UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE : STITS

Institut d'Electronique Fondamentale

***DISCIPLINE** PHYSIQUE*

THÈSE DE DOCTORAT

soutenue le 06/11/2012

par

Tarik SAIDANI

Optimisation multi-niveau d'une application de traitement d'images sur machines parallèles

Directeur de thèse :	Alain MERIGOT	Professeur	Université de Paris Sud XI
Composition du jury :			
Président du jury :	Patrick GARDA	Professeur	Université Pierre et Marie Curie
Rapporteurs :	François IRIGOIN	Maître de Recherche	Mines ParisTech
	Jocelyn SEROT	Professeur	Université Blaise Pascal
Examinateurs :	Lionel LACASSAGNE	Maître de Conférences	Université Paris Sud XI
	François VERDIER	Professeur	Université Nice Sophia Antipolis

Remerciements

Je tiens à remercier Alain Mérigot pour m'avoir permis de poursuivre cette thèse sous sa direction. Je remercie également Lionel Lacassagne pour m'avoir encadré et initié au calcul parallèle, ainsi que pour son soutien moral et logistique pendant toute la période de rédaction et de révision du manuscrit. Je remercie Samir Bouaziz pour avoir co-encadré mes travaux et pour son soutien précieux dans les moments les plus difficiles. Je remercie Joel Falcou qui a apporté son savoir faire, déterminant pour la suite de ma thèse. Merci également à Claude Tadonki pour ses apports à mes travaux de thèse et surtout pour toute la joie et la bonne humeur que nous avons connu ensemble. Je remercie François Irigoin et Jocelyn Sérot pour avoir accepté de rapporter mon manuscrit et pour toutes les corrections qui ont grandement amélioré la qualité de mon travail.

Un grand merci à Clémence qui est à mes côtés depuis tant d'années et qui m'a soutenu, aidé et supporté pendant toute la durée de ma thèse et au-delà. Enfin, je remercie mes parents pour m'avoir tant aidé pendant mes études et pour m'avoir donné les moyens et la volonté de faire de longues études. Cette thèse est dédiée à ma chère mère qui nous a quitté trop tôt et qui avait tant de qualités que toutes les pages qui suivent ne peuvent suffire à les décrire.

Résumé

Cette thèse vise à définir une méthodologie de mise en oeuvre d'applications performantes sur les processeurs embarqués du futur. Ces architectures nécessitent notamment d'exploiter au mieux les différents niveaux de parallélisme (grain fin, gros grain) et de gérer les communications et les accès à la mémoire. Pour étudier cette méthodologie, nous avons utilisé un processeur cible représentatif de ces architectures émergentes, le processeur CELL. Le détecteur de points d'intérêt de Harris est un exemple de traitement régulier nécessitant des unités de calcul intensif. En étudiant plusieurs schémas de mise en oeuvre sur le processeur CELL, nous avons ainsi pu mettre en évidence des méthodes d'optimisation des calculs en adaptant les programmes aux unités spécifiques de traitement SIMD du processeur CELL. L'utilisation efficace de la mémoire nécessite par ailleurs, à la fois une bonne exploitation des transferts et un arrangement optimal des données en mémoire. Nous avons développé un outil d'abstraction permettant de simplifier et d'automatiser les transferts et la synchronisation, CELL MPI. Cette expertise nous a permis de développer une méthodologie permettant de simplifier la mise en oeuvre parallèle optimisée de ces algorithmes. Nous avons ainsi conçu un outil de programmation parallèle à base de squelettes algorithmiques : SKELL BE. Ce modèle de programmation propose une solution originale de génération d'applications à base de méta-programmation. Il permet, de manière automatisée, d'obtenir de très bonnes performances et de permettre une utilisation efficace de l'architecture, comme le montre la comparaison pour un ensemble de programmes test avec plusieurs autres outils dédiés à ce processeur.

Mots-clés : Programmation parallèle, Processeur Cell, Traitement d'images, Squelettes algorithmiques, Calcul hautes performances, Méta-programmation, processeur embarqué

Abstract

This thesis aims to define a design methodology for high performance applications on future embedded processors. These architectures require an efficient usage of their different level of parallelism (fine-grain, coarse-grain), and a good handling of the inter-processor communications and memory accesses. In order to study this methodology, we have used a target processor which represents this type of emerging architectures, the Cell BE processor. We have also chosen a low level image processing application, the Harris points of interest detector, which is representative of a typical low level image processing application that is highly parallel. We have studied several parallelisation schemes of this application and we could establish different optimisation techniques by adapting the software to the specific SIMD units of the Cell processor. We have also developped a library named CELL MPI that allows efficient communication and synchronisation over the processing elements, using a simplified and implicit programming interface. This work allowed us to develop a methodology that simplifies the design of a parallel algorithm on the Cell processor. We have designed a parallel programming tool named SKELL BE which is based on algorithmic skeletons. This programming model provides an original solution of a meta-programming based code generator. Using SKELL BE, we can obtain very high performances applications that uses the Cell architecture efficiently when compared to other tools that exist on the market.

Keywords : Parallel programming, Cell processor, Image processing, Algorithmic skeletons, High performance computing, Meta-programming, embedded processors

Table des matières

Introduction **i**

1 Concepts généraux et architecture du Cell **1**

1.1 Concepts généraux 2

1.1.1 Architecture de *Von Neumann* 2

1.1.2 Classification de *Flynn* des machines parallèles 3

1.2 Architectures mémoire des machines parallèles 4

1.2.1 Les machines parallèles à mémoire partagée 4

1.2.2 Les Machines parallèles à mémoire distribuée 6

1.2.3 Les Machines parallèles à mémoire hybride 6

1.3 Modèles de programmation parallèle 7

1.3.1 Le Modèle *shared memory* 7

1.3.2 Le Modèle de programmation par *threads* 8

1.3.3 Le Modèle de programmation par passage de message 9

1.3.4 Le Modèle *data parallel* 9

1.3.5 Modèle de programmation de calcul par flux 10

1.4 Parallélisation 10

1.4.1 Parallélisation manuelle 11

1.4.2 Parallélisation automatique 11

1.4.3 Méthodologie de parallélisation manuelle 12

1.4.4 Dépendances de données 14

1.4.5 Equilibrage de charge 15

1.4.6 Granularité . . . 15
1.4.7 Limites et coût de la parallélisation . . . 16
1.5 Architecture du processeur Cell . . . 19
1.5.1 Vue générale . . . 20
1.5.2 Le PPE : Power Processor Element . . . 21
1.5.3 Les SPE (Synergistic Processing Element) . . . 22
1.5.4 Architecture de communication . . . 24
1.5.5 Contrôleur de flot mémoire (*Memory Flow Controler*) . . . 26
1.5.6 Flot d'exécution DMA . . . 29
1.5.7 Programmabilité et modèles de programmation . . . 31
1.5.8 Environnement de développement de base . . . 34
1.6 Conclusion . . . 46

2 Parallélisation de la détection de coins d'Harris 49
2.1 Algorithme de Harris . . . 50
2.1.1 Description de l'algorithme . . . 51
2.1.2 Détails de l'implémentation . . . 52
2.2 Exploitation du parallélisme et optimisations multi-niveau . . . 54
2.2.1 Techniques spécifiques au traitement d'images . . . 55
2.2.2 Techniques spécifiques à l'architecture du processeur Cell . . . 61
2.2.3 Schémas de parallélisation . . . 71
2.3 Évaluation des performances . . . 77
2.3.1 Métriques de mesure . . . 78
2.3.2 Méthode et plateforme de mesure . . . 78
2.3.3 Comparaison des schémas de parallélisation . . . 79
2.3.4 Influence de la taille de la tuile . . . 80
2.3.5 Analyse des résultats . . . 80
2.3.6 Mesure des métriques de passage à l'échelle . . . 82
2.4 Conclusion . . . 84

3 Outils de programmation haut-niveau pour le Cell 87
3.1 ***RapidMind*** . . . 87

3.1.1 Modèle de programmation et interface . . . 88
3.1.2 Evaluation partielle et algèbre du programme . . . 90
3.1.3 Les opérations collectives . . . 93
3.1.4 Spécificité du backend pour le Cell de ***RapidMind*** . . . 94
3.1.5 Conclusion sur ***RapidMind*** . . . 94
3.2 ***OpenMP*** pour le Cell . . . 94
3.2.1 *threads* et synchronisation . . . 96
3.2.2 Génération de code . . . 97
3.2.3 Modèle mémoire . . . 99
3.2.4 Conclusion sur ***OpenMP*** pour le Cell . . . 100
3.3 Le langage de programmation ***CellSS*** . . . 100
3.3.1 *Runtime* . . . 104
3.3.2 Conclusion sur ***CellSS*** . . . 107
3.4 Le langage de programmation ***Sequoia*** . . . 107
3.4.1 Mémoire hiérarchique . . . 108
3.4.2 Modèle de programmation : les tâches . . . 110
3.4.3 Implémentation du compilateur ***Sequoia*** . . . 116
3.4.4 Conclusion sur ***Sequoia*** . . . 117
3.5 Conclusion . . . 118

4 Squelettes algorithmiques pour le Cell **121**
4.1 Programmation parallèle par squelettes algorithmiques . . . 122
4.1.1 Squelettes dédiés au parallélisme de contrôle . . . 125
4.1.2 Squelettes dédiés au parallélisme de données . . . 126
4.1.3 Squelettes dédiés à la structuration de l'application . . . 129
4.1.4 Le squelette Sequence . . . 129
4.1.5 Le squelette Select . . . 129
4.2 Classification des implémentations de squelettes algorithmiques . . . 129
4.2.1 La coordination . . . 130
4.2.2 La programmation fonctionnelle . . . 130
4.2.3 L'approche orientée objet . . . 131
4.2.4 L'approche impérative . . . 131

4.3 Squelettes implantés sur le Cell . . . 131
4.4 Modèle de programmation SKELL BE . . . 132
4.5 Détails de l'implémentation . . . 135
4.6 Génération de code pour les SPEs . . . 137
4.7 Communications PPE/SPEs . . . 141
4.8 Résultats expérimentaux . . . 143
4.8.1 Benchmarks synthétiques . . . 143
4.8.2 Benchmarks de passage à l'échelle . . . 145
4.8.3 Algorithme de Harris . . . 147
4.8.4 Impact sur la taille de l'exécutable . . . 150
4.8.5 Impact sur le temps de compilation . . . 151
4.8.6 Gain en expressivité . . . 151
4.9 Conclusion . . . 157

5 Comparaison avec les autres architectures parallèles 159
5.1 Les processeurs multi-coeurs . . . 159
5.2 Les GPU Nvidia et CUDA . . . 160
5.3 Comparaison des architectures matérielles . . . 161
5.4 Comparaison des modèles de programmation . . . 163
5.4.1 Mise en oeuvre du code parallèle . . . 163
5.5 Architectures matérielles et environnement de développement . . . 167
5.6 Mesure de performance temporelle . . . 168
5.7 Mesure d'efficacité énèrgétique . . . 169
5.8 Conclusion . . . 171

Conclusion générale et perspectives 173

Table des figures

1.1 Architecture de *Von Neumann* 2
1.2 Machine parallèle à mémoire partagée UMA 5
1.3 Machine parallèle à mémoire partagée NUMA 5
1.4 Machine parallèle à mémoire distribuée 6
1.5 Machine parallèle à mémoire hybride 7
1.6 (a) partitionnement de domaine : parallélisme de données (b) partitionnement fonctionnel : parallélisme de tâches 12
1.7 Vue d'ensemble de l'architecture hétérogène du processeur Cell 20
1.8 Réseau d'interconnexion du Cell 25
1.9 Processus *dual source* de génération de code exécutable pour le Cell 45

2.1 Illustration de la détection de points d'intérêts sur une image niveaux de gris 512×512 avec $k = 0$ 53
2.2 Implémentation de l'algorithme de Harris sous forme de graphe flot de données 53
2.3 Exemples de convolution par un filtre Gaussien 3×3 : (a) version avec noyaux 2D et (b) version avec deux noyaux 1D, résultant de la séparation du noyau 2D. 56
2.4 Recouvrement de la fenêtre du filtre gaussien 3×3, certaines données sont conservées en décalant le masque de convolution. 58
2.5 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 60

2.6 Influence de la taille du transfert DMA sur la bande-passante 64
2.7 Influence du nombre de transferts dans le cas d'une communication entre deux mémoires locales de deux différents SPEs 65
2.8 Influence du nombre de transferts dans le cas d'une communication entre la mémoire externe et une mémoire locale de SPE 66
2.9 Redondances de données pour un opérateur de convolution 3×3 68
2.10 Tracé de la fonction $Q(h, w)$ dans l'espace 70
2.11 Schéma de parallélisation SPMD conventionnel 73
2.12 Schéma de parallélisation pipeline 74
2.13 Schéma de parallélisation chaînage d'opérateurs par paires (*Halfchain*) 75
2.14 Schéma de parallélisation chaînage et fusion d'opérateurs par paires (*Halfchain+Halfpipe*) 76
2.15 Schéma de parallélisation chaînage et fusion d'opérateurs par paires (*Halfchain + Fullpipe*) 77
2.16 Comparaison des modèles en cycles par point : SPMD correspond à la version data parallel, PIPELINE au pipeline, HCHAIN+HPIPE au chainage d'opérateurs par paires et HCHAIN+HPIPE au chaînage complet et fusion d'opérateurs par paires 79
2.17 Influence de la taille de la tuile sur la performance pour la version chainage entier et fusion d'opérateurs par paires 1 SPE (Fig. 2.15) 81

3.1 Illustration d'une opération collective de réduction, ici une opération de sommation est effectuée, le résultat final est l'accumulation des éléments du tableau 93
3.2 Procédure de génération de code de ***CellSS*** 101
3.3 Multiplication de matrices de tailles 1024x1024 structurée en hiérarchie de tâches indépendantes effectuant des multiplications sur des blocs de données plus petits 108
3.4 Modèle abstrait ***Sequoia*** du processeur Cell 110

4.1 Exemple de graphe de processus communicants avec hiérarchisation 123
4.2 Arbre représentant le squelette en Figure 4.1 124
4.3 Exemple de squelette du type **Pipeline** 125

4.4 Exemple de squelette du type **Pardo** 126
4.5 Exemple de squelette du type **Farm** 127
4.6 Exemple de squelette du type **SCM** 128
4.7 Surcoût en % du squelette PARDO en fonction du nombre de SPEs, pour différentes valeurs de durée de fonction en ms 144
4.8 Surcoût en % du squelette PIPE en fonction du nombre de SPEs, pour différentes valeurs de durée de fonction en ms 145
4.9 Squelette de la version complètement chaînée 148
4.10 Squelette de la versions chaînée à moitié 149
4.11 Squelette de la version non chaînée 150

Liste des tableaux

1.1 Classification de *Flynn* des machines parallèles 3

2.1 Réduction de la complexité arithmétique par séparabilité des noyaux 57
2.2 Réduction du nombre d'accès mémoire par décomposition des noyaux 57
2.3 Réduction de la complexité mémoire par chevauchement des noyaux 57
2.4 Réduction de la complexité arithmétique par séparabilité et chevauchement des noyaux 59
2.5 Réduction de la complexité mémoire par séparabilité et chevauchement des noyaux 59
2.6 Tableau récapitulatif de l'optimisation des accès mémoire 62
2.7 Tableau récapitulatif de l'optimisation des accès mémoire 63
2.8 Performances sur une image 512×512 71
2.9 Performances sur une image 2048×512 72
2.10 Performances sur une image 1200×1200 72
2.11 Performances sur une image 2048×2048 72
2.12 Apport de la composition de fonction à la performance, les versions de référence sont celles ayant une accélération de 1 81
2.13 Mesure des métriques de passage à l'échelle pour la version FCHAIN+HPIPE . 82
2.14 Mesure des métriques de passage à l'échelle pour la version SPMD 83

4.1 Interface utilisateur `SKELL BE` 136
4.2 Résultat du benchmark du produit scalaire `DOT` 146
4.3 Résultat du benchmark de la convolution 3x3 `CONVO` 147

4.4 Résultat du benchmark de la multiplication matrice vecteur `SGEMV` 147
4.5 Benchmarks de l'algorithme de Harris 150
4.6 Impact sur la taille de l'exécutable 151

5.1 Comparaison des niveaux de parallélisme et de la hiérarchie mémoire des architectures matérielles 162
5.2 OpenMP et les architectures parallèles 164
5.3 les Pthreads et les architectures parallèles 165
5.4 CUDA et les architectures Nvidia 166
5.5 Liste des architectures matérielles évaluées pour la comparaison de performances 167
5.6 Comparaison des implémentations de l'algorithme de Harris sur des architectures parallèles 170
5.7 Comparaison des architectures en énergie consommée pour des images 512×512 171
5.8 Comparaison des architectures en énergie consommée pour des images 300×300 172

Introduction

La grande majorité des classes d'applications ont connu une amélioration régulière et gratuite de leurs performances durant les dernières décennies. Les fabricants de processeurs, de mémoires et de systèmes de stockage ont amélioré leurs systèmes d'une manière sensible, à tel point que les logiciels montraient des améliorations de performances sans même que leur code source ne soit réadapté. Nous avons vu la fréquence des processeurs passer de 500 MHz à 1 GHz puis à 2 GHz pour atteindre 3 GHz de nos jours.
La question qui se pose alors est celle de la limite de l'amélioration des processeurs par l'augmentation de leur fréquence d'horloge. Alors que la loi de *Moore* [76] prédit une augmentation exponentielle, les dernières années ont montré que nous approchions la saturation. Par conséquent, cette augmentation de fréquence doit éventuellement ralentir voire même s'arrêter.
Pour les développeurs, l'augmentation de la fréquence d'horloge des processeurs fut souvent synonyme d'amélioration des programmes sans en modifier la moindre ligne de code source. Hélas, cette affirmation n'est plus valable à l'heure où l'on parle, et elle ne le sera pas non plus dans le futur proche.
Les architectures vont évidemment continuer à évoluer et à améliorer leurs performances, mais l'amélioration des logiciels ne se fera plus de manière aussi systématique que par le passé. En effet, les évolutions architecturales ont changé de direction et s'orientent désormais vers une multiplication des unités de calcul et une coordination plus efficace de leur exécution concurrente pour améliorer les performances des programmes. Durant les 30 dernières années, les concepteurs de processeurs ont pu améliorer les performances des architectures dans trois domaines :

– **La fréquence d'horloge :** cela consiste à réduire la durée du cycle d'horloge CPU et se traduit directement par l'exécution plus rapide d'un même programme.

– **L'optimisation de l'exécution :** cela se traduit par l'accomplissement de plus de travail en un cycle d'horloge. Parmi les techniques les plus connues : le *pipelining*, la prédiction de branchements, l'exécution de plusieurs instructions dans un même cycle (processeurs superscalaires) et le réordonnancement des instructions pour une exécution dans le désordre. Ces techniques ont pour but d'améliorer le flot d'exécution des instructions par la réduction de la latence et la maximisation du travail accompli en un cycle d'horloge
– **La mémoire cache :** L'augmentation de la mémoire cache sur puce a pour but de réduire les accès à la mémoire vive (RAM). En effet, les accès à la mémoire principale se font toujours avec des latences très grandes, et rapprocher les données des unités de calcul demeure une excellente technique pour améliorer les performances. Les tailles des caches ont augmenté sensiblement durant les dernières années pour atteindre des valeurs autour de 10 Mo sur les dernières générations de processeurs.

Les techniques citées dans ce qui précède ont toutes pour point commun d'être efficaces quelque soit la nature de l'application. Une amélioration induite par une de ces techniques induit automatiquement une accélération aussi bien des application séquentielles que des programmes parallèles. Il est important de le noter car la majorité des applications existantes sont séquentielles pour des raisons historiques. La seule technique qui exploite le parallélisme de données SIMD et qui se trouve sous forme d'extensions du jeu d'instructions (MMX, SSE, Altivec) est souvent gérée par les compilateurs, on parle alors de vectorisation automatique (*auto-vectorization*).

La pente de la courbe d'augmentation de la fréquence d'horloge CPU a commencé à s'aplatir en 2003. Il est devenu très difficile d'augmenter la fréquence d'horloge à cause des limitations physiques suivantes :

– La chaleur dégagée est devenue importante et très difficile à dissiper.
– La consommation d'énergie n'a cessé d'augmenter.
– Des problèmes de fuites de courants ont fait leur apparition.

Cela ne signifie pas que la loi de *Moore* arrive à son terme. Le nombre de transistors sur une puce continuera d'augmenter suivant cette loi, mais la fréquence d'horloge ne suivra pas la même tendance. La différence majeure réside dans le fait que l'amélioration des performances

des processeurs se fera d'une manière radicalement différente dans les prochaines années. Ainsi, la plupart des applications actuelles ne pourront plus bénéficier de l'amélioration gratuite des performances, qui était jusque là possible. Les directions futures comprennent une seule des anciennes approches qui est l'augmentation de la taille de la mémoire cache. Pour le reste, deux nouvelles techniques seront utilisées :

- ***L'hyperthreading*** : cette technique consiste à exécuter deux ou plusieurs *threads* sur un même CPU. Cela permet à certaines instructions de s'exécuter en parallèle. Malgré la présence de registres supplémentaires, ce type d'architectures ne possède q'un seul cache et qu'un seul couple ALU/FPU. On attribue à l'*hyperthreading* un potentiel d'amélioration des performances situé entre 5 et 15% pour une application contenant un proportion raisonnable de *multithreading* et jusqu'à 40% pour une application exploitant pleinement ce type de parallélisme.
- ***Le multicore*** : C'est l'intégration de deux ou plusieurs CPUs sur la même puce. Cette technique permet dans un cas idéal (mais pas réaliste) de diviser le temps d'exécution de l'application par un facteur égal au nombres de CPUs, à condition que l'application utilise plusieurs *threads*.

Il est certain que les architectures continueront à évoluer, mais l'amélioration de la performance des applications ne se fera pas de manière exponentielle sans exploiter le parallélisme. Le pas à franchir n'est pas facile, car toutes les applications ne sont pas naturellement parallélisables et la programmation parallèle est difficile.
La programmation parallèle existe depuis longtemps, mais elle était jusque là réservée à une poignée de développeurs pour qui les architectures classiques n'apportaient pas de résultats satisfaisants à des problèmes complexes et nécessitant une grande puissance de calcul.
Le parallélisme est la nouvelle révolution dans la manière avec laquelle nous écrivons les logiciels. Ce changement est aussi important que l'avènement de la programmation orientée objet dans les années 1990 notamment en termes de courbe d'apprentissage.

La démocratisation des architectures parallèles et massivement parallèles que l'on connait aujourd'hui a ouvert de nouvelles perspectives pour le développement et le déploiement d'applications de plus en plus complexes. Nous avons connu durant les cinq dernières années une émergence des architectures à plusieurs unités de calcul. Des processeurs jusque là réservés à des machines de calculs dédiées aux calculateurs des grands laboratoires scientifiques se

retrouvent de nos jours dans des stations de travail de particuliers et même dans des ordinateurs portables voire embarqués. Plusieurs types d'architectures parallèles ont alors vu le jours : les plus répandues sont les architectures multi-coeurs à mémoire partagée du type *SMP* (*Symmetric Multi-Processor*), il équipent la grande majorité des ordinateur de bureau du marché. L'autre catégorie de processeurs parallèles ayant vu le jour à la même période sont les architectures hétérogènes : le processeur Cell étant l'exemple le plus connu de ce type de machines. Ces deux dernières catégories sont plutôt classiques dans leur genre car elles existaient auparavant mais étaient jusque là réservées aux grands calculateurs pour les *SMP* et aux systèmes embarquées pour les architectures parallèles hétérogènes. L'autre type d'architectures ayant émergé peu après sont les GPU (*Graphics Processing Unit*). Là encore, ce sont des architectures qui existaient auparavant mais leur usage était uniquement réservé au rendu graphique. Les constructeurs on apporté des modifications architecturales et ils ont fourni de nouvelles interfaces de programmation : Nvidia CUDA[79] et plus récemment OpenCL[64], afin de permettre d'utiliser les architectures graphiques comme des machines de calcul généraliste.
Les architectures parallèles sont devenues accessibles aux plus grand nombre et n'ont cessé de s'améliorer en performances. De nouvelles perspectives se sont ouvertes alors pour les applications et leurs concepteurs. Le traitement d'images est l'un des domaines qui requiert une grande puissance de calcul et qui se prête bien à la parallélisation. Cependant, plusieurs obstacles peuvent constituer un frein à la mise en oeuvre des programmes sur ces nouvelles architectures parallèles.

Les spécificités architecturales

Même si des points communs existent entre elles, les architectures matérielles diffèrent entres elles selon plusieurs aspects. Par exemple, une architecture du type CPU multi-coeurs contient en général un nombre limité de processeurs capables d'exécuter des tâches avec un flot de contrôle complexe et peuvent également exécuter des applications où le flot de données est important. D'un autre côté , les architectures du type GPU, contiennent plusieurs centaines de coeurs (on parle alors de *many core*) dédiés aux problèmes massivement parallèles (*embarassingly parallel problem*) et sont donc plus appropriées pour le parallélisme de données. De plus, entre deux architectures d'une même famille, les GPU par exemple, subsistent

des différences dans les jeux d'instructions, l'organisation de la mémoire et la conception des unités de calcul, ce qui empêche la portabilité de la performance.

Les spécificités liés aux outils de développement

Les efforts de normalisation contribuent à la portabilité du code source entre architectures parallèles et OpenCL[64] (à l'initiative d'Apple) en est l'exemple le plus récent. Toutefois, chaque architecture possède en général un outil de programmation propriétaire différent de celui des ces concurrentes. De plus, chaque modèle de programmation est adapté à un type d'architecture donné. Par conséquent, la performance n'est pas toujours au rendez-vous et dépend d'une adéquation entre l'architecture matérielle, le modèle de programmation et l'implémentation de celui-ci.

Objectifs des travaux

Au vu des contraintes citées auparavant, l'objectif de nos travaux est à la fois d'établir des méthodes génériques d'optimisation de code pour le processeur Cell, et de proposer un modèle de programmation implanté sous forme d'une bibliothèque logicielle pour l'aide à la parallélisation de code. Nos travaux visent le traitement d'images bas-niveaux mais peuvent aisément s'étendre à d'autres domaines ayant des structures de données et des algorithmes similaires.

Le manuscrit relatant ces travaux est organisé comme suit :

– **Chapitre 1, Concepts généraux et architecture du Cell**. Les concepts généraux du calcul parallèle sont donnés dans ce chapitre. Les architectures parallèles y sont classées selon le type de parallélisme qu'elles exploitent et selon leur hiérarchie mémoire. Les principaux modèles de programmation y sont exposés et la méthodologie de parallélisation y est détaillée. Les problèmes et les limitations liés à la parallélisation sont également traités. Nous présentons également dans ce chapitre, une vue d'ensemble de l'architecture principale étudiée lors de nos travaux : le Cell Broadband Engine. Les principaux composants ainsi que les différents niveaux de parallélisme de l'architecture y sont décrits. L'environnement de développement de base fourni par IBM est également décrit avec une vue globale de l'API et des contraintes de programmation spécifiques à

l'architecture du Cell.

– **Chapitre 2, Parallélisation de la détection de coins d'Harris**. Nous présentons une étude approfondie de l'algorithme de détection de points d'intérêt de Harris. Cet algorithme représentatif des applications de traitement d'images bas-niveau nous a servi de cas d'étude pour l'analyse de schémas de parallélisation et des techniques d'optimisation sur notre architecture cible. Plusieurs paramètres algorithmiques et de l'architecture sont ainsi étudiés afin de déterminer la méthodologie de portage d'applications sur le processeur Cell. Le but de ces travaux est à la fois de permettre au développeur de tirer partie des dispositifs de l'architecture et de permettre aux concepteurs de compilateurs et d'outils de parallélisation d'avoir un socle sur lequel ils peuvent se reposer pour développer des processus automatiques d'optimisation.
– **Chapitre 3, Outils de programmation haut-niveau pour le Cell**. Certains outils de programmation développés à la fois dans le milieu académique et industriel sont présentés. Les modèles de programmation associés aux outils sont étudiés. Le but de cette analyse est également de comparer les approches pour en extraire les avantages et les inconvénients en ce qui concerne l'aide à la parallélisation de code sur l'architecture du processeur Cell.
– **Chapitre 4, Squelettes algorithmiques pour le Cell**. Nous présentons dans ce chapitre, la bibliothèque `SKELL_BE` à base de squelettes algorithmiques. Cet outil de développement issu d'un travail collaboratif entre Joel Falcou, Lionel Lacassagne et moi même, propose un générateur de code parallèle ayant pour entrée une description haut-niveau de l'application. Cette bibliothèque reprend des schémas de parallélisation communs qui servent de composants de base pour la construction d'une application plus complexe. Une analyse du surcoût de mise en oeuvre ainsi que des performances obtenues à l'aide du modèle sont également présentées.
– **Chapitre 5, Comparaison avec les autres architectures parallèles**. Nous proposons une étude comparative des architectures parallèles émergentes concurrentes du processeur Cell. L'algorithme de référence est celui étudié dans le second chapitre. Les architectures de type SMP multi-coeurs et les architectures de type GPU sont abordées. Plusieurs critères sont mis en avant pour la comparaison, la programmabilité, les performances pures, ainsi que l'efficacité énergétique.

– **Conclusion générale et perspectives** Nous résumons les apports de nos travaux dans ce chapitre et donnons les perspectives des outils développés ainsi que les évolutions dans le domaine de la parallélisation de code connues à ce jour.

Chapitre 1

Concepts généraux et architecture du Cell

Les logiciels ont été conçus historiquement pour une exécution séquentielle. 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*s. 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 bons candidats à la parallélisation. Parmi ces caractéristiques : la 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 démocratisé 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, la prospection pétrolière, les moteurs de recherche, la modélisation financière, les technologies de diffusion multimédia et les applications graphiques et de réalité augmentée.

1.1 Concepts généraux

1.1.1 Architecture de *Von Neumann*

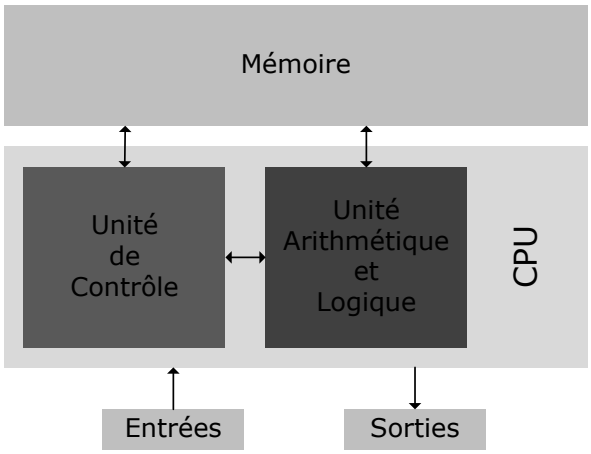

FIGURE 1.1 – Architecture de *Von Neumann*

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 [99]. A partir de ce moment, la majorité des ordinateurs ont été conçus sur ces bases. L'architecture *von Neumann* (Fig. 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 I/O font l'interface avec l'utilisateur humain.

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

TABLE 1.1 – Classification de *Flynn* des machines parallèles

1.1.2 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* [41] (*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*. Le tableau 1.1 illustre la classification de *Flynn*.

Single instruction, single data (SISD)

Une machine séquentille qui ne peut exécuter qu'un seul flux d'instructions 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 adapté pour des problèmes réguliers tels que le traitement d'images et le rendu graphique. L'exécution des programmes 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 indépendantes. Hennessy et Patterson dans [49] affirment qu'il n'y a pas de machines de ce type qui ait été conçue, alors que Flynn dans son article de 1996 [42] classe les architecture systoliques [68] dans cette catégorie.

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 processeurs multi-core. De plus, plusieurs de ces machines contiennent des unités de traitement SIMD.

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.

1.2.1 Les machines parallèles à mémoire partagée

Il existe plusieurs variantes de ces machines mais toutes partagent une propriété commune qui est la possibilité à tous les processeurs d'accéder à 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 machines peut être divisée en deux sous-classes basées sur les temps d'accès à la mémoire : UMA et NUMA.

Uniform memory access (UMA)

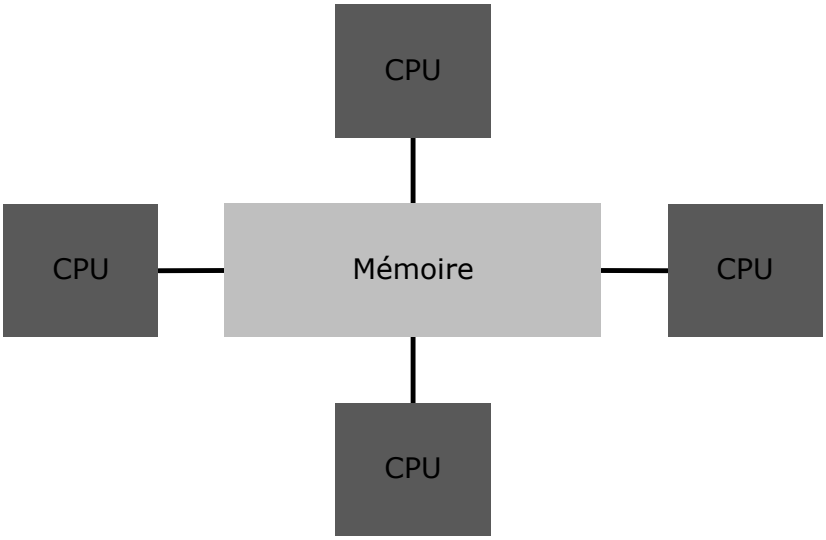

FIGURE 1.2 – Machine parallèle à mémoire partagée UMA

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 matériel.

Non-uniform memory access (NUMA)

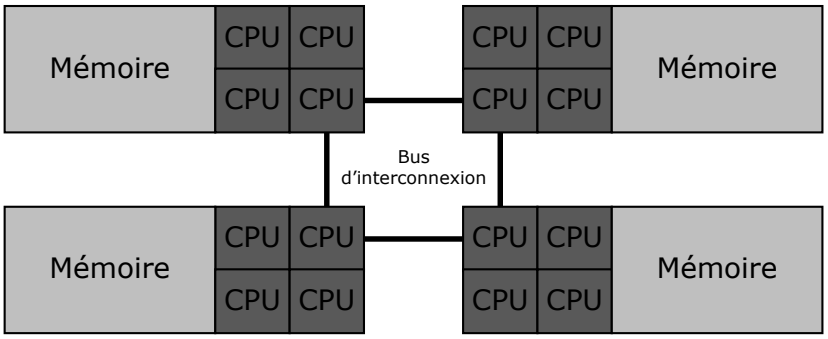

FIGURE 1.3 – Machine parallèle à mémoire partagée NUMA

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.

1.2.2 Les Machines parallèles à mémoire distribuée

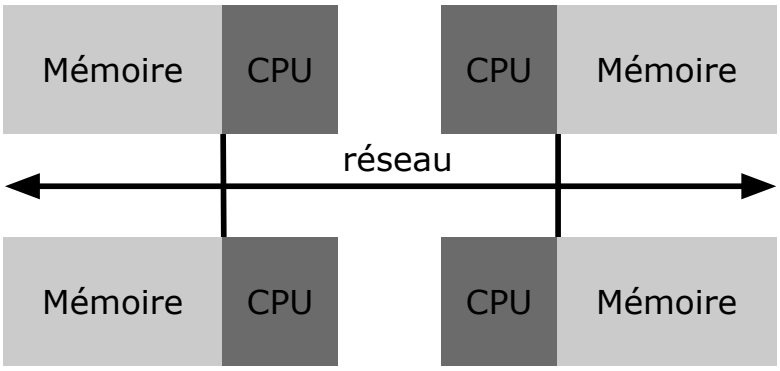

FIGURE 1.4 – Machine parallèle à mémoire distribuée

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 les mémoires des processeurs. Les différents processeurs possèdent leur propre mémoire locale. Les adresses mémoire d'un processeur donné 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 à 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.

1.2.3 Les Machines parallèles 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

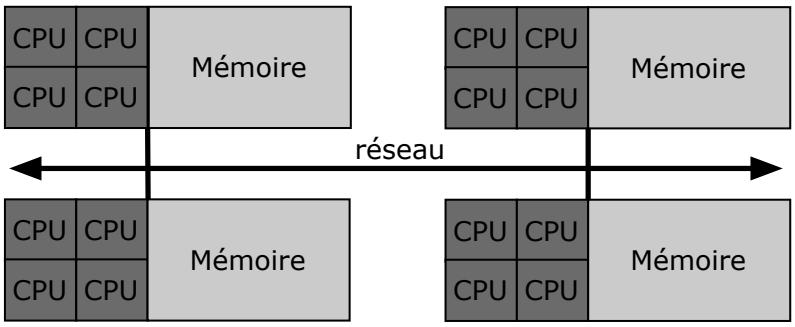

FIGURE 1.5 – Machine parallèle à mémoire hybride

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. La différence majeure entre ce type d'architectures et les SMP NUMA, est le fait que l'espace mémoire n'est pas partagé et que la communication inter-processeur passe par un réseau d'interconnexion de type *Ethernet* ou *Infiniband*.

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 donnée sur une machine donnée. Dans la suite nous décrivons les principaux modèles de programmation parallèles.

1.3.1 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 communications

explicites pour transférer des données d'une tâche à une autre. Toutefois, en terme de performances ce dernier point constitue un inconvénient car il engendre un surcout d'accès à la mémoire, de rafraichissement 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 partagée se résument au compilateur natif qui traduit les variables du programme en adresses mémoire globales. Il n'existe cependant pas d'implémentation de ce modèle sur des machines à mémoire distribuée.

1.3.2 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 concept à un programme principal qui inclue un certain nombre de sous-routines. Le programme principal est ordonnancé pour être exécuté par le système d'exploitation, et il acquière toutes les ressources système nécessaires à son exécution. Il effectue alors un ensemble d'instructions en série et crée un certain nombre 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 crées et détruits tout au long du déroulement du programme. Le modèle de programmation par *thread* est souvent associé aux machines à mémoire partagée. Les implémentations des *threads* comportent en général une bibliothèque de fonctions ou alors une série de directives enfouis 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 à donné naissance à deux implémentations qui sont devenues le standard de nos jours : Les *threads* POSIX [54] et OpenMP [23].

1.3.3 Le Modèle de programmation par passage de message

Dans ce modèle, la programmation parallèle se fait par passage de messages. Un ensemble de tâches 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 duale *receive*. Les implémentations du *Message Passing* prennent la forme d'une bibliothèque de sous-routines et le programmeur est responsable de la détection du parallélisme. Comme pour toute bibliothèque, 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* parmi lesquelles PVM [92] et Parmacs [15]. Deux normes ont alors vu le jour : MPI [43] 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 réseau sont tout simplement remplacées par des copies mémoire.

1.3.4 Le Modèle *data parallel*

Ce modèle est basé sur le parallélisme de données qui concentre le travail en parallèle sur un ensemble de données contenues dans un tableau ou dans une structure de données à plusieurs dimensions. Un ensemble de tâches travaillent collectivement sur la même structure de données mais chaque tâches 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 bibliothèque ou à des directives reconnues par un compilateur *data parallel*. Les implémentation de ce modèle sont souvent sous forme de compilateurs ou d'extensions de ces derniers. On peut citer les compilateur Fortran (F90 et F95) et leur extension HPF (*High Performance Fortran*)

[85] qui supportent la programmation *data parallel*. *HPF* inclut des directives qui contrôlent la distribution des données, des assertions qui peuvent améliorer l'optimisation du code généré ainsi que des construction *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 ainsi les données sur les différents processeurs et tout cela de manière transparente du point de vue de l'utilisateur.
Malgré un forte popularité à ses débuts, HPF n'a pas connu le succès escompté comme le prouve l'analyse de son principal auteur dans [63].

1.3.5 Modèle de programmation de calcul par flux

Ce modèle appelé communément *stream computing* est un modèle basé sur le parallélisme de données. Un même noyau de calcul *kernel* est appliqué à un ensemble de données. Ce modèle est le modèle dominant sur les unités de calcul graphique. C'est un modèle ou le parallélisme est du type SIMD, ou plusieurs unités de calcul (typiquement des centaines) exécutent la même instruction sur un ensemble de données en parallèle. Les machines supportant ce type de modèle sont les GPU, les FPGA et certains processeurs spécialisés tels que le Imagine [59] et Merrimac [24] de Stanford. Plusieurs langages ont été également développés afin de supporter ce type de matériel parmi lesquelles : StreamIt [94] et Brook [14]. CUDA [79] et OpenCL [64] sont également des implémentations de ce modèle de programmation parallèle et sont de loin les plus utilisés de nos jours. Le modèle consiste à simplifier à la fois le matériel et à restreindre le type de parallélisme exploité.

1.4 Parallélisation

Les architectures parallèles et les modèles de programmation associés étant définis. La question qui se pose alors est celle du choix à la fois de l'architecture et du modèle de programmation adéquats pour la mise en oeuvre d'une application parallèle donnée. L'efficacité des outils automatiques de parallélisation dépend souvent de plusieurs facteurs, parmi lesquelles : les caractéristiques de l'architecture matérielle et de la hiérarchie mémoire ainsi que la nature des algorithmes qui forment l'application à paralléliser. L'utilisation d'outils automatiques n'est pas toujours efficace, il est alors parfois nécessaire de gérer manuellement le

parallélisme et les optimisations qui lui sont associées. Deux choix se présentent alors : la parallélisation manuelle ou la parallélisation automatique.

1.4.1 Parallélisation manuelle

Elle permet un contrôle précis de la performance, et une grande flexibilité en termes de schéma de parallélisation possible (différents modèles de calcul). Par contre, les temps de développement sont importants, que cela soit pour la mise en oeuvre, le débogage ou la maintenance de l'application. Les erreurs sont parfois très difficiles à trouver, et le processus d'optimisation est souvent itératif.

1.4.2 Parallélisation automatique

L'automatisation du processus de parallélisation de code est un problème ouvert, et les efforts effectués en la matière sont de plus en plus nombreux, en particulier avec l'avènement des nouvelles architectures parallèles et leur démocratisation. On peut trouver deux formes d'outils automatiques de parallélisation. Certains outils sont entièrement automatiques : ils prennent en entrée un code source série et détectent automatiquement le parallélisme potentiel, ils génèrent en suite le code parallèle correspondant. D'autres outils sont semi-automatiques car l'utilisateur indique les portions de codes parallélisables, c'est le cas par exemple d'OpenMP via les directives de compilation. L'avantage des approches automatiques est avant tout la rapidité de mise en oeuvre d'une solution à base de calcul parallèle, d'autant plus que dans la majorité des cas, le code original est directement utilisable. Par contre, le contrôle est beaucoup moins précis qu'avec une version entièrement manuelle. Il peut y avoir par exemple des écarts entre les résultats numériques notamment sur les arrondis pour les calculs en virgule flottante. De plus, les modèles de programmation dans ce cas ne permettent pas une grande flexibilité dans le choix des schémas de parallélisation. Dans certains cas le gain de performance peut être médiocre, et on peut même observer une baisse de performances par rapport à la version originale. Enfin, ce genre d'outils n'est généralement efficace que sur des portions de code facilement exploitables comme les boucles. Dans la suite nous allons présenter les différentes étapes de mise en oeuvre d'un code parallèle manuellement, les étapes en question vont de la détermination de l'opportunité de parallélisation jusqu'à la mise en oeuvre et l'évaluation du gain ainsi obtenu.

1.4.3 Méthodologie de parallélisation manuelle

Comprendre le problème

Avant même de commencer à développer la version parallèle d'une application, la première question qui se pose et celle de la faisabilité d'une telle solution. En effet, il existe certains problèmes dans lesquels il n'existe aucune forme de parallélisme exploitable. Une fois la faisabilité validée, on doit identifier les portions de code qui prennent le plus de temps dans l'application (les points-chaud de l'application : *hotspots*). Les outils de profilage et d'analyse des performances sont très utiles pour déterminer ses portions de code critiques. Il est nécessaire ensuite, de détecter les goulots d'étranglement (*bottlenecks*) qui limitent la performance de l'application : les entrés/sorties sont un bon exemple de *bottlenecks*. La bande passante limite la performance d'une application consommant beaucoup d'entrées/sorties. Enfin dans certains cas, il peut s'avérer nécessaire de changer l'algorithme de calcul pour qu'il puisse bénéficier du parallélisme d'une architecture.

Partitionner le problème

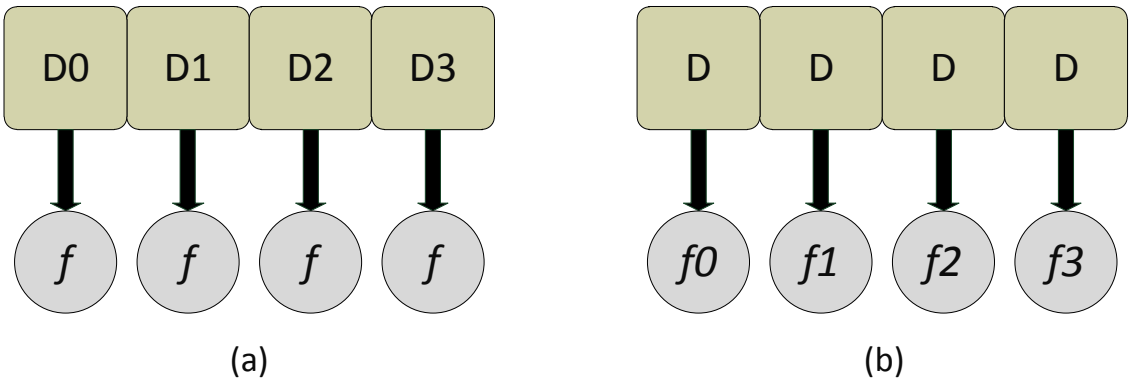

FIGURE 1.6 – (a) partitionnement de domaine : parallélisme de données (b) partitionnement fonctionnel : parallélisme de tâches

La deuxième étape de la parallélisation concerne le partitionnement du problème. Selon la nature du parallélisme contenu dans l'application : parallélisme de données ou de tâches. Il existe deux manières de décomposer le problème. La première décomposition qui exploite le parallélisme de données consiste à diviser la structure de données en plusieurs parties égales

ou pas et d'assigner à chacune des tâches une partition de données sur laquelle elle effectue des calculs. Dans ce cas précis, les tâches effectuent les mêmes opérations sur les données. La décomposition fonctionnelle est la deuxième manière de partitionner le problème, le parallélisme de tâches est alors exploité. Les tâches exécutent des portions de code différentes sur les mêmes données. La figure 1.6 illustre les deux décompositions. D désigne un même bloc de données. D0 à D3 désignent des blocs de données résultant d'une décomposition. f est une même fonction alors que $f0$ à $f3$ désignent une décomposition d'une fonction en plusieurs fonctions pouvant s'exécuter en parallèle.

Gestion des communications

Les communications sont souvent problématiques en programmation parallèle. En effet, le fait de décomposer le problème engendre parfois un besoin de communication entre les tâches. La première tâche consiste alors à déterminer si les tâches ont besoin de communiquer ou pas, ceci étant généralement déterminé par l'algorithme. Si une communication est nécessaire, il faut alors évaluer les facteurs qui influencent la vitesse des transferts qui sont, la latence et le débit. Ce dernier peut être altéré lors de situations dans lesquelles plusieurs transferts concurrents se partagent le bus de données. Lors d'une communication, il est nécessaire d'effectuer des points de synchronisation pour garantir la validité des traitements. Il faut alors évaluer le coût des opérations de synchronisation. On est souvent confrontés à des choix de conception lorsqu'il s'agit de communications entre les tâches. La multiplicité des transferts engendre autant de temps de latence que de transfert. Il est alors souvent utile de regrouper les transferts en un seul bloc, ce qui n'est pas tout le temps possible, car la largeur des bus et la capacité des mémoires sont limitées.

Visibilité des communications

Les communications sont explicites ou implicites selon le modèle de programmation. En MPI par exemple, l'utilisateur contrôle finement les communications et détermine lui-même leur déroulement, les tailles des transferts et les points de synchronisation. Par contre dans le modèle *data parallel*, les communications sont transparentes du point de vue de l'utilisateur et ne sont donc pas directement sous le contrôle du programmeur et tous ces aspects sont gérés automatiquement.

Communications synchrones vs asynchrones

Les communications synchrones sont bloquantes i.e : l'exécution du programme est suspendue jusqu'à la fin de la transaction. Elles peuvent limiter la performance car elles augmentent les durées d'inactivité des processeurs. A l'opposé, les communications asynchrones permettent l'entrelacement de tâches de calcul et de transfert, et ainsi un gain de performances potentiel lorsque l'architecture permet d'effectuer en parallèle des transferts et des calculs.

Gestion de la synchronisation

La gestion de plusieurs ressources en parallèle, engendre un besoin de synchronisation. Les tâches ont souvent besoin de se synchroniser soit pendant un échange de données, soit à la suite d'une opération collective. Parmi les opérations de synchronisation les plus utilisées on trouve les barrières, les sémaphores et les verrous.

Les Barrières

Ce type de synchronisation est utilisée pour les opérations collectives comme les réductions. Toutes les tâches effectuent leur travail et sont suspendues lorsqu'elles atteignent la barrière. Lorsque la dernière tâche atteint la barrière toutes les tâches sont synchronisées.

Les verrous et les sémaphores

Les *locks* et les sémaphores servent généralement à protéger l'accès à une variable globale ou à rendre une section de code critique i.e : une seule tâche peut alors exécuter ses instructions se trouvant dans cette portion de code. Une tâches possède alors l'accès exclusif à la ressource en effectuant un *lock()* et libère la ressource en effectuant un *unlock()*.

1.4.4 Dépendances de données

Les dépendances de données sont un des principaux inhibiteurs de la parallélisation. Une dépendance de données existe lorsqu'une modification de l'ordre d'exécution des instructions change le résultat du programme. Le concepteur de l'algorithme parallèle, doit gérer proprement les dépendances de données avec les opérations de synchronisations adéquates. Une

modification de l'algorithme peut éliminer ces dépendances et permettre ainsi une parallélisation plus efficace. L'exemple le plus parlant étant celui de l'utilisation d'une variable locale dupliquée au lieu d'une variable globale qui nécessite des synchronisations pour assurer un accès cohérent pour les tâches concurrentes.

1.4.5 Equilibrage de charge

L'équilibrage de charge ou *load balancing* est une des problématiques qui se posent lors du développement d'un code parallèle. En effet, une distribution équitable de la charge de travail est nécessaire afin de minimiser les durées d'inactivité des processeurs. Lorsque les tâches effectuent le même travail, l'équilibrage de charge est trivial : il suffit d'attribuer aux tâches les mêmes quantités de données. Si par contre les tâches exécutent un code différent, un ajustement de la charge de travail est parfois nécessaire. Il subsiste certains cas où la charge n'est pas prédictible, par exemple lors du calcul de trajectoires de particules. Il faut alors effectuer du *load balancing* dynamique.

1.4.6 Granularité

La granularité du parallélisme est définie comme étant le ratio calcul/communication. Il existe alors deux formes de granularités.

Parallélisme à grain fin *(Fine-Grain)*

Dans ce cas là, le ratio calcul/communication est faible et les opportunités d'optimisation. L'équilibrage de charge est alors simplifié puisque les tâches passent la majorité du temps en communications et pas en calcul.

Parallélisme à gros grain *(coarse-grain)*

Contrairement au parallélisme à grain fin le ratio calcul/communication est important. Les opportunités de gain de performance sont alors importantes car le calcul est prépondérant dans l'application. Ce type de granularité est idéal pour les architectures possédant plusieurs unités de traitement, limitées par la bande-passante mémoire. Par contre, l'équilibrage de charge n'est pas facile.

Choix de la granularité

Le choix de la granularité dépend de l'architecture et de l'algorithme à la fois. Le parallélisme grain-fin contribue à l'ajustement de l'équilibrage de charge.

1.4.7 Limites et coût de la parallélisation

Loi d'Amdhal

La loi d'Amdahl stipule que l'accélération d'un programme est limitée par la proportion parallélisable de celui-ci. Cela se traduit par l'équation suivante :

$$Accélération = \frac{\text{Temps d'exécution séquentielle}}{\text{Temps d'exécution parallèle}} = \frac{1}{(1 - FP) + \frac{FP}{N}} \tag{1.1}$$

Où FP est la fraction parallélisable du programme et N le nombre de processeurs.
On peut constater à partir de l'équation 1.1 que l'accélération est limitée par la proportion de code parallélisable. L'augmentation de l'accélération est quasi-linéaire avec l'augmentation de N pour une FP donnée. D'autre part, l'accélération est vite saturée et atteint $\frac{1}{1-FP}$ lorsque $N \to \infty$. Cela signifie qu'une augmentation linéaire de l'accélération n'est plus possible à partir d'une certaine valeur de N et qu'il est inutile d'ajouter des unités de traitement pour améliorer les performances. De plus, le terme $\frac{FP}{N}$ implique que la fraction parallélisable peut être réduite d'un facteur N ce qui est faux dans la réalité, car l'exécution parallèle induit un surcoût dû à plusieurs facteurs : la gestion des *thread*, la synchronisation et les communications.

Loi de *Gustafson-Barsis*

La loi de *Gustafson-Barsis*[47] vient corriger la loi d'*Amdahl* qui donne une limite à l'accélération atteignable, en considérant un problème de taille fixe. La loi est enoncée comme suit :

$$S = N + (1 - N) \times s \tag{1.2}$$

Où N est le nombre de processeurs, S l'accélération (nommée *Scaled Speedup* par *Gustafson*) et s est la portion non-parallélisable du programme. La loi de *Gustafson-Barsis* aborde un point que la loi d'*Amdahl* ne considère pas, à savoir la puissance de calcul disponible lorsque le nombre de processeurs augmente. L'idée générale de la loi consiste en l'augmentation de

la taille du problème afin d'utiliser toute la puissance disponible pour résoudre celui-ci en un temps constant. Ainsi, si une machine plus puissante (avec plus de processeurs) est disponible, des problèmes plus larges peuvent être résolus en un temps identique à celui nécessaire pour la résolution de problèmes de plus petite taille. La loi d'*Amdahl* au contraire, part du principe que la charge de travail du programme ne change pas en fonction du nombre de processeurs, ce qui correspond à un problème de taille fixe. Dans les deux lois, la portion parallélisable est supposée équitablement distribuée sur les processeurs.
La loi de *Gustafson-Barsis* a surtout permis aux chercheurs de réorienter les algorithmes afin qu'il résolvent des problèmes de plus grande taille plutôt que de se focaliser sur l'accélération d'un petit problème sur plusieurs processeurs.

Métrique de *Karp-Flatt*

La métrique de *Karp-Flatt*[61] est une mesure de la parallélisation de code sur les systèmes a processeurs parallèles. Cette métrique vient s'ajouter aux lois d'*Amdahl* et *Gustafson-Barsis* afin de donner une indication sur l'efficacité de la parallélisation d'un code sur une machine parallèle.
Selon *Karp-Flatt*, étant donné un programme parallèle ayant une accélération S sur p processeurs ($p > 1$), la portion séquentielle s déterminée de manière experimentale est donnée par :

$$s = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Ainsi, plus s est petit et plus la parallélisation du code est bonne.
Cette métrique est venue apporter une correction aux deux lois d'*Amdahl* et *Gustafson-Barsis* en déterminant de manière expérimentale la portion séquentielle du code. En effet, la loi d'*Amdahl* ne prend en compte ni les problèmes d'équilibrage de charge ni le surcoût induit par la parallélisation.
Pour un problème de taille fixe, l'efficacité d'un programme parallèle décroit lorsque le nombre de processeurs augmente. En utilisant la portion séquentielle obtenue expérimentalement, on peut déterminer si l'efficacité décroit à cause de la diminution des opportunités de parallélisation ou à cause de l'augmentation du surcoût algorithmique ou architectural.

Débogage d'un programme parallèle

Une fois que l'application a été parallélisée, le développeur fait face à un cycle classique de débogage qui inclut potentiellement de nouveaux bugs induits par l'exécution parallèle du programme. Les bugs non-liés à la parallélisation peuvent se résoudre avec les techniques classiques de débogage alors que ceux liés à l'exécution parallèle requièrent plus d'attention. On doit tout d'abord en comprendre les causes pour ensuite tenter de les résoudre. Ils sont difficiles à détecter et requièrent plus de temps et d'attention que des bugs classiques. Les plus communs sont :

La situation de compétition (*race condition*) Une *race condition* survient quand deux ou plusieurs *threads* essaient d'accéder à la même ressource en même temps. Si les *threads* ne communiquent pas de manière efficace, il est impossible de savoir quel *thread* accèdera à la ressource en premier. Cela conduit la plupart du temps à des résultat incohérents lors de l'exécution du programme. A titre d'exemple, dans un *data race* de lecture/écriture, un *thread* tente d'écrire dans une variable en même temps qu'un autre *thread* essaie d'en lire la valeur. Le *thread* qui lit la variable reçoit une valeur différente selon l'ordre relatif de l'opération de lecture par rapport à l'écriture dans la variable par l'autre *thread*. On corrige ce type d'erreurs par l'insertion d'une section critique autour de la ressource partagée. Cela restreint l'accès à un seul *thread* à la fois et assure ainsi une exécution séquentielle de la portion de code concernée. Ce type de synchronisation doit être utilisé modérément car il peut induire un surcoût important et dégrader ainsi les performances.

La famine (*ressource stagnation*) Cette situation peut survenir lors de l'utilisation de *locks*. Si un *thread* réserve une ressource et passe en suite au reste du programme sans libérer celle-ci. Quand un second *thread* essaie d'accéder à la même ressource, il se met en attente pendant un temps infini, ce qui cause une famine. Le développeur doit alors toujours s'assurer de libérer une ressource réservée avant de continuer dans le programme.

L'interblocage (*deadlock*) Un interblocage est similaire à une famine mais il survient lorsqu'une hiérarchie de *locks* est utilisée. Par exemple, le *thread 1* réserve la variable A et tente ensuite de réserver la variable B pendant que le *thread 2* tente de réserver la variable B et A

par la suite. On dit alors que les deux *threads* s'interbloquent car les deux essaient d'accéder à une variable que l'autre à réservé. Les hiérarchies complexes de blocage doivent être évitées car les erreurs résident souvent dans un mauvais ordre de réservation et de libération des ressources.

Le faux partage (*false sharing*) Le faux partage n'est pas une erreur dans le programme mais plutôt un problème qui peut affecter la performance. Le *false sharing* survient lorsque deux *threads* manipulent des données qui se trouvent dans la même ligne de cache. Dans une architecture de type *SMP*, la cohérence de cache est assurée par le système. Ainsi, la modification d'une ligne de cache doit être signalée à tous les processeurs. Lorsqu'un *thread* modifie une donnée de la ligne il cause une invalidation de celle-ci et les autres *threads* doivent attendre que la ligne de cache soit rechargée de la mémoire centrale. Si cela arrive souvent, par exemple dans une boucle, la performance est considérablement dégradée.

1.5 Architecture du processeur Cell

Le processeur Cell [57] est une architecture unique en son genre car elle renferme une multitude de dispositifs dédiés au calcul haute-performance. Son architecture parallèle à plusieurs niveaux permet aux utilisateurs aguerris d'atteindre des performances jusque là réservées aux seuls clusters de machines et utilisant des paradigmes de haut niveau tels que le *message-passing*. En ce sens l'architecture du Cell, destinée initialement au domaine des jeux vidéos, a trouvé d'autres débouchés notamment dans le calcul scientifique au sens large.

Le Cell est composé d'un processeur PowerPC classique nommé PPE (Power Processor Element) et de huit unités de calcul accélératrices appelées SPE (*Synergestic Processor Element*). Ces unités de calcul sont reliées par un bus interne qui permet également l'accès à la mémoire principale (*Main Storage*), ainsi qu'à d'autres périphériques externes. Le processeur Cell est considéré comme un processeur hétérogène car il comporte deux types d'architectures différentes : celle du PPE qui n'est autre qu'une déclinaison du PowerPC 970, et celle des SPEs qui sont des unités SIMD accélératrices spécialisées dans des traitements contenant un flot de données important comme le multimédia par exemple. Le jeu d'instructions vectorielles des SPE est très proche d'Altivec [28], présent sur les architectures de type PowerPC

1.5.1 Vue générale

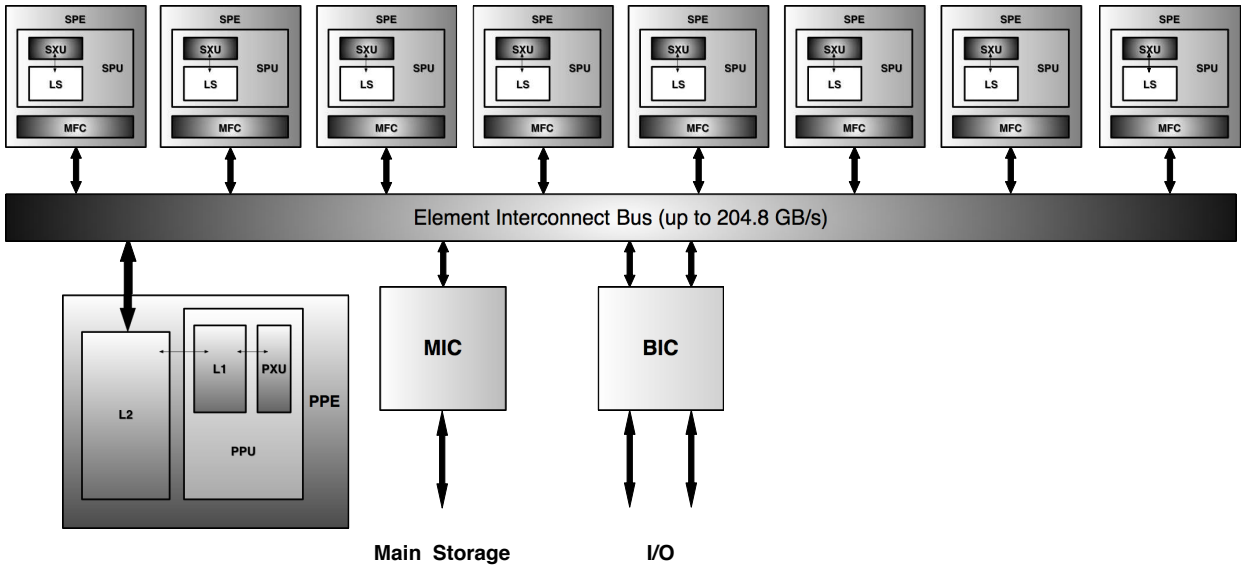

FIGURE 1.7 – Vue d'ensemble de l'architecture hétérogène du processeur Cell

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 à 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 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 par le biais d'un bus interne de communication très rapide appelé *Element Interconnect Bus* (EIB) (Fig. 1.7). Avec une fréquence d'horloge de 3.2 GHz, le processeur Cell peut atteindre théoriquement une performance crête de 204.8 GFlop/s en flottants simple-précision (32 bits) et 14.6 GFlop/s en flottants double-précision (64 bits). Il est important de noter que sur la première version du Cell, le débit des instructions arithmé-

tiques en double-précision était d'une instruction tous les 7 cycles ce qui explique les 14.6 GFlop/s. Ce débit a été amélioré pour atteindre une instruction par cycle sur la dernière génération *PowerXCell 8i*, ce qui correspond à une puissance de calcul de 102.4 GFlop/s.

1.5.2 Le PPE : Power Processor Element

Le PPE est un processeur 64 bits compatible avec l'architecture POWER [8], optimisé au niveau de l'efficacité énergétique[51]. La profondeur de pipeline du PPE est de 23 étages [58], chiffre qui peut paraitre faible par rapport au précédentes architectures PowerPC 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 réordonne pas dynamiquement les instructions (exécution dans l'ordre). 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 dix cycles. Le PPE supporte une hiérarchie conventionnelle de caches, avec un cache L1 (de niveau 1) données et instructions de 32 Ko, et un cache L2 de 512 Ko.
Le PPE peut lancer deux *threads* simultanément 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. Les registres sont dupliqués mais pas les caches qui sont partagés par les deux *threads*. Les instructions provenant de deux *threads* de calcul différents sont entrelacées afin d'optimiser l'utilisation de la fenêtre d'exécution.
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 celles des générations précédentes de PowerPC 970 et effectuent des opérations sur des registres 128 bits de données qui permettent un parallélisme de 2, 4, 8 ou 16, selon le type de données traité.

1.5.3 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, strictement équivalent au jeu d'instructions Altivec cité auparavant, mais optimisé au niveau de la consommation d'énergie et des performances pour les applications de calcul intensif et de multimedia. Le SPE contient une mémoire locale de 256 Ko (*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 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 contiguës de la mémoire en utilisant une liste de DMA, ce qui permet de définir un ensemble de transferts contigus de tailles différentes ; 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 dupliquées en mémoire centrale. La présence des mémoires locales introduit un autre niveau dans la hiérarchie mémoire au-dessus des registres. Le temps d'accès à ces mémoires est de l'ordre du cycle ce qui en fait de bons candidats pour réduire la latence d'accès à la mémoire centrale qui est de l'ordre du millier de cycles. De plus, le contrôleur DMA est indépendant de l'unité de calcul ce qui donne un niveau de parallélisme supplémentaire. La présence de ces mémoires privées permet d'appliquer différents modèles de programmation sur le processeur Cell.
La mémoire locale est le composant le plus important en taille du SPE, et il était important de l'implémenter de manière efficace. Une mémoire SRAM à un seul port est utilisée pour réduire la surface. En dépit du fait que la mémoire locale doit arbitrer entre lectures/écritures DMA, chargements d'instructions, et lecture/écriture mémoire, celle-ci a été conçue avec de ports de lecture moyens (128-bits) et larges (128 octets) dans le but de toujours fournir la meilleure performance possible. Le port le plus large est utilisé pour les commandes DMA et le chargement d'instructions. Cela est dû au fait qu'une commande DMA de 128 octets requière 16 cycles d'horloge pour placer les données sur le bus cohérent du Cell, même lorsque les com-

mandes DMA s'exécutent avec une bande passante maximale. Ainsi, 7 sur 8 cycles d'horloge restent disponibles pour les `load`, `store` et les chargements d'instructions. De la même manière, les instructions sont chargées par morceaux de 128 octets et la pression sur la mémoire locale est par conséquent réduite. La plus haute priorité est donnée aux commandes DMA, la seconde plus haute priorité aux commandes `load` et `store`, le chargement ou pré-chargement des instructions n'est fait que lorsqu'il y a des cycles disponibles. Toutefois, une instruction qui force la disponibilité d'une fenêtre d'exécution pour le chargement d'instructions existe.
Les unités d'exécution du SPE sont organisées autour d'un flot de données 128 bits. Un banc de 128 registres fournit assez d'entrées pour permettre à un compilateur de réorganiser des groupes d'instructions afin de masquer la latence d'exécution des instructions. Il n'y a qu'un seul banc de registres et toutes les instructions sont SIMD 128 bits avec une largeur d'élément différente selon le type (2x64 bits, 4x32 bits, 8x16 bits, 16x8 bits et 128x 1 bit). Deux instructions sont lancées à chaque cycle : une fenêtre d'exécution supporte les instructions en virgule fixe et flottante et l'autre exécute les instructions `load` et `store`, les permutations, ainsi que les instructions de branchement. Les opérations simples sur des entiers prennent deux cycles, les opérations sur des flottants simple-précision et les instructions `load` prennent 6 cycles. Les instructions vectorielles en flottants double-précision sont également supportées et leur débit maximal est d'une instruction tous les 7 cycles pour la première génération du Cell, et 1 cycle pour la dernière génération. Toutes les autres instructions sont entièrement pipelinées quelque soit la génération du processeur.
Afin de limiter la complexité du matériel dédié à la prédiction de branchement, une prédiction de branchement peut être fournie par le programmeur ou le compilateur. L'instruction de prédiction de branchement informe le matériel de l'adresse cible du branchement à venir, et le matériel répond en pré-chargeant au moins 17 instructions à partir de l'adresse cible de branchement. Une instruction de masque de sélection par bits peut également être utilisée afin d'éviter les branchements conditionnels dans le code. La surface dédiée aux unités de contrôle ne représente alors que 10 à 15 % de la surface totale de 10 mm^2 du SPE. Le SPE ne dissipe que très peu de *Watts* tout en opérant à une fréquence de 3.2 GHz.

1.5.4 Architecture de communication

Afin de tirer partie de toute la puissance de calcul enfouie dans le Cell, la charge de travail doit être distribuée et coordonnée entre le PPE et les SPEs. Les mécanismes spécifiques de communication du processeur permettent la collection et la distribution des données ainsi que la coordination d'activités concurrentes de manière efficace entre les unités de calcul. Puisque le SPE ne peut directement agir que sur des données et des instruction présentes dans sa mémoire locale, chaque SPE possède un contrôleur DMA qui effectue des transferts à haut débit entre la mémoire locale et la mémoire principale du Cell. Ces contrôleurs DMA permettent également des transferts directs entre les mémoires locales de deux SPEs dans le cas d'un schéma de calcul du type pipeline ou producteur-consommateur.
Par ailleurs, le SPE peut utiliser les signaux out les *mailboxes* pour effectuer des opérations de signalisation avec le PPE ou d'autres SPEs. Il existe également d'autre mécanismes de synchronisation disponibles sur le SPE qui opèrent de la même manière que les instructions atomiques des processeurs PowerPC. En réalité les opérations atomiques sur le SPE interagissent avec celles du PPE pour construire des mécanismes de synchronisation tels que les sémaphores entre PPE et SPE. Enfin, le Cell permet d'accéder à toutes les ressources du SPE ainsi qu'à l'intégralité de la mémoire locale de ce dernier par le biais d'une zone de la mémoire centrale réservée à cet effet. Cette variété de mécanismes de communication permet aux programmeurs d'implémenter différents modèles de programmation pour des applications parallèles [58].

Bus d'interconnection d'éléments *EIB*

La figure 1.8 illustre le *EIB* : le coeur de l'architecture de communication du processeur Cell et qui permet la communication entre le PPE, les SPEs, la mémoire centrale, et les entrées/sorties externes[65]. Le *EIB* possède des chemins de données séparés pour les commandes (requêtes à partir ou vers un élément sur le bus) et les données. Chaque élément du bus est connecté avec une liaison point à point au concentrateur d'adresses, qui reçoit et coordonne les commandes des éléments du bus, diffuse les commandes dans l'ordre à tous les éléments du bus (pour la scrutation), il agrège et diffuse ensuite la réponse des commandes. La réponse à la commande est le signal à l'élément du bus approprié pour démarrer le transfert de données.

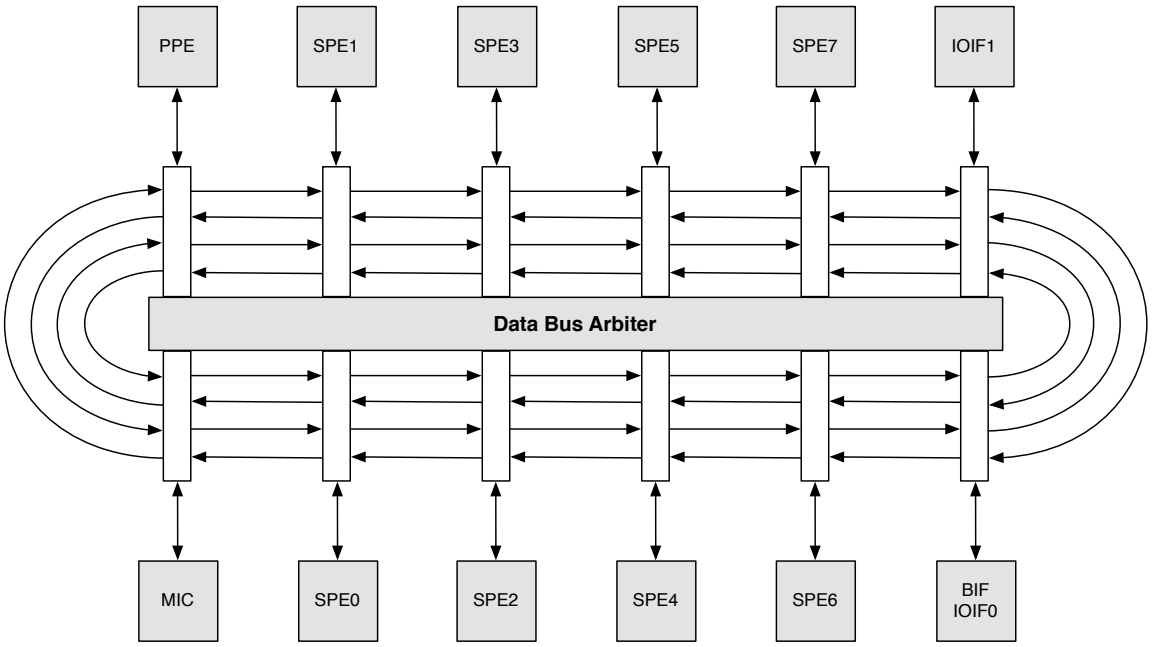

FIGURE 1.8 – Réseau d'interconnexion du Cell

Le réseau de données du *EIB* est constitué d'anneaux de données d'une largeur de 16 octets : deux dans le sens horaire, et deux dans le sens anti-horaire. Chaque anneau permet jusqu'à trois transferts concurrents, tant que leur chemins ne se croisent pas. Afin d'initier un transfert de données, les éléments du bus doivent demander l'accès au bus de données. L'arbitre de bus du *EIB* traite les requêtes et décide du noeud qui gère une requête donnée. L'arbitre choisit toujours l'anneau qui permet d'accomplir le chemin le plus court, et garantit ainsi que les données ne peuvent pas parcourir plus de la moitié de l'anneau avant d'atteindre leur destination. L'arbitre ordonnance également le transfert afin de s'assurer qu'il n'interfère pas avec une transaction en cours. Afin de minimiser les temps morts lors des lectures, l'arbitre donne la priorité aux requêtes venant du contrôleur mémoire. Il traite les autres requêtes équitablement suivant un ordonnancement *round-robin*. Ainsi, certains schémas de communication seront plus efficaces que d'autres. Le *EIB* opère à une fréquence deux fois moindres que celle du processeur. Chaque unité *EIB* peut simultanément envoyer et recevoir 16 octets de données durant chaque cycle de bus. La bande passante maximale du *EIB* est limitée par la fréquence à laquelle les adresses sont scrutées à travers les unités du système, dont la valeur est de une adresse par cycle de bus. Chaque scrutation d'adresse peut potentiellement trans-

férer jusqu'à 128 octets. Ainsi, dans un processeur Cell cadencé à 3.2 GHz, la bande passante crête théorique est 128 octets $\times$1.6 GHz = 204.8 Go/s.

Les interfaces d'entrées/sorties permettent à deux processeurs Cell d'être connectés en utilisant un protocole cohérent appelé *the broadband interface (BIF)*, qui permet d'étendre le réseau de multiprocesseurs en connectant les PPEs ainsi que 16 SPEs dans un seul réseau cohérent. Le protocole *BIF* opère à travers l'unité *IOIF0* (Fig. 1.8), une des deux interfaces d'entrées/sorties. La seconde interface *IOIF1* opère en mode non-cohérent seulement. La bande passante du *IOIF0* est configurée pour 30 Go/s en sortie et 25 Go/s en entrée.

La bande passante effective du *EIB* dépend de plusieurs facteurs : les positions relatives de la destination et la source, l'éventualité qu'un transfert puisse croiser d'autres transferts en cours, le nombre de processeurs Cell dans le système, le fait que les transferts soient entre mémoires locales de SPEs ou avec la mémoire principale et enfin de l'efficacité de l'arbitre de bus.

Une bande passante réduite peut résulter des situations suivantes :

- Tous les requérants accèdent à la même destination, par exemple à la mémoire locale en même temps.
- Tous les transferts sont dans la même direction et causent ainsi des temps morts sur deux ou sur la totalité des anneaux.
- Un nombre important de transferts de lignes de cache partielles réduit la bande passante.
- Tous les transferts de données doivent traverser la moitié de l'anneau pour atteindre leur destination et empêchent ainsi les unités sur le chemin d'utiliser l'anneau.

1.5.5 Contrôleur de flot mémoire (*Memory Flow Controler*)

Chaque SPE contient un *MFC* qui permet de connecter le SPE avec le *EIB* et gère les différents chemins de données entre le SPE et les autres éléments du Cell. Le *MFC* est cadencé à la même fréquence que celle du *EIB* donc à la moitié de celle du processeur. Le SPE interagit avec le *MFC* à travers l'interface de canaux du SPE. Les canaux sont des chemins de communication sans direction qui agissent comme des FIFO à capacité fixe. Cela signifie que chaque canal est défini soit en lecture seule, soit en écriture seule du point de vue du SPE. De plus, certains canaux sont définis avec une sémantique de blocage : ce qui signifie qu'une lecture

sur un canal en lecture seule ou une écriture sur un canal plein en écriture seule provoque le blocage du SPE jusqu'à la complétion de l'opération. Chaque canal possède un compteur associé qui indique le nombre d'éléments disponibles dans le canal. Le SPE utilise les instructions assembleur de lecture de canal (*rdch*), d'écriture de canal (*wrch*), et lecture de compteur de canal (*rchcnt*) pour accéder aux canaux du SPE.

Commandes DMA

Le *MFC* accepte et traite les commandes DMA que le PPE ou le SPE en utilisant l'interface du canal ou les registres d'entrées/sorties dupliqués en mémoire (*memory mapped I/O* MMIO). Les commandes DMA sont mises en file d'attente dans le *MFC*, et le SPE ou le PPE (celui qui a initié la commande) peut continuer l'exécution en parallèle du transfert de données, en utilisant les canaux de scrutation ou les interfaces bloquantes pour connaitre le statut d'un transfert. Cette exécution autonome des commandes DMA permet de couvrir les transferts par les calculs et d'améliorer ainsi les performances dans les cas où le calcul est dominant sur les transferts.
Le *MFC* supporte les transferts alignés de 1, 2, 4, 8 octets et multiples de 16 octets avec un maximum de 16 Ko par transfert. Les DMA *list* peuvent lancer jusqu'à 2048 transferts DMA en utilisant une seul commande DMA du *MFC*. Cependant, il n'y a que le *MFC* associé au SPE qui peut initier de telles commandes. Une DMA *list* est un tableau d'adresses source/destination et de taille de transfert, sauvegardées dans la mémoire locale du SPE. Lorsque un SPE initie une commande DMA *list*, il spécifie l'adresse et la longueur de la liste dans la mémoire locale. Une performance crête est atteignable pour les transferts quand l'adresse effective en mémoire principale et l'adresse en mémoire locale sont alignées sur 128 octets (taille de la ligne de cache) et la taille du transfert est un multiple pair de cette taille.

Signaux et *mailboxes*

Le mécanisme de signalisation supporte deux canaux : *Sig_Notify_1* et *Sig_Notify_2*. Le SPE peut lire ses propres canaux de signalisation en utilisant les lectures bloquantes *SPU_RdSigNotify1* et *SPU_RdSigNotify2*. Le PPE ou le SPE peuvent écrire dans ces canaux en utilisant les adresses dupliquées en mémoire centrale. Il existe une fonctionnalité spécifique des canaux de signalisation dans laquelle ils traitent les lectures comme des opérations OU logiques, permettant

ainsi une fonctionnalité de communication collective à travers les processeurs.
Chaque SPE possède également un ensemble de *mailboxes* qui peuvent fonctionner comme des canaux de communication étroits (32 bits) entre un SPE et d'autres SPEs ou le PPE. Le SPE possède une *mailbox* de réception à quatre entrées à lecture bloquante, et deux *mailboxes* d'émission à entrée unique en écriture bloquante également. Une de ces dernières peut générer une interruption pour le PPE lorsque le SPE y écrit. Le PPE utilise les adresses dupliquées en mémoire pour écrire dans les boites de réception des SPEs et pour lire dans une des boites d'envoi du SPE. Contrairement aux canaux de signalisation, les *mailboxes* se prêtent plus à des schémas de communication point à point tels que des modèles maître-esclave ou producteur-consommateur. Une communication en émission-réception typique, en utilisant les *mailboxes* dure approximativement 300 nano secondes.

Opérations atomiques

Afin de supporter des mécanismes de communication plus complexes, le SPE peut utiliser des commandes DMA spécifiques afin de mettre à jour de manière atomique une ligne bloquée en mémoire principale. Ces opérations appelées *get-lock-line-and-reserve* (*getllar*) et *put-lock-line-conditional* (*putllc*), sont conceptuellement équivalentes aux instructions *load-and-reserve* (*lwarx*) et *store-conditional* (*stcwx*) du *PowerPC*. L'instruction *getllar* lit la valeur d'une variable de synchronisation dans la mémoire principale et réserve son adresse. Si le PPE ou le SPE modifie la variable de synchronisation par la suite, le SPE perd sa réservation. L'instruction *putllc* ne met à jour une variable de synchronisation que si le SPE possède une réservation sur son adresse. Si *putllc* échoue, le SPE doit relancer une instruction *getllar* pour obtenir la nouvelle valeur de la variable de synchronisation et réessayer par la suite de la mettre à jour. L'unité atomique du *MFC* effectue les opérations DMA atomiques et gère les réservations prises par le SPE.
En utilisant les mises à jour atomiques, le SPE peut participer avec le PPE et d'autres SPEs à des protocoles de sémaphores, des barrières ou d'autres mécanismes de synchronisation.

Entrées/sorties dupliquées en mémoire : *Memory-Mapped I/O*

Les ressources dupliquées en mémoire jouent un rôle majeur dans la plupart des mécanismes de communication discutés auparavant. Elles se divisent en quatre catégories :

- **Mémoire locale** : la mémoire locale du SPE peut être entièrement dupliquée dans l'espace d'adressage effectif. Cela permet au PPE d'accéder à l'espace des SPE par l'intermédiaire d'opérations `load/store` simples, même si cela est beaucoup moins efficace qu'une opération DMA. l'accès à la mémoire locale dupliquée n'est pas synchronisé avec l'exécution du SPE. Ainsi le programmeur doit s'assurer que le programme SPE est conçu de telle sorte à autoriser des accès non synchronisés à ces données (par exemple, en utilisant des variables "volatiles").
- **Zone de *problem state*** : Les ressources de cet espace sont destinées à être utilisées directement par les applications, elles incluent l'accès au contrôleur DMA, au *mailboxes* et aux canaux de notification des signaux.
- **Zone *Privilege 1*** : Ces ressources sont disponibles à des programmes privilégiés tel que le système d'exploitation.
- **Zone *Privilege 2*** : Le système d'exploitation utilise ces ressources pour contrôler les ressources disponibles sur le SPE

1.5.6 Flot d'exécution DMA

Le contrôleur DMA du SPE gère la plupart des communications entre le SPE et les autres éléments du Cell, il exécute également les commandes DMA initiées par le PPE ou par d'autres SPEs. Une direction de transferts de données est toujours vue du côté SPE. Ainsi, les commandes qui transfèrent les données dans un SPE (à partir du PPE ou d'un autre SPE) sont considérées comme des opérations *get*, alors que les transferts du SPE vers la mémoire principale ou celle d'un autre SPE sont considérées comme des *puts*.
Les transferts DMA sont cohérents par rapport à la mémoire principale. Le *MFC* peut traiter les commandes DMA dans la file d'attente dans un ordre différent de celui de leur entrée dans la file d'attente. Lorsque l'ordre est important, le programmeur doit utiliser les commandes adéquates de *put* et *get* pour forcer des mécanismes de barrières par rapport à d'autres transferts dans la file d'attente. La *MMU* (*Memory Management Unit*) gère la traduction d'adresses des accès DMA à la mémoire principale, en utilisant des informations provenant des segments et

des tables définies dans l'architecture *PowerPC*. Le *MMU* possède un *TLB* (*Transfer Lookaside-Buffer*) pour mettre en cache les résultats des traductions faites récemment.
Le contrôleur DMA traite les commandes DMA présentes dans le *MFC*. Ce dernier possède deux files d'attente distinctes :

- *File d'attente SPE* : pour les commandes initiées par le SPE en utilisant l'interface de canaux.
- *File d'attente proxy* : pour les commandes initiées par le PPE ou d'autre éléments en utilisant les registres *MMIO*.

La file d'attente SPU contient 16 entrées et la file d'attente proxy en contient 8. Le flot basique d'un transfert DMA vers la mémoire principale initié par le SPE est le suivant :

1. Le SPE utilise l'interface de canal pour placer la commande DMA dans la file d'attente du *MFC*.
2. Le contrôleur DMA sélectionne une commande pour le traitement.
3. Si la commande est une DMA *list* et requière le chargement d'une liste. Le contrôleur DMA charge la liste dans la mémoire locale. Une fois que l'élément de la liste est chargé, les champs de l'élément sont mis à jour et le transfert suivant peut commencer
4. Si la commande requière une traduction d'adresse, le contrôleur la met dans le *MMU* pour le traitement. Lorsque la traduction est disponible dans le TLB, le traitement atteint l'étape suivante (déroulage). dans le cas d'un échec de lecture dans le TLB, le *MMU* effectue la traduction en utilisant les pages de tables en mémoire principale et met à jour le TLB.
5. Le DMA déroule la commande : il crée une requête sur le bus pour transférer le bloc de données suivant. Cette requête de bus peut transférer jusqu'à 128 octets de données mais peut également transférer une quantité moindre, en fonction des problèmes d'alignement ou de la quantité de données à transférer. Le contrôleur soumet la requête à l'interface de bus (*BIU*).
6. le *BIU* effectue les lectures dans la mémoire locale nécessaires au transfert, il envoie ensuite les données vers le *MIC* (*Memory Interface Controler*). Ce dernier transfert les donnés vers ou à partir de la mémoire principale.
7. Le processus de déroulage produit une séquence de requêtes de bus pour la commande

DMA, qui traverse le réseau de communication. La commande DMA demeure dans la file d'attente jusqu'à ce que toutes les requêtes soient terminées. Toutefois, le contrôleur peut très bien continuer à dérouler d'autres commandes. Lorsque toutes les requêtes vers le bus d'une commande sont terminées le contrôleur signale la complétion de la commande et la retire de la file d'attente.

En cas d'absence de congestion, un *thread* qui s'exécute sur le SPE peut lancer une requête DMA en 10 cycles d'horloge qui correspondent au temps d'écriture sur les canaux qui décrivent la source, la destination, la taille du transfert et l'étiquette de la commande. A partir de ce moment là, le contrôleur DMA peut exécuter la commande sans l'aide du SPE.
La latence globale, due à la génération de la commande DMA est de 30 cycles d'horloge lorsque toutes les ressources sont disponibles. Si le chargement d'un élément de liste est requis, cela rajoute 20 cycles supplémentaires. Si la file d'attente dans le *BIU* est pleine, le contrôleur est bloqué jusqu'à ce que les ressources soient disponibles à nouveau.
Une commande de transfert implique la scrutation de tous les éléments du bus pour assurer la cohérence et requière ainsi 50 cycles supplémentaire (100 cycles au total). Pour les commandes de type *get*, la latence restante est attribuée au transfert des données de la mémoire externe au contrôleur mémoire qui transitent seulement après par le bus interne vers la mémoire locale des SPEs. Pour les opérations *put* la latence n'inclue pas le temps de parcours du bus vers la mémoire externe car le SPE considère que le *put* est terminé une fois que les données ont été transmises au contrôleur mémoire.

1.5.7 Programmabilité et modèles de programmation

Si l'innovation dans l'architecture matérielle peut permettre d'atteindre de nouveaux niveaux de performances et/ou d'efficacité énergétique, il va de soi que l'effort fourni pour améliorer les performances doit être raisonnable. La programmabilité du Cell a été un souci pour ses concepteurs dès ces débuts, ils ont essayé de rendre le système le plus programmable possible et accessible au plus grand nombre. Mais il est clair que les aspects architecturaux qui sont les plus difficiles à appréhender par les programmeurs sont ceux qui renferment le plus grand potentiel d'amélioration des performances. L'existence de la mémoire locale et le fait que celle-ci doit être gérée par le logiciel est un bon exemple de verrou technologique. Cette gestion peut éventuellement être confiée à un compilateur mais la tâche peut être complexe

suivant le cas d'utilisation. Le second aspect de la conception qui affecte la programmabilité est la nature SIMD du flot de données. Le programmeur peut ignorer cet aspect là de l'architecture, mais en faisant abstraction de celui-ci, une grande partie de la performance est laissée de côté. Il faut noter, que comme pour les applications qui s'exécutent sur des processeurs classiques sans utiliser leur unité SIMD, le Cell peut être programmé comme un processeur scalaire. La nature SIMD des SPEs est gérée par les programmeurs et supportée par les compilateurs de la même manière que pour les processeurs possédant des unités SIMD.
Le SPE diffère des processeurs généralistes par plusieurs aspects : la taille du banc de registres (128 entrées), la manière dont les branchements et les instructions qui permettent de synchroniser la mémoire locale avec le lancement des instructions sont gérés . Ces particularités peuvent être utilisées par un compilateur pour l'optimisation, et le programmeur a intérêt d'en avoir connaissance afin de tirer profit au maximum des dispositifs de l'architecture, mais en tout état de cause il n'est pas nécessaire de programmer les SPEs en assembleur. Une autre différence majeure qui distingue les SPEs des processeurs conventionnels est le fait que ceux-ci ne puissent supporter qu'un seul contexte de programme à la fois. Ce contexte peut être un *thread* dans une application ou un *thread* dans un mode privilégié, qui étend le système d'exploitation. Le processeur Cell supporte la virtualisation et permet à plusieurs systèmes d'exploitation de s'exécuter de manière concurrente au dessus d'un logiciel de virtualisation qui s'exécute en mode hyperviseur.
L'intégration d'un processeur de contrôle compatible avec l'architecture PowerPC permet au Cell d'exécuter les applications Power et PowerPC 32 et 64 bits sans aucune modification. Toutefois, l'utilisation des SPEs est nécessaire pour atteindre les performances et profiter pleinement de l'efficacité énergétique du Cell. L'ensemble des dispositifs de communication et de transfert de données permet d'imaginer plusieurs modèles de programmation possibles pour le Cell. Par exemple il est possible d'utiliser le SPE comme coprocesseur sur lequel on décharge une partie du calcul afin de l'accélérer. Plusieurs changements ont été effectué sur l'architecture pour des raisons de programmabilité. Des programmes de test, des bibliothèque de calcul, des extensions de systèmes d'exploitation ont été écrits, analysés vérifiés sur un simulateur fonctionnel avant la finalisation de l'architecture et l'implémentation du Cell. Certains des modèles de programmation les plus communs sont décrits dans ce qui suit.

Modèle de décharge de fonction

Ce modèle peut être le plus rapide à mettre en oeuvre tout en bénéficiant du fort potentiel de performances du Cell. Dans ce modèle de programmation, les SPEs sont utilisés comme des accélérateurs de certaines fonctions critiques. L'application principale peut être une application existante ou une nouvelle application qui s'exécute sur le PPE. Dans ce modèle les fonctions critiques invoquées par le programme principal sont remplacées par des fonctions qui s'exécutent sur un ou plusieurs SPEs. La logique du programme principal reste inchangée. La fonction originale est optimisée et recompilée pour le SPE, et l'exécutable SPE généré est intégré au programme PPE dans une section en lecture seule avec la possibilité de l'invoquer à distance. Le programmeur détermine statiquement les portions de code à exécuter sur le PPE et celles à décharger sur les SPEs. Un compilateur *single-source* (à un seul fichier source en opposition à la compilation séparée du code PPE d'un côté et celui du SPE de l'autre) a été conçu par IBM. Ce dernier se base sur des directives de compilation qui permettent de décharger le calcul sur les SPEs pour une portion délimitée du code. Une des évolutions les plus significatives que pourrait connaitre les compilateurs est de pouvoir déterminer automatiquement les portions de code à déporter sur les SPEs.

Modèle d'accélération du calcul

Ce modèle est centré sur le SPE : il fournit une utilisation plus importante du SPE par l'application que le modèle de décharge de fonction. Le modèle est mis en oeuvre en effectuant les tâches de calcul intensif sur les SPEs. le code source PPE agit essentiellement comme serveur et unité de contrôle pour les SPEs. Les techniques de parallélisation peuvent être utilisées pour répartir le calcul sur les SPEs. La charge de travail peut être partitionnée manuellement par le programmeur ou parallélisée automatiquement par un compilateur. Ce partitionnement doit inclure un ordonnancement efficace des opération DMA pour le mouvement des données et du code entre les SPEs. Ce modèle de programmation peut s'appuyer sur une mémoire partagée ou sur un système par passage de message. Dans plusieurs situations, le modèle d'accélération de calcul peut être utilisé pour fournir une accélération à des fonctions mathématiques de calcul intensif sans que le code ne soit modifié de manière significative.

Modèle de calcul par flux

Comme mentionné auparavant, les SPEs peuvent se baser sur un système par passage de message pour faire un calcul en parallèle. Il est donc très simple de mettre en oeuvre des pipelines séquentiels ou parallèles, dans lesquels chaque SPE effectue un calcul distinct sur les données qui transitent par lui. Le PPE peut alors agir comme contrôleur de flux et les SPEs comme des processeurs de flux de données. Lorsque les SPEs ont une charge de travail équivalente, ce modèle peut s'avérer très efficace sur le Cell, à partir du moment où les données résident le plus longtemps possible dans les mémoires internes du SPE. Dans certains cas, il peut être plus efficace de faire migrer le code d'un SPE vers un autre SPE au lieu du mouvement de données plus conventionnel.

Modèle à mémoire partagée

Le processeur Cell peut être programmé comme un multi-processeur à mémoire partagée ayant des unités avec des jeux d'instructions différents qui permettent de couvrir un spectre de tâches qu'un seul jeu d'instructions ne peut pas couvrir. Le PPE et les SPEs peuvent interagir dans un modèle à mémoire partagée complètement cohérent. Les `load`s conventionnels en mémoire partagée sont remplacés par une combinaison d'opérations DMA de la mémoire partagée vers la mémoire locale du SPE, en utilisant une adresse effective commune au PPE et aux SPEs, et un `load` de la mémoire locale vers le banc de registres. Les opérations conventionnelles du type `store` sont remplacées par une combinaison d'un `store` du banc de registres vers la mémoire locale et d'une opération DMA de la mémoire locale vers la mémoire partagée en utilisant le même mécanisme d'adressage que pour le `load`. On peut alors imaginer un compilateur qui s'appuie sur les mêmes mécanismes pour utiliser la mémoire locale des SPEs comme un cache de données et d'instructions lues à partir de la mémoire partagée.

1.5.8 Environnement de développement de base

L'environnement de développement fourni par IBM est le *Cell Software Development Kit (Cell SDK)*. Plusieurs versions ont vu le jour, la première a précédé la livraison des premières puces Cell. L'exécution du code se faisait alors sur un simulateur *cycle-accurate* du processeur [55]. La dernière version du SDK connue à ce jour est la 3.1. La description qui suit est

basée sur une version antérieure 2.1 du SDK. L'environnement de développement contient les composants suivants :

- **La chaîne d'outils GNU :** Cette chaîne contient le compilateur *gcc* (*GNU C-language Compiler*) pour le PPE et le SPE. Les deux compilateurs peuvent effectuer de la *cross* compilation sur des plateformes x86. La présence d'une suite d'outils *open source* est le fruit d'une stratégie d'ouverture d'IBM vers le développement communautaire décrite dans [46] ;
- **Le compilateur IBM XL C/C++ :** C'est le compilateur conçu par IBM pour le processeur Cell [32]. Il contient également deux compilateurs spécifiques, un pour le PPE ; l'autre pour le SPE. La *cross* compilation est également possible avec les compilateurs *XLC* . La présence de deux chaînes de compilateurs distinctes s'explique par le fait que *gcc* est destiné à la communauté *open source* et dépend de sa contribution, alors que le compilateur *XLC* est développé et commercialisé par IBM. Dans nos travaux nous avons pu utiliser les deux compilateurs. *XLC* se distingue de *gcc* par la gestion d'OpenMP et un code généré plus efficace sur le PPE et le SPE.
- **IBM full-system simulator :** Le simulateur du Cell est une application logicielle qui émule le comportement d'un système complet contenant un processeur Cell. Un système d'exploitation linux est géré par le simulateur. On peut ainsi, exécuter des applications pré-compilées sur le simulateur. Il existe plusieurs modes de simulation possibles, du mode purement fonctionnel, au mode précis au cycle près (*cycle accurate*).
- **Noyau Linux :** C'est le noyau Linux pour le Cell [10] qui supporte notamment l'exécution parallèle sur les SPEs. Il est développé et maintenu par le *Barcelona Supercomputer Center*. Celui-ci s'exécute sur le PPE qui peut gérer un système d'exploitation alors que les SPEs sont des coprocesseurs dédiés au calcul intensif.
- **La bibliothèque de gestion du support d'exécution SPE :** Cette bibliothèque constitue l'API bas-niveau standard pour la programmation d'applications pour le Cell, en particulier pour l'accès au SPEs. La bibliothèque fournit une API neutre par rapport aux systèmes d'exploitation et la manière dont ils gèrent les SPEs, à travers une interface standard très proche de POSIX.

Nous avons cité ci-dessus les principaux composants du SDK que nous avons utilisé lors du développement des applications sur le processeur Cell. D'autres composants sont fournis avec

le SDK : des bibliothèques de calcul mathématique et numérique optimisées pour le Cell, des utilitaires de mesure de performance ainsi qu'un *plugin* pour l'environnement de développement Eclipse spécifique au développement sur le processeur Cell.

Les threads POSIX sur le Cell

Sur un système Linux pour le Cell, le *thread* principal s'exécute sur le PPE. Un *thread* PPE peut contenir un ou plusieurs *threads* Linux qui peuvent s'exécuter soit sur le PPE soit sur les SPEs. Un *thread* qui s'exécute sur le SPE possède son propre contexte incluant un banc de registres de 128 $\times$ 128 bits, 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 à travers un espace alloué en mémoire principale nommé *problem state space*, ou encore indirectement via la mémoire centrale ou les routines de la *SPE Runtime Management Library*. L'OS définit le mécanisme et la politique d'ordonnancement pour les SPEs 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 bibliothèque 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 aux SPEs en utilisant un modèle 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'affinité pour affecter les *threads* à un SPE spécifique. Les dispositifs architecturaux de communication entre les *threads* ainsi que les mécanismes de synchronisation (mailbox, signaux, etc...) peuvent être accèdés 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 bibliothèque 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*) intégré dans un exécutable Cell. Le *miniloader* un programme SPE de 256 bits, charge le 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 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 communication. Le code à exécuter réside alors dans la mémoire locale des SPEs.

Exemple de mise en oeuvre

Dans le but de donner plus de détails sur la mise en oeuvre d'un programme sur le processeur Cell, nous utilisons un exemple simple d'addition de deux vecteurs. Le listing 1.1 est un fichier d'entête qui contient la définition de certaines structures fondamentales telles que le *control block* servant à transmettre des informations nécessaires au SPEs pour l'exécution du calcul. Ce fichier contient également les déclarations de certaines constantes ainsi que la redéfinition de certains types pour une écriture plus compacte. Le listing 1.2 contient le code exécuté par le PPE, qui est en charge de la création, de la synchronisation et de la destruction des *threads* SPE. Ce code contient également la logique nécessaire à la distribution du calcul sur les SPEs. Le fichier listé dans 1.3 contient le code source de calcul : c'est la partie exécutée par le SPE. On pourra noter la distinction entre les sections de code de transfert et de calcul. Nous avons utilisé les instructions vectorielles pour le calcul car la vectorisation automatique n'était pas très bien gérée par les compilateurs. Enfin, le code source est volontairement verbeux car nous voulions démontrer la complexité que représente la mise en oeuvre d'un code très simple sur le processeur Cell, en utilisant les outils de base fournis par IBM. La structure de base du code PPE ainsi que les fonctions de transferts une fois implantées et testées ont été regroupées dans des librairies utilisées par la suite dans le développement de code et dans la bibliothèque `CELL_MPI`.

```
1  #ifndef __COMMON_H__
2  #define __COMMON_H__
3  #include <stdlib.h>
4  #define SPU_THREADS 8
5  #define SIZEX 1024
6  #define SIZEY 1024
7  #define MEM_ALIGN 128
8  #define VSIZE (SIZEX*SIZEY/4)
9  #define SIZE_CB    128
10 #define ALIGN_LOG2 7
11 #define DMA_MAX    16384
12 /* la stucture contient des informations qui servent
13  * a la creation des threads dans les SPE
14  */
15 typedef struct ppu_pthread_data {
16     spe_context_ptr_t speid;
17     pthread_t pthread;
18     void *argp;
19 } ppu_pthread_data_t;
20 /* union servant a acceder une adresse 64-bit
21  * de deux manieres differentes
22  */
23 typedef union
24 {
25   unsigned long long ull;
26   unsigned int ui[2];
27 } addr64;
28 /* structure definissant le control block qui contient
29  * toutes les informations necessaires a l'execution
30  * du code SPU
31  */
32 typedef struct _control_block {
33   unsigned long long source1_ad;
34   unsigned long long source2_ad;
35   unsigned long long target_ad;
36   unsigned int size;
37   unsigned int tileh;
38   unsigned int tilew;
```

```
39   unsigned char pad[92];   // padding pour remplir une ligne de cache (128 octets
         )
40 } control_block_t;
41 typedef vector float vFloat32;
42 typedef unsigned int uint32;
43 typedef unsigned long long uint64;
44 #endif /* __COMMON_H__ */
```

Listing 1.1 – Exemple d'addition de deux vecteurs en *pthreads* : fichier d'entête commun `common.h`

```
1  #include "common.h"
2  #include <sched.h>
3  #include <libspe2.h>
4  #include <pthread.h>
5  #include <dirent.h>
6  #include <stdio.h>
7  #include <errno.h>
8  #include <string.h>
9  /* cette fonction est executee a la creation du thread */
10 void *ppu_pthread_function(void *arg){
11     ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
12     int rc;
13     unsigned int entry = SPE_DEFAULT_ENTRY;
14     if ((rc = spe_context_run(datap->speid, &entry, 0, datap->argp, NULL, NULL))
           < 0) {
15         fprintf (stderr, "Failed spe_context_run(rc=%d, errno=%d, strerror=%s)\n"
               , rc, errno, strerror(errno));
16         exit (1);
17     }
18     pthread_exit(NULL);
19 }
20 /* Ceci est le pointeur vers le code SPU
21  * il doit etre utilise a la creation
22  * du thread
23  */
24 extern spe_program_handle_t vecadd_spu;
25 /* Ce sont les handles retournes par
26  * "spe_context_create" et "pthread_create"
```

```
27  */
28 ppu_pthread_data_t datas[SPU_THREADS];
29 int main(int argc, char *argv[])
30 {
31   int i,rc;
32   unsigned int chunkw,chunkh,chunksize, chunksizev;
33   unsigned int tilew, tileh, tilesize, nbtiles;
34   vFloat32 *vect1, *vect2, *vect3;
35   float *sptr_vect1, *sptr_vect2, *sptr_vect3;
36   control_block_t* cb;
37   uint64 *ad_input1, *ad_input2, *ad_output;
38   // chunk size computing
39   chunkw=SIZEY;
40   chunkh=SIZEX/SPU_THREADS;
41   chunksize=chunkw*chunkh;
42   chunksizev=chunksize/4;
43   tilew=SIZEY;
44   tileh=4;
45   tilesize=tileh*tilew;
46   nbtiles=chunksize/(tilesize);
47   // allocation des vecteurs
48   vect1=(vFloat32*)memalign(MEM_ALIGN,VSIZE*sizeof(vFloat32));
49   vect2=(vFloat32*)memalign(MEM_ALIGN,VSIZE*sizeof(vFloat32));
50   vect3=(vFloat32*)memalign(MEM_ALIGN,VSIZE*sizeof(vFloat32));
51   // allocation des control blocks
52   cb = (control_block_t*)memalign(MEM_ALIGN,SPU_THREADS*sizeof(control_block_t));
53   ad_input1 = (uint64*)memalign(MEM_ALIGN,SPU_THREADS*sizeof(uint64));
54   ad_input2 = (uint64*)memalign(MEM_ALIGN,SPU_THREADS*sizeof(uint64));
55   ad_output = (uint64*)memalign(MEM_ALIGN,SPU_THREADS*sizeof(uint64));
56   // initialisation du control blocks
57   for (i = 0; i < SPU_THREADS; i++){
58   ad_input1[i] = (uint64)((vFloat32*)vect1+i*chunksizev);
59   ad_input2[i] = (uint64)((vFloat32*)vect2+i*chunksizev);
60   ad_output[i] = (uint64)((vFloat32*)vect3+i*chunksizev);
61   cb[i].source1_ad = ad_input1[i];
62   cb[i].source2_ad = ad_input2[i];
63   cb[i].target_ad = ad_output[i];
64   cb[i].size =chunksize;
```

```
65  cb[i].tilex=tileh;
66  cb[i].tiley=tilew;
67  cb[i].count=nbtiles;
68  }
69  // pointeurs scalaires sur buffers vectoriels
70  sptr_vect1 = (float*)vect1;
71  sptr_vect2 = (float*)vect2;
72  sptr_vect3 = (float*)vect3;
73  // initialisation des buffers
74  for (int i = 0; i < SIZEX*SIZEY; i++){
75    sptr_vect1[i] = 1.0f;
76    sptr_vect1[i] = 2.0f;
77    sptr_vect3[i] = 0.0f;
78  }
79  /* allocation des taches SPE */
80  for (i = 0; i < SPU_THREADS; i++) {
81    // creation du contexte
82    if ((datas[i].speid = spe_context_create (0, NULL)) == NULL){
83      fprintf (stderr, "Failed spe_context_create(errno=%d strerror=%s)\n", errno
          , strerror(errno));
84      exit (3+i);
85    }
86    // chargement du programme
87    if ((rc = spe_program_load (datas[i].speid, &vecadd_spu)) != 0){
88      fprintf (stderr, "Failed spe_program_load(errno=%d strerror=%s)\n", errno,
          strerror(errno));
89      exit (3+i);
90    }
91    // initialisation du pointeur argp
92    datas[i].argp = (unsigned long long *)(&cb[i]);
93    // creation du thread
94    if ((rc = pthread_create (&datas[i].pthread, NULL, &ppu_pthread_function, &
         datas[i])) != 0){
95      fprintf (stderr, "Failed pthread_create(errno=%d strerror=%s)\n", errno,
          strerror(errno));
96      exit (3+i);
97    }
98  }
```

```
99   /* attendre que tous les SPEs aient fini */
100  for (i=0; i<SPU_THREADS; ++i){
101    // jointure des threads
102    if ((rc = pthread_join (datas[i].pthread, NULL)) != 0){
103      fprintf (stderr, "Failed pthread_join(rc=%d, errno=%d strerror=%s)\n", rc,
           errno, strerror(errno));
104      exit (1);
105    }
106    // destruction du contexte
107    if ((rc = spe_context_destroy (datas[i].speid)) != 0){
108      fprintf (stderr, "Failed spe_context_destroy(rc=%d, errno=%d strerror=%s)\n
           ", rc, errno, strerror(errno));
109      exit (1);
110    }
111  }
112  /* barriere de synchronisation */
113  __asm__ __volatile__ ("sync" : : : "memory");
114  // deallocation de la memoire
115  free((void*)vect1);
116  free((void*)vect2);
117  free((void*)vect3);
118  free((void*)ad_input1);
119  free((void*)ad_input2);
120  free((void*)ad_output);
121  return 0;
122 }
```

Listing 1.2 – Exemple d'addition de deux vecteurs en *pthreads* : code source du *PPE* `vecadd_ppe.c`

```
1  #include <spu_mfcio.h>
2  #include <stdio.h>
3  #include <malloc_align.h>
4  #include <free_align.h>
5  #include <spu_intrinsics.h>
6  /* control block local */
7  control_block_t m_cb __attribute__ ((aligned (SIZE_CB)));
8  int main(unsigned long long speid, addr64 argp, addr64 envp)
9  {
```

```
10  float *m_source1_ad, *m_source2_ad, *m_target_ad;
11  unsigned int m_tileh, m_tilew, m_tilewv;
12  unsigned int count, rest, offset;
13  vFloat32 *bufvect0, *bufvect1, *bufvect2;
14  const int tag0 = 10;
15  const int tag1 = 11;
16  const int tag2 = 12;
17  const int tag3 = 13;
18  unsigned int sizev = sizeof(vFloat32);
19  /* tranfert du control block de la memoire centrale vers celle du SPU */
20  spu_mfcdma32((void*)(&cb), (unsigned int)argp.ui[1], SIZE_CB, tag0, MFC_GET_CMD
        );
21  mfc_write_tag_mask(1<<tag0);
22  mfc_read_tag_status_all();
23  /* initialisation des adresses des buffers et de leurs tailles */
24  m_source1_ad = (vFloat32*)m_cb.source1_ad;
25  m_source2_ad = (vFloat32*)m_cb.source2_ad;
26  m_target_ad  = (vFloat32*)m_cb.target_ad;
27  m_tileh  = m_cb.tileh;
28  m_tilew  = m_cb.tilew;
29  m_tilewv = m_tilew/4;
30  /* allocation de la memoire SPU */
31  bufvect0 = (vFloat32*)_malloc_align(m_tileh*m_tilewv*sizev, ALIGN_LOG2);
32  bufvect1 = (vFloat32*)_malloc_align(m_tileh*m_tilewv*sizev, ALIGN_LOG2);
33  bufvect2 = (vFloat32*)_malloc_align(m_tileh*m_tilewv*sizev, ALIGN_LOG2);
34  // calcul du nombre de tuiles
35  offset = DMA_MAX/sizev;
36  count  = (m_tileh*m_tilewv)/offset;
37  rest   = (m_tileh*m_tilewv)%offset;
38  /* transfert du premier operande de la memoire centrale */
39  // premiers transferts
40  for (int i=0; i<count; i++){
41    mfc_get((void*)((uint32)bufvect0+i*offset*sizev,(uint64)m_source1_ad+i*offset
          *sizev, DMA_MAX, tag1, 0, 0);
42  }
43  // dernier DMA si pas multiple de 16Ko
44  if(rest != 0){
```

```
45      mfc_get((void*)((uint32)bufvect0+count*offset*sizev,(uint64)m_source1_ad+
            count*offset*sizev, rest*sizev, tag1, 0, 0);
46    }
47    mfc_write_tag_mask(1<<tag1);
48      mfc_read_tag_status_all();
49    /* transfert du second operande de la memoire centrale */
50    // premiers transferts
51    for (int i=0; i<count; i++){
52      mfc_get((void*)((uint32)bufvect1+i*offset*sizev,(uint64)m_source2_ad+i*offset
            *sizev, DMA_MAX, tag2, 0, 0);
53    }
54    // dernier DMA si pas multiple de 16Ko
55    if(rest != 0){
56      mfc_get((void*)((uint32)bufvect1+count*offset*sizev,(uint64)m_source2_ad+
            count*offset*sizev, rest*sizev, tag2, 0, 0);
57    }
58    mfc_write_tag_mask(1<<tag2);
59      mfc_read_tag_status_all();
60    /* addition des deux tuiles */
61    for(int i=0;i<m_tileh*m_tilewv;i++){
62      bufvect2[i]=spu_add(bufvect0[i],bufvect1[i]);
63    }
64    /* transfert du buffer de sortie vers la memoire externe */
65    // premiers transferts
66    for (int i=0; i<count; i++){
67      mfc_put((void*)((uint32)bufvect2+i*offset*sizev,(uint64)m_target_ad+i*offset*
            sizev, DMA_MAX, tag3, 0, 0);
68    }
69    // dernier DMA si pas multiple de 16Ko
70    if(rest != 0){
71      mfc_put((void*)((uint32)bufvect2+count*offset*sizev,(uint64)m_target_ad+count
            *offset*sizev, rest*sizev, tag3, 0, 0);
72    }
73    mfc_write_tag_mask(1<<tag3);
74    mfc_read_tag_status_all();
75    /* deallocation de la memoire SPU */
76    _free_align((void*)bufvect0);
77    _free_align((void*)bufvect1);
```

```
78    _free_align((void*)bufvect2);
79      return 0;
80  }
```

Listing 1.3 – Exemple d'addition de deux vecteurs en *pthreads* : code source du *SPE* `vecadd_spe.c`

Processus de génération de code

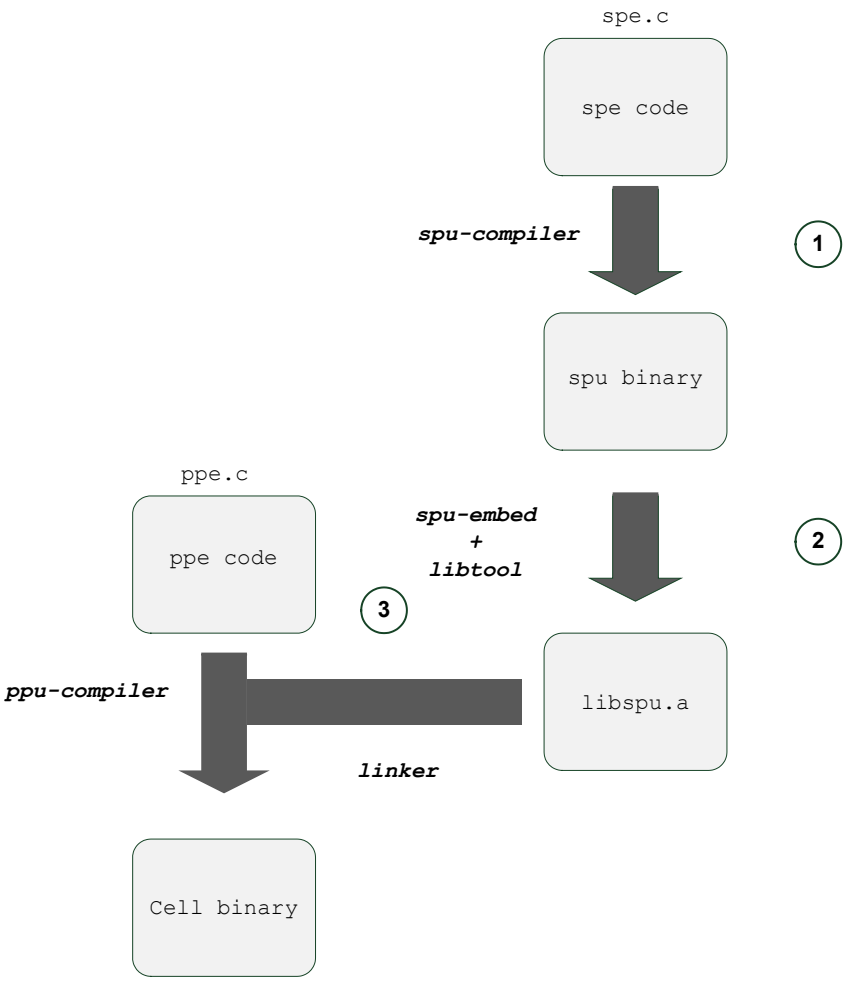

FIGURE 1.9 – Processus *dual source* de génération de code exécutable pour le Cell

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 codes sources distincts, un code pour le SPE (`spe.c` sur la figure 1.9) qui contient le code exécuté sur le SPE. Un

deuxième code source qui est celui s'exécutant sur le PPE (`ppe.c` sur la figure 1.9) contient le thread maître qui gère les threads SPE. Le processus de génération de code exécutable est décrit dans la figure 1.9. Dans la première étape le code SPE est compilé et un binaire exécutable SPE est ainsi généré. Celui-ci est par la suite traité par un outil spécifique *spu-embedd* qui permet de transformer ce binaire code binaire pouvant être enfoui dans l'exécutable du PPE. Cette procédure se fait par l'éditeur de liens qui considère alors le code SPE comme une bibliothèque dont le code objet doit être intégré dans l'exécutable final.

1.6 Conclusion

Les architectures parallèles ont fait leur apparition dans les premières années de l'informatique moderne. Le parallélisme y est exploité à plusieurs niveaux, instructions, données et tâches. Il existe plusieurs types de hiérarchies mémoire pour machines parallèles : essentiellement partagées et distribuées. Afin de pouvoir exploiter le parallélisme et la hiérarchie mémoire de ce type de machines, plusieurs modèles de programmation ont été proposés, leur but étant de faciliter la programmation ainsi que d'atteindre de meilleures performances grâce à une exécution parallèle.
La parallélisation de code qui consiste en l'adaptation d'un code source séquentiel pour son exécution sur une machine parallèle pose plusieurs défis. L'algorithme ainsi que son implémentation doivent être adaptés à une exécution parallèle. L'exécution concurrente induit des contraintes de synchronisation ainsi que la nécessité d'une communication inter-processeurs. Ces facteurs peuvent freiner l'exécution efficace du programme en parallèle.
Le processeur Cell possède une architecture parallèle hétérogène complexe. Il renferme plusieurs dispositifs qui font de son architecture un concentré de technologies parallèles. Plusieurs formes de parallélisme y sont présentes et à plusieurs niveaux : le parallélisme de données résultant de la nature vectorielle (SIMD) des processeurs SPE, le parallélisme d'instructions grâce à la possibilité d'exécuter deux instructions par cycle, le parallélisme de tâches car plusieurs *threads* peuvent s'exécuter de manière concurrente sur différents SPEs et enfin le parallélisme transfert/calcul qui résulte de contrôleurs DMA indépendants des unités de calcul sur les SPEs. L'architecture mémoire quand à elle, est similaire à celle d'un DSP embarqué. Cette hiérarchie mémoire distribuée nécessite une gestion explicite pour une utilisation

efficace. La programmation par *threads* sur le Cell est le modèle de base pour la mise en oeuvre de codes parallèles sur cette architecture. Le développement se fait alors à l'aide d'une API bas-niveau qui permet de garder un contrôle précis sur le déroulement de son application tout en ayant une grande flexibilité dans le choix de déploiement d'un algorithme donné. Grâce à des dispositifs architecturaux de signalisation et de synchronisation, on peut concevoir des programmes parallèles très efficaces en termes de performance sur le Cell. Toutefois, du point de vue du programmeur, la mise en oeuvre du code est certainement plus laborieuse que lorsque d'autres modèles de programmation sont utilisés, mais celle-ci peut être justifiée dans le cadre de fortes contraintes sur les temps d'exécution ou dans le cas où des schémas de parallélisation classiques ne sont pas adaptés à l'application déployée.

Au vu des difficultés de programmation énoncées ci-dessus, d'autres outils de programmation pour le Cell ont été développés. Ils se basent sur l'API de base et fournissent des outils de plus haut-niveau plus faciles à prendre en main par le développeur. La question qui se pose alors est celle de la garantie de la performance et de la flexibilité d'utilisation de ces outils par rapport à l'API fournie par IBM.

Chapitre 2

Parallélisation de la détection de coins d'Harris

Le chapitre précédent a permis de présenter l'architecture particulière du Cell ainsi que l'environnement de développement de base pour cette architecture. Nous avons mis en avant la complexité de l'architecture accentuée notamment par son hétérogénéité et les espaces d'adresses distincts entre PPE et SPEs. De plus, nous avons présenté l'outil de programmation de base proposé qui est assez difficile à prendre en main par les développeurs et nécessite des connaissances poussées de l'architecture. Au vu des difficultés citées auparavant, l'implémentation d'une application de traitement d'images sur le processeur Cell devient une tâche complexe. L'obtention de performances proches des chiffres théoriques donnés par le constructeur est encore moins évidente.

Ce chapitre aborde dans le détail, la démarche de parallélisation manuelle d'un exemple de code de traitement d'images sur le processeur Cell. Tout au long de cette étude, plusieurs aspects de l'optimisation sont traités. D'une part les optimisations spécifiques au traitement d'images et d'autre part ceux spécifiques au processeur Cell et ses différents niveaux de parallélisme.

L'algorithme considéré est celui de la détection de points d'intérêts selon l'algorithme 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 niveau qu'on retrouve dans plusieurs applications plus complexes, comme la reconstruction 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 dont la complexité arithmétique et le rapport calcul/transfert, justifient le portage sur des architectures massivement parallèles
- le respect d'un débit de traitement de 25 fps (*frame per second*) ne peut pas être obtenu par une implémentation triviale sur une architecture mono-coeur. .

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 calcul ainsi que la présence de plusieurs coeurs physiques qui permettent de répartir la charge de calcul de plusieurs manières possibles, soit sous forme de parallélisme de données uniquement, ou alors de parallélisme de tâches ou encore un mélange des deux.

2.1 Algorithme de Harris

La détection de points d'intérêt de *Harris* et *Stephen* [48] est utilisée dans les systèmes de vision par ordinateur pour l'extraction de connaissances comme la détection de mouvement, la mise en correspondance d'images, le suivi d'objets, la reconstruction 3D et la stabilisation d'images. Cet algorithme fut proposé pour palier les manques de l'algorithme de *Moravec* [77] qui était sensible au bruit et pas sensible à la rotation. Un coin peut être défini comme étant l'intersection de deux contours alors qu'un point d'intérêt peut être défini comme un point ayant une position bien déterminée et qui peut être détectée de manière robuste. Ainsi, le point d'intérêt peut être un coin mais aussi un point isolé d'intensité maximale ou minimale localement, une terminaison de ligne ou encore un point de courbe où la courbure est localement maximale.
Les résultats qui suivent sont présentés dans [89], [87] et [88].

2.1.1 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, l'opérateur de Harris est basé sur l'estimation de l'auto-corrélation locale S dont l'équation est la suivante :

$$S(x,y) = \sum_u \sum_v w(u,v) \left(I(u+x, v+y) - I(u,v)\right)^2 \tag{2.1}$$

Où $w(u,v)$ est un noyau de lissage Gaussien
$I(u+x, v+y)$ peu être approché par un développement de Taylor. Supposons I_x et I_y, les dérivées partielles de I, telles que

$$I(u+x, v+y) = I(u,v) + I_x(u,v)x + I_y(u,v)y$$

Ce qui donne l'approximation :

$$S(x,y) = \sum_u \sum_v w(u,v) \left(I_x(u,v)x + I_y(u,v)y\right)^2$$

Cette expression correspond à l'expression matricielles suivante :

$$S(x,y) = \begin{pmatrix} x & y \end{pmatrix} M \begin{pmatrix} x \\ y \end{pmatrix}$$

La matrice de Harris M est donnée par :

$$M = \sum_u \sum_v w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{2.2}$$

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 $\lambda_1 \approx 0$ et $\lambda_2 >> \lambda_1$, ou $\lambda_2 \approx 0$ et $\lambda_1 >> \lambda_2$, alors alors un contour est retrouvé.
3. Si λ_1 and λ_2 sont deux positives distinctes très grandes par rapport à 0 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 :

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

$$M = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix} ; \text{où :} S_{xx} = \left(\frac{\partial I}{\partial x}\right)^2 \otimes w, S_{yy} = \left(\frac{\partial I}{\partial y}\right)^2 \otimes w, S_{xy} = \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w \quad (2.3)$$

   Où $\otimes$ est l'opérateur de convolution

2. Construire la carte des coins en calculant la mesure de la fonction de réponse des coins $C(x, y)$ pour chaque pixel (x, y) :

$$C(x, y) = det(M) - k(trace(M))^2 \quad (2.4)$$

$$det(M) = S_{xx}.S_{yy} - S_{xy}^2$$
$$trace(M) = S_{xx} + S_{yy}$$

   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 2.1. 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 :

1. Seuillage de la carte d'intérêt en mettant toutes les valeurs de $C(x, y)$ inférieures à un seuil donné à zéro.
2. Extraction des maxima locaux en gardant les points qui sont plus grands que tous leurs voisins dans un voisinage 3×3.
3. Seuillage des maximas locaux à 10 % de la valeur du plus grand maximum

2.1.2 Détails de l'implémentation

Les images en niveaux de gris sont typiquement des données stockées dans des entiers 8-bit non signés et la sortie de l'algorithme de Harris est dans ce cas là un entier 32 bit afin

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

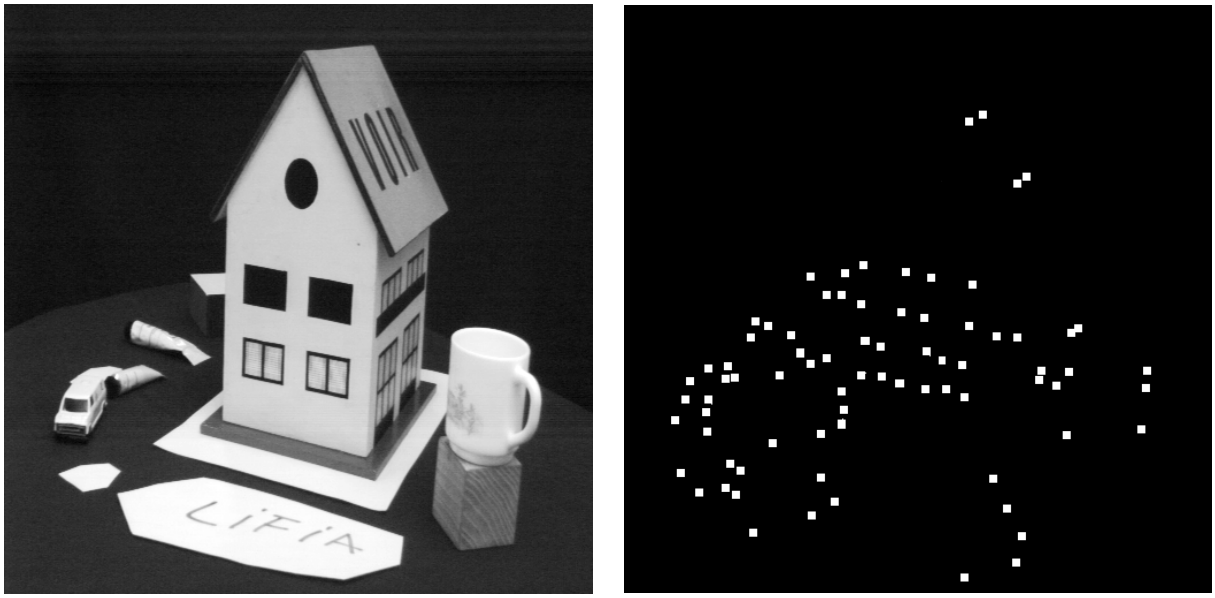

FIGURE 2.1 – Illustration de la détection de points d'intérêts sur une image niveaux de gris 512×512 avec $k = 0$

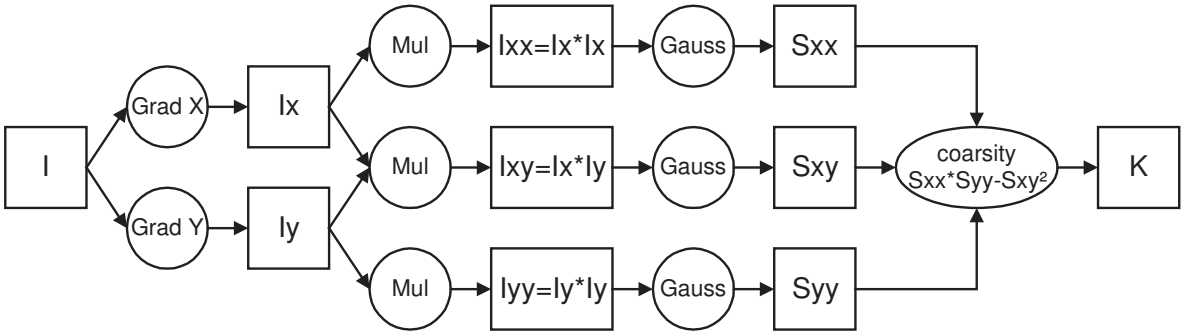

FIGURE 2.2 – Implémentation de l'algorithme de Harris sous forme de graphe flot de données

de satisfaire la dynamique des valeurs calculées qui font l'objet d'opérations de multiplication et d'addition tout au long de l'algorithme. Toutefois, pour des raisons de limitation du jeu d'instructions du SPU, et afin de garantir une comparaison directe avec les extension Altivec et SSE nous avons choisi le format flottant simple précision pour les données d'entrée et de sortie de l'algorithme. Dans notre implémentation nous avons divisé l'algorithme en 4 noyaux de traitement : un opérateur de *Sobel* qui calcule la dérivée dans les directions horizontale et verticale, un opérateur de multiplication, un noyau de lissage de *Gauss* (w dans l'équation 2.3), suivi d'un opérateur de calcul de réponse des coins. Nous avons fixé la constante k à

zéro ($k = 0$) (typiquement elle est fixée à 0.04) car ceci n'avait pas d'influence sur le résultat qualitatif et simplifie grandement l'équation 2.4 qui devient :

$$C(x, y) = S_{xx}.S_{yy} - S_{xy}^2$$

On obtient ainsi le graphe flot de données illustré par la figure 2.2 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 :

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

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

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

Étant donné 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 ils 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 problème limité par la mémoire (*memory-bounded problem*). C'est pour cette raison que les efforts d'optimisation sur l'algorithme de Harris sont faites par l'optimisation des accès à différents niveaux de la hiérarchie mémoire du processeur Cell.

2.2 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, on les 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 spécifiques à l'architecture particulière du Cell ont été employées également. Celles-ci ne sont généralement pas reproductibles 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 qui parfois résultent de contraintes de programmation spécifiques au Cell comme la taille limitée des mémoires locales des SPEs qui nécessitent une gestion logicielle explicite.

2.2.1 Techniques spécifiques au traitement d'images

Ces optimisations sont spécifiques au 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

Cette optimisation consiste à exploiter le fait que les noyaux de convolution 2D de *Sobel* et de *Gauss* sont séparables en deux filtres de convolution 1D (Fig. 2.3). Ainsi, la matrice des coefficients peut être exprimée comme un produit de deux vecteurs comme l'illustre les équations ci-dessous :

$$Grad_X = \frac{1}{16}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \frac{1}{16}\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

$$Grad_Y = \frac{1}{16}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{16}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

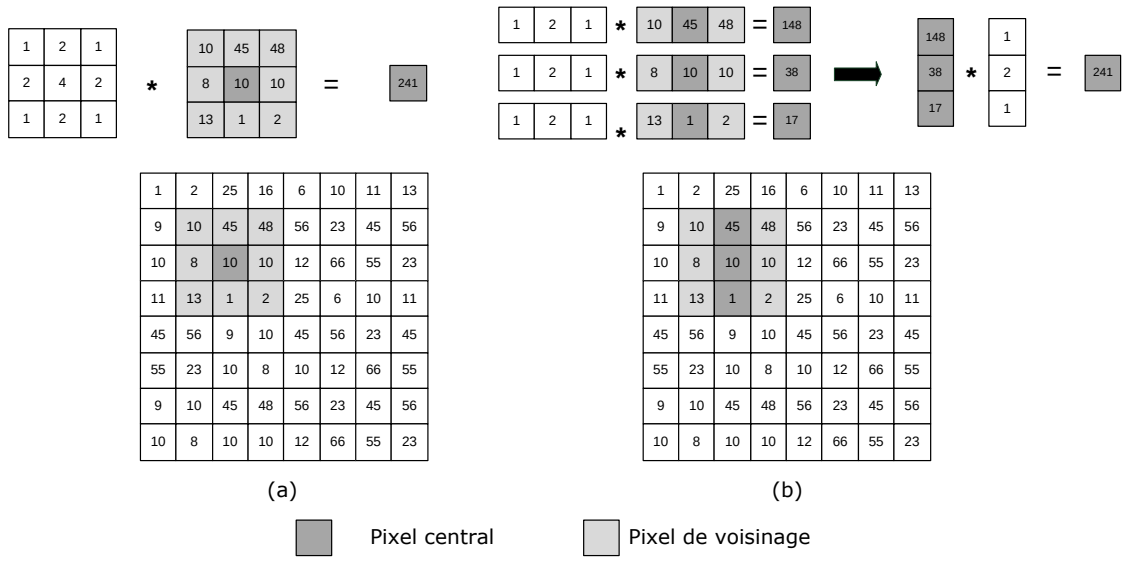

FIGURE 2.3 – Exemples de convolution par un filtre Gaussien 3×3 : (a) version avec noyaux 2D et (b) version avec deux noyaux 1D, résultant de la séparation du noyau 2D.

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

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'accès mémoire ainsi que la complexité arithmétique. La comparaison est illustrée dans les tableaux 2.1 et 2.2.

Chevauchement des noyaux

La deuxième particularité des noyaux de l'opérateur de convolution est une notion de chevauchement qui permet d'avoir une redondance d'une partie des données (Fig. 2.4). En effet, 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*.

Complexité arithmétique filtre de *Sobel*

Sans séparation du noyau			*Avec séparation du noyau*			*Gain*
MUL	ADD	**Total**	MUL	ADD	**Total**	**x1,75**
2	5	7	1	3	4	

Complexité arithmétique filtre de *Gauss*

Sans séparation du noyau			*Avec séparation du noyau*			*Gain*
MUL	ADD	**Total**	MUL	ADD	**Total**	**x2,16**
8	5	13	2	4	6	

TABLE 2.1 – 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**	**x1,16**
6	1	7	1	5	6	

Complexité mémoire filtre de *Gauss*

Sans séparation du noyau			*Avec séparation du noyau*			*Gain*
LOAD	STORE	**Total**	LOAD	STORE	**Total**	**x1,42**
9	1	10	6	1	7	

TABLE 2.2 – Réduction du nombre d'accès mémoire par décomposition des noyaux

Complexité mémoire filtre de *Sobel*

Sans chevauchement du noyau			*Avec chevauchement du noyau*			*Gain*
LOAD	STORE	**Total**	LOAD	STORE	**Total**	**x2,33**
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**	**x2,5**
9	1	10	3	1	4	

TABLE 2.3 – Réduction de la complexité mémoire par chevauchement des noyaux

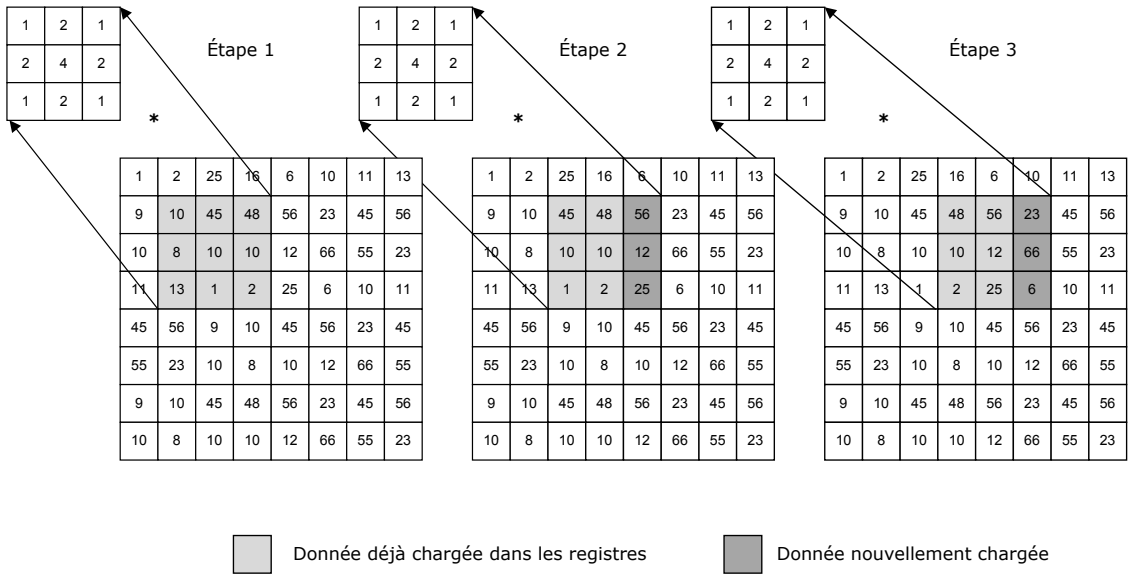

FIGURE 2.4 – Recouvrement de la fenêtre du filtre gaussien 3×3, certaines données sont conservées en décalant le masque de convolution.

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 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 2.4 et 2.5 donnent la différence en terme de complexité arithmétique et mémoire entre la version de base de l'algorithme et 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'ac-

Complexité arithmétique filtre de *Sobel*

Sans séparation du noyau + chevauchement			*Avec séparation du noyau + chevauchement*			*Gain*
MUL	ADD	**Total**	MUL	ADD	**Total**	**x1,75**
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**	**x2,16**
8	5	13	2	4	6	

TABLE 2.4 – Réduction de la complexité arithmétique par séparabilité et chevauchement des noyaux

Complexité mémoire filtre de *Sobel*

Sans séparation du noyau + chevauchement			*Avec séparation du noyau + chevauchement*			*Gain*
LOAD	STORE	**Total**	LOAD	STORE	**Total**	**x3,5**
6	1	7	1	1	2	

Complexité mémoire filtre de *Gauss*

Sans séparation du noyau + chevauchement			*Avec séparation du noyau + chevauchement*			*Gain*
LOAD	STORE	**Total**	LOAD	STORE	**Total**	**x5**
9	1	10	1	1	2	

TABLE 2.5 – Réduction de la complexité mémoire par séparabilité et chevauchement des noyaux

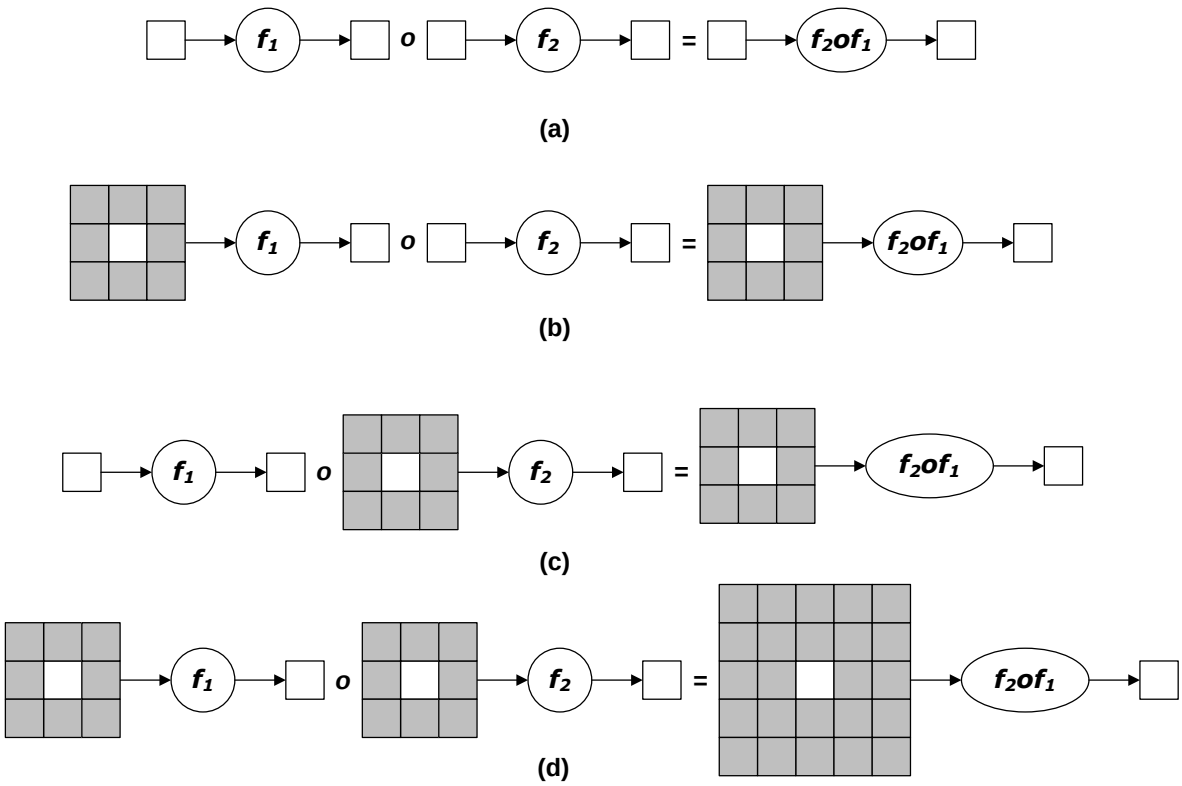

FIGURE 2.5 – Règle de composition de fonctions.**(a)** composition de deux opérateurs point à point **(b)** composition d'un noyaux de convolution suivi d'un opérateur point à point **(c)** composition d'un opérateur point à point suivi d'un noyaux de convolution **(d)** composition de deux noyaux de convolution successifs

cès mémoire puisque les opérations de sauvegarde et de chargement intermédiaires entre les deux fonctions initiales sont supprimées, les accès se font alors au niveau des registres. De plus, le coût de l'appel de fonction (sauvegarde dans 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 2.5. 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 ils s'enchainent. En terme d'apport de performances de cette technique, nous observons deux

aspects distincts :

- **Nombre d'accès mémoire** : on observe en effet que ceux-ci sont systématiquement réduits, car les opérations de `load/store` intermédiaires sont supprimées, sauf dans le cas de la composition de deux noyaux de convolution (cas **(d)** sur la figure 2.5)où la présence de bords supplémentaires augmente considérablement le nombre des accès mémoire.
- **Nombre d'opérations arithmétiques** : Celui-ci ne change pas dans le meilleur des cas notamment lorsque les opérateurs composés sont point à point ou alors lorsque l'opérateur de convolution est placé devant un opérateur point à point (cas **(b)** sur la figure 2.5). 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 2.5).

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. Elles peuvent même dans certains cas les dégrader. Le tableau 2.7 résume les complexité mémoire et arithmétique des différents cas de composition sur la figure 2.5 en incluant également les optimisations décrites auparavant.

2.2.2 Techniques spécifiques à l'architecture du processeur Cell

Dans cette partie, nous abordons des techniques d'optimisation qui sont spécifiques à notre architecture cible. On se limite dans l'étude aux seuls transferts de données présents dans l'application. Cette limitation s'explique par le fait que les applications que nous étudions sont souvent dominées par les transferts de données et où le ratio transfert/calcul est grand. Les transferts constituent donc le principal goulot d'étranglement dans nos algorithmes. On entend par transfert de données, toute communication qui met en jeu deux mémoires physiques sur le 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 réseau de communications [65] (*Network On Chip (NoC)*) du Cell et d'autres part la nature

Version de base sans composition				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel	2	6	1	14
Mul	3	2	1	9
Gauss	3	9	1	30
Coarsity	1	3	1	4
Harris	1	48	9	57
Version avec chevauchement et séparation des noyaux sans composition				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel	1	3	2	5
Mul	3	2	1	6
Gauss	3	3	1	12
Coarsity	1	3	1	4
Harris	1	21	9	30
Version avec composition de Mul∘Sobel et Coarsity∘Gauss				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel∘Mul	1	9	3	12
Gauss∘Coarsity	3	9	1	28
Harris	1	36	4	40

TABLE 2.6 – Tableau récapitulatif de l'optimisation des accès mémoire

Version avec chevauchement et séparation + composition de Mul∘Sobel et Coarsity∘Gauss				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel∘Mul	1	3	3	6
Gauss∘Coarsity	3	3	1	10
Harris	1	12	4	16
Version avec composition de Coarsity∘Gauss∘Mul∘Sobel				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel∘Mul∘Gauss∘Coarsity	1	25	1	26
Harris	-	-	-	26
Version avec chevauchement et séparation + composition de Coarsity∘Gauss∘Mul∘Sobel				
Opérateur	**Occurrences**	`LOAD`	`STORE`	**Total**
Sobel∘Mul∘Gauss∘Coarsity	1	5	1	6
Harris	-	-	-	6

TABLE 2.7 – Tableau récapitulatif de l'optimisation des accès mémoire

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

Taille du transfert Le premier paramètre influant sur le débit de transfert est la taille du bloc de données transféré. L'API de transfert limite les tailles d'un bloc transféré par une commande DMA à des valeurs de 1, 2, 4, 8 octets ou tout multiple de 16 octets et la taille de ce même bloc de données ne peut excéder 16 Ko si on veut le faire le transfert en une

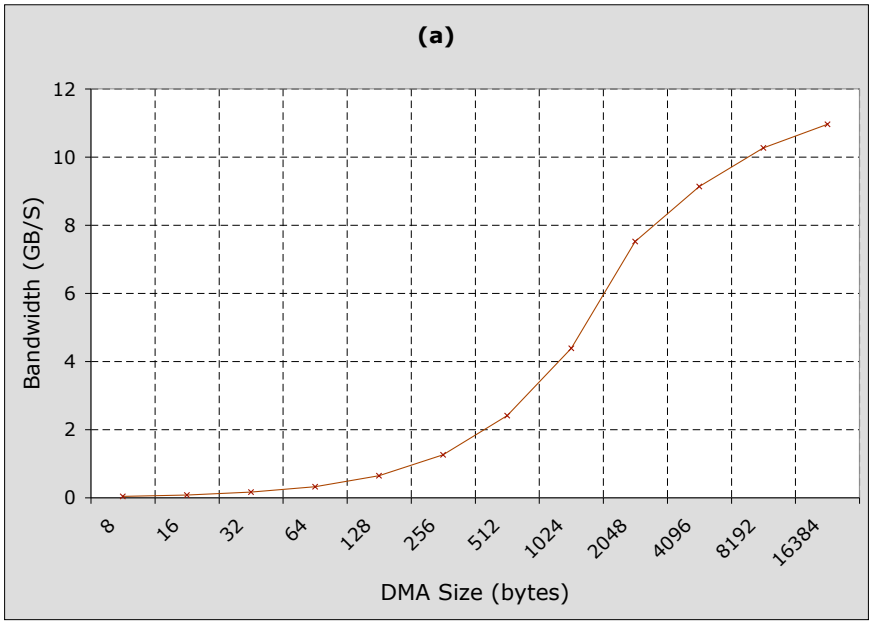

FIGURE 2.6 – Influence de la taille du transfert DMA sur la bande-passante

seule commande. De plus, les adresses doivent obligatoirement être alignées sur 16 octets et un alignement sur 128 octets est préférable à cause de la taille de la ligne de cache sur la mémoire locale du SPE qui est de 128 octets. D'autre part l'espace mémoire disponible sur les SPEs pour stocker données et instructions est limité à 256 Ko. Toutes ces contraintes, imposent d'accorder beaucoup d'attention à la taille de bloc transféré et à l'alignement des données qui ne sont pas du ressort du développeur dans les architectures à mémoire partagée. Plusieurs *benchmarks* ont été effectués dans [65] et [22] et qui démontrent la relation entre d'une part la taille et d'autre part le débit et la latence des transferts sur le Cell. Le graphe 2.6 donne les résultats sur un benchmark de bande-passante que nous avons effectué sur une BladeCenter QS20 et démontrent que celle-ci est d'autant plus grande que la taille des données transférées est importante. Ceci s'explique par la latence du transfert qui représente le temps d'initialisation d'un transfert, cette durée étant constante quelque soit la taille du paquet jusqu'à 16 Ko.

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

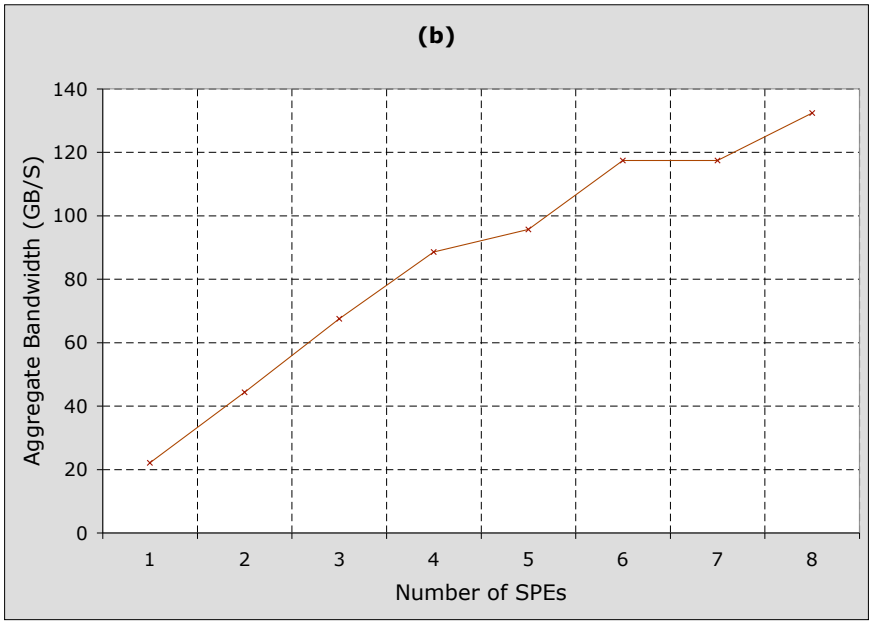

FIGURE 2.7 – Influence du nombre de transferts dans le cas d'une communication entre deux mémoires locales de deux différents SPEs

de 128 bits : deux dans le sens d'une aiguille d'une montre et les deux autres dans le sens opposé. En observant la topologie du réseau ainsi que le sens de circulation des données sur les anneaux du bus, on constate qu'il peut y avoir collision entre deux transferts s'exécutant de manière concurrente (Fig. 1.8). 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 lorsque le nombre de transferts concurrents augmente (au-delà de 12). Lorsqu'un tel conflit est détecté, l'arbitre de bus le résout avec un surcoût qui divise globalement la bande passante par deux. La courbe sur le figure 2.7 permet de constater l'influence du nombre de transferts concurrents sur le débit de transfert. On peut ainsi observer que la courbe perd sa linéarité au delà de 4 SPEs, ce qui correspond à un nombre de transferts 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 Go/s. Dans le cas où plusieurs SPEs font une requête vers la mémoire principale, les transferts ne peuvent être que sérialisés car il n'existe qu'une seule liaison vers le MS (*Main Storage*). On observe alors sur le graphique de la figure 2.8 que la

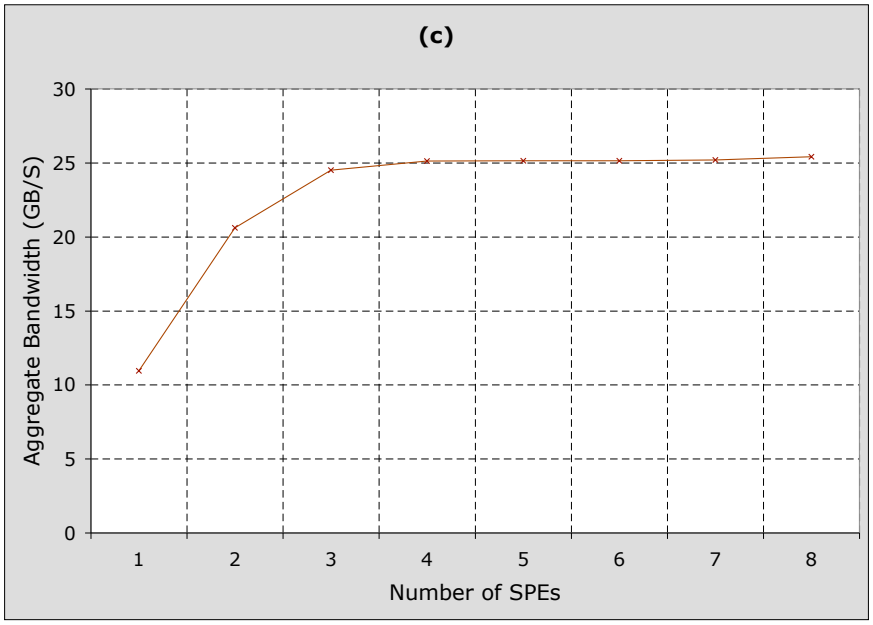

FIGURE 2.8 – Influence du nombre de transferts dans le cas d'une communication entre la mémoire externe et une mémoire locale de SPE

bande passante maximale n'est atteinte que lorsqu'au moins 4 SPEs font un transfert de la mémoire centrale vers leurs mémoires privées.

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 coûts 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 niveau des accès mémoires, sont les mêmes que pour la composition des opérateurs citée dans la section 2.2.1. Cette optimisation apporte beaucoup à la performance globale car l'application est caractérisée par un ratio transfert/calcul important et la différence de débit entre l'accès en mémoire locale et l'accès à une mémoire distante par DMA est d'un facteur dix [22].

Optimisation du tuilage des données

Le *loop tiling* [100] ou *loop blocking* est une technique d'optimisation très utilisée que cela soit par les programmeurs ou par les compilateurs optimisants[62]. Cette technique consiste en un découpage des données à différents niveaux de la hiérarchie mémoire, de telle sorte que la latence d'accès soit la plus petite possible. Dans une architecture à 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 inexistants 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 privées des SPEs. D'autre part l'espace de stockage est limité dans les mémoires locales des SPEs à 256 Ko. Ceci rend le découpage des données en tuiles pouvant tenir dans le cache obligatoire. L'unité de données atomique devient alors la tuile qui, représente le morceau de données le plus petit traité 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 de la mémoire locale 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 Ko. 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 chaine de traitement mise en jeu. D'autre part, selon ce qui a été vu précedemment, la taille de transfert qui garantit une bande-passante maximale sur le bus est de 16 Ko ou un multiple de cette taille.

Forme 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

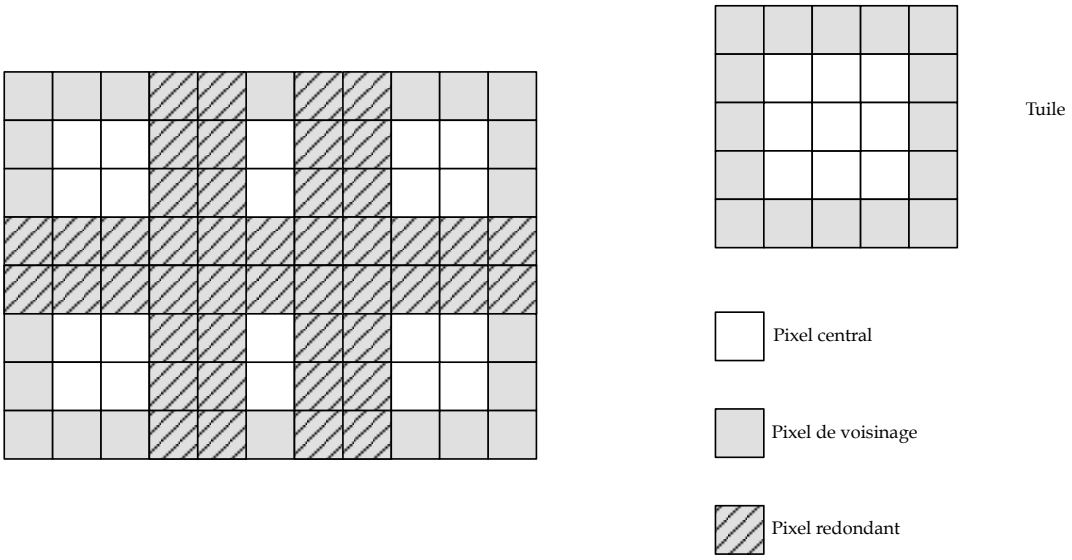

FIGURE 2.9 – Redondances de données pour un opérateur de convolution 3×3

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 2.9 certains des pixels sont rechargés. Nous étudions dans la suite l'influence des dimensions de la tuile sur ce nombre de pixels redondants.

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 3×3 pour chaque pixel. Afin de simplifier le calcul, on suppose que la matrice de convolution est carrée, ce qui fait que les bords des deux côtés sont égaux. La quantité totale de pixels transférés 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 et b la taille du bord en pixel.

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

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

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

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

Le minimum de la fonction Q est atteint lorsque : $\frac{\partial Q}{\partial h} = 0$ 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 2.10 on peut constater que cette fonction décroit quand h et w augmentent et que la valeur de Q est minimale pour une surface de tuile donnée ($h \times w = constante$) lorsque $h = w$ ce qui correspond à la première bissectrice du plan (hw). Ceci permet d'affirmer 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 l'implémentation. En effet, des tuiles de forme carrée signifient des accès à des zones non-contigues de la mémoire. Ce type d'accès est en général coûteux 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 nécessite 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

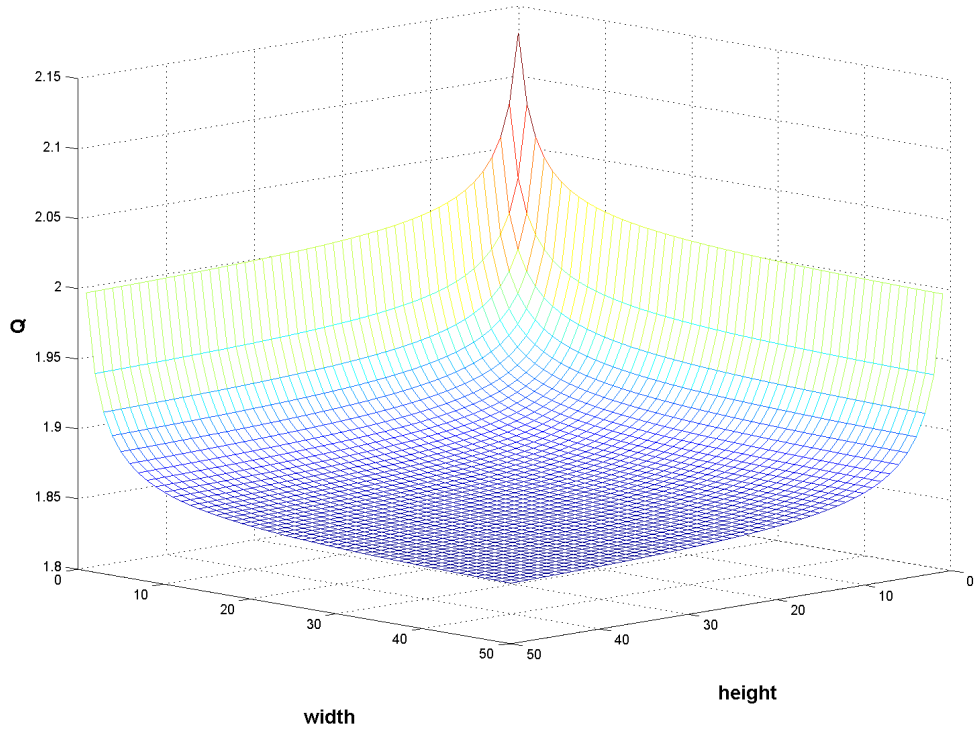

FIGURE 2.10 – Tracé de la fonction $Q(h, w)$ dans l'espace

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 en l'utilisation de tuiles ayant une largeur égale à celle de l'image. Ceci permet des accès sur des zones contigues de la mémoire et des transferts pouvant se faire en 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 à 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 taille d'images très grandes.

h_{tuile}	w_{tuile}	**temps total (sec)**
8	512	0.0494
16	256	0.0598
32	128	0.0485
64	64	0.0345
128	32	0.0517
256	16	0.0699
512	8	0.0734

TABLE 2.8 – Performances sur une image 512×512

Résultats expérimentaux Nous avons procédé à quelques expérimentations sur l'influence de la forme de la tuile. Cette étude à fait l'objet d'une publication dans [93]. Le but étant de valider notre implémentation en considérant plusieurs formes de tuiles, et en vérifiant la conformité des résultats obtenus avec l'étude théorique. Le programme s'exécute à partir du PPE qui utilise un seul SPE pour le calcul. Pour chaque image, nous avons choisi une taille de tuile fixe i.e : un volume de données transférées constant et avons fait varier la forme de la tuile. Nous observons d'après les tableaux 2.8, 2.9, 2.10 et 2.11 que la tuile qui se rapproche le plus de la forme carrée possède les meilleures performances globales. La différence de performances entre deux tuiles de tailles proches est marginale mais dans un cas réaliste où le flux d'images est continu, cette différence est plus importante. Pour les tailles d'images très grandes (Tab. 2.10 et 2.11) on constate que l'amélioration est de l'ordre de 50% en comparaison avec des tuiles qui ont la largeur de l'image. Il faut noter qu'il existe un surcoût à l'utilisation de tuiles carrées. En effet, une telle forme nécessite l'utilisation des DMA listes. Le compromis se situe dans ce cas, entre la redondance de données et les accès mémoire irréguliers. Les résultats expérimentaux montrent que les tuiles carrées donnent de meilleurs performances malgré ce compromis.

2.2.3 Schémas de parallélisation

Dans ce qui suit, nous abordons l'optimisation de notre algorithme à un niveau d'abstraction plus haut, 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 diffé-

h_{tuile}	w_{tuile}	**temps total (sec)**
8	512	0.198
16	256	0.238
32	128	0.187
64	64	0.110
128	32	0.180
256	16	0.218
512	8	0.352

TABLE 2.9 – Performances sur une image 2048 $\times$ 512

h_{tuile}	w_{tuile}	**temps total (sec)**
5	1200	0.494
10	600	0.360
20	300	0.264
40	150	0.235
80	75	0.183
160	37	0.247
320	18	0.275

TABLE 2.10 – Performances sur une image 1200 $\times$ 1200

h_{tuile}	w_{tuile}	**temps total (sec)**
8	512	0.985
16	256	0.726
32	128	0.643
64	64	0.438
128	32	0.692
256	16	0.866
512	8	1.422

TABLE 2.11 – Performances sur une image 2048 $\times$ 2048

rents 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. *MS* (*Main Storage*) désigne la mémoire centrale , alors que *LS* (*Local Store*) désigne une mémoire locale de SPE. Les flèches à trait fin représentent des instructions `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 la mémoire locale d'un SPE. La lettre S représente le gradient de *Sobel* en x et y combinés. La lettre G représente le noyau de lissage de *Gauss*. La lettre M représente une multiplication point à point et la lettre H le calcul de la coarsité. Si des chiffres précèdent les lettres citées auparavant, ils correspondent au nombre d'occurrences de l'opérateur. Par exemple, $3M$ représente trois instances de l'opérateur de multiplication. Les différents placements se basent sur le graphe de référence de la figure 2.2.

Data parallel : Fig. 2.11

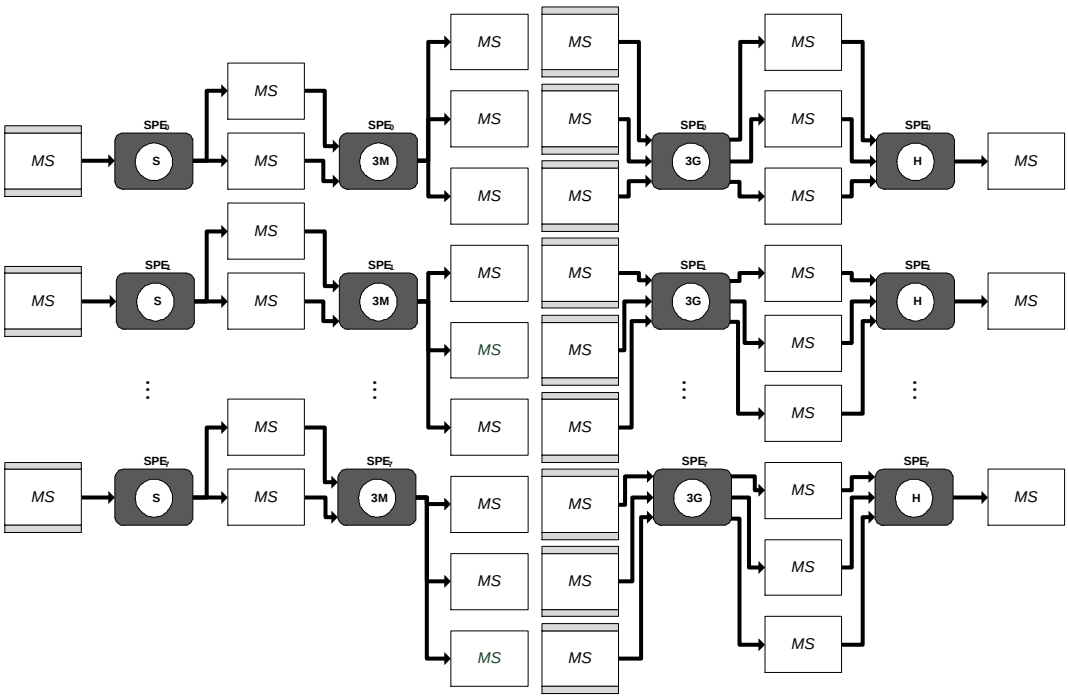

FIGURE 2.11 – Schéma de parallélisation SPMD conventionnel

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 indépendante les unes des autres. Toutefois, la bande passante sur le bus mémoire centrale vers le *local store* est sollicitée de manière importante, car les données sont systématiquement lues et écrites avant et après chaque opérateur.

Pipeline : Fig. 2.12

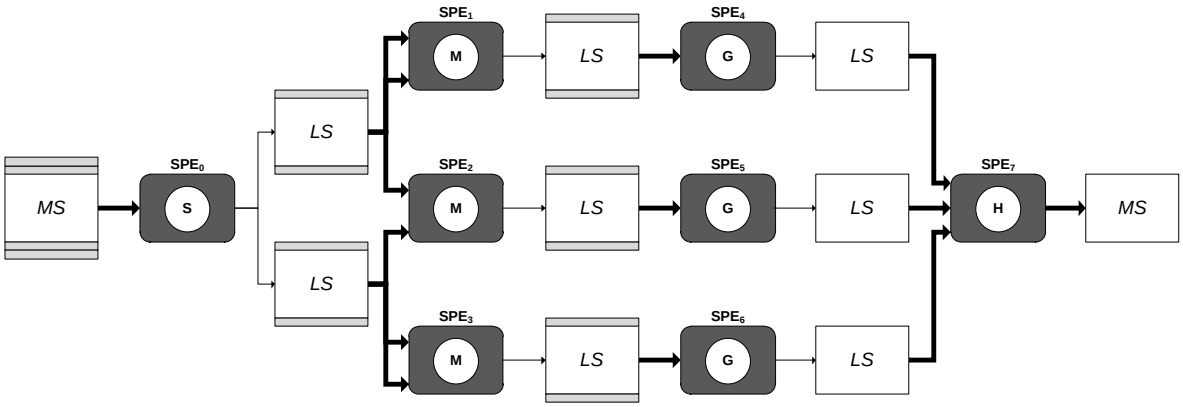

FIGURE 2.12 – Schéma de parallélisation pipeline

Cette implémentation de l'algorithme consiste à déployer le graphe d'opérateurs 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 sérialisé car la possibilité d'exécuter des *threads* concurrents y est réduite à cause des dépendances de données introduites dans le graphe. Par contre, la bande-passante inter-SPEs est bien exploitée car la majorité des transferts se font entre SPEs et par conséquent la pression exercée sur le bus précédemment est atténuée car elle est répartie sur l'ensemble de l'anneau.

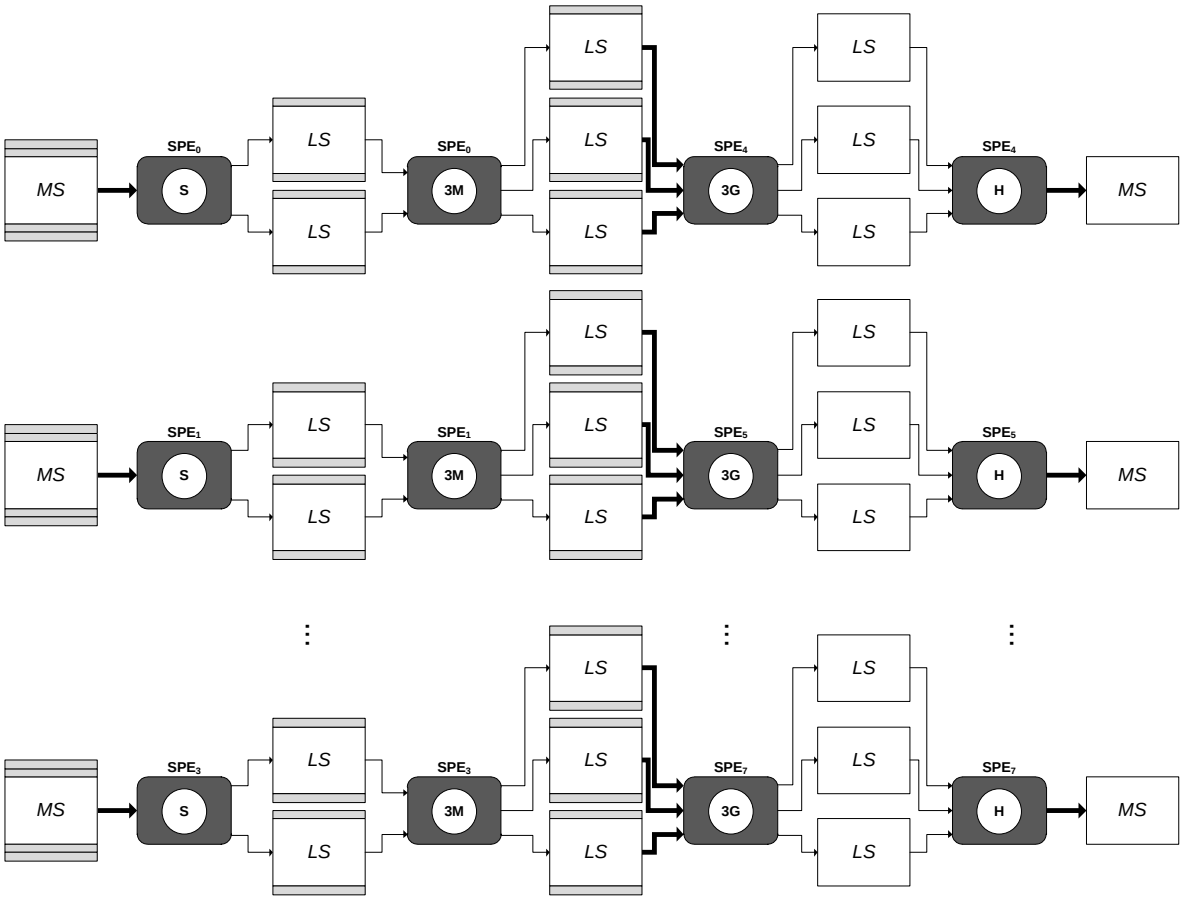

FIGURE 2.13 – Schéma de parallélisation chaînage d'opérateurs par paires (*Halfchain*)

Chainage d'opérateurs par paires (*Halfchain*) : Fig. 2.13

Dans cette version deux opérateurs successifs sont placés sur le même SPE. Ainsi, les opérateurs *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.

Chainage et fusion d'opérateurs par paires (*Halfchain+Halfpipe*) : Fig. 2.14

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`

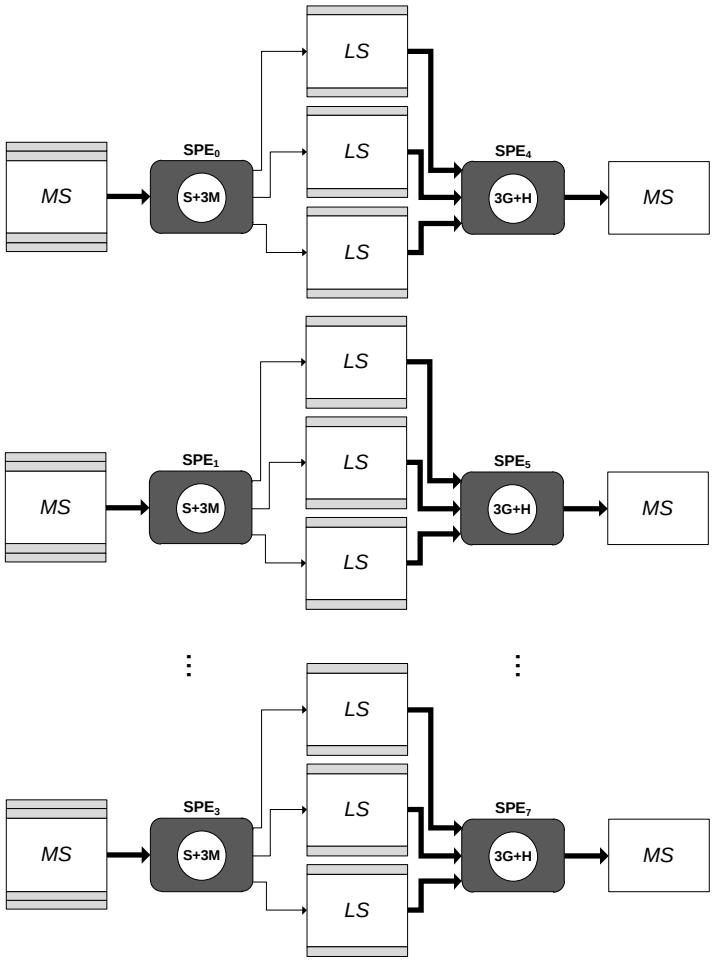

FIGURE 2.14 – Schéma de parallélisation chaînage et fusion d'opérateurs par paires (*Halfchain+Halfpipe*)

présents à la sortie 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 [53].

Chaînage complet et fusion d'opérateur par paires (*Halfchain + Fullpipe*) : Fig. 2.15

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

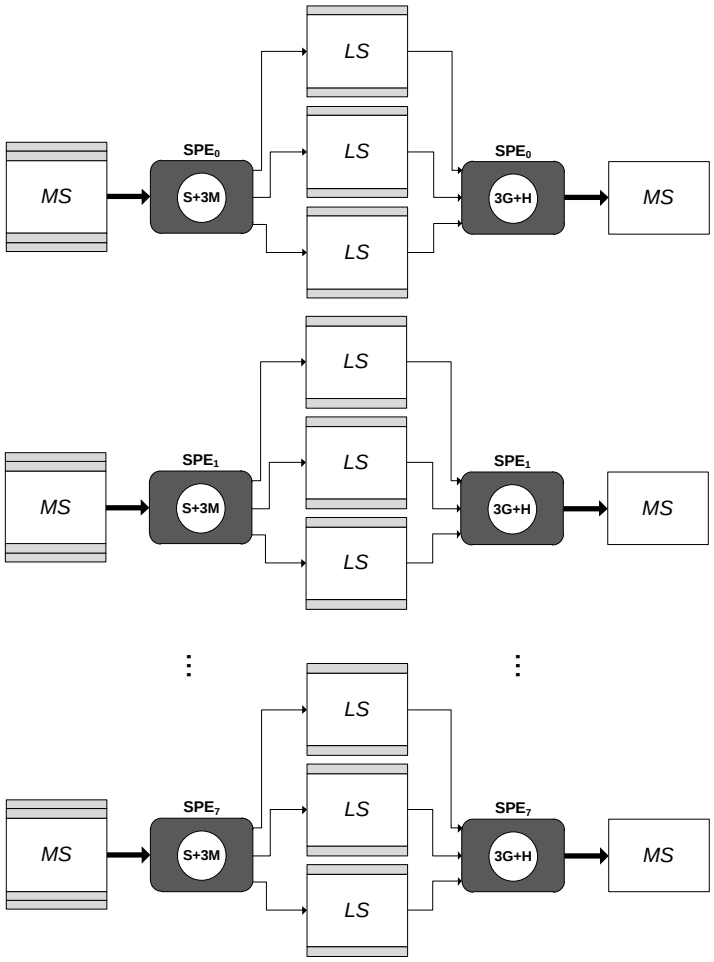

FIGURE 2.15 – Schéma de parallélisation chaînage et fusion d'opérateurs par paires (*Halfchain + Fullpipe*)

2.3 Évaluation des performances

Après avoir exposé les différentes techniques d'optimisation, nous procédons à la mesure des performances pour les différents modèles de déploiement qui ont été mis en oeuvre. Il s'agit ici de comparer les implémentations en terme de temps d'exécution sur le processeur Cell. Nous évaluons également l'influence des transferts sur la performance globale de chacun des modèles.

2.3.1 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 :

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

Le nombre de cycles cpu est mesuré à l'aide de primitives spécifiques $PowerPC$ et qui permettent de lire des registres des compteurs CPU. Les bancs de tests ont été conçus de sorte à mesurer 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*). Ces deux quantités ont été mesurées en se basant sur la loi d'Amdahl [4]. Nous avons choisi ce modèle pour sa simplicité, d'autres modèles plus précis qui prennent en compte l'équilibrage de charge et le surcout existent tels que celui de Karp et Flatt [61], mais nous ne les avons pas utilisés dans le cadre de cette étude.

2.3.2 Méthode et plateforme de mesure

La plate-forme sur laquelle a été menée l'évaluation des performance est un Blade Server 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 Go de *RAM* disponible. Dans notre expérimentation, un processeur Cell sur deux est utilisé. Le compteur 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 à 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 en ajoutant une boucle externe autour de l'implémentation sur une image et avons mesuré le nombre de cycles moyen pour une image. Ceci permet de négliger le surcoût 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.

2.3.3 Comparaison des schémas de parallélisation

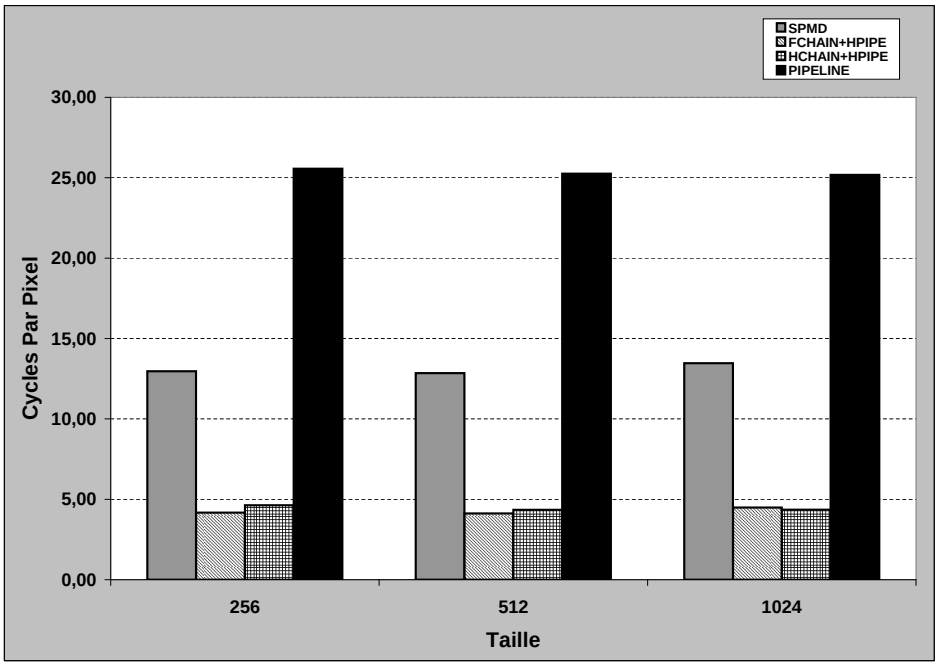

FIGURE 2.16 – Comparaison des modèles en cycles par point : SPMD correspond à la version data parallel, PIPELINE au pipeline, HCHAIN+HPIPE au chainage d'opérateurs par paires et HCHAIN+HPIPE au chaînage complet et fusion d'opérateurs par paires

La figure 2.16 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 résultat était attendu car cette version n'exploite pas entièrement le parallélisme de données offert par l'architecture du Cell. Les autres résultats correspondent également à nos attentes :

- 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 dans lesquelles les DMA inter-SPE sont utilisés possèdent de bonnes performance : à titre d'exemple la version sans composition de fonction et en regroupant deux opérateurs par SPE est plus rapide que la version *data parallel*.
- De plus, la version avec composition de fonction où 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 un 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, 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* imbriquée 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 niveau des transferts. En effet, en favorisant les communications inter-SPE on profite pleinement de la bande passante du bus interne (204.8 Go/s) qui peut gérer jusqu'à 12 transferts en parallèle, alors que dans un schéma de transfert 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 Go/s.

2.3.4 Influence de la taille de la tuile

Comme il est démontré dans [22] et [65] 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 transfert/calcul important. La figure 2.17 nous montre que la taille de tuile donnant les meilleures performances est 16Ko, et ceci pour deux raisons :

- Une taille de bloc de 16 Ko 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 la 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.

2.3.5 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 est le facteur principal

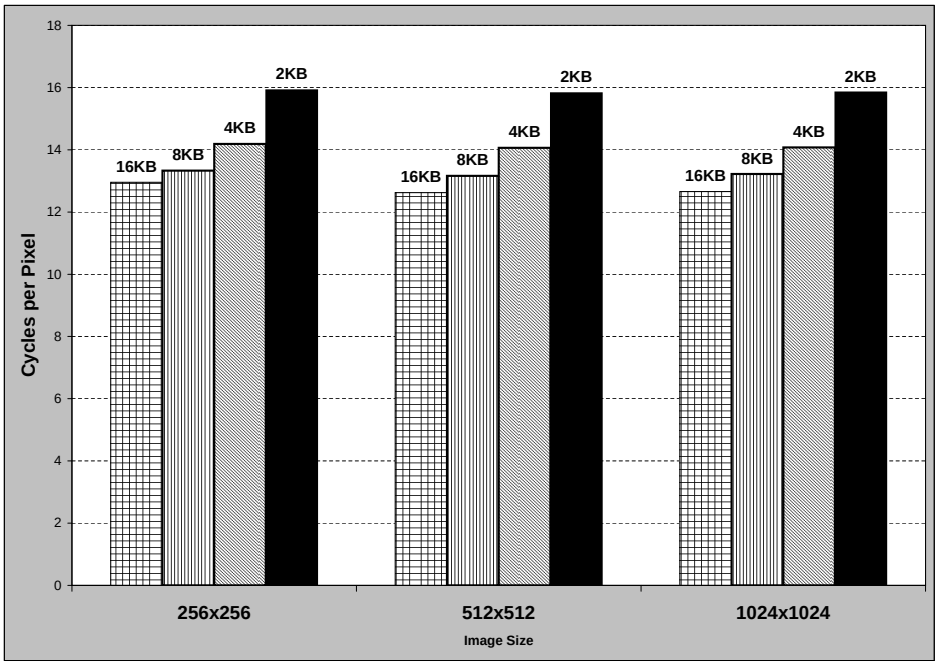

FIGURE 2.17 – Influence de la taille de la tuile sur la performance pour la version chainage entier et fusion d'opérateurs par paires 1 SPE (Fig. 2.15)

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 2.12 donne en nombre de cycles *SPU* les versions ou les opérateurs ne sont pas composés deux à deux et celle ou ils 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

Modèle	Opérateur	Nombre de Cycles	Accélération
Halfchain	Sobel+Mul+`LOAD/STORE`	119346	1
Halfchain+Halfpipe	Gauss+Coarsity+`LOAD/STORE`	188630	1
Halfchain	(Sobel $\circ$ Mul)+`LOAD/STORE`	16539	7.2
Halfchain+Halfpipe	(Gauss $\circ$ Coarsity)+`LOAD/STORE`	504309	3.5

TABLE 2.12 – Apport de la composition de fonction à la performance, les versions de référence sont celles ayant une accélération de 1

taille	*p*	*temps (sec)*	*s*	*e*	*f*
256×256	1	0.612	-	-	-
512×512	1	2.442	-	-	-
1024×1024	1	9.765	-	-	-
256×256	2	0.308	1.987	0.994	0.006
512×512	2	1.224	1.995	0.997	0.003
1024×1024	2	4.890	1.997	0.999	0.001
256×256	4	0.160	3.827	0.957	0.015
512×512	4	0.628	3.888	0.972	0.010
1024×1024	4	2.504	3.900	0.975	0.009
256×256	8	0.086	7.116	0.889	0.018
512×512	8	0.323	7.557	0.945	0.008
1024×1024	8	1.272	7.677	0.960	0.006

TABLE 2.13 – Mesure des métriques de passage à l'échelle pour la version FCHAIN+HPIPE

banc de registres du processeur (128 registres pour le SPE). 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 la mémoire locale, limité lui à 256 Ko.

2.3.6 Mesure des métriques de passage à l'échelle

Lorsque l'on mesure les performance 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 p unités par rapport à une seule. Les métriques les plus connues sont l'accélération (*Speedup*) et l'efficacité (*Efficiency*), et sont définies comme suit :

$$s = \frac{\text{Temps d'Exécution sur 1 Processeur}}{\text{Temps d'Exécution sur } p \text{ Processeurs}}$$

$$e = \frac{\text{Temps d'Exécution sur 1 Processeur}}{p \times \text{Temps d'Exécution sur } p \text{ Processeurs}}$$

taille	*p*	*temps (sec)*	*s*	*e*	*f*
256×256	1	1.628	-	-	-
512×512	1	6.509	-	-	-
1024×1024	1	26.236	-	-	-
256×256	2	0.821	1.983	0.992	0.011
512×512	2	3.282	1.983	0.992	0.008
1024×1024	2	13.221	1.984	0.992	0.008
256×256	4	0.432	3.770	0.942	0.020
512×512	4	1.706	3.815	0.954	0.016
1024×1024	4	6.872	3.818	0.954	0.016
256×256	8	0.266	6.124	0.776	0.044
512×512	8	1.035	6.124	0.766	0.044
1024×1024	8	4.151	6.320	0.790	0.038

TABLE 2.14 – Mesure des métriques de passage à l'échelle pour la version SPMD

Karp et Flatt définissent également la portion séquentielle expérimentale dans [61] par :

$$f = \frac{1/s - 1/p}{1 - 1/p}$$

Ces mesures sont essentielles pour l'adaptation d'un code séquentiel à une machine parallèle. Elles permettent d'une part de détecter les 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 d'interconnexion dans la distribution des données de manière parallèle aux différents noeuds. Elles peuvent également justifier l'ajout d'unités de traitement supplémentaires si cela n'altère pas la performance en induisant un coût de gestion supplémentaire.

D'après les tableaux 2.13 et 2.14 on peut constater que notre implémentation de l'algorithme de Harris passe bien à l'échelle sur l'architecture du Cell. En effet, on peut noter dans les deux cas étudiés à la fois, une accélération 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 version avec composition de fonctions et chainage sur le même SPE. La première implique une pression importante sur le bus de données car presque toutes les lectures/écritures mémoire se font sur la mémoire externe.

Au contraire, la seconde version 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 et moins le passage à l'échelle est bon. La mesure de la fraction séquentielle f conforte les remarques citées dans ce qui précède. En effet, la fraction séquentielle croit avec le nombre de processeurs p ce qui s'explique par le surcoût induit par un plus grand nombre de *threads*. D'autre part, f est plus important pour la version SPMD car la pression sur le bus d'interconnexion est plus importante ce qui provoque une diminution de la bande passante et par conséquent une sérialisation plus importante des communications.

2.4 Conclusion

Dans ce chapitre nous avons mis en oeuvre la parallélisation d'un code de traitement d'images sur le Cell. L'algorithme qui a été choisi est le détecteur de points d'intérêts de *Harris* qui est à la fois représentatif des traitements bas-niveau, présent dans plusieurs traitements plus complexes et permet plusieurs schémas de parallélisation et d'optimisation. Plusieurs techniques d'optimisation on été présentées : certaines sont spécifiques au traitement du signal et des images et sont applicables à d'autres architectures. D'autres techniques sont plus spécifiques et relèvent de l'adéquation entre l'algorithme et l'architecture spécifique du processeur Cell. L'accent à été mis sur l'optimisation des transferts mémoire à plusieurs niveaux et de plusieurs manières, car elles constituent le principal facteur limitant la performance pour notre domaine d'application. Plusieurs squelettes de parallélisation ont été mis en oeuvre afin d'avoir une idée globale de squelette de parallélisation qui donne les meilleures performances. Les performances ont été mesurées de plusieurs manières et à différents niveaux : au niveau des noyaux de calcul une estimation et une mesure des gains apportés par les optimisation bas-niveau ont été effectuées. Au niveau des transferts, une mesure de la bande passante fut menée en fonction du schéma de parallélisation, et l'influence de la taille et de la forme de la tuile de calcul ont été étudiées. Enfin, des métriques de passage à l'échelle (*Scalability*) ont été utilisées afin de mesurer l'efficacité de la parallélisation de notre algorithme sur le processeur Cell. Les performances obtenues lors de cette étude ont été obtenues après un temps de développement important. Elles sont le fruit d'une méthodologie d'optimisation itérative et de l'acquisition d'une expertise sur l'architecture du Cell. Dans le cadre de développement

d'applications industrielles, des contraintes de temps de livraison et de maintenabilité du code se posent. L'outil de base n'est pas adapté pour répondre à de telles contraintes. C'est pour ces raisons que des outils d'aide au portage de code sur le processeur Cell ont vu le jour.

Chapitre 3

Outils de programmation haut-niveau pour le Cell

Au vu des contraintes de programmation pour l'obtention de performances sur le processeur Cell et que nous avons mis en avant dans le chapitre précédent, plusieurs outils de programmation dédiés au processeur Cell ont vu le jour pour faciliter le passage d'un code série à un code totalement parallélisé sur cette architecture. Dans ce chapitre, nous faisons une analyse de ces outils de programmation pour le processeur Cell. L'étude comporte des outils industriels tels que ***RapidMind*** et***OpenMP*** ainsi que des outils développés par des laboratoires de recherche académiques : ***Sequoia*** et ***CellSS***. 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'application. Certains outils reprennent d'autres outils existants pour les architectures parallèles à mémoire partagée ou distribuée qui ont été adaptés pour l'architecture et la hiérarchie mémoire du Cell.

3.1 *RapidMind*

RapidMind[73] est un modèle de programmation parallèle multi-plateformes, GPU, multi-coeur symétrique et pour le processeur Cell. Il relève du modèle de programmation *stream processing*[60] et s'apparente à un langage de programmation enfoui dans *C++*. Il est basé sur la bibliothèque template *C++* et une bibliothèque de support d'exécution qui effectue la génération dynamique de code. La bibliothèque template permet l'invocation de code SPE à

l'intérieur du code PPE, avec l'ensemble du code SPE écrit en template.
La bibliothèque 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 bibliothèque de support d'exécution de capturer une représentation du code SPE. Les types de données ont été spécialement conçus pour exprimer de manière simple les opérations SIMD et les exposer facilement à la bibliothèque du support d'exécution. Le support d'exécution 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. le support d'exécution ***RapidMind*** assigne des tâches aux SPEs de manière dynamique 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 bien que étant initialement un modèle *data-parallel*. Un exemple de code ***RapidMind*** est donné dans le listing 3.1.

3.1.1 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 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, ou 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 usuelle comme le `Value4f` pour un quartet de flottants simple précision 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 on été ajoutées au type `Value` : ces opérations permettent la duplication d'une composante ou la permutation des composantes. 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 et 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 accèdés 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 valeur 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 le même type d'optimisations sur les calculs vectoriels. Les fonctionnalités du langage *C++* sont utilisées pour structurer les calculs et générer le code mais pas lors de l'exécution.

3.1.2 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.

```
1
2  // Reference Code
3  float f;
4  float a[512][512][3];
5  float b[512][512][3];
6  float func (float r, float s)
7  {
8    return (r + s) * f;
9  }
10 void func_arrays( )
11 {
12   for (int x = 0; x<512; x++)
13     for (int y = 0; y<512; y++) {
14       for (int k = 0; k<3; k++) {
15         a[y][x][k] = func(a[y][x][k],b[y][x][k]);
16       }
17     }
18   }
19 }
20
21 // RapidMind Code
22 #include <rapidmind/platform.hpp>
23 Value1f f;
24 Array<2,Value3f> a(512,512);
25 Array<2,Value3f> b(512,512);
26 void func_arrays( )
27 {
28   Program func_prog = BEGIN { // define program
29     In<Value3f> r, s;
30     Out<Value3f> q;
31     q = (r + s) * f;
32   } END;
33   a = func_prog(a,b); // execute program
34 }
```

Listing 3.1 – Exemple de parallélisation de code avec ***RapidMind***

En premier lieu, les `Program` peuvent être évalués partiellement. Si un objet `Program` **p** possède **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 lier 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 lier 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, on crée "**p(A)**" et on l'affecte à un objet `Program` "**q**", puis 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 liée.

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 d'un `Program` peut être liée à un `Array` ou un tuple de `Value`. Si des tableaux de tailles différentes sont liés 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 et 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 d'accès mémoire.

3.1.3 Les opérations collectives

L'opération de base supportée est l'application parallèle d'un programme sur un tableau de données. Toutefois, d'autres schémas de communication et de calcul sont supportés sous la forme d'opérations collectives. Les patrons de communication irréguliers sont fournis par les opérations de *scatter* et *gather*,et l'opération de réduction fournit un patron 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 quant 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. 3.1).

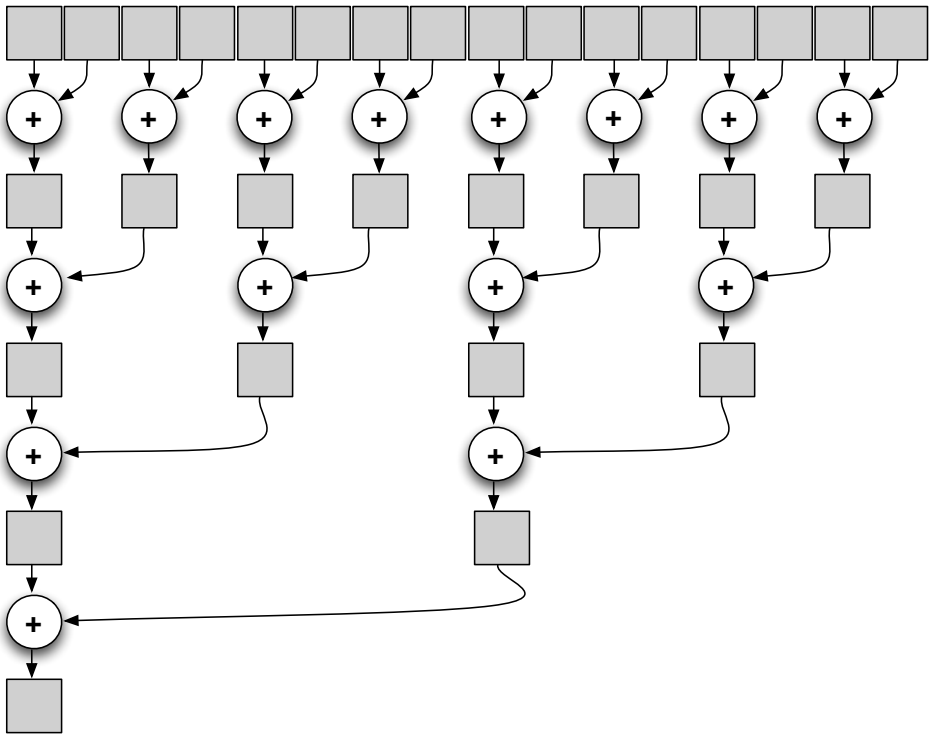

FIGURE 3.1 – Illustration d'une opération collective de réduction, ici une opération de sommation est effectuée, le résultat final est l'accumulation des éléments du tableau

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**.

3.1.4 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. En effet, les unités de calcul des SPEs étant purement vectorielles, la génération de code est rendue difficile à cause notamment des instructions de permutation additionnelles nécessaires à l'accomplissement du calcul. De plus 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 subdivisée 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 cache logiciel 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.

3.1.5 Conclusion sur *RapidMind*

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 bons résultats dans [73].
La société ***RapidMind*** a été rachetée par Intel en 2009 et l'outil de parallélisation se nomme désormais *Array Building Blocks*[78] qui fait partie de la suite Intel Parallel Studio.

3.2 *OpenMP* pour le Cell

OpenMP pour le Cell[80] intégré dans le compilateur XL d'IBM[33] est basé sur les transformations du compilateur et une bibliothèque de support d'exécution. 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 bibliothèque de support d'exécution du Cell. Cette dernière fournit des fonctionnalités basiques à ***OpenMP*** : 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 3.2, page 96.
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 bibliothèque de support d'exécution ***OpenMP*** dans la fonction parente de la fonction listée. Ces appels aux fonctions de la bibliothèque de support d'exécution 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 supportant ***OpenMP***, mais la spécificité de l'architecture du Cell a posé quelques problèmes qui sont les suivants :

- ***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 cache logiciel qui fait partie de la bibliothèque de support d'exécution du SPE.

- **Modèle mémoire** : le hardware du Cell assure que les transactions DMA sont cohérentes vis à vis du cache L2 du PPE, mais ne fournit pas de mécanisme de cohérence

pour les données résidant dans la mémoire locale, le modèle mémoire implémenté assure la cohérence des données qui est requise par les spécifications.

```
1  #include <omp.h>
2  #define CHUNKSIZE 100
3  #define N      1000
4
5  main ()
6  {
7
8    int i, chunk;
9    float a[N], b[N], c[N];
10
11   /* Some initializations */
12   for (i=0; i < N; i++)
13     a[i] = b[i] = i * 1.0;
14   chunk = CHUNKSIZE;
15
16 #pragma omp parallel shared(a,b,c,chunk) private(i)
17   {
18
19 #pragma omp for schedule(dynamic,chunk) nowait
20     for (i=0; i < N; i++)
21       c[i] = a[i] + b[i];
22
23   } /* end of parallel section */
24
25 }
```

Listing 3.2 – Exemple de code **OpenMP** où une boucle for est parallélisée

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

3.2.1 *threads* et synchronisation

Les *threads* peuvent s'exécuter sur le PPE ou les SPEs. Le *thread* maitre 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 explique le 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, et 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 bibliothèques du SDK (Software Development Kit). Le *thread* sur le PPE crée des *threads* SPE à l'exécution seulement, quand les structures parallèles sont rencontrées pour la première fois.
Pour les niveaux de parallélisme imbriqué (boucles imbriquées), 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 (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* maitre 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* maitre 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 bloquants ou non-bloquants entre le *thread* maitre et les *threads* de calcul. Les *locks* ***OpenMP*** sont implémentés grâce aux commandes DMA atomiques.

3.2.2 Génération de code

En premier lieu, le compilateur sépare chaque région dans le code source qui correspond à une construction ***OpenMP*** parallèle, et l'isole 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 bibliothèque de support d'exécution ***OpenMP*** au niveau de la fonction parente de la fonction listée, et insère un pointeur dans cette fonction

de la bibliothèque de support d'exécution. Le compilateur insère également des instructions de synchronisation lorsque celle-ci est nécessaire.

Le fait que l'architecture du Cell soit hétérogène impose que les fonctions isolées 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 isolée 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 isolées de telle sorte à ce qu'il n'y ait pas de confusion lors de l'exécution.

A la fin de l'étape d'optimisation haut-niveau (TPO), les procédures sur les différentes architectures sont séparées en deux unités de compilation différentes et celles-ci sont traitées une par une, par le *back-end* TOBEY[11].

L'unité PPE ne requière pas de traitement particulier. Par contre l'unité compilée SPE peut produire un binaire d'une taille importante et qui ne tient 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 par ce qu'une sous-section peut ne pas avoir une taille 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 crée 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 de l'arbre couvrant maximum [84] (*maximum spanning tree*). Le code SPE ainsi généré est intégré dans le code PPE avant d'être exécuté.

3.2.3 Modèle mémoire

OpenMP spécifie un modèle à mémoire partagée à cohérence faible (*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 : la bufferisation statique et le cache logiciel 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 cache logiciel est utilisé. Le compilateur remplace les `load` et `store` à partir 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 octets) 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.

3.2.4 Conclusion sur *OpenMP* pour le Cell

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 du Cell. Le support d'***OpenMP*** est assuré par des transformations du compilateur couplées à une bibliothèque de support d'exécution. Les problématiques qui sont posées pour le portage d'***OpenMP*** sur le Cell sont notamment celles de la synchronisation des *threads*, de la génération de code et du modèle mémoire. La solution proposée est innovante car elle propose un compilateur qui permet de générer un seul binaire exécutable destiné à 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 [80] démontrent l'efficacité de l'outil en comparaison avec un code optimisé à la main. Pour ce qui est du domaine d'application que nous étudions, à savoir le traitement d'images, ***OpenMP*** peut être une solution adaptée pour des calculs simples du type point à point où les corps de boucles sont réguliers et où la vectorisation automatique est triviale. Par contre, lorsqu'il s'agit de calculs du type convolution où il existe des accès non réguliers aux données et des dépendances entre les pas de boucle, les tâches de génération et d'optimisation du code deviennent complexes pour un compilateur. De plus, les optimisations de plus haut niveau telles que celles qui concernent les transferts de données et les optimisations au niveau algorithmique ne sont pas prises en charge par l'implémentation d'***OpenMP*** pour le Cell. C'est pour ces raisons là que cet outil reste efficace dans des cas de parallélisation de boucles simples et intenses en calcul mais la variété des algorithmes et des calculs en traitement d'images nous oblige à nous orienter vers un modèle moins automatique et plus orienté par le programmeur.

3.3 Le langage de programmation *CellSS*

CellSS (*Cell SuperScalar*)[9] 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[7] qui assimile un processeur super-scalaire à une grille de calcul : les unités fonctionnelles du processeur sont les ressources de la grille, les données contenues dans les registres

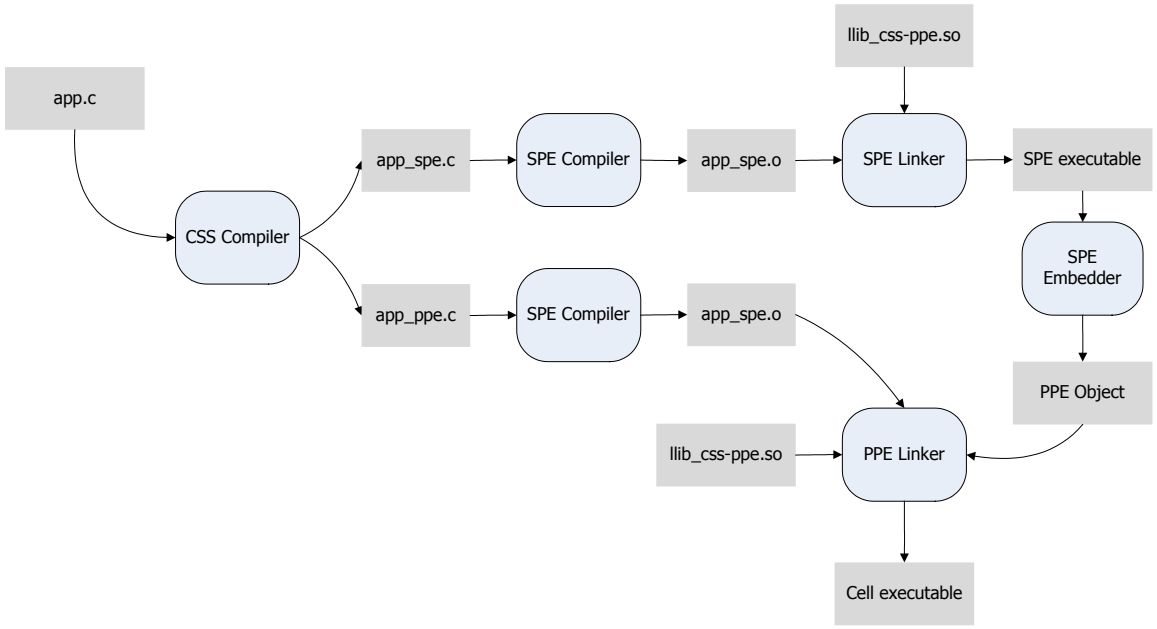

FIGURE 3.2 – Procédure de génération de code de ***CellSS***

correspondent aux fichiers dans la grille et les instructions assembleur sont assimilées aux tâches de calcul.

Cell SuperScalar est constitué de deux composants clés : un compilateur source vers source et une bibliothèque de support d'exécution. Le processus de génération de code est illustré dans la figure 3.2. 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 enfoui 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.

Une section parallèles est annotée par des clauses ***CellSS***, ce qui crée des tâches concurrentes (voir listing 3.3). L'exécution du programme est assurée par le PPE qui assigne les tâches aux SPEs au travers de la bibliothèque de support d'exécution. Le support d'exécution se charge de créer un noeud qui correspond à une tâche dans le graphe, et vérifie la dépendance avec une autre tâche lancée auparavant et ajoute un arc entre les deux. Si la tâche courante est

prête à être exécutée le support d'exécution envoie une requête au SPE pour qu'il se charge de l'exécution. Les transferts DMA sont gérés par le support d'exécution de manière transparente. Les appels au support d'exécution ne sont pas bloquants et de ce fait, si une tâche n'est pas prête ou tous les SPEs sont occupés, le programme principal continue son exécution.
On pourra noter que tout le processus (assignations de tâches, analyses des dépendances, transferts 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.

```
1  #pragma css task inout(diag[B][B]) // pragma surrounding a cellss task
2  void lu0(float *diag){
3    int i, j, k;
4    for (k=0; k<BS; k++)
5    for (i=k+1; i<BS; i++) {
6      diag[i][k] = diag[i][k] / diag[k][k];
7      for (j=k+1; j<BS; j++)
8        diag[i][j] -= diag[i][k] * diag[k][j];
9    }
10 }
11 #pragma css task input(diag[B][B]) inout(row[B][B])
12 void bdiv(float *diag, float *row){
13   ...
14 }
15 #pragma css task input(row[B][B],col[B][B]) inout(inner[B][B])
16 void bmod(float *row, float *col, float *inner){
17   ...
18 }
19 #pragma css task input(diag[B][B]) inout(col[B][B])
20 void fwd(float *diag, float *col){
21   ...
22 }
23 void write_matrix (FILE * file, float *matrix[NB][NB]){
24   int rows, columns;
25   int i, j, ii, jj;
26   fprintf (file, "%d\n %d\n", NB * B, NB * B);
27   for (i = 0; i < NB; i++)
28     for (ii = 0; ii < B; ii++){
29       for (j = 0; j < NB; j++){
30 #pragma css wait on(matrix[i][j])
31         for (jj = 0; jj < B; jj++)
32           fprintf (file, "%f ", matrix[i][j][ii][jj]);
33       }
34       fprintf (file, "\n");
35     }
36 }
```

Listing 3.3 – Exemple sparse LU avec ***CellSS*** : définition des tâches

```
1  int main(int argc, char **argv) {
2    int ii, jj, kk;
3    FILE *fileC;
4    Ě
5    initialize (A);
6  #pragma css start
7    for (kk=0; kk<NB; kk++) {
8      lu0(A[kk][kk]);
9      for (jj=kk+1; jj<NB; jj++)
10       if (A[kk][jj] != NULL)
11         fwd(A[kk][kk], A[kk][jj]);
12     for (ii=kk+1; ii<NB; ii++)
13       if (A[ii][kk] != NULL) {
14         bdiv (A[kk][kk], A[ii][kk]);
15         for (jj=kk+1; jj<NB; jj++)
16           if (A[kk][jj] != NULL) {
17             if (A[ii][jj]==NULL)
18               A[ii][jj]=allocate_clean_block();
19               bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
20           }
21       }
22   }
23   fileC = fopen (argv[3], "w");
24   write_matrix (fileC, A);
25   fclose (fileC);
26   #pragma css finish
27 }
```

Listing 3.4 – Exemple sparse LU avec ***CellSS*** : programme principal

3.3.1 *Runtime*

A l'exécution, 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 support d'exécution 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* [1].

- 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, la tâche peut être exécutée.

Ordonnancement dynamique

Durant l'exécution de l'application, le support d'exécution 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 support d'exécution reçoit une notification et le graphe de tâches 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 support d'exécution 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.

1. Read after Write, Write after Read et Write after Write

Intergiciel 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 compilateur ***CellSS*** et les bibliothèques de support d'exécution. Lors du démarrage du programme sur le PPE le programme 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 données 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 les tâches déjà présentes dans la mémoire des SPEs de celles qui ne 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 de placement est faite de sorte à exploiter la localité et donc à regrouper quand c'est possible les tâche interdépendantes dans le même SPE.

3.3.2 Conclusion sur *CellSS*

CellSS est un modèle de programmation basé sur un compilateur et un support d'exécution. Il permet l'exécution d'un code source séquentiel sur une architecture parallèle grâce à l'annotation de ce code par des directives de placement sur les SPEs. Le support d'exécution construit un graphe de dépendances des fonctions appelées et ordonnances ces 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.
Le modèle de programmation ***CellSS*** parait plus adapté qu'***OpenMP*** pour notre domaine d'application. D'une part, parce qu'il est basé sur une notion de graphe de tâches pouvant être placées suivant un schéma choisi par le programmeur et d'autre part, les tâches peuvent contenir du code vectoriel optimisé ce qui n'est pas le cas pour les sections ***OpenMP*** par exemple. Il est donc clair, que ce modèle est plus adapté pour notre domaine d'application. Toutefois, les essais que nous avons mené avec l'outil et qui ont donnée lieu à plusieurs échanges avec les développeurs du BSC (Barcelona Supercomputing Center), ont montré que l'outils était encore dans un état de maturité insuffisant car il ne prenait pas en charge toutes les fonctionnalités énumérées dans [9] et qu'il ne pouvait donc pas être utilisé en production.

3.4 Le langage de programmation *Sequoia*

Sequoia[39] est un langage de programmation bas-niveau dédié aux machines modernes, qu'elles soient 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, pour qu'ils soient efficaces au niveau de la bande passante et qu'ils restent portables sur de nouvelles machines. Le modèle de programmation repose sur les principes 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.

3.4.1 Mémoire hiérarchique

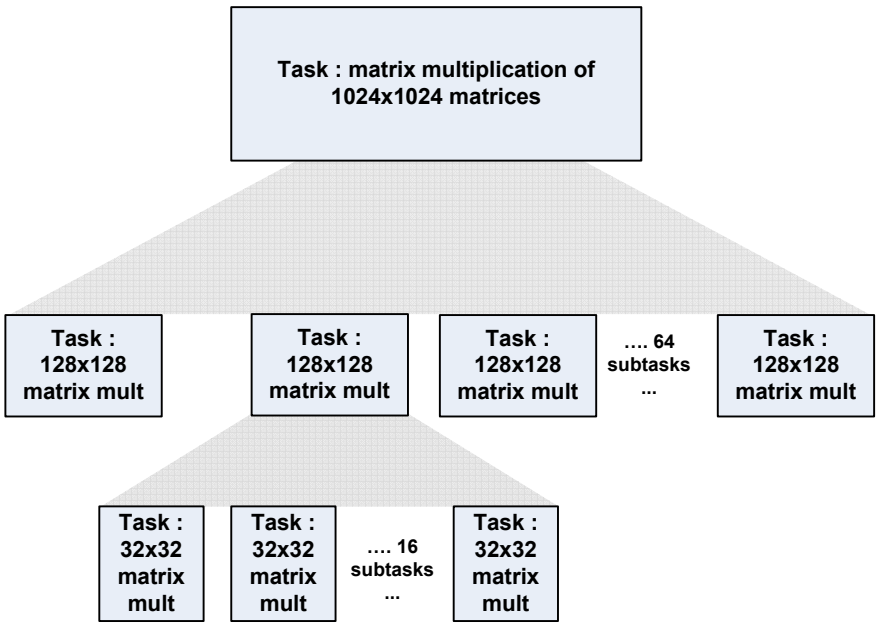

FIGURE 3.3 – Multiplication de matrices de tailles 1024x1024 structurée en hiérarchie de tâches indépendantes effectuant des multiplications sur des blocs de données plus petits

Le principe de mémoire hiérarchique est au coeur de l'outil ***Sequoia***. 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. Cela permet d'exposer le parallélisme et d'atténuer l'effet de la latence d'accès à la mémoire car les données sont physiquement proches des unités de traitement. Un exemple de découpage pour l'application produit de matrices est donné dans la figure 3.3. Cet exemple contient du parallélisme imbriqué et la localité des données y 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 chaine 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 support d'exécution pour le processeur Cell.

L'écriture d'un programme ***Sequoia*** implique la description d'une hiérarchie de tâches et le placement 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 uniquement 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.3.4) contient des noeuds correspondants à la mémoire principale ainsi qu'à chaque mémoire locale du SPE. Un 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*** peuvent se faire au travers d'une instruction de prefetch, un transfert DMA ou un message MPI selon les spécificité 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) à 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 le processeur Cell. Cela permet notamment d'encapsuler les communications horizontales inter-noeud tout en gardant le modèle d'abstraction en arbre

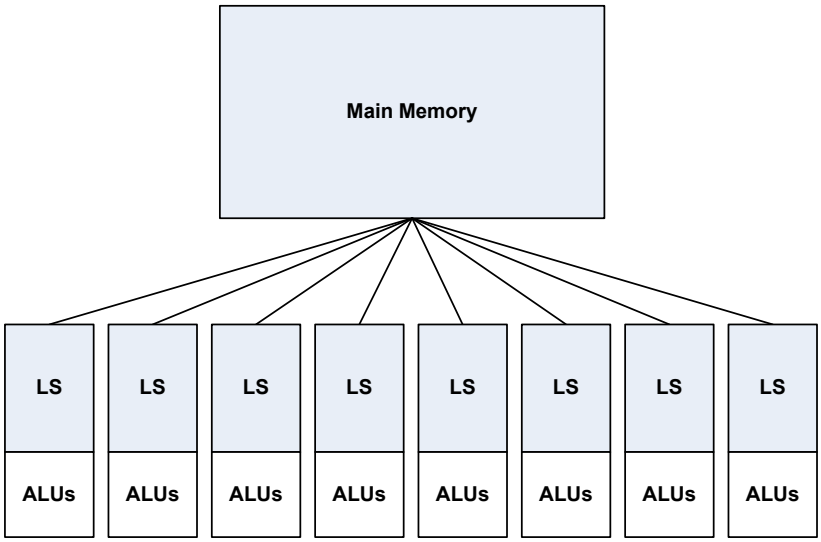

FIGURE 3.4 – Modèle abstrait ***Sequoia*** du processeur Cell

qui lui, ne permet que des communications verticales.

3.4.2 Modèle de programmation : les tâches

La notion de *task* est au coeur du modèle de programmation de ***Sequoia*** : c'est une fonction pure 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 3.5 page 115.

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 permis 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 appelante après le retour de la fonction appelée.
Le déploiement d'un programme ***Sequoia*** dicte quand une tâche appelée doit être exécutée dans le même module mémoire que la tâche appelante ou alors assignée à une mémoire fille.

Isolation et parallélisme

La granularité du parallélisme dans ***Sequoia*** est la *tâche* et l'exécution parallèle résulte de l'appel de *tâches* concurrentes. Une tâche s'exécute généralement sur une partie de la boucle externe 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 placement 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, ...)`

Génere un `blkset` qui contient des blocks qui ne se recouvrent pas et qui tuilent le tableau multidimensionnel A. Chaque bloc est multidimensionnel de taille `len0` × `xlen1`× ....

- `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 boule `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 boule 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.

Paramètrisation 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 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 différents niveaux de la hiérarchie mémoire d'une cible donnée. L'utilisation de paramètres comme la taille des tableaux ainsi que les paramètres ajustable, découple l'expression d'un algorithme de son implémentation sur une machine donnée.

Spécialisation de tâches et tunning

Dans ***Sequoia*** on donne au programmeur le contrôle complet sur les phases de spécialisation et de tunning 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 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 meilleurs performances possibles.

```
1  void task matmul :: inner (in float A[M][P],
2                             in float B[P][N],
3                             inout float C[M][N])
4  {
5    // Tunable parameters specify the size
6    // of subblocks of A, B, and C.
7    tunable int U;
8    tunable int X;
9    tunable int V;
10   // Partition matrices into sets of blocks
11   // using regular 2D chopping .
12   blkset Ablks = rchop(A, U, X);
13   blkset Bblks = rchop(B, X, V);
14   blkset Cblks = rchop(C, U, V);
15     // Compute all blocks of C in parallel.
16     mappar(int i=0 to M/U, int j=0 to N/V){
17       mapreduce (int k=0 to P/X){
18         // Invoke the matmul task recursively
19         // on the subblocks of A, B, and C.
20         matmul(Ablks[i][k], Bblks[k][j], Cblks[i][j]);
21       }
22     }
23 }
24 void task matmul :: leaf(in float A[M][P],
25                          in float B[P][N],
26                          inout float C[M][N])
27 {
28   // Compute matrix product directly
29   for (int i=0;i<M;i++)
30     for (int j=0;j<N;j++)
31       for (int k=0;k<P;k++)
32         C[i][j] += A[i][k]*B[k][j];
33 }
```

Listing 3.5 – Exemple de multiplication matricielle par blocs ***Sequoia***

3.4.3 Implémentation du compilateur *Sequoia*

Dans l'implémentation de ***Sequoia***, un compilateur source vers source génère du code C qui s'interface avec un support d'exécution 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 support d'exécution

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 support d'exécution est piloté par les évènements : un système de notification via *mailbox* est mis en place entre le PPE est les SPEs, les tâches sont assignées par le PPE au SPEs. Une fois que la notification est reçue, un mécanisme du support d'exécution 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 utilisés entre les tâches, un seul point de synchronisation est requis lors de la fin des sections parallèles, ce qui minimise le surcoût.

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 de la hiérarchie mémoire, où les copies inutiles sont détectées et supprimées. D'autres optimisations dans le *software-pipelining* et le déplacement des invariants de boucle, sont effectuées de manière statique à la compilation.

3.4.4 Conclusion sur *Sequoia*

Sequoia est un langage de programmation *data parallel* qui tente d'allier la portabilité avec la performance pour le portage d'algorithmes 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 découplée de l'implémentation. L'expression explicite des communications, des mouvements des données au travers de la hiérarchie mémoire, des calculs parallèles et la définition d'un ensemble de travaux sont effectués grâce à une seule abstraction, la *tâche*. ***Sequoia*** fournit des primitives de structuration des calculs 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. Cependant, l'outil possède des limitations quand aux schéma de parallélisation possibles car il ne supporte que le parallélisme SPMD. Les schémas que nous avons implémentés lors de notre étude ne sont pas couverts par l'outil. De plus, les optimisations bas-niveau sont à la charge du programmeur. Cela impose d'apprendre un *framework* de programmation supplémentaire avec des limitations imposées par l'outil. Nous n'avons donc pas retenu cette solution pour la mise en oeuvre de nos implémentations.

3.5 Conclusion

Dans ce qui précède, nous avons décrit différents outils 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'évaluations significatives. Les outils diffèrent par le modèle de programmation sur lequel ils se basent. 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'optimiser le code. D'autre part, ***RapidMind*** et ***Sequoia*** se reposent sur un langage spécifique accompagné d'un support d'exécution et d'un compilateur, mais une bonne partie du travail d'optimisation est laissée à la charge du programmeur. D'autres modèles de programmation qui ne sont pas exposés ici, ont fait l'objet d'études sur le processeur Cell, dont le modèle de programmation par passage de message MPI. N'ayant pas pu avoir accès aux implémentations décrites dans [81] et [67], nous n'avons pas pu en évaluer la facilité de mise en oeuvre et les performances.

L'ensemble de ces outils ont vu le jour pour faciliter la programmation sur le Cell, et fournir une alternative à l'utilisation fastidieuse 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 et de son optimisation. Un code à base de threads est de ce fait long à mettre en oeuvre est difficile à maintenir, mais il permet d'un autre côté d'avoir un contrôle total sur son implémentation. On notera que la mise en oeuvre à nécessité 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 des jeux d'instructions différents ce qui rend difficile l'étape de génération de code.

L'ensemble des outils étudiés dans ce chapitre adoptent un modèle de programmation du type SPMD, qui ne permet d'exploiter que le parallélisme de données dans les applications. De plus, ces outils se basent sur les compilateurs fournis pour les tâches d'optimisation du code. Les outils présentés sont limités à un seul schéma de parallélisation et ne permettent pas de faire certaines optimisations très profitables pour les applications de traitement d'images telles que les optimisations interprocédurales et la vectorisation de code. Cette analyse a permis de mettre en avant le fait que ces outils ne sont pas adaptées à notre domaine d'application. Dans le chapitre qui suit nous présentons un outil de programmation pour le Cell que nous

avons développé et qui propose une approche de programmation différente, plus adaptée au traitement d'images et qui permet d'exploiter d'autres formes de parallélisme.

Chapitre 4

Squelettes algorithmiques pour le Cell

Les outils de programmation pour le Cell présentés dans le chapitre qui précède, sont basés sur un modèle de programmation qui exploite le parallélisme de données. Nous avons constaté que ces outils qui visent une facilité de mise en oeuvre du code et une généricité au niveau du domaine d'application, n'apportent pas une réponse adéquate à l'optimisation de la performance sur le processeur Cell.

Dans ce chapitre nous présentons un outil développé conjointement par Dr. Joel Falcou, Dr. Lionel Lacassagne et moi même, et qui vise à répondre au mieux aux exigences de performances du domaine du traitement d'images.

La programmation parallèle structurée qu'on appelle programmation par squelettes algorithmiques [20, 26, 91, 66] restreint l'expression du parallèlisme à la composition d'un nombre prédéfini de patrons nommés squelettes. Les squelettes algorithmiques sont des briques de base génériques, portables et réutilisables pour lesquelles une implémentation parallèle peut exister. Ils sont issus des langages de programmation fonctionnelle. Un système de programmation basé sur les squelettes fournit un ensemble fixe de patrons. 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-hiérarchique en mettant en séquence les différents blocs en utilisant des variables temporaires pour sauvegarder les ré-

sultats intermédiaires, ou alors de manière hiérarchique en imbriquant les fonctions squelette et 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 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 utilisés en parallèle dans un niveau particulier de la hiérarchie. Les squelettes ne pouvant pas être imbriqués sont généralement implémentés avec juste une librairie générique alors que les squelettes imbriqués requièrent un pré-traitement qui déroule la hiérarchie du squelette en utilisant par exemple les templates *C++* [95] ou les macros de pré-processeur en C. L'outil `SKELL BE` développé trouve ses origines dans les travaux de thèse de Joel Falcou [35]. Dans ces travaux une bibliothèque de vectorisation pour les applications de vision par ordinateurs E.V.E [38] est développée. L'outil QUAFF [34] utilisant le modèle de programmation par squelettes algorithmiques est également développé pour la parallélisation de code sur cluster de machines distribuées.

SKELL BE est une adaptation de QUAFF et EVE pour le processeur Cell. Ma contribution personnelle dans l'outil `SKELL BE`, s'est concrétisée dans l'apport des briques logicielles de base pour l'outil de génération de code déjà développé. J'ai également participé à la conception de la librairie de communication et de synchronisation Cell-MPI [90], nécessaire au déploiement de QUAFF sur le processeur Cell. Les travaux développés dans ce chapitre ont été publiées dans [56] et [86]

4.1 Programmation parallèle par squelettes algorithmiques

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 4.1. On peut distinguer sur cette figure une structure **A** imbriquée dans une structure B :

– un système d'équilibrage de charge **(A)** qui utilise k processeurs pour traiter les données

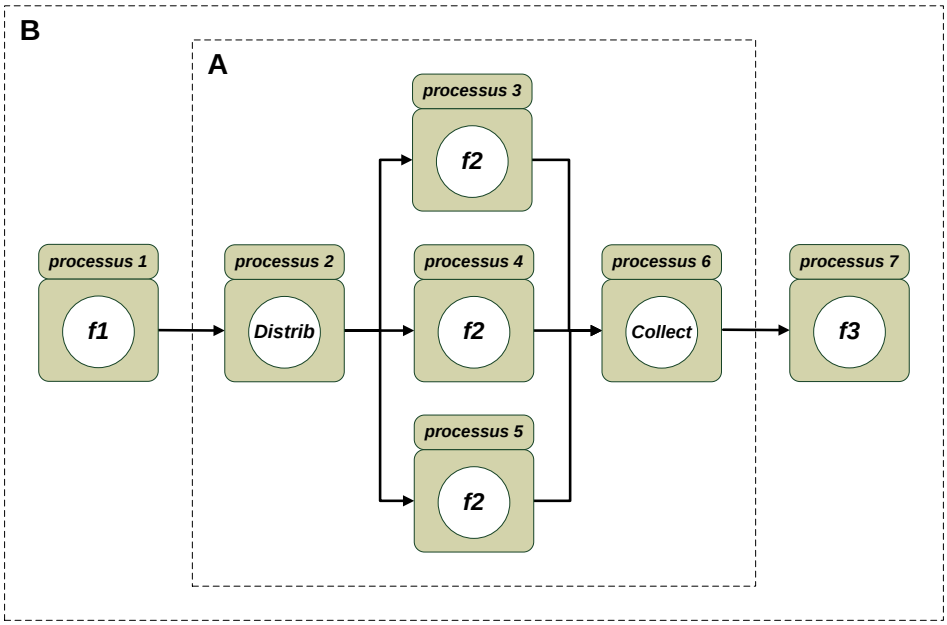

FIGURE 4.1 – Exemple de graphe de processus communicants avec hiérarchisation

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

– le mécanisme de contrôle **(B)** qui permet de séquencer les traitements 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 à un ou plusieurs processus P_{i+j}. On pourra noter que le schéma d'exécution dans ce cas là est du type *pipeline* (Fig. 4.1 et 4.2).

Le modèle de programmation parallèle par squelettes algorithmiques repose sur l'extraction de tels schémas génériques. Un squelette est ainsi défini comme étant un schéma générique paramètré 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 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 composition peut se faire en encodant cette dernière sous la forme d'un **arbre** (Fig. 4.2) dont les noeuds représentent les squelettes utilisés et les feuilles, les fonctions séquentielles passées en paramètres à ces squelettes. Dans la figure 4.2, le squelette

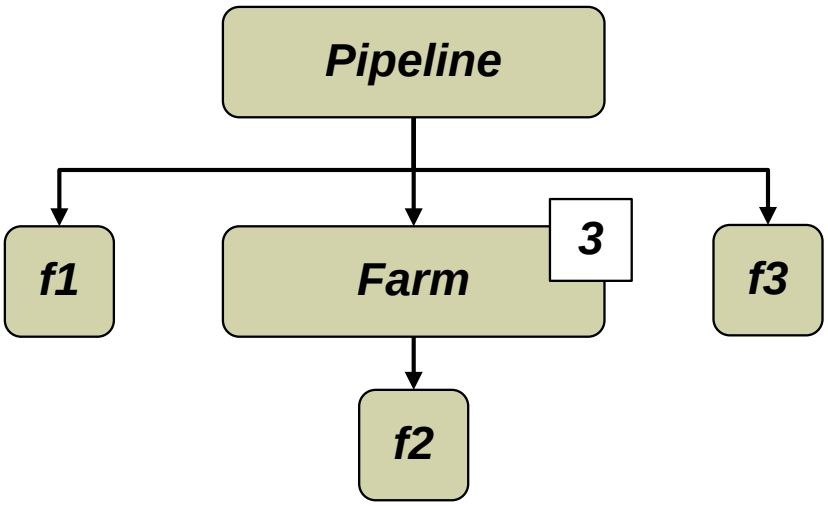

FIGURE 4.2 – Arbre représentant le squelette en Figure 4.1

Pipeline décrit le schéma générique correspondant à la section **B** du graphe de processus communicant 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. Une classification possible consiste à les répartir en trois groupes : les squelettes dédiés au parallélisme de tâches, les squelettes dédiés au parallélisme de données et les squelettes dédiés à la structuration séquentielle de l'application.

4.1.1 Squelettes dédiés au parallélisme de contrôle

Cette classe de squelettes permet d'exploiter le parallélisme de tâches des applications. On peut citer deux squelettes qui couvrent cette forme de parallélisme **Pipeline** et **Pardo**

Le squelette Pipeline

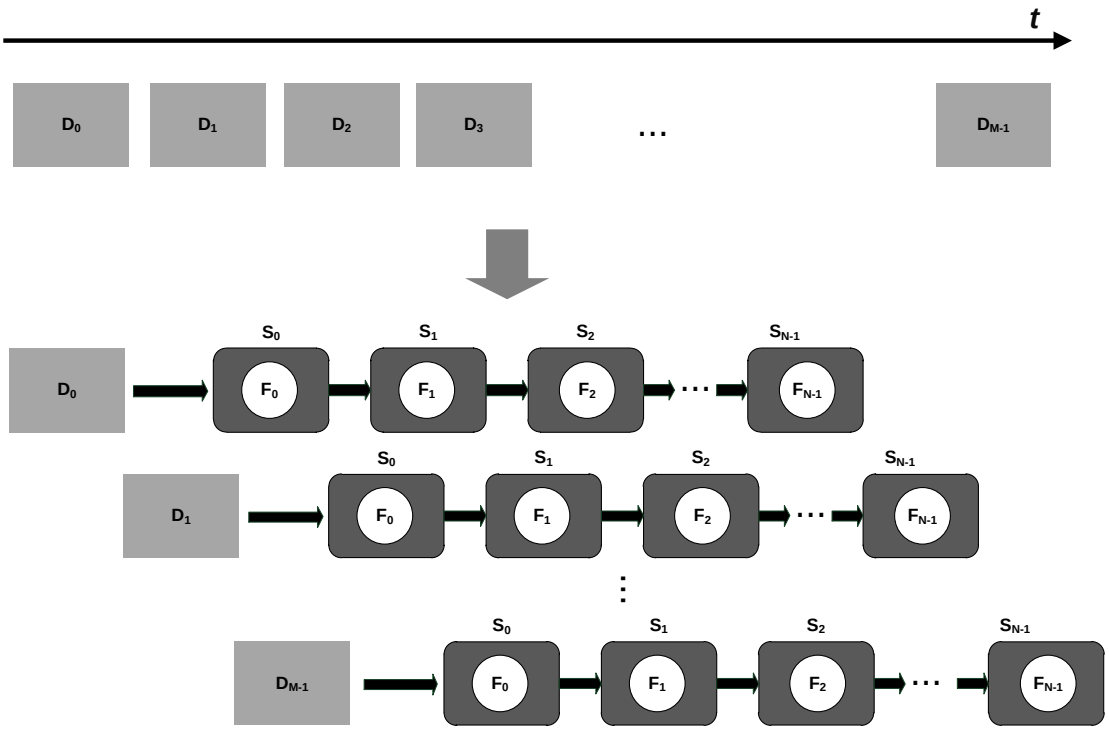

FIGURE 4.3 – Exemple de squelette du type **Pipeline**

Ce squelette couvre les situations dans lesquelles une liste de fonctions qui doivent s'exécuter en série, sont réparties 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 4.3 illustre ce fonctionnement en régime permanent. Le parallélisme résulte du fait que l'évaluation des différentes fonctions du **Pipeline** sur des éléments différents du flux (D_0, D_1 par exemple sur la figure 4.3) 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

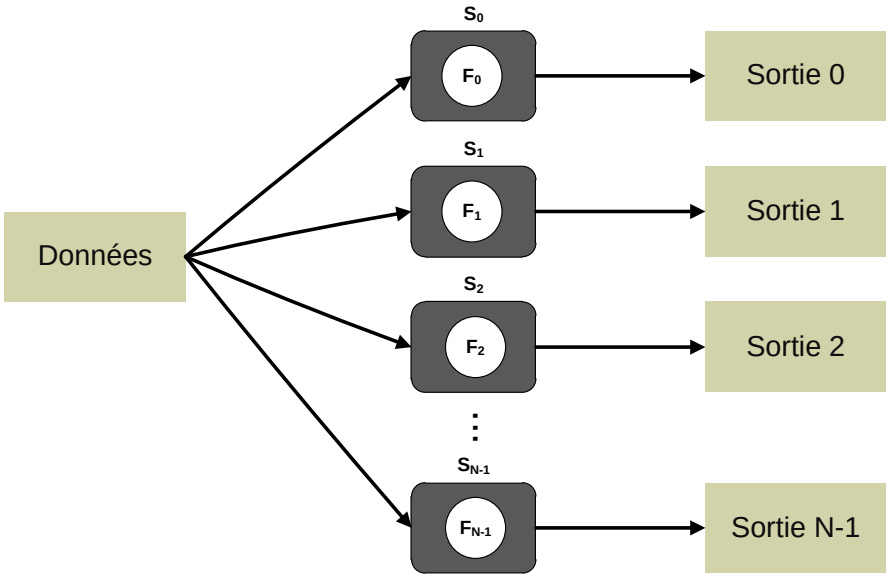

FIGURE 4.4 – Exemple de squelette du type **Pardo**

Le squelette **Pardo** permet de placer de manière *ad hoc* N fonctions sur N processeurs (Fig. 4.4). 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.

4.1.2 Squelettes dédiés au parallélisme de données

Le parallélisme de données très présent dans beaucoup d'applications, peut être couvert par deux squelettes **Farm** et **SCM**, le premier gère le parallélisme de données simple et le second gère le parallélisme de donnée par décomposition de domaine

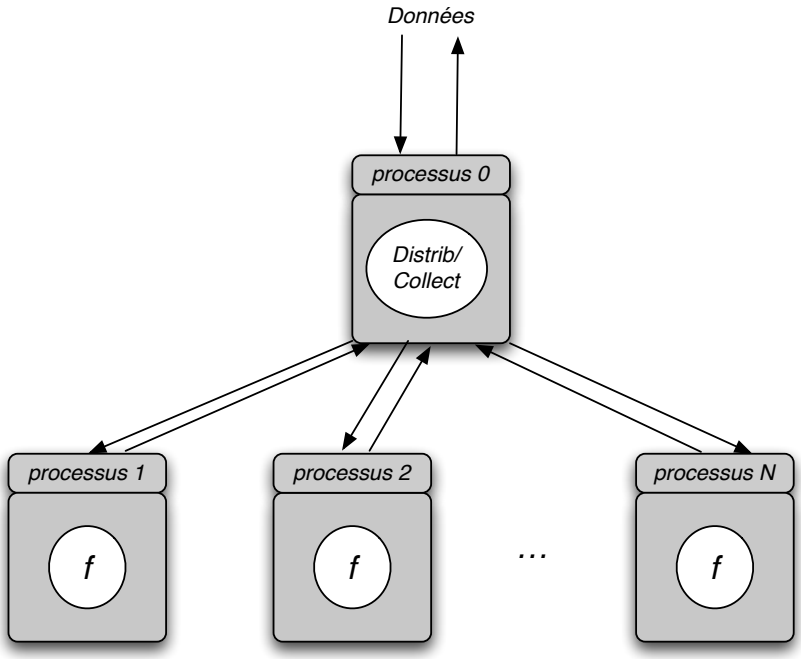

FIGURE 4.5 – Exemple de squelette du type **Farm**

Squelette Farm

Ce squelette permet de gérer des situations où la même tâche est effectuée par tous les processus parallèles et où ceux-ci sont alimentés par un mécanisme d'équilibrage de charge. Pour ce faire, une partie du flux de données est envoyée à une des p unités de calculs disponibles. Le parallélisme provient alors du fait que le traitement sur un élément $d(i)$ de donnée se fait en parallèle de celui d'un élément $d(j)$. Un éléments de données à traiter est envoyé au premier processus esclave libre. Ce dernier effectue le traitement et renvoie le résultat au processus de collecte. Ce processus est fait en continu par le processus maître qui envoie les données à traiter en permanence jusqu'à épuisement de celles-ci. Une illustration du squelette **Farm** est donnée en figure 4.5.

Le squelette **Farm** est caractérisé par le temps de mise à disposition qui est la grandeur qui représente le temps passé entre l'entrée d'un élément du flot de données dans le squelette et la disponibilité du résultat de calcul correspondant. Sa valeur est la somme du temps d'exécution d'un processus esclave et du temps de communication qui englobe la durée de l'envoi des données et celui de la réception des résultats. Dans l'éventualité où toutes les unités de

calcul sont occupées, un temps d'attente est ajouté à la durée totale :

$$T_{disp} = T_{func} + T_{comm} + T_{attente}$$

Squelette SCM

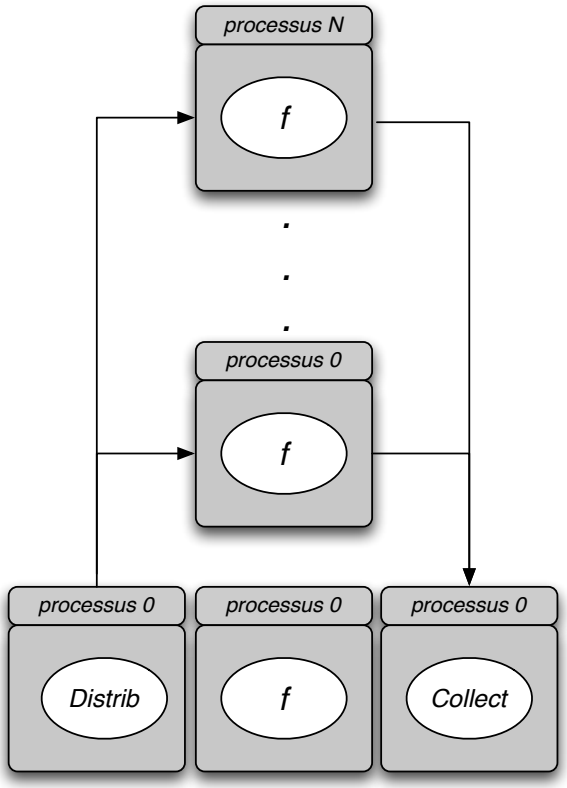

FIGURE 4.6 – Exemple de squelette du type **SCM**

Le squelette SCM (Fig. 4.6) est un squelette dédié au parallélisme de données régulier. Il permet d'utiliser un schéma fixe de décomposition des données. Dans ce schéma le traitement commence par une décomposition de domaine des données. Les morceaux de données sont ensuite envoyés aux processus esclaves qui appliquent une fonction f sur ces données. Les données résultats de l'évaluation de la fonction f sont ensuite collectées par le processus maître qui reconstruit une structure de données résultats. Ce squelette est caractérisé par le rapport entre le temps engendré par la distribution et la collecte des données, et le temps effectif de calcul sur les processus esclaves.

4.1.3 Squelettes dédiés à la structuration de l'application

Cette classe de squelettes permet de structurer l'application sans y introduire pour autant une quelconque forme de parallélisme. Des constructions comme les branchements conditionnels et la composition séquentielle de fonctions sont utilisées pour structurer l'application.

4.1.4 Le squelette Sequence

Ce squelette permet la composition séquentielle d'un nombre arbitraire de tâches. Il est très semblable au squelette pipeline par sa structure. Seul le parallélisme inhérent à chaque fonction qui compose la séquence est exploité. La grandeur caractéristique de la séquence est sa latence, donnée par la somme des temps d'exécution des fonctions f_i qui la composent.

$$T_{séquence} = \sum f_i$$

4.1.5 Le squelette Select

Le squelette **Select** est équivalent à la construction "`if ... then ... else`" du langage C. Il est composé de trois fonctions représentant respectivement : la condition du test, la fonction qui correspond au cas vrai et enfin celle qui correspond au cas faux. Le squelette **Select** se comporte exactement comme une construction "`if ... then ... else`" séquentielle du point de vue opérationnel.

4.2 Classification des implémentations de squelettes algorithmiques

Plusieurs implémentations s'appuyant sur l'approche de programmation parallèle par squelettes algorithmiques ont vu le jour depuis la publication du manifeste de Cole [19]. Ces différents outils sont classés dans [44] selon leur paradigme de programmation :

- Coordination
- Fonctionnel
- Orienté Objet
- Impératif

4.2.1 La coordination

Cette approche préconise l'utilisation d'un langage haut-niveau pour décrire le comportement algorithmique et un langage hôte pour gérer l'intégration avec l'infrastructure. SCL (*The Structural Coordination Language*) [27], Skil (*the Skeleton Imperative Language*) [13], P3L (*the Pisa Parallel Programming Language*) [6], le langage LLC [31] et le langage SAC (*the Single Assignment C*) [45], complètent les langages impératifs avec un langage de coordination pour décrire les squelettes à un niveau supérieur. En traduisant la description sous forme de squelettes dans le langage hôte, ils permettent au programmeur de générer un programme en assemblant la portion haut-niveau avec la structure du langage hôte au dessus de l'infrastructure bas-niveau du programme parallèle, typiquement MPI. Les principaux inconvénients de cette approche sont le besoin d'apprendre un nouveau langage et la nécessité de préparer une infrastructure système optimisée pour le langage hôte, sous la forme de traducteurs et de compilateurs. La principale force de cette approche est par conception, la séparation des primitives de coordination et de communication.

4.2.2 La programmation fonctionnelle

Le parallélisme structuré a été intégré dans les langages de programmation fonctionnels parallèles sous forme d'extensions syntaxiques, ou alors sous forme de foncteurs dans les langages existants. HDC (*the Higherorder Divide-and-Conquer language*) [50] et Eden [71] ont élargi Haskell pour y ajouter des extensions parallèles qui permettent de décrire les structures des squelettes. Ils génèrent des programmes parallèles, soit en traduisant le programme en source C/MPI pour ce qui est de HDC, soit en utilisant le compilateur GHC *Glasgow Haskell Compiler* comme infrastructure dans le cas d'Eden. De plus, ces outils fournissent des instructions spécifiques pour manipuler les processus et les flots de données en entrée/sortie. Les squelettes on été introduits à travers les foncteurs Haskell dans Concurrent Clean [52], dans ML avec la notation PMLS (*Parallel ML with Skeletons*) [75], OCamlP3I [18], dans le système Skipper [91] et enfin dans Hope [26]. Ces foncteurs permettent l'expression des squelettes sans interférer dans la syntaxe du langage original. Tandis que les squelettes à base de langages fonctionnels offrent l'approche la plus élégante pour l'abstraction du parallélisme, les implémentations à base de langage *C* fournissent les meilleures performances.

4.2.3 L'approche orientée objet

Les squelettes sont introduits dans les langages orientés objet en utilisant les classes. Les projets SkeTo (*Skeletons in Tokyo*) [72], Muesli (*the Münster Skeleton Library*)[17] et la bibliothèque Mallba (*the Malaga-La Laguna-Barcelona*) [1] sont basés sur des classes *C++* et la bibliothèque MPI. Ils déploient les squelettes *data parallel*, *task parallel* et *resolution*. De plus, Calcium [16], JaSkel [40], Lithium [3], muskel [2], Quaff [34] et Skandium [70] fournissent des squelettes distincts sous forme de classes *Java* ou *C++* dans leur *framework* orienté objet. Il est important de noter que les outils cités précédemment s'appuient sur les capacités d'abstraction du langage orienté objet hôte, sans introduire un surcoût important, car ils n'imposent pas de syntaxe particulière. Ce type de paradigme est resté très porteur grâce à la popularité des langages orientés objet et au fait que ce type d'approche résout partiellement les problèmes de portabilité et de performances.

4.2.4 L'approche impérative

Les squelettes algorithmiques sont également déployés sous forme d'API dans les languages procéduraux. SkIE (*the Skeleton-based Integrated Environment*) [5], ASSIST (*the Software development System based upon Integrated Skeleton Technology*) [96], SKElib (*the Pisa's Skeleton Library*) [25] et eSkel (*the Edinburgh Skeleton Library*) [19] fournissent des squelettes *data parallel* et *task parallel* en se basant sur des appels à des routines bas-niveau.

4.3 Squelettes implantés sur le Cell

Plusieurs squelettes algorithmiques ont été proposés dans la littérature [20]. Ils couvrent les deux principales formes de parallélisme : données et tâches. Il n'existe pas de liste standard de squelettes mais on note qu'il existe un sous-ensemble récurrent :

- Le squelette **SEQ** qui permet d'encapsuler des fonctions définies par l'utilisateur afin de les utiliser comme paramètres de squelettes.
- Le squelette **PIPELINE** qui est équivalent à la composition parallèle de fonctions (Fig. 4.3)
- Le squelette **MAP** permet de modéliser le parallélisme de données régulier dans lequel les données sont partitionnées et envoyées au unités de traitement esclaves. Une fois

ces partitions de données traitées elles sont collectées au sein de la même tâche qui les a envoyées.

– Le squelette **Farm** permet de modéliser le parallélisme de données irrégulier et asynchrone basé sur une stratégie d'équilibrage de charge arbitraire.

Les spécificités de l'architecture du Cell induit des choix au niveau de la conception logicielle de l'application parallèle décrits dans le chapitre 2. Parmi ces schémas de parallélisation on trouve la possibilité de chainer des noyaux de calcul au sein d'un même SPE et de répliquer une séquence de noyaux sur des groupes distincts de SPEs. Ces deux derniers schémas permettent de réaliser les meilleurs performances sur le Cell. Ainsi, nous avons défini un ensemble de squelettes supportés par le Cell :

– Les squelettes **SEQ** et **PIPELINE** définis ci-dessus.
– Le squelette **CHAIN** qui modélise l'appel séquentiel de deux fonctions de traitement au sein du même SPE.
– Le squelette **PARDO** (Fig. 4.4) modélise des tâches s'exécutant de manière concurrente sur les SPEs.

Ce sous-ensemble est assez riche pour supporter plusieurs schémas de parallélisation sans induire des schémas de communication et de synchronisation complexes. Cependant, l'imbrication des squelettes n'est pas supportées car les SPEs ne peuvent supporter qu'un seul *thread* qui ne peut pas en engendrer d'autres.

4.4 Modèle de programmation `SKELL BE`

Dans le modèle de programmation `SKELL BE` [86], [56], le processeur Cell est considéré comme une machine asymétrique. Le modèle inclue deux processus, 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* (listing 4.1). Le *kernel* est une fonction de calcul qui prend en entrée des données, applique un traitement défini par son code source et renvoie les données résultantes en sortie.

```
1  #include <skell.hpp>
2  // skell be kernel declaration
3  // performs threads initialisation
4  SKELL_KERNEL(sample,(2,(float const*,float*)));
5  int main(int argc, char** argv)
6  {
7   float in[256], out[256];
8
9   skell::environment(argc,argv);
10  sample(in,out);
11
12  return 0;
13 }
```

Listing 4.1 – 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 constante sont les entrées du *kernel*. La ligne 9 du listing 4.1 montre un exemple d'appel de *kernel* à l'exécution. 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 noeud de *cluster* qui supporte la communication point-à-point. Les applications sont construites en composant des squelettes instanciés avec des fonctions définies par l'utilisateur. Les squelettes voient la mémoire centrale comme une mémoire distante à partir de laquelle peuvent être lues ou écrite des données de manière asynchrone au travers des commandes DMA de la librairie standard ou des fonctions fournies par SKEL1_BE (listing 4.2).

```
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 4.2 – Exemple de définition d'un kernel SPE dans `SKELL BE`

Ces deux derniers codes source illustrent plusieurs aspects :

- la macro `SKELL BE` (listing 4.1 ligne 4) 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 aplication 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 (listing 4.2 lignes 7 et 9).
- les fonction `pull` et `push` (lignes 7 et 9 du listing 4.2) qui permettent un accès asynchrone à la mémoire principale adressée par le PPE. Ces fonctions déduisent la meilleurs manière de rapatrier les données à partir de leurs arguments. Si celles-ci sont dans la mémoire locale des SPE une commandes DMA est effectuée alors que si les données sont déjà présentes en mémoire externe, une copie inutile est évitée. Les données sont découpées statiquement en fonction du nombre de SPEs impliqués et de la taille de leur mémoire privée.
- La fonction `terminate` déclenche la terminaison du *kernel*. Celle-ci n'est appelée qu'une

fois per *kernel*, dès que toutes les données transférées de la mémoire centrale on été traitées.

Le tableau 4.1 résume les principales fonctions de l'interface utilisateur (API) de `SKELL BE`.

4.5 Détails de l'implémentation

Le développement d'une librairie de calcul parallèle à base de squelettes algorithmiques à la fois efficace et expressive est une tâche complexe. Plusieurs tentatives [26, 91, 66] 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 le sourcoût induit par le polymorphisme à l'exécution 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 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 déclaratif *domain-specific*(*domain specific language [74]*) [1]. 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 pour produire des valeurs, 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

1. en opposition à un langage *general-purpose*

Gestion de l'application	
`environement(argc,argv)`	Démarrage de l'application
`rank()`	retourne l'indentifiant PID du SPE courant
`terminate()`	Signale la fin du flux de données et termine l'application
`run(skeleton)`	Exécute une application squelette
Constructeurs de squelettes	
`seq(f)`	Transforme une fonction utilisateur en une tâche squelette

`operator,(`s_1`,`s_2`)`
`chain(`s_1`,...,`s_n`)`
`chain<N>(s)`
	Constructeur de composition séquentielle

`operator|(`s_1`,`s_2`)`
`pipeline(`s_1`,...,`s_n`)`
`pipeline<N>(s)`
	Constructeur de *Pipeline*

`operator&(`s_1`,`s_2`)`
`pardo(`s_1`,...,`s_n`)`
`pardo<N>(s)`
	Constructeur de *Pardo*

Transfert de données	
`pull(arg`$_N$`,v,sz=0,o=0)`	Récupère sz éléments de la $N^{ième}$ donnée de la mémoire centrale et les sauvegarde dans v avec un *offset* o
`push(arg`$_N$`,v,sz=0,o=0))`	Envoie sz éléments de la donnée v à la $N^{ième}$ donnée dans la mémoire centrale avec un *offset* o

TABLE 4.1 – Interface utilisateur SKELL BE

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.

4.6 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 : Abstract Syntax Tree*). Ceci est effectué en général à l'aide d'une technique connue sous le nom de *Expression Templates* [97]. 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 appelle une **évaluation partielle** [98].

Pour transformer un *AST* en un code exploitable, il faut transformer l'arbre en un réseau de processus. Afin d'y parvenir, la sémantique opérationnelle définie par Falcou et *al.* dans [37] est transformée en méta-programme capable de générer une liste statique de processus. Chacun des constructeurs de squelettes de `SKELL BE` génère un objet sans état dont le type encode le structure du squelette. Le code de l'opérateur `pipe` est donné dans le listing 4.3 à titre d'exemple. 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 4.3 – L'opérateur `pipe`

Le type *template* est désormais utilisable avec nos méta-fonctions. Celles-ci se chargent de la génération de structures représentant un réseau de processus. La fonction `run` appelle une méta-function qui parse le *template* AST et génère l'instantiation 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 le 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 4.4 – La structure d'un squelette connue à la compilation

La structure de cet objet temporaire est désormais claire. Les appels successifs à l'opérateur *pipe* 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 :

```
network< int_<0>, int_<2>
  , list< process< int_<0>
    , desc< pid<-1>, pid<1>
      , instrs<Call<&A>,Send>
      >
    >
  , process< int_<1>
    , desc< pid<0>, pid<2>
      , instrs<Recv,Call<&B>,Send>
      >
    >
  , process< int_<2>
    , desc< pid<1>, pid<-1>
      , instrs<Recv,Call<&C> >
      >
    >
  >
>
```

Listing 4.5 – Représentation sous forme de réseau de processus

Le type *template* `network` contient toutes les informations qui décrivent le réseau de processus série communicants construit à partir des squelettes, notamment : le PID du premier noeud du réseau, le PID du dernier noeud du réseau et une liste de processus. 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 4.6 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 4.6 – Code source généré

La structure SPMD des instructions `if` chaînées montre la nature itérative du générateur de code. Chacun des ces blocs effectue les mêmes opérations. En premier lieu, les types de entrées/sorties sont récupérés de l'analyse de type. Ces types sont ensuite instanciés sous forme d'un tuple. Le code du processus est ensuite exécuté, dans une boucle qui se met 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 un transfert DMA, soit à un proxy d'appels de fonctions qui extrait les données d'un tuple, alimente une

fonction définie par l'utilisateur 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 à l'exécution [36].

Le processus de génération permet de mieux comprendre pourquoi `SKELL BE` est plus performant que d'autres solutions 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 à l'exécution et le compilateur est capable d'aligner plus de code et d'effectuer plus d'optimisations.

4.7 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 transferts de données entre la mémoire centrale et les mémoires locales 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 aux 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 méta-pogrammation *template* qui analyse 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 l'algorithme suivant.

- Les entrées de types natifs sont stockés par valeurs.
- Les sorties de types natifs sont stockées 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 4.7 – Code source généré

La fonction suivante est ensuite générée pour l'initialiser :

```
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 4.8 – Code source génére

Du côté du SPE, les objets `argN` fournissent un opérateur de cast *template* implicite qui récupère les valeurs du $N^{ième}$ élément du tuple à l'aide d'une commande DMA, ainsi que d'un opérateur d'affectation qui transfert une valeur à la donnée 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 manière que le compilateur puisse appeler la bonne primitive de transfert DMA en se basant sur l'indice des arguments et le type de la valeur.

4.8 Résultats expérimentaux

Nous avons effectué des mesures qui ont eu pour but de prouver que `SKELL BE` ne provoque pas de pertes importantes au niveau de la performance. Pour ce faire, nous avons effectué plusieurs tests. Le premier vise à créer des applications synthétiques à base de squelettes afin d'évaluer le surcoût des méta-programmes qui est matérialisé par le temps que le système de génération de code ajoute au temps d'exécution total de l'application sans génération automatique. Le deuxième test consiste en l'évaluation de la performance d'algorithmes de calcul numérique de base et enfin le troisième et celui de l'algorithme de Harris pour la détection de point d'intérêt vu précédemment. Les tests on été effectués sur une lame IBM QS20 et la compilation est faite avec la chaîne `gcc`. La métrique temporelle utilisée est le nombre de cycles moyen par point traité.

4.8.1 Benchmarks synthétiques

Cette mesure a pour but de prouver que le surcoût induit par la couche de *méta-programmation* d'un squelette est négligeable. Pour ce faire, nous avons évalué le temps d'exécution d'un

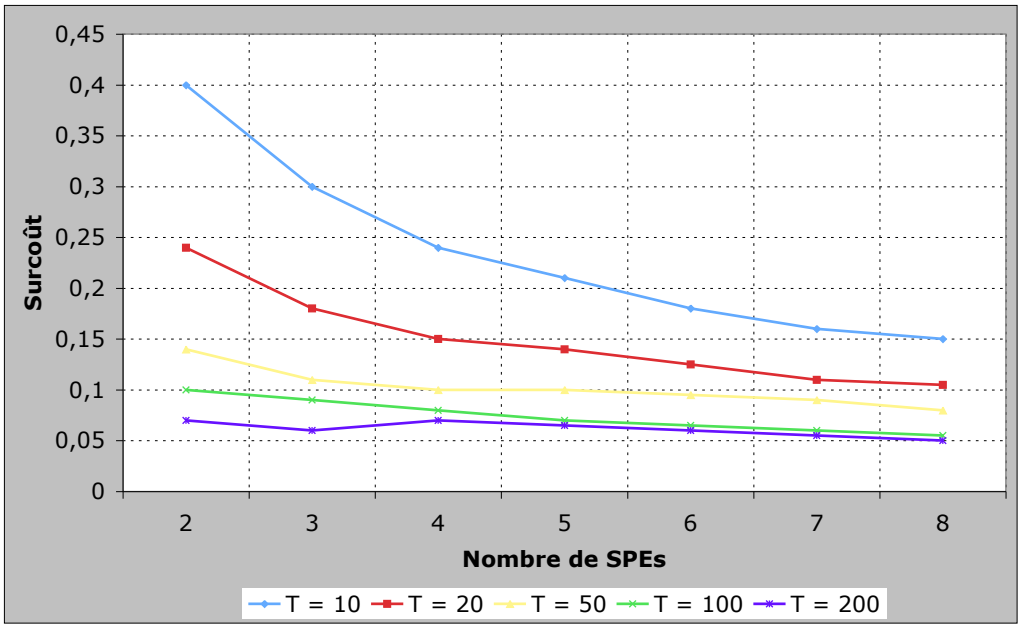

FIGURE 4.7 – Surcoût en % du squelette PARDO en fonction du nombre de SPEs, pour différentes valeurs de durée de fonction en ms

squelette SKELL BE représentatif en augmentant le nombre de SPEs mis en jeu et l'avons comparé à un code source similaire écrit à la main. Les premiers résultats permettent de constater que l'exécution d'une fonction au travers des opérateur CHAIN ou SEQ , n'induit pas un sourcoût important par rapport à l'implémentation manuelle du code. 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 Pipeline construit un *pipeline* de 2 jusqu'à 8 SPEs dans lequel la quantité de transferts de données est négligeable. Chaque étage de ce *pipeline* exécute une fonction dont la durée est comprise entre 10 ms et 200 ms. Les mêmes tests ont été effectués pour le squelette PARDO avec une fonction qui dure entre 10 ms et 200 ms et un nombre de SPEs variant de 2 à 8. Les figures 4.7 et 4.8 illustrent les résultats des mesures. On peut

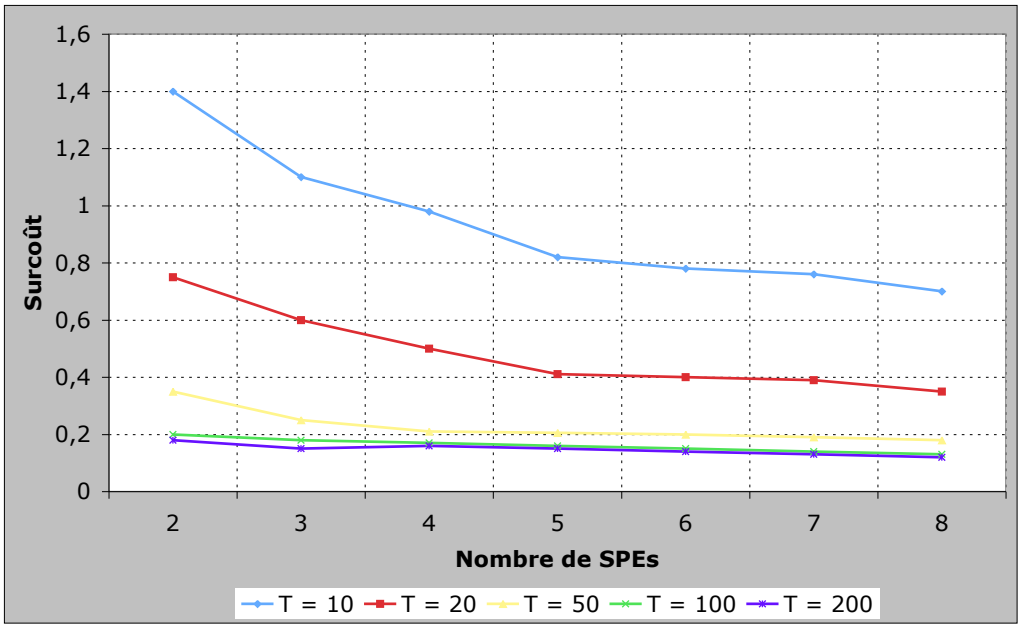

FIGURE 4.8 – Surcoût en % du squelette PIPE en fonction du nombre de SPEs, pour différentes valeurs de durée de fonction en ms

ainsi constater que le surcoût ne dépasse jamais 1.5 % pour les différentes valeurs de la durée des fonctions de tests exécutées (10 ms à 200 ms). Le sourcoût diminue également avec l'augmentation du nombre de SPEs ce qui peut paraître étonnant à première vue, mais qui est justifié par le fait que la mesure ne tiens pas en compte du sourcoût induit par la création du *thread* SPE mais celui engendré par la génération de code SPE qui est amortie au fur et à mesure que le nombre de SPEs augmente.

4.8.2 Benchmarks de passage à l'échelle

L'évaluation qui suit, mesure le passage à l'echelle (*scalability*) de notre outil `SKELL BE` et fait une comparaison avec d'une part, un code équivalent écrit à la main et d'autre part un code *OpenMP* compilé avec *XLC single source compiler* pour le Cell. La mesure s'est faite sur certains noyaux de calculs de l'API *BLAS 1.2*[30] (*Basic Linear Algebra Subprograms*), produit

SPE	OMP (ms)	Manual (ms)	SKELL BE (ms)	*Surcoût*
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 %

TABLE 4.2 – Résultat du benchmark du produit scalaire `DOT`

scalaire (DOT), convolution 3x3 (CONVO) et la multiplication matrice vecteur (SGEMV) (Tab. 4.2, 4.3 et 4.4).L'évaluation du passage à l'échelle se fait en mesurant une accélération relative en comparaison avec l'exécution sur un SPE. Le PPE ayant une architecture foncièrement différente de celle des SPEs, nous ne l'avons pas considéré comme référence pour la mesure du passage à l'échelle. Sauf mention du contraire, les benchmarks effectués adoptent tous un squelette du type **Farm** donc *data parallel* par définition.

Le noyau `DOT`

Dans ce programme nous effectuons le produit scalaire de deux tableaux d'un milliard d'éléments flottants simple-précision. La versions OpenMP utilise une directive de réduction alors que les versions écrite à la main et `SKELL BE` collectent explicitement 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 $\times$7,98. La version OpenMP est limitée par le surcoût induit par la gestion implicite des communications et de la synchronisation. Dans la même situation, `SKELL BE` fournit un accélération maximale de $\times$7,85 ce qui représente un surcoût 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$\times$4096 avec un masque de taille 3$\times$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 $\times$2,24 et celui de `SKELL BE` mesuré à $\times$6,52. Le surcoût quant

SPE	OMP(ms)	Manual (ms)	SKELL BE (ms)	*Surcoût*
1	2402	649	672	3.6%
2	4289	391	411	5.0 %
4	2146	172	181	5.2 %
8	1073	98	103	5.4%

TABLE 4.3 – Résultat du benchmark de la convolution 3x3 `CONVO`

SPE	OMP (ms)	Manual (ms)	SKELL BE (ms)	*Surcoût*
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%

TABLE 4.4 – Résultat du benchmark de la multiplication matrice vecteur `SGEMV`

à lui a augmenté à cause notamment de la gestion des transferts non-alignés (qui se fait à l'exécution) du noyau de convolution. Il est autour de 5%.

Le noyau `SGEMV`

Ce benchmark est un produit entre une matrice 4096×4096 et un vecteur 4096×1. On remarque la même chose que dans le cas précèdent, c'est à dire un bon passage à l'échelle de `SKELL BE` avec un sourcoût toujours autour de 5%.

4.8.3 Algorithme de Harris

Cette application est plus complexe que celles vues 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 le traitement d'images. Plusieurs schémas de parallélisation sont possibles pour cet algorithme, comme vu dans le chapitre 2. Nous avons choisi ici trois versions de déploiement : une version complètement chaînée (Fig. 4.9) où tous les opérateurs sont regroupés au sein d'un seul et même SPE et dupliqués 8 fois ; la versions chaînée à moitié (Fig. 4.10), où les opérateurs sont chaînés deux à deux sur un SPE, un *pipeline* est ensuite formé entre deux SPEs ce qui permet de dupliquer 4 fois ; Et enfin une

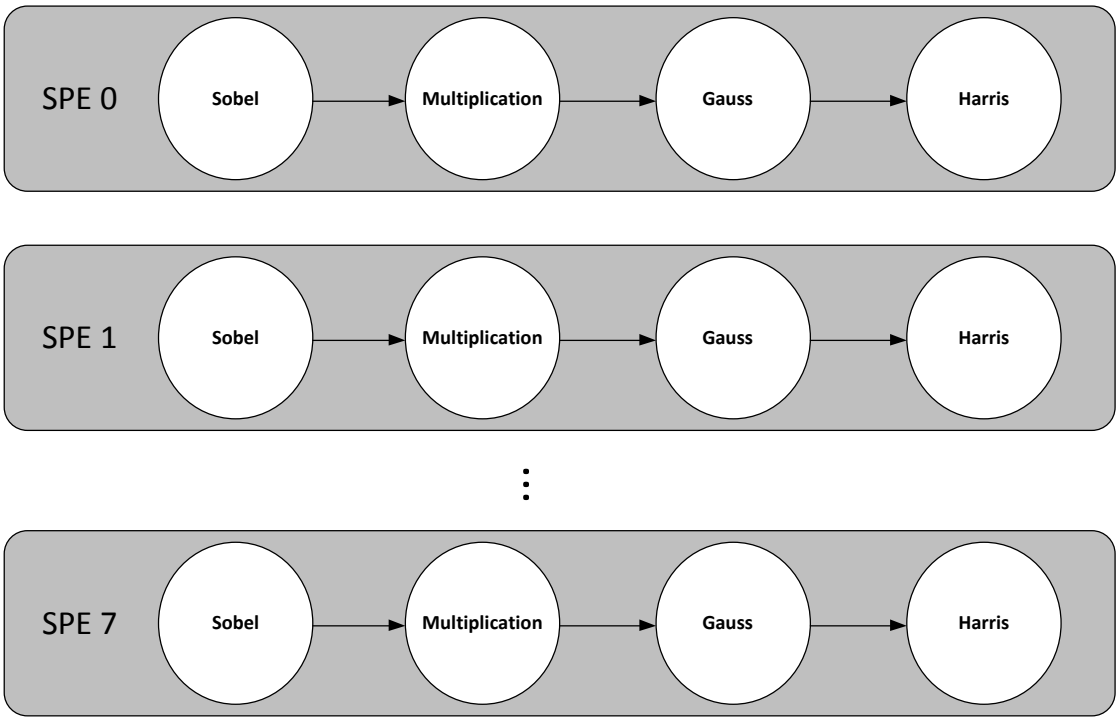

FIGURE 4.9 – Squelette de la version complètement chaînée

version où chaque opérateur occupe un SPE (Fig. 4.11), ce qui permet de dupliquer 2 fois. Le listing 4.9 représente les *kernels* SPE des différentes versions. Le chaînage y est représenté par une virgule (,) et le *pipeline* par un opérateur |.

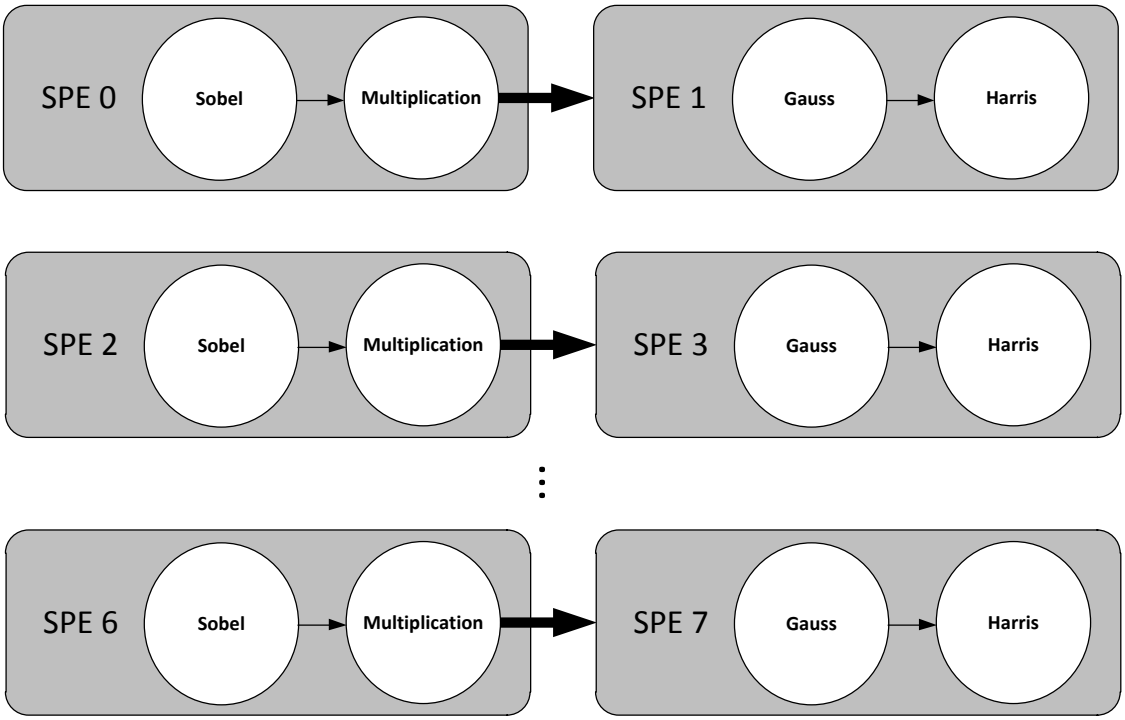

FIGURE 4.10 – Squelette de la versions chaînée à moitié

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

Listing 4.9 – Les *kernels* SPE des implémentations de l'opérateur de Harris

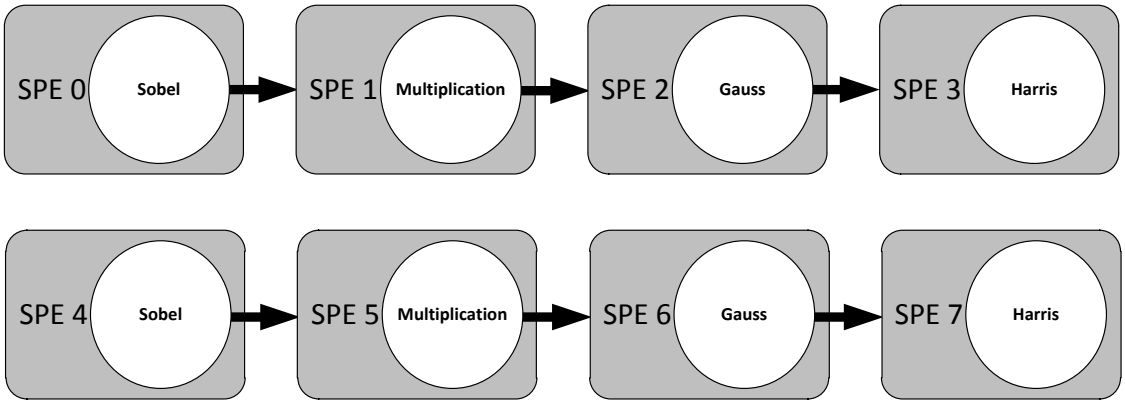

FIGURE 4.11 – Squelette de la version non chaînée

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 4.5). Le sourcoût est également de 5% ce qui valide notre approche car le compromis sourcoût rapidité de mise en oeuvre est très bon.

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

TABLE 4.5 – Benchmarks de l'algorithme de Harris

4.8.4 Impact sur la taille de l'exécutable

La *méta-programmation* est souvent blâmée par-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 Ko) 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 Ko du *local 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*

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

TABLE 4.6 – Impact sur la taille de l'exécutable

qui englobe le code de tous les SPEs, ce qui donne approximativement un code 8 fois plus important. 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.

4.8.5 Impact sur le temps de compilation

L'autre problème 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 qui englobe les directives du pré-processeur qui gèrent les fonction 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.

4.8.6 Gain en expressivité

Un des principaux apports de la programmation par squelettes algorithmiques et le gain en expressivité. En effet, tous les aspects liés à la création des *threads*, à la distribution des données et à la synchronisation ne sont pas exposés au programmeur. La représentation du schéma de calcul sous forme de squelette permet la génération du code de calcul, de transfert et de synchronisation correspondant. On peut ainsi constater à partir des listings 4.10 et 4.11, le gain considérable en expréssivité. On peut comparer ce code source avec celui de la ligne 3 dans le listing 4.9.

```
1 /* fichier : harris_spmd_ppe.c
2  */
3 // inclusion des fichiers d'entete standard
4 ...
5 /* la stucture contient des informations qui servent
6  * a la creation des threads dans les SPE
7  */
8 typedef struct ppu_pthread_data {
```

```
9     spe_context_ptr_t speid;
10    pthread_t pthread;
11    void *argp;
12  } ppu_pthread_data_t;
13  /* cette fonction est executee Ãă la creation du thread */
14  void *ppu_pthread_function(void *arg){
15    ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
16    int rc;
17    unsigned int entry = SPE_DEFAULT_ENTRY;
18    if ((rc = spe_context_run(datap->speid, &entry, 0, datap->argp, NULL, NULL))
          < 0){
19      fprintf (stderr, "Failed spe_context_run(rc=%d, errno=%d, strerror=%s)\n"
            , rc, errno, strerror(errno));
20      exit (1);
21    }
22    pthread_exit(NULL);
23  }
24  //pointeurs sur les threads SPE
25  extern spe_program_handle_t spe_spmd;
26  ppu_pthread_data_t* datas;
27
28  int main(int argc,char **argv)
29  {
30    // declarations des variables et des pointeurs
31      ...
32    // allocation du control block et de la structure contenant son adresse
33    ...
34    // initialisation des parametres
35    ...
36    // allocation des buffers entrees/sorties
37      ...
38    // initialisation des buffers
39    ...
40    // initialisation du control block
41      ...
42     /* allocation des taches SPE */
43    for (i = 0; i < spu_threads; i++){
44       // creation du contexte
```

```
45      if ((datas[i].speid = spe_context_create (0, NULL)) == NULL){
46        fprintf (stderr, "Failed spe_context_create(errno=%d strerror=%s)\n",
              errno, strerror(errno));
47    exit (3+i);
48      }
49      // chargement du programme
50      if ((rc = spe_program_load (datas[i].speid, &spe_spmd)) != 0){
51         fprintf (stderr, "Failed spe_program_load(errno=%d strerror=%s)\n",
               errno, strerror(errno));
52     exit (3+i);
53       }
54     // initialisation du poiteur qui contient l'adresse des control blocks
55     datas[i].argp = (unsigned long long *)(&element[i]);
56       // creation des thread
57       if ((rc = pthread_create (&datas[i].pthread, NULL, &ppu_pthread_function,
              &datas[i])) != 0){
58         fprintf (stderr, "Failed pthread_create(errno=%d strerror=%s)\n", errno,
               strerror(errno));
59         exit (3+i);
60       }
61    }
62    /* Barriere de synchronisation des SPE */
63    for (i=0; i<spu_threads; ++i){
64    // pthread join
65      if ((rc = pthread_join (datas[i].pthread, NULL)) != 0){
66        fprintf (stderr, "Failed pthread_join(rc=%d, errno=%d strerror=%s)\n", rc
              , errno, strerror(errno));
67        exit (1);
68    }
69      // Destruction du contexte
70      if ((rc = spe_context_destroy (datas[i].speid)) != 0){
71        fprintf (stderr, "Failed spe_context_destroy(rc=%d, errno=%d strerror=%s)
              \n", rc, errno, strerror(errno));
72    exit (1);
73      }
74  }
75  //Liberation de la mÃľmoire
76    ...
```

```
77      return (0);
78  }
```

Listing 4.10 – Exemple d'implantation de la version SPMD en *pthreads* : code du PPE

```
1   /* fichier : harris_spmd_spe.c
2    */
3   // inclusion des fichiers d'entete standard
4   ...
5   int main (unsigned long long spu_id __attribute__ ((__unused__)), addr64 parm)
6   {
7     // declaration des buffers de donnees
8     vFloat32 *B0, *B1, *B2, *B3;
9     vFloat32 *B4, *B5, *B6, *B7, *B8;
10    // copie du control block dans le SPE
11      ...
12      // copie du control block de la memoire vers le SPE
13      ...
14    // allocation des buffers de donnees
15      ...
16    // filtrage de Sobel sur toutes les tuiles
17    for (i=0;i<nbtiles;i++)
18    {
19        // transfert d'une tuile de memoire ext vers le SPE
20        get_tile(source0, B5, tile1_h*tile1_w,0);
21        //calcul du filtrage Sobel
22        sobel_RED_SPE_1F32(B5+tile1_w,tile2_h,tile2_ws,B0,B1);
23        // transfert d'une tuile Ix vers la memoire ext
24        put_tile(source10,B0,tile2_h*tile2_w,1);
25        // transfert d'une tuile Iy vers la memoire ext
26        put_tile(source11,B1,tile2_h*tile2_w,2);
27        // tuiles suivantes
28        source0+=(tile2_h*tile2_w);
29      source10+=(tile2_h*tile2_w);
30        source11+=(tile2_h*tile2_w);
31    }
32    // reinitialisation du poiteur vers les donnÃ©es source
33    source10=(vFloat32*)elem.ad_source10;
34    source11=(vFloat32*)elem.ad_source11;
```

```
35    source20+=tile2_w;
36    source21+=tile2_w;
37    source22+=tile2_w;
38    for (i=0;i<nbtiles;i++)
39    {
40        // transfert d'une tuile Ix de la memoire ext
41        get_tile(source10, B2,tile2_h*tile2_w,3);
42        // transfert d'une tuile Iy de la memoire ext
43        get_tile(source11, B3,tile2_h*tile2_w,4);
44        // calcul du produit Ixx=Ix*Ix
45        mul_SPE_1F32(B2,B2,tile2_h,tile2_ws, B0);
46        // calcul du produit Ixy=Ix*Iy
47        mul_SPE_1F32(B2,B3,tile2_h,tile2_ws, B1);
48        // calcul du produit Iyy=Iy*Iy
49        mul_SPE_1F32(B3,B3,tile2_h,tile2_ws, B4);
50        // transfert d'une tuile Ixx vers la memoire ext
51        put_tile(source20, B0,tile2_h*tile2_w,5);
52        // transfert d'une tuile Ixy vers la memoire ext
53        put_tile(source21, B1,tile2_h*tile2_w,6);
54        // transfert d'une tuile Iyy vers la memoire ext
55        put_tile(source22, B4,tile2_h*tile2_w,7);
56        // tuiles suivantes
57        source10+=(tile2_h*tile2_w);
58        source11+=(tile2_h*tile2_w);
59        source20+=(tile2_h*tile2_w);
60        source21+=(tile2_h*tile2_w);
61        source22+=(tile2_h*tile2_w);
62    }
63    // reinitialisation du poiteur vers les donnÃ©es source
64    source20=(vFloat32*)elem.ad_source20;
65    source21=(vFloat32*)elem.ad_source21;
66    source22=(vFloat32*)elem.ad_source22;
67    for (i=0;i<nbtiles;i++)
68    {
69      // transfert d'une tuile Ixx de la memoire ext
70        get_tile(source20, B5,tile1_h*tile1_w,8);
71        // transfert d'une tuile Ixy de la memoire ext
72        get_tile(source21, B6,tile1_h*tile1_w,9);
```

```
73      // transfert d'une tuile Iyy de la memoire ext
74        get_tile(source22, B7,tile1_h*tile1_w,10);
75        // calcul du Gaussien Sxx
76        gauss_RED_SPE_1F32(B5+tile2_w,tile2_h, tile2_ws, B0);
77        // calcul du Gaussien Sxy
78      gauss_RED_SPE_1F32(B6+tile2_w,tile2_h, tile2_ws, B1);
79        // calcul du Gaussien Syy
80        gauss_RED_SPE_1F32(B7+tile2_w,tile2_h, tile2_ws,B2);
81        // transfert d'une tuile Sxx vers la memoire ext
82        put_tile(source30, B0,tile2_h*tile2_w,11);
83        // transfert d'une tuile Sxy vers la memoire ext
84        put_tile(source31, B1,tile2_h*tile2_w,12);
85        // transfert d'une tuile Syy vers la memoire ext
86        put_tile(source32, B2,tile2_h*tile2_w,13);
87        // tuiles suivantes
88        source20+=(tile2_h*tile2_w);
89      source21+=(tile2_h*tile2_w);
90        source22+=(tile2_h*tile2_w);
91        source30+=(tile2_h*tile2_w);
92        source31+=(tile2_h*tile2_w);
93        source32+=(tile2_h*tile2_w);
94    }
95    // reinitialisation du poiteur vers les donnees source
96    source30=(vFloat32*)elem.ad_source30;
97    source31=(vFloat32*)elem.ad_source31;
98    source32=(vFloat32*)elem.ad_source32;
99    for (i=0;i<nbtiles;i++)
100   {
101     //transfert d'une tuile Sxx de la memoire ext
102       get_tile(source30, B3,tile2_h*tile2_w,14);
103       // transfert d'une tuile Sxx de la memoire ext
104       get_tile(source31, B4,tile2_h*tile2_w,15);
105     // transfert d'une tuile Syy de la memoire ext
106       get_tile(source32, B0,tile2_h*tile2_w,16);
107     // calcul de la coarsite Sxx*Syy-Sxy*Sxy
108       harris_SPE_1F32(B3, B4, B0, tile2_h, tile2_ws, B8);
109     // transfert d'une tuile K vers la memoire ext
110       put_tile(target30, B8,tile2_h*tile2_w,17);
```

```
111         // tuiles suivantes
112         source30+=(tile2_h*tile2_w);
113         source31+=(tile2_h*tile2_w);
114         source32+=(tile2_h*tile2_w);
115         target30+=(tile2_h*tile2_w);
116     }
117     // deallocation de la memoire
118       ...
119     return 0;
120 }
```

Listing 4.11 – Exemple d'implantation de la version SPMD en *pthreads* : code du SPE

4.9 Conclusion

SKELL BE est une solution à base de langage spécifique à un domaine, enfouis dans *C++*. Cet outil 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 présentés tendent à prouver que l'on obtient de bonnes performances tant au niveau temporel brut qu'au niveau du passage à l'échelle. Les mesures sur des squelettes simples ont prouvé que le sourcoût 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` selon plusieurs schémas de parallélisation, permet d'apprécier la flexibilité et l'expressivité fournies par l'outil. Les performances obtenues sur l'algorithme de Harris prouvent l'efficacité de `SKELL BE` sur une application de moyenne complexité.

Dans les chapitres précédents nous avons étudié le processeur Cell et ses contraintes de programmation. Nous avons également étudié le portage d'une application de traitement d'images bas-niveau, sur cette architecture parallèle. Nous n'avons pas pu mener une comparaison plus large avec tous les outils présentés dans le chapitre précédent exception faite de OpenMP. *RapidMind* est devenu un outil payant que nous n'avons pas pu nous procurer et les autres outils libres n'était pas suffisamment stables pour pouvoir mener une telle étude.

Le chapitre suivant est consacré à une étude comparative entre le Cell est d'autres architectures parallèles du marché ayant leurs propres outils de programmation. Cette étude nous est

parue pertinente pour tirer une conclusion globale sur l'adéquation du processeur Cell pour notre domaine d'application à savoir le traitement d'images bas-niveau.

Chapitre 5

Comparaison avec les autres architectures parallèles

Le chapitre précédent a permis d'expérimenter plusieurs techniques d'optimisation pour le processeur Cell, sur un algorithme de traitement d'images bas-niveau : le détecteur de point d'interêts de Harris. Afin d'avoir une évaluation plus complète, il est nécessaire d'étudier d'autres architecture parallèles qui présentent un potentiel aussi intéressant que celui du Cell pour le traitement d'images. Dans ce chapitre, on se propose de comparer l'implantation du même algorithme de détection de points d'intérêts de Harris sur d'autres architectures parallèles émérgentes, utilisant éventuellement d'autres modèles de programmation que ceux utilisés pour le processeur Cell. Les architectures considérées dans cette étude sont des architectures du type SMP à mémoire partagée (Intel et PowerPC) ainsi que les cartes graphiques Nvidia et leur langage de programmation CUDA (*Compute Unified Device Architecture*). Les travaux qui sont exposés dans ce qui suit ont été publiés dans [21] et [83].

5.1 Les processeurs multi-coeurs

Ce type de processeurs constitue aujourd'hui le *main stream* en terme de conception d'architectures parallèles et est également le plus répandu dans les machines grand public. Le parallélisme y est présent à plusieurs niveaux car ces architectures reprennent les concepts des architectures classiques mono-coeurs et dupliquent les coeurs afin d'obtenir un niveau de parallélisme supérieur. La mémoire y est généralement partagée et la hiérarchie mémoire est

basée sur plusieurs niveaux de caches communs ou pas. Les modèles et outils de programmation utilisés pour tirer profit du parallélisme de ces architectures sont les bibliothèques de *threads Pthread* ainsi que les directives de compilation *OpenMP*. En ce qui concerne les optimisations bas-niveau comme celles au niveau des instructions ou certaines optimisations SIMD, elles sont généralement bien gérées par les compilateurs modernes. La cohérence mémoire est assurée par le matériel à travers une hiérarchie mémoire à plusieurs niveaux de caches.

5.2 Les GPU Nvidia et CUDA

Les cartes graphiques (*GPU*) ont connu ces dernières années un essor particulier, car elles ont subi une véritable révolution dans leur domaine d'utilisation. Les premiers pas du *GPGPU* (*General-Purpose computing on Graphics Processing Units*) [82] ont consisté en l'utilisation de langages de rendu graphique, *Cg* par exemple, pour en faire un usage plus généraliste, à savoir le calcul intensif pour des domaines d'applications qui relèvent plus du *HPC*. Cette première évolution a démontré que l'architecture des processeurs graphiques était bien adaptée au calcul intensif, mais le détournement des langages graphiques pour un usage généraliste était trop complexe pour le développeur habitué à programmer en langages *C/C++*. La définition d'un modèle de programmation approprié est alors devenue une nécessité pour les fabricants de cartes graphiques. C'était également un moyen pour eux de concurrencer les constructeurs de processeurs généralistes sur le marché du *HPC*.
CUDA[79] (*Compute Unified Device Architecture*) de Nvidia est sans doute la plus importante des initiatives dans ce sens. En effet, CUDA constitue un écosystème complet de programmation parallèle sur les architectures des cartes graphiques. Il définit à la fois une architecture matérielle constituée d'un ensemble de processeurs parallèles organisés en Multi-processeurs SIMD et une hiérarchie mémoire adaptée au problèmes massivement parallèles, similaires aux traitements caractérisant le rendu graphique avancé. Un modèle de programmation propriétaire CUDA, a été développé pour l'exploitation de ce parallélisme. Celui-ci se décline sous forme de plusieurs outils de programmation :

- une extension du langage C définissant de nouveaux types et mots clés propres aux architectures CUDA ;
- une *API Runtime* permettant l'exécution d'un modèle haut-niveau de programmation

basé sur le parallélisme de données ;
– une *API Driver* permettant un contrôle plus fin de l'application mais aux prix d'une programmation plus verbeuse que celle de l'API de *Runtime*.

5.3 Comparaison des architectures matérielles

Au delà des performances intrinsèques sur un algorithme donné, obtenues sur une architecture donnée, il nous a paru important de comparer dans un premier temps les architectures utilisées lors de la comparaison de l'algorithme de Harris sur les différentes plateformes parallèles émergentes. Cette comparaison peut se faire sous différents critères mais les plus pertinents pour l'exploitation efficace du parallélisme sont les formes de parallélisme et la hiérarchie mémoire. Le tableau 5.1 contient cette comparaison. Au vu des informations contenues dans le tableau, il est important de noter que les architectures même si elles possèdent plusieurs points communs, diffèrent par leur nature. Ainsi, les processeurs multi-core sont des machines complexes, capables de gérer un ou plusieurs systèmes d'exploitation en plus des tâches purement calculatoires, ceci explique leur architecture complexe qui permet d'avoir une généricité dans les tâches exécutées. Les GPU à l'opposé, ont été conçus initialement pour exécuter des tâches de rendu graphique 3D. Leurs architectures sont intimement liées à ce domaine d'application et sont ainsi non-adaptées à d'autres types d'applications que celles massivement parallèles ne contenant pas de flot de contrôle complexe. Elles ne peuvent fonctionner qu'avec la présence d'un processeur hôte de type CPU. Le *Cell* enfin, est une architecture hétérogène qui tient plus des processeurs présents dans l'embarqué que des architectures *HPC* classiques. En effet, celui-ci a été conçu à l'image d'un DSP capable d'exécuter une quantité de calculs flottants en simple et double précision tout en ayant l'avantage des systèmes embarqués temps réel au niveau de la garantie d'un temps d'exécution prédictible. Dans le domaine du traitement d'images, la majorité des applications sont *data parallel* ce qui donne un avantage certain aux architectures GPU en terme d'adéquation avec les algorithmes. Par contre certains facteurs comme les fréquences d'horloge ainsi que les contraintes des bus de transfert mémoire qui sont critiques dans notre domaine d'application, rétablissent un équilibre avec les autres types d'architectures.

Architecture	Multi-core	Cell BE	Nvidia CUDA
Type	Homogène (SMP à mémoire partagée)	Hétérogène (à mémoire distribuée)	SIMD (Stream Processors)
Parallélisme d'instructions	oui	oui	oui (parallélisme entre calculs et transferts)
Parallélisme de données	oui	oui	oui
Parallélisme de tâches	oui	oui	oui depuis Fermi
Hiérarchie mémoire	Partagée (Plusieurs niveaux de cache)	distribuée (mémoire privées dans SPE)	hybride (mémoire partagée, globale, texture ...)
Optimisation de la mémoire	partagée (plusieurs niveaux de cache)	explicite (programmation du DMA)	explicite (utilisation des différents niveaux mémoire)

TABLE 5.1 – Comparaison des niveaux de parallélisme et de la hiérarchie mémoire des architectures matérielles

5.4 Comparaison des modèles de programmation

Les modèles de programmation représentent l'interface entre le développeur et l'architecture matérielle qui permet au premier d'exploiter pleinement les dispositifs de celle-ci. C'est pour cette raison que la comparaison des modèles de programmation des différentes plate-formes parallèles nous a paru nécessaire. Il est évident que la comparaison ne peut pas permettre de dégager une architecture ou un modèle de programmation idéal, mais plutôt une adéquation entre un modèle et une architecture donnée ou encore une adéquation entre ce couple et un domaine d'applications, dans notre cas le traitement d'images. Il faut noter que le tableau 5.2 n'adresse pas les architectures GPU car il n'existait pas d'implantation d'OpenMP pour GPU utilisable lors de nos expérimentations. Toutefois il faut noter qu'un traducteur OpenMP vers CUDA est présenté dans [69]. De plus, l'outil HMPP [29, 12] (*Hybrid Multicore Parallel Programming*) développé par la société CAPS Entreprise adopte une approche à base d'annotation de code très similaire à OpenMP.

5.4.1 Mise en oeuvre du code parallèle

Un des points critiques pour le passage d'un code d'une version séquentielle vers une version partiellement ou entièrement parallèle est celui du temps de développement nécessaire à cette tâche. Afin de simplifier la comparaison, on se place dans le cas où le code possède un fort potentiel de parallélisation et les portions de code exploitables ont été identifiées. L'hypothèse supplémentaire pour avoir une comparaison objective est la possibilité d'exploiter la forme de parallélisme sur toute les architectures et avec les modèles de programmation associés. Dans notre cas, ces hypothèses sont vérifiées en pratique car notre application exploite essentiellement le parallélisme de données, qui est présent sur toutes les architectures et pris en charge par les modèles de programmation OpenMP, Pthreads et CUDA.

Pthreads

Le modèle de programmation par *threads* existe depuis les débuts de la programmation parallèle. C'est un modèle complexe à mettre en oeuvre car très proche de l'architecture. Celui-ci était pendant longtemps la seul alternative pour la programmation du processeur Cell, ce qui a rendu sa programmation fastidieuse, en plus des aspects de gestion explicite

OpenMP	Multi-core	Cell BE
Implémentation	oui	oui
Adéquation avec l'architecture	– Bonne au niveau mémoire (mémoire partagée) – Les SMP sont plus faciles à gérer pour OpenMP	– Mauvaise au niveau mémoire (mémoire distribuée) – L'architecture hétérogène du Cell complique la répartition de charge
Exploitation du parallélisme	– Parallélisme de tâches – Parallélisme de données – Vectorisation et parallélisme d'instructions bien gérés par le compilateur	– Parallélisme de données uniquement – Vectorisation mal gérée par le compilateur
Gestion de la mémoire	Implicite (Gérée par le compilateur)	Implicite (à l'aide du compilateur et d'un cache logiciel)

TABLE 5.2 – OpenMP et les architectures parallèles

PThreads	Multi-core	Cell BE
Implémentation	oui	oui
Adéquation avec l'architecture	– Modèle plus flexible – Programmation très verbeuse	– A constitué pendant longtemps le seul outil de programmation pour le Cell – La flexibilité permet une exploitation plus riche de l'architecture
Exploitation du parallélisme	– Détermination du parallélisme gérée par le programmeur – Synchronisation gérée par le programmeur – Vectorisation et optimisation bas-niveau	– Parallélisme de données et de tâches – Vectorisation mal gérée par le compilateur
Gestion de la mémoire	Explicite (à la main par le programmeur)	Explicite (à la main par le programmeur)

TABLE 5.3 – les Pthreads et les architectures parallèles

CUDA	Architectures Nvidia CUDA
Adéquation avec l'architecture	– Modèle dédié à l'architecture – Programmation de complexité intermédiaire
Exploitation du parallélisme	– Parallélisme de données uniquement (parallélisme de tâches depuis Fermi) – Détermination du parallélisme gérée par le programmeur – Synchronisation gérée par le programmeur – Optimisation des transferts mémoire et du taux d'occupation des multiprocesseurs
Gestion de la mémoire	Explicite (à la main par le programmeur)

TABLE 5.4 – CUDA et les architectures Nvidia

des transferts mémoire. La mise en oeuvre du code parallèle nécessite une refonte complète du code et une attention particulière aux détails ce qui rallonge le temps de développement mais aussi de débogage et de validation.

OpenMP

OpenMP est une infrastructure basée sur des directives de compilation et une API de *runtime* permettant de masquer les aspects les plus désagréables des *Ptheads*. Celui-ci repose en effet sur les threads, tout en évitant au programmeur les aspects bas-niveau de la parallélisation. Il permet des temps de développement assez courts, si le code de base n'est pas très complexe. Il a également l'avantage de permettre de garder le code séquentiel dans sa forme originale (garantie du *code legacy*).

CUDA

La programmation parallèle avec le langage CUDA sur les GPU est globalement accessible aux développeurs *C/C++*. La mise en oeuvre du code parallèle requiert une modification du

Processeur	Référence	Nb coeurs	Techno (nm)	Fréq (Ghz)	Cache (Mo)	TDP(W)
PowerPC G4	PPC4470	1	130	1.0	512 Ko	10
bi PowerPC G5	PPC970MP	2 x 2	90	2,5	2 x 512 Ko	70
Penryn	U9300	2	45	1.0	3	10
Penryn	P8700	2	45	2.53	3	25
Penryn	T9600	2	45	2.8	6	35
Yorkfield	Q9550	4	45	2.8	12	95
bi Yorkfield	X3370	2 x 4	45	3.0	2x12	260
bi Nehalem	X5550	2 x 4	45	2.67	2x8	260
Cell	CellXR8i	8	65	3.2	8 x 256 Ko	70
GeForce	9400M	16	55	1.110	-	12
GeForce	8600M GT	32	65	0.900	-	60
GeForce	120 GT	32	55	1.400	-	50
GeForce	FX 4600	112	65	1.400	-	155
GeForce	285 GTX	240	55	1.460	-	183

TABLE 5.5 – Liste des architectures matérielles évaluées pour la comparaison de performances

code original pour coller au modèle de programmation CUDA. Selon la finesse de contrôle souhaitée, la programmation est plus ou moins complexe. L'API de *Runtime* est plus haute en niveau d'abstraction que l'API *Driver*. La première est accessible aisément au programmeur et permet une mise ne oeuvre rapide. La seconde API plus bas niveau permet un contrôle plus fin de l'application au prix d'une programmation plus bas-niveau et donc plus complexe.

5.5 Architectures matérielles et environnement de développement

Les architectures qui ont servi pour les *benchmarks* sont listées ci-dessous (Tab. 5.5). Le panel comporte à la fois des processeurs du type multi-coeurs PowerPC et Intel, une gamme de GPU Nvidia avec les familles d'architectures G80 et GT200. Enfin, l'architecture hétérogène du Cell est également comparée. L'architecture PowerPC G4 avec un seul coeur est également présentée en tant que référence de la version séquentielle la plus performante.

5.6 Mesure de performance temporelle

L'évaluation des performances s'est faite sur les tailles d'images allant de 128×128 jusqu'à 2048×2048. Les résultats sont donnés en cpp

$$cpp = \frac{temps * Frequence}{taille^2}$$

Les versions implantées sont celles présentées auparavant dans le chapitre 2.
Ce que l'on peut observer à partir des résultats dans le tableau 5.6 en premier lieu, c'est que les transformations visant à optimiser la bande passante mémoire sont applicables à toutes les architectures ciblées pour le benchmark. Cela signifie que les transformations en question sont à un niveau d'abstraction suffisant pour pouvoir être appliquées à tout type d'architecture. On peut noter également une accélération super-linéaire sur la version *Planar* sur le **Intel X3370** due au fait qu'une pression particulière est exercée sur la mémoire cache dans cette version, ce qui dégrade sensiblement la version mono-core. L'autre observation importante est celle concernant l'efficacité. En effet, l'architecture du processeur Cell reste la plus *scalable*, c'est celle où l'accélération pour p cores est la plus proche de la valeur maximale théorique. Ce qui est notable également, c'est que le GPU reste très loin en performance des architectures CPU ou du Cell, car malgré les centaines de *stream processors*, la meilleure version reste loin des meilleures versions sur le Cell ou les multi-core, exception faite de l'architecture GeForece 285 GTX de seconde génération (GT200) qui contient 240 *stream processors* et où les contraintes d'alignement de la mémoire (*memory coalescing*) sont beaucoup plus souples que sur les architectures CUDA de première génération (G80).
Il apparaît également que les GPP, grâce à l'amplitude des gains (l'efficacité cumulée des transformations algorithmiques et optimisations logicielles) sont les plus rapides. Les deux premières places reviennent aux octo-coeurs Yorkfield et Nehalem. Le Cell se classe en quatrième position, en étant 2,9 fois plus lent que le Nehalem. En ne prenant en compte que le temps de calcul, le plus rapide des GPU arrive en troisième position. Mais en prenant en compte le temps de transfert, le temps total pour le 285 GTX est multiplié par 8 et passe à 2,1 ms. Cela ramène les performances d'une machine *many-cores* à celle d'un processeur dual core. Concernant les GPU mobiles, l'écart se creuse, car même en ne prenant en compte que le temps de calcul, le 8600M GT et ses 32 PE n'est qu'au niveau du Penryn P8700 qui n'est plus dans sa zone d'efficacité (sortie de cache). Comparé au Penryn T9600 qui n'a pas encore

eu de sortie de cache, il est alors 5 fois plus lent.

5.7 Mesure d'efficacité énèrgétique

Le second benchmark consiste à estimer l'énergie consommée ($E = t \times P$) en se basant sur les TDP (*Thermal Design Power*) annoncés par les constructeurs. Les résultats dans le tableau 5.7 sont intéressants et paradoxaux. Si c'est effectivement un processeur de la gamme pour mobile qui est le plus efficace (Penryn T9600), ce n'est pas celui qui consomme le moins (Penryn U9300). Ce dernier, avec un TDP de 10 W, fait jeu égal avec l'octo-processeur Yorkfield et un TDP de 190 W. La seconde meilleure performance revient au Cell. La performance du Cell vis-à-vis des processeurs généralistes (Intel, AMD et IBM) avait déjà été observée lors de précédents congrès Super Computing/Top500 avec l'apparition d'un classement de Green Computing : pour les grands besoins en puissance de calcul, le Cell est le modèle de calcul le plus efficace énèrgétiquement : les machines *Roadrunner*, composés de Cell, de GPP et de GPU sont actuellement parmi les plus efficaces (en MFlops/Watt).
Concernant les GPU, ils arrivent tous en fin de classement, même si seul le cpp de calcul est pris en compte. Le plus efficace étant paradoxalement celui qui consomme le plus (285 GTX). Il est intéressant d'observer l'efficacité énergétique de ces machines pour des images plus petites : 300×300 (tableau 5.8). Dans ce cas, toutes les machines sont surdimensionnées : le bi-Yorkfield affichant une cadence de traitement de 26 315 images/sec et le "petit" Penryn U9300 une cadence de 2 777 images/sec. L'ordre de performance est maintenant respecté : le U9300 est le plus efficace et le Cell se positionne devant les processeurs octo-coeurs.
Jusqu'à maintenant, les performances des machines étaient évaluées pour des tailles fixes d'images. Il peut être intéressant de prendre le problème à l'envers et de s'interroger sur l'intervalle de taille d'image pour lequel ces processeurs sont performants.
Si l'efficacité du Cell est constante (absence de cache) et celle des GPU croît avec la taille des images (problème de l'alimentation en données), les GPP ne sont efficaces que tant que les données tiennent dans les caches. Le tiling (automatique ou manuel) des données est nécessaire dans ce cas pour assurer une performance moyenne constante quand la taille des données augmente. Une configuration de type serveur est plus efficace (bus rapide) plus longtemps (2 processeurs quadri-coeurs Yorkfield ont un total de 24 Mo de cache L2) qu'une

Architecture	**ISA**	**Planar** $p = 1$	**Planar** $p = p_{max}$	**HalfPipe** $p = 1$	**HalfPipe** $p = p_{max}$	**Gain**	**Temps**
G4	scalaire	518	-	248	-		
	SIMD	189	-	73.4	-	x7.1	19.2
G5	scalaire	254	79	35	15		
	SIMD	79	43	8.9	2.9	x87.6	0.32
Penryn U9300	scalaire	152.0	94.0	40.2	23.6		
	SIMD	35.6	29.4	13.8	11.8	x12.9	2.58
Penryn P8700	scalaire	151.0	88.3	34.8	22.6		
	SIMD	56.8	56.6	24.0	22.2	x6.8	2.30
Penryn T9600	scalaire	140.7	81.6	32.0	16.1		
	SIMD	53.7	51.6	8.4	4.7	x35.0	0.44
Yorkfield	scalaire	140	38	45	11.0		
	SIMD	48.0	11.0	20.0	6.3	x 22.2	0.59
bi-Yorkfield	scalaire	145	16.9	32	4.3		
	SIMD	55.0	3.6	8.4	1.6	x90.6	0.14
bi-Nehalem	scalaire	49	7.2	24.5	3.4		
	SIMD	18.6	3.3	6.0	1.4	x35.0	0.11
Cell	scalaire	857	402	199	140		
	SIMD	79.5	12.6	29.8	4.0	x217	0.33
GeForce 9400M	scalaire		27.6		21.9	x1.3	5.18
GeForce 8600M	scalaire		11.7		7.7	x1.5	2.23
GeForce GT120	scalaire		10.2		6.9	x 1.5	1.30
Quadro FX4600	scalaire		5.9		4.3	x1.4	0.81
GeForce GTX285	scalaire		2.2		1.5	x1.5	0.26

TABLE 5.6 – Comparaison des implémentations de l'algorithme de Harris sur des architectures parallèles

Architecture	Techno (nm)	Puissance (W)	Temps (ms)	Energie (mJ)
PowerPC G4	130	10	19.24	192.4
PowerPC G5	90	2 x 70	0.32	44.0
Penryn U9300	45	10	2.58	25.8
Penryn P8700	45	25	2.30	57.5
Penryn T9600	45	35	0.44	15.4
Yorkfield	45	95	0.59	56.0
bi-Yorkfield	45	2x95	0.14	26.6
bi-Nehalem	45	2x130	0.11	29.7
Cell	65	70	0.33	22.9
GeForce 9400M	55	12	5.12	62.2
GeForce 8600M GT	80	60	2.11	126.6
GeForce GT120	55	50	1.30	64.8
Quadro FX4600	90	134	0.81	108.6
GeForce GTX285	55	183	0.26	47.6

TABLE 5.7 – Comparaison des architectures en énergie consommée pour des images 512×512

configuration portable (les dual coeurs ont entre 4 et 6 Mo de cache L2). En recalculant l'efficacité des Penryn pour une taille d'image 300×300 au lieu de 512×512, le classement des processeurs efficaces est modifié. Le Penryn U9300 qui était aussi efficace que le bi-Yorkfield devient alors 2,5 fois plus efficace.

La taille des caches est un facteur primordial pour les performances des GPP, plus même que le nombre de coeurs (pour le moment). Les optimisations présentées, en repoussant le seuil des sortie de cache (version Halfpipe) et en diminuant leur amplitude, voire en les annulant (version Fullpipe et a priori version Halfpipe), sont nécessaires pour limiter l'accroissement de la taille des caches tout en ayant de bonnes performances pour des images de grande taille.

5.8 Conclusion

Dans ce qui précède des comparaisons avec plusieurs critères ont été effectuée sur différentes architectures parallèles. Les différents modèles de programmation ont été comparés

Architecture	Techno (nm)	Puissance (W)	Temps (ms)	Energie (mJ)
Penryn U9300	45	10	0.360	3.6
Penryn P8700	45	25	0.157	3.9
Penryn T9600	45	35	0.148	5.2
bi-Yorkfield	45	2x95	0.048	9.1
bi-Nehalem	45	2x130	0.0.38	9.8
Cell	65	70	0.113	7.9

TABLE 5.8 – Comparaison des architectures en énergie consommée pour des images 300×300

en terme de difficulté de mise en oeuvre et d'adéquation avec les architectures matérielles. L'algorithme de détection de point d'intérêts de Harris a été utilisé pour comparer les architectures selon deux critères : la performance temporelle pure et l'efficacité énergétique.
Nous avons évalué les performances de notre algorithme de référence sur des architectures GPP, GPU et le Cell. A travers les métriques utilisées, il a été mis en évidence l'importance des transformations algorithmiques qui combinées aux instructions SIMD et à la parallélisation multi-coeur font que les processeurs généralistes restent compétitifs face aux nouvelles architectures (Cell et GPU) un facteur $\times 90$ par rapport à la version de référence a été atteint. Grâce à cela, ils dépassent les performances brutes du Cell. De plus, et contrairement au Cell, le nombre de coeurs des machines généralistes a réussi à croître rapidement : Intel et AMD annoncent des processeurs octocoeurs et des machines bi ou quadri-processeurs (soit un maximum de 32 coeurs).
Certains processeurs pour serveur sont déclinés en version basse-consommation, ce qui les rend plus compétitifs encore, tout en maintenant un modèle de programmation simple et des temps de développement rapides.

Conclusion générale et perspectives

Les travaux exposés dans ce manuscrit traitent entre autres de la problématique de parallélisation de code de traitement d'images sur le processeur Cell. Les détails de l'architecture particulière du Cell sont étudiés avec ses contraintes de programmation. Une méthodologie de portage de code est proposée aux travers d'un exemple d'implémentation d'un algorithme de traitement d'images bas-niveau. Les outils de programmation haut-niveau de l'architecture sont étudiés et un nouvel outil SKELL BE développé au cours de ces travaux est présenté. Ce manuscrit contient également une comparaison de performances entre architectures parallèles sur le même algorithme de traitement d'images.

Méthodologie de portage d'application de traitement d'images

Nous avons porté un algorithme de traitement d'images bas-niveau : le détecteur de points d'intérêts de Harris sur le processeur Cell. Cette étude a permis de développer une méthodologie d'optimisation de code sur l'architecture du Cell. Plusieurs volets de la parallélisation et de l'optimisation ont été abordés. Nous avons mis en évidence, d'une part des techniques d'optimisation relevant du traitement d'images, et d'autre part des techniques plus bas-niveau et applicables à d'autres domaines d'application.

L'optimisation de l'utilisation de la mémoire a fait l'objet d'une étude approfondie. L'optimisation de la taille des transferts et de la forme des tuiles de données sont des exemples de techniques apportant un gain important en performances. La fusion d'opérateurs est également une technique efficace sur le Cell. L'importance du schéma de parallélisation a été démontrée par le biais de différentes implémentations, *data parallel*, *task parallel* ou encore hybrides.

Une étude comparative des performances temporelles a été menée. La mesure de métriques

de passage à l'échelle a également fait l'objet d'une étude. Au delà de l'expertise acquise en développement et parallélisation de code sur le processeur Cell, l'étude se veut un guide méthodologique pour les concepteurs d'outils de parallélisation et de compilateurs qui sont souvent confrontés à des problèmes de performances du code généré.

SKELL BE : programmation par squelettes algorithmiques pour le processeur Cell

J'ai développé les briques de base ainsi que la première version de la couche de communication **CELL-MPI**. Ces développement ont constitué le point de départ du développement de l'outil de parallélisation pour le processeur Cell SKELL BE, en collaboration avec Joel Falcou, auteur de l'outil QUAFF : un outil similaire dédié aux clusters de machines distribuées. Nous avons participé à la conception de la version du générateur de code pour le Cell. Nous avons été également à l'origine des briques logicielles bas-niveau de communication et de synchronisation utilisées par l'outil de génération de code.
SKELL BE apporte une réponse différente par rapport aux outils existants qui n'adressent que le parallélisme de données. En effet, SKELL BE donne le choix de différents schémas de parallélisation grâce à des squelettes de base qui peuvent être composés avec d'autres squelettes pour former un graphe complexe exploitant à la fois le parallélisme de données et de tâches.

Comparaison avec d'autres architectures parallèles

En plus d'avoir étudié dans le détail l'architecture du Cell et avoir élaboré une méthodologie d'optimisation sur le processeur Cell à travers un exemple d'algorithme de traitement d'images bas-niveau, nous avons étudié les autres architectures parallèles émergente, concurrentes du Cell. Dans cette étude, nous avons mis en avant plusieurs aspects liés au développement et à l'optimisation d'applications sur les architectures parallèles émergentes. Nous avons comparé les modèles de programmation en terme de difficulté de mise en oeuvre. Les performances temporelles, le passage à l'échelle ainsi que l'efficacité énergétique ont également fait l'objet d'une étude comparative. Le but de cette étude est de guider les développeurs de code de traitement d'images dans le choix d'un couple architecture parallèle/outil de programma-

tion pour la mise en oeuvre d'applications exploitant pleinement le parallélisme offert par ces architectures.

Perspectives

La couche logicielle de communication pour le Cell **CELL-MPI** qui a été finalisée après l'achèvement de mes travaux de thèse, a fait l'objet d'un dépôt APP (Agence pour la Protection des Programmes) à l'Université Paris Sud XI. **CELL-MPI** développé dans le cadre de cette thèse à également fait l'objet d'une OMTE (Opération de Maturation Technico-Economique) soutenue par le RTRA (Réseau thématique de recherche avancée) Digiteo.
En ce qui concerne le processeur Cell. Malgré un succès fulgurant lors de sa sortie en 2007, notamment dans le milieu académique, l'architecture du Cell n'a pas connu de version ultérieure à celle conçue initialement. La difficulté de programmation y est pour beaucoup et l'arrivée tardive d'OpenCL pour le Cell à la fin 2009 n'a pas provoqué un engouement nouveau pour le Cell.
Les travaux de cette thèse peuvent trouver plusieurs débouchés dans le futur. Les méthodologies d'optimisation de code étudiées peuvent être utilisées pour la conception ou l'amélioration d'un compilateur optimisant. Par exemple, un compilateur qui supporte la norme OpenMP 3.0 et qui à introduit la notion de tâche, peut utiliser les techniques d'optimisation des transferts et de placement des tâches pour la génération de code automatique de code sur le Cell ou une architecture hétérogène à mémoire distribuée similaire.
L'outil `SKELL BE` basé sur les squelettes algorithmique peut facilement être étendu à d'autre types de machines : les GPUs par exemple. L'approche à base de graphe de processus communicants peut également être utilisée sur des architectures de calcul embarquées. En effet, celles-ci requièrent une réactivité et un comportement prédictible qui se prête bien à une modélisation de l'application sous forme de squelettes. La génération automatique de code peut faciliter le développement d'applications pour ce type d'architectures souvent hétérogènes.
Enfin, dans le cadre du développement d'outils de déploiement de code de plus haut-niveau, à base de description graphique de code (Simulink par exemple), les couches de communication et de génération de code développées, peuvent servir comme interface pour le Cell.

Bibliographie

[1] Enrique Alba, Francisco Almeida, Maria J. Blesa, J. Cabeza, Carlos Cotta, M. Díaz, I. Dorta, Joachim Gabarró, C. León, J. Luna, Luz Marina Moreno, C. Pablos, Jordi Petit, A. Rojas, and Fatos Xhafa. Mallba : A library of skeletons for combinatorial optimisation (research note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 927–932, London, UK, UK, 2002. Springer-Verlag.

[2] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel : an expandable skeleton environment. *Scalable Computing : Practice and Experience*, 8(4) :325–341, December 2007.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in java. *Future Gener. Comput. Syst.*, 19(5) :611–626, July 2003.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. Skie : a heterogeneous environment for hpc applications. *Parallel Comput.*, 25(13-14) :1827–1852, December 1999.

[6] Bruno Bacci, Marco Danelutto, Susanna Pelagatti, Marco Vanneschi, and Salvatore Orlando. Summarising an experiment in parallel programming language design. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, HPCN Europe '95, pages 7–13, London, UK, UK, 1995. Springer-Verlag.

[7] 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 Metascheduler 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.

[8] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye. The IBM RISC System/6000 processor : hardware overview. *IBM J. Res. Dev.*, 34 :12–22, January 1990.

[9] 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*, SC '06, New York, NY, USA, 2006. ACM.

[10] Arnd Bergmann. Linux on Cell Broadband Engine status update. In *Proceedings of the Linux Symposium*, volume 1, pages 21–27, 2007.

[11] R. J. Blainey. Instruction scheduling in the tobey compiler. *IBM J. Res. Dev.*, 38(5) :577–593, September 1994.

[12] Francois Bodin and Stephane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming*, 17 :325–336, December 2009.

[13] George Horatiu Botorog and Herbert Kuchen. Skil : An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 243–, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3) :777–786, August 2004.

[15] R Calkin, R Hempel, H.-C Hoppe, and P Wypior. Portable programming with the parmacs message-passing library. *Parallel Computing*, 20(4) :615 – 632, 1994.

[16] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *Euro-Par*, pages 72–81, 2007.

[17] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Münster Skeleton Library Muesli - A Comprehensive Overview. Technical report, University of Münster, 2009.

[18] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain decomposition and skeleton programming with ocamlp31. *Parallel Comput.*, 32(7) :539–550, September 2006.

[19] Murray Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3) :389–406, March 2004.

[20] Murray I. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989.

[21] Pierre Courbin, Antoine Pédron, Tarik Saidani, and Lionel Lacassagne. Parallélisation d'opéateurs de TI : multi-coeurs, Cell ou GPU ? In *Actes de la Conférence du GRETSI*, 2009.

[22] 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*, pages 210–219. IEEE, April 2007.

[23] L. Dagum and R. Menon. OpenMP : an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1) :46–55, 1998.

[24] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac : Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[25] Marco Danelutto and Massimiliano Stigliani. Skelib : Parallel programming with skeletons in c. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 1175–1184, London, UK, UK, 2000. Springer-Verlag.

[26] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 146–160, London, UK, UK, 1993. Springer-Verlag.

[27] John R. Davey and Peter M. Dew, editors. *Abstract machine models for highly parallel computers*. Oxford University Press, Oxford, UK, 1995.

[28] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20 :85–95, March 2000.

[29] Romain Dolbeau. Hmpp : A hybrid multi-core parallel. In *First Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–5, 2007.

[30] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1) :1–17, March 1990.

[31] A. J. Dorta, J. A. González, C. Rodríguez, and F. De Sande. llc : a Parallel Skeletal Language . *Parallel Processing Letters*, 13(3) :437–448, 2003.

[32] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Syst. J.*, 45(1) :59–84, January 2006.

[33] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[34] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff : efficient c++ design for parallel skeletons. *Parallel Comput.*, 32(7) :604–615, September 2006.

[35] Joel Falcou. *Un cluster pour la Vison Temps Réel Architecture, Outils et Applications*. PhD thesis, Université Blase Pascale - Clermont II, 2006.

[36] Joel Falcou. High Level Parallel Programming EDSL - A BOOST Libraries Use Case. In *BOOST'CON 09*, Aspen, CO, May 2009. BOOSTPRO Consulting.

[37] Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In *PARCO*, pages 243–252, 2007.

[38] Joel Falcou and Jocelyn Sérot. Eve, an object oriented simd library. In *International Conference on Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pages 314–321. Springer, 2004.

[39] 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.

[40] J. F. Ferreira, J. L. Sobral, and A. J. Proenca. Jaskel : A java skeleton-based framework for structured cluster and grid computing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 301–304, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9) :948–960, September 1972.

[42] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1) :67–70, March 1996.

[43] Message Passing Forum. MPI : A Message-Passing Interface Standard. Technical report, Message Passing Interface Forum, Knoxville, TN, USA, 1994.

[44] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12) :1135–1160, November 2010.

[45] Clemens Grelck. Shared memory multiprocessor support for functional array processing in sac. *J. Funct. Program.*, 15(3) :353–401, May 2005.

[46] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter. An Open Source Environment for Cell Broadband Engine System Software. *Computer*, 40(6) :37–47, June 2007.

[47] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5) :532–533, May 1988.

[48] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th ALVEY Vision Conference*, pages 147–151, 1988.

[49] J.L. Hennessy and D.A. Patterson. *Computer Architecture : A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[50] Christoph A. Herrmann and Christian Lengauer. Hdc : A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2/3) :239–250, 2000.

[51] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Z. Horvath, V. Zsok, P. Serrarens, and R. Plasmeijer. Parallel elementwise processable functions in concurrent clean. *Mathematical and Computer Modelling*, 38 :865 – 875, 2003.

[53] IBM. *Cell Broadband Engine Programming Handbook*. IBM, version 1.0 edition, 2006.

[54] IEEE. IEEE Standard . 1003.1c-1995 thread extensions. Technical report, IEEE, 1995.

[55] International Business Machines Corporation. *IBM Full-System Simulator User's Guide : Modeling Systems based on the Cell Broadband Engine Processor*, 2009.

[56] Lionel Lacassagne Joel Falcou, Tarik Saidani and Daniel Etiemble. Programmation par squelettes algorithmiques pour le processeur cell. In *SYMPA '08 : SYMPosium en Architectures nouvelles de machines*, February 2008.

[57] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM J. Res. Dev.*, 51 :503–519, September 2007.

[58] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49 :589–604, July 2005.

[59] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[60] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8) :54–62, August 2003.

[61] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33 :539–543, May 1990.

[62] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures : a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[63] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran : an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[64] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[65] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network : Built for speed. *IEEE Micro*, 26(3) :10–23, 2006.

[66] Herbert Kuchen. A skeleton library. In *Euro-Par*, pages 620–629, 2002.

[67] Arun Kumar, Ganapathy Senthilkumar, Murali Krishna, Naresh Jayam, Pallav Baruah, Raghunath Sharma, Ashok Srinivasan, and Shakti Kapoor. A buffered-mode mpi implementation for the cell be&lt ;sup&gt ;tm&lt ;/sup&gt ; processor. In Yong Shi, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science Ű ICCS 2007*, volume 4487 of *Lecture Notes in Computer Science*, pages 603–610. Springer Berlin / Heidelberg, 2007.

[68] H.T. Kung, C.E. Leiserson, and Carnegie-Mellon University. Dept. of Computer Science. *Systolic Arrays for (VLSI)*. CMU-CS. Carnegie-Mellon University, Department of Computer Science, 1978.

[69] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu : a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[70] Mario Leyton and José M. Piquer. Skandium : Multi-core programming with algorithmic skeletons. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 289–296, Washington, DC, USA, 2010. IEEE Computer Society.

[71] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3) :431–475, May 2005.

[72] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[73] 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.

[74] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, December 2005.

[75] Greg Michaelson, Norman Scaife, Paul Bristow, and Peter King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, 16(3) :181–206, 2001.

[76] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8) :114–117, April 1965.

[77] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *Robotics Institute, Carnegie Mellon University & doctoral dissertation*, 1980.

[78] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks : A retargetable, dynamic compiler and embedded language. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society.

[79] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6 :40–53, March 2008.

[80] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36 :289–311, 2008. 10.1007/s10766-008-0072-7.

[81] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1) :85–102, January 2006.

[82] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[83] Antoine PEDRON, Florence LAGUZET, Tarik SAIDANI, Pierre COURBIN, Lionel LACASSAGNE, and Michele GOUIFFES. Parallelisation d'opérateurs de ti : multi-coeurs, cell ou gpu ? *TS. Traitement du signal*, 27(2) :161–187, 2010.

[84] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics : Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, New York, NY, USA, 2003. pp 336-337.

[85] CORPORATE Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 12(4) :1–86, December 1993.

[86] 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.

[87] Tarik Saidani, Lionel Lacassagne, Samir Bouaziz, and Taj Muhammad Khan. Parallelization strategies for the points of interests algorithm on the cell processor. In *ISPA '07 : Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications*, August 2007.

[88] Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor : A case study on the harris corner detector. In Per Stenstrom, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 177–200. Springer Berlin Heidelberg, 2011.

[89] Tarik Saidani, Stéphane Piskorski, Lionel Lacassagne, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor : A case study of image

processing algorithm. In *MEDEA '07 : Proceedings of the 2007 workshop on MEmory performance*, September 2007.

[90] Sebastian Schaetz, Joel Falcou, and Lionel Lacassagne. Cell-MPI mastering the cell broadband engine architecture through a boost based parallel communication library. In *the 5th Annual Boost Libraries Conference (BoostCon 2011)*, Aspen, CO, USA, 2011. Boost, Boost.

[91] Jocelyn Sérot and Dominique Ginhac. Skeletons for parallel image processing : an overview of the skipper project. *Parallel Comput.*, 28 :1685–1708, December 2002.

[92] V. S. Sunderam. PVM : a framework for parallel distributed computing. *Concurrency : Pract. Exper.*, 2(4) :315–339, November 1990.

[93] Claude Tadonki, Lionel Lacassagne, Tarik Saidani, Joel Falcou, and Khaled Hamidouche. The harris algorithm revisited on the cell processor. In *Proceedings of the 1st International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies HEART 2010*, 2010.

[94] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit : A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[95] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates : The Complete Guide*. Addison-Wesley Professional, 1 edition, November 2002.

[96] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12) :1709–1732, December 2002.

[97] Todd Veldhuizen. Expression templates. Technical Report 5, June 1995.

[98] Todd L. Veldhuizen. C++ templates as partial evaluation. In *PEPM*, pages 13–18, 1999.

[99] John von Neumann. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4) :27–75, 1993.

[100] 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.