

N° D'ORDRE



[THÈSE DE DOCTORAT]

SPECIALITE : PHYSIQUE

*Ecole Doctorale " Sciences et Technologies de l'Information des
Télécommunications et des Systèmes "*

Présentée par :

Tarik Saidani

Sujet :

**Optimisation Multi-niveau d'Applications de Traitement d'Images
sur Machine Parallèle**

Soutenue le ... devant les membres du jury :

M ..

M ..

M ..

M ..

M ..

M ..

Table des matières

1	Introduction	11
1.1	Introduction au Calcul Parallèle	11
1.1.1	Concepts Généraux	12
1.1.2	Architectures Mémoire des Machines Parallèles	14
1.1.3	Modèles de Programmation Parallèle	17
1.2	Traitement d'Images et Parallélisme	21
1.2.1	Le Partitionnement	22
1.3	Aperçu de l'Architecture	22
2	Modèles de Programmation	29
2.1	Les Threads POSIX	29
2.1.1	l'API Pthread	30
2.1.2	Les Threads POSIX sur le Cell	33
2.1.3	Processus de Génération de Code	34
2.1.4	conclusion	34
2.2	RapidMind	36
2.2.1	Modèle de Programmation et Interface	36
2.2.2	Evaluation Partielle et Algèbre du Programme	38
2.2.3	Les Opérations Collectives	40
2.2.4	Spécificité du Backend pour le Cell de RapidMind	41
2.2.5	Conclusion	42
2.3	OpenMP	42
2.3.1	<i>threads</i> et Synchronisation	44
2.3.2	Génération de Code	45

2.3.3	Modèle Mémoire	46
2.3.4	Conclusion	47
2.4	Cell SuperScalar	47
2.4.1	<i>runtime</i>	50
2.4.2	Conclusion	52
2.5	<i>Sequoia</i>	53
2.5.1	Mémoire Hiérarchique	53
2.5.2	Modèle de Programmation	56
2.5.3	Implémentation	60
2.5.4	Conclusion	61
2.6	Squelettes Algorithmiques pour le Cell	61
2.6.1	Modèle de Programmation	62
2.6.2	Squelettes dédiés à la structuration de l'application	66
2.6.3	Modèle de Programmation SKELL_BE	66
2.6.4	Détails de l'Implémentation	68
2.6.5	Génération de Code pour les SPEs	70
2.6.6	Communications PPE/SPEs	74
2.6.7	Résultats Expérimentaux	75
2.6.8	Conclusion	80
2.7	Conclusion Générale	80
3	Parallélisation de Code de Traitement d'Images	83
3.1	Algorithme de Harris	84
3.1.1	Description de l'Algorithme	84
3.1.2	Détails de l'Implémentation	85
3.2	Exploitation du Parallélisme et Optimisations Multi-niveau	87
3.2.1	Techniques Spécifiques au Domaine	88
3.2.2	Optimisation des Transferts Mémoire	95
3.2.3	Schémas de Parallélisation	103
3.3	Évaluation des Performances	107
3.3.1	Métriques de Mesure	108
3.3.2	Méthode et Plateforme de Mesure	108
3.3.3	Comparaison des Schémas de Parallélisation	108
3.3.4	Influence de la Taille de la Tuile	110

3.3.5 Analyse des Résultats	111
3.3.6 Mesure des Métriques de Passage à l'Échelle	112

Table des figures

1.1	Architecture de <i>von Neumann</i>	12
1.2	Machine Parallèle à Mémoire Partagée UMA	15
1.3	Machine Parallèle à Mémoire Partagée NUMA	15
1.4	Machine Parallèle à Mémoire Distribuée	16
1.5	Machine Parallèle à Mémoire Hybride	17
1.6	Partitonnement par décomposition du domaine	22
1.7	Partitonnement par décomposition du domaine en traitement d'images . .	23
1.8	Vue d'ensemble de l'architecture hétérogène du processeur Cell	25
2.1	Processus <i>dual source</i> de génération de code exécutable pour le Cell	35
2.2	Illustration d'une opération collective de réduction	41
2.3	Procédure de génération de code de CellSS	48
2.4	Multiplication de matrices de tailles 1024x1024 structurée en hiérarchie de tâches indépendantes effectuant des multiplications sur des blocs de données plus petits	54
2.5	Modèle abstrait Sequoia du processeur Cell	55
2.6	Exemple de graphe de processus communicants avec hiérarchisation . . .	63
2.7	Arbre représentant le squelette en Figure 2.6	64
2.8	Exemple de squelette du type Pipeline	65
2.9	Exemple de squelette du type Pardo	66
3.1	Illustration de la détection de points d'intérêts sur une image niveaux de gris 512×512	86
3.2	Implémentation de l'algorithme de Harris sous forme de graphe flot de données	86

3.3	Exemple de convolution par un filtre Gaussien 3×3 (a)version avec noyaux 2D (b) version avec deux noyaux 1D , résultant de la séparation du noyau 2D.	88
3.4	Chevauchement des données pour un calcul d'un filtre Gaussien 3×3 , certaines données sont conservées en décalant le masque de convolution.	90
3.5	Règle de composition de fonctions.(a) composition de deux opérateurs point à point (b) composition d'un noyaux de convolution suivi d'un opérateur point à point (c) composition d'un opérateur point à point suivi d'un noyaux de convolution (d) composition de deux noyaux de convolution successifs	93
3.6	Réseau d'interconnection du Cell	95
3.7	Influence de la taille du transfert DMA sur la bande-passante	96
3.8	Influence du nombre de transferts dans le cas d'une communication LS<->LS	97
3.9	Influence du nombre de transferts dans le cas d'une communication MS<->LS	98
3.10	Redondances de données pour un opérateur de convolution 3×3	100
3.11	Tracé de la fonction $Q(h, w)$ dans l'espace	102
3.12	Schéma de Parallélisation SPMD Conventionnel	104
3.13	Schéma de Parallélisation Pipeline	104
3.14	Schéma de Parallélisation Chainage d'opérateurs par paires	105
3.15	Schéma de Parallélisation Chainage et fusion d'opérateurs par paires	106
3.16	Schéma de Parallélisation Chaînage et fusion d'opérateurs par paires	107
3.17	Comparaison des modèles en cycles par point	109
3.18	Influence de la taille de la tuile sur la performance pour la version chainage entier et fusion d'opérateur par paires 1 SPU	110
3.19	Mesure du <i>Speedup</i> en fonction du nombre de SPEs	112
3.20	Mesure de l'Efficacité en fonction du nombre de SPEs	113

Liste des tableaux

1.1	Classification de Flynn des machines parallèles	13
2.1	Interface utilisateur SKELL_BE	69
2.2	Benchmark DOT	77
2.3	Benchmark CONVO	77
2.4	Benchmark SGEMV	78
2.5	Benchmarks de l'algorithme de Harris	79
2.6	Benchmarks de l'algorithme de Harris	79
3.1	Réduction de la complexité arithmétique par séparabilité des noyaux . . .	89
3.2	Réduction de la complexité mémoire par séparabilité des noyaux	89
3.3	Réduction de la complexité mémoire par chevauchement des noyaux . .	91
3.4	Réduction de la complexité arithmétique par séparabilité et chevauchement des noyaux	91
3.5	Réduction de la complexité mémoire par séparabilité et chevauchement des noyaux	92
3.6	Tableau récapitulatif de l'optimisation des accès mémoire en combinant l'ensemble des optimisations	94
3.7	Apport de la composition de fonction à la performance	111

Introduction

1.1 Introduction au Calcul Parallèle

Les logiciels ont été conçus historiquement pour une exécution en série. Les programmes devaient s'exécuter sur une seule machine contenant une seule unité de traitement centrale (*CPU*) et le problème est décomposé en une suite d'instructions qui sont exécutées les unes après les autres. Ainsi, une seule instruction peut être exécutée à la fois. Le calcul parallèle est par opposition à la précédente approche, l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème. Un logiciel peut ainsi s'exécuter sur plusieurs *CPU*. Le problème est décomposé en plusieurs parties qui peuvent être résolues de manière concurrente. Ces parties sont à leur tour décomposées en plusieurs instructions et chaque paquet d'instructions s'exécute de manière indépendante l'un de l'autre. Les ressources de calcul incluent une seule machine avec plusieurs processeurs, un nombre arbitraire de machines connectées via un réseau ou alors une combinaison des deux. Une bonne partie des problèmes de calcul intensif possèdent certaines caractéristiques qui en font de bon candidats à la parallélisation. Parmi ces caractéristiques : le possibilités de les décomposer en plusieurs sous-problèmes qui peuvent être résolus simultanément et la possibilités d'être résolus en moins de temps avec plusieurs ressources qu'avec une seule. Le calcul parallèle était auparavant, réservé exclusivement à la modélisation de problèmes et de phénomènes scientifiques provenant de la réalité tels que l'environnement, la physique nucléaire, les biotechnologies, la géologie et les mathématiques. A ce jour, le calcul parallèle s'est ouvert à d'autre domaines grâce notamment à l'évolution fulgurante de la technologie des semi-conducteurs qui a rendu les

plate-formes haute performance plus accessibles. On peut citer des applications comme les bases de données, l'exploration pétrolière, les moteurs de recherche, la modélisation financière, les technologies de diffusion multimédia et les applications graphiques et de réalité virtuelle.

1.1.1 Concepts Généraux

Architecture de *von Neumann*

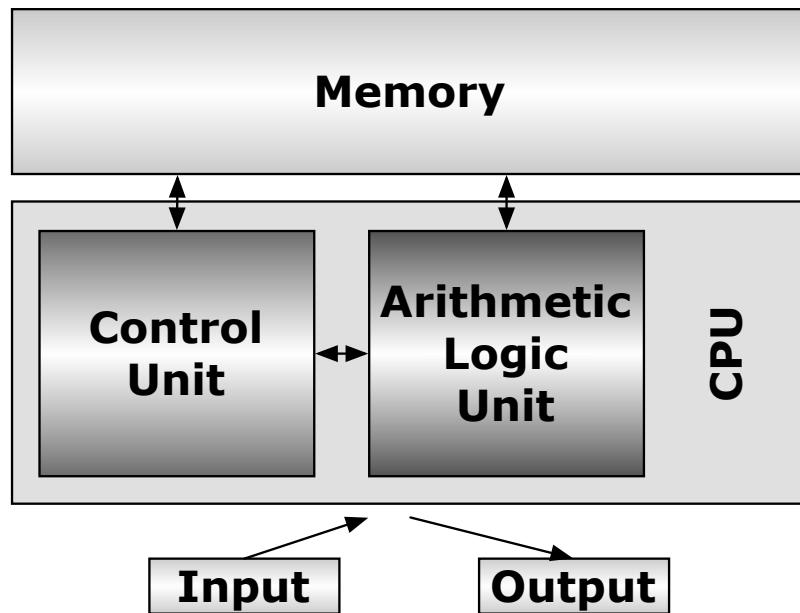


FIG. 1.1 – Architecture de *von Neumann*

Ce modèle fut inventé par le mathématicien hongrois *John von Neumann* qui a posé les premières bases de la conception d'un ordinateur dans son papier de 1945 [20]. A partir de ce moment tous les ordinateurs ont été conçus sur ces bases. L'architecture *von Neumann* 1.1 est constituée de 4 composants principaux : une mémoire, une unité de contrôle, une unité arithmétique et logique(ALU) des Entrées/Sorties (*I/O*). La mémoire à accès aléatoire (*RAM*) en lecture/écriture est utilisée pour stocker les instructions ainsi que les données. L'unité de contrôle va chercher les instructions ou les données de la mémoire, décode les instructions et coordonne séquentiellement les opérations afin d'accomplir la tâche programmée. L'ALU effectue les opérations arithmétiques de base. les

SISD	SIMD
Single Instruction Single Data	Single Instruction Multiple Data
MISD	MIMD
Multiple Instruction Single Data	Multiple Instruction Multiple Data

TAB. 1.1 – Classification de Flynn des machines parallèles

I/O font l'interface avec l'utilisateur humain.

Classification de Flynn des Machines Parallèles

Il existe plusieurs manières de classer les machines parallèles. Toutefois, il existe une classification qui est largement utilisée depuis 1966 et qui est celle de Flynn [8] (*Flynn's Taxonomy*). Cette classification distingue les architectures parallèles selon deux paramètres indépendants qui sont les instructions et les données : chacun de ces deux paramètres peut avoir deux états possibles *Single* ou *Multiple*. Ainsi le tableau 1.1 illustre la classification de Flynn.

Single Instruction, Single Data (SISD) Une machine série qui ne peut exécuter qu'un seul flux d'instruction en un cycle d'horloge *CPU*. De plus, un seul flux de données est utilisé comme entrée en un cycle d'horloge. L'exécution du programme y est déterministe et il constitue le type de machines à la fois le plus ancien est le plus répandu de nos jours.

Single Instruction, Multiple Data (SIMD) C'est un type de machines parallèles dont les processeurs exécutent la même instruction en un cycle d'horloge donné. Cependant, chaque unité de traitement peut opérer sur un élément de données différent. Ce type de machines est bien taillé pour des problèmes réguliers tels que le traitement d'images et le rendu graphique. L'exécution des programme y est synchrone et déterministe. Deux variantes de ces machines existent :

- Processor Arrays : Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV
- Vector Pipelines : IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10

De plus, la majorité des processeurs des stations de travail actuelles et des unités de traitement graphiques, comportent une unité de traitement spécialisée SIMD, on parle alors de *SWAR (SIMD Within A Register)*.

Multiple Instruction, Single Data (MISD) Un seul flux de données alimente plusieurs unités de traitement et chaque unité de traitement opère sur les données de manière indépendante grâce à un flot d'instructions indépendants.

Multiple Instruction, Multiple Data (MIMD) C'est actuellement le type le plus commun de machines parallèles. Chaque processeur de ces machines peut exécuter un flux d'instructions différent et peut opérer sur un flux de données différent. L'exécution peut être synchrone ou asynchrone, déterministe ou non-déterministe. On peut citer les *Supercomputers* actuels, les clusters de machines parallèles mis en réseau, les grilles de calculs, les multi-processeurs SMP (Symetric Multi-Processor) et les processeur multi-core. De plus, plusieurs de ces machines contiennent des unités de traitement SIMD.

1.1.2 Architectures Mémoire des Machines Parallèles

Dans la suite nous donnons une classification des machines parallèles selon le type de leur hiérarchie mémoire. Cette classification permet d'une part de distinguer les machines parallèles d'un autre point de vue que celui du CPU et permet également de mieux comprendre les motivations des modèles de programmation pour les machines parallèles.

Les Machines Parallèles à Mémoire Partagée

Il existe plusieurs variantes de ces machines mais toutes partagent une propriété commune qui est la capacité de tous les processeurs d'accéder toute la mémoire comme un espace d'adressage global. Ainsi, plusieurs processeurs peuvent opérer d'une manière indépendante mais partagent la même ressource mémoire. Un changement opéré par un processeurs dans un emplacement mémoire est visible à tous les autres processeurs. Cette classe de machine peut être divisée en deux sous-classes basées sur les temps d'accès à la mémoire : UMA et NUMA.

Uniform Memory Access (UMA) Ce sont principalement les machines de type SMP qui possèdent plusieurs processeurs identiques et qui peuvent accéder de manière égale et en un temps identique à la mémoire. Elles sont parfois appelées CC-UMA - Cache Coherent UMA. La cohérence de cache signifie que si un processeur met à jour un emplacement de la mémoire tous les autres processeurs sont au courant de ce changement. Cette fonctionnalité est assurée au niveau du *hardware*.

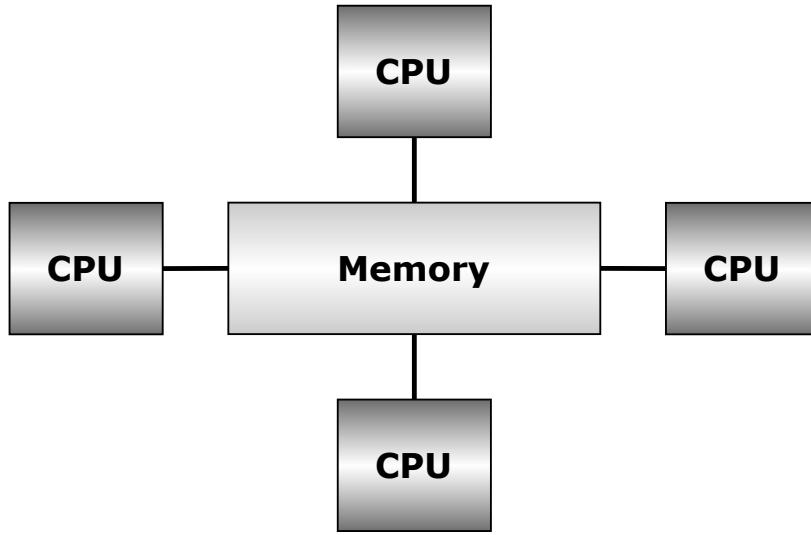


FIG. 1.2 – Machine Parallèle à Mémoire Partagée UMA

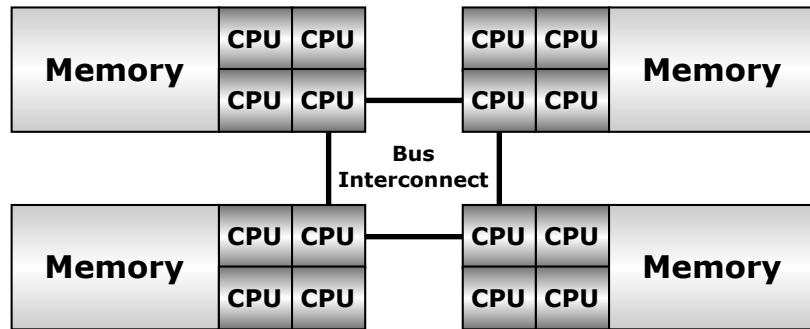


FIG. 1.3 – Machine Parallèle à Mémoire Partagée NUMA

Non-Uniform Memory Access (NUMA) Ce type de machines est souvent conçu en connectant deux ou plusieurs SMPs. Un SMP peut avoir un accès direct à la mémoire d'un autre SMP. Le temps d'accès à une mémoire donnée n'est pas égal pour tous les processeurs et lorsque un noeud est traversé l'accès est plus lent. Si la cohérence de cache est garantie on parle alors de CC-NUMA.

Avantages et Incovénients Parmi les avantages de ce type d'architectures mémoire est une perspective simplifiée de la mémoire du point de vue du programmeur. Le partage des données entre les tâches est à la fois rapide et uniforme. Le premier inconvénient

est le manque de mis à l'échelle (*scalability*) entre la mémoire et les CPUs. Le fait d'augmenter le nombre de CPUs augmente le trafic sur le bus mémoire et provoque un goulot d'étranglement et la gestion de la cohérence devient de plus en plus complexe. Le programmeur est responsable de la synchronisation des tâches qui garantit un accès correcte à la mémoire globale. Par conséquent, la conception de machine parallèles à mémoire partagée avec de plus en plus de processeurs devient difficile et coûteux.

Les Machines Parallèles à Mémoire Distribuée

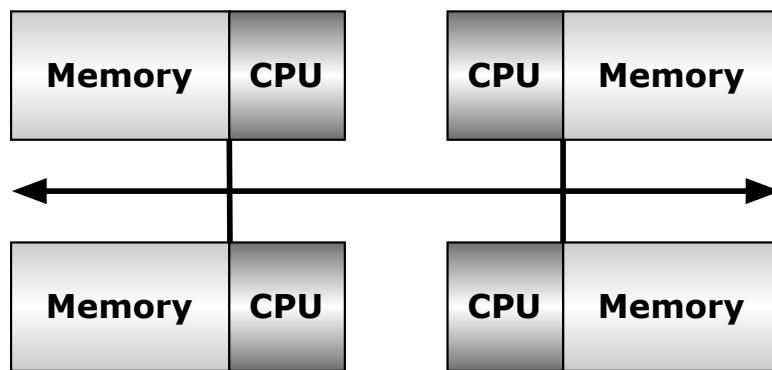


FIG. 1.4 – Machine Parallèle à Mémoire Distribuée

Comme les machines à mémoire partagée, les machines à mémoire distribuée varient mais elles partagent tout de même un point commun : elles requièrent un réseau de communication pour connecter la mémoire inter-processeurs. Les différents processeurs possèdent leur propre mémoire locale. Les adresses mémoire d'un processeur donnée ne correspondent pas à celles d'un autre et par conséquent le concept de mémoire globale n'existe pas. Puisque chaque processeur possède sa propre mémoire privée il opère de manière indépendante. En effet, chaque changement opéré sur sa mémoire locale n'a aucun effet sur la mémoire des autres processeurs ce qui exclue le concept de cohérence de cache. Lorsqu'un processeur a besoin des données contenues dans la mémoire d'un autre processeur, le programmeur est en charge de définir quand et comment les données sont transférées. Ce dernier est aussi responsable de la synchronisation.

Avantages et Incovénients L'avantage majeur de ce type d'architectures est le fait que la mémoire soit *scalable* avec le nombre de processeurs. En effet, la taille de la mémoire

croît proportionnellement avec le nombre de processeurs. Chaque processeur peut aussi accéder rapidement à sa mémoire locale sans interférence et sans engendrer de surcout du au maintien de la cohérence de cache. Le principal inconvénient de ce type d'architectures mémoire et la gestion explicite par le logiciel des communications entre les processeurs. Les accès à la mémoire se font souvent à des temps non-uniformes et la présence de plusieurs espaces d'adressage rend complexe l'adaptation de programmes écrits pour une mémoire partagée.

Les Machines Parallèles à Mémoire Hybride

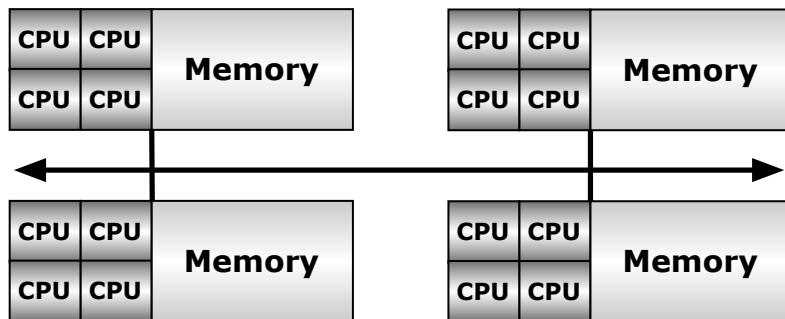


FIG. 1.5 – Machine Parallèle à Mémoire Hybride

Les machines les plus rapides du monde emploient des architectures mémoire dites hybrides qui regroupent les deux types précédents : partagée et distribuée. La composante mémoire partagée est souvent une machine SMP. La composante distribuée quant à elle consiste en la mise en réseau de plusieurs machines SMP. Les différents SMPs ne peuvent adresser que leur propre mémoire et le transfert de données entre deux SMPs requiert des communications au travers du réseau. Selon le niveau dans lequel on se trouve, ce type de machines possède les inconvénients et avantages des deux précédentes architectures mémoire.

1.1.3 Modèles de Programmation Parallèle

Il existe plusieurs modèles de programmation pour les machines parallèles. Ces modèles existent à un niveau d'abstraction au dessus de l'architecture matérielle et de celle de la mémoire. Même si à première vue les modèles de programmation sont intimement liés à l'architecture de la machine ils sont supposés pouvoir être implémentés sur

n’importe quelle machine parallèle quelqu’en soient les caractéristiques. Il n’existe pas de modèle de programmation idéal mais certains modèles de programmation sont bien adaptés pour une application données sur une machine donnée. Dans la suite nous décrivons les principaux modèles de programmation parallèles.

Le Modèle *Shared Memory*

Dans ce modèle de programmation les tâches partagent un espace d’adressage commun sur lequel ils peuvent lire et écrire des données de manière asynchrone. Plusieurs mécanismes, tels que les *locks* et les sémaphores peuvent être utilisés pour contrôler l’accès à la mémoire partagée. Ce modèle de programmation est simplifié du point de vue de l’utilisateur car il n’y a pas de notion d’appartenance des données à une tâche ce qui évite les communication explicite pour transférer des données d’une tâche à une autre. Toutefois, en terme de performances ce dernier point constitue inconvénient car il engendre un surcout d’accès à la mémoire de rafraîchissement de cache et de trafic sur le bus lorsque plusieurs processeurs utilisent les mêmes données. Les implémentations de ce modèle sur les machines à mémoire partagés se résument au compilateur natif qui traduit les variables du programme en adresse mémoire globales. Il n’existe cependant pas d’implémentation de ce modèle sur des machines à mémoire distribuée.

Le Modèle de Programmation par *Threads*

Dans le modèle de programmation par threads, un seul *process* peut avoir des chemins d’exécution multiples et concurrents. On peut assimiler ce concepts à un programme principal qui inclue un certain nombre de sous-routines. Le programme principal est ordonné pour être exécuté par les système d’exploitation, et il acquièrent toutes les ressources système nécessaires à son exécution. Il effectue alors un ensemble d’instructions en série et crée un certain nombres de tâches (*threads*) qui peuvent être ordonnancées et exécutées par l’OS de manière concurrente. Chaque *thread* possède ses données locales mais partage également les ressources du programme principal avec les autres *threads*. Chaque *thread* possède un accès à la mémoire globale car il partage l’espace d’adressage du programme principal. La charge du travail d’un *thread* peut être considérée comme une sous-routine du programme principal mais qui peut s’exécuter en parallèle d’un autre *thread*. Les *threads* communiquent entre eux via la mémoire globale ce qui nécessite des opérations de synchronisation afin de garantir l’exclusivité de l’accès à un emplacement

donnée à un instant donné pour un seul *thread*. Les *threads* ont une durée de vie variable et peuvent être créés et détruits tout au long du déroulement du programme. Le modèle de programmation par *thread* est souvent associé avec les machines à mémoire partagée. Les implémentations des *threads* comportent en général une librairie de fonctions ou alors une série de directives enfouies dans le code parallèle. Dans les deux cas l'utilisateur est responsable de la définition du parallélisme. Il existe plusieurs implémentations des *threads*, et la plupart des constructeurs ont développé leur propre version ce qui a affecté la portabilité des codes parallèles. Cependant, un effort de standardisation a donné naissance à deux implémentations qui sont devenues le standard de nos jours.

Les Threads POSIX Ils sont basé sur une librairie de programmation parallèle et spécifiées par le standard *IEEE POSIX 1003.1c standard (1995)* [12]. Ils sont implementés uniquement en langage C et plus connus sous le nom de *Pthreads*. Le parallélisme y est explicite et l'interface bas-niveau force le programmeur à donner beaucoup d'attention au détails.

OpenMP C'est un modèle de programmation basé sur des directives de compilation et peut être directement utilisé sur du code série. Ce standard a été défini par un consortium de vendeurs de processeurs et de logiciel. L'API Fortran a été délivrée en 1997 alors que l'API C/C++ ne l'a été qu'une année plus tard. C'est une API portable et multi-plateforme et est très simple d'utilisation.

Le Modèle *Message Passing*

Dans ce modèle, la programmation parallèle se fait par passage de messages. Un ensemble de tâche utilisent leur propre mémoire locale durant le calcul. Plusieurs tâches peuvent résider sur la même machine physique ou alors sur un nombre arbitraire de machines. Les tâches échangent des données au travers des communications en envoyant et recevant des messages. Les transferts de données requièrent des opérations coopératives pour être effectuées par chaque *process*. Par exemple, une opération *send* doit avoir une opération symétrique *receive*. Les implémentations du *Message Passing* prennent la forme d'une librairie de sous-routines et le programmeur est responsable de la détermination du parallélisme. Comme pour toute librairie, plusieurs versions ont été développées, ce qui a provoqué des problèmes de compatibilité. En 1992 le *MPI Forum* a vu le jour dans le but de standardiser les implémentations du *Message Passing* et a délivré deux standard

MPI [9] en 1994 et MPI-2 en 1996. Des nos jours MPI est le modèle de programmation le plus utilisé pour le *Message Passing*. Dans les implémentations MPI sur des architectures à mémoire partagée les communications réseaux sont tout simplement remplacées par des copies mémoire.

Le Modèle *Data Parallel*

Ce modèle est basé sur le parallélisme de données qui concentre le travail en parallèle sur un ensemble de données sur un tableau à une ou plusieurs dimensions. Un ensemble de tâches travaillent collectivement sur la même structure de données mais chaque tâche opère sur une partition différente de cette structure. Les tâches effectuent toutes la même opération sur leur partition de données. Sur les architectures à mémoire partagée toutes les tâches peuvent avoir accès à la structure de données via la mémoire globale. Par contre lorsque l'architecture mémoire est distribuée les données sont divisées en morceaux qui résident dans la mémoire locale de chaque tâche. La programmation avec ce modèle se fait en général en écrivant du code avec des constructions de parallélisme de données. Ces dernières peuvent avoir la forme d'appel à des fonction d'une librairie ou à des directives reconnues par un compilateur *data parallel*. Les implémentations de ce modèle sont souvent des extensions ou de nouveaux compilateurs on peut citer les compilateurs Fortran (F90 et F95) et leur extension High Performance Fortran (HPF) qui supportent la programmation *data parallel*. HPF inclue des directives qui contrôlent la distribution des données, des assertions qui peuvent améliorer l'optimisation du code généré ainsi que des constructions *data parallel*. Les implémentations sur les architectures mémoire distribuées de ce modèle sont sous forme d'un compilateur qui convertit le code standard en code *Message Passing* (MPI) qui distribue les données sur les différents processeurs et tout cela de manière transparent du point de vue de l'utilisateur.

Autres Modèles

D'autres modèles existent et existeront dans le futur proche en plus de ceux mentionnés auparavant. On peut en mentionner trois :

Modèle Hybride Dans ce modèle deux ou plusieurs modèles sont combinés. On peut citer par exemple la combinaison de *MPI* avec les *Pthreads* ou avec *OpenMP*. Ainsi, différents niveaux de parallélisme sont gérés, par exemple un réseau de SMPs. On peut citer

également la combinaison de *HPF* avec *MPI* pour le même type de configuration.

Modèle Single Program Multiple Data Le modèle *SPMD* est un modèle haut niveau qui peut être construit sur la base d'une combinaison des modèles cités précédemment. Un seul programme est exécuté par toutes les tâche simultanément. A n'importe quel instant les tâches peuvent exécuter des instructions différentes ou similaires du même programme. Un programme *SPMD* peut toutefois contenir des branchements qui permettent à une tâche de n'exécuter qu'une portion du code et toutes les tâches peuvent utiliser différentes données.

Modèle Multiple Program Multiple Data Tout comme le modèle *SPMD*, le modèle *MPMD* est haut-niveau et peut englober l'ensemble des modèles citées précédemment. Les programmes *MPMD* ont typiquement plusieurs objets exécutables. Lors de l'exécution parallèle du programme une tâche peut exécuter le même programme ou un programme différent et toutes les tâches peuvent utiliser des données différentes.

1.2 Traitement d'Images et Parallélisme

Le traitement d'images est une discipline du traitement du signal dont l'entrée est une image. Les techniques de traitement du signal sont appliquées à un signal de 2 dimensions pour donner en sortie soit une image, soit une certaine caractéristique de l'image. Dans notre cas on s'intéresse au traitement d'images de bas niveau, où les opérateurs sont souvent des opérateurs point-à-point ou des noyaux de convolution. Ce type de calcul est bien adapté aux machines parallèles car il n'y a généralement pas de dépendances de données et les algorithmes de bas-niveau sont majoritairement des opérations de calcul pur. Toutefois, pour des applications de moyen-niveau, par exemple la segmentation, il existe une forte dépendance de données et les algorithmes sont souvent des imbriques de structures conditions. Ceci rend la tâche de parallélisation pour ce types d'algorithmes fastidieuse, et les accélérations sont souvent médiocres en comparaison avec l'effort fourni pour l'adaptation de l'algorithme série. La conception d'une version parallèle d'un algorithme devient une tâche fastidieuse et qui dépend de plusieurs facteurs. Il est alors important d'accorder de l'importance à plusieurs aspects déterminants pour la performance.

1.2.1 Le Partitionnement

La première tâche lors de la parallélisation d'une application de traitement d'image est le choix du type partitionnement à adopter. Ce principe prend ça source du fait qu'il y a plusieurs ressources et que le but est de repartir la charge de travail sur ces ressources sous formes de morceaux qui peuvent être distribués sur plusieurs tâches. Il existe deux méthodes basiques pour le partitionnement la décomposition de domaine (*domain decomposition*) et la décomposition fonctionnelle (*functional decomposition*)

Décomposition de Domaine

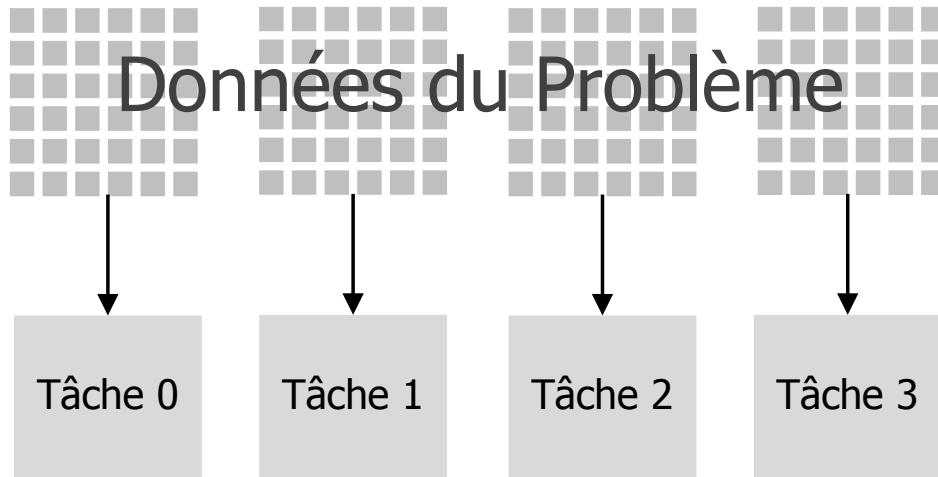


FIG. 1.6 – Partitonnement par décomposition du domaine

Dans ce type de partitionnement, les données associées au problème sont décomposées. Chaque tâche parallèle travaille donc sur une portion des données. Il existe alors plusieurs manières de décomposer les données. En traitement d'images, les données sont en 2 dimensions les décompositions possibles sont illustrées en figure 1.7.

1.3 Aperçu de l'Architecture

Le processeur Cell est la première implémentation de l'architecture *Cell Broadband Engine* (CBEA), qui est entièrement compatible avec l'architecture *PowerPC 64-bit*. Ce processeur a été initialement conçu pour la console de jeux *PlayStation 3* mais ses performances hors normes ont très vite fait de lui un bon candidat pour d'autres domaines

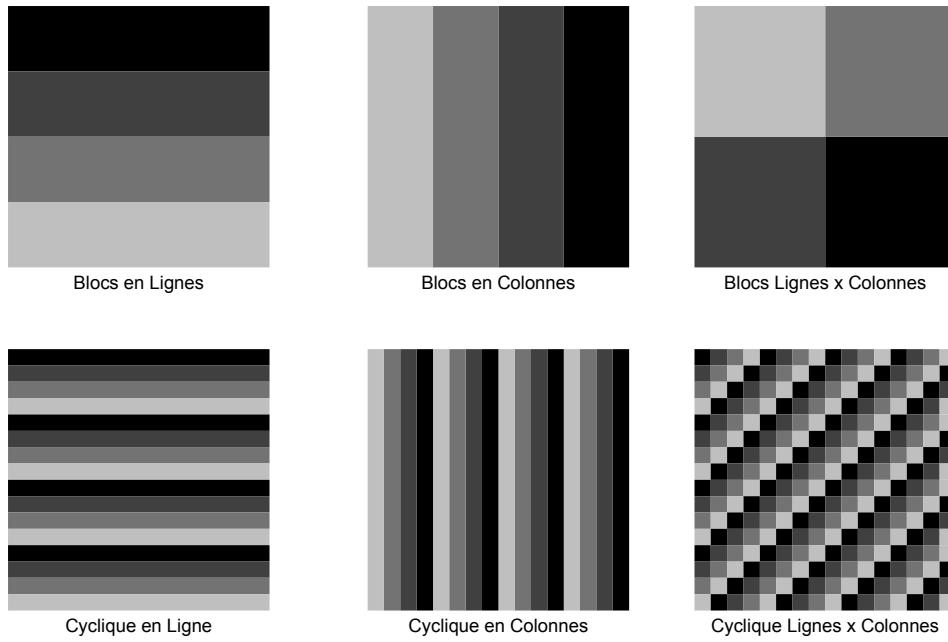


FIG. 1.7 – Partitonnement par décomposition du domaine en traitement d’images

d’applications qui requièrent une grande puissance de calcul, comme le traitement du signal et des images.

Le processeur Cell est une machine multi-coeurs hétérogène, capable d’effectuer une quantité de calcul en virgule flottante considérable, sur des données occupant une large bande-passante. Il est composé d’un processeur 64-bit appelé *Power Processor Element* (PPE), huit co-processeurs spécialisés appelés *Synergistic Processor Element* (SPE), un contrôleur mémoire haute-vitesse et une interface de bus à large bande-passante. Le tout intégré sur une seule et même puce.

Le PPE et les SPEs communiquent au travers d’un bus interne très rapide appelé *Element Interconnect Bus* (EIB). Si l’on considère une fréquence d’horloge de 3.2 Ghz, le processeur Cell peut atteindre théoriquement une performance crête de 204.8 Gflop/s en simple-précision et 14.6 Gflop/s en double-précision.

Le bus interne supporte une bande passante qui peut aller jusqu’à 204.8 Gbytes/s pour les transferts internes à la puce (impliquant le PPE, les SPEs, la mémoire et les contrôleurs I/O). le contrôleur mémoire : *Memory Interface Controller* (MIC) fournit une bande-passante de 25.6 Gbytes/s vers la mémoire principale. Le contrôleur I/O quand à lui fournit 25 Gbytes/s en entrée et 35 Gbytes/s en sortie.

Le rôle du PPE est celui d’un chef d’orchestre, il prend en charge l’OS (*Operating System*)

et coordonne les SPEs. Au niveau de l'architetcure c'est un *PowerPC* 64-bit classique avec une extension SIMD (VMX), un cache L1 de 32 KB (données et instructions) et un cache L2 de 512 KB. C'est un processeur à exécution dans l'ordre (*in-order execution processor*), il supporte le dual-issue (parallélisme d'instructions) ainsi que le multi-threading d'ordre 2 (parallélisme de tâches).

Chaque SPE est composé d'une SPU (*Synergistic Processor Unit*) et d'un MFC (*Memory Flow Controller*). Le MFC contient à son tour un contrôleur DMA, une unité de gestion de la mémoire (MMU), une unité interface de bus, et une unité atomique pour la synchronisation avec les autres SPEs et le PPE. Le SPU est un processeur de type RISC avec un jeu d'instructions et une micro-architecture conçus pour les applications flot de données haute-performance ou de calcul intensif. Le SPU inclut une mémoire locale de 256 KB qui contient les données et les instructions. Le SPU ne peut pas accéder directement à la mémoire principale mais par le biais de commandes DMA via le MFC qui permettent de lire et d'écrire dans la mémoire principale. Les deux unités MFC et SPU sont indépendantes ce qui permet l'exécution des tâches de calcul et de transferts en parallèle sur le SPE.

Il n'existe pas de mécanisme hardware tel que la mémoire cache pour la gestion des mémoires locales, et celles-ci doivent être gérées par le software. Le MFC effectue des commandes DMA pour transférer entre la mémoire centrale et les mémoires locales. Les instructions DMA pointent des emplacements de la mémoire centrales en utilisant des adresses virtuelles compatibles *PowerPC*. Les commandes DMA peuvent transférer des données à partir de n'importe quel emplacement lié au bus d'interconnexion (mémoire principale, la mémoire locale d'un autre SPE, ou un périphérique I/O). Des transferts SPE vers SPE en parallèle sont faisables à raison de 16 bytes par cycle d'horloge SPE, tandis que la bande-passante de la mémoire centrale est de 25.6 Gbytes/s pour le processeur entier.

Chaque SPU contient 128 registres SIMD de taille 128-bits. Cette quantité importante de registres facilite l'ordonnancement efficace des instructions ainsi que d'autres optimisations comme le déroulage de boucle (*loop-unrolling*).

Toutes les instructions SIMD sont des instructions que le pipeline peut exécuter à 4 granularités : 16 entiers 8-bit, 8 entiers 16-bit, 4 entiers 32-bits ou flottants simple-précision. Le processeur SPU est un processeur à exécution dans l'ordre (*in-order-execution processor*), il possède deux pipelines d'instructions connus sous les dénominations pair (*even*) et impair (*odd*).

Les instructions flottantes et entières sont dans le pipeline *even* alors que le reste est dans le pipeline *odd*. Chaque SPU peut lancer et compléter jusqu'à deux instructions par cycle, une par pipeline. Cette limite théorique peut être atteinte pour un large éventail d'applications. Toutes les instructions flottantes double-précision peuvent être lancées en un cycle d'horloge du processeur. Par contre les instructions flottantes double-précision ne sont pipelinées que partiellement, il en résulte un débit d'exécution moindre (deux instructions double-précision tous les 7 cycles d'horloge SPU).

Si l'on prend une instruction de multiplication accumulation en flottant simple-précision (qui compte pour deux opérations) les 8 SPEs peuvent exécuter un total de 64 opérations par cycle.

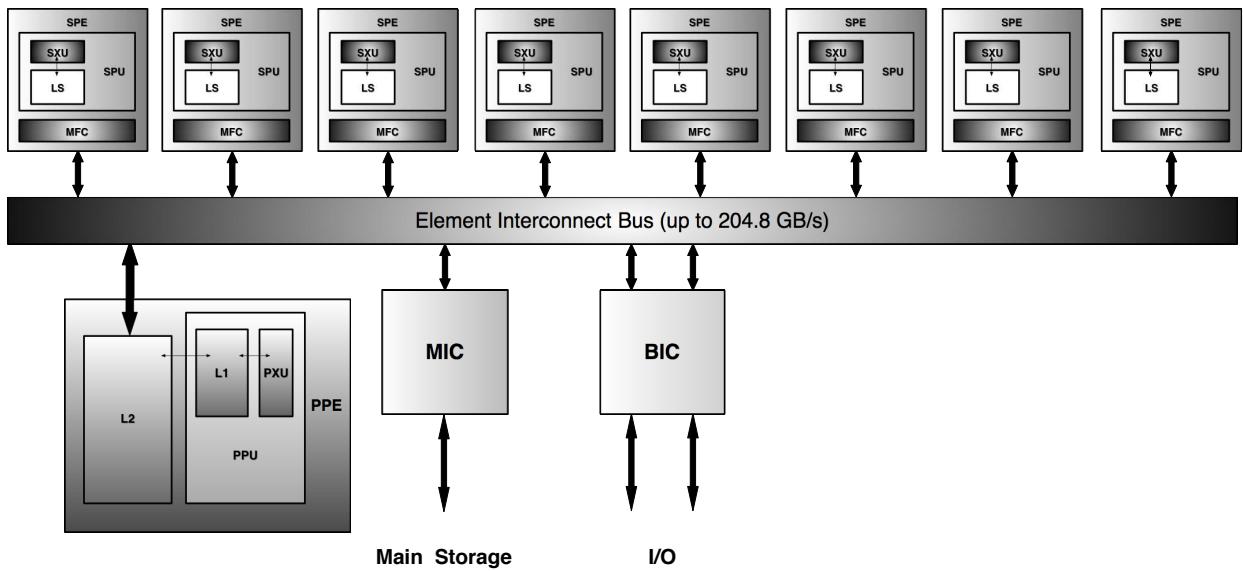


FIG. 1.8 – Vue d'ensemble de l'architecture hétérogène du processeur Cell

Le PPE : Power Processor Element

Le PPE est un processeur 64-bit compatible avec l'architecture Power, optimisée au niveau de l'efficacité énergétique. La profondeur de pipeline du PPE est de 23 étages, chiffre qui peut paraître faible par rapport aux précédentes architectures surtout quand on sait que la durée de l'étage est réduite d'un facteur 2. Le PPE est une architecture dual-issue (deux instructions peuvent être lancées par cycle) qui ne reordonne pas

dynamiquement les instructions l'exécution (exécution dans l'ordre). Le processeur entrelace des instructions provenant de deux *threads* de calcul différents pour optimiser l'utilisation de la fenêtre d'exécution. Les instructions arithmétiques simples s'exécutent et fournissent leur résultat en deux cycles. Les instructions de chargements (LOAD) s'exécutent également en deux cycles. Une instruction flottante en double précision s'exécute en 10 cycles. Le PPE supporte une hiérarchie conventionnelle de caches avec un cache L1 (de niveau 1) données et instructions de 32-KB, et un cache L2 de 512-KB.

Le processeur fournit deux *threads* d'exécution simultanés et peut être vu comme un processeur double-coeur avec un flot de données partagé, ceci donne l'impression au logiciel d'avoir deux unités de traitement distinctes. Certains registres sont dupliqués mais pas les caches qui sont partagés par les deux *threads*.

Le processeur est composé de trois unités : l'unité d'instructions (UI) responsable du chargement, décodage, branchements, exécution et complétion des instructions. Une unité d'exécution des opérations en arithmétique point-fixe (XU) qui est également responsable des instructions LOAD/STORE. Et enfin l'unité VSU qui exécute les instructions en virgule flottante ainsi que les instructions vectorielles. Les instructions SIMD dans le PPE sont celle des anciennes générations de PowerPC 970 et effectuent des opérations sur des registres 128-bit de données qui donnent un parallélisme de 2, 4, 8 ou 16, selon le type de données considéré.

Les SPE (Synergistic Processing Element)

Le SPE contient un jeu d'instructions nouveau mais qui n'est autre qu'une version réduite du jeu d'instructions SIMD VMX (AltiVec), mais qui est optimisé au niveau de consommation d'énergie et des performances pour les applications de calcul intensif et de multimedia. Le SPE contient une mémoire locale de 256 KB (scratchpad) qui est une mémoire de données et d'instructions. Les données et les instructions sont transférées de la mémoire centrale vers cette mémoire privée au travers de commandes DMA synchrones et cohérentes qui sont exécutées par le MFC (Memory Flow Controller) qui est présent dans chaque SPE. Chaque SPE peut supporter jusqu'à 16 commandes DMA en suspens. L'unité DMA peut être programmée de trois manières différentes : 1) avec des instructions sur le SPE qui insèrent des commandes DMA dans la file d'attente ; 2) Par la programmation de transferts permettant de faire des accès sur des zones non contigues de la mémoire au travers d'une liste de DMA ; 3) Par l'insertion d'une commande DMA dans la file d'attente d'un autre processeur par les commandes de DMA-write. Afin de

faciliter la programmation et de permettre des transferts entre SPEs les mémoires locales sont mappées en mémoire centrale. La présence des mémoires locales introduit un autre niveau dans la hiérarchie mémoire au dessus des registres. Les temps d'accès de ces mémoires sont de l'ordre du cycle ce qui en fait de bon candidats pour réduire la latence d'accès la mémoire centrale qui est de l'ordre de 1000 cycles, d'autant plus que le fait que le contrôleur DMA soit indépendant de l'unité donne un niveau de parallélisme supplémentaire. La présence de ces mémoires privées permet différents modèles de programmation qui peuvent être appliqués au processeur Cell.

Le Bus Interne (Element Interconnect Bus)

Le bus interne du processeur permet de relier les unités de traitement PPE, SPE la fois entre elles, la mémoire centrale ainsi qu'une sortie externe. Le bus contient des chemins de données différents de ceux des requêtes. Les éléments autour du bus sont connectés par des liaisons point-à-point et un arbitre de bus est responsable de la réception des commandes et de leur diffusion vers les unités. Le bus est constitué de 4 anneaux d'une largeur de 16-octets deux fonctionnent dans le sens d'une aiguille d'une montre et les deux autres dans le sens inverse. Chaque anneau peut potentiellement gérer 3 transferts en parallèle si toutefois leurs chemins ne se croisent pas. Le EIB opère à une fréquence qui est la moitié de celle du processeur, chaque unité du bus peut simultanément envoyer et recevoir 16 octets par cycle d'horloge du bus.

Chapitre 2

Modèles de Programmation

Dans ce chapitre on se propose de faire une revue des modèles de programmation pour le processeur Cell. Par modèle de programmation on entend outil de déploiement de code ou de parallélisation conçus pour le Cell. Les approches citées dans ce qui suit sont celles qui nous ont paru pertinentes et assez mures pour pouvoir être utilisées dans notre domaine d'applications. Certaines approches reprennent des outils existants pour les architectures parallèles à mémoire partagée ou distribuée qui ont été adaptés pour l'architecture et la hiérarchie mémoire du Cell.

2.1 Les Threads POSIX

Les *threads* POSIX¹ ou *Pthreads* sont une standardisation[12] du modèle de programmation par *threads* pour les systèmes UNIX. Ce modèle est basé sur une API de programmation parallèle qui permet la gestion des *threads* ainsi que la synchronisation par *mutex* ou variables conditionnelles. Historiquement, les concepteurs de *hardware* ont développé leurs implémentations propriétaires des *threads*, ceci a rendu la portabilité du code des programmeurs quasi impossible. La nécessité d'une API standard est donc devenue vitale, c'est pour cette raison que la majorité des vendeurs de *hardware* possèdent actuellement leur implémentation standard des *threads* POSIX. Les *Pthreads* sont définis autour d'un ensemble de procédures et de types en langage C contenus dans le fichier d'entête "pthread.h".

D'une manière générale un *thread* est un flux d'instructions pouvant s'exécuter de ma-

¹Portable Operating System Interface for Unix

nière indépendante sur un OS donné. Du point de vue du programmeur ceci s'apparente à une procédure qui peut s'exécuter indépendamment du programme principal, un programme contenant des procédures de ce type est dit *multi-threaded*. Afin de détailler le principe de fonctionnement des *threads* il est nécessaire de faire un rappel sur les *process* UNIX. Un *process* est créé par l'OS et contient un certain *overhead* qui consiste en certaines ressources nécessaires à son exécution. Les *threads* résident à l'intérieur de ces ressources et sont capables d'être ordonnancés et exécutés en tant qu'entités indépendantes car ils ne dupliquent qu'une partie des ressources qui leurs permettent d'être des morceaux de code exécutables. Le flot de contrôle est rendu indépendant car le *thread* possède ses propres : pointeur de pile, registres, propriétés d'ordonnancement (priorité et politique), ensemble de signaux et données spécifiques. En somme, un *thread* existe dans un *process* dont il utilise les ressources. Il possède son propre flot de contrôle et ne duplique que les ressources nécessaires à son exécution indépendante. Il peut partager les ressources du *process* avec d'autres *threads* et s'exécuter en coordination avec ces derniers. La complexité due à sa création et à sa gestion est légère comparativement à celle du *process* et sa durée de vie est celle de son *process* parent.

2.1.1 l'API Pthread

L'API des *threads* POSIX peut être décomposée en trois types de routines :

- **Les routines de gestion des *threads*** : comprends les tâches de création, propriétés d'exécution et terminaison des *threads*.
- **Les *mutex*** (abréviation de *mutual exclusion*) : ils permettent de synchroniser les *threads*, les routines gèrent la création, destruction, réservation et libération des *mutex*.
- **Les variables conditionnelles** : ces dernières gèrent la communication entre des *threads* qui partagent les *mutex*. Elles sont basées sur des conditions fixées par le programmeur, elles incluent des fonctions de création, destruction, attente et signalisation basées sur certaines valeurs de ces variables.

Gestion des Threads

- **Création et Terminaison des *threads*** : Initialement le programme contient un seul *thread* qui est le `main()`. Les autres *threads* doivent être créés explicitement par le programmeur. `pthread_create` créé un nouveau *thread* et le rend exécutable, un exemple de code à base de *Pthreads* est donné dans le listing 2.1. Cette routine peut être appelée autant de fois que l'on veut et à n'importe quel endroit dans le code. Le nombre de *threads* maximal créé par un *process* dépend de l'implémentation. Une fois créés les *threads* peuvent créer à leur tour d'autres *threads* et il n'existe aucune dépendance ni hiérarchie entre les *threads*. Le *thread* est créé avec certains attributs par défaut, ceux-ci pouvant être changés ultérieurement. Il existe plusieurs manières de terminer un *thread* : soit par le *thread* lui-même qui le fait par un `return` de sa routine principale ou par la fonction `pthread_exit()`, ou alors par un autre *thread* en utilisant `pthread_cancel()` et enfin en cas de terminaison du *process* parent.
- **Passage d'Arguments au *threads*** : On peut passer un argument au *thread* via la routine `pthread_create()`. On peut également passer plusieurs arguments en les rassemblant dans une structure et en passant l'adresse de celle-ci.
- **Jonction et Détachement des *threads*** : La jonction est une manière de faire une synchronisation entre les *threads*, la fonction `pthread_join()` bloque le *thread* appelant jusqu'à ce que le *thread* appelé termine son exécution. Le caractère joignable ou pas d'un *thread* est spécifié à sa création : s'il est créé en tant que *thread* détaché il ne pourra pas être joignable. La routine `pthread_detach()` sert à détacher un *thread* qui était joignable à sa création.
- **Gestion de la Pile** : Le standard ne définit pas la taille par défaut de la pile du *thread*, celle-ci dépend de l'implémentation. Toutefois, le programmeur peut en spécifier la taille ainsi que l'emplacement de la mémoire dans laquelle elle doit être stockée.

Les Variables *mutex*

Les *mutex* sont un des principaux mécanismes de synchronisation de *threads*. Ils permettent par exemple de synchroniser des *threads* ou alors de protéger des données partagées en cas d'écriture simultanée. Un *mutex* peut être réservé (*lock*) par un seul *thread* à un moment donné, le propriétaire du *mutex* est le seul à le posséder : tout autre

thread qui essaye de réserver ce même *mutex* échoue jusqu'à ce que le propriétaire le libère (*unlock*). Il existe des routines de création et de destruction des *mutex*, la réservation des *mutex* se fait soit par appel à la routine `pthread_mutex_lock()` (bloquant), `pthread_mutex_trylock()` (non bloquant) et se libère par `pthread_mutex_unlock()`. Parmi les exemples d'utilisation des *mutex* on peut citer les cas de *race condition* où plusieurs *threads* essayent de mettre à jour une variable globale, il est nécessaire dans ce genre de situation de protéger cette variable par un *mutex* afin que celle-ci ait la même valeur du point de vue de tous les *threads*. On dit alors que l'on crée une *section critique*.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello (void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Listing 2.1 – Exemple de code *Pthread* basique montrant les routines de création de *threads*

Les Variables de Condition

Les variables de condition fournissent aux *threads* une autre manière de se synchroniser. Contrairement aux *mutex* qui sont basés sur le contrôle de l'accès à une variable, la synchronisation par variable de condition se fait selon une valeur spécifique. Les variables de condition sont utilisées en conjonction avec les *mutex*. L'appel à la fonction `pthread_cond_wait()` bloque le *thread* appelant jusqu'à ce que la condition spécifiée est signalée. Cette routine doit être appelée tant que le *mutex* est réservé et libère automatiquement le *mutex* quand elle est en attente. Une fois que le signal est reçu, le *thread* est réveillé et le *mutex* est réservé automatiquement pour être utilisé par le *thread*. La responsabilité de libérer le *mutex* est à la charge du programmeur qui le fait une fois que son utilisation n'est plus nécessaire. La fonction `pthread_cond_signal()` est utilisée pour signaler (ou réveiller) un autre *thread* qui est en attente de la variable de condition. Elle doit être appelée après que le *mutex* est réservé et doit libérer le *mutex* dans l'ordre pour permettre à `pthread_cond_wait()` de s'achever. Il existe une routine nommée `pthread_cond_broadcast()` qui met en attente plusieurs *threads* en même temps.

2.1.2 Les Threads POSIX sur le Cell

Sur un système Linux pour le Cell, le *thread* principal s'exécute sur le PPE, celui-ci pouvant produire un ou plusieurs tâches sur le processeur. Une tâche peut contenir un ou plusieurs *threads* Linux qui peuvent s'exécuter soit sur le PPE soit sur le SPE. Un *thread* qui s'exécute sur le SPE possède son propre contexte incluant un banc de registre de 128 x 128-bit, un compteur de programme et une file d'attente de commandes MFC, et il peut communiquer avec d'autres unités d'exécution au travers de l'interface des canaux MFC. Un *thread* PPE peut interagir directement avec un *thread* SPE via sa mémoire locale ou son espace de *problem state* ou indirectement via la mémoire centrale ou les routines la *SPE Runtime Management Library*. L'OS définit le mécanisme et la politique d'ordonnancement pour les SPE disponibles, il est également responsable de la gestion des priorités entre les tâches, du chargement du programme de la notification des événements au SPEs ainsi que du support du *debugger*. Une API de gestion des *threads* SPE similaire à la librairie POSIX a été conçue, dans le but de fournir à la fois un environnement de programmation familier et une flexibilité dans la gestion des SPEs. Cette API supporte à la fois la création et la terminaison des tâches SPE ainsi que l'exclusion mutuelle par des primitives de mise à jour atomiques. L'API peut accéder au SPE en utilisant un mo-

dèle virtuel dans lequel l'OS affecte dynamiquement les *threads* aux SPEs dans l'ordre de leur disponibilité. Les applications, peuvent spécifier de manière optionnelle un masque d'affinités pour affecter les *threads* à un SPE spécifique. Les dispositifs architecturaux de communication entre les *threads* et de synchronisation (mailbox, signaux, etc...) peuvent être accédés via un ensemble d'appels système ou alors via l'application qui mappe un bloc de contrôle du SPE dans l'espace mémoire de l'application. Sur le Cell il existe trois blocs de contrôle du SPE, un accédé par l'application, un autre par l'OS et un troisième par un superviseur. Une interface accessible à l'utilisateur permet la communication directe entre les processeurs SPEs ou PPE, ceci permet d'éviter des appels système couteux. Lorsque l'application fait une requête de création de *threads*, la librairie de *threads* SPE envoie la requête à l'OS pour allouer un SPE et créer un *thread* SPE à partir d'un fichier objet de format ELF (Executable and Linkable Format) intégrée dans un exécutable Cell. Le *miniloader* un programme SPE de 256-bit, télécharge la segment de code à exécuter sur le SPE, l'avantage de cette approche est d'une part d'éviter au PPE d'effectuer cette tâche et d'autre part de profiter du fait que les transferts PPE-SPE quand il se font du côté SPE, sont nettement plus efficace grâce à une interface qui contient plus de canaux de communications.

2.1.3 Processus de Génération de Code

Dans le modèle de programmation décrit ci-dessus le processus de génération de code binaire exécutable sur le Cell est dit *dual-source*. En effet, il existe deux code sources distincts, un code pour le SPE (`spe.c` sur la figure 2.1)) qui contient le code exécuté sur le SPE. Un deuxième code source qui est celui s'exécutant sur le PPE contient le thread maître qui gère les threads SPE. Le processus de génération de code exécutable est décrit dans la figure 2.1. Dans la première étape le code SPE est compilé ce qui donne un binaire exécutable SPE. Celui-ci est par la suite traité par un outils spécifique *spu-embedd* qui permet de transformer ce binaire en bout de code qui peut être enfouis dans l'exécutable du PPE. Cette procédure se fait par l'outil d'édition de lien qui considère alors le code SPE comme une librairie dont le code objet doit être intégré dans l'exécutable final.

2.1.4 conclusion

La programmation par *threads* sur le Cell est le modèle de base pour la mise en œuvre de code parallèle sur cette architecture. Il est caractérisé par une API très bas niveau qui

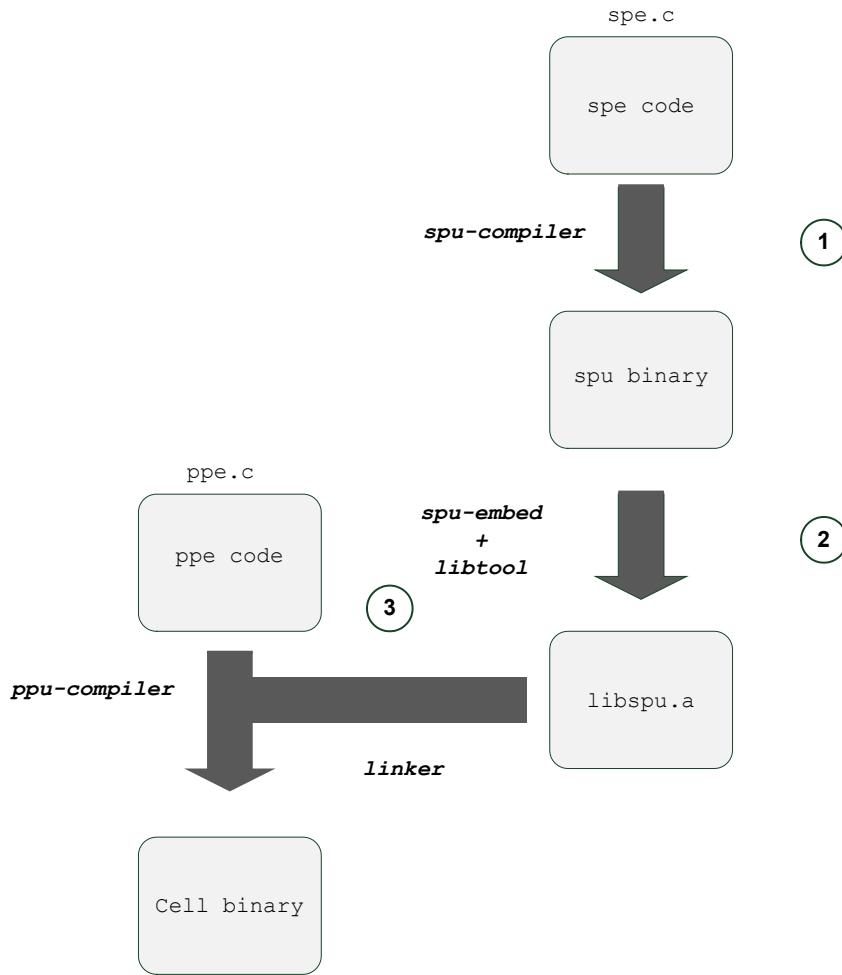


FIG. 2.1 – Processus *dual source* de génération de code exécutable pour le Cell

permet en même temps de garder un contrôle précis sur le déroulement de son application et d'avoir une grande flexibilité en terme de choix de déploiement d'un algorithme donné. Grâce au dispositifs architecturaux de signalisation et de synchronisation, l'interface est rendue très efficace en terme de performance sur le Cell. Toutefois, du point de vue du programmeur, la mise en oeuvre du code est sûrement plus laborieuse que pour d'autres modèles de programmation, mais celle-ci peut être justifiée dans le cadre de fortes contraintes sur les temps d'exécution ou dans le cas où le modèle de calcul SPMD n'est pas adapté à l'application déployée.

2.2 RapidMind

RapidMind[14] est un modèle de programmation parallèle multi-plateformes, GPU, multi-coeur symétrique et pour le processeur Cell, il relève du modèle de programmation *stream programming* et s'apparente à un langage de programmation enfoui dans C++. Il est basé sur la bibliothèque template C++ et une librairie de *runtime* qui effectue la génération dynamique de code. La librairie template permet l'invocation de code SPE à l'intérieur du code PPE, avec l'ensemble du code SPE écrit en template.

La librairie template de **RapidMind** fournit un ensemble de types de données, des macros de contrôle, des opérations de réduction et des fonctions communes qui permettent à la librairie de runtime de capturer une représentation du code SPE (*retained code*). Les types de données ont été spécialement conçus pour exprimer de manière simple les opérations SIMD et les exposer facilement à la librairie *runtime*. Le *runtime* à son tour extrait le parallélisme à partir de ces opérations en vectorisant le code et en divisant les calculs sur les tableaux et les vecteurs sur les différents SPEs. Il peut également effectuer des optimisations de boucle comme la détection des invariants de boucle. **RapidMind** assigne des tâches aux SPEs dynamiquement et peut effectuer des optimisations à plus haut niveau comme la superposition des calculs et des transferts qui permet de masquer la latence de ces derniers. Enfin, le modèle de calcul est un modèle SPMD, il diffère du modèle SIMD du fait que les programmes peuvent contenir du flot de contrôle et par le fait que celui-ci puisse gérer une certaine forme de parallélisme de tâches même si étant initialement un modèle *data-parallel*. Un exemple de code *RapidMind* est donné dans le listing *rapidmindcode*.

2.2.1 Modèle de Programmation et Interface

L'interface est basée sur trois types C++ principaux : `Value<N,T>`, `Array<D,T>` et `Program`, tous sont des conteneurs, les deux premiers pour les données et le dernier pour les opérations. Le calcul parallèle est effectué soit en appliquant des `Program` sur des `Array` pour créer de nouveaux `Array`, ou en appliquant une opération collective parallèle qui peut être paramétrée par un objet `Program` comme la réduction par exemple.

A première vue, les types `Value` et `Array` ne sont pas une grande nouveauté. En effet, tout développeur C++ a pour habitude d'utiliser les types N-tuples pour exprimer le calcul numérique sur des vecteurs, et le type `Array` est une manière usuelle d'encapsuler

la vérification des valeurs limites (*boundary checking*). Toutefois ces types constituent une interface pour une machine parallèle puissante basée sur la génération dynamique de code. Ceci est rendu possible grâce au type **Program** qui est la principale innovation du modèle de programmation **RapidMind**. Un mode d'exécution symbolique *retained* est utilisé pour collecter dynamiquement des opérations arbitraires sur les **Value** et les **Array** dans les objets **Program**.

le type **Value**

le type **Value**<N, T> est un N-tuple, les instances de ce type contiennent N valeurs de type T, où T peut être un type numérique de base (un flottant simple ou double précision ou tout autre type entier), les flottants 16-bits sont également supportés. Des notations courtes existent pour certaines tailles usuelles comme le **Value4f** pour un quartet de floats ou **Value3ub** pour un triplet d'entiers 8-bits non signés.

Les opérations arithmétiques standard et les opérations logiques sont surchargées pour les types tuples et opèrent composante par composante. Les fonctions de la bibliothèque standard sont également supportées, comme les fonctions trigonométriques et logarithmiques. En plus des opérations arithmétiques, des opérations de réorganisation des données ont été ajoutées au type **value** : ces opérations permettent la duplication d'une composante ou la permutation des composantes. Par exemple, si **a** est une valeur de type **Value**<4, float> qui représente une couleur RGBA, **a(2,1,0)** est l'inverse représentant le triplet BGR.

Les calculs sont exprimés en utilisant les tuples de **Value** et les opérateurs sur ces types peuvent être utilisés directement pour exprimer du parallélisme SWAR (SIMD Within A Register).

le type **Array**

Le type **Array**<D, T> est également un conteneur de données. Ce qui le distingue du type **Value** est le fait qu'il peut avoir plusieurs dimensions et que sa taille est variable. L'entier D représente la dimensionnalité (1, 2 ou 3), le type T est le type des éléments du conteneur. Le type des éléments est pour le moment restreint aux instances du type **Value**<N, T>.

Les instances du type **Array** supportent les opérateurs "[]" et "() pour l'accès aléatoire aux données. L'opérateur "[]" utilise des entiers en arguments tandis que l'opérateur "()"

utilise des coordonnées réelles comprises dans [0, 1] dans chaque dimension, cette particularité est utile par exemple pour les modes d'interpolation des images.

Les sous-tableaux peuvent être accédés en utilisant les fonctions **slice**, **offset** et **stride**. Les effets de bords sont gérés en utilisant la fonction membre **boundary**, qui inclut différents modes de traitement pour les bords. les types **Value** et **Array** suivent une sémantique par valeurs qui permet d'éviter l'aliasing de pointeurs et simplifie la programmation et l'optimisation. Il existe également d'autres types de sous-tableaux, les références sur tableaux et les accesseurs.

le type **Program**

Un objet **Program** contient une séquence d'opérations, ces opérations sont spécifiées par le passage en mode *retained* qui est indiqué par la macro mot-clé **BEGIN**. Normalement, le système fonctionne en mode *immediate*. Dans ce mode les opérations sur un tuple de valeurs s'exécutent à la spécification comme pour une bibliothèque matrice-vecteur classique : les calculs sont effectués sur la même machine que le programme hôte et le résultat est sauvegardé dans le tuple **Value** de sortie. En mode *retained* un nouvel objet **Program** qui est retourné par la macro **BEGIN** est créé. Les opérations dans ce mode ne sont pas exécutées ; elles sont symboliquement évaluées et sauvegardées dans l'objet **Program**. La sortie du mode *retained* est marquée par la macro **END**, qui ferme l'objet **Program** et le marque comme étant prêt à être compilé, étape à la suite de laquelle l'objet **Program** est utilisé pour le calcul. Les objets **Program** sont compilés de manière dynamique ce qui permet d'exploiter les caractéristiques bas-niveau de la machine cible. Il est à noter que même si les types **RapidMind** sont des classes C++, le compilateur est plutôt assimilable à un compilateur FORTRAN et peut ainsi effectuer les mêmes optimisations agressives. Les fonctionnalités du langage C++ sont utilisées pour structurer les calculs et générer le code mais pas lors de l'exécution.

2.2.2 Evaluation Partielle et Algèbre du Programme

Les objets **Program** sont des conteneurs d'opérations, et ces opérations peuvent être manipulées par le système de manière explicite. Cela permet l'implémentation de plusieurs dispositifs avancés de programmation.

```

// Reference Code
float f;
float a[512][512][3];
float b[512][512][3];
float func (float r, float s)
{
    return (r + s) * f;
}
void func_arrays( )
{
    for (int x = 0; x<512; x++)
        for (int y = 0; y<512; y++) {
            for (int k = 0; k<3; k++) {
                a[y][x][k] = func(a[y][x][k],b[y][x][k]);
            }
        }
    }
}

// RapidMind Code
#include <rapidmind/platform.hpp>
Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);
void func_arrays( )
{
    Program func_prog = BEGIN { // define program
        In<Value3f> r, s;
        Out<Value3f> q;
        q = (r + s) * f;
    } END;
    a = func_prog(a,b); // execute program
}

```

Listing 2.2 – Exemple de parallélisation de code avec *RapidMind*

En premier lieu, les **Program** peuvent être évalués partiellement. Si un objet **Program** **p** ayant **n** entrées, l'expression **p(A)** retourne un objet **Program** avec (**n-1**) entrées. En d'autres termes les entrées de l'objet **Program** ne sont pas obligatoirement fournies en une fois. Il est possible de binder toutes les entrées d'un objet **Program** mais de différer

son exécution effective. L'exécution d'un objet **Program** n'est déclenchée que quand il est affecté à un *bundle*, **Array** ou **Value**. L'opérateur "`()`" est utilisé pour binder les entrées de l'objet **Program**. Ceci est appelé le *tight binding*. Un changement de l'entrée après un *tight binding*, n'affecte pas les entrées d'un **Program**, même si son exécution est différée. Dans l'exemple précédent, où l'on crée "`p(A)`" et on l'affecte à un objet **Program** "`q`", et qu'on modifie ensuite l'entrée "`A`". Lors de l'exécution de "`q`", celui-ci utilisera la valeur de "`A`" au moment du binding dans "`p`", pas la valeur modifiée. Le *tight binding* permet l'optimisation de l'objet **Program** basée sur les valeurs effectives dans l'entrée bindée. Toutefois, le système supporte également le *loose binding*, spécifié par l'opérateur "`<<`". L'expression "`p<<A`" est similaire à "`p(A)`" sauf que les changements sur "`A`" sont visibles à l'exécution différée de "`p<<A`".

Une entrée à un **Program** peut être bindée à un **Array** ou un tuple de **Value**. Si des arrays de tailles différentes sont bindées à un **Program** la plus petite entrée est répliquée suivant les conditions aux bords pour avoir la taille de l'entrée de l'entrée la plus grande.

Les **Program** peuvent être combinés pour créer de nouveaux **Program** en utilisant deux opérations : la composition fonctionnelle est le *bundling*. Ces opérations forment une algèbre fermée dans l'ensemble des objets **Program**. L'opérateur de composition fonctionnelle ("`<<`") quand il est appliqué à deux objets **Program** "`p`" et "`q`" "`p << q`" transmet toutes les entrées du **Program** à droite de l'opérateur à l'entrée de celui de gauche, il crée un nouvel objet **Program** ayant les entrées de "`q`" et les sorties de "`p`". L'opérateur *bundle* quand à lui utilise la fonction "**bundle**". Cette fonction concatène toutes les entrées/sorties de ses arguments dans l'ordre et crée un nouvel objet **Program** équivalent à la concaténation des sources de ces **Program** d'entrée en séquence.

Ces opérations combinées avec la compilation dynamique permettent une amélioration considérable des performances surtout quand le programme est dominé par des instructions de mémoire.

2.2.3 Les Opérations Collectives

L'opération de base supportée est l'application parallèle d'un programme sur un tableau de données. Toutefois, d'autres patterns de communication et de calcul sont supportés sous la forme d'opérations collectives. Les patterns de communication irréguliers sont fournis par les opérations de *scatter* et *gather*, et l'opération de réduction fournit un pattern de calcul hiérarchique.

L'opération *gather* permet de récupérer des données résidant dans des emplacements non-contigus de la mémoire et l'opération *scatter* l'écriture dans des zones de même nature. L'opération de réduction quand à elle est programmable, elle prend deux entrées et fournit une sortie. Elle permet par exemple de sommer les éléments d'un vecteur d'une manière hiérarchique (Fig. 2.2).

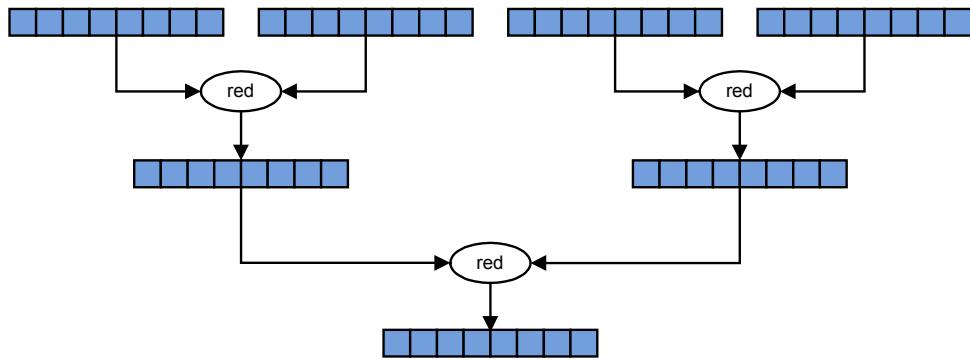


FIG. 2.2 – Illustration d'une opération collective de réduction

On pourra noter que l'opérateur impliqué dans la réduction doit être associatif. Parmi les opérateurs qui ont été implémentés on peut citer **sum**, **product**, **min** et **max**.

2.2.4 Spécificité du Backend pour le Cell de RapidMind

L'implémentation de RapidMind pour le Cell possède certaines particularités qui tiennent compte de l'architecture particulière de ce processeur. Le parallélisme SWAR peut être mappé directement sur les registres SIMD du SPE mais les opérations de permutation de données ne sont pas implémentées de manière aussi efficace que les autres. D'autre part la parallélisation qui consiste à appliquer un **Program** à un tableau permet en théorie de faire des millions de calculs en parallèle, mais le Cell ne possède qu'un nombre limité d'unités de traitement. C'est pour cela que les données sont subdivisées en plusieurs paquets et transférées dans les mémoires locales des SPEs avant d'être traitées. Le triple *buffering* est utilisé pour cacher la latence des transferts DMA. Pour les accès mémoire non réguliers un software-cache est utilisé. Si les programmes incluent un flux de contrôle, des tâches différentes peuvent prendre des temps d'exécution différents et

un système de *load balancing* (équilibrage de charge) est mis en place.

2.2.5 Conclusion

La plate-forme de développement RapidMind combine une interface basée sur la compilation dynamique et un modèle de calcul *data-parallel*. Elle peut être considérée soit comme une API de calcul parallèle ou un langage de programmation parallèle enfoui dans C++. Elle supporte plusieurs niveaux de parallélisme notamment le parallélisme SWAR et le parallélisme SPMD. Le modèle de programmation est commun à plusieurs plate-formes GPU, multi-coeur et Cell. C'est un modèle de programmation assez simple à utiliser et qui repose en grande partie sur un compilateur C++ ce qui lui confère une grande popularité auprès des utilisateurs. Au niveau des performances on peut relever de bon résultats dans [14] mais pour ce qui est de notre domaine d'applications qui est le traitement d'images, une analyse approfondie sera faite par la suite.

2.3 OpenMP

OpenMP pour le Cell[16] intégré dans le compilateur XL d'IBM est basé sur les transformations du compilateur et une librairie de *runtime*. Le compilateur transforme des pragmas OpenMP en code source intermédiaire qui implémente les constructions OpenMP correspondantes. Ce code inclut des appels aux fonctions de la librairie de *runtime* du Cell. Cette dernière fournit des fonctionnalités basiques à OpenMP incluant la gestion des *threads*, la répartition de la charge de travail ainsi que la synchronisation. Un exemple de parallélisation d'une boucle for est donné dans le listing 2.3.

Chaque segment de code compris dans une construction parallèle est listé par le compilateur dans une fonction séparée. Le compilateur insère les appels à la librairie de *runtime* OpenMP dans la fonction parente de la fonction listée. Ces appels aux fonctions de la librairie de *runtime* vont ainsi invoquer les fonctions listées et gérer leur exécution.

Le *framework* est basé sur le compilateur IBM XL. Ce dernier possède des *front-end* pour C/C++ et FORTRAN, et contient la même infrastructure d'optimisation pour ces langages. Le *framework* d'optimisation se divise en deux composants TPO et TOBEY. TPO est chargé des optimisations haut niveau indépendantes de la machine cible tandis que TOBEY effectue les optimisations bas-niveau spécifiques à l'architecture.

Le compilateur résulte d'une adaptation de versions existantes du compilateur XL suppor-

tant OpenMP, mais la spécificité de l'architecture du Cell a posé quelques problématiques qui sont les suivantes :

- **threads et Synchronisation** : les *threads* s'exécutant sur le PPE diffèrent de ceux du SPE en termes de capacité de traitement. Le système a été conçu pour prendre en compte la différence entre les deux architectures.
- **Génération de Code** : Le jeu d'instruction du PPE diffère de celui du SPE. Il en résulte que l'optimisation du code PPE est faite séparément de celle du SPE. L'espace de stockage sur le SPE étant limité, le code SPE s'il excède cette capacité, peut être partitionné en sections binaires (*overlays*) au lieu d'une section monolithique. De plus, les données partagées dans le code SPE nécessitent un transfert DMA de la mémoire centrale vers la mémoire locale. Ceci est fait soit par le compilateur qui insère explicitement des commandes DMA dans le code, soit par un mécanisme de software cache qui fait partie de la librairie de *runtime* du SPE.
- **Modèle Mémoire** : le hardware du Cell assure que les transactions DMA sont cohérents, mais ne fournit pas de mécanisme de cohérence pour les données résidant dans la mémoire locale, le modèle mémoire de OpenMP implémenté assure une cohérence de données qui est requise par les spécifications.

```
#include <omp.h>
#define CHUNKSIZE 100
#define N      1000

main ()
{
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
```

```

for ( i=0; i < N; i++)
    c[ i ] = a[ i ] + b[ i ];

} /* end of parallel section */

}

```

Listing 2.3 – Exemple de code *OpenMP* où une boucle for est parallélisée

Les sections qui suivent décrivent la manière avec laquelle ces problèmes ont été traités.

2.3.1 *threads* et Synchronisation

les *threads* peuvent s'exécuter sur le PPE ou les SPE. Le *thread* maître est toujours exécuté sur le PPE. Celui-ci est responsable de la création des autres *threads*, de la répartition et de l'ordonnancement des tâches, et des opérations de synchronisation. L'absence d'OS sur le SPE fait que le PPE gère toutes les requêtes OS. Cette répartition permet au SPE de se consacrer uniquement aux tâches de calcul.

Actuellement, un seul *thread* est sensé s'exécuter sur le PPE, est le nombre de *threads* parallèles exécutés sur les SPEs se déclare par la variable d'environnement OMP_NUM_THREADS. La création et synchronisation des *threads* est implémentée à l'aide des librairies du SDK (Software Development Kit). Le *thread* sur le PPE crée des *threads* SPE au runtime seulement quand les structures parallèles sont rencontrées pour la première fois.

Pour les niveaux de parallélisme nichés (boucles nichées), chaque *thread* dans la région parallèle la plus externe exécute séquentiellement la région parallèle interne. les itérations de boucles sont divisées en autant de morceaux qu'il y a de *threads*, avec un mécanisme d'ordonnancement et de synchronisation simplifié. Lorsque le *thread* SPE est créé, il effectue des initialisations et entre dans une phase d'attente d'affectation de tâches de la part du PPE, exécute ces tâches et se met en attente d'autres tâches, jusqu'à ce qu'il reçoive un signal de fin de tâche. Une tâche SPE peut être l'exécution d'une région parallèle listée (boucle ou section), ou alors l'exécution d'un flush de cache ou encore la participation à une opération de synchronisation par barrière.

Il existe une file d'attente dans la mémoire système correspondant à chaque *thread*. Quand le *thread* maître assigne une tâche à un *thread* SPE, il écrit les informations sur cette tâche sur la file d'attente qui lui est consacrée, incluant le type de tâche, les bornes supérieure et inférieure de la boucle parallèle, ainsi que le pointeur de fonction pour la région de code listée qui doit être exécutée. Une fois que le *thread* SPE prend une tâche

de la file d'attente, il signale au *thread* maître que l'espace dans la file d'attente est à nouveau libre. Les mécanismes de synchronisation sont assurés au travers de mailbox qui permettent l'échange de messages bloquant ou non-bloquant entre le *thread* maître et les *threads* de calcul. Les *locks* OpenMP sont implémentés grâce au commandes DMA atomiques.

2.3.2 Génération de Code

En premier lieu, le compilateur sépare chaque région dans le code source qui correspond à une construction OpenMP parallèle, et la liste dans une fonction séparée. La fonction peut prendre des paramètres supplémentaires comme les bornes supérieure et inférieure de la boucle. Le compilateur insère un appel à la librairie de *runtime* OpenMP au niveau de la fonction parente de la fonction listée, et insère un pointeur dans cette fonction de la librairie de *runtime*. Le compilateur insère également des instructions de synchronisation quand c'est nécessaire.

Le fait que l'architecture du Cell soit hétérogène impose que les fonctions listées contenant des tâches parallèles soit clonées afin d'être exécutables aussi bien par le SPE que le PPE. Le clonage est effectué quand le graphe d'appel global est disponible de telle sorte que le sous-graphe d'un appel à une fonction listée puisse être entièrement cloné. Le clonage permet aussi lors des étapes ultérieures d'effectuer des optimisations qui dépendent de l'architecture comme la vectorisation de code qui ne peut pas se faire dans une étape commune à cause des différences entre les jeux d'instructions SPU et VMX. Une table de mise en correspondance entre les versions PPE et SPE contient les pointeurs des fonctions listées de telle sorte à ce qu'il n'y ai pas de confusion lors de l'exécution. A la fin de l'étape TPO, les procédures sur les différentes architectures sont séparées en deux unités de compilation différente et celles-ci sont traitées une par une par le *backend* TOBEY.

L'unité PPE ne requiert pas de traitement particulier. Par contre l'unité compilée SPE peut produire un binaire d'une taille importante et qui ne tiens pas dans la mémoire locale. Il existe deux approches pour remédier à ce problème. La première consiste au partitionnement de la section parallèle dans un programme en plusieurs sections de taille moindre et la génération d'un binaire distinct pour chaque sous section. Cette approche est limitée, d'une part parce qu'une sous-section peut avoir une taille pas assez petite pour tenir dans la mémoire locale et d'autre part la complexité générée par la création

et la synchronisation de plusieurs *threads* affecte considérablement les performances. La deuxième approche qui est celle utilisée dans le compilateur IBM XL, est le partitionnement du graphe d'appel et les *overlays* de code. Le code SPE est ainsi partitionné et un code *overlays* est créé pour chaque partition. Ces *overlays* partagent l'espace d'adresses mais n'occupent pas la mémoire locale en même temps. Un poids est affecté à chacun des arcs du graphe représentant la fréquence d'appels de la fonction. Le graphe d'appel est partitionné afin de maximiser cette fréquence d'appel dans une partition en utilisant l'algorithme du *maximum spanning tree*. Le code SPE ainsi généré est intégré dans le code PPE avant d'être exécuté.

2.3.3 Modèle Mémoire

OpenMP spécifie un modèle mémoire *relaxed-consistency, shared memory*. Ce modèle permet à chaque *thread* d'avoir sa propre vue temporaire de la mémoire. Une valeur écrite dans une variable, ou une valeur lue à partir d'une mémoire peut rester dans la vue temporaire du *thread* jusqu'à ce qu'elle soit obligée de partager la mémoire par une opération de flush OpenMP.

Ce modèle est adapté au Cell car il prend en compte la mémoire limitée des SPEs. Les données privées accédées dans le code SPE sont allouées en mémoire privée. Les variables partagées sont elles allouées en mémoire centrale et peuvent être accédées via DMA par les SPEs. Deux mécanismes distincts sont utilisés pour les transferts DMA : le *static-buffering* et le *software-cache* contrôlé par le compilateur. Dans les deux cas, les données globales peuvent avoir une copie dans la mémoire locale SPE.

Certaines références sont considérées comme étant régulières du point de vue du compilateur. Ces références interviennent dans les boucles, les adresses mémoires vers lesquelles elle pointent peuvent être exprimées en utilisant des expressions affines de variables d'induction de la boucle, et la boucle qui les contient ne possède aucune dépendance induite par la boucle (vraie, de sortie ou anti-dépendance) impliquant ces références. Pour ces références régulières aux données partagées, un buffer temporaire est utilisé dans le SPE. Des opération DMA *get* et *put* sont utilisées respectivement pour lire et écrire de et vers ce buffer à partir de la mémoire centrale. Plusieurs buffers peuvent être utilisés afin de recouvrir les calculs par des transferts mémoire.

Pour les références irrégulières à la mémoire le *software-cache* est utilisé. Le compilateur

remplace les *LOAD* et *STORE* de et vers ces zones mémoire par des instructions qui vont chercher les adresses effectives, dans le répertoire du cache. Si une ligne de cache pour l'adresse effective est trouvée (*cache hit*) la valeur dans le cache est utilisée. Dans le cas contraire (*cache miss*) la donnée est récupérée en mémoire via un DMA *get* dans le cas d'une lecture.

La taille de la ligne de cache (128 bytes) et son associativité (4) sont choisies respectivement pour optimiser les transferts DMA et exploiter le jeu d'instructions SIMD pour le calcul des adresses (4x 32-bits). Le système assure également au SPE l'accès à des données qui seraient dans la pile d'une fonction PPE qui appelle une fonction SPE.

2.3.4 Conclusion

Le modèle de programmation OpenMP intégré dans le compilateur IBM XL pour le processeur Cell est une approche qui a le mérite de permettre à l'utilisateur de réutiliser son code OpenMP existant, sans se soucier des détails de l'architecture de Cell. Le support d'OpenMP est assuré par des transformations du compilateur couplés à une librairie de *runtime*. Les problématiques qui sont posées pour le portage d'OpenMP sur le Cell sont notamment celles de la synchronisation des *threads*, la génération de code et le modèle mémoire. La solution proposée est innovante car elle propose un compilateur qui permet de générer un seul binaire exécutable qui s'exécute sur des jeux d'instructions différents et sur un espace mémoire distribué. Les performances sur des benchmarks simples ainsi que sur des codes plus complexes donnés dans [16] démontrent l'efficacité de l'outil en comparaison avec un code optimisé à la main.

2.4 Cell SuperScalar

CellSS[2] est un environnement qui a pour objectif de fournir à l'utilisateur un outil de programmation simple mais qui donne des exécutables qui exploitent efficacement l'architecture du processeur Cell. Il découle d'un modèle de programmation nommé GRID[1] qui assimile un processeur superscalaire à une grille de calcul : les unités fonctionnelles du processeur sont les ressources de la grille, les données contenues dans les registres correspondent aux fichiers dans la grille et les instructions assembleur sont assimilées aux tâches de calcul.

Cell SuperScalar est constitué de deux composantes clés un compilateur *source-to-source*

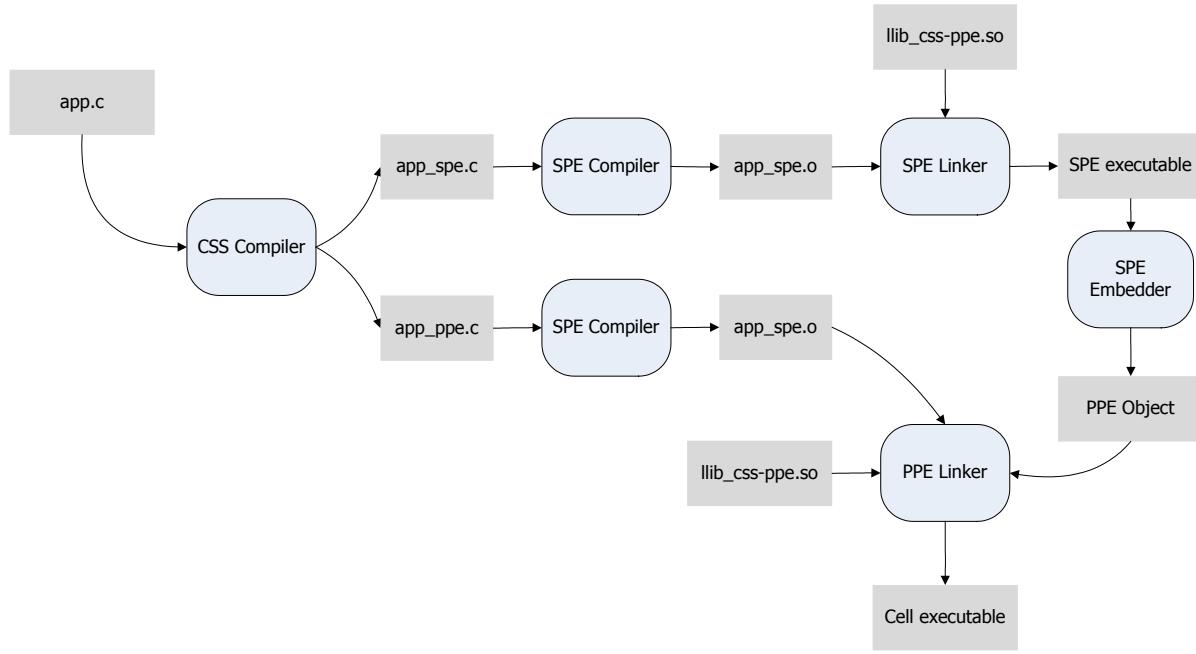


FIG. 2.3 – Procédure de génération de code de CellSS

et une librairie de *runtime*. Le processus de génération de code est illustré dans la figure 2.3. Partant d'un code source séquentiel écrit en langage C, des annotations en CellSS sont insérées dans le code. Le compilateur *source-to-source* est utilisé pour générer deux fichiers C distincts. Le premier correspond au programme principal de l'application, il est compilé par un compilateur PPE qui crée un objet pour ce même processeur. Le deuxième code source correspond à celui exécuté par le SPE sous contrôle du PPE. Cet exécutable est enfouis dans le binaire du PPE pour pouvoir être exécuté. Cette procédure est la même que celle utilisée par le SDK d'IBM qui est basé sur deux compilateurs distincts et une phase d'intégration du code SPE dans celui du PPE.

L'exécution du programme est assurée par le PPE qui assigne les *task* aux SPEs au travers de la librairie de *runtime*. Le *runtime* se charge de créer un noeud qui correspond à la *task* dans un graphe, et vérifie la dépendance avec une autre *task* lancée auparavant et ajoute un arc entre les deux. Si la *task* courante est prête à être exécutée le *runtime* envoie une requête au SPE pour qu'il se charge de l'exécution. Les transferts DMA sont gérés par le *runtime* de manière transparente. Les appels au *runtime* ne sont pas bloquants et de ce fait si une *task* n'est pas prête ou tous les SPEs sont occupés, le programme principal continue son exécution.

On pourra noter que tout le processus (assignation de tâches, analyse des dépendances,

transfert de données) sont transparents du point de vue de l'utilisateur, qui écrit un code séquentiel dans lequel il annote la partie à exécuter par les SPEs. Le système peut changer dynamiquement le nombre des SPEs utilisés en prenant en compte le maximum de concurrence contenu dans l'application à chaque phase. Un exemple de code *CellSS*.

```

float *A[NB][NB];
#pragma css task inout(diag[B][B])
void lu0(float *diag){
    int i, j, k;
    for (k=0; k<BS; k++)
        for (i=k+1; i<BS; i++) {
            diag[i][k] = diag[i][k] / diag[k][k];
            for (j=k+1; j<BS; j++)
                diag[i][j] -= diag[i][k] * diag[k][j];
        }
    }
#pragma css task input(diag[B][B]) inout(row[B][B])
void bdiv(float *diag, float *row){
    ...
}
#pragma css task input(row[B][B], col[B][B]) inout(inner[B][B])
void bmod(float *row, float *col, float *inner){
    ...
}
#pragma css task input(diag[B][B]) inout(col[B][B])
void fwd(float *diag, float *col){
    ...
}
void write_matrix (FILE * file, float *matrix[NB][NB])
{
    int rows, columns;
    int i, j, ii, jj;
    fprintf (file, "%d\n %d\n", NB * B, NB * B);
    for (i = 0; i < NB; i++)
        for (ii = 0; ii < B; ii++){
            for (j = 0; j < NB; j++)
#pragma css wait on(matrix[i][j])
            for (jj = 0; jj < B; jj++)
                fprintf (file, "%f ", matrix[i][j][ii][jj]);
        }
    fprintf (file, "\n");
}

```

```

    }
}

int main(int argc, char **argv) {
    int ii, jj, kk;
    FILE *fileC;
    E
    initialize (A);
#pragma css start
    for (kk=0; kk<NB; kk++) {
        lu0(A[kk][kk]);
        for (jj=kk+1; jj <NB; jj++)
            if (A[kk][jj] != NULL)
                fwd(A[kk][kk], A[kk][jj]);
        for (ii=kk+1; ii<NB; ii++)
            if (A[ii][kk] != NULL) {
                bdiv (A[kk][kk], A[ii][kk]);
                for (jj=kk+1; jj <NB; jj++)
                    if (A[kk][jj] != NULL) {
                        if (A[ii][jj]==NULL)
                            A[ii][jj]=allocate_clean_block();
                        bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
                    }
            }
    }
    fileC = fopen (argv[3], "w");
    write_matrix (fileC, A);
    fclose (fileC);
#pragma css finish
}

```

Listing 2.4 – Exemple sparse LU avec *CellSS*

2.4.1 runtime

Au *runtime*, les appels à une fonction *Execute* seront responsables du comportement de l’application sur le processeur Cell. Pour chaque appel à la fonction *Execute* le *runtime* effectue les actions suivantes :

- L’addition d’un noeud dans un graphe des tâches qui représente la tâche appelée.
- L’analyse des dépendances de données de la nouvelle tâche avec les tâches appelées précédemment. Cette analyse prend comme hypothèse que deux paramètres sont

les mêmes s'ils ont la même adresse. Le système cherche les types de dépendances de données *RaW*, *WaR* et *WaW*².

- Le renommage de paramètres similaire au renommage de registres, une technique issue des processeurs superscalaires, le renommage se fait sur les paramètres *output* et *input/output* pour chaque appel de fonction qui possède un paramètre qui va être écrit, au lieu d'écrire dans l'adresse originale de celui-ci, un emplacement mémoire nouveau sera utilisé, celui-ci sera un renommage de l'emplacement du paramètre original. Ceci permet l'exécution de la fonction indépendamment d'un appel précédent à une fonction qui écrit ou lit ce paramètre. Cette technique permet de ce fait de supprimer efficacement toutes les dépendances *WaR* et *WaW* en utilisant de l'espace mémoire supplémentaire et simplifie ainsi le graphe des dépendances et augmente les chances d'extraire du parallélisme.
- Enfin, sous certaines conditions, la tâche peut être exécutée.

Durant l'exécution de l'application le *runtime* maintient une liste des tâches prêtes. Une tâche est étiquetée comme étant prête, à partir du moment où il n'existe aucune dépendance entre cette tâche et d'autres tâches ou alors que ces dépendances ont été résolues (les tâches précédentes dans le graphe ont fini leur exécution). Le graphe de dépendances de tâches ainsi que la liste des tâches prêtes sont mis-à-jour à chaque fois qu'une tâche finit son exécution. Lorsqu'une tâche est terminée le *runtime* reçoit une notification et le graphe de tâche est vérifié pour établir les dépendances qui ont été satisfaites et les tâches dont toutes les dépendances ont été résolues et qui sont ajoutées dans la liste des tâches prêtes.

Étant donné une liste de tâches prêtes et une liste de ressources disponibles, le *runtime* choisit la meilleure correspondance entre les tâches et les ressources et soumet les tâches pour l'exécution. La soumission de la tâche comprend toutes les actions nécessaires pour pouvoir exécuter la tâche : le transfert des paramètres et la requête d'exécution de la tâche.

Middleware pour le Cell

Une application CellSS est composée de deux types de binaires exécutables : le programme principal, qui s'exécute sur le PPE et le programme tâche qui lui s'exécute sur le SPE. Ces binaires sont obtenus par compilation de deux sources générés par le compila-

²Read after Write, Write after Read and Write after Write

teur CellSS et les librairies de *runtime*. Lors du démarrage du programme sur le PPE le programmes de type *task* est lancé sur tous les SPEs durant l'exécution, celui-ci se met en attente de requêtes de la part du programme principal. Lorsque la politique d'ordonnancement choisit une *task* de la liste des tâches prêtes à être lancées et un SPE sur lequel elle va être exécutée, une structure de donnée nommée *task control buffer* est construite. Celle-ci contient des informations telles que l'identifiant de la tâche, l'adresse de chacun des paramètres ainsi que des informations de contrôle. L'identifiant sert à distinguer des tâches déjà présente dans la mémoire des SPEs de celles qui en le sont pas et qui doivent être chargées avant exécution. Les requêtes émanant du programme principal pour exécuter une tâche dans les SPEs se font via mailbox, celle-ci contient l'adresse et la taille du *task control buffer* correspondant à la tâche. L'exécution de la tâche se fait de la manière suivante : le SPE se met en attente d'une requête sur la mailbox, une fois la requête reçue il rapatrie les données et éventuellement le code de la tâche une fois la tâche finie, selon le contenu du *task control buffer* il garde les données dans sa mémoire ou les transfert en mémoire centrale. La synchronisation se fait à la fin de l'exécution et lorsque toutes les données résultantes sont transférées vers la mémoire centrale.

Exploitation de la Localité

Lorsque le graphe de dépendance de tâches construit par CellSS contient un arc allant d'un noeud vers un autre, il existe une dépendance de données entre les tâches qui impose un transfert de données, le but étant de minimiser la quantité de données transférées entre le PPE et les SPEs et entre SPEs. La politique d'ordonnancement est faite de telle sorte à exploiter la localité et donc à regrouper quand c'est possible les tâche interdépendantes dans le même SPE.

2.4.2 Conclusion

CellSS est un modèle de programmation basé sur un compilateur et un *runtime*. Il permet l'exécution d'un code source séquentiel sur une architecture parallèle grâce à l'annotation de ce code par des directives de parallélisation. Le *runtime* construit un graphe de dépendances des fonctions appelées et ordonne ses fonctions sur le SPE en gérant les transferts mémoire de manière transparente. L'algorithme d'ordonnancement exploite la localité afin de minimiser les transferts mémoire.

2.5 Sequoia

Sequoia[7] est un langage de programmation bas-niveau dédié aux machines modernes, que cela soit des processeurs parallèles ou alors superscalaires, et dans lesquels l'allocation de la mémoire et le transfert des données au travers de la hiérarchie mémoire est primordial pour la performance. **Sequoia** est une extension au langage C même si les constructions qu'il permet de faire sont très différentes du modèle de programmation du C. Il est basé sur un modèle de programmation qui assiste l'utilisateur dans la structuration des programmes parallèles efficaces au niveau de la bande passante qui restent portables sur de nouvelles machines. Le modèle de programmation repose sur quelques principes qui sont les suivants :

- La notion de mémoire hiérarchique est directement introduite dans le modèle de programmation ce qui permet un gain de portabilité et de performance. **Sequoia** s'exécute sur des machines qui sont représentées sous forme d'un modèle abstrait en arbre de modules mémoire distincts qui décrit comment les données sont transférées et où elles résident dans la hiérarchie mémoire.
- Les *task* sont utilisées comme une abstraction des unités de calcul auto-suffisantes qui incluent la description des communications et de la charge de travail. La *task* isole chaque calcul dans son propre espace mémoire local et contient le parallélisme.
- Afin de garantir la portabilité, une stricte séparation est maintenue entre l'expression générique de l'algorithme et les optimisations spécifiques à une machine donnée. Pour minimiser l'impact de cette séparation sur la performance les détails du déploiement sur une machine spécifique sont sous le contrôle de l'utilisateur.

Ainsi, **Sequoia** adopte une approche pragmatique pour fournir un outil de programmation parallèle portable en fournissant un ensemble limité d'abstractions qui peut être implémenté de manière efficace et sous contrôle de l'utilisateur.

2.5.1 Mémoire Hiérarchique

Le principe de mémoire hiérarchique est au cœur de l'outil **Sequoia**, car dans les systèmes modernes contenant plusieurs unités de traitement avec une hiérarchie mémoire à plusieurs niveaux, il est primordial de diviser un calcul de taille importante en des opérations plus petites pour atteindre des bonnes performances car cela permet d'exposer le parallélisme et d'atténuer l'effet de la latence d'accès à la mémoire car les données

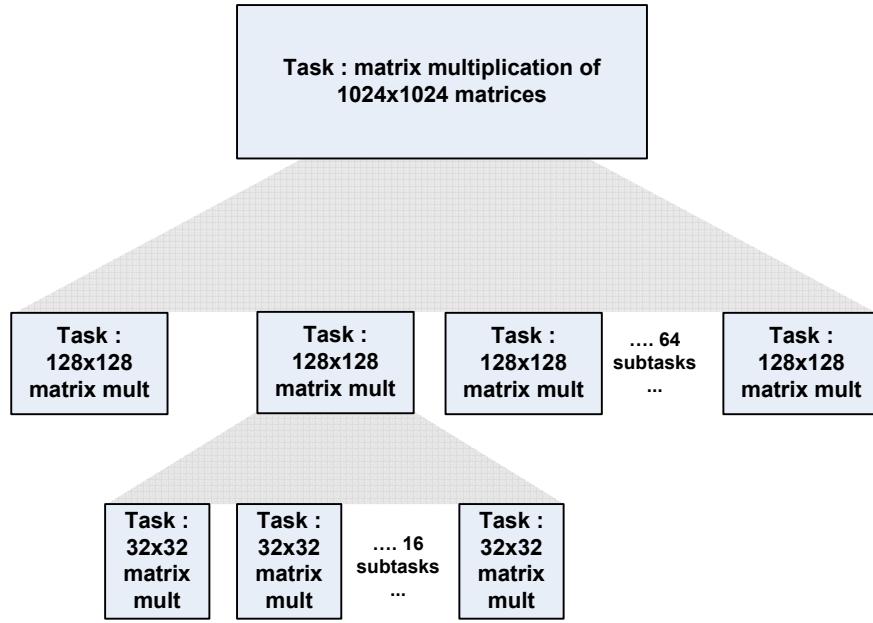


FIG. 2.4 – Multiplication de matrices de tailles 1024x1024 structurée en hiérarchie de tâches indépendantes effectuant des multiplications sur des blocs de données plus petits

sont physiquement proches des unités de traitement. Un exemple de découpage pour l’application produit de matrices est donné dans 2.4, dans cet exemple qui contient du parallélisme imbriqué et où la localité des données est primordiale. *Sequoia* requiert une telle réorganisation hiérarchique dans les programmes, qui a été inspirée de l’idée des *space-limited procedures* qui prône les stratégies *divide-and-conquer* tenant compte de la hiérarchie mémoire. Les *space-limited procedures* requièrent à chaque fonction dans une chaîne d’appels d’accepter des arguments occupant beaucoup moins d’espace mémoire que les fonctions appelantes. Un système complet a été implémenté autour de cette abstraction incluant un compilateur et un *runtime* pour le processeur Cell.

L’écriture d’un programme *Sequoia* implique la description abstraite d’une hiérarchie de tâches et le mapping de ces tâches sur la hiérarchie mémoire de la machine cible. Cela impose à l’utilisateur de considérer une machine parallèle comme un arbre de modules mémoire distincts. Les transferts de données entre les niveaux de la hiérarchie se font par blocs éventuellement asynchrones. La logique du programme décrit le transfert des données à tous les niveaux, mais les noyaux de calcul sont contraints de travailler que sur les données qui sont sur les noeuds feuilles (de niveau 0) de l’arbre représentant la

machine. La représentation abstraite du processeur Cell (Fig.2.5) contient des noeuds correspondants à la mémoire principale ainsi qu'à chaque mémoire locale du SPE. Un

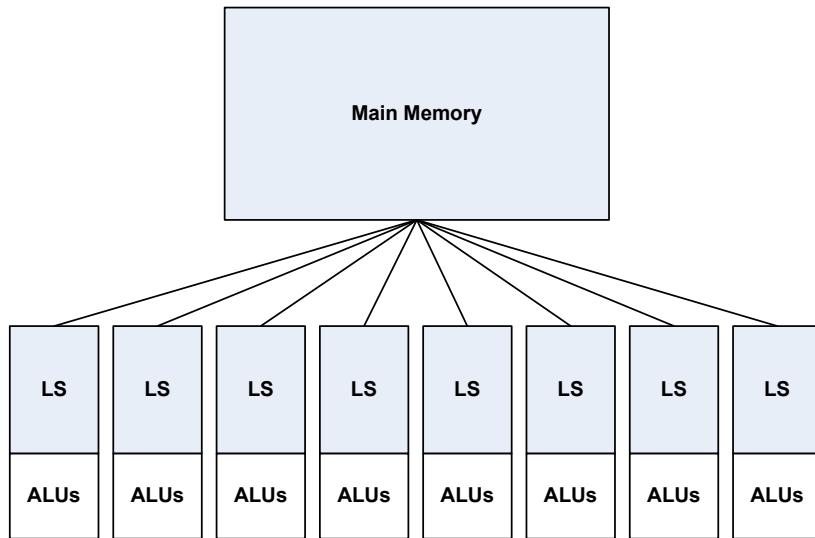


FIG. 2.5 – Modèle abstrait Sequoia du processeur Cell

code ***Sequoia*** ne fait pas de référence explicite à un certain niveau de la hiérarchie et les transferts de données entre les modules mémoire se font de manière implicite. Ainsi, les communications décrites dans ***Sequoia*** peut se faire au travers d'une instruction de prefetch, un transfert DMA ou un message MPI selon les spécificités de l'architecture cible. Ceci garantit la portabilité de l'application tout en bénéficiant des performances des communications explicites. Il y a certaines machines qui possèdent une topologie qui n'est pas facilement représentable sous forme d'arbre c'est pour cela que la notion de *virtual level* (niveau virtuel) a été introduite dans ***Sequoia***, ce niveau ne correspond à aucune mémoire physique. Ce niveau permet par exemple de représenter l'aggrégation des mémoires locales des SPEs sur processeur Cell. Cela permet de ce fait d'encapsuler les communications horizontales inter-noeud tout en gardant le modèle d'abstraction en arbre qui lui ne permet que des communications verticales.

2.5.2 Modèle de Programmation

La notion de *task* est au cœur du modèle de programmation de *Sequoia* : c'est une fonction exempt de tout effet de bord avec une sémantique de passage de paramètre par valeur. Les propriétés qui seront énoncées dans la suite garantissent la portabilité du modèle sans sacrifier les performances. Un exemple de code de multiplication de matrices par blocs est donné dans le listing 2.5.

Communication Explicite et Localité

La définition d'une tâche exprime à la fois la localité et la communication dans un programme. Lorsqu'une tâche s'exécute, l'ensemble de toutes les données référencées doivent rester dans un seul noeud de l'arbre abstrait de la machine. Ainsi une tâche doit s'exécuter dans un endroit précis de la machine. Les pointeurs et les références ne sont pas permises à l'intérieur d'une tâche ce qui permet de dire que l'ensemble des données traitées par la tâche est contenu dans sa définition.

L'implémentation contient un appel récursif dans lequel un sous-ensemble des données est passé en paramètre. Les communications sont encapsulées par les tâches en utilisant une sémantique de passage de paramètre dit *call-by-value-result* qui est un passage de paramètres par valeur dans lequel les copies locales des données sont réécrites dans l'espace global à la fin de l'appel de la fonction. Chaque tâche s'exécute dans son espace d'adresses local, toutes les données d'entrée des tâches appelantes sont copiées dans l'espace mémoire de la fonction appelée et les résultats sont recopiées vers l'espace mémoire de la fonction appelantes après le retour de la fonction appelées.

Le mapping d'un programme Sequoia dicte quand une tâche appelée doit exécutée dans le même module mémoire que la tâche appelante ou alors assignées à une mémoire enfant.

Isolation et Parallélisme

La granularité du parallélisme dans *Sequoia* et la *tâche* et l'exécution parallèle résulte de l'appel de *tâches* concurrentes. Une tâche s'exécute généralement sur une partie de la boucle sous forme de plusieurs *sous-tâches* parallèles, chaque *sous-tâche* s'exécutant en isolation, une propriété qui garantit la portabilité et la performance. Une des contraintes imposées au modèles et que les tâches s'exécutant en parallèle ne peuvent pas coopérer

entre elles car elles n'ont aucun moyen de communiquer. Ceci limite le modèle de programmation au modèle SPMD mais évite le recours aux mécanismes de synchronisation couteux.

Décomposition de Tâche

Sequoia introduit des primitives de décomposition de tableaux et de mapping de tâches, elles sont décrites ci-dessous :

Sequoia Blocking Primitives

- **blkset**

Un objet Sequoia opaque représentant une collection de blocs de tableaux.

- **rchop(A, len0, len1, ...)**

Génère un **blkset** qui contient des blocks qui ne se recouvrent pas et qui tuilent le tableau multidimensionnel A. Chaque bloc est multidimensionnel de taille $\text{len0} \times \text{len1} \times \dots$

- **rchop(A, rchop_t(offset0, len0, stride0), ...)**

Généralisation de

- **rchop** qui génère des ensembles de blocs qui contiennent potentiellement des blocs qui se recouvrent. L'offset du tableau de départ, la taille du bloc, et le saut entre les blocs est spécifié pour toutes les dimensions du tableau source.

- **ichop(A, Starts, Ends, N)**

Génère un ensemble de blocs du tableau A de tailles non régulières. Les indices de départ et de fin du bloc sont donnés par les éléments dans le tableau de longueur N Starts et Ends.

- **gather(A, IdxBlkset)**

Génère un ensemble de blocs en rassemblant les éléments d'un tableau source A en utilisant des indices fournis dans les blocs de IdxBlkset. Le **blkset** résultant possède les mêmes nombre et taille que les blocs de IdxBlkset.

Sequoia Mapping Primitives

- `mappar(i=i0 to iM, j=j0 to jN ...)` ...

Une boucle `for` multi-dimensionnelle contenant uniquement un appel à une *sous-tâche* dans le corps de boucle. La tâche est mappée en parallèle en une collection de blocs.

- `mapseq(i=i0 to iM, j=j0 to jN ...)` ...

Une boucle multi-dimensionnelle contenant uniquement un appel à une *sous-tâche* dans le corps de boucle. La tâche est mappée séquentiellement en une collection de blocs.

- `mapreduce(i=i0 to iM, j=j0 to jN ...)` ...

Permet de faire le mapping en une collection de blocs, qui effectue une réduction sur au moins un argument de la tâche. Pour le support des réductions d'arbres parallèles, une tâche supplémentaires de recombinaison est requise.

Variantes de Tâches

Sequoia inclut deux types de tâches qui servent essentiellement à distinguer le code de mapping du code de calcul, elles sont décrites ci-dessous :

- *Inner Tasks* : ce sont les tâches qui appellent des sous-tâches. Elles n'ont pas d'accès direct à leurs arguments de type tableau mais elles passent aux sous-tâches sous forme de blocs. Les *Inner Tasks* utilisent les primitives de *mapping* et de *blocking* pour structurer les calculs sous forme de sous-tâches. La définition d'une *Inner Task* n'est associée à aucun module mémoire particulier de la machine, elle peu s'exécuter dans n'importe quel niveau de la hiérarchie mémoire dans lequel les données traitées tiennent.
- *Leaf Tasks* : ce sont des tâches qui ne font pas appel à des sous-tâches et qui opèrent directement sur des données résidant dans les niveaux feuilles de la hiérarchie mémoire.

Paramétrisation de Tâches

Les tâches sont écrites de manière à ce qu'elles soient paramétrables pour la spécialisation à de multiples machines cibles. La spécialisation est le processus de de création

d'instances d'une tâche qui est personnalisée pour s'exécuter sur un certain niveau de la hiérarchie mémoire de la machine. La paramétrisation des tâches permet à une stratégie de décomposition décrite par une variante de tâche d'être appliquée dans différents contextes, rendant la tâche portable sur différentes machines et sur différentes niveaux de la hiérarchie mémoire d'une cible donnée. L'utilisation de paramètres comme la taille des tableaux ainsi que les paramètres ajustable découpe l'expression d'un algorithme de son implémentation sur une machine donnée.

Spécialisation de Tâches et Tuning

Dans *Sequoia* on donne au programmeur le contrôle complet sur les phases de spécialisation et de tuning du code, au travers d'une phase dite de *task mapping and specification* qui est créée par l'utilisateur pour une machine donnée est maintenue indépendamment du code source. En plus, cette phase permet au programmeur de fournir des directives d'optimisation et de tuning qui sont propres à une cible donnée. Les spécification de mapping ont pour but de donner à l'utilisateur un contrôle précis sur le mapping d'une hiérarchie de tâches sur une machine en isolant les optimisations spécifiques à une cible donnée dans un autre endroit. Au lieu de confier le travail à un compilateur, *Sequoia* permet au programmeur d'optimiser son code lui-même afin d'obtenir les meilleures performances possibles.

```
void task matmul :: inner (in float A[M][P],  
                           in float B[P][N],  
                           inout float C[M][N])  
{  
    // Tunable parameters specify the size  
    // of subblocks of A, B, and C.  
    tunable int U;  
    tunable int X;  
    tunable int V;  
    // Partition matrices into sets of blocks  
    // using regular 2D chopping .  
    blkset Ablk = rchop(A, U, X);  
    blkset Bblk = rchop(B, X, V);  
    blkset Cblk = rchop(C, U, V);  
    // Compute all blocks of C in parallel.  
    mappar(int i=0 to M/U, int j=0 to N/V){
```

```

mapreduce (int k=0 to P/X){
    // Invoke the matmul task recursively
    // on the subblocks of A, B, and C.
    matmul(Ablk[i][k], Bblk[k][j], Cblk[i][j]);
}
}

void task matmul :: leaf(in float A[M][P],
                         in float B[P][N],
                         inout float C[M][N])
{
    // Compute matrix product directly
    for (int i=0;i<M; i++)
        for (int j=0;j<N; j++)
            for (int k=0;k<P; k++)
                C[i][j] += A[i][k]*B[k][j];
}

```

Listing 2.5 – Exemple de multiplication matricielle par blocs *Sequoia*

2.5.3 Implémentation

Dans l’implémentation de *Sequoia* un compilateur source-to-source génère du code C qui s’interface avec un *runtime* spécifique à la plate-forme. Les paramètres d’entrée du compilateur sont un programme *Sequoia* ainsi que des spécifications de mapping sur la machine cible.

Compilateur Cell et *runtime*

Les instances de tâches *Inner* correspondant à la mémoire principale sont exécutées par le PPE alors que les instances correspondant au niveau mémoire des Local Store sont exécutées par les SPE. Les codes sources PPE et SPE sont compilés séparément, un code binaire est ainsi combiné pour l’exécution, si toutefois le code SPE dépasse la capacité du local store il est découpé sous forme d’overlays. Le *runtime* est *event driven* : un système de notification via mailbox est mis en place entre le PPE et les SPEs, les tâches sont assignées par le PPE aux SPEs. Une fois que la notification est reçue un mécanisme du *runtime* charge le morceau de code à exécuter dans les SPEs et initie les transferts de données via DMA. Le système permet la superposition de transferts et de calculs afin d’améliorer

les performances. Les mécanismes de synchronisation ne sont pas constamment utilisées entre les tâches, un seul point de synchronisation est requis lors de la fin des sections parallèles, ce qui minimise l'overhead.

Optimisations

En plus de permettre au programmeur d'optimiser son implémentation des tâches dans *Sequoia*, certaines optimisations visant à utiliser efficacement la mémoire sont effectuées, notamment l'optimisation des transferts au niveau d'un même niveau la hiérarchie mémoire, ou les copies inutiles sont détectées et supprimées. D'autres optimisation incluant le *software-pipelining* et le déplacement des invariants de boucle sont effectués de manière statique à la compilation.

2.5.4 Conclusion

Sequoia est un modèle de programmation qui tente d'allier la portabilité avec la performance pour le portage d'algorithme sur des architectures parallèles. La performance est assurée par l'octroi à l'utilisateur d'une grande liberté dans l'optimisation du code de calcul qui s'exécute sur les noeuds au plus bas niveau de la hiérarchie mémoire. La portabilité est garantie par le fait que l'expression de l'algorithme est découpée de l'implémentation. L'expression explicite des communications, du mouvement des données au travers de la hiérarchie mémoire, du calcul parallèle et la définition d'un ensemble de travail isolé sont effectués grâce à une seule abstraction, la *tâche*. *Sequoia* fournit des primitives de structuration des calcul en tant que hiérarchie de tâches afin d'améliorer la localité et laisse au programmeur le soin d'optimiser la tâche pour une architecture donnée. Toutefois il existe des limitations, en l'occurrence pour le type de parallélisme qui doit être SIMD ce qui limite le schéma de déploiement et le champ d'applications. Aussi, le mapping des applications est fait de manière manuelle et n'est pas contenue dans le code source.

2.6 Squelettes Algorithmiques pour le Cell

La programmation parallèle structurée qu'on appelle programmation par squelettes algorithmiques [3] restreint l'expression du parallélisme à la composition d'un nombre prédéfini de *patterns* nommés squelettes. Les squelettes algorithmiques sont des briques

de base génériques, portables et réutilisables pour lesquelles une implémentation parallèle peut exister. Ils sont issues des langages de programmation fonctionnelle. Un système de programmation basé sur les squelettes fournit un ensemble fixe et relativement limité de squelettes. Chaque squelette représente une manière unique d'exploiter le parallélisme dans une organisation spécifique du calcul, tels que le parallélisme de données, de tâches, le *divide-and-conquer* parallèle ou encore le pipeline. En combinant ces *patterns* le développeur peut construire une spécification haut-niveau de son programme parallèle. Le système peut ainsi exploiter cette spécification pour la transformation de code, l'exploitant efficace des ressources ou encore le placement.

La composition des squelettes peut se faire d'une manière non-hiéarchique en mettant en séquence les différents blocs en utilisant des variables temporaires pour sauvegarder les résultats intermédiaires, ou alors de manière hiérarchique en imbriquant les fonctions squelette et ce en construisant une fonction composée dans laquelle le code de plusieurs squelettes est passé en paramètre d'un autre squelette. Ceci présente une manière élégante d'exprimer le parallélisme multi-niveau.

Dans un environnement de programmation declarative, comme dans les langages fonctionnels ou alors dans la programmation par squelettes, la composition hiérarchique procure au générateur de code plus de liberté de choix pour les transformations automatiques et l'utilisation efficace des ressources, comme par exemple le nombre de processeurs utilisé en parallèle dans un niveau particulier de la hiérarchie. les squelettes ne pouvant pas être imbriqués sont généralement implémentées avec juste une bibliothèque générique alors que les squelettes nichés requièrent un pré-traitement qui déroule la hiérarchie du squelette en utilisant par exemple les templates C++ ou les macros de préprocesseur en C.

2.6.1 Modèle de Programmation

Dans le modèle de programmation parallèle par squelettes algorithmiques, une application est définie comme étant un graphe de processus communicants. Cette représentation permet de spécifier les schémas de communication entre les processus P_i , de mettre en évidence les fonctions séquentielles F_i contenues dans l'application, ainsi que les processus nécessaires à l'exécution parallèle de l'application. Un exemple de représentation est donné dans la figure 2.6. On peut distinguer sur cette figure deux parties distinctes :

- Le système d'équilibrage de charge (**A**) qui utilise k processeurs qui traite les don-

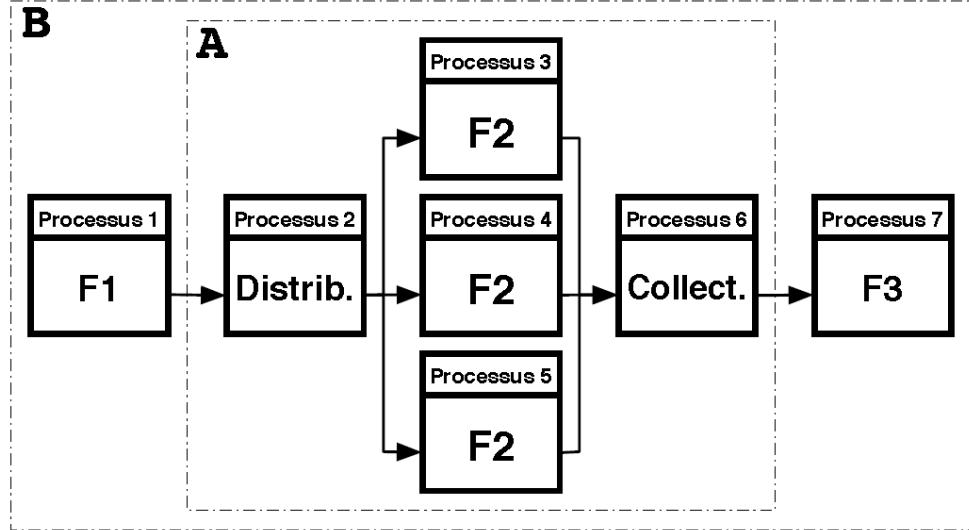


FIG. 2.6 – Exemple de graphe de processus communicants avec hiérarchisation

nées fournies par le processus *Distrib*, ces dernières étant regroupées par le processus *Collect*. Il faudra noter que le flux alimente le processeurs de manière dynamique dès qu'il trouve un processus libre.

- Le mécanisme de contrôle (A) qui permet de séquencer les traitement dans l'ordre donnée par le graphe. Ce mécanisme assure que les données, une fois traitées (fonction F_i) par le processeur P_i , sont transmises au processus P_{i+1} . On pourra noter que le schéma d'exécution dans ce cas là est du type *pipeline*.

Le modèle de programmation parallèle par squelettes algorithmiques repose sur l'extraction de tels schémas récurrents. Un squelette est ainsi défini comme étant un schéma générique paramétré par une liste de fonctions qu'il est possible d'instancier et de composer. Fonctionnellement, les squelettes algorithmiques sont des **fonctions d'ordre supérieur**, c'est à dire des fonctions prenant une ou plusieurs fonctions comme arguments et retournant une fonction comme résultat. La programmation par squelettes devient permet au programmeur d'utiliser un modèle haut-niveau pour décrire son application, sans se soucier de certains détails complexes comme l'ordonnancement ou le placement. Il peut alors définir une application parallèle comme suit :

- Instancier des squelettes en spécifiant les fonctions qui les définissent.

- Exprimer la composition des ces squelettes.

L'expression de la compositions peut se faire en encodant cette dernière sous la forme d'un **arbre**(2.7) dont les noeuds représentent les squelettes utilisés et les feuilles, les fonctions séquentielles passées en paramètres à ces squelettes. Dans la figure 2.7, le

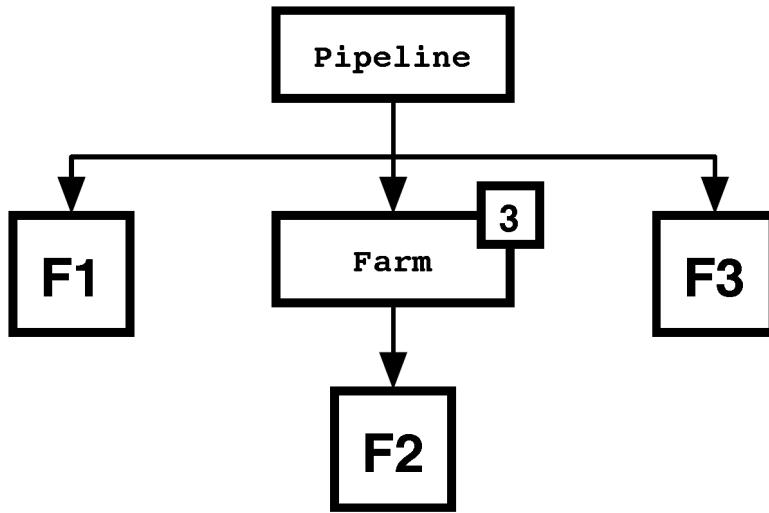


FIG. 2.7 – Arbre représentant le squelette en Figure 2.6

squelette **Pipeline** décrit le schéma générique correspondant à la section **B** du graphe de processus communiquant initial. Le squelette **Farm** représente quant à lui la partie **A** de ce même schéma. Les fonctions F_i apparaissent aux feuilles de l'arbre, c'est à dire en argument des squelettes. On note aussi que les fonctions **Distrib** et **Collect** n'apparaissent plus explicitement, car elles font partie intégrante du squelette **Farm**. Cette représentation met en avant un des aspects les plus importants de l'approche à base de squelettes algorithmiques : à partir d'un nombre restreint de squelettes (classiquement moins d'une dizaine), il est possible de définir des applications complexes. Ceci suppose toutefois que l'on ait formalisé le type d'application que l'on va chercher à paralléliser, de définir précisément le jeu de squelettes que l'on désire mettre à disposition du développeur et de spécifier leurs sémantiques fonctionnelles et opérationnelles. Il existe plusieurs classifications des squelettes. Toutefois, on peut les répartir en trois groupes : les squelettes dédiés au parallélisme de contrôle, les squelettes dédiés au parallélisme de données et les squelettes dédiés à la structuration séquentielle de l'application.

Squelettes dédiés au parallélisme de contrôle

Le Squelette Pipeline Ce squelette couvre les situations dans laquelle une liste de fonction qui doivent s'exécuter en série, est répartie sur un ensemble de processeurs différents. En régime permanent l'exécution de la fonction F_i sur les données D_{i+1} se fait alors en parallèle avec celle de la fonction F_{i+1} sur les données D_i . La figure 2.8 illustre ce fonctionnement en régime permanent. Le parallélisme résulte du fait que l'évaluation des

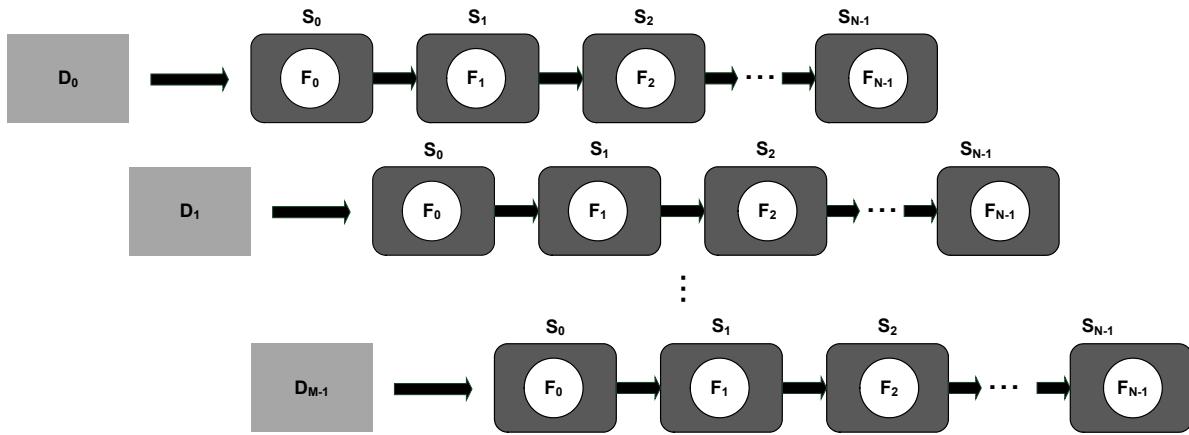


FIG. 2.8 – Exemple de squelette du type Pipeline

différents fonctions du **Pipeline** sur des éléments différents du flux (D_0 , D_1 par exemple sur la figure 2.8) se fait de manière indépendante. Deux grandeurs caractérisent alors le **Pipeline** : (1) la latence qui est la durée de traitement d'un élément de flux par tous les étages du pipeline, (2) le débit, qui mesure le nombre de résultats fournis par unités de temps et qui est déterminé par l'étage le plus lent du **Pipeline**.

Le Squelette Pardo Le squelette **Pardo** permet de placer de manière *ad hoc* N fonctions sur N processeurs (Fig. 2.9). Le schéma de communication est alors implicite. Ce type de squelette est fait pour faciliter la mise en oeuvre d'applications qui ne correspondent pas à un squelette bien défini. Le squelette **Pardo** est notamment utilisé pour rassembler plusieurs fonctions indépendantes opérant sur un flux de données. Le temps d'exécution d'un tel schéma est alors celui de la fonction qui prend le plus de temps à s'exécuter.

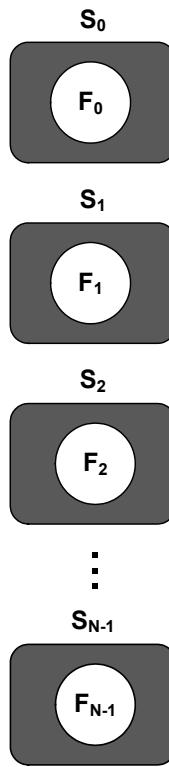


FIG. 2.9 – Exemple de squelette du type Pardo

2.6.2 Squelettes dédiés à la structuration de l'application

Le Squelette Sequence

Le Squelette Select

2.6.3 Modèle de Programmation SKELL_BE

Dans le modèle de programmation SKELL_BE [17], le processeur Cell est considéré comme une machine asymétrique. Le modèle inclue deux codes source, un pour le PPE et un autre pour le SPE. Du point de vue du PPE une application peut appeler un *kernel* de calcul qui est compilé et exécuté sur le SPE comme dans une application de *stream processing* (2.6).

```

1 #include <skell.hpp>
2 SKELL_KERNEL(sample,(2,(float const*,float*)));
3
4 int main(int argc, char** argv)
  
```

```
5 {  
6     float in[256], out[256];  
7  
8     skell::environment(argc, argv);  
9     sample(in, out);  
10  
11    return 0;  
12 }
```

Listing 2.6 – Exemple d'un appel à un kernel PPE dans SKELL_BE

Un *kernel* est défini par la macro SKELL_KERNEL comme un prototype de fonction dans lequel les arguments passés par référence sont considérés comme des sorties du *kernel*, alors que les arguments passées par valeur ou par référence contante sont les entrées du *kernel*. La ligne 9 du listing 2.6 montre un exemple d'appel de *kernel* au runtime, l'initialisation du processeur Cell qui est effectuée par un appel à `skell :environment` démarre un groupe de threads SPE et les mets en attente d'un appel à un *kernel* ce qui réduit le surcoût de création de threads à chaque fois. La terminaison des threads est effectuée de la même manière à la fin de la portée de la fonction *main*.

Du point de vue des SPEs, chacun d'eux est un nœud de *cluster* qui supporte la communication point-à-point. Les applications sont conçus à base d'une composition de squelettes instanciées avec des fonctions définies par l'utilisateur qui considèrent la mémoire centrale comme un mémoire distante à partir de laquelle peuvent être lues ou écrite de manière asynchrone au travers des commandes DMA de la librairie standard ou à des fonctions fournies par SKELL_BE 2.7.

```
1 #include <skell.hpp>  
2  
3 void sqr()  
4 {  
5     float in[32], out[32];  
6  
7     pull(arg0_, in);  
8     for(int i=0;i<32;++i) out[i] = in[i]*in[i];  
9     push(arg1_, out);  
10  
11    terminate();  
12 }  
13  
14 SKELL_KERNEL(sample,(2,(float const*,float*)))
```

```

15 {
16   run( pardo<8>( seq(sqr) ) ;
17 }
```

Listing 2.7 – Exemple de définition d'un kernel SPE dans SKELL_BE

Ce dernier code source illustre plusieurs aspects :

- la macro SKELL_BE génère le *stub* de la fonction `main` et l'introspection de code requise par SKELL_BE.
- la fonction `run` qui est utilisée dans la fonction de *kernel* pour construire une application en utilisant les constructeurs de squelettes `pardo` et `seq`.
- l'objet `argN_` qui fournit un accès transparent au *N^{ième}* argument du *kernel* stocké dans la mémoire centrale.
- les fonctions `pull` et `push` qui permettent un accès asynchrone à la mémoire principale adressée par le PPE. Ces fonctions déduisent la meilleure manière de rapatrier les données à partir de leurs arguments et utilisent un découpage statique des données à partir du nombre de SPEs impliquées et de la taille de leur mémoire privée.
- La fonction `terminate` déclenche la terminaison du *kernel*. Celle-ci n'est appelée qu'une fois par *kernel*, dès que toutes les données transférées de la mémoire centrale ont été traitées.

Le tableau 2.1 résume les principales fonctions de l'interface utilisateur (API) de SKELL_BE.

2.6.4 Détails de l'Implémentation

Le développement d'une librairie de calcul parallèle à base squelettes algorithmiques à la fois efficace et expressive est une tâche complexe. Plusieurs tentatives ont montré que le compromis entre expressivité et efficacité était déterminant pour le succès d'une telle librairie. Le polymorphisme est à première vue une bonne solution pour exprimer la relation entre les squelettes et les objets fonctions, l'expérience démontre que l'*overhead* induit par son runtime affecte considérablement la performance globale d'une application. Dans le cas des squelettes le polymorphisme au *runtime* n'est pas vraiment nécessaire : par conception, la structure d'une application exprimée sous forme de squelettes imbriqués est connue à la compilation. Il suffit juste de trouver une manière propre d'exploiter cette information disponible à la compilation d'une manière judicieuse.

Considérons les constructeurs de squelettes comme des mots-clés d'un petit langage

Gestion de l'Application

<code>environement(argc,argv)</code>	Démarrage de l'application
<code>rank()</code>	retourne l'indentifiant PID su SPE courant
<code>terminate()</code>	Signale la fin du flux de données et termine l'application
<code>run(skeleton)</code>	Execute une application squelette

Constructeurs de Squelettes

<code>seq(f)</code>	Transforme une fonction utilisateur en une tâche squelette
<code>operator,(s₁,s₂)</code>	
<code>chain(s₁,...,s_n)</code>	Constructeur de composition séquentielle
<code>chain<N>(s)</code>	
<code>operator (s₁,s₂)</code>	
<code>pipeline(s₁,...,s_n)</code>	Constructeur de <i>Pipeline</i>
<code>pipeline<N>(s)</code>	
<code>operator&(s₁,s₂)</code>	
<code>pardo(s₁,...,s_n)</code>	Constructeur de <i>Pardo</i>
<code>pardo<N>(s)</code>	

Transfert de Données

<code>pull(arg_N,v,sz=0,o=0)</code>	Récupère <i>sz</i> éléments de la <i>N^{ième}</i> donnée de la mémoire centrale et les sauvegarde dans <i>v</i> avec un <i>offset o</i>
<code>push(arg_N,v,sz=0,o=0))</code>	Envoie <i>sz</i> éléments de la donnée <i>v</i> à la <i>N^{ième}</i> donnée dans la mémoire centrale avec un <i>offset o</i>

TAB. 2.1 – Interface utilisateur SKELL_BE

déclaratif *domain-specific*³. L'information sur l'application à générer est donnée par la sémantique opérationnelle de ces constructeurs. Dans notre cas, le défi était de trouver une manière de définir un tel langage comme une extension de C++ qui définit un EDSL (*Embedded Domain Specific Language*), sans construire une nouvelle variation d'un compilateur mais seulement en utilisant la méta-programmation.

La méta-programmation est un ensemble de techniques héritées de la programmation générative qui permet la manipulation, la génération et l'introspection de fragments de code dans un langage. A titre comparatif, lorsqu'une fonction est exécutée au runtime pour produire des valeurs à l'exécution, une méta-fonction opère à la compilation sur des fragments de code pour générer des fragments de code plus spécialisés qui seront compilés. l'exécution d'un tel code se fait par conséquent en deux passes. En C++, un tel système est mis en oeuvre par les classes et les fonctions *template*. En utilisant la flexibilité de la surcharge d'opérateur et de fonctions en C++ et le fait que les *templates* C++ peuvent effectuer des calculs arbitraires à la compilation, on peut évaluer la structure d'une application parallèle décrite par une combinaison de squelettes **à la compilation**. Pour ce faire, la structure extraite de la définition de l'application doit être transformée en une représentation intermédiaire basée sur un réseau de processus séquentiels. Dans le cas de SKELL_BE la difficulté fut d'enfouir les constructeur de squelettes dans des éléments de langage, de générer le code sur les SPEs et d'effectuer le transfert d'arguments entre le PPE et les SPEs.

2.6.5 Génération de Code pour les SPEs

L'implémentation d'un *EDSL* en C++ impose d'avoir une méthode pour trouver de manière adéquate des informations non-triviales à partir de l'arbre de syntaxe abstraite d'une expression (AST). Ceci est effectué en général à l'aide d'une technique connue sous le nom de *Expression Templates* [18]. Les *Expression Templates* utilisent la surcharge de fonctions et d'opérateurs pour construire une représentation simplifiée de l'arbre de syntaxe abstraite d'une expression. La structure arbre est un type template complexe structuré comme une représentation linéaire de l'arbre. Les information sur les terminaux de l'expression sont sauvegardées en tant que références dans l'objet AST. Cet objet temporaire peut alors être passé comme un argument à d'autres fonctions qui analysent son type est extraient les informations requises pour la tâche en effectuant ce que l'on

³en opposition à un langage *general-purpose*

appelle une **évaluation partielle** [19].

Pour transformer un *AST* en un code exploitable, il faut transformer l’arbre en un réseau de *process*. Afin d’y parvenir, la sémantique opérationnelle définie dans [6] est transformé en méta-programme capable de générer une liste statique de process. Chacune des constructeurs de squelettes de SKELL_BE génère un objet sans état dont le type encode la structure du squelette. A titre d’exemple, le code de l’opérateur *pipe* est donné dans le listing 2.8. On notera qu’aucun calcul n’est effectué à cette étape mais que la structure du squelette est elle même enfouie dans le type de retour.

```

1 template<class LS, class RS>
2 expr<pipe, args<LS,RS> >
3 operator|( LS const&, RS const& )
4 {
5     return expr<pipe, args<LS,RS> >();
6 }
```

Listing 2.8 – L’opérateur *pipe*

Le type *template* est désormais utilisable avec nos méta-fonctions. Celles-ci se chargent de la génération de structures représentants un réseau de *process*. La fonction *run* appelle une méta-fonction qui parse le *template* *AST* et génère l’instanciation du type *process_network* adéquat. La règle de sémantique appropriée est appliquée sur chaque squelette rencontré dans l’*AST* en utilisant la spécialisation partielle des *templates* comme mécanisme de *pattern matching*. Une fois défini, ce réseau est transformé en code en itérant sur ses noeuds et en générant une séquence de fragments de codes SPMD dans lesquelles la liste d’instructions du *process* est exécutée. Ceci est réalisé en construisant un tuple d’objets fonction qui contient le code d’opérations de base qui sont instantiées une fois par SPE. Par exemple, considérons l’expression squelette suivante qui construit un *pipeline* simple à trois étages :

```
run( seq(A) | seq(B) | seq(C) );
```

Cette expression produit au squelette d’*AST* suivant :

```

1 expr< pipe
2   , args<expr< seq, args<function<&A> > >
3   ,expr< pipe
4     , args<expr<seq, args<function<&B> > > >
5     , args<expr<seq, args<function<&C> > > >
6   >
```

```

7      >
8      >

```

Listing 2.9 – La structure d'un squelette connue à la compilation

La structure de cet objet temporaire est désormais claire. Les appels successifs à l'opérateur *pipeline* sont clairement visibles et les objets fonctions terminaux apparaissent explicitement. Pour des raisons de performance, on utilise le fait que l'adresse d'une fonction est une constante valide connue à la compilation que l'on peut stocker directement comme paramètre *template*. Le type est ainsi converti en une représentation sous forme de réseau de *process*. Le résultat est le type suivant :

```

1 network< int_<0>, int_<2>
2   , list< process< int_<0>
3     , desc< pid<-1>, pid<1>
4       , instrs<Call<&A>,Send>
5     >
6   >
7   , process< int_<1>
8     , desc< pid<0>, pid<2>
9       , instrs<Recv, Call<&B>,Send>
10    >
11   >
12   , process< int_<2>
13     , desc< pid<1>, pid<-1>
14       , instrs<Recv, Call<&C> >
15     >
16   >
17 >
18 >

```

Listing 2.10 – Représentation sous forme de réseau *process*

Le type *template* *network* contient toutes les informations qui décrivent le réseau de *process* série communicants construit à partir des squelettes, notamment : le PID du premier noeud du réseau, e PID du dernier noeud du réseau et une liste de *process*. Dans le même esprit, la structure *template* *process* contient des informations dont son propre PID et un descripteur de code. Ce descripteur contient les PID des *process* prédécesseur et successeur et ainsi qu'une liste de macro-instructions qui sont construites à partir de la sémantique du squelette.

La dernière étape est l’itération sur ces types et l’instantiation du code SPMD adéquat. Le listing 2.11 illustre le code final ainsi généré.

```

1 if(rank() == 0)
2 {
3     result_of<A>::type out;
4     do
5     {
6         call<A>(out); DMA_send(out,1);
7     } while( status() );
8 }
9
10 if(rank() == 1)
11 {
12     parameters<B>::type in;
13     result_of<B>::type out;
14     do
15     {
16         DMA_recv(in,0); call<B>(in,out); DMA_send(out,1);
17     } while( status() );
18 }
19
20 if(rank() == 2)
21 {
22     parameters<C>::type in;
23     do
24     {
25         DMA_recv(in,1); call<C>(in);
26     } while( status() );
27 }
```

Listing 2.11 – Code source généré

La structure SPMD des instructions `if` chaînées montrent la nature itérative du générateur de code. Chacun des ces blocs effectue les même opérations. En premier lieu, les types de entrées/sorties sont récupérées de l’analyse du type de fonction. Ces types sont ensuite instanciés sous forme d’un tuple. Le code du *process* est ensuite exécuté dans une boucle qui se met dans une boucle en attente d’un signal de terminaison. Dans cette boucle, chacune des macro-instructions qui apparaissent dans la description du type réseau de *process* génère un appel concret à soit un transfert DMA ou à un proxy d’appels de fonctions qui extraie les données d’un tuple, alimente une fonction définie par l’uti-

lisateur et retourne un tuple de résultats. Une grande partie de ce processus est facilité par les librairies Boost telles que Proto qui gère la génération des règles de sémantique méta-programmées et Fusion qui gère la transition entre les comportement à la compilation et au runtime [5].

Le processus de génération permet de mieux comprendre pourquoi SKELL_BE est plus performant que d'autre solution à base de C++. Dans le code généré, toutes les fonctions et le code dépendant des squelettes sont résolues statiquement. Tous les types de données sont concrets et tous les appels de fonctions sont directs. Il n'y a pas donc pas de polymorphisme au runtime et le compilateur est capable d'*inliner* plus de code et d'effectuer plus d'optimisation.

2.6.6 Communications PPE/SPEs

L'autre difficulté dans la conception d'une librairie de parallélisation de code pour le Cell, est son architecture mémoire distribuée qui requiert une gestion explicite des transfert de données entre la mémoire centrale et les mémoire locale des SPEs. Le but étant de trouver une manière de transférer les données de la mémoire centrale vers les SPEs d'une manière transparente du point de vue de l'utilisateur. Une stratégie usuelle passe par le transfert au début du programme, d'une structure appelée *control block* qui contient les informations communes au SPEs et nécessaires à l'exécution du *kernel*. En général, cette structure dépend de l'application et contient toutes les données dont le *kernel* a besoin. Dans notre cas, ces données sont fournies comme arguments de l'appel de fonction du *kernel* principal. Il est ensuite nécessaire de construire à la compilation la structure *control block* appropriée. Ceci est rendu possible en utilisant la métaprogrammation *template* qui parse le prototype de la fonction pour extraire une liste des types de ces arguments. Le *control block* contiendra ainsi l'adresse de base de l'espace mémoire de chaque SPE et un tuple en utilisant sur l'algorithme suivant.

- Les entrées de types natifs sont stockés par valeurs.
- Les sorties de types natifs sont stockés dans une paire contenant leur adresse et une valeur statique contenant leur taille.
- Les tableaux sont stockés dans une paire contenant l'adresse de leurs éléments et une valeur statique contenant leur taille.
- Les types définis par l'utilisateur sont stockés dans une paire contenant l'adresse de l'objet et sa taille.

Cette structure est ensuite remplie avec les vraies valeurs des données passées en arguments de l'appel du *kernel* avant de lancer les threads SPE. A titre d'exemple le prototype de fonction suivant :

```
void f( int, int[5], float& );
```

est transformé en une structure :

```
1 struct f_args
2 {
3     int                 arg0;
4     pair<int, int_<5> >    arg1;
5     pair<float, int_<4> >   arg2;
6 };
```

Listing 2.12 – Code source généré

La fonction suivante est ensuite générée pour la remplir :

```
1 void f_args_fill( f_args& a, int v0, int v1[5], float& v2 )
2 {
3     a.arg0 = v0;
4     a.arg1.first = &v1[0];
5     a.arg2.first = &v2;
6 };
```

Listing 2.13 – Code source généré

Du côté du SPE, les objets `argN` fournissent un opérateur de cast *template* implicite qui récupère les valeurs du $N^{ième}$ élément du tuple au travers d'une commande DMA, ainsi qu'un opérateur d'affectation qui transfert une valeur à la données correspondante dans l'espace d'adressage du PPE. La déduction automatique des arguments *template* permet une syntaxe à la fois compacte et intuitive de telle sorte à ce que le compilateur puisse appeler la bonne primitive de transfert DMA en se basant sur l'index des arguments et le type de la valeur.

2.6.7 Résultats Expérimentaux

Nous avons mené une campagne de mesure qui a pour but de prouver que SKELL BE ne provoque pas de pertes importante au niveau de la performance. Pour ce faire, nous avons effectué plusieurs tests. La première vise à créer des applications synthétiques

à bas de squelettes afin d'évaluer l'*overhead* des méta-programmes. Le deuxième test consiste en l'évaluation de la performance d'algorithmes numériques et de l'algorithme Harris pour la detection de point d'intérêt traité plus en détail par la suite. Les tests ont été effectués sur une lame IBM QS20 et la compilation s'est faite à l'aide avec la chaîne gcc et la métrique temporelle utilisée est le nombre de cycles moyen par point traité.

Benchmarks Synthétiques

Cette mesure a pour but de prouver que le surcoût induit par la couche de métaprogrammation d'un squelette est négligeable. Pour ce faire, nous avons évalué le temps d'exécution d'un squelette SKELL BE synthétique en augmentant le nombre de SPEs mis en jeu avec un code source similaire écrit à la main. Les premiers résultats permettent de constater que l'exécution d'une fonction au travers des opérateurs CHAIN ou SEQ, n'induit pas un *overhead* important. L'examen du code source assembleur généré démontre que la seule différence entre l'appel direct et la version squelette est une indirection de pointeur qui permet de retrouver l'adresse de la fonction à partir de l'objet adaptateur de fonction utilisé en interne.

Le test pour le squelette PIPE construit un *pipeline* de 2 jusqu'à 8 SPEs dans lequel la quantité de transfert de données est négligeable. Chaque étage de ces *pipeline* exécutent donc la durée est comprise entre 10 ms et 200 ms. Les mêmes tests ont été effectués pour le squelette PARDO. Les figures ?? et ?? illustrent les résultats de mesures. On peut ainsi constater que le surcoût ne dépasse jamais 1.5 %.

Benchmarks de Scalability

L'évaluation qui suit mesure la *scalability* (passage à l'échelle) de notre outil SKELL BE et fait une comparaison avec d'une part un code équivalent écrit à la main ou alors un code *OpenMP* compilé avec *XLC single source compiler* pour le Cell. La mesure s'est faite sur des noyaux de calculs de l'API *BLAS Basic Linear Algebra Subprograms*. L'évaluation de la *scalability* se fait en mesurant un *speedup* relatif en comparaison avec l'exécution sur un SPE.

Le noyau DOT Dans ce programme nous effectuons le produit scalaire de deux tableaux de 10^9 éléments flottants simple-précision. La version OpenMP utilise une directive de réduction alors que les versions écrites à la main et SKELL BE collectent explicitement

SPE	OMP	Manual	SKELL BE	<i>overhead</i>
1	219.7	65.9	67.9	3.1 %
2	263.7	32.9	34.5	4.7 %
4	131.9	16.5	17.3	4.78 %
8	66.1	8.3	8.7	4.9 %

TAB. 2.2 – Benchmark DOT

SPE	OMP	Manual	SKELL BE	<i>overhead</i>
1	2402	649	672	3.6%
2	4289	391	411	5.0 %
4	2146	172	181	45.2 %
8	1073	98	103	5.4%

TAB. 2.3 – Benchmark CONVO

les résultats partiels pour les additionner. Dans ce cas là le *cpp* minimum pour la version OpenMP est de 66 donnant un *speedup* maximal relatif de $\times 3.32$ lorsque la version manuelle donne elle, un *speedup* de 7.98. La version OpenMP est limitée par le surcoût induit par la gestion implicite des communication et de la synchronisation. Dans la même situation SKELL BE fournit un accélération maximale de $\times 7.85$ ce qui représente un *overhead* de 5% par rapport à la version écrite à la main.

Le noyau CONVO Dans ce deuxième opérateur testé, nous effectuons un produit de convolution sur des images de taille 4096×4096 avec un masque de taille 3×3 . Dans toutes les versions le masque est dupliqué dans chaque SPE. Les versions manuelles atteignent jusqu'à $\times 6.54$ comparativement au *speedup* OpenMP de 2.24 et celui de SKELL BE mesuré à 6.52. L'*overhead* quand à lui a augmenté à cause notamment de la gestion des transfert non-alignés (qui se fait au runtime) du noyaux de convolution, il est autour de 5%.

Le noyau SGEMV Dans ce benchmark, un produit entre une matrice 4096×4096 et un vecteur 4096×1 . On remarque la même chose que dans ce qui précède, c'est à dire une bonne *scalability* de SKELL BE avec un *overhead* toujours autour de 5%.

SPE	OMP	Manual	SKELL BE	overhead
1	200.7	179.8	187.9	4.6%
2	208.5	79.9	83.9	4.9 %
4	104.3	42.2	44.5	5.5 %
8	52.2	23.6	25.0	5.9%

TAB. 2.4 – Benchmark SGEMV

Algorithme de Harris

Cette application est plus complexe que celle étudiées précédemment car elle comporte plusieurs opérateurs, à la fois point-à-point et noyaux de convolution. Ce benchmark a pour but de prouver que SKELL BE est également adapté pour notre domaine d'application. Plusieurs schémas de parallélisation sont possible pour cet algorithmes, ils sont traités avec plus de détails dans le chapitre suivant. Nous avons choisi ici trois versions de déploiement : une version complètement chaînée où tous les opérateurs sont regroupés au sein d'un seul et même SPE et répliqués 8 fois ; la versions chaînée à moitié, ou les opérateurs sont chaînés deux à deux sur un SPE, un *pipeline* est ensuite formé entre deux SPEs ce qui permet de répliquer 4 fois ; Et enfin une version où chaque opérateur occupe un SPE, ce qui permet de répliquer 2 fois. Le listing 2.14 représente les *kernels* SPE des différentes versions. Le chaînage y est représenté par une virgule (,) et le *pipeline* par un opérateur |.

```

1 void full_chain_harris(tile const&, tile &)
2 {
3     run( pardo<8>((seq(grad), seq(mul)
4                     , seq(gauss), seq(coarsity)));
5 }
6
7 void half_chain_harris(tile const&, tile &)
8 {
9     run( pardo<4>( (seq(grad), seq(mul))
10          | (seq(gauss), seq(coarsity)));
11 }
12
13 void no_chain_harris(tile const&, tile &)
14 {
15     run( pardo<2>( seq(grad) | seq(mul)
16                  | seq(gauss) | seq(coarsity));

```

Listing 2.14 – Les kernels SPE des implémentations de l’opérateur de Harris

On compare les différentes versions en terme de nombre de cycles par pixel avec des implémentations manuelles de l’algorithme sur des images 512×512 (Tableau 2.5). L’overhead est également de 5% ce qui valide notre approche car le compromis overhead rapidité de mise en oeuvre est très bon.

Manual	Full-Chain	Half-Chain	No-Chainl
Manual	11.26	8.36	9.97
SKELL BE	11.86	8.64	10.43
Overhead	5.33 %	3.35 %	4.61 %

TAB. 2.5 – Benchmarks de l’algorithme de Harris

Impact sur la Taille de l’Exécutable

La méta-programmation est souvent blâmée pa-ce-qu’elle produit des exécutables trop grands à cause de la réPLICATION de code. Sur le processeur Cell, ce problème devient critique à cause de la taille limitée des *local store* (256 KB) et doit par conséquent rester sous contrôle. Pour évaluer l’impact de la méta-programmation sur la taille du code source nous avons comparé la taille des codes générés par SKELL BE avec ceux écrits à la main. De manière générale, le code SKELL BE tiens largement dans les 256 KB du *local*

Opérateur	DOT	CONVO	SGEMV	HARRIS
Code écrit à la main	1.1 KB	1.2 KB	2.3 KB	5.3 KB
Code Généré par SKELL BE	12.6 KB	14.5 KB	22.7 KB	49.4 KB

TAB. 2.6 – Benchmarks de l’algorithme de Harris

store mais celui-ci est 10 fois plus grand qu’un code équivalent écrit à la main. Ceci est principalement du au fait que SKELL BE génère par défaut un code SPMD contenant une structure du type *switch* qui englobe le code de tous les SPEs, ce qui donne approximativement un code 8 fois plus gros. Une des solutions dans ce cas là serait de passer le PID du SPE comme symbole au pré-processeur et de ne faire compiler que le code propre à ce même SPE.

Impact sur la Temps de Compilation

L'autre problème s'agissant de la méta-programmation est le temps de compilation. En effet, le temps de compilation d'un programme SKELL BE peut être décomposé en deux étapes principales : une première étape durant 1.5 s qui englobe les directives du préprocesseur qui gère les fonctions définies par l'utilisateur ; une deuxième étape proportionnelle au nombre de types de squelettes utilisés. Dans le pire des cas, qui est celui du squelette Half-Pipe la compilation prend 10s.

2.6.8 Conclusion

SKELL BE est une solution à base de langage spécifique au domaine enfouis dans C++. Il est basé sur les squelettes algorithmiques ,un modèle de programmation parallèle très flexible. Il est très adapté au processeur Cell car le placement du graphe d'application y est primordial pour la performance. Les résultats qui précèdent tendent à prouver que l'on obtient de bonne performance tant au niveau temporel brut qu'au niveau de la *scalability*. Les mesures sur les benchmarks simples ont prouvé que l'*overhead* induit est négligeable comparativement à un code écrit à la main. Le déploiement de l'algorithme de Harris sur le Cell à l'aide de SKELL BE et selon plusieurs schémas de parallélisation, permettent d'apprécier la flexibilité et l'expressivité fournies par l'outil. Les performances obtenus sur l'algorithme prouvent l'efficacité de l'outil.

2.7 Conclusion Générale

Dans ce qui précède nous avons décrit les différents outils ou modèle de programmation pour le processeur Cell. Ce chapitre, qui n'est pas une étude exhaustive, fait état des principaux outils qui ont fait objet de publications et d'évaluation significatives. Les outils diffèrent par leur mise en oeuvre et leur difficulté d'utilisation. Les plus simples à utiliser, qui sont les approches à base d'annotation de code (OpenMP, CellSS), laissent le soin au compilateur de faire le travail de parallélisation et d'optimisation du code. De l'autre côté RapidMind et Sequoia adoptent une approche qui définit un langage spécifique accompagné d'un runtime et d'un compilateur, mais une bonne partie du travail d'optimisation est laissée à la charge du programmeur. Enfin, l'approche de programmation la plus fastidieuse et celle des *Pthreads* qui sont un outil de programmation parallèle

très bas niveaux qui laisse une grande liberté au programmeur pour ce qui est du déploiement de son code sur le Cell. Un code à base de threads est de ce fait long à mettre en oeuvre et difficile à maintenir, mais il permet d'un autre côté d'avoir un contrôle total sur son implémentation. Ceci a un intérêt en cas de contraintes fortes en termes de temps d'exécution et de prédictibilité de comportement. On notera que la mise en oeuvre nécessite un grand effort de la part des concepteurs pour deux raisons principales : (1) la mémoire distribuée du Cell qui impose la gestion explicite des transferts mémoire à partir de et vers la mémoire centrale (2) Le PPE et les SPEs possèdent un jeux d'instructions différents ce qui rend difficile l'étape de génération de code.

Parallélisation de Code de Traitement d'Images

Dans ce chapitre nous rentrons dans le vif du sujet, à savoir, la parallélisation de code de traitement d'images pour le processeur Cell. L'algorithme considéré est celui de la détection de points d'intérêts de Harris. Le choix de cet algorithme s'est fait selon plusieurs critères qui sont les suivants :

- C'est un algorithme de traitement d'images bas niveaux qu'on retrouve dans plusieurs applications plus complexes, comme la reconstructions 3D et le suivi d'objets.
- Il est composé de blocs de traitement de base qui sont représentatifs des algorithmes bas niveau comme les opérateurs de convolution et les opérateurs point à point.
- C'est un algorithme qui ne peut pas s'exécuter en temps réel sans optimisations spécifiques.

Étant donné les caractéristiques de l'architecture du Cell ainsi que celles de l'algorithme, le but est de trouver la meilleure implémentation permettant d'exploiter au mieux les dispositifs haute performance de l'architecture. Le processeur Cell est un vrai concentré de dispositifs accélérateurs parmi lesquels les unités SPE purement SIMD, les contrôleurs DMA permettant un parallélisme entre transferts mémoire et tâches de calculs ainsi que la multiplicité des coeurs qui permettent de répartir la charge de calcul de plusieurs manières possible, soit sous forme de parallélisme de donnée uniquement, ou alors de parallélisme de tâches ou un mélange des deux.

3.1 Algorithme de Harris

La détection de point d'intérêts de *Harris* et *Stephen* [10] est utilisée dans les systèmes de vision par ordinateur pour l'extraction de connaissance comme la détection de mouvement, la mise en correspondance d'images, le suivi d'objets, la reconstruction 3D et la reconnaissance d'objets. Cet algorithme fut proposé pour palier aux manques de l'algorithme de *Moravec* [15] qui était sensible au bruit et pas invariant à la rotation. Un coin peut être défini comme étant l'intersection de deux contours lorsque un point d'intérêt peut être défini comme un point ayant une position bien déterminée et qui peut être détecté de manière robuste. Ainsi, le point d'intérêt peut être un coin mais aussi un point isolé d'intensité maximum ou minimum localement, une terminaison de ligne ou encore un point de courbe où la courbure est localement maximale.

3.1.1 Description de l'Algorithme

Si l'on considère des zones de l'image de dimensions $u \times v$ (dans notre cas 3×3 dans une images 2-dimensions en niveaux de gris I qui est décalée de (x, y) , l'opérateur de Harris est basé sur l'estimation de l'auto-corrélation locale S dont l'équation est la suivante :

$$S(x, y) = \sum_u \sum_v w(u, v) (I(u, v) - I(u - x, v - y))^2 \quad (3.1)$$

Par l'approximation de S avec une série de Taylor du second ordre la matrice de Harris M est donnée par :

$$M = \sum_u \sum_v w(u, v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \quad (3.2)$$

Un point d'intérêt est caractérisé par une large variation de S dans toutes les directions du vecteur (x, y) . En analysant les valeurs propres de M , cette caractérisation peut être exprimée de la manière suivante. Soit λ_1, λ_2 les valeurs propres de M :

1. Si $\lambda_1 \approx 0$ et $\lambda_2 \approx 0$ alors il n'y a pas de point d'intérêt au pixel (x, y) .
2. Si $\lambda_1 \approx 0$ and λ_2 a une grande valeur positive alors un contour est retrouvé.
3. Si λ_1 and λ_2 sont deux grandes valeurs positives distinctes alors un coin est détecté.

Harris et *Stephen* ont constaté que le calcul des valeurs propres était couteux car il requiert le calcul d'une racine carrée, et ont proposé à la place l'algorithme suivant :

- Pour chaque pixel (x, y) de l'image calculer la matrice de corrélation M :

$$M = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}; \text{ où } S_{xx} = \left(\frac{\partial I}{\partial x} \right)^2 \otimes w, S_{yy} = \left(\frac{\partial I}{\partial y} \right)^2 \otimes w, S_{xy} = \left(\frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \right) \otimes w \quad (3.3)$$

Où \otimes est l'opérateur de convolution w un noyau Gaussien.

- Construire la carte de coarsité en calculant la mesure de coarsité $C(x, y)$ pour chaque pixel (x, y) :

$$C(x, y) = \det(M) - k(\text{trace}(M))^2 \quad (3.4)$$

$$\begin{aligned} \det(M) &= S_{xx} \cdot S_{yy} - S_{xy}^2 \\ \text{trace}(M) &= S_{xx} + S_{yy} \end{aligned}$$

et k une constante empirique.

Une illustration d'une détection de points d'intérêt sur une image 512×512 en niveaux de gris est donnée en figure 3.1. Afin d'obtenir ce résultat, deux étapes supplémentaires sont nécessaires qui permettent d'extraire une information visuelle à partir de la matrice $C(x, y)$ ¹. Ces étapes sont les suivantes :

- Seuillage de la carte d'intérêt en mettant toute les valeurs de $C(x, y)$ inférieurs à un seuil donné à zéro.
- Extraction des maxima locaux en gardant les points qui sont plus grand que tous leurs voisins dans un voisinage 3×3 .

3.1.2 Détails de l'Implémentation

Les images en niveaux de gris sont typiquement des données stockées une des entiers 8-bit non signés et la sortie de l'algorithme de Harris est dans ce cas là un entier 32 bit signé. Toutefois, pour des raisons de limitation du jeux d'instruction du SPU, et afin de garantir une comparaison objective avec les extension Altivec et SSE nous avons choisi le

¹Ces étapes ont pour but de visualiser le résultat et ne sont donc pas incluses dans le graphe de l'algorithme

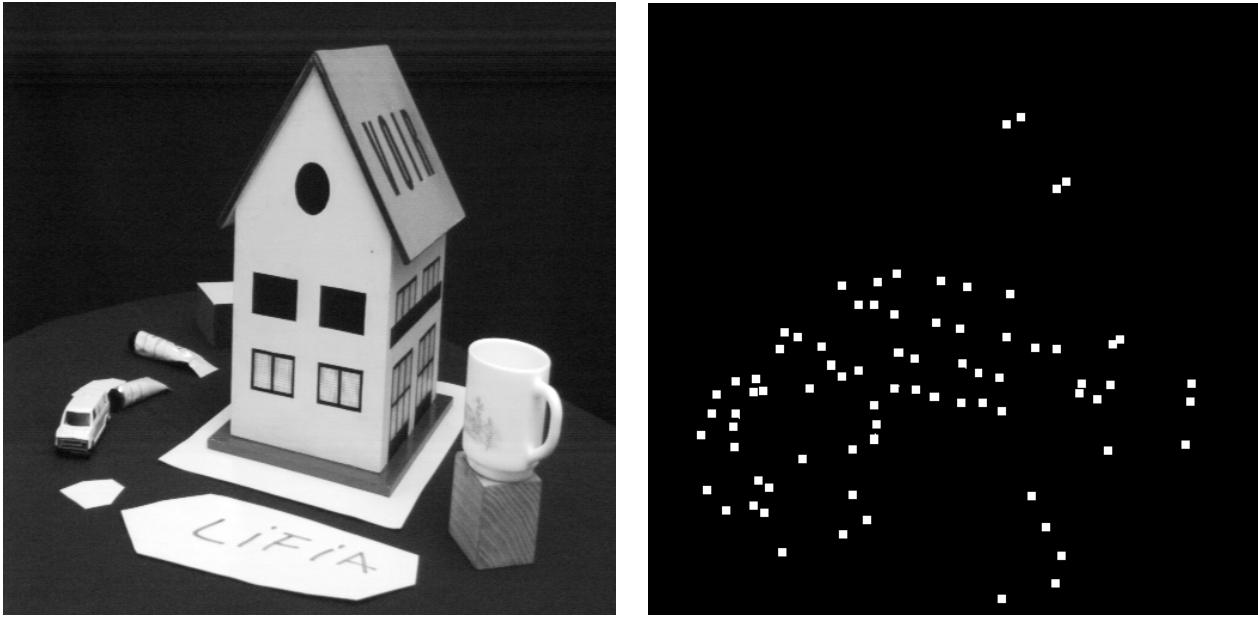


FIG. 3.1 – Illustration de la détection de points d'intérêts sur une image niveaux de gris 512×512

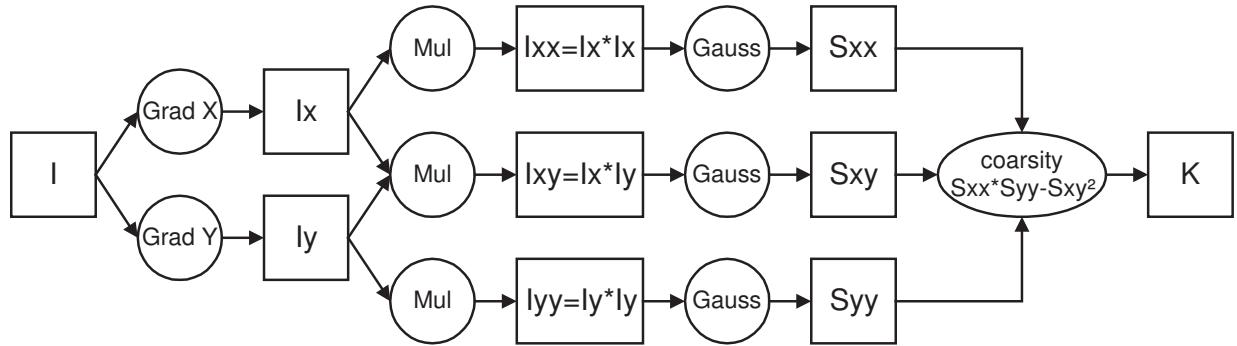


FIG. 3.2 – Implémentation de l'algorithme de Harris sous forme de graphe flot de données

format flottant simple précision pour l'entrée et la sortie de l'algorithme. Dans notre implémentation nous avons divisé l'algorithme en 4 noyaux de traitement : un opérateur de *Sobel* qui représente la dérivée dans les directions horizontale et verticale, un opérateur de multiplication, un noyau de lisage de *Gauss* (w dans l'équation 3.3 suivie d'un opérateur de coarsité. Nous avons fixé la constante k à zéro (typiquement elle est fixée à 0.04) car ceci n'avait pas d'influence sur le résultat qualitatif. On obtient ainsi le graphe flot de données donné dans la figure 3.2 qui est représentatif d'un algorithme de traitement d'images bas niveau car il englobe des noyaux de convolution et des opérateurs point à

point. Les noyaux de convolution de Sobel ($GradX$ et $GradY$) et le noyau de $Gauss$ sont définis comme suit :

$$Grad_X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}; Grad_Y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}; Gauss = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Étant donnée qu'ils consomment plus d'entrées qu'ils ne produisent de sorties, les noyaux de convolution sont le goulot d'étranglement de l'algorithme car elles augmentent considérablement le trafic mémoire. Au vu de la nature des calculs effectués dans les différents noyaux, et qui sont très simples généralement (une suite de multiplications/accumulation) on peut considérer que les instructions mémoire sont prépondérantes dans l'application et que de ce fait, on peut qualifier l'algorithme de *memory-bounded problem* (problème limité par la mémoire). C'est pour cela que les efforts d'optimisation sur l'algorithme de Harris sont faites par l'optimisation des accès mémoire à différent niveaux de la hiérarchie mémoire du processeur Cell.

3.2 Exploitation du Parallélisme et Optimisations Multi-niveau

Les techniques d'optimisation démontrées ici sont multiples et variées. Certaines sont de nature algorithmique et relèvent plutôt du domaine du traitement du signal et des images. D'autres techniques génériques relèvent plutôt du domaine de l'optimisation logicielle est qu'on retrouve parfois dans certains compilateurs optimisants. Les techniques précédentes sont générales et peuvent être appliquées à la majorité des processeurs généralistes car elle ne tiennent pas compte des aspects spécifiques d'une architecture donnée. Par contre, des optimisations de plus haut niveau et qui sont spécifiques à l'architecture particulière du Cell ont aussi été employées. Celles-ci ne sont généralement pas reproducibles sur d'autres architectures parallèles car elles relèvent plus d'une adéquation entre l'algorithme et l'architecture qui contient certains dispositifs qui n'existent que sur le Cell et des fois elles résultent de contraintes de programmation spécifiques au Cell comme la taille limitée des mémoire locale des SPEs et par conséquent la gestion logicielle de l'utilisation de la mémoire.

3.2.1 Techniques Spécifiques au Domaine

Ces optimisations relèvent plutôt du domaine du traitement du signal et des images. Elles peuvent donc être appliquées à plusieurs algorithmes et sur n'importe quelle architecture. Celles que nous avons utilisé concernent les noyaux de convolution et sont : la séparabilité, le chevauchement (*overlapping*) et la factorisation des calculs.

Séparabilité des Noyaux

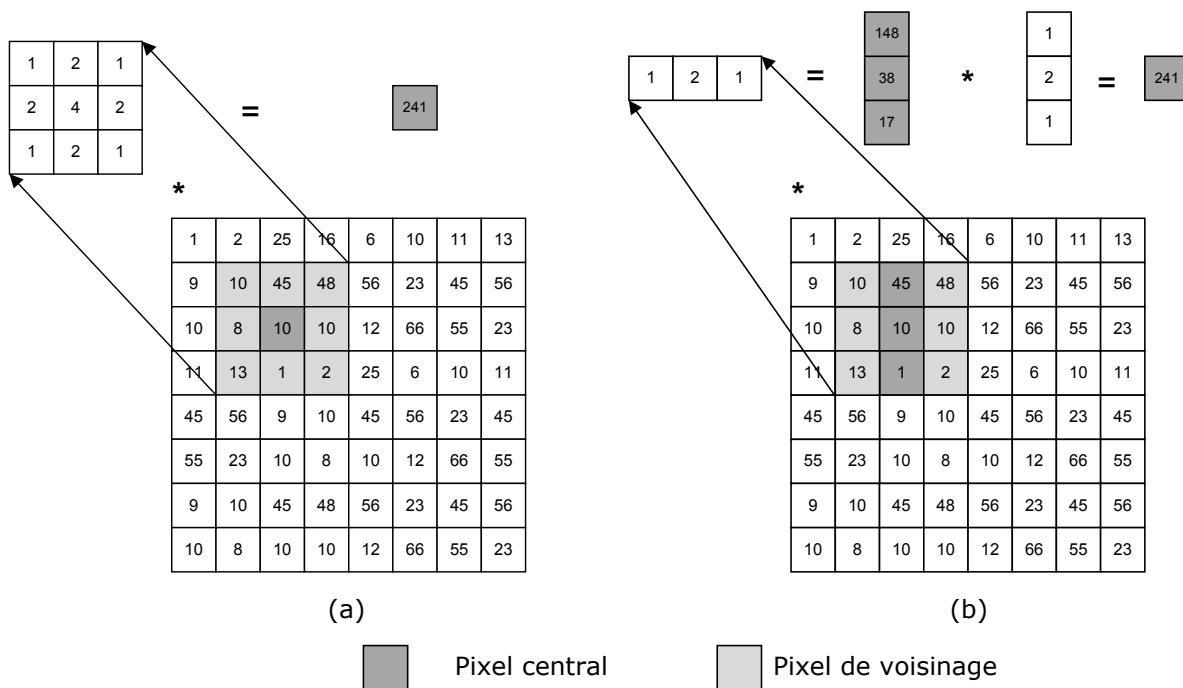


FIG. 3.3 – Exemple de convolution par un filtre Gaussien 3×3 (a)version avec noyaux 2D (b) version avec deux noyaux 1D , résultant de la séparation du noyau 2D.

Cette optimisation consiste à exploiter le fait que les noyaux de convolution 2D de *Sobel Gauss* soient séparables en deux filtre de convolution 1D (Fig. 3.3). Ainsi, la matrice des coefficients peut être exprimée comme un produit de deux vecteur comme l'illustre

Complexité arithmétique filtre de Sobel

Sans séparation du noyau			Avec séparation du noyau			Gain %
MUL	ADD	Total	MUL	ADD	Total	
2	5	7	1	3	4	75

Complexité arithmétique filtre de Gauss

Sans séparation du noyau			Avec séparation du noyau			Gain %
MUL	ADD	Total	MUL	ADD	Total	
8	5	13	2	4	6	116

TAB. 3.1 – Réduction de la complexité arithmétique par séparabilité des noyaux

Complexité mémoire filtre de Sobel

Sans séparation du noyau			Avec séparation du noyau			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
6	1	7	1	5	2	0

Complexité mémoire filtre de Gauss

Sans séparation du noyau			Avec séparation du noyau			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
9	1	10	6	1	8	25

TAB. 3.2 – Réduction de la complexité mémoire par séparabilité des noyaux

les équations ci-dessous :

$$Grad_X = \frac{1}{8} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \frac{1}{8} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}; Grad_Y = \frac{1}{8} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{8} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$Gauss = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{16} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Lorsque l'on sépare les noyaux de convolution, le calcul se fait en deux passes : une pour chaque vecteur. Grâce à la séparabilité des noyaux on arrive à réduire le nombre d'instructions mémoire ainsi que la complexité arithmétique. La comparaison est illustrée dans les tableaux 3.2.1 et 3.2.1.

Chevauchement des Noyaux

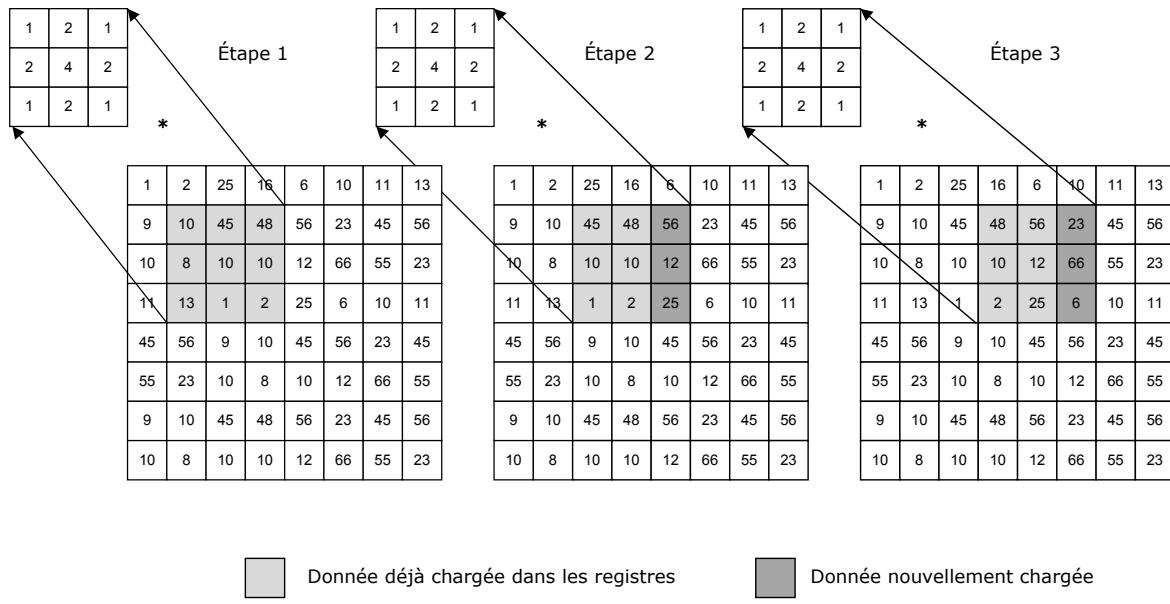


FIG. 3.4 – Chevauchement des données pour un calcul d'un filtre Gaussien 3×3 , certaines données sont conservées en décalant le masque de convolution.

La deuxième particularité des noyaux de convolution est une notion de chevauchement qui permet d'avoir une redondance d'une partie des données (Fig. 3.4). En effet, comme le démontre la figure ??, à chaque itération du calcul de la convolution il n'y a qu'une seule nouvelle colonne chargée. Les colonnes redondantes sont copiées dans les registres en les décalant d'un pas à droite par rapport à leur position précédente (*rotation de registres*). On notera que le même type d'optimisation peut se faire grâce à un *déroulage de boucle*.

Séparation des noyaux et chevauchement

Cette optimisation n'est en fait qu'une combinaison des deux précédentes. En effet on profite d'une part du fait que les noyaux soient séparables pour réduire la complexité arithmétique, et d'autre part du chevauchement des noyaux pour mémoriser le résultat précédent. Ainsi, au lieu de mémoriser les deux dernières colonnes, on mémorise le résultat du filtrage par le premier filtre 1D pour le réutiliser à l'itération suivante de la

Complexité mémoire filtre de Sobel

Sans chevauchement du noyau			Avec chevauchement du noyau			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
6	1	7	2	1	3	

Complexité mémoire filtre de Gauss

Sans chevauchement du noyau			Avec chevauchement du noyau			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
9	1	10	3	1	4	

TAB. 3.3 – Réduction de la complexité mémoire par chevauchement des noyaux

Complexité arithmétique filtre de Sobel

Sans séparation du noyau + chevauchement			Avec séparation du noyau + chevauchement			Gain %
MUL	ADD	Total	MUL	ADD	Total	
2	5	7	1	3	4	

Complexité arithmétique filtre de Gauss

Sans séparation du noyau + chevauchement			Avec séparation du noyau + chevauchement			Gain %
MUL	ADD	Total	MUL	ADD	Total	
8	5	13	2	4	6	

TAB. 3.4 – Réduction de la complexité arithmétique par séparabilité et chevauchement des noyaux

boucle. Dans ce cas là ; la compléxité arithmétique est la même que celle de la version avec séparation des noyaux, alors que la complexité mémoire est réduite d'avantage. Les tableau 3.2.1 et 3.2.1 donnent la différence en terme de complexité arithmétique et mémoire entre la version de base de l'algorithme est la version tenant compte des deux optimisations combinées.

Composition de Fonctions

Cette technique d'optimisation s'avère très efficace surtout dans les codes où les instructions mémoire sont prépondérantes. En effet, le fait de composer deux fonctions de calcul pour en faire une seule qui effectue les deux calculs réduit considérablement le nombre d'accès mémoire puisque les opérations de sauvegarde et de chargement intermédiaires entre les deux fonctions initiales sont supprimées, et la transaction se fait au

Complexité mémoire filtre de Sobel

Sans séparation du noyau + chevauchement			Avec séparation du noyau + chevauchement			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
6	1	7	1	1	2	250

Complexité mémoire filtre de Gauss

Sans séparation du noyau + chevauchement			Avec séparation du noyau + chevauchement			Gain %
LOAD	STORE	Total	LOAD	STORE	Total	
9	1	10	1	1	2	400

TAB. 3.5 – Réduction de la complexité mémoire par séparabilité et chevauchement des noyaux

niveaux des registres. De plus, le cout de l'appel de fonction (taille de la pile, et branchement) est également réduit, car le résultat de la composition de fonctions est une seule fonction. Dans les cas les plus simples, la mise en oeuvre de cette technique n'est pas difficile. Toutefois, lorsqu'il s'agit d'opérateurs de convolution comme dans notre cas, des nouvelles contraintes apparaissent afin de garantir la validité du résultat du calcul. Ainsi, des règles de composition s'imposent en fonction de l'ordre dans lequel s'enchainent les fonctions. Ces règles sont illustrées sur la figure 3.2.1. On peut alors constater qu'il existe plusieurs règles de composition des fonctions selon d'une part, la nature des opérateurs mis en jeu et d'autre part l'ordre dans lequel il s'enchainent. En terme d'apport de cette technique en terme de performances nous observons deux aspects distincts :

- **Complexité mémoire** : on observe en effet que ceux ci sont systématiquement réduits, car les opérations de LOAD/STORE intermédiaires sont supprimés, sauf dans le cas de la composition de deux noyaux de convolution (cas **(d)** sur la figure 3.2.1) ou la présence de bords supplémentaires augmente considérablement le nombre des accès mémoire.
- **Complexité arithmétique** : Celle-ci ne change pas dans le meilleurs des cas notamment lorsque les opérateurs composées sont point à point ou alors lorsque l'opérateur de convolution est placé devant un opérateur point à point (cas **(b)** sur la figure 3.2.1). Toutefois, si la convolution est placée à la suite d'un quelconque traitement, elle impose que l'opérateur qui la précède soit effectué sur tous les pixels de voisinage, ce qui augmente considérablement la complexité arithmétique, en particulier lorsque l'opérateur qui précède est une convolution (cas **(d)** sur la figure

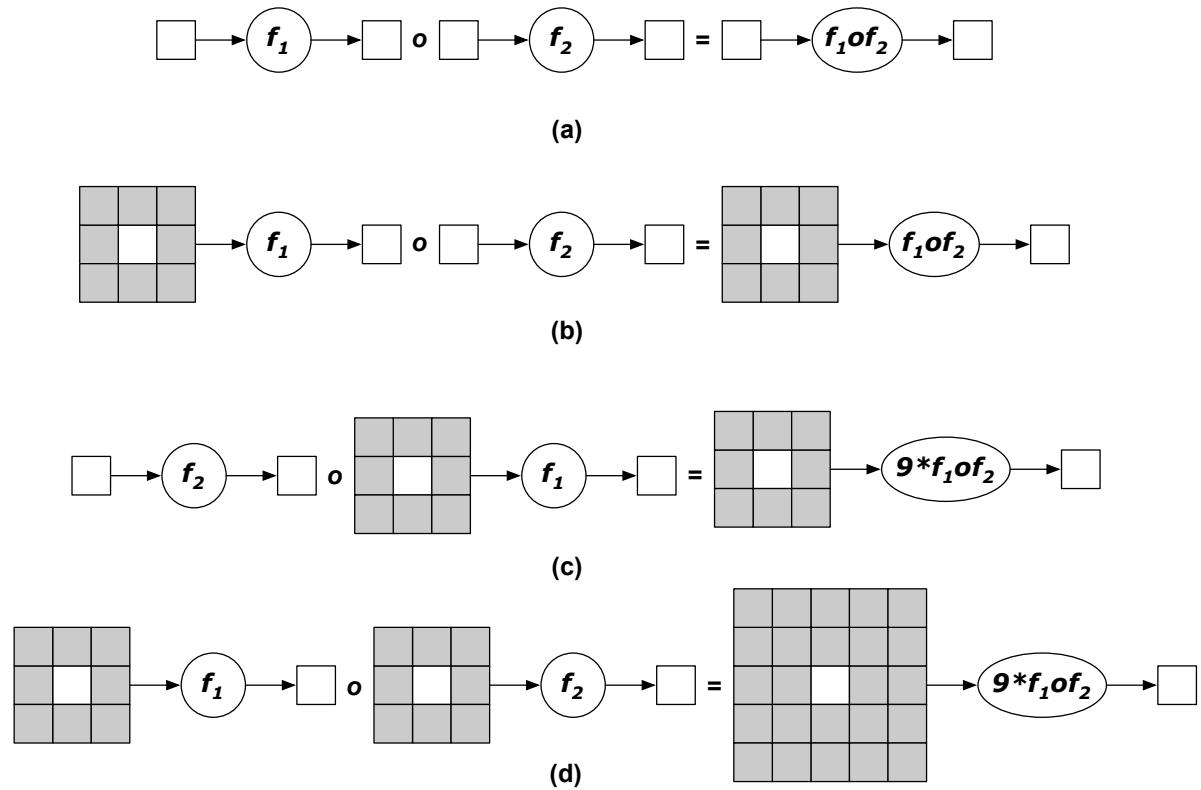


FIG. 3.5 – Règle de composition de fonctions. **(a)** composition de deux opérateurs point à point **(b)** composition d'un noyaux de convolution suivi d'un opérateur point à point **(c)** composition d'un opérateur point à point suivi d'un noyaux de convolution **(d)** composition de deux noyaux de convolution successifs

3.2.1).

D'après les observations ci-dessus, la composition de fonction peut être un bon choix pour l'optimisation d'une chaîne de traitement telle que l'algorithme de détection de point d'intérêts de Harris. Toutefois, toutes les combinaisons n'apportent pas forcément une amélioration des performances, elle peuvent même dans certains cas les dégrader. Le tableau 3.2.1 résume les complexité mémoire et arithmétiques des différents cas de composition sur la figure 3.2.1 en incluant également les optimisations décrites auparavant.

CHAPITRE 3. PARALLÉLISATION DE CODE DE TRAITEMENT D'IMAGES

Version de base sans composition				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel</i>	2	6	1	14
<i>Mul</i>	3	2	1	9
<i>Gauss</i>	3	9	1	30
<i>Coarsity</i>	1	3	1	4
<i>Harris</i>	1	48	9	57
Version avec chevauchement et séparation des noyaux sans composition				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel</i>	1	3	2	5
<i>Mul</i>	3	2	1	6
<i>Gauss</i>	3	3	1	12
<i>Coarsity</i>	1	3	1	4
<i>Harris</i>	1	21	9	30
Version avec composition de Sobel◦Mul et Gauss◦Coarsity				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel◦Mul</i>	1	9	3	12
<i>Gauss◦Coarsity</i>	3	9	1	28
<i>Harris</i>	1	36	4	40
Version avec chevauchement et séparation + composition de Sobel◦Mul et Gauss◦Coarsity				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel◦Mul</i>	1	3	3	6
<i>Gauss◦Coarsity</i>	3	3	1	10
<i>Harris</i>	1	12	4	16
Version avec composition de Sobel◦Mul◦Gauss◦Coarsity				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel◦Mul◦Gauss◦Coarsity</i>	1	25	1	26
<i>Harris</i>	-	-	-	26
Version avec chevauchement et séparation + composition de Sobel◦Mul◦Gauss◦Coarsity				
Opérateur	Occurrences	LOADS	STORE	Total
<i>Sobel◦Mul◦Gauss◦Coarsity</i>	1	5	1	6
<i>Harris</i>	-	-	-	6

TAB. 3.6 – Tableau récapitulatif de l'optimisation des accès mémoire en combinant l'ensemble des optimisations

3.2.2 Optimisation des Transferts Mémoire

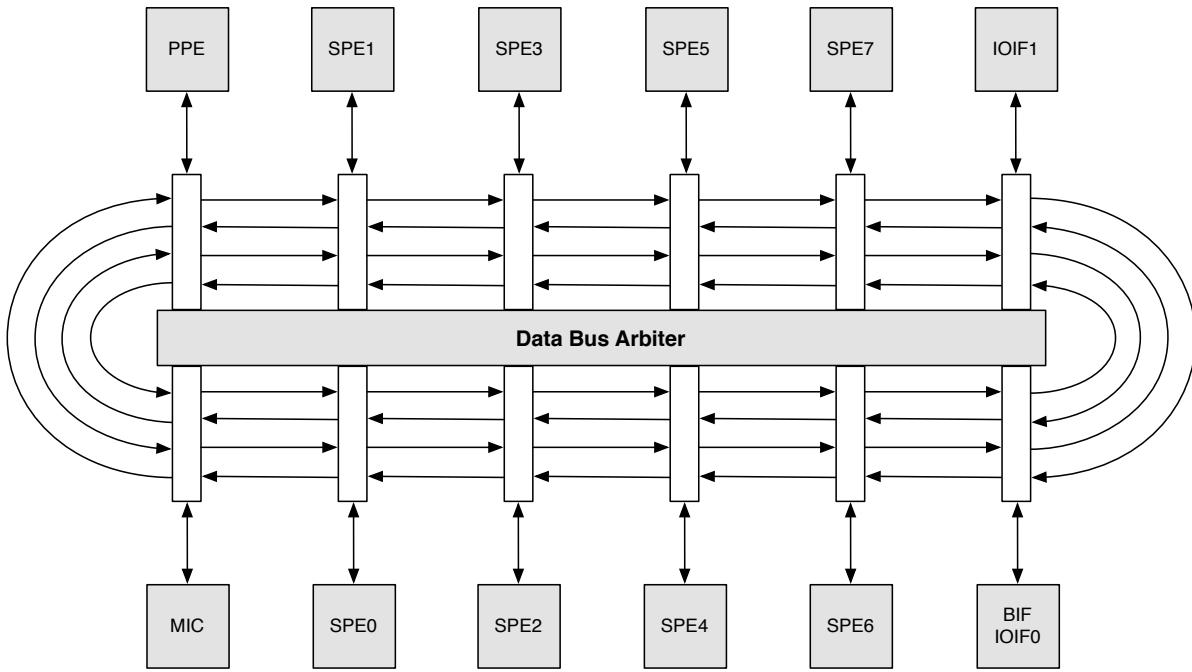


FIG. 3.6 – Réseau d'interconnection du Cell

Dans cette partie, nous abordons des techniques d'optimisations qui sont relatives au transferts de données présents dans l'application. On entend, par transfert de donnée, toute communication qui met en jeu deux mémoires physiques sur Cell. Cela comporte les transferts DMA entre la mémoire principale et les mémoires locales des SPEs ainsi que les communications mettant en jeu deux mémoires privées de SPEs. Plusieurs aspects sont mis en avant. D'une part, les caractéristiques internes de l'architecture du réseaux de communications [13] (*Network On Chip (NoC)*) du Cell et d'autres part la nature des transferts de données imposées par l'algorithme et la partition des données à plusieurs niveaux de la hiérarchie mémoire.

Optimisation de la Bande-Passante du NoC

Dans le modèle de programmation utilisé qui est basé sur le *SDK* du Cell et les librairies *libspe*, les données présentes en mémoire centrale sont transférées aux mémoires locales des SPEs avant d'être traitées. Ces transferts se font d'une manière explicite dans

le logiciel par appel à des fonctions de l'API de gestion du *MFC* et ils sont gérées par le *Noc* au travers de contrôleurs mémoire et de l'arbitre de bus. L'architecture du Bus et la topologie du réseau impose quelques contraintes qui jouent un rôle primordial dans la minimisation des latences des communications point à point.

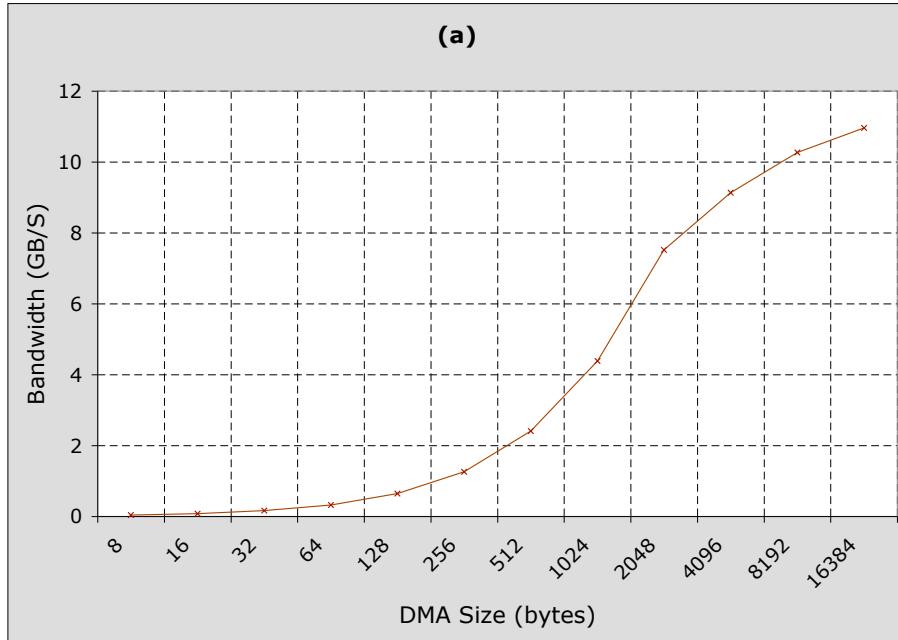


FIG. 3.7 – Influence de la taille du transfert DMA sur la bande-passante

Taille du Transfert Le premier paramètre influant sur la bande passante est la taille du bloc de données transféré. D'une part, il y a des contraintes imposées par l'API de transfert qui limite les tailles d'un bloc transféré par une commande DMA à 1, 2, 4, 8 octets ou tout multiple de 16 bytes et la taille de données maximale transférée en une seule commande est de 16 KB. Des plus les adresses doivent obligatoirement être alignées sur 16 bytes et un alignement sur 128 bytes est préférable à cause de la taille de la ligne de cache sur la mémoire locale du SPE qui est de 128 bytes. D'autre part l'espace mémoire disponible sur les SPEs pour stocker données et instructions est limité à 256 KB. Toutes ces contraintes, imposent une attention des contraintes fortes en termes de taille de bloc transféré et d'alignement des données qui ne sont pas du ressort du développeur dans les architectures à mémoire partagée. Plusieurs *benchmarks* ont été effectués dans [13] et [4] et qui démontrent la relation entre la taille et à la fois le débit et la latence

des transferts sur le Cell. Le graphe ?? donne les résultats sur un benchmark de bande-passante que nous avons effectué sur une BladeCenter QS20 et démontrent que celle-ci est proportionnelle à la taille des données transférées. Ceci s'explique par le fait que la *latence* du transfert qui représente le temps d'initialisation d'un transfert, cette durée étant la même quelque soit la taille du paquet jusqu'à 16 KB.

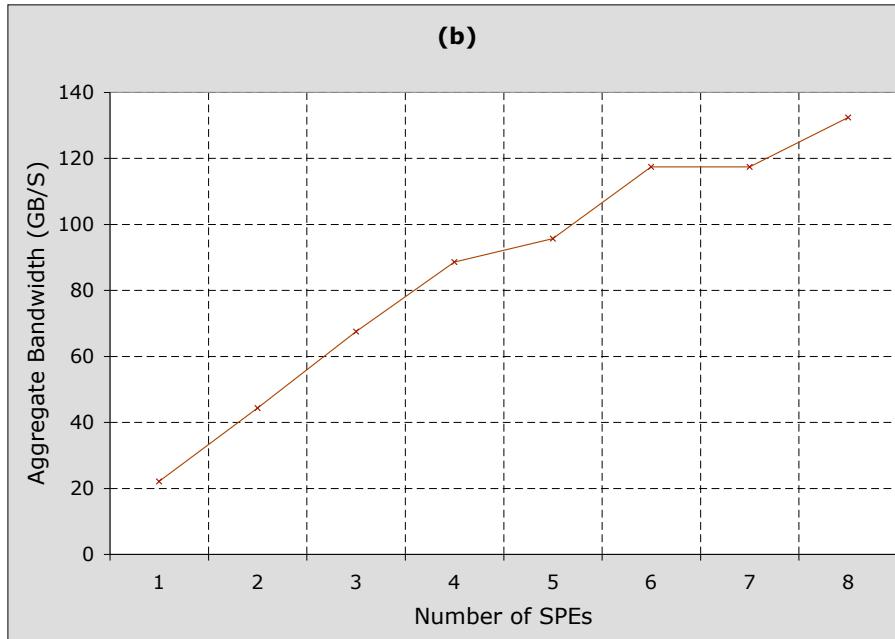


FIG. 3.8 – Influence du nombre de transferts dans le cas d'une communication LS<->LS

Nombre de Transferts Concurrents Les second facteur qui influe sur l'efficacité du réseau lors des transferts est le nombre de transferts s'exécutant en parallèle. En effet, l'architecture du réseau dont la topologie est du type *token ring* contient quatre anneaux d'une largeur de 128-bit : deux dans le sens d'une aiguille d'une montre et les deux autres dans le sens contraire. En observant la topologie du réseau ainsi que le sens de circulation des données sur les anneaux du bus, il s'avère évident qu'il peut y avoir collision entre deux transferts s'exécutant de manière concurrente (3.2.2). Sachant qu'un anneau peut gérer 3 transferts concurrents tant que ceux-ci n'entrent pas en collision. Ce risque est d'autant plus important si le nombre de transferts concurrents augmente (au-delà de 12). Lorsqu'un tel conflit est détecté, l'arbitre de bus le résout avec un surcout qui divise globalement la bande-passante par deux. La courbe sur le figure 3.2.2 permet de consta-

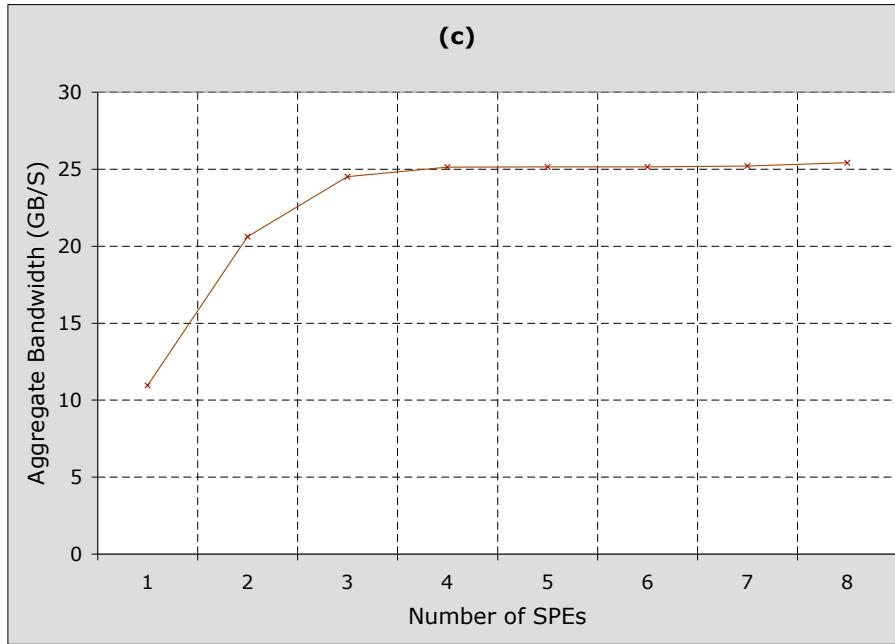


FIG. 3.9 – Influence du nombre de transferts dans le cas d'une communication MS<->LS

ter l'influence du nombre de transferts concurrents, on peut ainsi observer que la courbe perd sa linéarité quand au delà de 4 SPEs et ceci car le nombre de transferts devient trop important pour ne pas provoquer de collisions sur le bus.

Dans le cas d'une communication du PPE vers le SPE, la bande passante maximale qui peut être atteinte est de 25.6 GB/s. Dans le cas où plusieurs SPEs font une requête vers la mémoire principale les transferts ne peuvent être que sérialisés car il existe qu'une seule liaison vers le MS (*Main Storage*). On observe alors que le graphique de la figure ?? que la bande passante maximale n'est atteinte que lorsqu'au moins 4 SPEs font un transfert de la mémoire centrale vers leurs mémoires privée.

Optimisation de la localité temporelle par chainage des opérateurs

Le but visé par cette technique est de rapprocher le plus possible les données des unités de traitement. En effet, les couts liés au transfert des données de la mémoire centrale vers les mémoires locales étant important, il est pertinent de garder les données en mémoire locale après chaque traitement au lieu de multiplier les lectures/écritures vers la mémoire centrale. Les règles de chainage des opérateurs au niveaux des accès mémoires, sont les mêmes que pour la composition des opérateurs citée dans la section

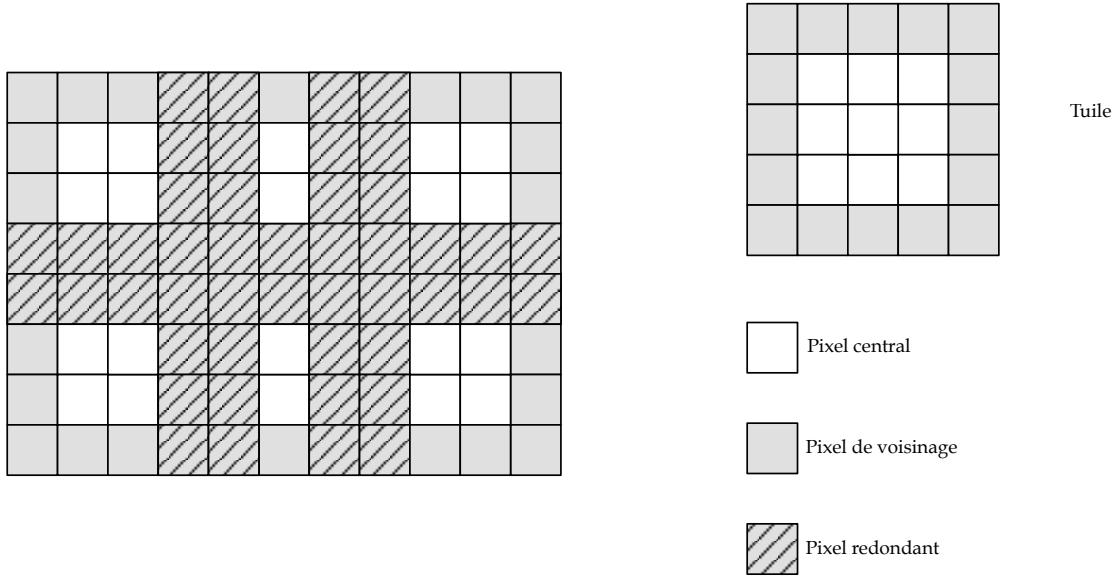
3.2.1. Cette optimisation apporte beaucoup à la performance globale car l'application est caractérisée par un ratio transfert/calcul important et la différence de bande-passante entre l'accès en mémoire locale et l'accès à une mémoire distante par DMA est d'un facteur dix [4].

Optimisation du *Tiling* des données

Le *loop tiling* [21] ou *loop blocking* est une technique d'optimisation très utilisée que cela soit par les programmeurs ou par les compilateurs optimisants. Cette technique consiste en un découpage des données à différents niveaux de la hiérarchie mémoire de telle sorte à ce que la latence d'accès soit la plus petite possible. Dans une architecture mémoire partagée contenant des caches, ceci revient à un découpage qui garantit que les données utilisées par un traitement donné tiennent toujours dans le cache. Ainsi, le temps d'accès aux données est de l'ordre du cycle et les défauts de cache (*cache misses*) sont quasi inexistant quelque soit la taille des données.

Au vu de la nature de la hiérarchie mémoire du processeur Cell, le *tiling* est une obligation. D'une part, il n'existe pas de mémoire cache pour gérer les mémoires privée des SPEs, le découpage des données est une tâche confiée au programmeur. D'autre part l'espace de stockage étant limité dans les *local store* (256 KB), le découpage des données en tuiles pouvant tenir dans le cache est obligatoire. L'unité de données atomique devient alors la tuile qui représente le morceau de donné le plus petit qui va être traitée par le code du SPE.

Taille de la Tuile La taille de la tuile est un paramètre primordial lorsqu'il s'agit de découper les données de manière optimale. Dans le cas du processeur Cell, les contraintes imposées par l'architecture font que le choix de la taille optimale est restreint. En effet, la taille limitée du *local store* pour le code source et les données, impose que la taille de la tuile ne dépasse pas la capacité de stockage qui est de 256 KB. L'autre paramètre dont on doit tenir compte est le nombre d'entrées et de sorties des fonctions de traitement car celui-ci donne le nombre de tuiles. On peut en déduire globalement, que la taille de la tuile est égale à la capacité de stockage restante pour les données, divisée par le nombre de tuiles en entrée et en sortie de la chaîne de traitement mise en jeu. D'autre part, selon ce qui a été vu précédemment, la taille de transfert qui garantit une bande-passante maximale sur le bus est de 16 KB ou un multiple de cette taille.


 FIG. 3.10 – Redondances de données pour un opérateur de convolution 3×3

Dimensions de la Tuile Les tuiles que l'on traite dans notre cas sont en général d'une forme rectangulaire. Toutefois, à cause de la présence d'opérateurs de convolution, les dimensions hauteur et largeur de la tuile (h et w) doivent également être considérées. En effet, dans le cas d'opérateurs de convolution les bords qui représentent les voisinages des pixels traités augmentent la quantité de données transférées de la mémoire. Cette quantité peut être réduite avec un choix judicieux des dimensions de la Tuile. Comme le montre la figure 3.2.2 certains des pixels sont rechargés, nous démontrons dans la suite l'influence des dimensions de la tuile sur ce nombre de pixels redondants.

Si l'on considère une image de hauteur H et de largeur W , une tuile de dimensions h et w et un opérateur de convolution nécessitant un voisinage pour chaque pixel. On suppose pour simplifier le calcul, que la matrice de convolution est carrée, ce qui fait que les bords des deux côtés sont égaux. La quantité totale de pixels transférées pour le traitement est alors :

$$Q = (h + 2b) \times (w + 2b) \times nb_{tiles}$$

où nb_{tiles} est le nombres de tuiles dans l'image.

$$nb_{tiles} = \frac{H \times W}{h \times w}$$

Le but étant de trouver à taille de tuile constante $h \times w$ quels sont les dimensions h et w qui minimisent Q

$$Q = (h + 2b) \times (w + 2b) \times \frac{H \times W}{h \times w}$$

Posons alors $\lambda = h \times w = C^{te}$, la fonction à minimiser devient alors :

$$Q = (h + 2b) \times (w + 2b) \times \frac{H \times W}{\lambda}$$

Calculons alors les dérivées :

$$\frac{\partial Q}{\partial h} \text{ et } \frac{\partial Q}{\partial w}$$

$$Q = \frac{H \times W}{\lambda} (h + 2b) \times \left(\frac{\lambda}{h} + 2b\right) \text{ et symétriquement } Q = \frac{H \times W}{\lambda} (w + 2b) \times \left(\frac{\lambda}{w} + 2b\right)$$

$$\frac{\partial Q}{\partial h} = \frac{H \times W \times 2b}{\lambda \times h^2} (h^2 - \lambda)$$

$$\frac{\partial Q}{\partial w} = \frac{H \times W \times 2b}{\lambda \times w^2} (w^2 - \lambda)$$

Le minimum de la fonction Q est atteint lorsque :

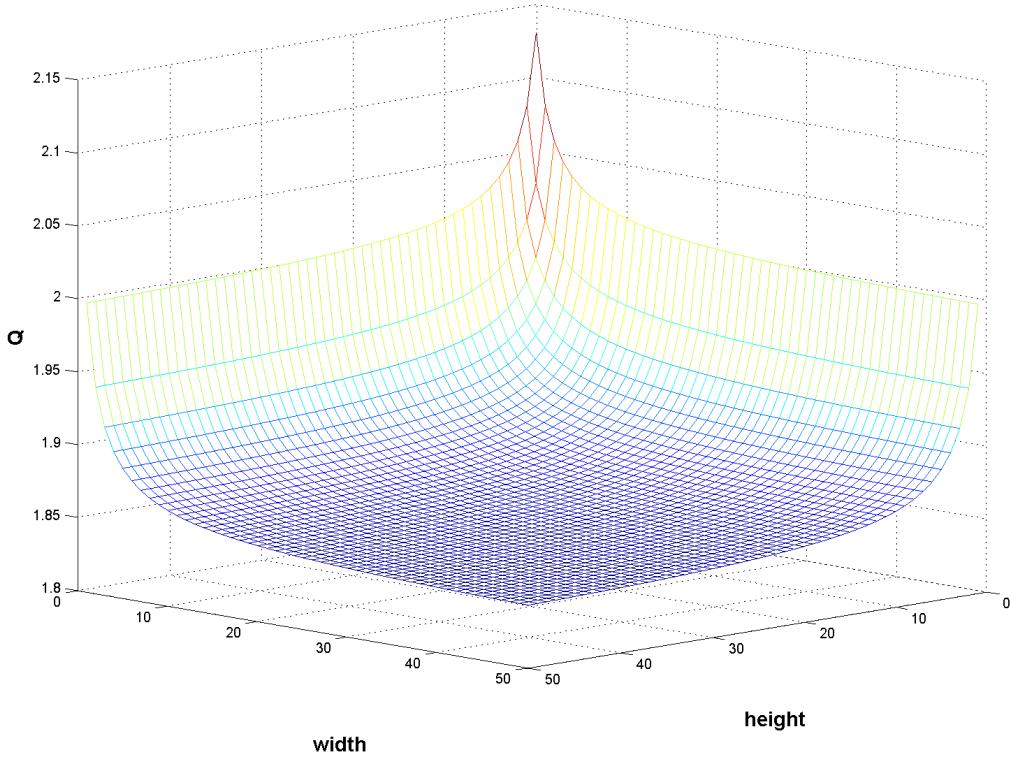
$$\frac{\partial Q}{\partial h} = 0 \text{ et } \frac{\partial Q}{\partial w} = 0$$

ce qui donne

$$h = w = \sqrt{\lambda}$$

Ce qui permet de déduire que la forme de la tuile qui minimise la quantité de données transférée est une forme carrée. D'autre part lorsque l'on observe le tracé de la fonction Q sur la figure 3.11 on peut constater que cette fonction décroît quand h et w augmentent et que la valeur de Q est minimale pour une surface de tuile donnée ($h \times w = C^{te}$) lorsque $h = w$ ce qui correspond à la première bissectrice du plan (hw). Ceci permet d'affirmer d'une part que la tuile carrée est optimale pour une surface donnée et que la valeur de Q diminue d'autant plus lorsque h et w augmentent. Les valeurs de h et w sont alors limitées par l'espace mémoire disponible pour une tuile dans la mémoire locale des SPEs.

Ce résultat nous a permis de démontrer que la forme de la tuile avait une influence sur la quantité de données transférées et par conséquent sur la performance globale de l'application. Cependant, ce découpage n'est pas forcément optimal lorsqu'on passe à


 FIG. 3.11 – Tracé de la fonction $Q(h, w)$ dans l'espace

l'implémentation. En effet, des tuiles de formes carrées signifient des accès à des zones non-contigues de la mémoire. Ce type d'accès est en général couteux car il provoque des sauts dans la mémoire. De plus, sur le processeur Cell, ceci se traduit en commandes DMA sur des zones non-contigues de la mémoire ce qui nécessitent des commandes du type *DMA list*. Ces dernières requièrent la création d'une liste qui contient chaque DMA élémentaire et qui est d'autant plus grande que le nombre de transferts est important. Cette liste doit également être mise à jour lors de chaque nouveau transfert. Toutes ces contraintes nous ont poussé dans un premier temps à adopter un découpage en bandes qui consiste à ce que les tuiles aient une largeur égale à celle de l'image. Ceci permet que les accès se fassent uniquement sur des zones contigues de la mémoire et que les transferts puissent se faire avec une seule commande DMA. De plus, lorsqu'une tuile ne contient pas de bords latéraux comme dans notre cas, le problème d'alignement des transferts est également contourné. Par contre, ce choix induit des limitations en terme

de taille d'image pouvant être traitée. En effet, sachant que la taille de la tuile est limitée et que sa largeur est égale celle de l'image, la hauteur de la tuile elle, diminue au fur et à mesure que la largeur de l'image augmente. De ce fait, les accès non-contigus sont nécessaires pour des tailles d'images très grandes.

3.2.3 Schémas de Parallélisation

Dans ce qui suit, nous abordons l'optimisation de notre algorithme à un niveau d'abstraction plus haut qui est celui des tâches. L'architecture du Cell permet plusieurs placements possibles du graphe d'opérateurs par la présence de 8 SPEs et la possibilité de mettre en place différents schémas de communication. Dans les figures qui suivent les opérateurs sont représentés par des cercles, les processeurs par des rectangles à coins arrondis, les tuiles sont de forme rectangulaire et peuvent contenir des bords. Les flèches à trait fin représentent des instruction LOAD/STORE dans la mémoire privée du SPE alors que les flèches plus épaisses représentent des commandes DMA inter-SPE ou alors entre la mémoire centrale et le *local store* d'un SPE.

SPMD Conventionnel

Dans ce schéma de déploiement, l'image est divisée en 8 régions de même taille, afin que chacun des 8 SPEs ait une charge de calcul équivalente. Tous les SPEs exécutent le même code. Les opérateurs sont exécutés successivement sur l'image entière l'un après l'autre. A titre d'exemple l'opérateur de multiplication n'est exécuté que lorsque le calcul du filtre de *Sobel* est achevé sur toute l'image. Ce modèle de calcul est dit *data-parallel* car les données sont envoyées en parallèle sur les SPEs et traitées de manière complètement indépendantes les unes des autres. Toutefois, la bande passante sur le bus mémoire centrale vers *local store* est beaucoup sollicitée car les données sont systématiquement lues et écrites avant et après chaque opérateur.

Pipeline Conventionnel

Cette implémentation de l'algorithme consiste à déployer le graphe d'opérateur sous forme de pipeline. L'image n'est pas subdivisée en régions de traitement mais chaque tuile traitée traverse le pipeline avant qu'une nouvelle tuile ne puisse le faire. L'algorithme est par conséquent fortement séquentialisé. Par contre la bande-passante inter-SPEs est bien exploitée car la majorité des transferts se font entre SPEs et par conséquent

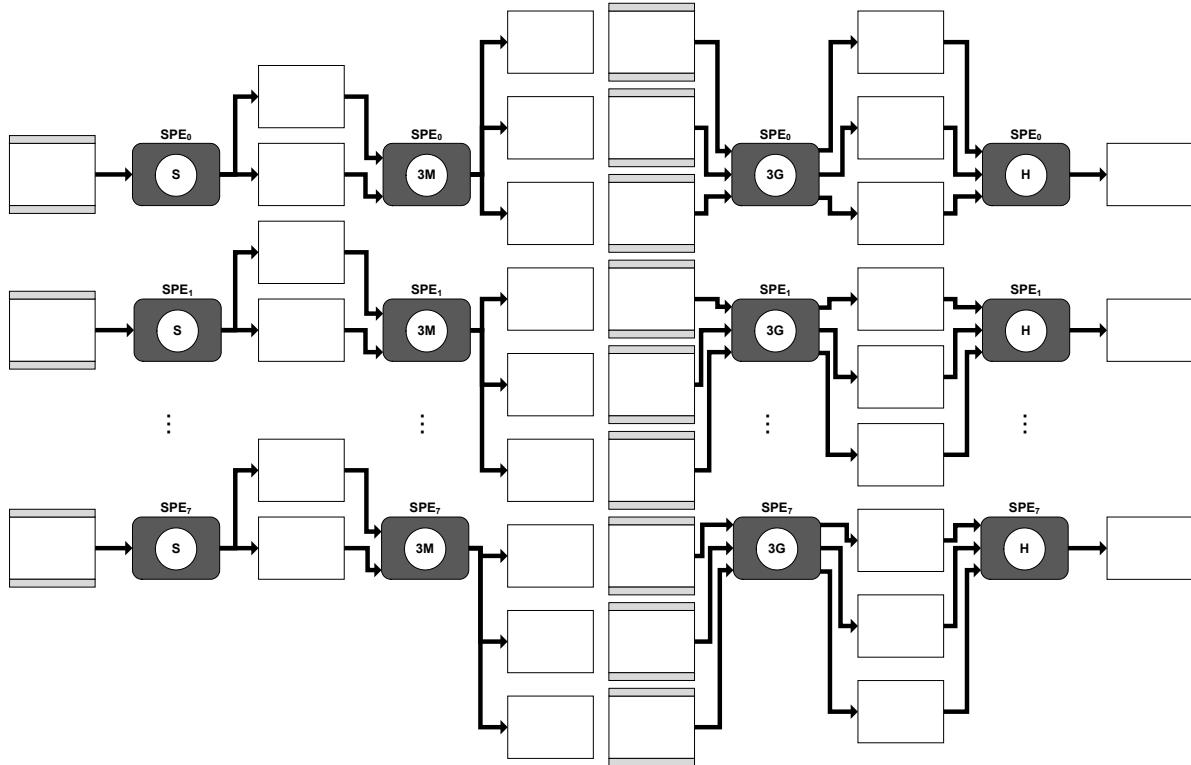


FIG. 3.12 – Schéma de Parallélisation SPMD Conventionnel

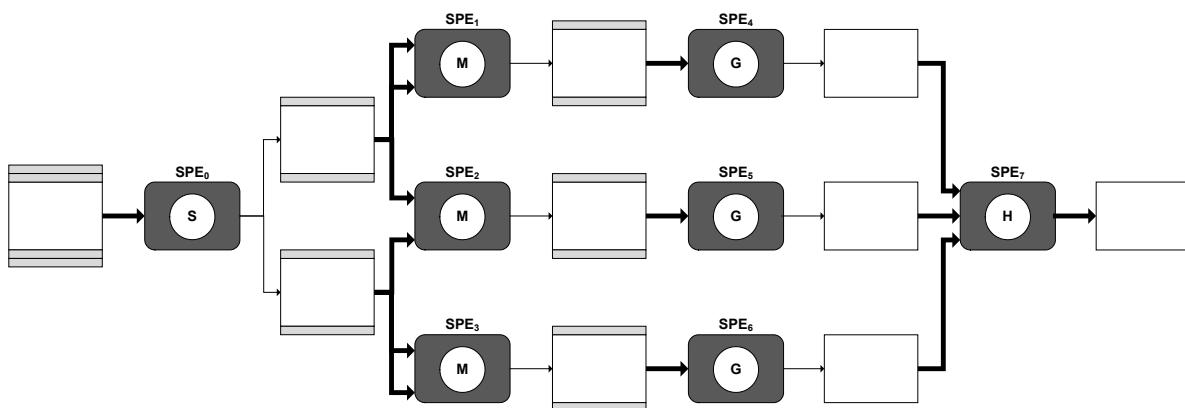


FIG. 3.13 – Schéma de Parallélisation Pipeline

la pression exercée sur le bus précédemment est atténuée car elle est répartie sur l'ensemble de l'anneau.

Chainage d'opérateur par paires

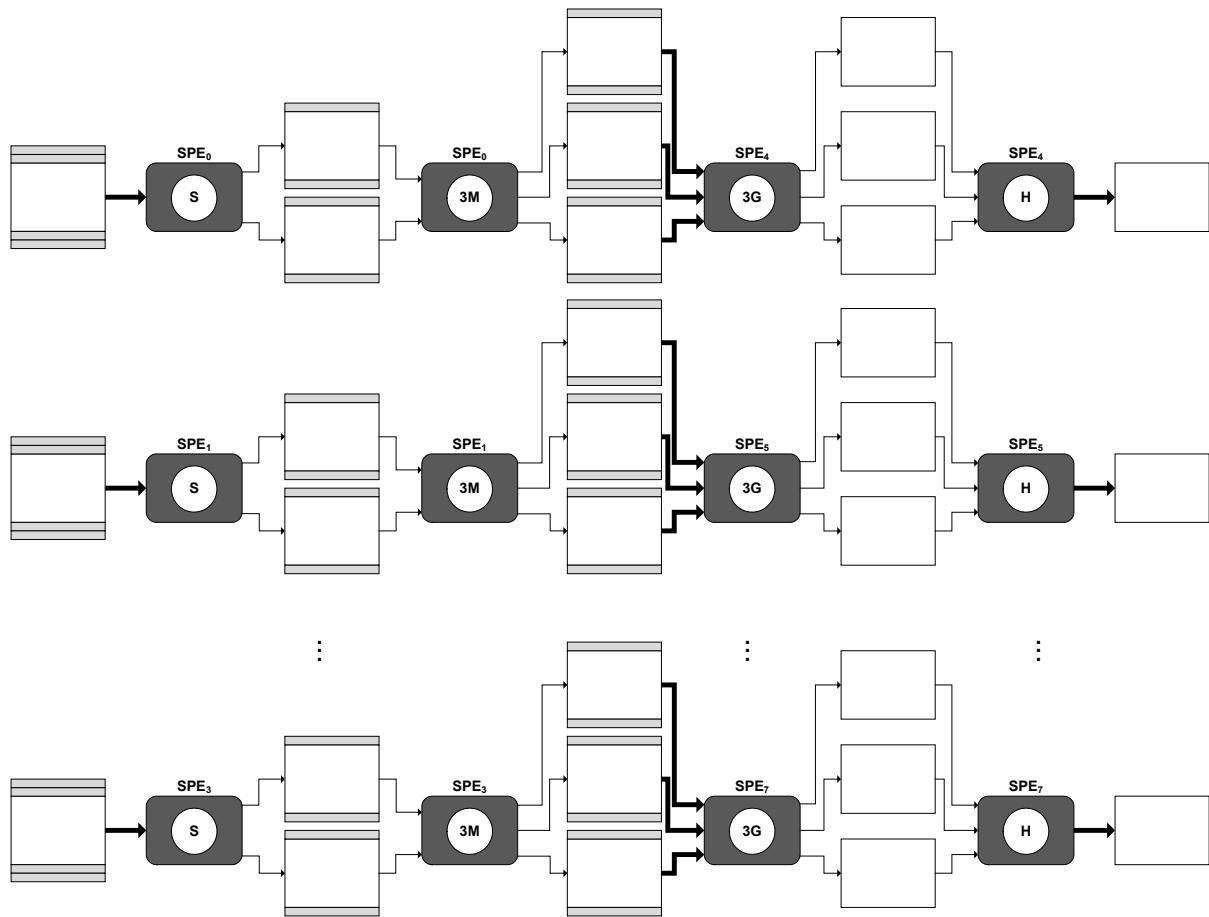


FIG. 3.14 – Schéma de Parallélisation Chainage d'opérateurs par paires

Dans cette version deux opérateurs successifs sont placés sur le même SPE. Ainsi, les opérateur *Sobel* et *Mul* sont placés sur un même SPE et *Gauss* et *Coarsity* sont placés sur un autre SPE. Le nombre de SPEs dans un processeur Cell étant de 8, ce schéma permet d'avoir 4 régions de l'image traitées en parallèle. Si l'on compare cette version à la précédente, on notera que la pression sur le bus d'interconnexion est plus importante car les transferts concurrents sont plus nombreux et par conséquent le risque de contention du bus est plus probable.

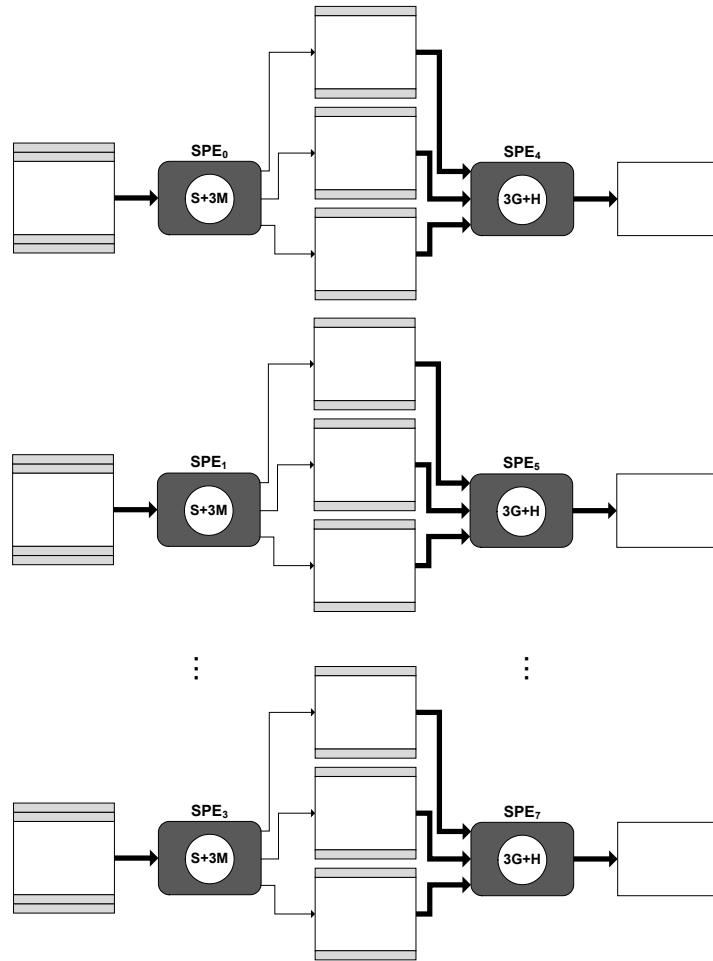


FIG. 3.15 – Schéma de Parallélisation Chainage et fusion d'opérateurs par paires

Chainage et fusion d'opérateur par paires

Cette version est une variante de celle qui la précède. L'idée étant de fusionner les opérateurs présents sur un même SPE, et ce afin d'éliminer les instructions de LOAD et STORE présent à la sortie de du premier et à l'entrée du second. De ce fait, le nombre de cycles peut être considérablement réduit, surtout lorsque l'on sait que la latence des instructions mémoire est de 6 cycles sur les SPE [11].

Chaînage entier et fusion d'opérateur par paires

Dans cette implémentation tous les opérateurs sont exécutés sur le même SPE. D'une part ceci permet d'éviter les transferts DMA entre SPE et minimise donc les transactions

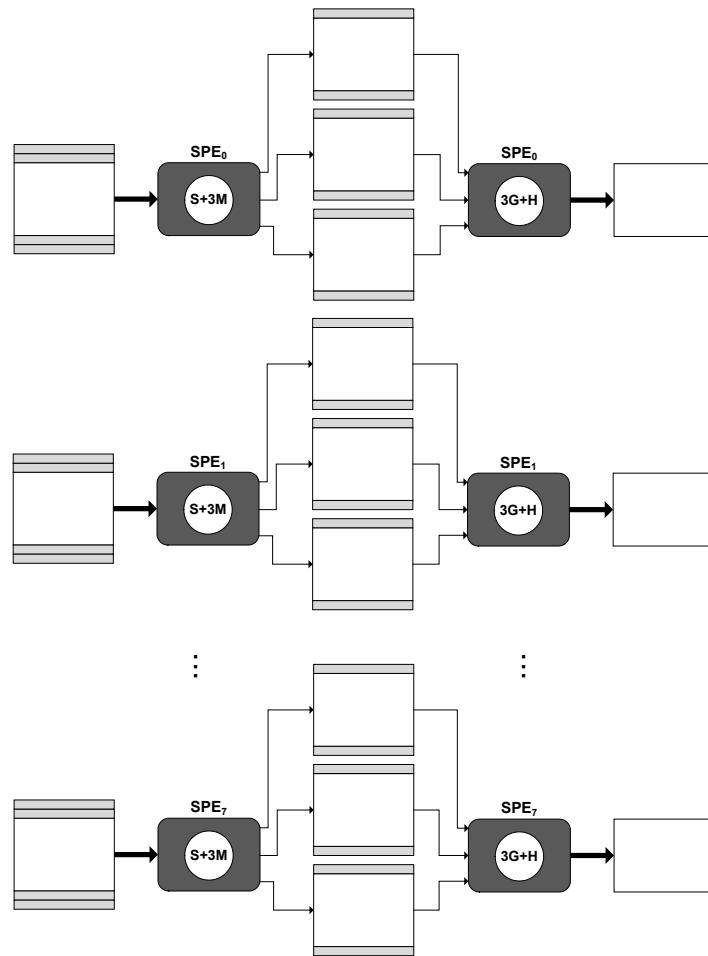


FIG. 3.16 – Schéma de Parallélisation Chaînage et fusion d'opérateurs par paires

sur le bus d'interconnexion.

3.3 Évaluation des Performances

après avoir exposé les différentes techniques d'optimisations nous procédons à la mesure de performance pour les différents modèles de déploiement précédents. Il s'agit ici de comparer les modèles en terme de temps d'exécution sur le processeur Cell. Dans un deuxième temps nous évaluons l'influence des transferts sur la performance globale de chacun des modèles.

3.3.1 Métriques de Mesure

La métrique que nous avons choisi d'utiliser pour la mesure de performance, est le nombre de cycles moyen par pixel ou *cpp*,

$$cpp = \frac{nombre_{cycles\ cpu}}{nombre_{pixels}} = \frac{nombre_{cycles\ cpu}}{H \times W}$$

. Les bancs de tests ont été conçus de telle sorte à mesurer d'une part la performance brute ainsi que d'autres métriques spécifiques aux architectures parallèles qui sont des métriques de passage à l'échelle (*scalability*) notamment l'accélération *speedup* et l'efficacité *efficiency*.

3.3.2 Méthode et Plateforme de Mesure

La plate-forme sur laquelle a été menée l'évaluation des performances est un Blade QS20 d'IBM cadencé à 3.2 GHz. Cette lame contient deux processeurs Cell, chacun relié à une mémoire de 512 MB, ce qui donne au total 1 GB de *RAM* disponible. Le timer utilisé pour la mesure de la performance est appelé *timebase* et il est cadencé à 14.318 *Mhz*. L'OS installé est un Linux, distribution Fedora 7. Le développement a été réalisé sur le *Cell SDK 3.0* et les compilateurs utilisés sont *ppu-gcc* et *spu-gcc*, ce qui implique une compilation du type *dual-source* un pour le SPE et un pour le PPE.

La méthodologie de mesure adoptée est en accord avec les systèmes de vision par ordinateur. Nous avons simulé le cas d'un flux d'images continu, et avons mesuré le temps d'exécution moyen pour une image. Ceci permet de négliger l'*overhead* induit par la création et la synchronisation des *threads*. L'interface utilisateur permet de faire varier plusieurs paramètres, notamment la taille de l'image, la taille de la tuile et le nombre de SPEs.

3.3.3 Comparaison des Schémas de Parallélisation

La figure 3.17 donne une comparaison entre les différents modèles de déploiement de l'algorithme de Harris sur le processeur Cell décrits précédemment. Ce que l'on peut observer en premier lieu c'est que le schéma *pipeline* donne les plus mauvaises performances ce qui était attendu : cette version est volontairement sérialisée et n'exploite pas entièrement le parallélisme de données offert par l'architecture du Cell. Les autres résultats correspondent également à nos attentes :

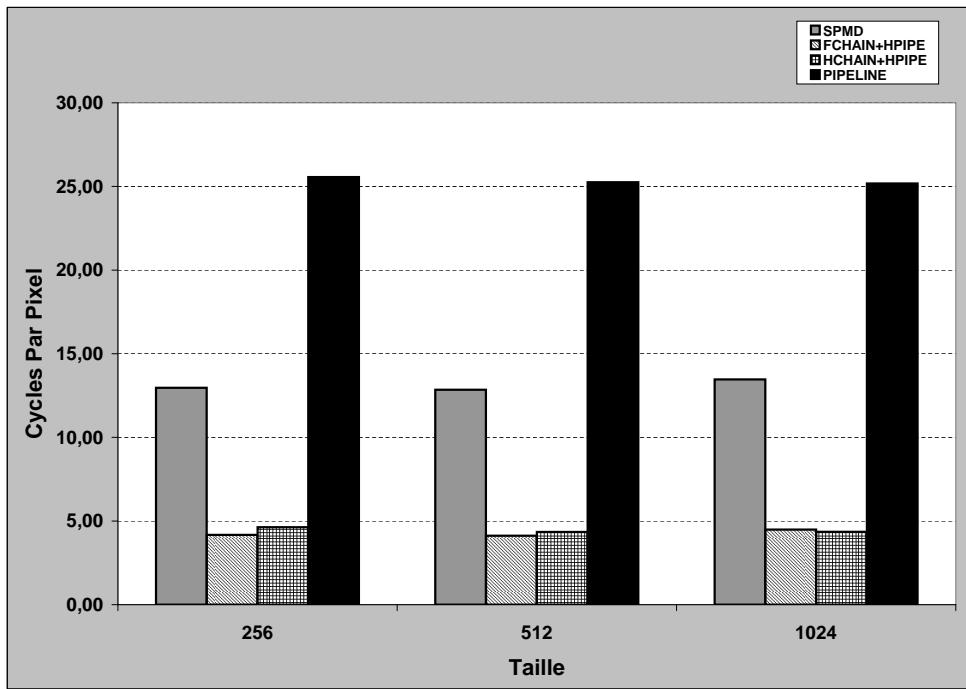


FIG. 3.17 – Comparaison des modèles en cycles par point

- Nos techniques d'optimisation des accès mémoires améliorent sensiblement la performance globale car la version la plus rapide est celle où les fonctions sont composées deux par deux et où tous les opérateurs sont exécutés sur le même SPE (remplacement des DMA par des LOAD/STORE).
- Les versions où les DMA inter-SPE sont utilisés possèdent de bonnes performances : à titre d'exemple la version sans composition de fonction et en regroupant deux opérateurs par SPE est plus rapide que la version SPMD.
- De plus, la version avec composition de fonction ou les paires d'opérateurs sont sur deux SPEs différents est presque aussi rapide que la même version où tous les opérateurs sont placés sur le même SPE. Ceci prouve que la bande-passante inter-SPE est comparable à celle des instructions LOAD/STORE sur les mémoires locales.

Dans ce qui précède nous avons pu constater que le placement optimal d'un graphe de fonctions sur le processeur Cell n'était pas forcément évident lorsqu'il s'agit d'un algorithme de traitement d'image comme dans notre cas. D'une part, l'on a vu que le choix d'un schéma complètement *data parallel* n'était pas forcément optimal, et qu'un schéma mixte avec une partie *pipeline* imbriqué dans un schéma *data parallel* donnait un résultat aussi bon qu'un schéma basé purement sur le parallélisme de données. Ceci est en

grande partie possible, grâce à un niveau supplémentaire de parallélisme au niveaux de transferts. En effet, en favorisant les communications inter-SPE on profite pleinement de la bande passante du bus interne (204.8 GB/s) qui peut gérer jusqu'à 12 transferts en parallèle alors que dans un schéma de transferts où plusieurs SPEs communiquent avec la mémoire centrale les commandes sont forcément sérialisées et dans ce cas là la bande passante est limitée à 25.6 GB/s.

3.3.4 Influence de la Taille de la Tuile

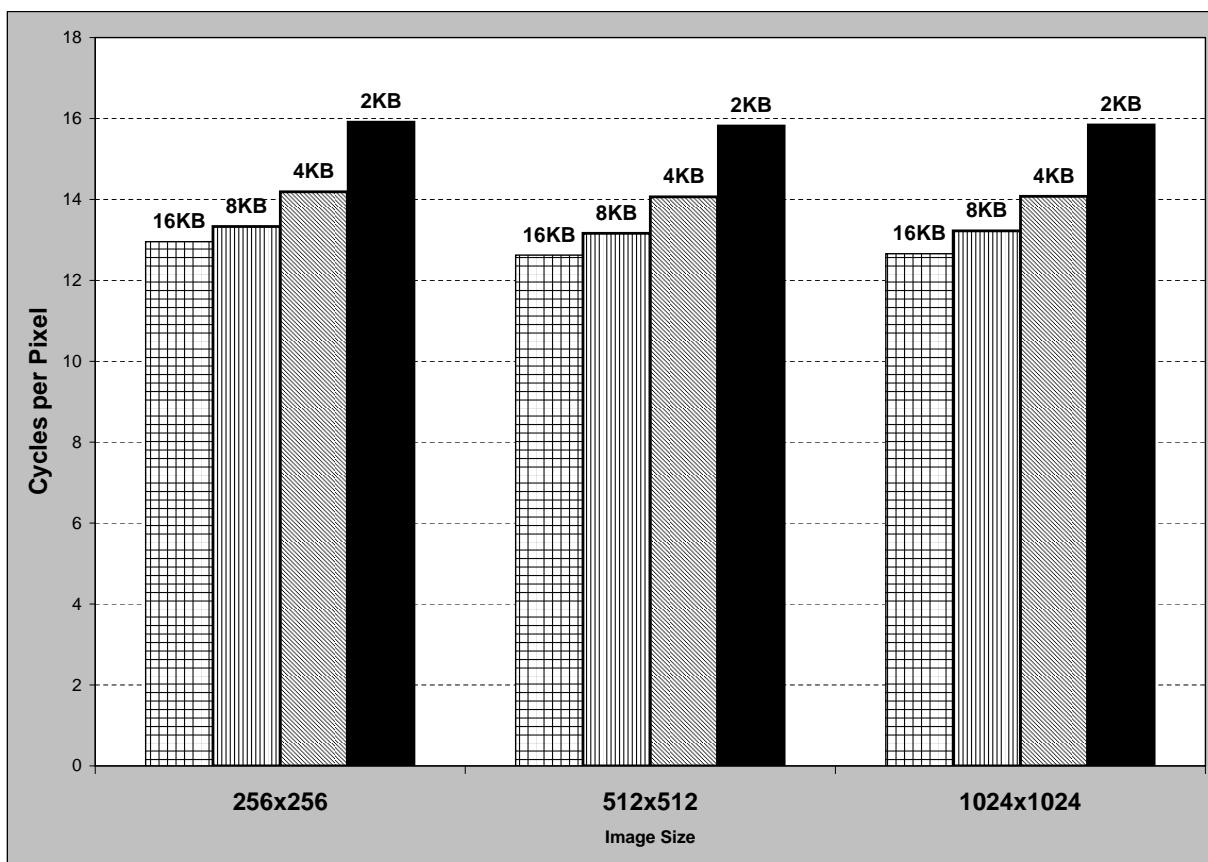


FIG. 3.18 – Influence de la taille de la tuile sur la performance pour la version chainage entier et fusion d'opérateur par paires 1 SPU

Comme il est démontré dans [4] et [13] la taille du bloc de données transféré possède une influence sur la bande-passante du bus interne. Dans notre domaine d'application, la bande-passante mémoire est critique pour la performance globale de l'application, car les algorithmes de traitement d'image sont généralement caractérisés par un ratio

Modèle	Opérateur	Nombre de Cycles	Speedup
Halfchain	Sobel+Mul+LOAD/STORE	119346	x
Halfchain+Halfpipe	Gauss+Coarsity+LOAD/STORE	188630	x
Halfchain	(Sobel \circ Mul)+LOAD/STORE	16539	7.2
Halfchain+Halfpipe	(Gauss \circ Coarsity)+LOAD/STORE	504309	3.5

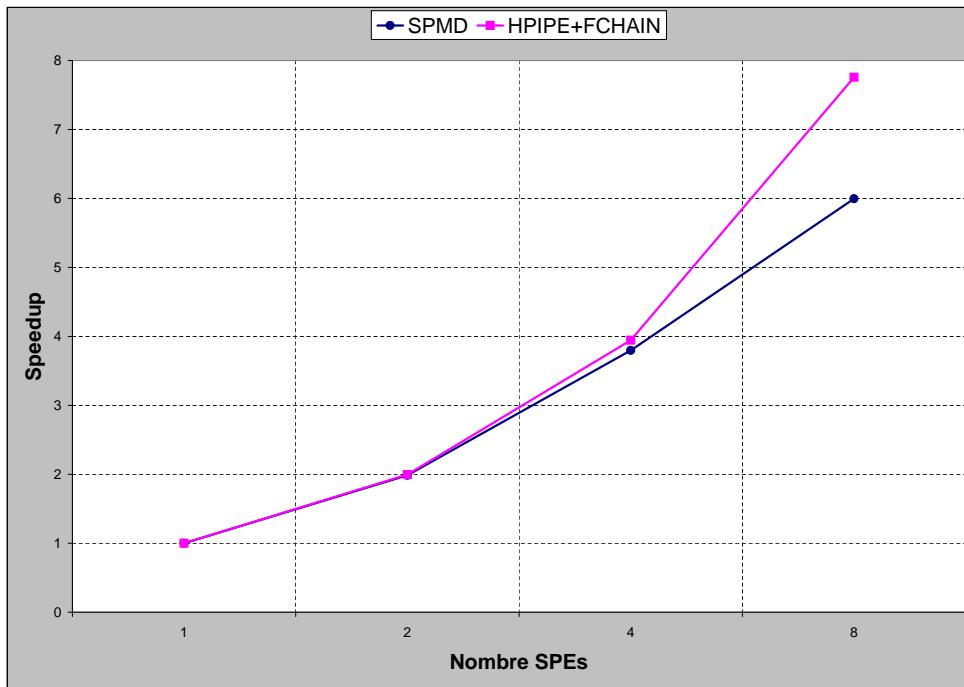
TAB. 3.7 – Apport de la composition de fonction à la performance

transfert/calcul important. La figure 3.18 nous montre que la taille de tuile donnant les meilleures performances est 16KB, et ceci pour deux raisons :

- Une taille de bloc de 16 KB ou multiple de cette taille, garantit une bande-passante maximale sur le bus interne.
- Comme nous l'avons vu dans la discussion sur la forme de la tuile, il est démontré que plus la taille de tuile est grande moins il y a de pixels redondants, et la quantité de données totale transférées est minimisée en ce qui concerne les noyaux de convolution.

3.3.5 Analyse des Résultats

La comparaison de la performance globale des différents schémas d'implémentation ne suffit pas à prouver que l'optimisation de l'utilisation de la mémoire sont les facteurs principaux de l'amélioration de la performance. Dans le but de donner une analyse plus précise et donc plus complète, nous avons effectué des mesures au niveau du SPE où nous avons estimé le gain apporté par la composition des fonctions et les transferts DMA inter-SPE. Le tableau 3.7 donne en nombre de cycles *SPU* les versions où les opérateurs ne sont pas composés deux à deux et celle où elles le sont. Comme on peut le constater, cette optimisation permet d'avoir une accélération pouvant atteindre $\times 7.2$. Cette technique, permet une utilisation maximale des registres au détriment de la mémoire et par conséquent elle est limitée par la taille du banc de registres du processeur (128 registres pour le SPU). D'autre part l'optimisation de la localité des données en chainant les opérateurs d'un même SPE est certe bénéfique, mais elle est limitée par le niveau de hiérarchie mémoire juste au dessus des registres qui est le *local store*, limité lui à 256 KB.


 FIG. 3.19 – Mesure du *Speedup* en fonction du nombre de SPEs

3.3.6 Mesure des Métriques de Passage à l’Échelle

Lorsque l'on mesure les performances d'une machine parallèle on ne peut se contenter de mesurer les performances temporelles brutes. Il est également intéressant de s'attarder sur d'autres paramètres qui permettent de comparer deux architectures parallèles ou alors d'isoler un goulot d'étranglement qui limite la performance. Ce que l'on veut évaluer est le passage à l'échelle ou *scalability* qui permet de mesurer sur un système à plusieurs processeurs, le rendement des N unités par rapport à un seul. Les métriques les plus connues sont l'accélération (*Speedup*) et l'efficacité (*Efficiency*), et sont définies comme suit :

$$Speedup = \frac{\text{Temps d'Exécution sur 1 Processeur}}{\text{Temps d'Exécution sur } P \text{ Processeur}}$$

$$Efficiency = \frac{\text{Temps d'Exécution sur 1 Processeur}}{P \times \text{Temps d'Exécution sur } P \text{ Processeur}}$$

Ces mesures sont essentielles pour l'adaptation d'un code séquentiel à une machine parallèle. Elles permettent d'une part de détecter des limitations d'un système parallèle pour la parallélisation d'un code. Par exemple, on peut évaluer la complexité due au contrôle des *threads* et à leur synchronisation. D'autre part, on peut mesurer l'efficacité du réseau

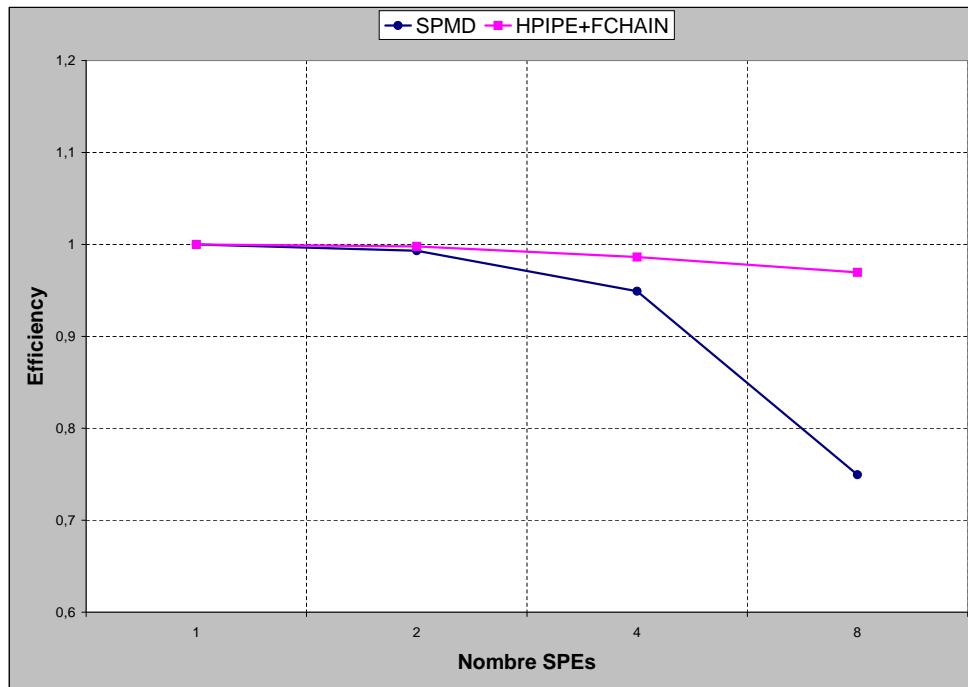


FIG. 3.20 – Mesure de l'Efficacité en fonction du nombre de SPEs

d'interconnection à distribuer les données de manière parallèle au différents noeuds. Elles peuvent également justifier l'ajout d'une quantité supplémentaires de processeurs si cela n'altère pas la performance en induisant un coût de gestion supplémentaire.

D'après les figures 3.19 et 3.20 on peut constater que le Cell est une architecture ayant une bonne "scalabilité". En effet, on peut noter dans les deux cas étudiés à la fois, un *speedup* proche de P le nombre de SPEs et une efficacité proche de 1. On peut tout de même noter certaines différences entre les deux implémentations étudiées, d'une part la version SPMD et d'autre part la versions avec composition de fonctions et chainage sur le même SPE. La première implique une pression importante sur le bus de données car presque tous les lectures/écritures mémoire se font sur la mémoire externe. Au contraire, la seconde minimise le trafic vers la mémoire externe. Ceci explique, la différence au niveau de l'efficacité et permet de déduire que plus le ratio transfert/calcul est grand est moins le passage à l'échelle est bon.

Bibliographie

- [1] R. M. Badia, D. Du, E. Huedo, A. Kokossis, I. M. Llorente, R. S. Montero, M. Palol, R. Sirvent, and C. Vázquez. Integration of grid superscalar and gridway meta-scheduler with the drmaa ogf standard. In *Euro-Par '08 : Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 445–455, Berlin, Heidelberg, 2008. Springer-Verlag.
- [2] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss : a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.
- [3] Murray I. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989.
- [4] X. Martorell D. Jimenez-Gonzalez and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. In *in Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, 2007.
- [5] Joel Falcou. High Level Parallel Programming EDSL - A BOOST Libraries Use Case. In *BOOST'CON 09*, Aspen, CO, May 2009. BOOSTPRO Consulting.
- [6] Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In *PARCO*, pages 243–252, 2007.
- [7] Kayvon Fatahalian, Thimothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia : Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

- [8] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9) :948–960, September 1972.
- [9] Message P Forum. Mpi : A message-passing interface standard. Technical report, Message Passing Interface Forum, Knoxville, TN, USA, 1994.
- [10] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th ALVEY Vision Conference*, 1988.
- [11] IBM. *Cell Broadband Engine Programming Handbook*. IBM, version 1.0 edition, 2006.
- [12] IEEE. IEEE Standard . 1003.1c-1995 thread extensions. Technical report, IEEE, 1995.
- [13] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network : Built for speed. *IEEE Micro*, 26(3) :10–23, 2006.
- [14] Michael D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *Proceeding of GSPx Multicore Applications Conference*, 2006.
- [15] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *Robotics Institute, Carnegie Mellon University & doctoral dissertation*, 1980.
- [16] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. In *Proceedinf of International Workshop on OpenMP 2007*, 2007.
- [17] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne, and Daniel Etiemble. Algorithmic skeletons within an embedded domain specific language for the cell processor. In *PACT '09 : Proceedings of the 18th international conference on Parallel architectures and compilation techniques*. ACM, 2009.
- [18] Todd Veldhuizen. Expression templates. Technical Report 5, June 1995.
- [19] Todd L. Veldhuizen. C++ templates as partial evaluation. In *PEPM*, pages 13–18, 1999.
- [20] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4) :27–75, 1993.
- [21] M. Wolfe. More iteration space tiling. In *Supercomputing '89 : Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.