

Chapter 1

Introduction

Part I

Vue d'Ensemble de l'Architecture 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-cœur 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 coprocesseurs spécialisés appelés *Synergistic Processor Element* (SPE), un contrôleur mémoire haute-vitesse et une interface de bus à large bande-passante. Le tout intégré sur une seule et même puce.

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

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

Le rôle du PPE est celui d'un chef d'orchestre, il prend en charge l'OS (*Operating System*) et coordonne les SPEs. Au niveau de l'architecture c'est un *PowerPC* 64-bit classique avec une

extension SIMD (VMX), un cache L1 de 32 KB (données et instructions) et un cache L2 de 512 KB. C'est un processeur à exécution dans l'ordre (in-order execution processor), il supporte le dual-issue (parallélisme d'instructions) ainsi que le multi-threading d'ordre 2 (parallélisme de tâches).

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

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

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

Toutes les instructions SIMD sont des instructions que le pipeline peut exécuter à 4 granularités: 16x des entiers 8-bit, 8x des entiers 16-bit, 4x des entiers 32-bits ou des flottants simple-précision. Le processeur SPU est un processeur à exécution dans l'ordre (in-order-execution

processor), il possède deux pipelines d'instructions connus sous les dénominations pair (*even*) et impair (*odd*).

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

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

INSERER FIGURE DU CELL

Le PPE: Power Processor Element

Le PPE est un processeur 64-bit compatible avec l'architecture Power, optimisé au niveau de l'efficacité énergétique. La profondeur de pipeline du PPE est de 23 stages, chiffre qui peut paraître faible par rapport aux précédentes architectures surtout quand on sait que la durée de l'étape a été 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 lors de l'exécution (exécution dans l'ordre). Le processeur entrelace des instructions provenant de deux threads de calcul différents pour optimiser l'utilisation de la fenêtre d'exécution. Les instructions arithmétiques simples s'exécutent et fournissent leur résultat en deux cycles. Les instructions de chargements (loads) s'exécutent également en deux cycles. Une instruction flottante en double précision s'exécute en 10 cycles. Le PPE supporte une hiérarchie conventionnelle de caches avec un cache L1 (de niveau 1) de 32-KB, et un cache L2 de 512-KB.

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

les deux threads.

Le processeur est composé de trois unités : l'unité d'instructions (UI) responsable du chargement, du décodage, des branchements, de l'exécution et de la 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 anciennes générations de PowerPC 970 et effectuent des opérations sur des registres 128-bit de données qui donnent un parallélisme de 2, 4, 8 ou 16, selon le type de données considéré.

Les SPE (Synergistic Processing Element)

Le SPE contient un jeu d'instructions nouveau mais qui n'est autre qu'une version réduite du jeu d'instructions SIMD VMX (AltiVec), mais qui est optimisé au niveau de consommation d'énergie et des performances pour les applications de calcul intensif et de multimédia. Le SPE contient une mémoire locale de 256 KB (scratchpad) qui est une mémoire de données et d'instructions. Les données et les instructions sont transférées de la mémoire centrale vers cette mémoire privée au travers de commandes DMA synchrones et cohérentes qui sont exécutées par le MFC (Memory Flow Controller) qui est présent dans chaque SPE. Chaque SPE peut supporter jusqu'à 16 commandes DMA en suspens. L'unité DMA peut être programmée de trois manières différentes : 1) avec des instructions sur le SPE qui insèrent des commandes DMA dans la file d'attente ; 2) Par la programmation de transferts permettant de faire des accès sur des zones non contigües de la mémoire au travers d'une liste de DMA ; 3) Par l'insertion d'une commande DMA dans la file d'attente d'un autre processeur par les commandes de DMA-write. Afin de faciliter la programmation et de permettre des transferts entre SPEs les mémoires locales sont mappées en mémoire centrale. La présence des mémoires locales introduit un autre niveau dans la hiérarchie mémoire au-dessus des registres. Les temps d'accès de ces mémoires sont de l'ordre du cycle ce qui en fait de bons candidats pour réduire la latence d'accès à la mémoire centrale qui est de l'ordre de 1000 cycles, d'autant plus que le fait que le contrôleur DMA soit indépendant de l'unité donne un niveau de parallélisme supplémentaire. La présence de ces mémoires privées permet différents modes de programmation qui

peuvent tre appliqués au processeur Cell.

Le Bus Interne (Element Interconnect Bus)

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

Part II

Modèles de Programmation pour le

Processeur Cell

1.1 RapidMind

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

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

1.1.1 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 *retained* est utilisé pour collecter dynamiquement des opérations arbitraires sur les **Value** et les **Array** dans les objets **Program**.

le type **Value**

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

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

Les calculs sont exprimés en utilisant les tuples de **Value** et les opérateurs sur ces types peuvent être utilisés directement pour exprimer du parallélisme SWAR (SIMD Within A Register). Les constructions C++ de modularité tels que namespaces, classes et fonctions.

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

le type **Program**

Un objet **Program** contient une séquence d'opérations, ces opérations sont spécifiées par le passage en mode *retained* qui est indiqué par la macro mot-clé **BEGIN**. Normalement, le système fonctionne en mode *immediate*. Dans ce mode les opérations sur un tuple de valeurs s'exécutent à la spécification comme pour une bibliothèque matrice-vecteur classique: les calculs sont effectués sur la même machine que le programme hôte et le résultat est sauvegardé dans le tuple **Value** de sortie. En mode *retained* un nouvel objet **Program** qui est retourné par la macro **BEGIN** est créé. Les opérations dans ce mode ne sont pas exécutées; elles sont symboliquement évaluées et sauvegardées dans l'objet **Program**. La sortie du mode *retained* est marquée par la macro **END**, qui ferme l'objet **Program** et le marque comme étant prêt à être compilé, étape à la suite de laquelle l'objet **Program** est utilisé pour le calcul. Les objets **Program** sont compilés de manière dynamique ce qui permet d'exploiter les caractéristiques bas-niveau de la machine cible.

Il est à noter que même si les types **RapidMind** sont des classes C++, le compilateur est plutôt assimilable à un compilateur FORTRAN et peut ainsi effectuer les mêmes optimisations agressives. Les fonctionnalités du langage C++ sont utilisées pour structurer les calculs et générer le code mais n'est pas utilisé lors de l'exécution.

1.2 OpenMP Cell pour le Compilateur XL

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

Chaque segment de code compris dans une construction parallèle est listé par le compilateur dans une fonction séparée. Le compilateur insère les appels à la librairie de runtime OpenMP dans la fonction parente de la fonction listée. Ces appels aux fonctions de la librairie de runtime vont ainsi invoquer les fonction 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 problématiques qui sont les suivantes:

- **Threads et Synchronisation:** les threads s'exécutant sur le PPE diffèrent de ceux du SPE en termes de capacité de traitement. Le système a été conçu pour prendre en compte la différence entre les deux architectures.

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

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

1.2.1 Threads et Synchronisation

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

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

Pour les niveaux de parallélisme nichés (boucles nichées), chaque thread dans la region 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 tâches. Une tâche SPE peut être l'exécution d'une région parallèle listée (boucle ou section), ou alors l'exécution d'un flush de cache ou encore la participation à une opération de synchronisation par barrière.

Il existe une file d'attente dans la mémoire système correspondant à chaque thread. Quand le thread maître assigne une tâche à un thread, il écrit les informations sur la tâche à la file d'attente de tâche correspondante., incluant le type de tâche, les bornes supérieure et inférieure de la boucle parallèle, ainsi que le pointeur de fonction pour la région de code listée qui doit être exécutée. Une fois que le thread SPE prend une tâche de la file d'attente, il signale au thread maître que l'espace dans la file d'attente est à nouveau libre. Les mécanismes de synchronisation sont assurés au travers de mailbox qui permettent l'échange de messages bloquant ou non-bloquant entre le thread maître et les threads de calcul. Les locks OpenMP sont implémentés grâce aux commandes DMA atomiques.

1.2.2 Génération de Code

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

Le fait que l'architecture du Cell soit hétérogène impose que les fonctions listées contenant des tâches parallèles soit clonées afin d'être exécutables aussi bien par le SPE que le PPE. Le clonage est effectué quand le graphe d'appel global est disponible de telle sorte que le sous-graphe d'un appel à une fonction listée puisse être cloné entièrement. Le clonage permet aussi lors des

étapes d'optimisation ultérieures d'effectuer des optimisations qui dépendent de l'architecture comme la vectorisation de code qui ne peut pas se faire dans une étape commune à cause des différences entre les jeux d'instructions SPU et VMX. Une table de mise en correspondance entre les versions PPE et SPE contient les pointeurs des fonctions listées de telle sorte à ce qu'il n'y ai pas de confusion lors de l'exécution.

A la fin de l'étape TPO, les procédures sur les différentes architectures sont séparées en deux unités de compilation différente et celles-ci sont traitées une par une par le back-end TOBEY.

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 tiens pas dans la mémoire locale. Il existe deux approches pour remédier à ce problème. La première consiste au partitionnement de la section parallèle dans un programme en plusieurs sections de taille moindre et la génération d'un binaire distinct pour chaque sous section. Cette approche est limitée, d'une part par-ce-qu'une sous-section peut avoir une taille pas assez petite pour tenir dans la mémoire locale et d'autre part la complexité générée par la création et la synchronisation de plusieurs threads affecte considérablement les performances.

La deuxième approche qui est celle utilisée dans le compilateur IBM XL, est le partitionnement du graphe d'appel et les *overlays* de code. Le code SPE est ainsi partitionné et un code *overlays* est 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 du *maximum spanning tree*. Le code SPE ainsi généré est intégré dans le code PPE avant d'être exécuté.

1.2.3 Modèle Mémoire

OpenMP spécifie un modèle mémoire *relaxed-consistency, shared memory*. Ce modèle permet à chaque thread d'avoir sa propre vue temporaire de la mémoire. Une valeur écrite dans une variable, ou une valeur lue à partir d'une mémoire peut rester dans la vue temporaire du thread

jusqu'à ce qu'elle soit obligée de partager la mémoire par une opération de flush OpenMP.

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

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

Pour les références irrégulières à la mémoire le software-cache est utilisé. Le compilateur remplace les *texttload* et *texttstore* de et vers ces zones mémoire par des instructions qui vont chercher les adresses effectives, dans le directory du cache. Si une ligne de cache pour l'adresse effective est trouvée (*cache hit*) la valeur dans le cache est utilisée. Dans le cas contraire (*cache miss*) la donnée est récupérée en mémoire via un DMA *get* dans le cas d'une lecture.

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