ++v) {
293                                     const int next_extended_left_window_col = c0 + v0 + v;
294
295                                     float cost = 0;
296                                     //l10, l20, l81, l95
297                                     for (int r = 0; r <= averaging_radius_2; ++r) {
298                                             const Vec3b *prev_extended_window_row =
299                                                     prev_extended.ptr<Vec3b>(
300                                                             prev_extended_top_window_row + r);
301                                             const Vec3b *next_extended_window_row =
302                                                     next_extended.ptr<Vec3b>(
303                                                             next_extended_top_window_row + r);
304                                             const float* weight_window_row =
305                                                     weight_window.ptr<float>(r);
306                                     //l11, l21, l82, l96
307                                     for (int c = 0; c <= averaging_radius_2; ++c) {
308                                             cost +=
309                                                     weight_window_row[c]
310                                                     * dist(
311                                                             prev_extended_window_row[
312                                                                     prev_extended_left_window_col+
                                                                     ↪  c],
313                                                             next_extended_window_row[
314                                                                     next_extended_left_window_col+
                                                                     ↪  c]);
315                                             }
316                                     }
317                                     // cost should be divided by sum(weight_window), but because
318                                     // we interested only in min(cost) and sum(weight_window) is
                                     ↪  constant
319                                     // for every point - we remove it
320
321                                     if (cost < min_cost) {
322                                             min_cost = cost;
323                                             best_u = (float) (u + u0);
324                                             best_v = (float) (v + v0);
325                                     }
326                             }
327                     }
328                     flow.at<Vec2f>(r0, c0) = Vec2f(best_u, best_v);
329             }
330     }
331 }
```

```
332
333 static Mat upscaleOpticalFlow(int new_rows, int new_cols, const Mat& image,
334                 const Mat& confidence, Mat& flow, int averaging_radius,
335                 float sigma_dist, float sigma_color) {
336         crossBilateralFilter(flow, image, confidence, flow, averaging_radius,
337                         sigma_color, sigma_dist, true);
338         Mat new_flow;
339         resize(flow, new_flow, Size(new_cols, new_rows), 0, 0, INTER_NEAREST);
340         new_flow *= 2;
341         return new_flow;
342 }
343
344 static Mat calcIrregularityMat(const Mat& flow, int radius) {
345         const int rows = flow.rows;
346         const int cols = flow.cols;
347         Mat irregularity = Mat::zeros(rows, cols, CV_32F);
348         //l27, l41
349         for (int r = 0; r < rows; ++r) {
350                 const int start_row = std::max(0, r - radius);
351                 const int end_row = std::min(rows - 1, r + radius);
352                 //l28, l42
353                 for (int c = 0; c < cols; ++c) {
354                         const int start_col = std::max(0, c - radius);
355                         const int end_col = std::min(cols - 1, c + radius);
356                         //l29, l43
357                         for (int dr = start_row; dr <= end_row; ++dr) {
358                                 //l30, l44
359                                 for (int dc = start_col; dc <= end_col; ++dc) {
360                                         const float diff = dist(flow.at<Vec2f>(r, c),
361                                                         flow.at<Vec2f>(dr, dc));
362                                         if (diff > irregularity.at<float>(r, c)) {
363                                                 irregularity.at<float>(r, c) = diff;
364                                         }
365                                 }
366                         }
367                 }
368         }
369         return irregularity;
370 }
371
372 static void selectPointsToRecalcFlow(const Mat& flow,
373                 int irregularity_metric_radius, float speed_up_thr, int curr_rows,
374                 int curr_cols, const Mat& prev_speed_up, Mat& speed_up, Mat& mask) {
375         const int prev_rows = flow.rows;
376         const int prev_cols = flow.cols;
377
378         Mat is_flow_regular = calcIrregularityMat(flow, irregularity_metric_radius)
379                 < speed_up_thr;
380         Mat done = Mat::zeros(prev_rows, prev_cols, CV_8U);
381         speed_up = Mat::zeros(curr_rows, curr_cols, CV_8U);
382         mask = Mat::zeros(curr_rows, curr_cols, CV_8U);
383
384         //l31, l45
385         for (int r = 0; r < is_flow_regular.rows; ++r) {
386                 //l32, l46
387                 for (int c = 0; c < is_flow_regular.cols; ++c) {
388                         if (!done.at<uchar>(r, c)) {
389                                 if (is_flow_regular.at<uchar>(r, c) && 2 * r + 1 < curr_rows
390                                         && 2 * c + 1 < curr_cols) {
```

```
391
392                         bool all_flow_in_region_regular = true;
393                         int speed_up_at_this_point = prev_speed_up.at<uchar>(r, c);
394                         int step = (1 << speed_up_at_this_point) - 1;
395                         int prev_top = r;
396                         int prev_bottom = std::min(r + step, prev_rows - 1);
397                         int prev_left = c;
398                         int prev_right = std::min(c + step, prev_cols - 1);
399
400                         //l33, l47
401                         for (int rr = prev_top; rr <= prev_bottom; ++rr) {
402                                 //l34, l48
403                                 for (int cc = prev_left; cc <= prev_right; ++cc) {
404                                         done.at<uchar>(rr, cc) = 1;
405                                         if (!is_flow_regular.at<uchar>(rr, cc)) {
406                                                 all_flow_in_region_regular = false;
407                                         }
408                                 }
409                         }
410
411                         int curr_top = std::min(2 * r, curr_rows - 1);
412                         int curr_bottom = std::min(2 * (r + step) + 1,
413                                         curr_rows - 1);
414                         int curr_left = std::min(2 * c, curr_cols - 1);
415                         int curr_right = std::min(2 * (c + step) + 1,
416                                         curr_cols - 1);
417
418                         if (all_flow_in_region_regular && curr_top != curr_bottom
419                                         && curr_left != curr_right) {
420                                 mask.at<uchar>(curr_top, curr_left) =
                                ↪  MASK_TRUE_VALUE;
421                                 mask.at<uchar>(curr_bottom, curr_left) =
422                                                 MASK_TRUE_VALUE;
423                                 mask.at<uchar>(curr_top, curr_right) =
                                ↪  MASK_TRUE_VALUE;
424                                 mask.at<uchar>(curr_bottom, curr_right) =
425                                                 MASK_TRUE_VALUE;
426                                 //l35, l49
427                                 for (int rr = curr_top; rr <= curr_bottom; ++rr) {
428                                         //l36, l50
429                                         for (int cc = curr_left; cc <= curr_right;
                                        ↪  ++cc) {
430                                                 speed_up.at<uchar>(rr, cc) =
431                                                                 (uchar)
                                                                ↪  (speed_up_at_this_point
                                                                ↪  + 1);
432                                         }
433                                 }
434                         } else {
435                                 //l37, l51
436                                 for (int rr = curr_top; rr <= curr_bottom; ++rr) {
437                                         //l38, l52
438                                         for (int cc = curr_left; cc <= curr_right;
                                        ↪  ++cc) {
439                                                 mask.at<uchar>(rr, cc) =
                                                ↪  MASK_TRUE_VALUE;
440                                         }
441                                 }
442                         }
```

```
443                                 } else {
444                                         done.at<uchar>(r, c) = 1;
445                                         //l39, l53
446                                         for (int dr = 0; dr <= 1; ++dr) {
447                                                 int nr = 2 * r + dr;
448                                                 //l40, l54
449                                                 for (int dc = 0; dc <= 1; ++dc) {
450                                                         int nc = 2 * c + dc;
451                                                         if (nr < curr_rows && nc < curr_cols) {
452                                                                 mask.at<uchar>(nr, nc) =
                                                                 ↪  MASK_TRUE_VALUE;
453                                                         }
454                                                 }
455                                         }
456                                 }
457                         }
458                 }
459         }
460 }
461
462 static inline float extrapolateValueInRect(int height, int width, float v11,
463                 float v12, float v21, float v22, int r, int c) {
464         if (r == 0 && c == 0) {
465                 return v11;
466         }
467         if (r == 0 && c == width) {
468                 return v12;
469         }
470         if (r == height && c == 0) {
471                 return v21;
472         }
473         if (r == height && c == width) {
474                 return v22;
475         }
476
477         CV_Assert(height > 0 && width > 0);
478         float qr = float(r) / height;
479         float pr = 1.0f - qr;
480         float qc = float(c) / width;
481         float pc = 1.0f - qc;
482
483         return v11 * pr * pc + v12 * pr * qc + v21 * qr * pc + v22 * qc * qr;
484 }
485
486 static void extrapolateFlow(Mat& flow, const Mat& speed_up) {
487         const int rows = flow.rows;
488         const int cols = flow.cols;
489         Mat done = Mat::zeros(rows, cols, CV_8U);
490         //l97, l101
491         for (int r = 0; r < rows; ++r) {
492                 //l98, l102
493                 for (int c = 0; c < cols; ++c) {
494                         if (!done.at<uchar>(r, c) && speed_up.at<uchar>(r, c) > 1) {
495                                 int step = (1 << speed_up.at<uchar>(r, c)) - 1;
496                                 int top = r;
497                                 int bottom = std::min(r + step, rows - 1);
498                                 int left = c;
499                                 int right = std::min(c + step, cols - 1);
500
```

```
501                                 int height = bottom - top;
502                                 int width = right - left;
503                                 //l99, l103
504                                 for (int rr = top; rr <= bottom; ++rr) {
505                                         //l100, l104
506                                         for (int cc = left; cc <= right; ++cc) {
507                                                 done.at<uchar>(rr, cc) = 1;
508                                                 Vec2f flow_at_point;
509                                                 Vec2f top_left = flow.at<Vec2f>(top, left);
510                                                 Vec2f top_right = flow.at<Vec2f>(top, right);
511                                                 Vec2f bottom_left = flow.at<Vec2f>(bottom, left);
512                                                 Vec2f bottom_right = flow.at<Vec2f>(bottom, right);
513
514                                                 flow_at_point[0] = extrapolateValueInRect(height,
                                                    ↪  width,
515                                                                 top_left[0], top_right[0],
                                                                    ↪  bottom_left[0],
516                                                                 bottom_right[0], rr - top, cc -
                                                                    ↪  left);
517
518                                                 flow_at_point[1] = extrapolateValueInRect(height,
                                                    ↪  width,
519                                                                 top_left[1], top_right[1],
                                                                    ↪  bottom_left[1],
520                                                                 bottom_right[1], rr - top, cc -
                                                                    ↪  left);
521                                                 flow.at<Vec2f>(rr, cc) = flow_at_point;
522                                         }
523                                 }
524                         }
525                 }
526         }
527 }
528
529 static void buildPyramidWithResizeMethod(const Mat& src,
530                 std::vector<Mat>& pyramid, int layers, int interpolation_type) {
531         pyramid.push_back(src);
532         //l0, l1
533         for (int i = 1; i <= layers; ++i) {
534                 Mat prev = pyramid[i - 1];
535                 if (prev.rows <= 1 || prev.cols <= 1) {
536                         break;
537                 }
538
539                 Mat next;
540                 resize(prev, next, Size((prev.cols + 1) / 2, (prev.rows + 1) / 2), 0, 0,
541                                 interpolation_type);
542                 pyramid.push_back(next);
543         }
544 }
545
546 CV_EXPORTS_W void orig_calcOpticalFlowSF(InputArray _from, InputArray _to,
547                 OutputArray _resulted_flow, int layers, int averaging_radius,
548                 int max_flow, double sigma_dist, double sigma_color,
549                 int postprocess_window, double sigma_dist_fix, double sigma_color_fix,
550                 double occ_thr, int upscale_averaging_radius, double upscale_sigma_dist,
551                 double upscale_sigma_color, double speed_up_thr) {
552         Mat from = _from.getMat();
553         Mat to = _to.getMat();
```

```
554
555         std::vector<Mat> pyr_from_images;
556         std::vector<Mat> pyr_to_images;
557
558         buildPyramidWithResizeMethod(from, pyr_from_images, layers - 1,
559                         INTER_CUBIC);
560         buildPyramidWithResizeMethod(to, pyr_to_images, layers - 1, INTER_CUBIC);
561
562         CV_Assert(
563                         (int )pyr_from_images.size() == layers
564                                         && (int )pyr_to_images.size() == layers);
565
566         Mat curr_from, curr_to, prev_from, prev_to;
567         Mat curr_from_extended, curr_to_extended;
568
569         curr_from = pyr_from_images[layers - 1];
570         curr_to = pyr_to_images[layers - 1];
571
572         copyMakeBorder(curr_from, curr_from_extended, averaging_radius,
573                         averaging_radius, averaging_radius, averaging_radius,
574                         BORDER_DEFAULT);
575         copyMakeBorder(curr_to, curr_to_extended, averaging_radius,
576                         averaging_radius, averaging_radius, averaging_radius,
577                         BORDER_DEFAULT);
578
579         Mat mask = Mat::ones(curr_from.size(), CV_8U);
580         Mat mask_inv = Mat::ones(curr_from.size(), CV_8U);
581
582         Mat flow = Mat::zeros(curr_from.size(), CV_32FC2);
583         Mat flow_inv = Mat::zeros(curr_to.size(), CV_32FC2);
584
585         Mat confidence;
586         Mat confidence_inv;
587
588         calcOpticalFlowSingleScaleSF(curr_from_extended, curr_to_extended, mask,
589                         flow, averaging_radius, max_flow, (float) sigma_dist,
590                         (float) sigma_color);
591
592         calcOpticalFlowSingleScaleSF(curr_to_extended, curr_from_extended, mask_inv,
593                         flow_inv, averaging_radius, max_flow, (float) sigma_dist,
594                         (float) sigma_color);
595
596         removeOcclusions(flow, flow_inv, (float) occ_thr, confidence);
597
598         removeOcclusions(flow_inv, flow, (float) occ_thr, confidence_inv);
599
600         Mat speed_up = Mat::zeros(curr_from.size(), CV_8U);
601         Mat speed_up_inv = Mat::zeros(curr_from.size(), CV_8U);
602
603         //l26
604         for (int curr_layer = layers - 2; curr_layer >= 0; --curr_layer) {
605                 curr_from = pyr_from_images[curr_layer];
606                 curr_to = pyr_to_images[curr_layer];
607                 prev_from = pyr_from_images[curr_layer + 1];
608                 prev_to = pyr_to_images[curr_layer + 1];
609
610                 copyMakeBorder(curr_from, curr_from_extended, averaging_radius,
611                                 averaging_radius, averaging_radius, averaging_radius,
612                                 BORDER_DEFAULT);
```

```
613                 copyMakeBorder(curr_to, curr_to_extended, averaging_radius,
614                                 averaging_radius, averaging_radius, averaging_radius,
615                                 BORDER_DEFAULT);
616
617                 const int curr_rows = curr_from.rows;
618                 const int curr_cols = curr_from.cols;
619
620                 Mat new_speed_up, new_speed_up_inv;
621
622                 selectPointsToRecalcFlow(flow, averaging_radius, (float) speed_up_thr,
623                                 curr_rows, curr_cols, speed_up, new_speed_up, mask);
624
625                 selectPointsToRecalcFlow(flow_inv, averaging_radius,
626                                 (float) speed_up_thr, curr_rows, curr_cols, speed_up_inv,
627                                 new_speed_up_inv, mask_inv);
628
629                 speed_up = new_speed_up;
630                 speed_up_inv = new_speed_up_inv;
631
632                 flow = upscaleOpticalFlow(curr_rows, curr_cols, prev_from, confidence,
633                                 flow, upscale_averaging_radius, (float) upscale_sigma_dist,
634                                 (float) upscale_sigma_color);
635
636                 flow_inv = upscaleOpticalFlow(curr_rows, curr_cols, prev_to,
637                                 confidence_inv, flow_inv, upscale_averaging_radius,
638                                 (float) upscale_sigma_dist, (float) upscale_sigma_color);
639
640                 calcConfidence(curr_from, curr_to, flow, confidence, max_flow);
641                 calcOpticalFlowSingleScaleSF(curr_from_extended, curr_to_extended, mask,
642                                 flow, averaging_radius, max_flow, (float) sigma_dist,
643                                 (float) sigma_color);
644
645                 calcConfidence(curr_to, curr_from, flow_inv, confidence_inv, max_flow);
646                 calcOpticalFlowSingleScaleSF(curr_to_extended, curr_from_extended,
647                                 mask_inv, flow_inv, averaging_radius, max_flow,
648                                 (float) sigma_dist, (float) sigma_color);
649
650                 extrapolateFlow(flow, speed_up);
651                 extrapolateFlow(flow_inv, speed_up_inv);
652
653                 //TODO: should we remove occlusions for the last stage?
654                 removeOcclusions(flow, flow_inv, (float) occ_thr, confidence);
655                 removeOcclusions(flow_inv, flow, (float) occ_thr, confidence_inv);
656         }
657
658         crossBilateralFilter(flow, curr_from, confidence, flow, postprocess_window,
659                         (float) sigma_color_fix, (float) sigma_dist_fix);
660
661         GaussianBlur(flow, flow, Size(3, 3), 5);
662
663         _resulted_flow.create(flow.size(), CV_32FC2);
664         Mat resulted_flow = _resulted_flow.getMat();
665         int from_to[] = { 0, 1, 1, 0 };
666         mixChannels(&flow, 1, &resulted_flow, 1, from_to, 2);
667 }
668
669 CV_EXPORTS_W void calcOpticalFlowSF(InputArray from, InputArray to,
670                 OutputArray flow, int layers, int averaging_block_size, int max_flow) {
671         orig_calcOpticalFlowSF(from, to, flow, layers, averaging_block_size, max_flow,
```

```
672                         4.1, 25.5, 18, 55.0, 25.5, 0.35, 18, 55.0, 25.5, 10);
673 }
```

Listing A.1 – Algorithme original *simpleflow*

Annexe B

Représentation spinale de l'algorithme *simpleFlow*

La représentation spinale, constituant cette annexe, correspond à une vue globale de l'algorithme *simplefow*, présenté en section 4.2.1. En raison du format de ce manuscrit, rendant complexe l'affichage intégral de notre représentation pour cet algorithme volumineux, nous avons choisi de représenter une version *compact* de cette représentation. De ce fait, la trace d'exécution du code source est moins détaillée et l'affichage ainsi que les dépendances sur les scalaires ont été omises. Les différentes parties de cet algorithme, concernées par un portage sur GPU, sont intégralement détaillées au cours du chapitre 3.

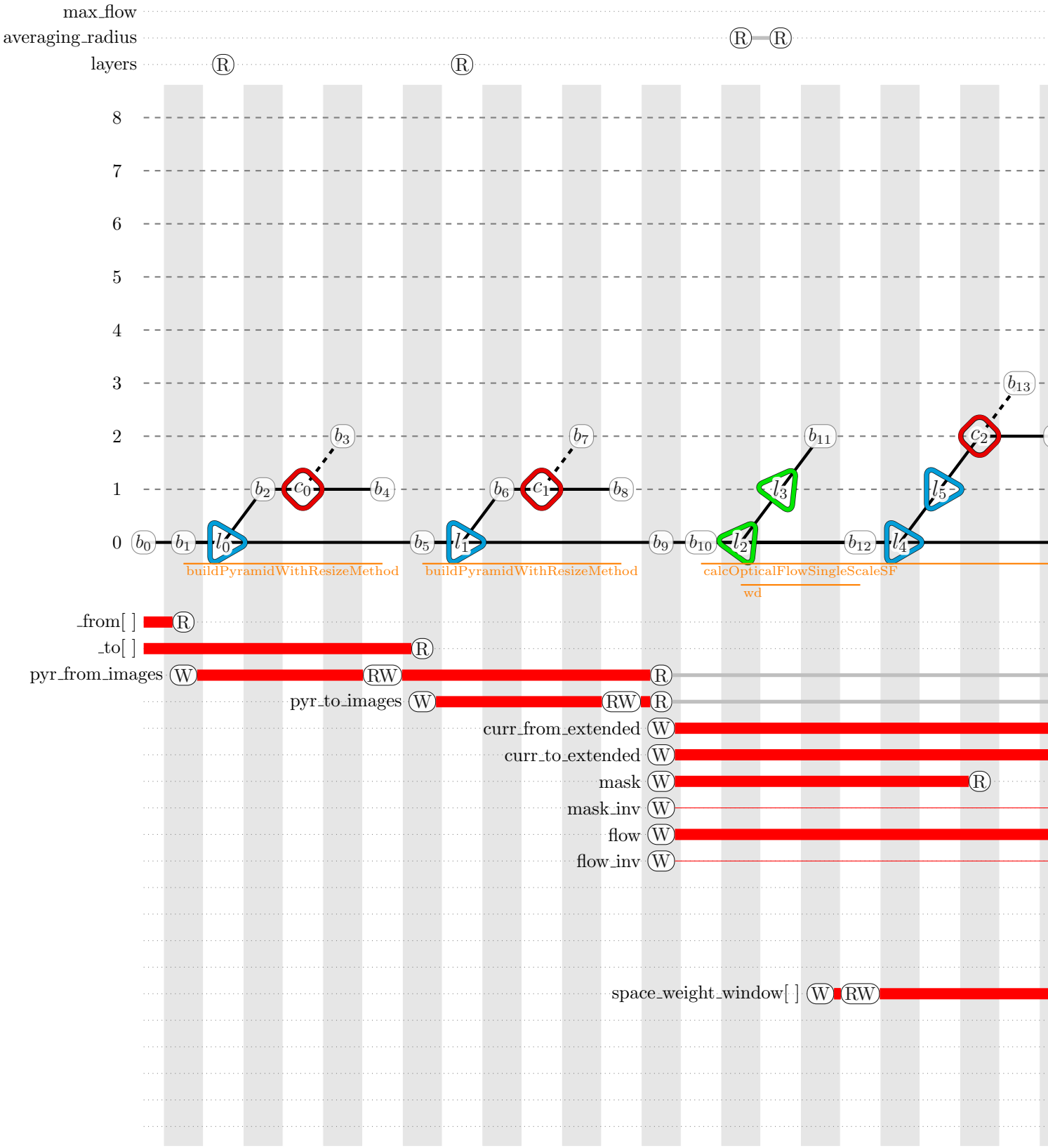


FIGURE B.1 – Représentation spinale du programme *simpleflow* (1/18)

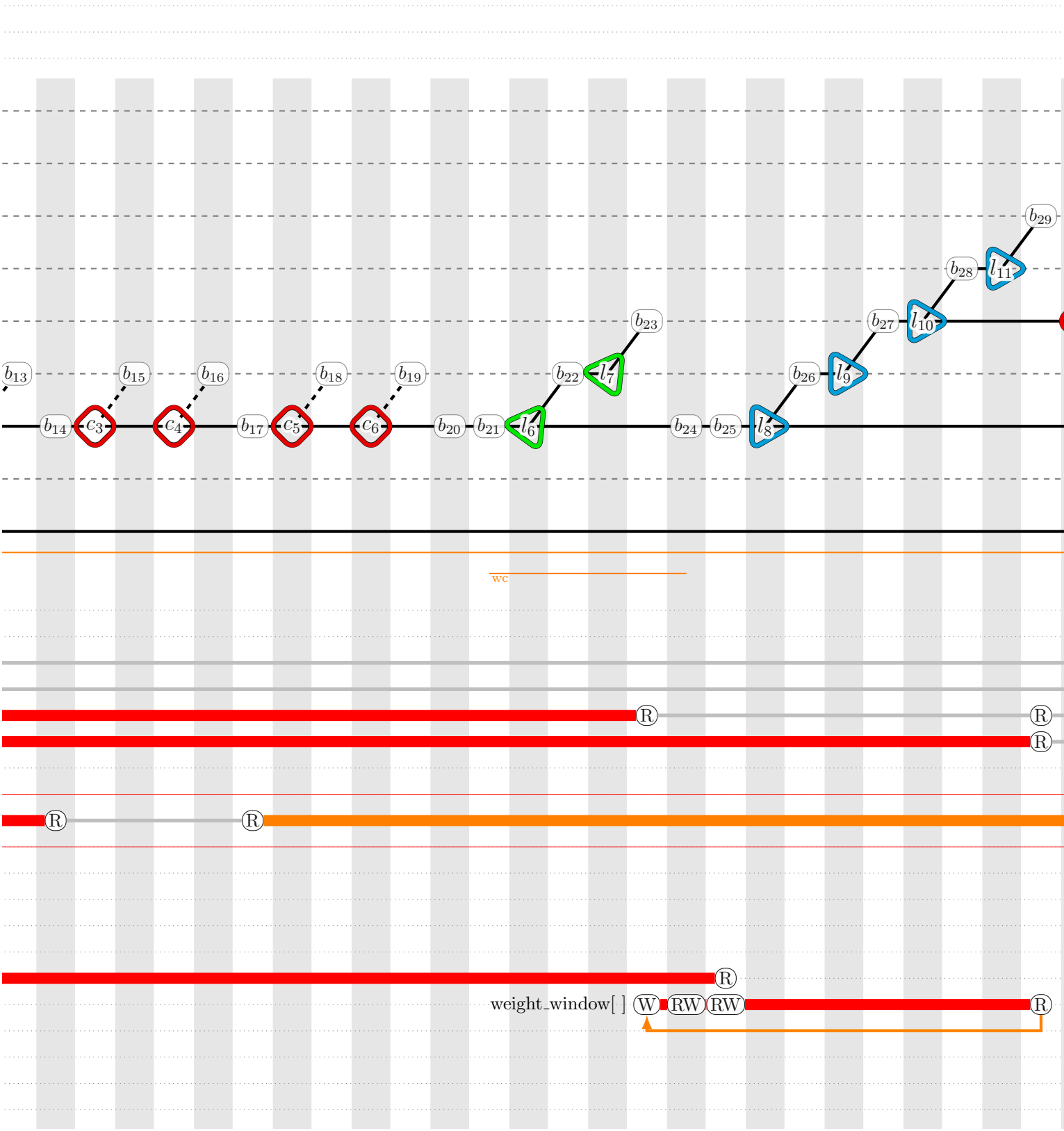


FIGURE B.1 – Représentation spinale du programme *simpleflow* (2/18)

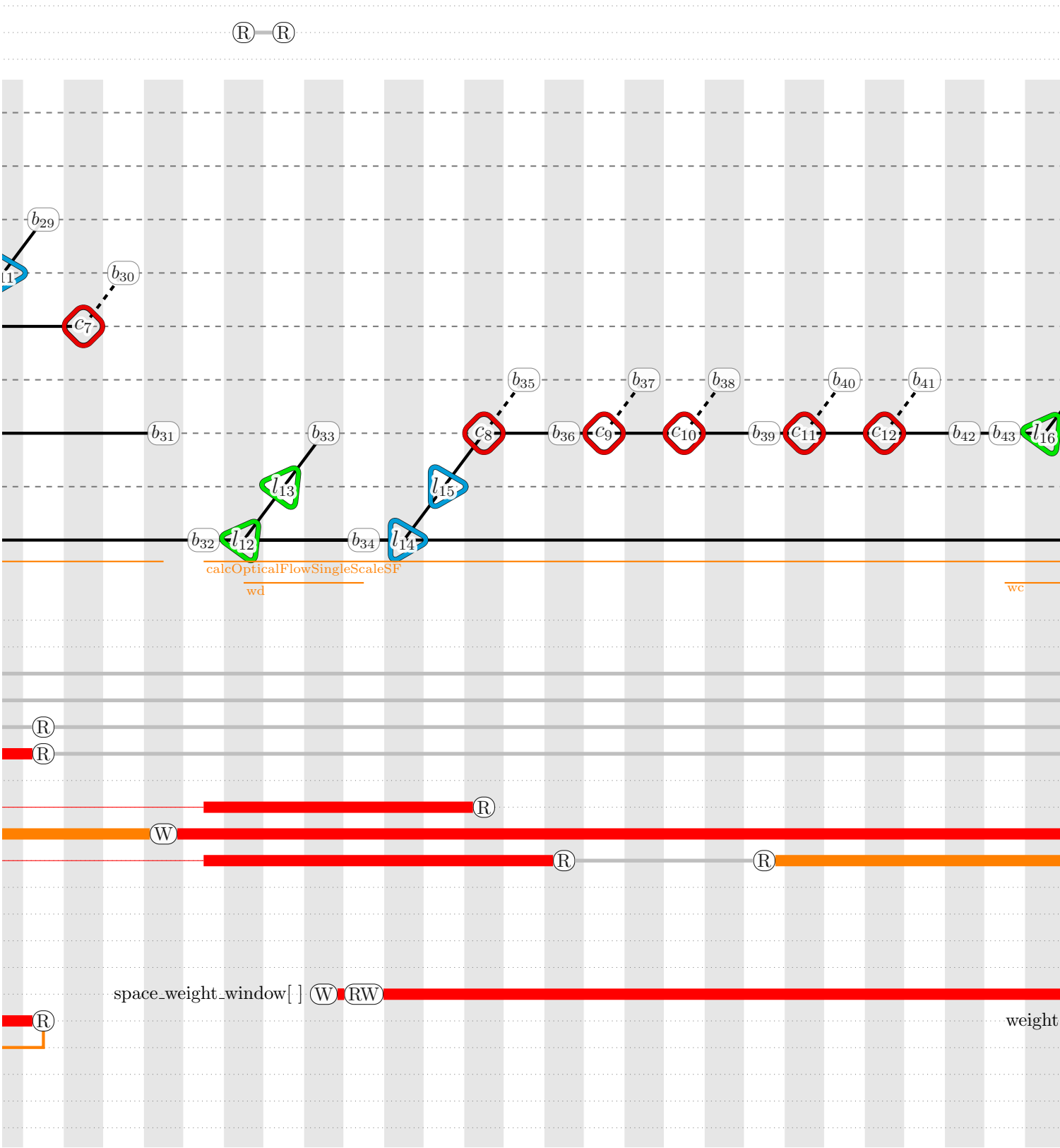


FIGURE B.1 – Représentation spinale du programme *simpleflow* (3/18)

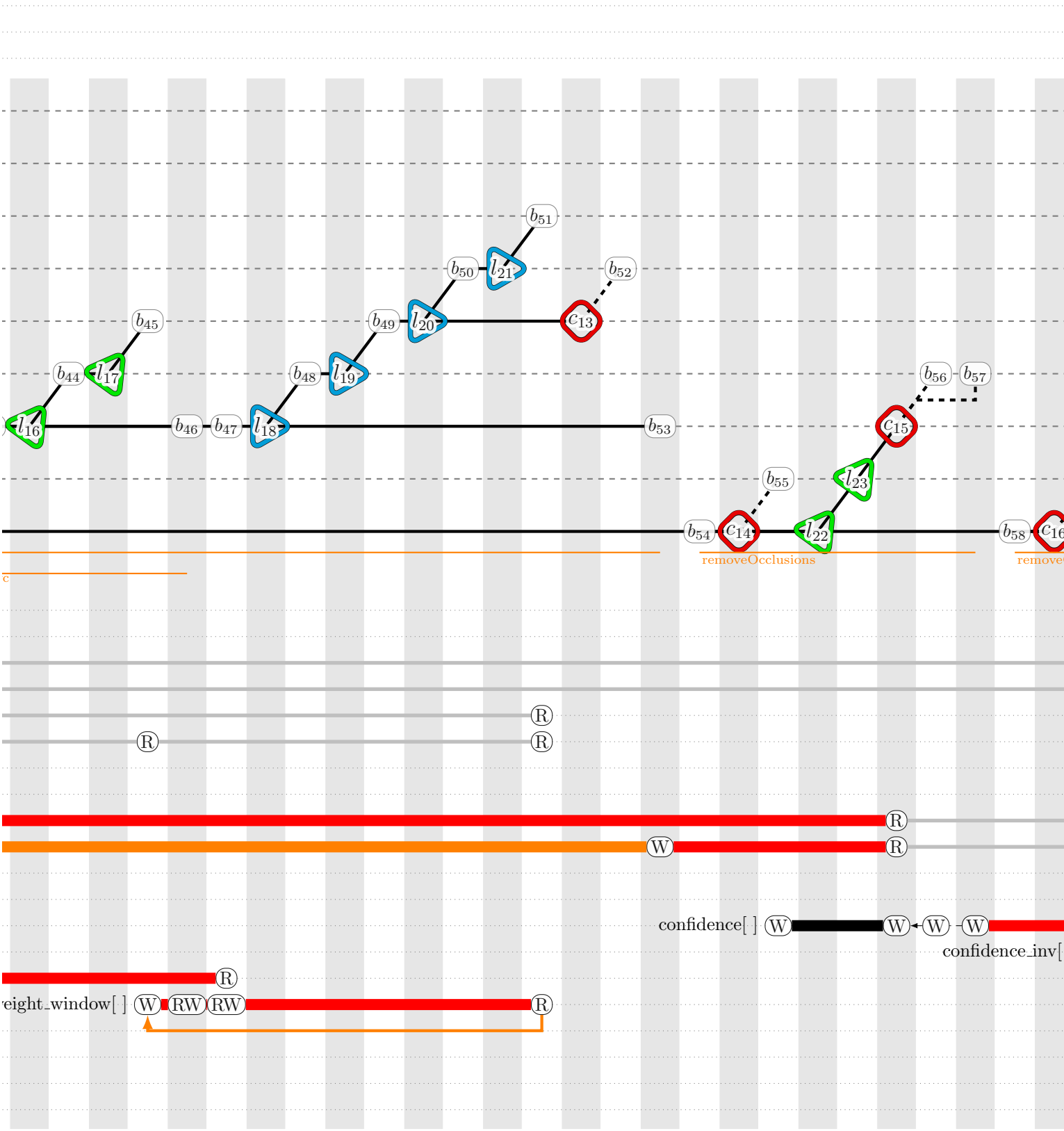


FIGURE B.1 – Représentation spinale du programme *simpleflow* (4/18)

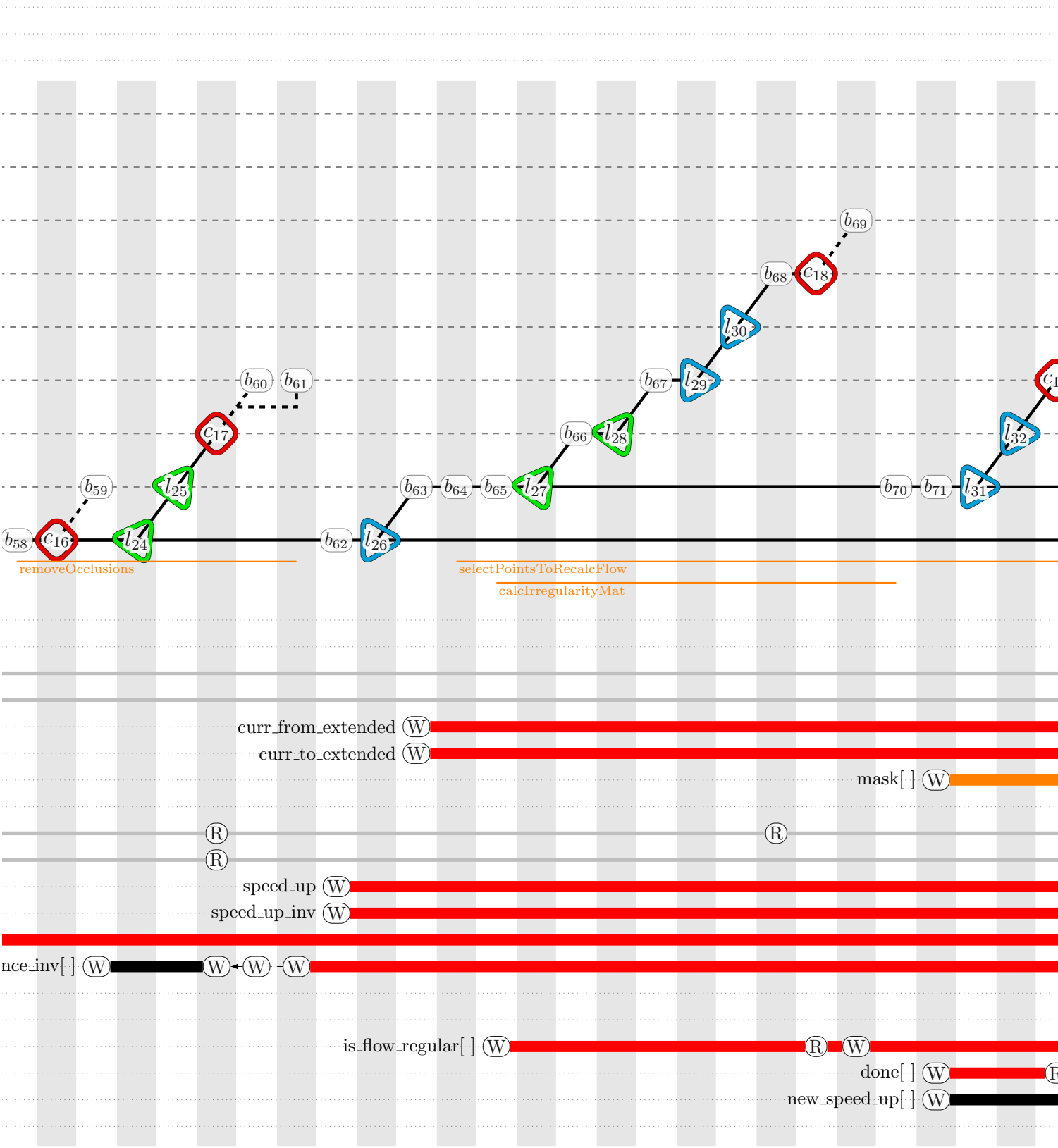


FIGURE B.1 – Représentation spinale du programme *simpleflow* (5/18)

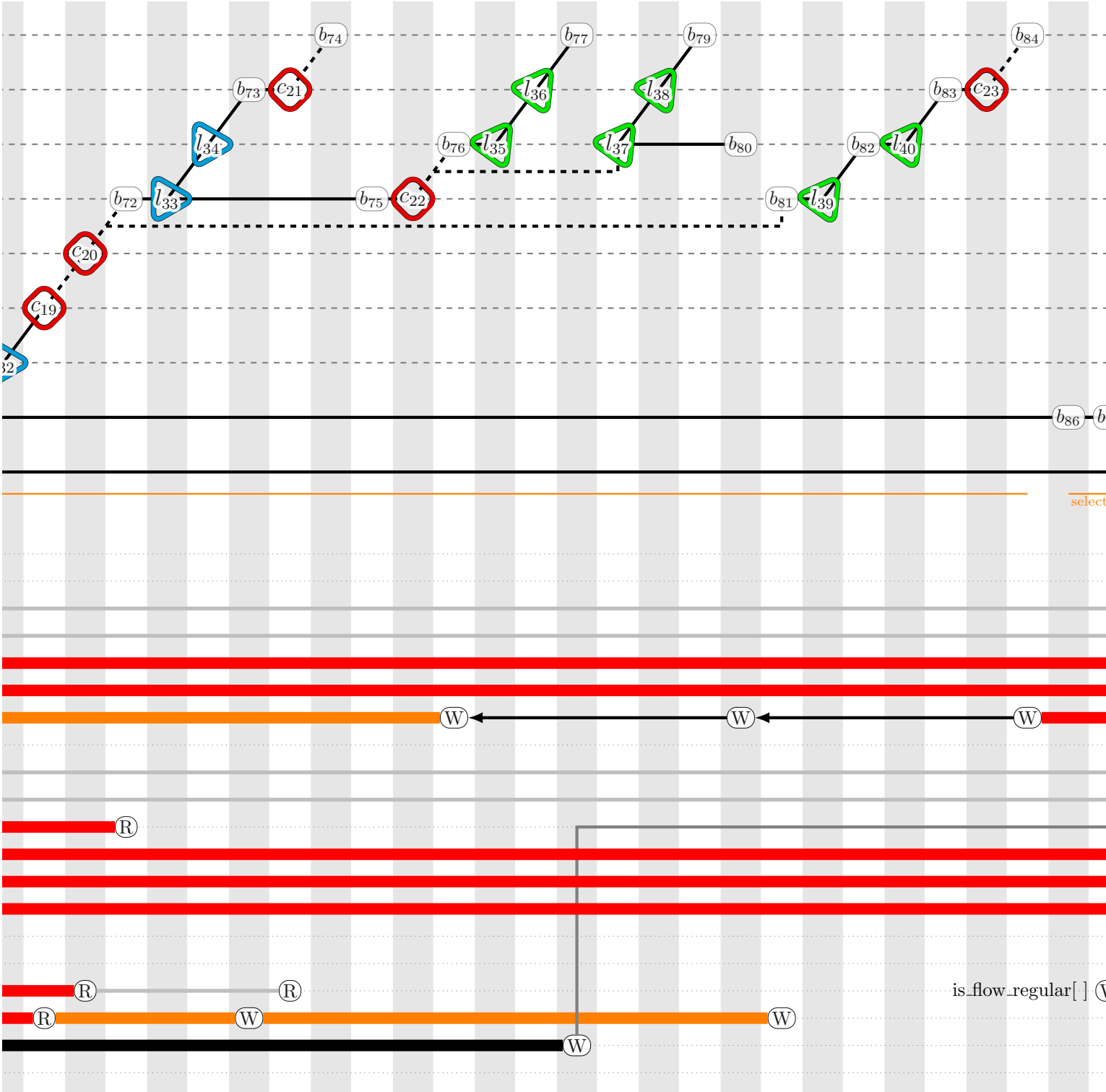


FIGURE B.1 – Représentation spinale du programme *simpleflow* (6/18)

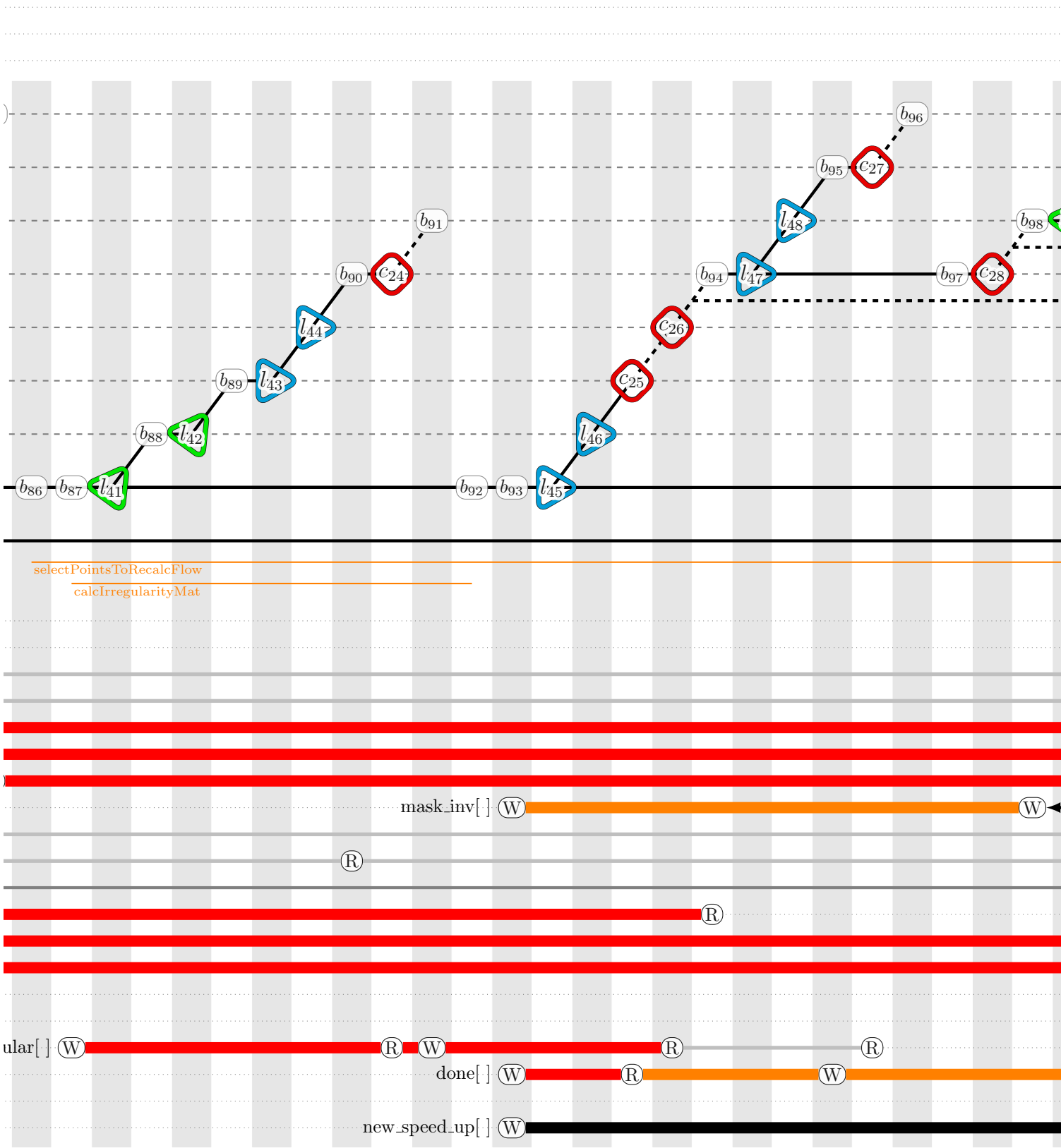


FIGURE B.1 – Représentation spinale du programme *simpleflow* (7/18)

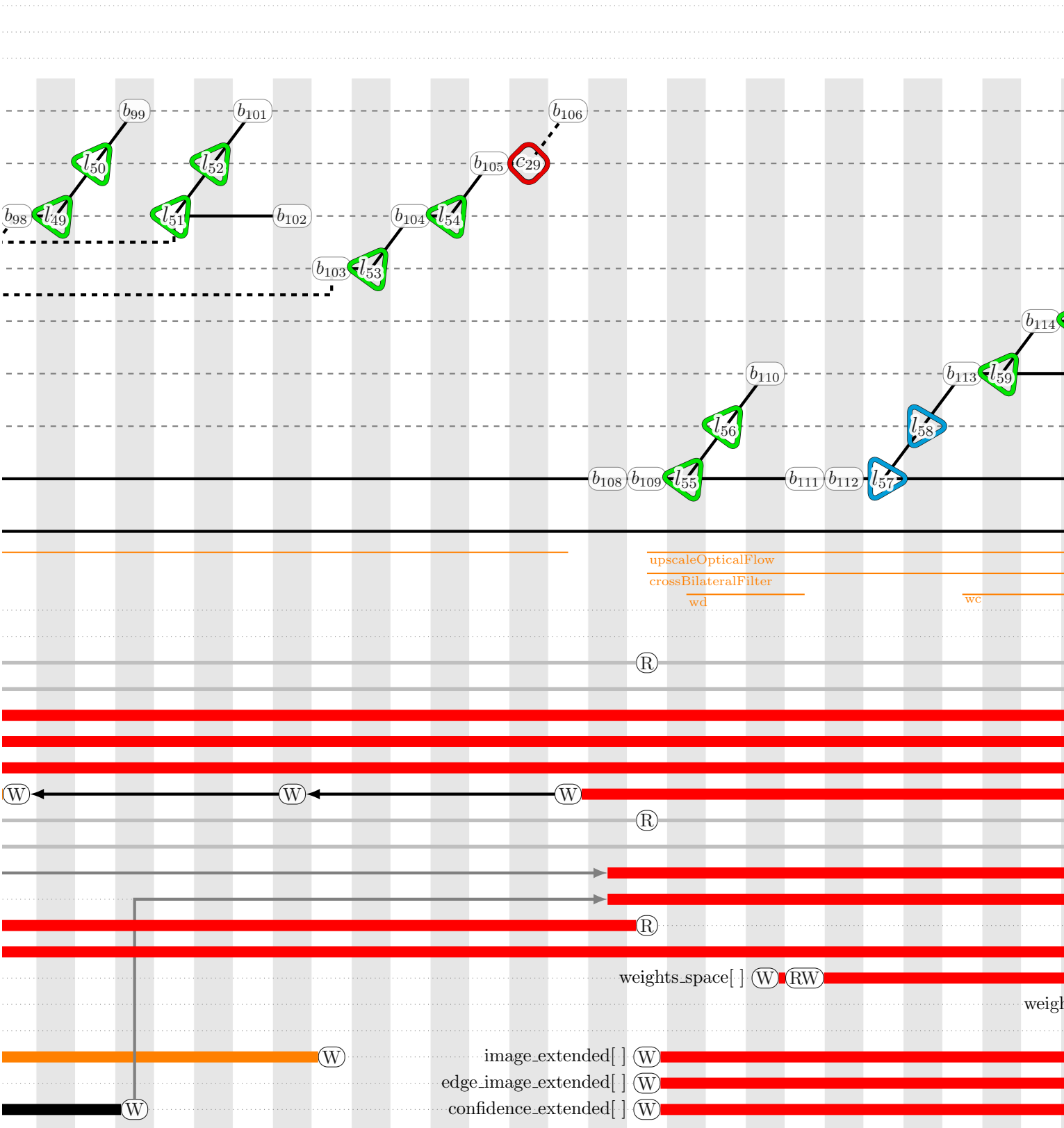


FIGURE B.1 – Représentation spinale du programme *simpleflow* (8/18)

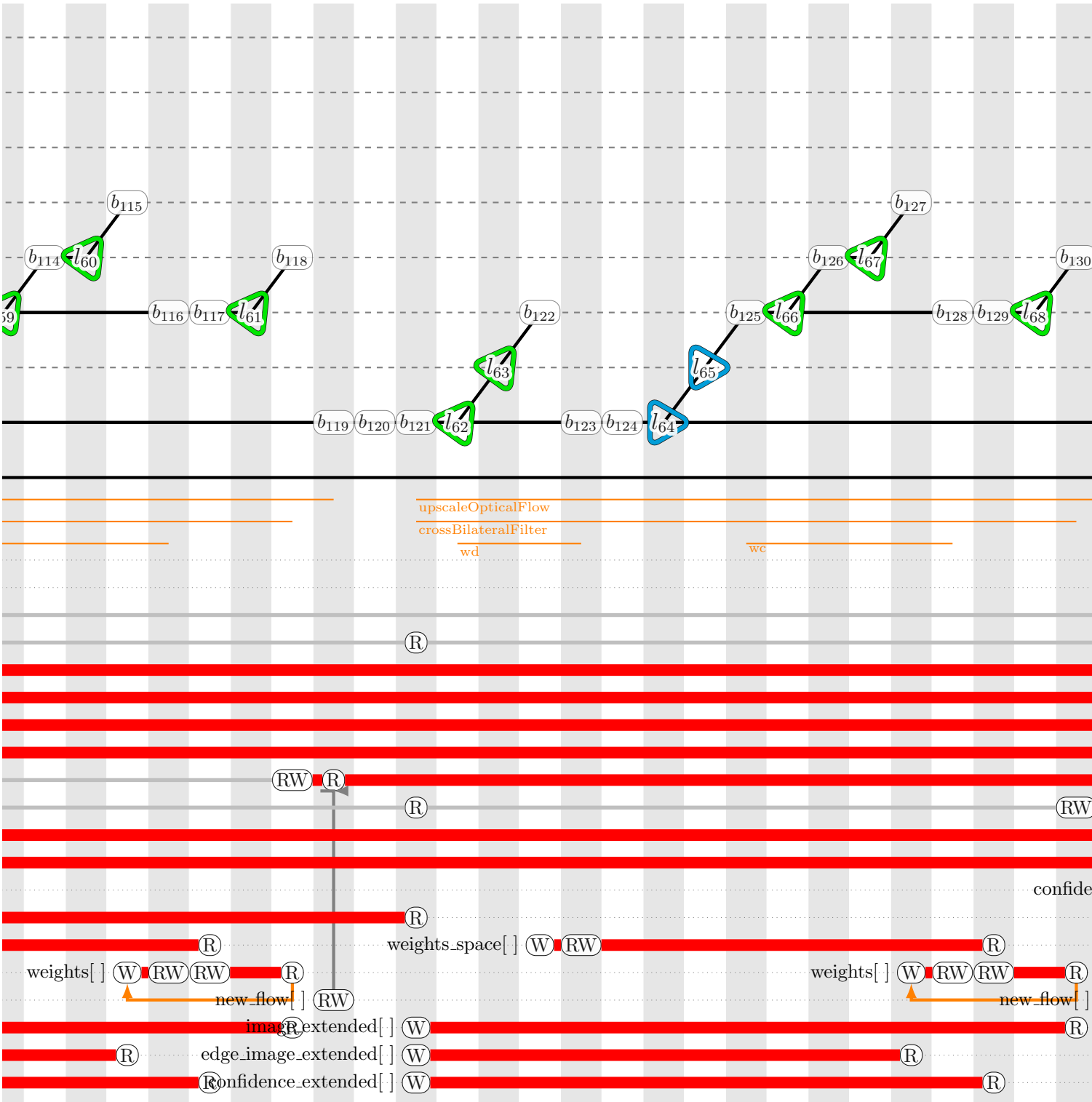


FIGURE B.1 – Représentation spinale du programme *simpleflow* (9/18)

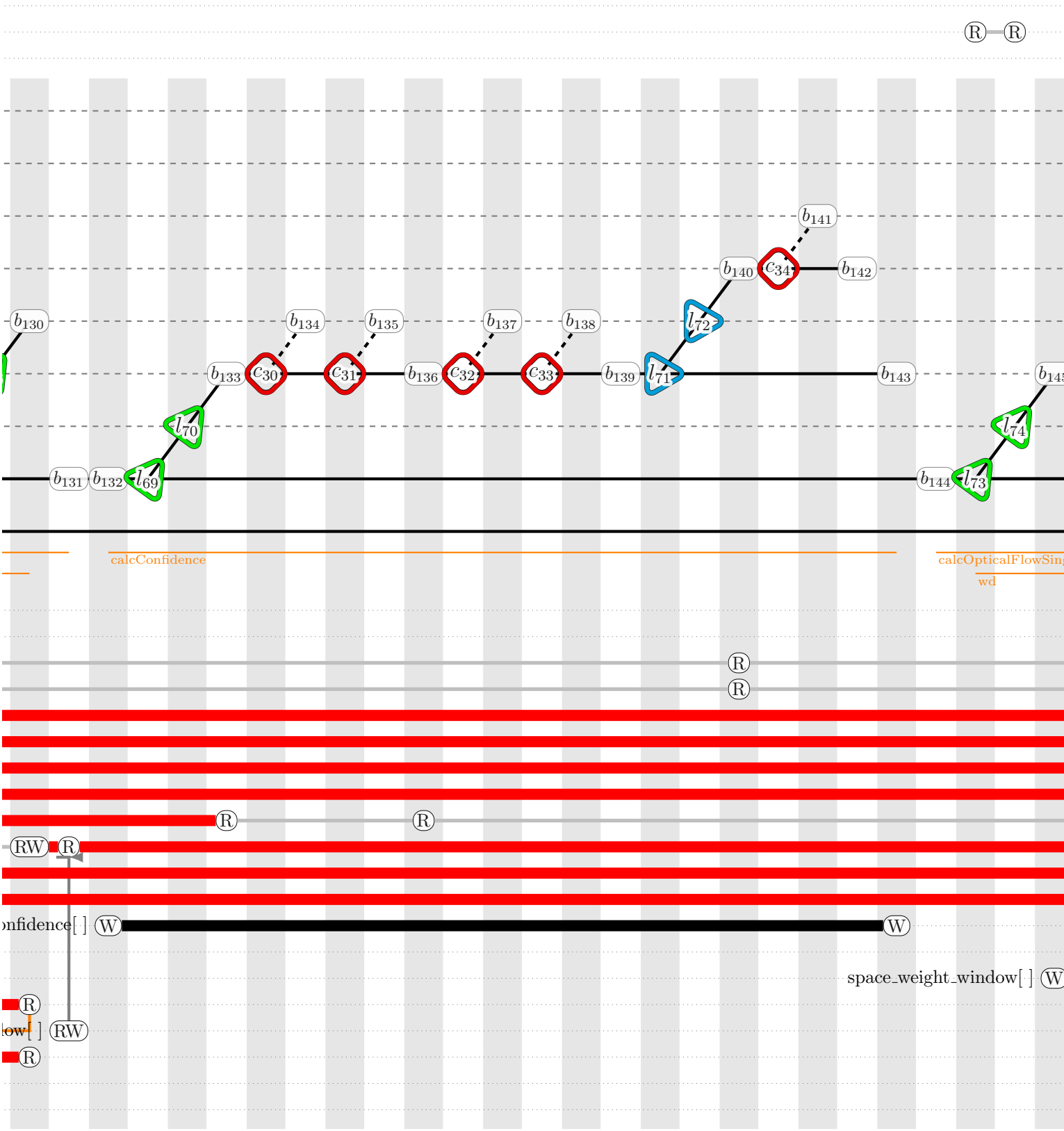


FIGURE B.1 – Représentation spinale du programme *simpleflow* (10/18)

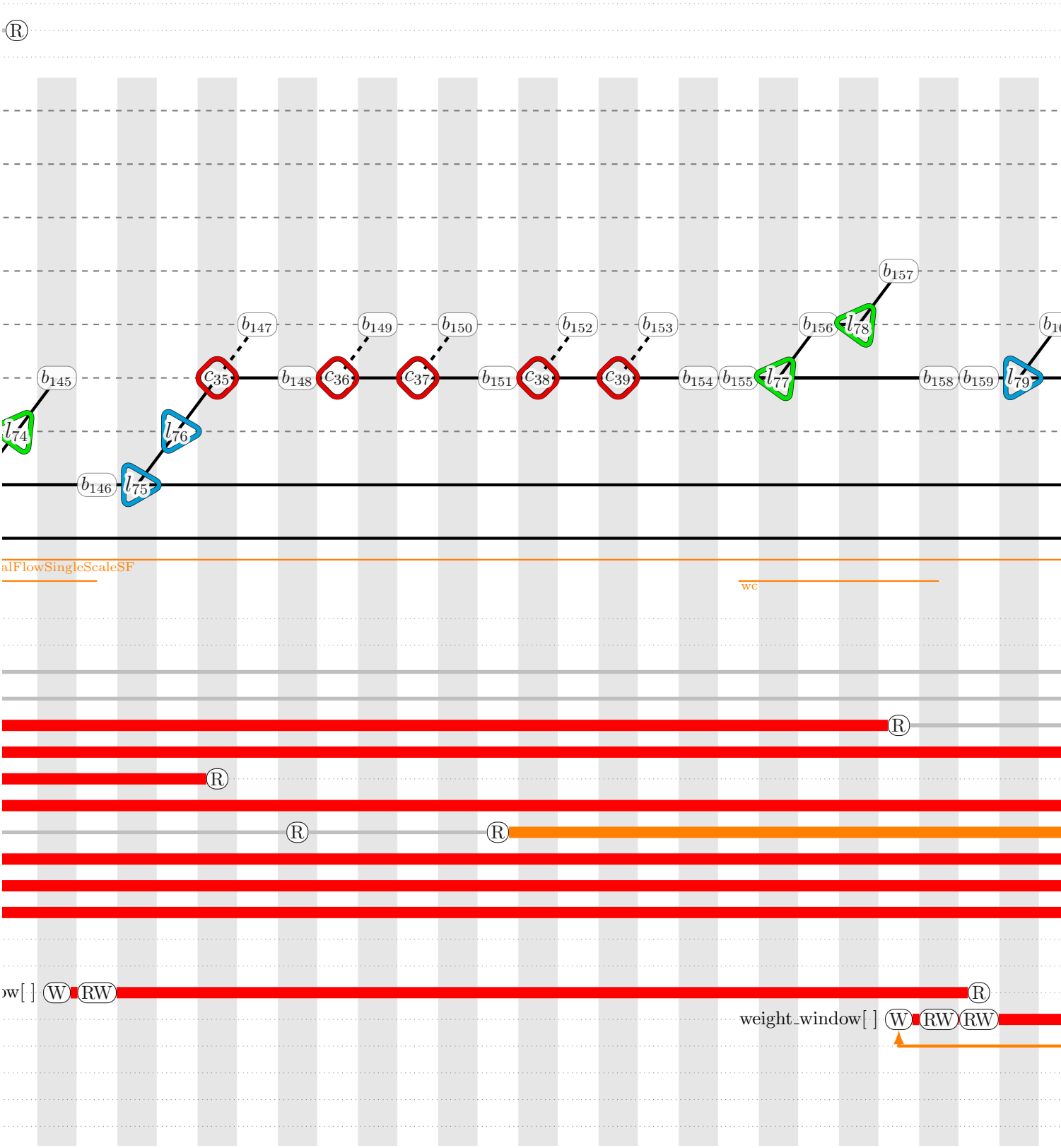


FIGURE B.1 – Représentation spinale du programme *simpleflow* (11/18)

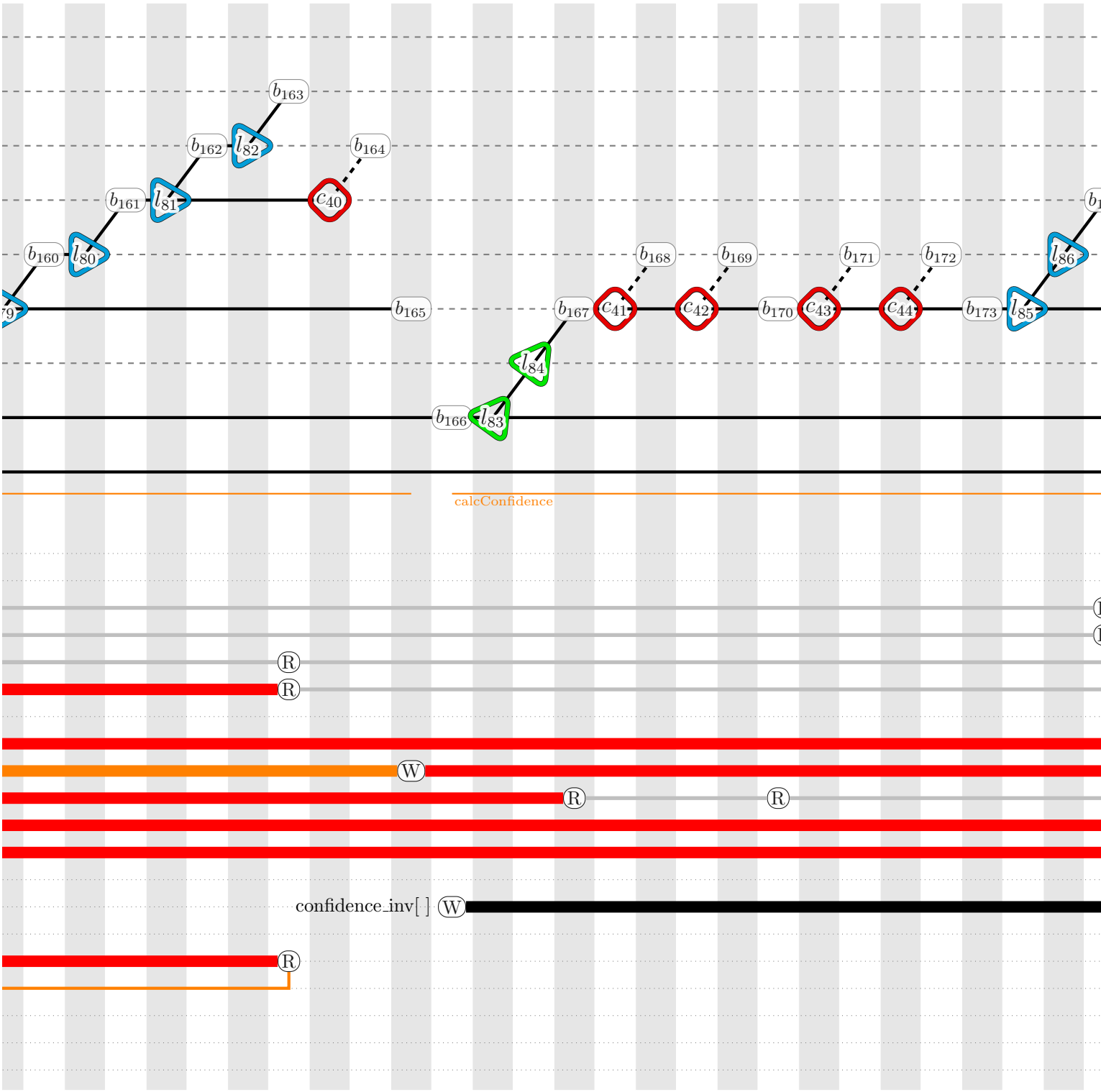


FIGURE B.1 – Représentation spinale du programme *simpleflow* (12/18)

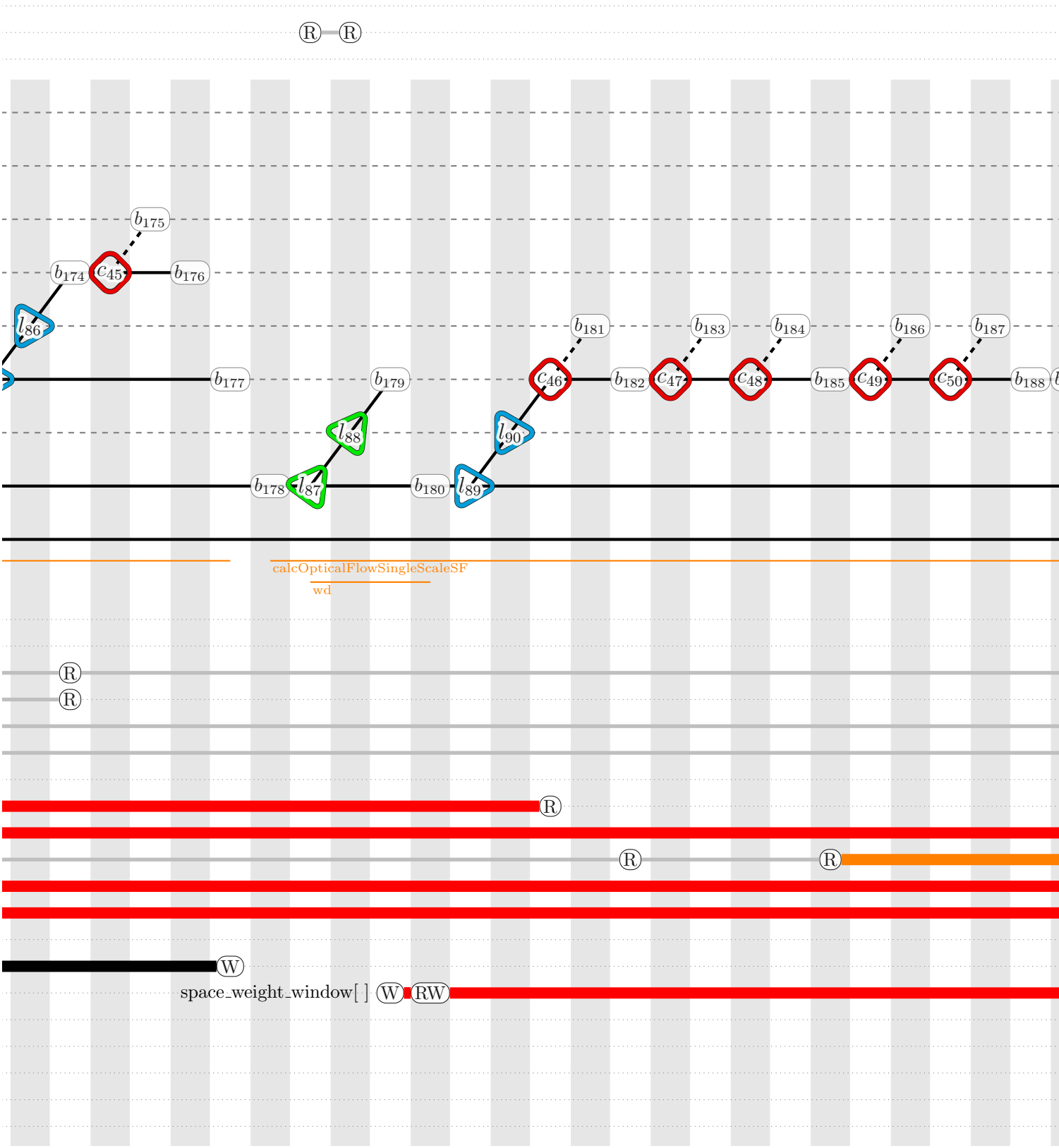


FIGURE B.1 – Représentation spinale du programme *simpleflow* (13/18)

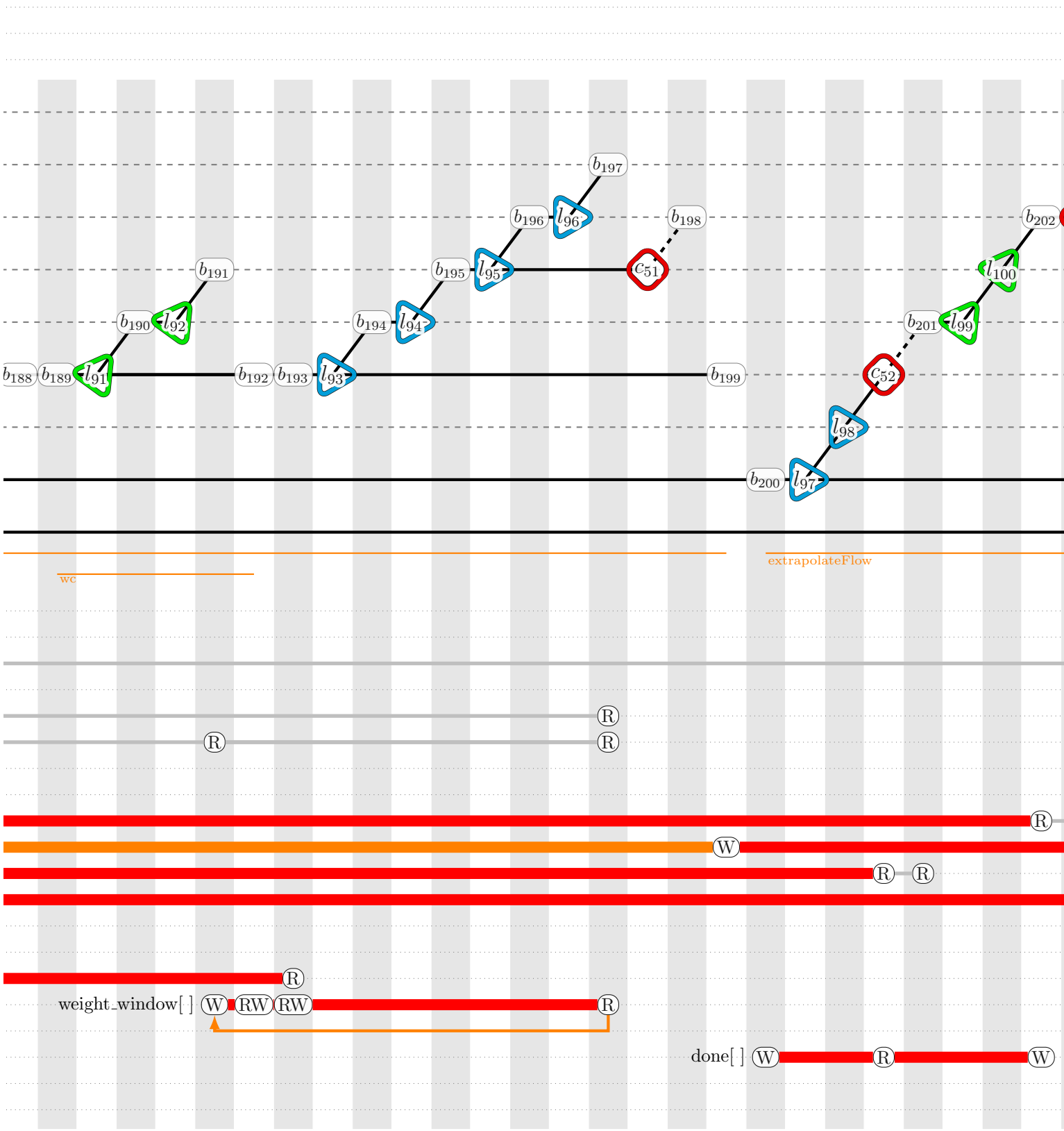


FIGURE B.1 – Représentation spinale du programme *simpleflow* (14/18)

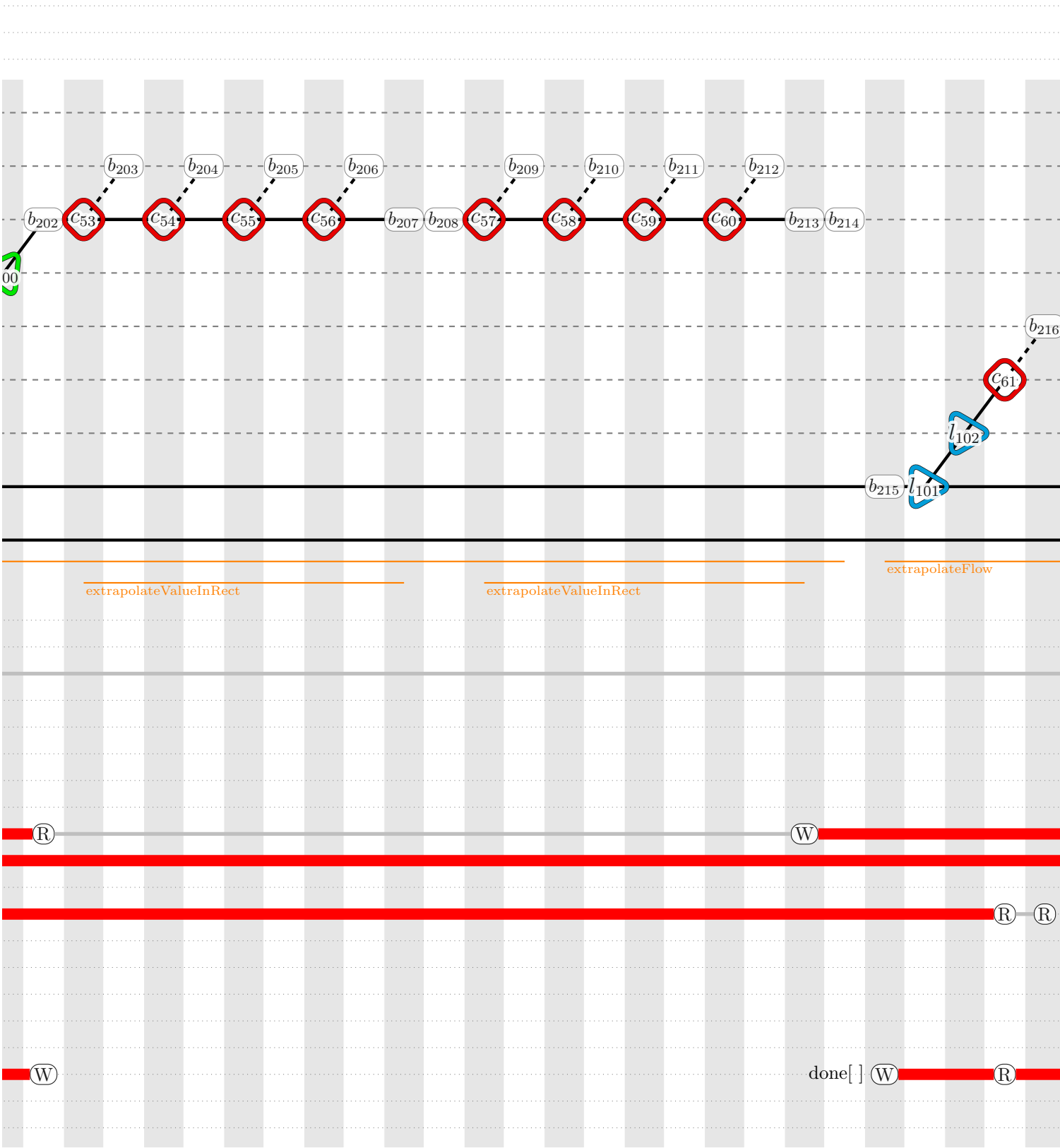


FIGURE B.1 – Représentation spinale du programme *simpleflow* (15/18)

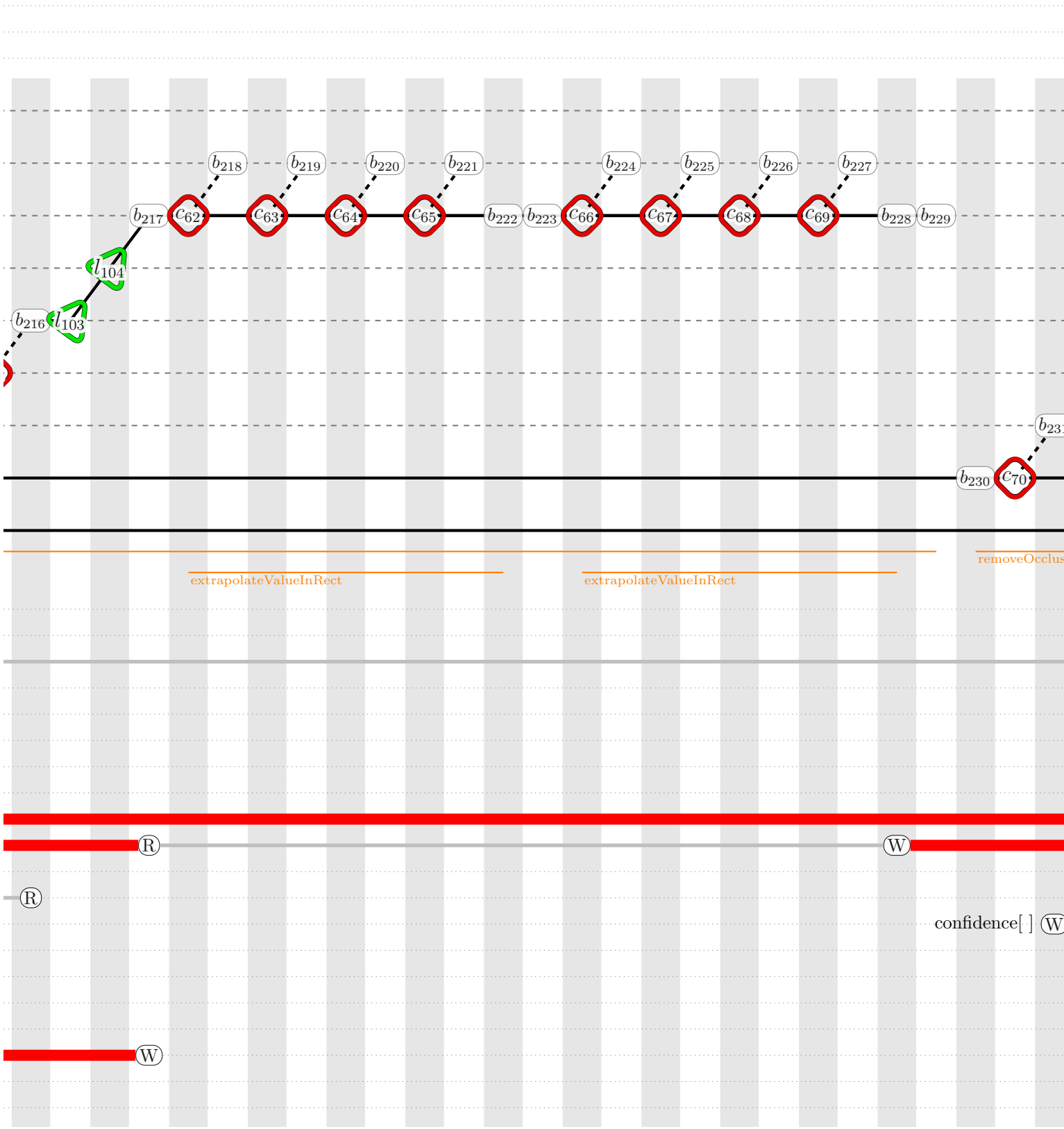


FIGURE B.1 – Représentation spinale du programme *simpleflow* (16/18)

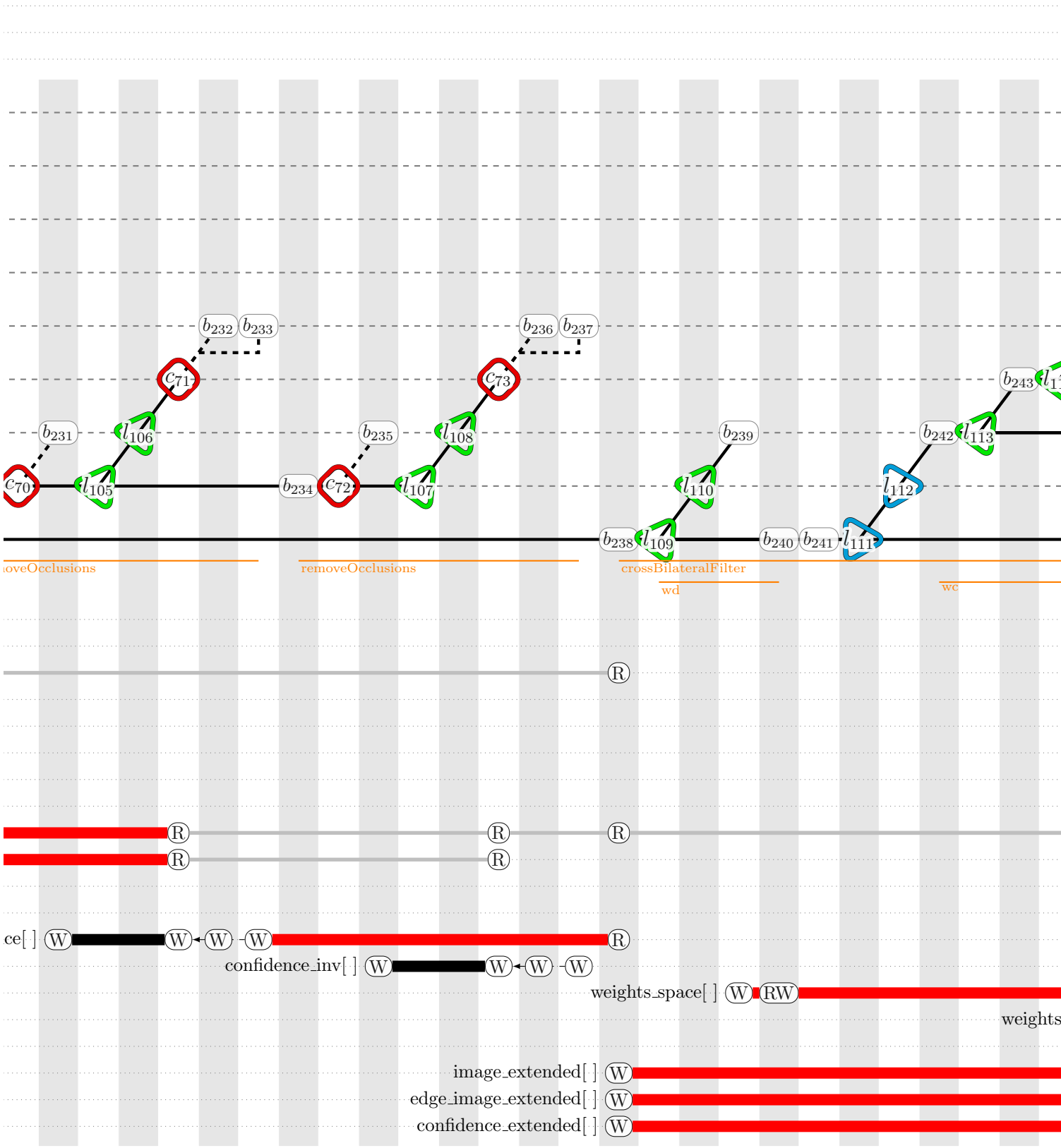


FIGURE B.1 – Représentation spinale du programme *simpleflow* (17/18)

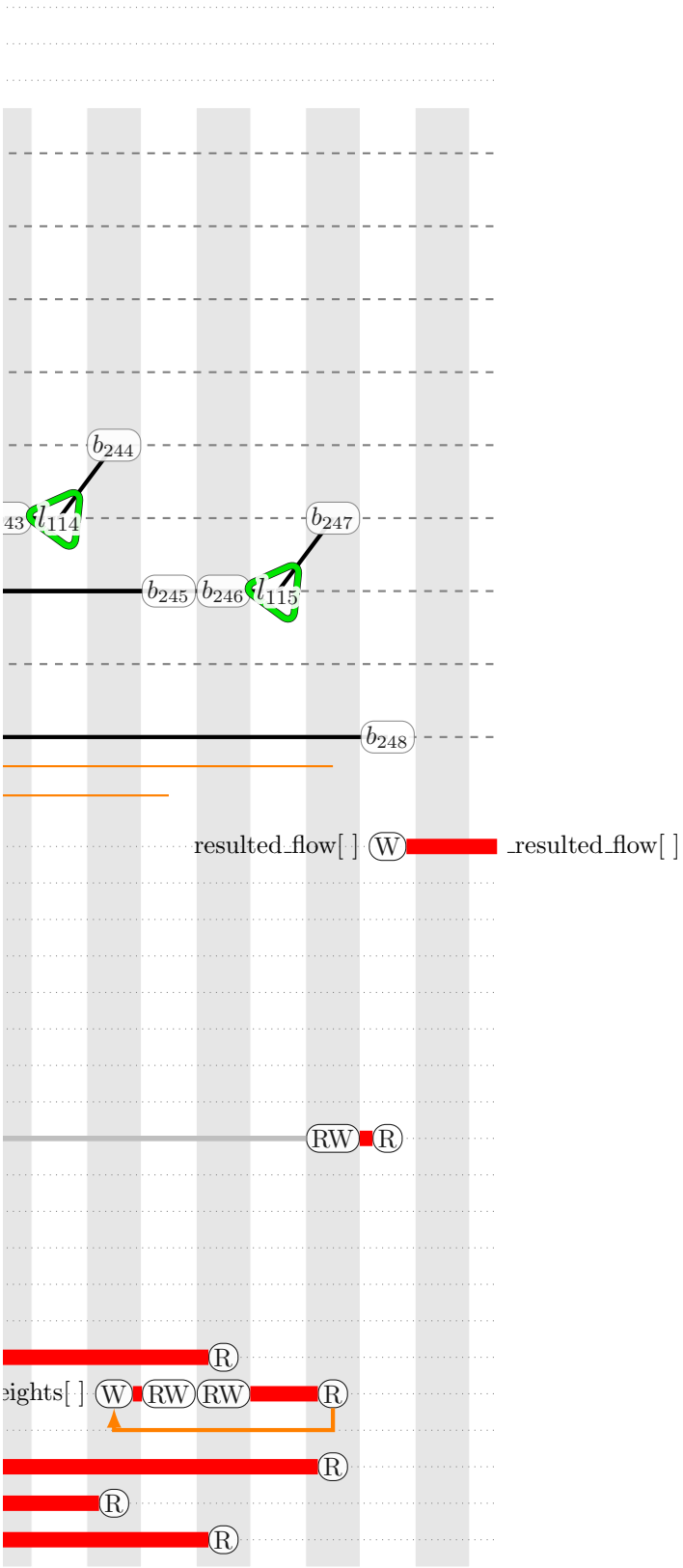


FIGURE B.1 – Représentation spinale du programme *simpleflow* (18/18)

Annexe C

Résultat d'application de la méthodologie : *kernels* GPU pour l'algorithme *simpleFlow*

Sommaire

C.1 calcIrregularityMat

```
492 __global__ void calcIrregularityMat_kernel(const float2* flow, float* irregularity, int radius,const
    ↪  uint2 imgSize){
493         int col = blockIdx.x * blockDim.x + threadIdx.x;
494         int row = blockIdx.y * blockDim.y + threadIdx.y;
495 
496         if(col<imgSize.x && row<imgSize.y){
497                 const int start_row = max(0, row - radius);
498                 const int end_row = min(imgSize.y - 1, row + radius);
499 
500                 const int start_col = max(0, col - radius);
501                 const int end_col = min(imgSize.x - 1, col + radius);
502 
503                 for (int dr = start_row; dr <= end_row; ++dr) {
504                         for (int dc = start_col; dc <= end_col; ++dc) {
505 
506                                 const float diff =
                                    ↪  dist(flow[row*imgSize.x+col],flow[dr*imgSize.x+dc]);
507 
508                                 if (diff > irregularity[row*imgSize.x+col]) {
509                                         irregularity[row*imgSize.x+col] = diff;
510                                 }
511                         }
512                 }
513         }
514 }
```

Listing C.1 – *Kernel* CUDA *calcIrregularityMat*

C.2 calcOpticalFlowSingleScaleSF

```
346 __global__ static void calcOpticalFlowSingleScaleSF_kernel(const uchar3* prev_extended,
347                 const uchar3* next_extended, const uchar* mask, float2* flow,
348                 int averaging_radius, int max_flow, float sigma_color, float* space_weight_window,
349                 uint2 imgSize, uint2 imgExtSize) {
350         int c0 = blockIdx.x * blockDim.x + threadIdx.x;
351         int r0 = blockIdx.y * blockDim.y + threadIdx.y;
352 
353         if(c0<imgSize.x && r0<imgSize.y && mask[r0*imgSize.x+c0]){
354                 const int averaging_radius_2 = (averaging_radius << 1)+1;
355 
356                 int u0 = round(flow[r0*imgSize.x+c0].x);
357                 if (r0 + u0 < 0) {u0 = -r0;}
358                 if (r0 + u0 >= imgSize.y) {u0 = imgSize.y - 1 - r0;}
359                 int v0 = round(flow[r0*imgSize.x+c0].y);
360                 if (c0 + v0 < 0) {v0 = -c0;}
361                 if (c0 + v0 >= imgSize.x) {v0 = imgSize.x - 1 - c0;}
362 
363                 const int top_row_shift = -min(r0 + u0, max_flow);
364                 const int bottom_row_shift = min(imgSize.y - 1 - (r0 + u0), max_flow);
365                 const int left_col_shift = -min(c0 + v0, max_flow);
366                 const int right_col_shift = min(imgSize.x - 1 - (c0 + v0), max_flow);
367 
368                 float min_cost = FLT_MAX;
```

```
369                 float2 best;
370                 best.x = (float) u0;
371                 best.y = (float) v0;
372 
373                 for (int u = top_row_shift; u <= bottom_row_shift; ++u) {
374                         for (int v = left_col_shift; v <= right_col_shift; ++v) {
375                                 float cost = 0;
376                                 for (int r = 0; r < averaging_radius_2; ++r) {
377                                         for (int c = 0; c < averaging_radius_2; ++c) {
378                                                 float weight = -dist(
379                                                         prev_extended[(r0 +
                                                        ↪  averaging_radius)*imgExtSize.x+(c0 +
                                                        ↪  averaging_radius)],
380                                                         prev_extended[(r0+r)*imgExtSize.x+(c0+c)]);
381                                                 weight *= 1.0 / (2.0 * sigma_color * sigma_color);
382                                                 weight = exp(weight);
383                                                 weight *=
                                                        ↪  space_weight_window[r*averaging_radius_2+c];
384                                                 cost += weight
385                                                         * dist( prev_extended[(r0 +
                                                        ↪  r)*imgExtSize.x+(c0 + c)],
386                                                                 next_extended[(r0 + u0 + u +
                                                                ↪  r)*imgExtSize.x+(c0 + v0 + v +
                                                                ↪  c)]);
387                                         }
388                                 }
389                                 if (cost < min_cost) {
390                                         min_cost = cost;
391                                         best.x = (float) (u + u0);
392                                         best.y = (float) (v + v0);
393                                 }
394                         }
395                 }
396                 flow[r0*imgSize.x+c0] = best;
397         }
398 }
399 
400 unsigned int numFile =0;
```

Listing C.2 – *Kernel* CUDA *calcOpticalFlowSingleScaleSF*

C.3 crossBilateralFilter

```
195 __global__ void crossBilateralFilter_kernel(const float2* image, const float2* image_extended,
196                 const uchar3* edge_image_extended, const float* confidence_extended,
197                 float2* dst, float* weights_space,
198                 int d, float sigma_color,
199                 uint2 imgSize, uint2 imgExtSize, bool flag = false) {
200 
201         int col = blockIdx.x * blockDim.x + threadIdx.x;
202         int row = blockIdx.y * blockDim.y + threadIdx.y;
203 
204         if(col<imgSize.x && row<imgSize.y){
205                 float weights_sum = 0.0;
206                 float2 total_sum;
207                 total_sum.x=0.0;
```

```
208                 total_sum.y = 0.0;
209
210                 for (int dr = row, r = 0; r < 2*d+1; ++dr, ++r) {
211                         for (int dc = col, c = 0; c < 2*d+1; ++dc, ++c) {
212
213                                 float weight = -dist(edge_image_extended[(row + d)*imgExtSize.x+(col
                                    ↪  + d)],
214                                         edge_image_extended[dr*imgExtSize.x+dc]);
215                                 weight *= 1.0 / (2.0 * sigma_color * sigma_color);
216                                 weight = exp(weight);
217                                 weight *= confidence_extended[dr*imgExtSize.x+dc];
218                                 weight *= weights_space[r*(2*d+1)+c];
219                                 weights_sum += weight;
220
221                                 total_sum.x += weight * image_extended[dr*imgExtSize.x+dc].x;
222                                 total_sum.y += weight * image_extended[dr*imgExtSize.x+dc].y;
223                         }
224                 }
225
226                 if(flag && fabs(weights_sum) < 1e-9){
227                         dst[row*imgSize.x+col].x = image[row*imgSize.x+col].x;
228                         dst[row*imgSize.x+col].y = image[row*imgSize.x+col].y;
229                 }else{
230                         dst[row*imgSize.x+col].x = total_sum.x / weights_sum;
231                         dst[row*imgSize.x+col].y = total_sum.y / weights_sum;
232                 }
233         }
234 }
```

Listing C.3 – *Kernel* CUDA *crossBilateralFilter*

C.4 dist

```
74  __device__ inline static float dist(const uchar3 p1, const uchar3 p2) {
75          return (float) ((p1.x - p2.x) * (p1.x - p2.x)
76                          + (p1.y - p2.y) * (p1.y - p2.y)
77                          + (p1.z - p2.z) * (p1.z - p2.z));
78  }
79
80  __device__ inline static float dist(const float2 p1, const float2 p2) {
81          return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
82  }
```

Listing C.4 – Fonctions CUDA *dist*

C.5 removeOcclusions

```
109 __global__ void removeOcclusions_kernel(const float2* flow, const float2* flow_inv, float occ_thr,
    ↪  float* confidence, unsigned int size) {
110         int p = blockIdx.x * blockDim.x + threadIdx.x;
111
112         if(p<size){
```

```
113                 float2 flow1 = flow[p];
114                 float2 flow2 = flow_inv[p];
115                 flow2.x = -flow2.x;
116                 flow2.y = -flow2.y;
117 
118                 confidence[p] = (dist(flow1, flow2) > occ_thr);
119         }
120 }
```

Listing C.5 – *Kernel* CUDA *removeOcclusions*

Annexe D

Temps d'exécution de l'algorithme *simpleFlow*

Sommaire

Les temps en noir correpondent au temps d'exécution de fonctions. Les temps en bleu, à un nid de boucle sur CPU, en vert à un *kernel* exécuté sur GPU et enfant en orange les transferts hôte/accélérateur.

D.1 Résultats pour la plateforme Jetson TX1

D.1.1 Temps d'exécution de l'algorithme original

Description	***Début*** (s)	***Fin*** (s)	**Durée** ***(s)***
niveau 0			
calcOpticalFlowSF	*0.000*	*210.089*	**210.089**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.021*	**0.021**
buildPyramidWithResizeMethod	*0.021*	*0.033*	**0.012**
calcOpticalFlowSingleScaleSF	*0.036*	*1.866*	**1.830**
calcOpticalFlowSingleScaleSF	*1.866*	*3.694*	**1.829**
removeOcclusions	*3.694*	*3.696*	**0.001**
removeOcclusions	*3.696*	*3.697*	**0.001**
l_{26}	*3.697*	*124.731*	**121.033**
crossBilateralFilter	*124.731*	*210.036*	**85.305**
niveau 2			
l_0	*0.000*	*0.021*	**0.021**
l_1	*0.021*	*0.033*	**0.012**
wd	*0.036*	*0.036*	**<0.001**
$l_{[4,11]}$	*0.036*	*1.866*	**1.830**
wd	*1.866*	*1.866*	**<0.001**
$l_{[14,21]}$	*1.866*	*3.694*	**1.829**
$l_{[22,23]}$	*3.695*	*3.696*	**0.001**
$l_{[24,25]}$	*3.696*	*3.697*	**0.001**
selectPointsToRecalcFlow	*3.701*	*3.730*	**0.029**
selectPointsToRecalcFlow	*3.730*	*3.757*	**0.028**
upscaleOpticalFlow	*3.757*	*9.055*	**5.297**
upscaleOpticalFlow	*9.055*	*14.585*	**5.530**
calcConfidence	*14.585*	*14.883*	**0.298**
calcOpticalFlowSingleScaleSF	*14.883*	*22.593*	**7.710**
calcConfidence	*22.593*	*22.876*	**0.284**
calcOpticalFlowSingleScaleSF	*22.876*	*30.561*	**7.685**
extrapolateFlow	*30.561*	*30.564*	**0.003**
extrapolateFlow	*30.564*	*30.568*	**0.003**
removeOcclusions	*30.568*	*30.572*	**0.004**
removeOcclusions	*30.572*	*30.576*	**0.004**
selectPointsToRecalcFlow	*30.589*	*30.708*	**0.119**
selectPointsToRecalcFlow	*30.708*	*30.831*	**0.124**
upscaleOpticalFlow	*30.831*	*52.227*	**21.396**
upscaleOpticalFlow	*52.228*	*73.570*	**21.342**
calcConfidence	*73.571*	*74.714*	**1.144**
calcOpticalFlowSingleScaleSF	*74.714*	*98.848*	**24.134**
calcConfidence	*98.848*	*99.990*	**1.142**
calcOpticalFlowSingleScaleSF	*99.990*	*124.630*	**24.640**
extrapolateFlow	*124.630*	*124.667*	**0.037**
extrapolateFlow	*124.667*	*124.702*	**0.036**
removeOcclusions	*124.702*	*124.717*	**0.014**

removeOcclusions	*124.717*	*124.731*	**0.014**
wd	*124.762*	*124.762*	**<0.001**
$l_{[111,115]}$	*124.780*	*210.036*	**85.256**
niveau 3			
$l_{2,3}$	*0.036*	*0.036*	**<0.001**
$l_{12,13}$	*1.866*	*1.866*	**<0.001**
calcIrregularityMat	*3.701*	*3.725*	**0.024**
$l_{31,32}$	*3.726*	*3.730*	**0.004**
calcIrregularityMat	*3.730*	*3.753*	**0.024**
$l_{45,46}$	*3.754*	*3.757*	**0.004**
crossBilateralFilter	*3.757*	*9.051*	**5.294**
crossBilateralFilter	*9.055*	*14.581*	**5.526**
$l_{[69,72]}$	*14.587*	*14.883*	**0.296**
wd	*14.883*	*14.883*	**<0.001**
$l_{[75,82]}$	*14.883*	*22.593*	**7.710**
$l_{[83,86]}$	*22.594*	*22.876*	**0.282**
wd	*22.876*	*22.876*	**<0.001**
$l_{[89,96]}$	*22.876*	*30.561*	**7.685**
$l_{[97,100]}$	*30.561*	*30.564*	**0.003**
$l_{[101,104]}$	*30.565*	*30.568*	**0.003**
$l_{105,106}$	*30.568*	*30.572*	**0.004**
$l_{107,108}$	*30.572*	*30.576*	**0.004**
calcIrregularityMat	*30.589*	*30.687*	**0.098**
$l_{31,32}$	*30.692*	*30.708*	**0.015**
calcIrregularityMat	*30.708*	*30.810*	**0.102**
$l_{45,46}$	*30.814*	*30.831*	**0.017**
crossBilateralFilter	*30.831*	*52.213*	**21.382**
crossBilateralFilter	*52.228*	*73.557*	**21.329**
$l_{[69,72]}$	*73.577*	*74.714*	**1.137**
wd	*74.714*	*74.714*	**<0.001**
$l_{[75,82]}$	*74.714*	*98.848*	**24.134**
$l_{[83,86]}$	*98.855*	*99.990*	**1.136**
wd	*99.990*	*99.990*	**<0.001**
$l_{[89,96]}$	*99.990*	*124.630*	**24.640**
$l_{[97,100]}$	*124.630*	*124.667*	**0.036**
$l_{[101,104]}$	*124.667*	*124.702*	**0.036**
$l_{105,106}$	*124.702*	*124.717*	**0.014**
$l_{107,108}$	*124.717*	*124.731*	**0.014**
$l_{109,110}$	*124.762*	*124.762*	**<0.001**
niveau 4			
$l_{[27,30]}$	*3.701*	*3.725*	**0.023**
$l_{[41,44]}$	*3.730*	*3.753*	**0.023**
wd	*3.759*	*3.759*	**<0.001**
$l_{[57,61]}$	*3.761*	*9.051*	**5.290**
wd	*9.057*	*9.057*	**<0.001**
$l_{[64,68]}$	*9.058*	*14.581*	**5.523**
$l_{73,74}$	*14.883*	*14.883*	**<0.001**
$l_{87,88}$	*22.876*	*22.876*	**<0.001**
$l_{[27,30]}$	*30.591*	*30.687*	**0.096**

$l_{[41,44]}$	*30.710*	*30.810*	**0.101**
wd	*30.839*	*30.839*	**<0.001**
$l_{[57,61]}$	*30.843*	*52.213*	**21.370**
wd	*52.235*	*52.235*	**<0.001**
$l_{[64,68]}$	*52.239*	*73.557*	**21.317**
$l_{73,74}$	*74.714*	*74.714*	**<0.001**
$l_{87,88}$	*99.990*	*99.990*	**<0.001**
niveau 5			
$l_{55,56}$	*3.759*	*3.759*	**<0.001**
$l_{62,63}$	*9.057*	*9.057*	**<0.001**
$l_{55,56}$	*30.839*	*30.839*	**<0.001**
$l_{62,63}$	*52.235*	*52.235*	**<0.001**

TABLE D.1 – Temps d'exécution de l'algorithme *simpleflow* original sur la Tegra X1

D.1.2 Temps d'exécution du placement initial

Description	***Début*** (s)	***Fin*** (s)	**Durée** ***(s)***
niveau 0			
calcOpticalFlowSF	*0.000*	*206.306*	**206.306**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.021*	**0.021**
buildPyramidWithResizeMethod	*0.021*	*0.036*	**0.015**
calcOpticalFlowSingleScaleSF	*0.039*	*1.964*	**1.925**
calcOpticalFlowSingleScaleSF	*1.964*	*3.889*	**1.925**
removeOcclusions	*3.889*	*3.955*	**0.066**
removeOcclusions	*3.955*	*3.960*	**0.004**
l_{26}	*3.960*	*120.741*	**116.782**
crossBilateralFilter	*120.741*	*206.253*	**85.511**
niveau 2			
l_0	*0.000*	*0.021*	**0.021**
l_1	*0.021*	*0.036*	**0.015**
wd	*0.039*	*0.040*	**<0.001**
$l_{[4,11]}$	*0.040*	*1.964*	**1.925**
wd	*1.964*	*1.964*	**<0.001**
$l_{[14,21]}$	*1.964*	*3.889*	**1.925**
Mem. Transf.	*3.951*	*3.954*	**0.003**
removeOcclusions_kernel	*3.954*	*3.955*	**<0.001**
Mem. Transf.	*3.955*	*3.955*	**0.001**
Mem. Transf.	*3.956*	*3.959*	**0.003**
removeOcclusions_kernel	*3.959*	*3.959*	**<0.001**
Mem. Transf.	*3.959*	*3.959*	**0.001**
selectPointsToRecalcFlow	*3.963*	*3.977*	**0.014**
selectPointsToRecalcFlow	*3.977*	*3.990*	**0.013**
upscaleOpticalFlow	*3.990*	*9.327*	**5.337**
upscaleOpticalFlow	*9.327*	*14.602*	**5.275**
calcOpticalFlowSingleScaleSF	*14.604*	*21.972*	**7.368**
calcOpticalFlowSingleScaleSF	*21.974*	*29.342*	**7.368**
extrapolateFlow	*29.342*	*29.345*	**0.003**
extrapolateFlow	*29.345*	*29.348*	**0.003**
removeOcclusions	*29.348*	*29.367*	**0.019**
removeOcclusions	*29.367*	*29.377*	**0.010**
selectPointsToRecalcFlow	*29.390*	*29.441*	**0.051**
selectPointsToRecalcFlow	*29.441*	*29.473*	**0.032**
upscaleOpticalFlow	*29.473*	*50.855*	**21.382**
upscaleOpticalFlow	*50.856*	*72.115*	**21.260**
calcOpticalFlowSingleScaleSF	*72.122*	*96.244*	**24.122**
calcOpticalFlowSingleScaleSF	*96.250*	*120.605*	**24.355**
extrapolateFlow	*120.605*	*120.642*	**0.037**
extrapolateFlow	*120.642*	*120.677*	**0.036**
removeOcclusions	*120.677*	*120.714*	**0.036**
removeOcclusions	*120.714*	*120.741*	**0.028**
wd	*120.773*	*120.773*	**<0.001**

$l_{[111,115]}$	*120.789*	*206.253*	**85.464**
niveau 3			
$l_{2,3}$	*0.039*	*0.039*	**<0.001**
$l_{12,13}$	*1.964*	*1.964*	**<0.001**
calcIrregularityMat	*3.963*	*3.972*	**0.009**
$l_{31,32}$	*3.973*	*3.977*	**0.004**
calcIrregularityMat	*3.977*	*3.986*	**0.009**
$l_{45,46}$	*3.987*	*3.990*	**0.004**
crossBilateralFilter	*3.990*	*9.323*	**5.332**
crossBilateralFilter	*9.327*	*14.598*	**5.270**
wd	*14.604*	*14.604*	**<0.001**
$l_{[75,92]}$	*14.604*	*21.972*	**7.368**
wd	*21.974*	*21.974*	**<0.001**
$l_{[89,96]}$	*21.974*	*29.342*	**7.368**
$l_{[97,100]}$	*29.342*	*29.345*	**0.003**
$l_{[101,104]}$	*29.345*	*29.348*	**0.003**
Mem. Transf.	*29.349*	*29.359*	**0.010**
removeOcclusions_kernel	*29.359*	*29.364*	**0.005**
Mem. Transf.	*29.364*	*29.367*	**0.004**
Mem. Transf.	*29.368*	*29.374*	**0.006**
removeOcclusions_kernel	*29.374*	*29.375*	**0.001**
Mem. Transf.	*29.375*	*29.377*	**0.001**
calcIrregularityMat	*29.390*	*29.421*	**0.031**
$l_{31,32}$	*29.426*	*29.441*	**0.015**
calcIrregularityMat	*29.441*	*29.455*	**0.014**
$l_{45,46}$	*29.458*	*29.473*	**0.015**
crossBilateralFilter	*29.473*	*50.839*	**21.366**
crossBilateralFilter	*50.856*	*72.099*	**21.243**
wd	*72.122*	*72.122*	**<0.001**
$l_{[75,82]}$	*72.122*	*96.244*	**24.122**
wd	*96.250*	*96.250*	**<0.001**
$l_{[89,96]}$	*96.250*	*120.605*	**24.355**
$l_{[97,100]}$	*120.605*	*120.642*	**0.036**
$l_{[101,104]}$	*120.642*	*120.677*	**0.036**
Mem. Transf.	*120.678*	*120.707*	**0.029**
removeOcclusions_kernel	*120.707*	*120.710*	**0.003**
Mem. Transf.	*120.710*	*120.714*	**0.004**
Mem. Transf.	*120.714*	*120.736*	**0.021**
removeOcclusions_kernel	*120.736*	*120.738*	**0.002**
Mem. Transf.	*120.738*	*120.741*	**0.004**
$l_{109,110}$	*120.773*	*120.773*	**<0.001**
niveau 4			
Mem. Transf.	*3.964*	*3.969*	**0.005**
calcIrregularityMat_kernel	*3.969*	*3.972*	**0.003**
Mem. Transf.	*3.972*	*3.972*	**<0.001**
Mem. Transf.	*3.977*	*3.980*	**0.003**
calcIrregularityMat_kernel	*3.980*	*3.986*	**0.005**
Mem. Transf.	*3.986*	*3.986*	**<0.001**
wd	*3.993*	*3.993*	**<0.001**

$l_{[57,61]}$	*3.994*	*9.323*	**5.329**
wd	*9.329*	*9.329*	**<0.001**
$l_{[64,68]}$	*9.331*	*14.598*	**5.267**
$l_{73,74}$	*14.604*	*14.604*	**<0.001**
$l_{87,88}$	*21.974*	*21.974*	**<0.001**
Mem. Transf.	*29.392*	*29.399*	**0.007**
calcIrregularityMat_kernel	*29.399*	*29.420*	**0.022**
Mem. Transf.	*29.420*	*29.421*	**0.001**
Mem. Transf.	*29.443*	*29.446*	**0.003**
calcIrregularityMat_kernel	*29.446*	*29.454*	**0.008**
Mem. Transf.	*29.454*	*29.455*	**0.001**
wd	*29.480*	*29.480*	**<0.001**
$l_{[57,61]}$	*29.485*	*50.839*	**21.355**
wd	*50.863*	*50.863*	**<0.001**
$l_{[64,68]}$	*50.867*	*72.099*	**21.232**
$l_{73,74}$	*72.122*	*72.122*	**<0.001**
$l_{87,88}$	*96.250*	*96.250*	**<0.001**
niveau 5			
$l_{55,56}$	*3.993*	*3.993*	**<0.001**
$l_{62,63}$	*9.329*	*9.329*	**<0.001**
$l_{55,56}$	*29.480*	*29.480*	**<0.001**
$l_{62,63}$	*50.863*	*50.863*	**<0.001**

TABLE D.2 – Temps d'exécution de l'algorithme *simpleflow* suite à son placement initial sur le GPU de la Tegra X1

D.1.3 Temps d'exécution du placement amélioré

Description	*Début* (s)	*Fin* (s)	Durée *(s)*
niveau 0			
calcOpticalFlowSF	*0.000*	*8.367*	**8.367**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.016*	**0.016**
buildPyramidWithResizeMethod	*0.016*	*0.029*	**0.012**
calcOpticalFlowSingleScaleSF	*0.032*	*0.212*	**0.181**
calcOpticalFlowSingleScaleSF	*0.212*	*0.341*	**0.129**
removeOcclusions	*0.341*	*0.345*	**0.004**
removeOcclusions	*0.345*	*0.350*	**0.004**
l_{26}	*0.350*	*6.903*	**6.553**
crossBilateralFilter	*6.903*	*8.315*	**1.412**
niveau 2			
l_0	*0.000*	*0.016*	**0.016**
l_1	*0.016*	*0.029*	**0.012**
wd	*0.032*	*0.032*	**<0.001**
Mem. Transf.	*0.032*	*0.043*	**0.011**
calcOpticalFlowSingleScaleSF_kernel	*0.043*	*0.211*	**0.168**
Mem. Transf.	*0.211*	*0.212*	**0.001**
wd	*0.212*	*0.212*	**<0.001**
Mem. Transf.	*0.212*	*0.216*	**0.004**
calcOpticalFlowSingleScaleSF_kernel	*0.216*	*0.340*	**0.124**
Mem. Transf.	*0.340*	*0.341*	**0.001**
Mem. Transf.	*0.342*	*0.345*	**0.003**
removeOcclusions_kernel	*0.345*	*0.345*	**<0.001**
Mem. Transf.	*0.345*	*0.345*	**<0.001**
Mem. Transf.	*0.346*	*0.349*	**0.003**
removeOcclusions_kernel	*0.349*	*0.349*	**<0.001**
Mem. Transf.	*0.349*	*0.349*	**<0.001**
selectPointsToRecalcFlow	*0.353*	*0.362*	**0.009**
selectPointsToRecalcFlow	*0.362*	*0.369*	**0.008**
upscaleOpticalFlow	*0.369*	*0.463*	**0.094**
upscaleOpticalFlow	*0.464*	*0.561*	**0.098**
calcOpticalFlowSingleScaleSF	*0.564*	*1.112*	**0.548**
calcOpticalFlowSingleScaleSF	*1.114*	*1.660*	**0.546**
extrapolateFlow	*1.660*	*1.665*	**0.005**
extrapolateFlow	*1.665*	*1.669*	**0.004**
removeOcclusions	*1.669*	*1.677*	**0.009**
removeOcclusions	*1.677*	*1.685*	**0.008**
selectPointsToRecalcFlow	*1.698*	*1.728*	**0.030**
selectPointsToRecalcFlow	*1.728*	*1.758*	**0.030**
upscaleOpticalFlow	*1.758*	*2.153*	**0.395**
upscaleOpticalFlow	*2.154*	*2.514*	**0.360**
calcOpticalFlowSingleScaleSF	*2.522*	*4.640*	**2.118**
calcOpticalFlowSingleScaleSF	*4.648*	*6.768*	**2.120**
extrapolateFlow	*6.768*	*6.811*	**0.043**

extrapolateFlow	*6.811*	*6.847*	**0.036**
removeOcclusions	*6.847*	*6.875*	**0.029**
removeOcclusions	*6.875*	*6.903*	**0.028**
wd	*6.930*	*6.930*	**<0.001**
Mem. Transf.	*6.931*	*6.960*	**0.029**
crossBilateralFilter_kernel	*6.960*	*8.305*	**1.345**
Mem. Transf.	*8.305*	*8.314*	**0.010**

niveau 3

$l_{2,3}$	*0.032*	*0.032*	**<0.001**
$l_{12,13}$	*0.212*	*0.212*	**<0.001**
calcIrregularityMat	*0.353*	*0.357*	**0.004**
$l_{31,32}$	*0.358*	*0.362*	**0.004**
calcIrregularityMat	*0.362*	*0.365*	**0.003**
$l_{45,46}$	*0.366*	*0.369*	**0.004**
crossBilateralFilter	*0.369*	*0.460*	**0.091**
crossBilateralFilter	*0.464*	*0.558*	**0.094**
wd	*0.564*	*0.564*	**<0.001**
Mem. Transf.	*0.564*	*0.572*	**0.008**
calcOpticalFlowSingleScaleSF_kernel	*0.572*	*1.110*	**0.538**
Mem. Transf.	*1.110*	*1.112*	**0.002**
wd	*1.114*	*1.114*	**<0.001**
Mem. Transf.	*1.114*	*1.122*	**0.007**
calcOpticalFlowSingleScaleSF_kernel	*1.122*	*1.658*	**0.536**
Mem. Transf.	*1.658*	*1.660*	**0.002**
$l_{[97,100]}$	*1.660*	*1.665*	**0.005**
$l_{[101,104]}$	*1.665*	*1.669*	**0.004**
Mem. Transf.	*1.669*	*1.676*	**0.007**
removeOcclusions_kernel	*1.676*	*1.676*	**0.001**
Mem. Transf.	*1.676*	*1.677*	**0.001**
Mem. Transf.	*1.678*	*1.684*	**0.006**
removeOcclusions_kernel	*1.684*	*1.684*	**0.001**
Mem. Transf.	*1.684*	*1.685*	**0.001**
calcIrregularityMat	*1.698*	*1.710*	**0.012**
$l_{31,32}$	*1.713*	*1.728*	**0.015**
calcIrregularityMat	*1.728*	*1.739*	**0.010**
$l_{45,46}$	*1.743*	*1.758*	**0.015**
crossBilateralFilter	*1.758*	*2.139*	**0.381**
crossBilateralFilter	*2.154*	*2.502*	**0.348**
wd	*2.522*	*2.522*	**<0.001**
Mem. Transf.	*2.523*	*2.541*	**0.019**
calcOpticalFlowSingleScaleSF_kernel	*2.541*	*4.632*	**2.091**
Mem. Transf.	*4.632*	*4.640*	**0.008**
wd	*4.648*	*4.648*	**<0.001**
Mem. Transf.	*4.649*	*4.670*	**0.021**
calcOpticalFlowSingleScaleSF_kernel	*4.670*	*6.759*	**2.089**
Mem. Transf.	*6.759*	*6.768*	**0.009**
$l_{[97,100]}$	*6.769*	*6.811*	**0.042**
$l_{[101,104]}$	*6.811*	*6.847*	**0.036**
Mem. Transf.	*6.847*	*6.868*	**0.021**

removeOcclusions_kernel	*6.868*	*6.871*	**0.003**
Mem. Transf.	*6.871*	*6.875*	**0.004**
Mem. Transf.	*6.876*	*6.897*	**0.021**
removeOcclusions_kernel	*6.897*	*6.899*	**0.002**
Mem. Transf.	*6.899*	*6.903*	**0.004**
$l_{109,110}$	*6.930*	*6.930*	**<0.001**
niveau 4			
Mem. Transf.	*0.353*	*0.356*	**0.002**
calcIrregularityMat_kernel	*0.356*	*0.357*	**0.001**
Mem. Transf.	*0.357*	*0.357*	**<0.001**
Mem. Transf.	*0.362*	*0.363*	**0.001**
calcIrregularityMat_kernel	*0.363*	*0.365*	**0.001**
Mem. Transf.	*0.365*	*0.365*	**<0.001**
wd	*0.371*	*0.371*	**<0.001**
Mem. Transf.	*0.372*	*0.375*	**0.003**
crossBilateralFilter_kernel	*0.375*	*0.460*	**0.085**
Mem. Transf.	*0.460*	*0.460*	**0.001**
wd	*0.466*	*0.466*	**<0.001**
Mem. Transf.	*0.466*	*0.471*	**0.005**
crossBilateralFilter_kernel	*0.471*	*0.557*	**0.086**
Mem. Transf.	*0.557*	*0.558*	**0.001**
$l_{73,74}$	*0.564*	*0.564*	**<0.001**
$l_{87,88}$	*1.114*	*1.114*	**<0.001**
Mem. Transf.	*1.700*	*1.705*	**0.005**
calcIrregularityMat_kernel	*1.705*	*1.709*	**0.005**
Mem. Transf.	*1.709*	*1.710*	**0.001**
Mem. Transf.	*1.729*	*1.732*	**0.003**
calcIrregularityMat_kernel	*1.732*	*1.738*	**0.006**
Mem. Transf.	*1.738*	*1.739*	**0.001**
wd	*1.765*	*1.765*	**<0.001**
Mem. Transf.	*1.766*	*1.773*	**0.007**
crossBilateralFilter_kernel	*1.773*	*2.137*	**0.364**
Mem. Transf.	*2.137*	*2.139*	**0.002**
wd	*2.162*	*2.162*	**<0.001**
Mem. Transf.	*2.162*	*2.172*	**0.010**
crossBilateralFilter_kernel	*2.172*	*2.499*	**0.328**
Mem. Transf.	*2.499*	*2.502*	**0.002**
$l_{73,74}$	*2.522*	*2.522*	**<0.001**
$l_{87,88}$	*4.648*	*4.648*	**<0.001**
niveau 5			
$l_{55,56}$	*0.371*	*0.371*	**<0.001**
$l_{62,63}$	*0.466*	*0.466*	**<0.001**
$l_{55,56}$	*1.765*	*1.765*	**<0.001**
$l_{62,63}$	*2.162*	*2.162*	**<0.001**

TABLE D.3 – Temps d'exécution de l'algorithme simpleflow suite à l'amélioration de la quantité de placement sur le GPU de la Tegra X1

D.2 Résultats pour la plateforme Endicott

D.2.1 Temps d'exécution de l'algorithme original

Description	*Début* (s)	*Fin* (s)	Durée *(s)*
niveau 0			
calcOpticalFlowSF	*0.000*	*45.948*	**45.948**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.050*	**0.050**
buildPyramidWithResizeMethod	*0.050*	*0.054*	**0.004**
calcOpticalFlowSingleScaleSF	*0.055*	*0.677*	**0.622**
calcOpticalFlowSingleScaleSF	*0.677*	*1.299*	**0.622**
removeOcclusions	*1.299*	*1.299*	**<0.001**
removeOcclusions	*1.299*	*1.300*	**<0.001**
l_{26}	*1.300*	*32.448*	**31.148**
crossBilateralFilter	*32.448*	*45.938*	**13.490**
niveau 2			
l_0	*0.000*	*0.050*	**0.050**
l_1	*0.050*	*0.054*	**0.004**
wd	*0.055*	*0.055*	**<0.001**
$l_{[4,11]}$	*0.055*	*0.677*	**0.622**
wd	*0.677*	*0.677*	**<0.001**
$l_{[14,21]}$	*0.677*	*1.299*	**0.622**
$l_{[22,23]}$	*1.299*	*1.299*	**<0.001**
$l_{[24,25]}$	*1.299*	*1.300*	**<0.001**
selectPointsToRecalcFlow	*1.301*	*1.311*	**0.010**
selectPointsToRecalcFlow	*1.311*	*1.321*	**0.010**
upscaleOpticalFlow	*1.321*	*2.243*	**0.922**
upscaleOpticalFlow	*2.243*	*3.165*	**0.922**
calcConfidence	*3.165*	*3.256*	**0.091**
calcOpticalFlowSingleScaleSF	*3.256*	*5.758*	**2.502**
calcConfidence	*5.758*	*5.849*	**0.091**
calcOpticalFlowSingleScaleSF	*5.849*	*8.351*	**2.502**
extrapolateFlow	*8.351*	*8.352*	**0.001**
extrapolateFlow	*8.352*	*8.353*	**0.001**
removeOcclusions	*8.353*	*8.355*	**0.001**
removeOcclusions	*8.355*	*8.356*	**0.001**
selectPointsToRecalcFlow	*8.360*	*8.398*	**0.039**
selectPointsToRecalcFlow	*8.398*	*8.435*	**0.037**
upscaleOpticalFlow	*8.435*	*11.912*	**3.477**
upscaleOpticalFlow	*11.912*	*15.393*	**3.481**
calcConfidence	*15.393*	*15.759*	**0.366**
calcOpticalFlowSingleScaleSF	*15.759*	*23.874*	**8.115**
calcConfidence	*23.874*	*24.240*	**0.366**
calcOpticalFlowSingleScaleSF	*24.240*	*32.416*	**8.177**
extrapolateFlow	*32.416*	*32.427*	**0.011**
extrapolateFlow	*32.427*	*32.437*	**0.010**
removeOcclusions	*32.437*	*32.443*	**0.005**

removeOcclusions	*32.443*	*32.448*	**0.005**
wd	*32.456*	*32.456*	**<0.001**
$l_{[111,115]}$	*32.460*	*45.938*	**13.478**
niveau 3			
$l_{2,3}$	*0.055*	*0.055*	**<0.001**
$l_{12,13}$	*0.677*	*0.677*	**<0.001**
calcIrregularityMat	*1.301*	*1.309*	**0.009**
$l_{31,32}$	*1.310*	*1.311*	**0.001**
calcIrregularityMat	*1.311*	*1.319*	**0.008**
$l_{45,46}$	*1.320*	*1.321*	**0.001**
crossBilateralFilter	*1.321*	*2.242*	**0.921**
crossBilateralFilter	*2.243*	*3.164*	**0.921**
$l_{[69,72]}$	*3.166*	*3.256*	**0.091**
wd	*3.256*	*3.256*	**<0.001**
$l_{[75,82]}$	*3.256*	*5.758*	**2.502**
$l_{[83,86]}$	*5.759*	*5.849*	**0.091**
wd	*5.849*	*5.849*	**<0.001**
$l_{[89,96]}$	*5.849*	*8.351*	**2.502**
$l_{[97,100]}$	*8.351*	*8.352*	**0.001**
$l_{[101,104]}$	*8.352*	*8.353*	**0.001**
$l_{105,106}$	*8.353*	*8.355*	**0.001**
$l_{107,108}$	*8.355*	*8.356*	**0.001**
calcIrregularityMat	*8.360*	*8.392*	**0.032**
$l_{31,32}$	*8.393*	*8.398*	**0.005**
calcIrregularityMat	*8.398*	*8.429*	**0.031**
$l_{45,46}$	*8.430*	*8.435*	**0.005**
crossBilateralFilter	*8.435*	*11.909*	**3.474**
crossBilateralFilter	*11.912*	*15.389*	**3.477**
$l_{[69,72]}$	*15.395*	*15.759*	**0.364**
wd	*15.759*	*15.759*	**<0.001**
$l_{[75,82]}$	*15.759*	*23.874*	**8.115**
$l_{[83,86]}$	*23.876*	*24.240*	**0.364**
wd	*24.240*	*24.240*	**<0.001**
$l_{[89,96]}$	*24.240*	*32.416*	**8.177**
$l_{[97,100]}$	*32.416*	*32.427*	**0.011**
$l_{[101,104]}$	*32.427*	*32.437*	**0.010**
$l_{105,106}$	*32.437*	*32.443*	**0.005**
$l_{107,108}$	*32.443*	*32.448*	**0.005**
$l_{109,110}$	*32.456*	*32.456*	**<0.001**
niveau 4			
$l_{[27,30]}$	*1.301*	*1.309*	**0.009**
$l_{[41,44]}$	*1.311*	*1.319*	**0.008**
wd	*1.322*	*1.322*	**<0.001**
$l_{[57,61]}$	*1.322*	*2.242*	**0.921**
wd	*2.244*	*2.244*	**<0.001**
$l_{[64,68]}$	*2.244*	*3.164*	**0.920**
$l_{73,74}$	*3.256*	*3.256*	**<0.001**
$l_{87,88}$	*5.849*	*5.849*	**<0.001**
$l_{[27,30]}$	*8.360*	*8.392*	**0.032**

$l_{[41,44]}$	*8.398*	*8.429*	**0.031**
wd	*8.437*	*8.437*	**<0.001**
$l_{[57,61]}$	*8.438*	*11.909*	**3.470**
wd	*11.915*	*11.915*	**<0.001**
$l_{[64,68]}$	*11.916*	*15.389*	**3.474**
$l_{73,74}$	*15.759*	*15.759*	**<0.001**
$l_{87,88}$	*24.240*	*24.240*	**<0.001**
niveau 5			
$l_{55,56}$	*1.322*	*1.322*	**<0.001**
$l_{62,63}$	*2.244*	*2.244*	**<0.001**
$l_{55,56}$	*8.437*	*8.437*	**<0.001**
$l_{62,63}$	*11.915*	*11.915*	**<0.001**

TABLE D.4 – Temps d'exécution de l'algorithme *simpleflow* original sur Endicott

D.2.2 Temps d'exécution du placement initial

Description	*Début* (s)	*Fin* (s)	Durée *(s)*
niveau 0			
calcOpticalFlowSF	*0.000*	*45.068*	**45.068**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.051*	**0.051**
buildPyramidWithResizeMethod	*0.051*	*0.055*	**0.004**
calcOpticalFlowSingleScaleSF	*0.056*	*0.679*	**0.623**
calcOpticalFlowSingleScaleSF	*0.679*	*1.301*	**0.622**
removeOcclusions	*1.301*	*1.367*	**0.066**
removeOcclusions	*1.367*	*1.369*	**0.002**
l_{26}	*1.369*	*31.606*	**30.237**
crossBilateralFilter	*31.606*	*45.058*	**13.452**
niveau 2			
l_0	*0.000*	*0.051*	**0.051**
l_1	*0.051*	*0.055*	**0.004**
wd	*0.056*	*0.056*	**<0.001**
$l_{[4,11]}$	*0.056*	*0.679*	**0.623**
wd	*0.679*	*0.679*	**<0.001**
$l_{[14,21]}$	*0.679*	*1.301*	**0.622**
Mem. Transf.	*1.365*	*1.366*	**0.001**
removeOcclusions_kernel	*1.366*	*1.367*	**<0.001**
Mem. Transf.	*1.367*	*1.367*	**<0.001**
Mem. Transf.	*1.368*	*1.368*	**0.001**
removeOcclusions_kernel	*1.368*	*1.369*	**<0.001**
Mem. Transf.	*1.369*	*1.369*	**<0.001**
selectPointsToRecalcFlow	*1.370*	*1.374*	**0.004**
selectPointsToRecalcFlow	*1.374*	*1.378*	**0.004**
upscaleOpticalFlow	*1.378*	*2.296*	**0.918**
upscaleOpticalFlow	*2.296*	*3.216*	**0.920**
calcOpticalFlowSingleScaleSF	*3.217*	*5.723*	**2.506**
calcOpticalFlowSingleScaleSF	*5.723*	*8.230*	**2.507**
extrapolateFlow	*8.230*	*8.231*	**0.001**
extrapolateFlow	*8.231*	*8.232*	**0.001**
removeOcclusions	*8.232*	*8.237*	**0.005**
removeOcclusions	*8.237*	*8.242*	**0.005**
selectPointsToRecalcFlow	*8.246*	*8.260*	**0.014**
selectPointsToRecalcFlow	*8.260*	*8.274*	**0.014**
upscaleOpticalFlow	*8.274*	*11.734*	**3.460**
upscaleOpticalFlow	*11.734*	*15.197*	**3.464**
calcOpticalFlowSingleScaleSF	*15.199*	*23.345*	**8.146**
calcOpticalFlowSingleScaleSF	*23.347*	*31.554*	**8.208**
extrapolateFlow	*31.554*	*31.565*	**0.011**
extrapolateFlow	*31.565*	*31.575*	**0.010**
removeOcclusions	*31.575*	*31.591*	**0.016**
removeOcclusions	*31.591*	*31.606*	**0.015**
wd	*31.614*	*31.614*	**<0.001**

$l_{[111,115]}$	*31.619*	*45.058*	**13.439**
niveau 3			
$l_{2,3}$	*0.056*	*0.056*	**<0.001**
$l_{12,13}$	*0.679*	*0.679*	**<0.001**
calcIrregularityMat	*1.370*	*1.373*	**0.002**
$l_{31,32}$	*1.373*	*1.374*	**0.001**
calcIrregularityMat	*1.374*	*1.377*	**0.002**
$l_{45,46}$	*1.377*	*1.378*	**0.001**
crossBilateralFilter	*1.378*	*2.296*	**0.918**
crossBilateralFilter	*2.296*	*3.216*	**0.919**
wd	*3.217*	*3.217*	**<0.001**
$l_{[75,92]}$	*3.217*	*5.723*	**2.506**
wd	*5.723*	*5.723*	**<0.001**
$l_{[89,96]}$	*5.723*	*8.230*	**2.507**
$l_{[97,100]}$	*8.230*	*8.231*	**0.001**
$l_{[101,104]}$	*8.231*	*8.232*	**0.001**
Mem. Transf.	*8.233*	*8.236*	**0.003**
removeOcclusions_kernel	*8.236*	*8.236*	**0.001**
Mem. Transf.	*8.236*	*8.237*	**0.001**
Mem. Transf.	*8.238*	*8.241*	**0.003**
removeOcclusions_kernel	*8.241*	*8.241*	**<0.001**
Mem. Transf.	*8.241*	*8.242*	**0.001**
calcIrregularityMat	*8.246*	*8.253*	**0.008**
$l_{31,32}$	*8.254*	*8.260*	**0.006**
calcIrregularityMat	*8.260*	*8.268*	**0.008**
$l_{45,46}$	*8.268*	*8.274*	**0.005**
crossBilateralFilter	*8.274*	*11.730*	**3.456**
crossBilateralFilter	*11.734*	*15.194*	**3.460**
wd	*15.199*	*15.199*	**<0.001**
$l_{[75,92]}$	*15.199*	*23.345*	**8.146**
wd	*23.347*	*23.347*	**<0.001**
$l_{[89,96]}$	*23.347*	*31.554*	**8.208**
$l_{[97,100]}$	*31.555*	*31.565*	**0.010**
$l_{[101,104]}$	*31.565*	*31.575*	**0.010**
Mem. Transf.	*31.576*	*31.587*	**0.011**
removeOcclusions_kernel	*31.587*	*31.588*	**0.001**
Mem. Transf.	*31.588*	*31.591*	**0.003**
Mem. Transf.	*31.592*	*31.602*	**0.010**
removeOcclusions_kernel	*31.602*	*31.603*	**0.001**
Mem. Transf.	*31.603*	*31.606*	**0.003**
$l_{109,110}$	*31.614*	*31.614*	**<0.001**
niveau 4			
Mem. Transf.	*1.371*	*1.371*	**0.001**
calcIrregularityMat_kernel	*1.371*	*1.372*	**0.001**
Mem. Transf.	*1.372*	*1.373*	**<0.001**
Mem. Transf.	*1.374*	*1.375*	**0.001**
calcIrregularityMat_kernel	*1.375*	*1.376*	**0.001**
Mem. Transf.	*1.376*	*1.377*	**<0.001**
wd	*1.379*	*1.379*	**<0.001**

$l_{[57,61]}$	*1.379*	*2.296*	**0.917**
wd	*2.297*	*2.297*	**<0.001**
$l_{[64,68]}$	*2.297*	*3.216*	**0.918**
$l_{73,74}$	*3.217*	*3.217*	**<0.001**
$l_{87,88}$	*5.723*	*5.723*	**<0.001**
Mem. Transf.	*8.246*	*8.248*	**0.002**
calcIrregularityMat_kernel	*8.248*	*8.253*	**0.004**
Mem. Transf.	*8.253*	*8.253*	**0.001**
Mem. Transf.	*8.261*	*8.263*	**0.002**
calcIrregularityMat_kernel	*8.263*	*8.267*	**0.004**
Mem. Transf.	*8.267*	*8.268*	**0.001**
wd	*8.276*	*8.276*	**<0.001**
$l_{[57,61]}$	*8.277*	*11.730*	**3.453**
wd	*11.736*	*11.736*	**<0.001**
$l_{[64,68]}$	*11.737*	*15.194*	**3.456**
$l_{73,74}$	*15.199*	*15.199*	**<0.001**
$l_{87,88}$	*23.347*	*23.347*	**<0.001**
niveau 5			
$l_{55,56}$	*1.379*	*1.379*	**<0.001**
$l_{62,63}$	*2.297*	*2.297*	**<0.001**
$l_{55,56}$	*8.276*	*8.276*	**<0.001**
$l_{62,63}$	*11.736*	*11.736*	**<0.001**

TABLE D.5 – Temps d'exécution de l'algorithme *simpleflow* suite à son placement initial sur le GPU d'Endicott

D.2.3 Temps d'exécution du placement amélioré

Description	*Début* (s)	*Fin* (s)	Durée *(s)*
niveau 0			
calcOpticalFlowSF	*0.000*	*9.125*	**9.125**
niveau 1			
buildPyramidWithResizeMethod	*0.000*	*0.043*	**0.043**
buildPyramidWithResizeMethod	*0.043*	*0.047*	**0.004**
calcOpticalFlowSingleScaleSF	*0.048*	*0.191*	**0.144**
calcOpticalFlowSingleScaleSF	*0.191*	*0.335*	**0.143**
removeOcclusions	*0.335*	*0.337*	**0.002**
removeOcclusions	*0.337*	*0.339*	**0.002**
l_{26}	*0.339*	*7.523*	**7.184**
crossBilateralFilter	*7.523*	*9.116*	**1.593**
niveau 2			
l_0	*0.000*	*0.043*	**0.043**
l_1	*0.043*	*0.047*	**0.004**
wd	*0.048*	*0.048*	**<0.001**
Mem. Transf.	*0.049*	*0.050*	**0.001**
calcOpticalFlowSingleScaleSF_kernel	*0.050*	*0.190*	**0.141**
Mem. Transf.	*0.190*	*0.191*	**<0.001**
wd	*0.191*	*0.191*	**<0.001**
Mem. Transf.	*0.192*	*0.193*	**0.001**
calcOpticalFlowSingleScaleSF_kernel	*0.193*	*0.334*	**0.141**
Mem. Transf.	*0.334*	*0.334*	**<0.001**
Mem. Transf.	*0.335*	*0.336*	**0.001**
removeOcclusions_kernel	*0.336*	*0.336*	**<0.001**
Mem. Transf.	*0.336*	*0.337*	**<0.001**
Mem. Transf.	*0.338*	*0.338*	**0.001**
removeOcclusions_kernel	*0.338*	*0.339*	**<0.001**
Mem. Transf.	*0.339*	*0.339*	**<0.001**
selectPointsToRecalcFlow	*0.340*	*0.344*	**0.004**
selectPointsToRecalcFlow	*0.344*	*0.348*	**0.004**
upscaleOpticalFlow	*0.348*	*0.450*	**0.102**
upscaleOpticalFlow	*0.450*	*0.552*	**0.102**
calcOpticalFlowSingleScaleSF	*0.552*	*1.149*	**0.597**
calcOpticalFlowSingleScaleSF	*1.150*	*1.752*	**0.602**
extrapolateFlow	*1.752*	*1.753*	**0.001**
extrapolateFlow	*1.753*	*1.753*	**0.001**
removeOcclusions	*1.753*	*1.758*	**0.005**
removeOcclusions	*1.758*	*1.763*	**0.005**
selectPointsToRecalcFlow	*1.766*	*1.780*	**0.014**
selectPointsToRecalcFlow	*1.780*	*1.793*	**0.013**
upscaleOpticalFlow	*1.793*	*2.197*	**0.404**
upscaleOpticalFlow	*2.197*	*2.602*	**0.405**
calcOpticalFlowSingleScaleSF	*2.604*	*5.034*	**2.430**
calcOpticalFlowSingleScaleSF	*5.035*	*7.472*	**2.437**
extrapolateFlow	*7.472*	*7.483*	**0.011**

extrapolateFlow	*7.483*	*7.493*	**0.010**
removeOcclusions	*7.493*	*7.508*	**0.015**
removeOcclusions	*7.508*	*7.523*	**0.015**
wd	*7.531*	*7.531*	**<0.001**
Mem. Transf.	*7.532*	*7.547*	**0.016**
crossBilateralFilter_kernel	*7.547*	*9.110*	**1.563**
Mem. Transf.	*9.110*	*9.115*	**0.005**

niveau 3

$l_{2,3}$	*0.048*	*0.048*	**<0.001**
$l_{12,13}$	*0.191*	*0.191*	**<0.001**
calcIrregularityMat	*0.340*	*0.343*	**0.002**
$l_{31,32}$	*0.343*	*0.344*	**0.001**
calcIrregularityMat	*0.344*	*0.346*	**0.002**
$l_{45,46}$	*0.347*	*0.348*	**0.001**
crossBilateralFilter	*0.348*	*0.449*	**0.101**
crossBilateralFilter	*0.450*	*0.551*	**0.101**
wd	*0.552*	*0.552*	**<0.001**
Mem. Transf.	*0.553*	*0.556*	**0.003**
calcOpticalFlowSingleScaleSF_kernel	*0.556*	*1.147*	**0.591**
Mem. Transf.	*1.147*	*1.149*	**0.001**
wd	*1.150*	*1.150*	**<0.001**
Mem. Transf.	*1.150*	*1.153*	**0.003**
calcOpticalFlowSingleScaleSF_kernel	*1.153*	*1.750*	**0.596**
Mem. Transf.	*1.750*	*1.751*	**0.001**
$l_{[97,100]}$	*1.752*	*1.753*	**0.001**
$l_{[101,104]}$	*1.753*	*1.753*	**0.001**
Mem. Transf.	*1.754*	*1.757*	**0.003**
removeOcclusions_kernel	*1.757*	*1.757*	**<0.001**
Mem. Transf.	*1.757*	*1.758*	**0.001**
Mem. Transf.	*1.759*	*1.761*	**0.003**
removeOcclusions_kernel	*1.761*	*1.762*	**<0.001**
Mem. Transf.	*1.762*	*1.762*	**0.001**
calcIrregularityMat	*1.766*	*1.773*	**0.008**
$l_{31,32}$	*1.774*	*1.780*	**0.005**
calcIrregularityMat	*1.780*	*1.787*	**0.007**
$l_{45,46}$	*1.788*	*1.793*	**0.005**
crossBilateralFilter	*1.793*	*2.194*	**0.401**
crossBilateralFilter	*2.197*	*2.599*	**0.402**
wd	*2.604*	*2.604*	**<0.001**
Mem. Transf.	*2.605*	*2.615*	**0.010**
calcOpticalFlowSingleScaleSF_kernel	*2.615*	*5.028*	**2.413**
Mem. Transf.	*5.028*	*5.033*	**0.005**
wd	*5.035*	*5.035*	**<0.001**
Mem. Transf.	*5.036*	*5.046*	**0.010**
calcOpticalFlowSingleScaleSF_kernel	*5.046*	*7.466*	**2.420**
Mem. Transf.	*7.466*	*7.472*	**0.005**
$l_{[97,100]}$	*7.472*	*7.483*	**0.011**
$l_{[101,104]}$	*7.483*	*7.493*	**0.010**
Mem. Transf.	*7.494*	*7.504*	**0.010**

removeOcclusions_kernel	*7.504*	*7.505*	**0.001**
Mem. Transf.	*7.505*	*7.508*	**0.003**
Mem. Transf.	*7.509*	*7.519*	**0.010**
removeOcclusions_kernel	*7.519*	*7.520*	**0.001**
Mem. Transf.	*7.520*	*7.523*	**0.003**
$l_{109,110}$	*7.531*	*7.531*	**<0.001**
niveau 4			
Mem. Transf.	*0.340*	*0.341*	**0.001**
calcIrregularityMat_kernel	*0.341*	*0.342*	**0.001**
Mem. Transf.	*0.342*	*0.343*	**<0.001**
Mem. Transf.	*0.344*	*0.345*	**0.001**
calcIrregularityMat_kernel	*0.345*	*0.346*	**0.001**
Mem. Transf.	*0.346*	*0.346*	**<0.001**
wd	*0.349*	*0.349*	**<0.001**
Mem. Transf.	*0.350*	*0.351*	**0.001**
crossBilateralFilter_kernel	*0.351*	*0.448*	**0.097**
Mem. Transf.	*0.448*	*0.449*	**<0.001**
wd	*0.451*	*0.451*	**<0.001**
Mem. Transf.	*0.452*	*0.453*	**0.002**
crossBilateralFilter_kernel	*0.453*	*0.550*	**0.097**
Mem. Transf.	*0.550*	*0.551*	**<0.001**
$l_{73,74}$	*0.552*	*0.552*	**<0.001**
$l_{87,88}$	*1.150*	*1.150*	**<0.001**
Mem. Transf.	*1.767*	*1.769*	**0.002**
calcIrregularityMat_kernel	*1.769*	*1.773*	**0.004**
Mem. Transf.	*1.773*	*1.773*	**0.001**
Mem. Transf.	*1.780*	*1.782*	**0.002**
calcIrregularityMat_kernel	*1.782*	*1.786*	**0.004**
Mem. Transf.	*1.786*	*1.787*	**0.001**
wd	*1.795*	*1.795*	**<0.001**
Mem. Transf.	*1.796*	*1.801*	**0.004**
crossBilateralFilter_kernel	*1.801*	*2.192*	**0.391**
Mem. Transf.	*2.192*	*2.193*	**0.001**
wd	*2.200*	*2.200*	**<0.001**
Mem. Transf.	*2.201*	*2.205*	**0.004**
crossBilateralFilter_kernel	*2.205*	*2.597*	**0.392**
Mem. Transf.	*2.597*	*2.598*	**0.001**
$l_{73,74}$	*2.604*	*2.604*	**<0.001**
$l_{87,88}$	*5.035*	*5.035*	**<0.001**
niveau 5			
$l_{55,56}$	*0.349*	*0.349*	**<0.001**
$l_{62,63}$	*0.451*	*0.451*	**<0.001**
$l_{55,56}$	*1.795*	*1.795*	**<0.001**
$l_{62,63}$	*2.200*	*2.200*	**<0.001**

TABLE D.6 – Temps d'exécution de l'algorithme simpleflow suite à l'amélioration de la quantité de placement sur le GPU d'Endicott

Table des figures

Liste des tableaux

Liste des codes source

Acronymes

AMD Advanced Micro Devices. 8, 10, 11, 13, 14, 17, 18, 21, 38, 45, 61, 62, 89, 155

ANOVA Analysis Of Variance. 81

API Application Programming Interface. 4, 15–18, 20, 25, 27, 28, 38, 46, 87

APU Accelerated Processing Unit. 11, 21, 89, 155

AST Abstract Syntaxic Tree. 24, 31, 38, 50, 92

ATI Array Technologies Incorporated. 10, 17

AVX Advanced Vector Extensions. 8, 10, 20

CAST C Abstract Syntax Tree. 38, 42

CLOOG Chunky LOOp Generator. 31, 33

CPU Central Processing Unit. 8, 10, 11, 14, 16–19, 23–28, 31–39, 41, 45, 57, 62, 64, 89–93, 101, 104–108, 116, 117, 142, 144, 146, 149, 151, 155, 156, 158, 199, 219

CTM Close To Metal. 17

CUB Cuda UnBound. 80

CUDA Compute Unified Device Architecture. 12, 15–20, 26–35, 37–42, 79, 80, 85, 86, 88, 89, 91–93, 97, 99, 118, 122–125, 127, 143–147, 149, 151, 152

DARPA Defense Advanced Research Projects Agency. 19

DMA Direct Memory Access. 145

DSL Domain Specific Language. 19, 20, 25, 44, 156

DSP Digital Signal Processor. 15

FPGA Field-Programmable Gate Array. 15

GCN Graphics Core Next. 10, 11, 13, 61

GLSL openGL Shading Language. 15, 16, 19

GPC Graphics Processing Cluster. 12, 14, 59

GPGPU General-purpose Processing on Graphics Processing Units. 1, 8, 10, 11, 15, 17, 19, 21, 155

GPU Graphics Processing Unit. v, vii, 1–5, 8, 10–21, 23–42, 44–46, 48, 57–69, 71–73, 75–83, 85–93, 95–99, 101–103, 105–111, 114–119, 121–124, 126–129, 136, 137, 142–149, 151–153, 155–158, 173, 199, 205, 208, 214, 217, 219–221, 223, 240

HD High Definition. 102, 119

HLSL High Level Shader Language. 16

HMPP Hybrid Multicore Parallel Programming. 25, 26
HPC High Performance Computing. 10–12, 24, 155, 240

IGP Integrated Graphics Processor. 10
ILP Instruction Level Parallelism. 78, 143, 152
Intel GMA Intel Graphics Media Accelerator. 10
Intel IPL Intel Image Processing Library. 19
Intel IPP Intel Integrated Performance Primitive. 19, 20
IR Internal Representation. 41
ISA Instruction Set Architecture. 4, 16, 41, 77, 125, 127
ISL Integer Set Library. 33

JIT Just In Time. 16, 17, 36, 37

LIDAR LIght Detection And Ranging. 1

MIMD Multiple Instructions on Multiple Data. 14, 44, 144, 152, 157
MMX MultiMedia eXtension. 8
MPI Message Passing Interface. 31
MPPA Multi-Purpose Processor Array. 155
MSI Modified Shared Invalid. 35

NASA National Aeronautics and Space Administration. 118
NPP Nvidia Performance Primitive. 19, 20
NUMA Non Uniform Memory Access. 145
NVCC NVidia Cuda Compiler. 16, 41, 79, 102, 118, 123, 125, 127, 128
NVPTX NVidia Parallel Thread eXecution. 16

OpenACC Open ACCelerators. 30
OpenCL Open Computing Language. 14–20, 30–38, 42
OpenCLIPP OpenCL Image Processing Primitives. 20
OpenCV Open Computer Vision. 19, 21, 38, 46, 48, 49, 87, 100, 102, 119, 125, 157
OpenGL Open Graphics Library. 15–20
OpenGL ES OpenGL for Embedded System. 15
OpenGL SC OpenGL for Safety Critical applications. 15
OpenMP Open Multi-Processing. 25, 27–33, 35, 39, 42
OS Operating System. 16, 62, 155

PCIe Peripheral Component Interconnect express. 99, 145, 155
PET Polyhedral Extraction Tool. 33
PGCD Plus Grand Commun Diviseur. 54
PIPS Programming Integrated Parallel System. 31, 32
PPCG Polyhedral Parallel Code Generator. 32, 44, 124

PTX Parallel Thread eXecution. 16, 41, 125

RISC Reduced Instruction Set Computer. 11

SCOP Static COntrol Part. 24, 41

SFU Special Function Unit. 12

SIMD Single Instruction on Multiple Data. 8, 10, 11, 14, 21, 44

SIMT Single Instruction Multiple Thread. 8, 11, 19, 21, 86

SLAM Simultaneous Localization And Mapping. 101

SM Streaming Multiprocessor. 12–15, 65, 83, 98–100, 111, 122, 123, 126, 129, 130, 143, 146, 149, 151, 152, 221

SMM Maxwell Streaming Multiprocessor. 96–99

SMP Symmetric MultiProcessing. 13, 145

SMT Simultaneous multithreading. 146

SMX Next Generation Streaming Multiprocessor. 96, 99, 149, 151, 152

SOC System On Chip. 10–13, 89, 96, 97, 116, 117, 119, 123–126, 155, 158

SSE Streaming SIMD Extensions. 8

STL Standard Template Library. 35, 37

TDP Thermal Design Power. 96, 97, 99

TPC Texture Processor Cluster. 12

UVA Unified Virtual Addressing. 89

VLIW Very Long Instruction Word. 10

Bibliographie

[1] Complexité en espace. https ://fr.wikipedia.org/wiki/Complexit%C3%A9_en_espace.

[2] Dépôt des contributions à OpenCV. https ://github.com/opencv/opencv_contrib.

[3] GNU gprof. https ://sourceware.org/binutils/docs/gprof/.

[4] Intel Integrated Performance Primitives. https ://software.intel.com/en-us/intel-ipp.

[5] Intel® VTune™ Amplifier. https ://software.intel.com/en-us/intel-vtune-amplifier-xe.

[6] Khronos OpenVX. http ://www.khronos.org/openvx.

[7] NVIDIA CUB. https ://nvlabs.github.io/cub/.

[8] Parallel computing toolbox. https ://fr.mathworks.com/products/parallel-computing/.

[9] SYCL. https ://www.khronos.org/sycl.

[10] Top500. https ://www.top500.org.

[11] PGI Accelerator : The portland group, pgi fortran and c accelarator programming model. https ://www.pgroup.com/resources/accel.htm, 2009.

[12] Alfred V Aho, Ravi Sethi et Jeffrey D Ullman : Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[13] M Akhloufi et A Campagna : Openclipp : Opencl integrated performance primitives library for computer vision applications. *In Proc. SPIE Electronic Imaging*, pages 25–31, 2014.

[14] Markus Ålind, Mattias V Eriksson et Christoph W Kessler : Blocklib : a skeleton library for cell broadband engine. *In Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14. ACM, 2008.

[15] Yannick Allusse, Patrick Horain, Ankit Agarwal et Cindula Saipriyadarshan : Gpucv : an opensource gpu-accelerated framework forimage processing and computer vision. *In Proceedings of the 16th ACM international conference on Multimedia*, pages 1089–1092. ACM, 2008.

[16] Mehdi Amini : *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD Thesis, Ecole Nationale Supérieure des Mines de Paris, décembre 2012.

[17] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell et Pierre Villalon : PIPS is not (just) polyhedral software adding GPU code generation in PIPS. *In First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, 2011.

[18] Mehdi Amini, Fabien Coelho, François Irigoin et Ronan Keryell : Static compilation analysis for host-accelerator communication optimization. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 237–251. Springer, 2011.

[19] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan et Pierre Villalon : Par4all : From convex array regions to heterogeneous computing. *In IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[20] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault et Raymond Namyst : Data-aware task scheduling on multi-accelerator based platforms. *In Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 291–298. IEEE, 2010.

[21] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier : StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23:187–198, février 2011.

[22] Utpal Banerjee : *Dependence analysis*, volume 3. Springer Science & Business Media, 1997.

[23] Muthu Baskaran, J Ramanujam et P Sadayappan : Automatic C-to-CUDA code generation for affine programs. *In Compiler Construction*, pages 244–263. Springer, 2010.

[24] Cédric Bastoul : Code generation in the polyhedral model is easier than you think. *In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[25] Nathan Bell et Jared Hoberock : Thrust : A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

[26] Michel Bilodeau *et al.* : Freia : Framework for embedded image applications, 2008.

[27] Uday Bondhugula, Vinayaka Bandishti et Irshad Pananilath : Diamond tiling : Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2017.

[28] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev et Ponnuswamy Sadayappan : Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *In International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[29] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam et Ponnuswamy Sadayappan : A practical automatic polyhedral parallelizer and locality optimizer. *In ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[30] Ian Buck, T Foley, D Horn, J Sugerman, P Hanrahan, M Houston et K Fatahalian : Brookgpu, 2003.

[31] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston et Pat Hanrahan : Brook for gpus : stream computing on graphics hardware. *In ACM transactions on graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[32] Fabien Coelho et François Irigoin : Api-compiling for image hardware accelerators technical report–mines paristech a/500/cri. 2012.

[33] Sylvain Collange, David Defour et Arnaud Tisserand : Power consumption of gpus from a software perspective. *In International Conference on Computational Science*, pages 914–923. Springer, 2009.

[34] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee et Adriana Susnea : Nova : A functional language for data parallelism. *In Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 8. ACM, 2014.

[35] Jay LT Cornwall, Lee Howes, Paul HJ Kelly, Phil Parsonage et Bruno Nicoletti : High-performance simt code generation in an active visual effects library. *In Proceedings of the 6th ACM conference on Computing frontiers*, pages 175–184. ACM, 2009.

[36] Béatrice Creusillet et François Irigoin : Exact versus approximate array region analyses. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 86–100. Springer, 1996.

[37] Béatrice Creusillet et François Irigoin : Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[38] Béatrice Creusillet, Ronan Keryell, Stéphanie Even, Serge Guelton et François Irigoin : Par4all : Auto-parallelizing c and fortran for the cuda architecture. 2009.

[39] Rommel Cruz, Lucia Drummond, Esteban Clua et Cristiana Bentes : Analyzing and estimating the performance of concurrent kernels execution on gpus. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

[40] George B Dantzig : Fourier-motzkin elimination and its dual. Rapport technique, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[41] George B Dantzig, Alex Orden, Philip Wolfe *et al.* : The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[42] Alain Darte : On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[43] Alain Darte et Frédéric Vivien : On the optimality of allen and kennedy's algorithm for parallelism extraction in nested loops. *In European Conference on Parallel Processing*, pages 379–388. Springer, 1996.

[44] Usman Dastgeer, Johan Enmyren et Christoph W Kessler : Auto-tuning skepu : a multi-backend skeleton programming framework for multi-gpu systems. *In Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 25–32. ACM, 2011.

[45] Usman Dastgeer et Christoph Kessler : Smart containers and skeleton programming for gpu-based systems. *International journal of parallel programming*, 44(3):506–530, 2016.

[46] Usman Dastgeer, Christoph W Kessler et Samuel Thibault : Flexible runtime support for efficient skeleton programming on heterogeneous gpu-based systems. *In PARCO*, pages 159–166, 2011.

[47] Usman Dastgeer, Lu Li et Christoph Kessler : Adaptive implementation selection in the skepu skeleton programming library. *In International Workshop on Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.

[48] Marianne de Michiel, Armelle Bonenfant, Hugues Cassé et Pascal Sainrat : Loop normalization (suite).

[49] Vassilios V Dimakopoulos, Elias Leontiadis et George Tzoumas : A portable c compiler for openmp v. 2.0. *In Proc. EWOMP*, pages 5–11, 2003.

[50] Romain Dolbeau, Stéphane Bihan et François Bodin : Hmpp : A hybrid multi-core parallel programming environment. *In Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[51] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers et Thomas Brox : Flownet : Learning optical flow with convolutional networks. *In Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[52] Johan Enmyren, Usman Dastgeer et Christoph W Kessler : Towards a tunable multi-backend skeleton programming framework for multi-gpu systems. *In Proceedings of the 3rd Swedish Workshop on Multicore Computing*, 2010.

[53] Johan Enmyren et Christoph W Kessler : Skepu : a multi-backend skeleton programming library for multi-gpu systems. *In Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[54] CAPS Enterprise : Hmpp : A hybrid multicore parallel programming platform.

[55] August Ernstsson : Skepu 2 : language embedding and compiler support for flexible and type-safe skeleton programming, 2016.

[56] August Ernstsson : Skepu 2 user guide. 2016.

[57] August Ernstsson, Lu Li et Christoph Kessler : Skepu 2 : Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, pages 1–19, 2017.

[58] Jean-Philippe Farrugia, Patrick Horain, Erwan Guehenneux et Yannick Alusse : Gpucv : A framework for image processing acceleration with graphics processors. *In Multimedia and Expo, 2006 IEEE International Conference on*, pages 585–588. IEEE, 2006.

[59] Paul Feautrier : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[60] Michael Flynn : Flynn's taxonomy. *In Encyclopedia of parallel computing*, pages 689–697. Springer, 2011.

[61] Antoine Fraboulet, Karen Kodary et Anne Mignotte : Loop fusion for memory space optimization. *In Proceedings of the 14th international symposium on Systems synthesis*, pages 95–100. ACM, 2001.

[62] Florian Gouin : Performance optimization and profiling of image processing algorithms on parallel architectures, 2018.

[63] Florian Gouin, Corinne Ancourt et Christophe Guettier : Méthode de calcul de variance locale adaptée aux processeurs graphiques. *In COMPAS2016, Conférence d'informatique en Parallélisme, Architecture et Système*, 2016.

[64] Florian Gouin, Corinne Ancourt et Christophe Guettier : Threewise : a local variance algorithm for gpu. *In 19th IEEE International Conference on Computational Science and Engineering (CSE 2016)*, pages 257–262, 2016.

[65] Florian Gouin, Corinne Ancourt et Christophe Guettier : An up to date Mapping Methodology for GPUs. *In 20th Workshop on Compilers for Parallel Computing (CPC 2018)*, Dublin, Ireland, avril 2018.

[66] Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan et Sven Verdoolaege : Hybrid hexagonal/classical tiling for gpus. *In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 66. ACM, 2014.

[67] Tobias Grosser, Sven Verdoolaege, Albert Cohen et P Sadayappan : The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014.

[68] Serge Guelton, François Irigoin et Ronan Keryell : Compilation for heterogeneous computing : Automating analyses, transformations and decisions. 2011.

[69] Marisabel Guevara, Chris Gregg, Kim Hazelwood et Kevin Skadron : Enabling task parallelism in the cuda scheduler. *In Workshop on Programming Models for Emerging Architectures*, volume 9, 2009.

[70] Pierre Guillou, Benoît Pin, Fabien Coelho et François Irigoin : A dynamic to static dsl compiler for image processing applications. 2017.

[71] Tian Yi David Han : *Directive-Based General-Purpose GPU Programming*. University of Toronto, 2009.

[72] Tianyi David Han et Tarek S Abdelrahman : hi cuda : a high-level directive-based language for gpu programming. *In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[73] Tianyi David Han et Tarek S Abdelrahman : hicuda : High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed systems*, 22(1):78–90, 2011.

[74] Mark Harris, Shubhabrata Sengupta et John D Owens : Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[75] John L Hennessy et David A Patterson : *Computer architecture : a quantitative approach*. Elsevier, 2011.

[76] W Daniel Hillis et Guy L Steele Jr : Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[77] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy et Thomas Brox : Flownet 2.0 : Evolution of optical flow estimation with deep networks. *In Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2462–2470, 2017.

[78] Intel IPL : *Intel® Image Processing Library, Reference Manual*, août 2000.

[79] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet et Ronan Keryell : Polyedres et compilation. *In Rencontres francophones du Parallélisme (RenPar'20)*, 2011.

[80] Abhinav Jangda et Uday Bondhugula : An effective fusion and tile size model for optimizing image processing pipelines. *In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–275. ACM, 2018.

[81] Ken Kennedy et Kathryn S McKinley : Maximizing loop parallelism and improving data locality via loop fusion and distribution. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.

[82] Ronan Keryell, Ruyman Reyes et Lee Howes : Khronos sycl for opencl : a tutorial. *In Proceedings of the 3rd International Workshop on OpenCL*, page 24. ACM, 2015.

[83] David B Kirk et W Hwu Wen-Mei : *Programming massively parallel processors : a hands-on approach*. Morgan kaufmann, 2016.

[84] Leslie Lamport : The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

[85] Chris Lattner : Llvm and clang : Next generation compiler technology. *In The BSD Conference*, pages 1–2, 2008.

[86] Chris Lattner et Vikram Adve : Llvm : A compilation framework for lifelong program analysis & transformation. *In Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[87] Sang-Ik Lee, Troy A Johnson et Rudolf Eigenmann : Cetus–an extensible compiler infrastructure for source-to-source transformation. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2003.

[88] Sean Lee, Manuel MT Chakravarty, Vinod Grover et Gabriele Keller : Gpu kernels as data-parallel array computations in haskell. *In Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, pages 1–9, 2009.

[89] Seyong Lee et Rudolf Eigenmann : Openmpc : Extended openmp programming and tuning for gpus. *In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[90] Seyong Lee et Rudolf Eigenmann : Openmpc : extended openmp for efficient programming and tuning on gpus. *International Journal of Computational Science and Engineering*, 8(1):4–20, 2013.

[91] Seyong Lee, Seung-Jai Min et Rudolf Eigenmann : Openmp to gpgpu : a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4): 101–110, 2009.

[92] Seyong Lee et Jeffrey S Vetter : Openarc : extensible openacc compiler framework for directive-based accelerator programming study. *In Proceedings of the First Workshop on Accelerator Programming using Directives*, pages 1–11. IEEE Press, 2014.

[93] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt et Vijay Janapa Reddi : Gpuwattch : enabling energy optimizations in gpgpus. *In ACM SIGARCH Computer Architecture News*, volume 41, pages 487–498. ACM, 2013.

[94] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul et Richard Lethin : A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. *In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010.

[95] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen et Weimin Zheng : Openuh : An optimizing, portable openmp compiler. *Concurrency and Computation : Practice and Experience*, 19(18):2317–2332, 2007.

[96] Chunhua Liao, Daniel J Quinlan, Richard W Vuduc et Thomas Panas : Effective source-to-source outlining to support whole program empirical optimization. *In LCPC*, volume 9, pages 308–322. Springer, 2009.

[97] AMD Mantle : *Mantle Programming Guide and API Reference*, mars 2015.

[98] Matthew Marangoni et Thomas Wischgoll : Togpu : Automatic source transformation from c++ to cuda using clang/llvm. *Electronic Imaging*, 2016(1):1–9, 2016.

[99] Matt Martineau, Simon McIntosh-Smith et Wayne Gaudin : Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. *In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 338–347. IEEE, 2016.

[100] Ian Masliah, Marc Baboulin et Joel Falcou : Meta-programming and multi-stage programming for gpgpus. *In 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 369–376. IEEE, 2016.

[101] Kathryn S McKinley, Steve Carr et Chau-Wen Tseng : Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[102] Nimrod Megiddo et Vivek Sarkar : Optimal weighted loop fusion for parallel programs. *In Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 282–291. ACM, 1997.

[103] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung et Richard Lethin : R-stream compiler. *In Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[104] Sparsh Mittal et Jeffrey S Vetter : A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[105] Sparsh Mittal et Jeffrey S Vetter : A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):19, 2015.

[106] Theodore S Motzkin et Ernst G Straus : Maxima for graphs and a new proof of a theorem of turán. *Canad. J. Math*, 17(4):533–540, 1965.

[107] Ravi Teja Mullapudi, Vinay Vasista et Uday Bondhugula : Polymage : Automatic optimization for image processing pipelines. *In ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[108] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo et Satoshi Matsuoka : Statistical power modeling of gpu kernels using performance counters. *In Green Computing Conference, 2010 International*, pages 115–122. IEEE, 2010.

[109] Gabriel Noaje : *un environnement parallèle de développement haut niveau pour les accélérateurs graphiques : mise en œuvre à l'aide d'OpenMP*. Thèse de doctorat, Université de Reims-Champagne Ardenne, 2013.

[110] Gabriel Noaje, Christophe Jaillet et Michaël Krajecki : Source-to-source code translator : Openmp c to cuda. *In High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 512–519. IEEE, 2011.

[111] Cedric Nugteren : The bones source-to-source compiler manual. 2012.

[112] Cedric Nugteren et Henk Corporaal : A modular and parameterisable classification of algorithms. *Eindhoven University of Technology, Tech. Rep. ESR-2011-02*, 2011.

[113] Cedric Nugteren et Henk Corporaal : Introducing'bones' : a parallelizing source-to-source compiler based on algorithmic skeletons. *In Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 1–10. ACM, 2012.

[114] Cedric Nugteren et Henk Corporaal : Bones : an automatic skeleton-based c-to-cuda compiler for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):35, 2015.

[115] Cédric Nugteren, Henk Corporaal et Bart Mesman : Skeleton-based automatic parallelization of image processing algorithms for gpus. *In Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 25–32. IEEE, 2011.

[116] Cedric Nugteren, Rosilde Corvino et Henk Corporaal : Algorithmic species revisited : A program code classification based on array references. *In Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–8. IEEE, 2013.

[117] Cedric Nugteren, Pieter Custers et Henk Corporaal : Algorithmic species : a classification of affine loop nests for parallel programming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):40, 2013.

[118] Cedric Nugteren, Pieter Custers et Henk Corporaal : Automatic skeleton-based compilation through integration with an algorithm classification. *In International Workshop on Advanced Parallel Processing Technologies*, pages 184–198. Springer, 2013.

[119] CUDA NVIDIA : CUDA Occupancy Calculator. NVIDIA. url : http ://developer. download. nvidia. com/compute/cuda/CUDA_Occupancy_calculator.xls.

[120] CUDA Nvidia : *Tuning CUDA applications for FERMI*, juillet 2010.

[121] CUDA Nvidia : *Whitepaper NVIDIA Tegra X1*, 2015.

[122] CUDA Nvidia : *CUDA C best practices guid*, septembre 2017.

[123] CUDA Nvidia : *CUDA C programming guide*, septembre 2017.

[124] CUDA Nvidia : *CUDA compiler driver NVCC*, septembre 2017.

[125] CUDA Nvidia : *CUDA driver API*, juillet 2017.

[126] CUDA Nvidia : *CUDA Runtime API*, juillet 2017.

[127] CUDA Nvidia : *Parallel Thread Execution ISA*, juillet 2017.

[128] CUDA Nvidia : *Thrust quick start guide*, décembre 2017.

[129] CUDA Nvidia : *Tuning CUDA applications for KEPLER*, septembre 2017.

[130] CUDA Nvidia : *Tuning CUDA applications for MAXWELL*, septembre 2017.

[131] CUDA Nvidia : *Tuning CUDA applications for PASCAL*, septembre 2017.

[132] CUDA Nvidia : *Tuning CUDA applications for VOLTA*, septembre 2017.

[133] CUDA Nvidia : *Whitepaper : NVIDIA GeForce GTX 1080*, décembre 2017.

[134] Karl Pauwels et Marc M Van Hulle : Realtime phase-based optical flow on the gpu. *In 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8. IEEE, 2008.

[135] A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier et L. Lacassagne : Energy and execution time comparison of optical flow algorithms on simd and gpu architectures. *In 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 25–30, Oct 2018.

[136] Andrea Petreto, Arthur Hennequin, Thomas Koehler, Thomas Romera, Yohan Fargeix, Boris Gaillard, Manuel Bouyer, Quentin Meunier et Lionel Lacassagne : Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU. *In Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2018)*, Toulouse, France, juillet 2018.

[137] William Pugh : The omega test : a fast and practical integer programming algorithm for dependence analysis. *In Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[138] William Pugh et David Wonnacott : Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, 1995.

[139] Apan Qasem et Ken Kennedy : Profitable loop fusion and tiling using model-driven empirical search. *In Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258. ACM, 2006.

[140] Dan Quinlan : Rose : Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[141] Dan J Quinlan *et al.* : Rose compiler project, 2012.

[142] Radu Rugina et Martin Rinard : Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *In ACM Sigplan Notices*, volume 35, pages 182–195. ACM, 2000.

[143] Vivek Sarkar : Optimized unrolling of nested loops. *In Proceedings of the 14th international conference on Supercomputing*, pages 153–166. ACM, 2000.

[144] Sharad K Singhai et Kathryn S. McKinley : A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[145] Oskar Sjöström et Christoph Kessler : Skepu user guide. Rapport technique, Technical report, 2015.(Cited on page 24.), 2015.

[146] Yaya SLIMANI et Denis TRYSTRAM : Papiers présentés à la conférence renpar 2002. 2004.

[147] Michel Steuwer, Philipp Kegel et Sergei Gorlatch : Skelcl-a portable skeleton library for high-level gpu programming. *In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE, 2011.

[148] John E Stone, David Gohara et Guochun Shi : Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[149] Narayanan Sundaram, Thomas Brox et Kurt Keutzer : Dense point trajectories by gpu-accelerated large displacement optical flow. *In European conference on computer vision*, pages 438–451. Springer, 2010.

[150] Michael W. Tao, Jiamin Bai, Pushmeet Kohli et Sylvain Paris : Simpleflow : A non-iterative, sublinear optical flow algorithm. *Computer Graphics Forum (Eurographics 2012)*, 31(2), mai 2012.

[151] Xiaonan Tian, Rengan Xu et B Chapman : Openuh : open source openacc compiler. *GTC2014, HPCTools Group Computer Science Department University of Houston*, 2014.

[152] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran et Barbara Chapman : Compiling a high-level directive-based programming model for gpgpus. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2013.

[153] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi et W Hwu Wen-mei : Cuda-lite : Reducing gpu programming complexity. *In LCPC*, volume 8, pages 1–15. Springer, 2008.

[154] Didem Unat, Xing Cai et Scott B Baden : Mint : realizing cuda performance in 3d stencil methods with annotated c. *In Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[155] Nicolas Vasilache, Benoit Meister, Muthu Baskaran et Richard Lethin : Joint scheduling and layout optimization to enable multi-level vectorization. *IMPACT, Paris, France*, 2012.

[156] Sven Verdoolaege : isl : An integer set library for the polyhedral model. *In ICMS*, volume 6327, pages 299–302. Springer, 2010.

[157] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens et P Catthoor : Multi-dimensional incremental loop fusion for data locality. *In Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pages 17–27. IEEE, 2003.

[158] Sven Verdoolaege et Tobias Grosser : Polyhedral extraction tool. *In Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012.

[159] Sven Verdoolaege et Gerda Janssens : Scheduling for ppcg. 2017.

[160] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado et Francky Catthoor : Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54 :1–54 :23, janvier 2013.

[161] Vasily Volkov : Better performance at lower occupancy. *In Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[162] Vasily Volkov : *Understanding latency hiding on gpus*. Thèse de doctorat, UC Berkeley, 2016.

[163] David Williams, Valeriu Codreanu, Po Yang, Baoquan Liu, Feng Dong, Burhan Yasar, Babak Mahdian, Alessandro Chiarini, Xia Zhao et Jos BTM Roerdink : Evaluation of autoparallelization toolkits for commodity gpus. *In International Conference on Parallel Processing and Applied Mathematics*, pages 447–457. Springer, 2013.

[164] Samuel Williams, Andrew Waterman et David Patterson : Roofline : an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[165] Michael Wolfe : More iteration space tiling. *In Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[166] Michael Wolfe : Implementing the pgi accelerator model. *In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.

[167] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng et Robert

Hundt : Gpucc : An open-source gpgpu compiler. *In Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 105–116. ACM, 2016.

[168] Yi Yang, Ping Xiang, Jingfei Kong et Huiyang Zhou : A gpgpu compiler for memory optimization and parallelism management. *In ACM Sigplan Notices*, volume 45, pages 86–97. ACM, 2010.

RÉSUMÉ

Dans le secteur industriel, la course à l'amélioration des définitions des capteurs vidéos se répercute directement dans le domaine du traitement d'images par une augmentation des quantités de données à traiter. Dans le cadre de l'embarqué, les mêmes algorithmes ont fréquemment pour contrainte supplémentaire de devoir supporter le temps réel. L'enjeu est alors de trouver une solution présentant une consommation énergétique modérée, une puissance calculatoire soutenue et une bande passante élevée pour l'acheminement des données.
Le GPU est une architecture adaptée pour ce genre de tâches notamment grâce à sa conception basée sur le parallélisme massif. Cependant, le fait qu'un accélérateur tel que le GPU prenne place dans une architecture globale hétérogène, ou encore ait de multiples niveaux hiérarchiques, complexifient sa mise en œuvre. Ainsi, les transformations de code visant à placer un algorithme sur GPU tout en optimisant l'exploitation des capacités de ce dernier, ne sont pas des opérations triviales.
Dans le cadre de cette thèse, nous avons développé une méthodologie permettant de porter des algorithmes sur GPU. Cette méthodologie est guidée par un ensemble de critères de transformations de programme. Certains d'entre-eux sont définis afin d'assurer la légalité du portage, tandis que d'autres sont utilisés pour améliorer les temps d'exécution sur cette architecture. En complément, nous avons étudié les performances des différentes mémoires ainsi que la gestion du parallélisme gros grain sur les architectures GPU Nvidia. Ces travaux sont une étape préalable à l'ajout de nouveaux critères dans notre méthodologie, visant à maximiser l'exploitation des capacités de ces GPUs.
Les résultats expérimentaux obtenus montrent non seulement la fiabilité du placement mais aussi une accélération des temps d'exécution sur plusieurs applications industrielles de traitement d'images écrites en langage *C* ou *C++*.

MOTS CLÉS

GPGPU, calcul parallèle, calcul haute perfornance, programmation parallèle, traitement d'images

ABSTRACT

In industries, the curse of image sensors for higher definitions increases the amount of data to be processed in the image processing domain. The concerned algorithms, applied to embedded solutions, also have to frequently accept real-time constraints. So, the main issues are to moderate power consumption, to attain high performance computings and high memory bandwidth for data delivery.
The massively parallel conception of GPUs is especially well adapted for this kind of tasks. However, this achitecture is complex to handle. Some reasons are its multiple memory and computation hierachical levels or the usage of this accelerator inside a global heterogeneous architecture. Therefore, mapping algorithms on GPUs, while exploiting high performance capacities of this architecture, aren't trivial operations.
In this thesis, we have developped a mapping methodology for sequential algorithms and designed it for GPUs. This methodology is made up of code analysis phases, mapping criteria verifications, code transformations and a final code generation phase. Part of the defined mapping criteria has been designed to assure the mapping legality, by considering GPU hardware specifities, whereas the other part are used to improve runtimes. In addition, we have studied GPU memories performances and the capacity of GPU to efficiently support coarse grain parallellism. This complementary work is a foundation for further improvments of GPU resources exploitation inside this mapping methodology.
Last, the experimental results have revealed the functional reliability of the codes mapped on GPU and a speedup on the runtime of many *C* and *C++* image processing applications used in industry.

KEYWORDS

GPGPU, parallel processing, HPC, parallel programming, image processing