Les optimisations d'algorithmes de traitement de signal sur les architectures modernes parallèles et embarquées

Jean-Paul Perez-Seva

UNIVERSITÉ de NICE-SOPHIA ANTIPOLIS - UFR SCIENCES

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en SCIENCES

de l'Université de Nice Sophia Antipolis

Spécialité : **INFORMATIQUE**

présentée et soutenue par

Jean-Paul PEREZ-SEVA

Les optimisations d'algorithmes de traitement de signal sur les architectures modernes parallèles et embarquées

Thèse dirigée par **Michel Cosnard** et **Serge Tissot**

Préparée à l'INRIA Sophia Antipolis, projet MASCOTTE, et Kontron Modular Computers SAS

Soutenue le **24 août 2009**

Jury :

Examinateurs	M.	Jean-Claude	Bermond	Directeur de Recherche CNRS
	M.	Ghislain	Oudinet	Professeur ISEN Toulon
	M.	Damien	Jugie	Chargé de Recherche
Directeurs	M.	Michel	Cosnard	Professeur UNSA
	M.	Serge	Tissot	Chargé de Recherche
Rapporteurs	M.	François	Bodin	Professeur Université Rennes
	M.	Yves	Sorel	Directeur de Recherche INRIA

à Audrey...

Remerciements

Je tiens à remercier en premier lieu Michel Cosnard d'avoir accepté d'être le directeur de cette thèse.

Mes remerciements vont conjointement et tout particulièrement à Serge Tissot et Ghislain Oudinet pour leur encadrement, leur soutien et leur confiance qui m'ont permis de mener à bien ce projet.

Je remercie également les rapporteurs de cette thèse François Bodin et Yves Sorel de s'être portés volontaires à la lecture de mon mémoire. Merci également aux autres membres du jury: Damien Jugie et Jean-Claude Bermond qui ont accepté de juger mon travail.

Mon expérience à Austin n'aurait pas été réussie sans l'aide de Joe Eicher et John Gibson. Je remercie ainsi Joe Eicher pour son accueil sur le continent américain et pour ses corrections apportées à mon papier traduit dans la langue de Shakespeare. De même pour John Gibson d'avoir supporté une répétition de mon exposé et de m'avoir fait ses retours.

Je tiens à remercier une dernière fois Michel Cosnard, Ghislain Oudinet et Serge Tissot pour leur relecture de mes travaux et de m'avoir soumis leurs corrections tant sur la forme que sur mon français parfois approximatif. Merci également à ma compagne, Audrey, pour sa relecture et son intérêt pour mes travaux.

Enfin, je tiens à montrer ma gratitude à tout le soutien apporté par mes proches, ma famille, plus particulièrement mes parents, mon frère et mon amie pour leurs encouragements; sans oublier mes amis qui ont su me motiver en s'informant régulièrement de l'avancement de mon travail.

Cette thèse n'aurait sûrement pas abouti sans cet entourage qui a su apporter, chacun à leur manière, une aide réconfortante et précieuse. Merci.

Table des matières

Table des figures

Table des tableaux

1. Intr` ducti` n

Kontron Modular Computers SAS (KOM-SA) est une entreprise implantée sur le marché de l'embarqué et spécialisée dans la conception de cartes électroniques de calcul. Le marché de l'embarqué est un milieu exigeant où diverses notions opposées sont mises en jeu. La consommation et la dissipation thermique sont deux caractéristiques liées prises en considération dès la conception d'un système. Ainsi l'utilisation de chaque produit conçu par KOM-SA est certifiée suivant des contraintes de températures et de vibrations. Les performances de calcul est aussi déterminante dans la conception du système.

Sur un processeur donné, la consommation et la dissipation augmentent avec la fréquence. De même, les performances du processeur sont proportionnelles à la fréquence. Ce sont donc deux problèmes contraires qui imposent de faire un compromis entre dissipation et performances. C'est pourquoi on observe de manière générale un écart de performances en fréquence entre les processeurs destinés au marché de l'embarqué et ceux destinés au marché grand public et serveur.

La puissance de calcul est néanmoins toute aussi omniprésente dans le domaine embarqué que la dissipation et la consommation. La détection RADAR ou SONAR sont deux exemples d'applications exigeantes où les latences de calcul sont très faibles et déterminantes. Ainsi les performances du processeur influent directement sur le nombre de calculateurs constituant le système et par conséquent sur le poids, la consommation totale, le prix et la compétitivité du système.

Le milieu des microprocesseurs est un marché en constante évolution. L'enjeu est de proposer toujours plus de puissance de calcul. Cette évolution est souvent représentée par les lois de Moore. La première loi de Moore, énoncée en 1965 par Gordon Moore, un des trois fondateurs d'Intel, voulait que la complexité des semi-conducteurs proposés en entrée de gamme double tous les ans à coût constant. Le circuit le plus performant de l'époque comportait alors 64 transistors. Cette loi fut une extrapolation empirique de l'observation de l'évolution des semi-conducteurs de l'époque. Elle fut ensuite réajustée, toujours par Gordon Moore en 1975, en proposant que le nombre de transistors des microprocesseurs sur une puce de silicium double tous les deux ans. Cette dernière s'est avérée étonnamment juste entre 1971 et 2001 où la densité des transistors a doublé tous les 1,96 années.

Depuis 2001, les différents fabricants de microprocesseurs ont éprouvé de nombreuses difficultés à faire croître leurs produits de façon à suivre cette loi de Moore. La plupart se sont heurtés à des courants de fuite de plus en plus conséquents, dus à la réduction de la taille des transistors de moins en moins maîtrisée, causant des consommations et des dissipations trop importantes. Si les lois de Moore sont aujourd'hui encore respectées, ceci est dû à un changement radical de philosophie dans le design des microprocesseurs.

Un certain nombre de techniques d'accélération matérielle se sont vues intégrées dans les microprocesseurs afin de rattraper ce retard de montée en fréquence. Ces techniques, qui seront détaillées ultérieurement, sont effectivement des choix judicieux mais leur exploitation nécessite le plus souvent une connaissance plus poussée de l'architecture du processeur cible, soit parce qu'elles dépendent des choix faits par les constructeurs, soit parce qu'elles imposent l'usage de langages spécifiques qui suppriment tout l'intérêt d'un langage haut niveau tel que le langage C par exemple.

Ainsi, l'exploitation maximale des performances du processeur peut permettre à KOM-SA de réduire le nombre de calculateurs au minimum et de répondre de manière compétitive à chacun de ses clients. Ceci concerne essentiellement l'usage de bibliothèques constituées d'algorithmes optimisés de traitement de signal et d'image.

L'optimisation d'un algorithme pour un processeur demande une forte connaissance de l'architecture ciblée. Il existe de nombreuses méthodes d'optimisation proposées par les différents processeurs. Ces méthodes vont de l'exploitation de la profondeur du *pipeline* à la multiplication du nombre de cœurs de calcul en passant par le parallélisme d'exécution interne. Leur exploitation exige une maîtrise approfondie du fonctionnement des microprocesseurs et l'usage d'une programmation de bas niveau proche de l'assembleur.

C'est donc un travail difficilement automatisable, long et minutieux qui nécessite l'apport d'un oeil expert. La prise en compte de paramètres bas niveau entraîne une dépendance entre l'architecture et les optimisations apportées à algorithme. La problématique de ce travail de recherche est donc de proposer une méthodologie d'optimisation d'algorithmes de traitement de signal sur les processeurs embarqués modernes. Afin d'être le plus exhaustif possible, ce document est structuré comme suit:

Construit autour des trois notions clés de la problématique, un état de l'art abordera, dans un premier temps, le sujet des microprocesseurs embarqués actuels et futurs. Il présentera ensuite les méthodes d'optimisation existantes à ce jour pour finir sur les algorithmes de traitement de signal.

À travers l'optimisation de la transformée de Fourier rapide sur PowerPC 970 FX, nous chercherons à mettre en évidence les enjeux et les techniques utilisés. La synthétisation de ce travail permettra de définir notre méthodologie d'optimisation.

Le chapitre qui suit aura pour but de généraliser notre méthodologie d'optimisation sur l'aspect multi-architecture de la problématique.

Enfin, avant de conclure, nous présenterons les résultats obtenus suite à ce travail de recherche.

2. Quelques n` ti` ns imp` rtantes

Cet état de l'art veillera à parcourir l'intégralité des thèmes qui ont trait à la problématique de recherche étudiée. Il est donc construit autour de trois chapitres distincts. Le premier concerne les architectures processeur de type embarqué. Le but de ce chapitre est de fournir une liste exhaustive d'architectures embarquées actuelles, chacune accompagnée d'une description détaillée de son fonctionnement interne. Nous présenterons ensuite les méthodes existantes permettant d'exploiter le maximum de performances de nos processeurs embarqués. Le troisième et dernier volet traitera du domaine d'application de notre étude, c'est-à-dire les algorithmes de traitement de signal.

2.1. Les n` uvelles architectures micr` pr` cesseur

Les processeurs actuels sont devenus des systèmes complexes capables de faire plusieurs calculs simultanément, d'opérer sur plusieurs données à la fois, de prédire les différents branchements du programme ou encore de décider de l'ordre d'exécution des instructions. Toutes ces innovations ont été créées dans un seul et unique but, celui d'augmenter considérablement les performances des processeurs, but difficile à atteindre par l'augmentation seule de la fréquence. En d'autres mots, la fréquence représente la rapidité du processeur tandis que ces diverses innovations caractérisent l'efficacité de l'architecture.

Il existe donc un certain nombre d'optimisations matérielles dans les diverses architectures de microprocesseurs, cependant leur exploitation nécessite une connaissance avancée du fonctionnement de ces dernières. C'est pourquoi, une étude approfondie du processeur ciblé est fortement recommandée avant tout développement de programme lorsque le but recherché est avant tout la performance.

Dans cette optique, les différentes solutions existantes dans le domaine de l'embarqué seront présentées à travers deux grandes familles de processeurs, les architectures PowerPC [1][2][3] et les architectures dites x86 car compatibles avec le jeu d'instruction x86 [4][5]. Ceci permettra de mettre en avant les différents facteurs permettant de décrire une architecture processeur.

2.1.1. M` dèles d'architectures pr` cesseurs

Il existe depuis le commencement des architectures processeurs deux modèles d'exécution concurrents, qui sont les machines séquentielles dites de Von Neumann et les machines à flux de données . Après de nombreuses années d'évolution disjointe, ce n'est qu'à la fin des années quatre-vingt, lors de l'introduction des machines multi-flots ou de celles à flux de données à grosse granularité , que ces deux approches ne sont plus considérées comme indépendantes mais bel et bien comme étant les deux extrêmes de l'ensemble des solutions d'architectures. Ainsi un processeur donné intégrera systématiquement différentes notions voulues d'une ou de l'autre approche.

C'est pourquoi leur présentation est nécessaire afin de bien comprendre les différents raisonnements qui ont mené à de multiples évolutions qui rendent les architectures d'aujourd'hui aussi performantes.

2.1.1.1. M` dèle de V` n Neumann

Le modèle d'exécution de Von Neumann, aussi appelé modèle à flux de contrôle, considère un programme comme un ensemble d'instructions adressables qui modifie l'état des données contenues en mémoire. Chaque instruction effectue donc soit une opération, soit un transfert de contrôle vers une autre instruction, sachant que, par défaut, la prochaine instruction à exécuter est celle qui se trouve à l'adresse consécutive à celle de l'instruction actuelle. L'instruction courante est pointée par le compteur de programme. Ce système est entièrement déterminé par la valeur du pointeur de programme.

2.1.1.2. Le m` dèle à flux de d` nnées

Le modèle d'exécution à flux de données diffère du modèle de Von Neumann de par l'exécution du programme. Ce dernier n'est plus déterminé par l'exécution de l'instruction précédente mais uniquement par la disponibilité des opérandes. De ce fait, le compteur de programme et la mémoire globale n'existent plus dans ce modèle, éliminant deux goulots d'étranglement possibles. Le programme est compilé sous forme d'un graphe à flux de données où les nœuds sont les instructions et les arcs représentent les dépendances entre instructions. Ainsi, lors de l'exécution, les données se propagent le long des arcs sous forme de jetons et lorsqu'un nœud dispose de tous ses jetons, il est exécuté et produit à son tour un jeton qui transitera vers les prochaines dépendances.

En théorie, un tel modèle permet d'exploiter la totalité du parallélisme d'exécution disponible au sein d'un programme. Cependant, les architectures générées à partir de ce modèle ont d'abord donné lieu à des architectures très complexes et n'ont pas apporté les gains de performance promis.

2.1.2. Év` luti` ns imp` rtantes des architectures pr` cesseurs

2.1.2.1. Le *pipeline*

Le modèle d'exécution de Von Neumann a été utilisé lors de la conception des processeurs *CISC* et *RISC* de première génération. Chaque instruction est exécutée lorsque l'instruction précédente est terminée. Rapidement, les concepteurs ont vu qu'il était facile d'optimiser le flux d'instructions en divisant en plusieurs étages le traitement d'une instruction, ce qui améliore considérablement les performances du processeur. Ces étages sont, par exemple dans le cas d'une machine RISC, lecture de l'opération en mémoire, décodage de l'opération en micro-opérations (instructions interprétables par le processeur), lecture des opérandes, exécution et écriture du résultat.

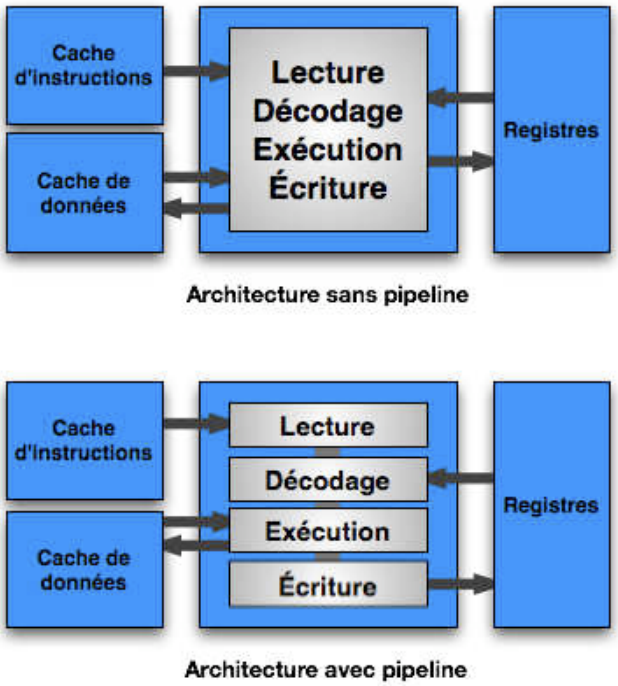

Figure 2.1.1: Exemple d'architecture avec et sans pipeline

La durée de traitement d'un étage étant plus courte que celle de l'instruction dans son ensemble, la durée du cycle d'horloge se trouve par conséquent réduite et permet ainsi aux architectures de monter plus haut en fréquence. Cette technique, nommée le *pipeline* [7], permet de produire le résultat d'une opération sur la durée de traitement d'un étage, celui de l'exécution, et non plus sur la durée de traitement de l'instruction. Ainsi en régime constant, la vitesse d'exécution est considérablement augmentée. De plus, le compteur de programme peut commencer à traiter l'instruction à exécuter avant même que ses opérandes soient disponibles. C'est ici qu'apparaît la première forme de parallélisme qui exploite donc le parallélisme entre les différents étages. La technique de codage associée s'appelle le software pipelining [6].

2.1.2.2. Les architectures CISC / RISC

Nous avons déjà mentionné deux familles de processeurs, les processeurs CISC et les processeurs RISC [1] . La famille des processeurs CISC, pour Complex Instruction Set Computer, dispose d'un jeu d'instructions dit complexe. A l'inverse, les architectures RISC, pour Reduced Instruction Set Computer, emploient un jeu d'instructions dit réduit.

Historiquement, la programmation se faisait en langage machine ou assembleur. Afin de simplifier la tâche des programmeurs, les concepteurs de micro-processeurs s'efforcèrent de mettre à disposition des instructions de plus en plus complexes permettant de coder de manière plus concise et plus

efficace. L'arrivée des langages de haut-niveau et des compilateurs n'a fait qu'amplifier la tendance avec notamment l'ajout d'instructions d'appel et de retour de fonction ou d'instructions capable de décrémenter un registre et d'effectuer un saut conditionnel si le résultat est non nul. Un des raisons clés de cette tendance était le manque de mémoire embarquée sur les architectures dû à son tarif prohibitif et à sa lenteur. Ce sont ces contraintes qui nécessitaient l'emploi de programmes compacts.

Le jeu d'instructions CISC était donc la réponse à cette problématique. Il se voulait ainsi le plus exhaustif possible aux yeux du programmeur. Typiquement les architectures x86 sont conçues autour de ce jeu d'instructions CISC.

Cette complexité des jeux d'instructions se répercute sur le décodage de ces dernières. D'une part l'intégration de cette étape dans les processeurs finit par représenter prés de la moitié des transistors utilisés dans la conception de la puce. C'est donc une surface non réductible qu'occupe l'unité de décodage, mais aussi une consommation minimale incontournable. D'autre part, le traitement des opérations simples, qui constituent la grande majorité d'un programme, se voit ralenti par cette phase de décodage complexe.

Partant d'une étude statistique montrant que seules 20% des instructions étaient utilisées par 80% des programmes, l'architecture RISC a pour enjeu de favoriser les opérations simples constituant la grande majorité des programmes. Seuls quelques modes d'adressage sont conservés. L'interface entre la mémoire et les unités de calcul se fait par l'intermédiaire de registres. Les opérations ne se font qu'entre registres. Cette simplification du jeu d'instructions et du fonctionnement du processeur donne naissance aux architectures RISC. Les architectures PowerPC, Power, SPARC et MIPS sont, elles, des architectures RISC.

Bien que sujets à une longue concurrence, ces deux types d'architecture ne sont aujourd'hui plus aussi disjoints. D'un coté, les architectures x86 actuelles ont un fonctionnement interne proche de la philosophie RISC. Seul la compatibilité avec le jeu d'instructions x86 est conservée. De l'autre, les architectures PowerPC voient leur jeu d'instructions étoffé avec l'ajout d'unités vectorielles par exemple.

2.1.2.3. Les architectures superscalaires

Contrairement à ce qui est décrit dans le modèle de Von Neumann, les instructions ne dépendent pas systématiquement des précédentes. Il se peut que plusieurs instructions consécutives traitent des données totalement indépendantes entre elles. Par conséquent, de telles instructions peuvent être calculées suivant un ordre différent de celui proposé par le programme voire même être calculées simultanément. Le modèle de Von Neumann peut donc induire de fausses dépendances entre les instructions.

Le modèle à flux de données, quant à lui, avait bien anticipé les bénéfices de ce parallélisme d'exécution. Cependant, en s'intéressant de trop près au parallélisme global du programme, ce modèle a généré des solutions beaucoup trop coûteuses.

Or si il est trop compliqué de traiter le parallélisme du programme entier, il est tout à fait réalisable de considérer le parallélisme local d'un programme.

Cette nouvelle forme de parallélisme au niveau de l'exécution des instructions s'appelle l'*ILP* pour *Instruction Level Parallelism* [7].

La performance d'un processeur peut être évaluée par la quantité d'instructions qu'il exécute à chaque seconde, c'est-à-dire le rapport *i/s*. Ce rapport peut aussi s'écrire:

$$i/s = i/c * c/s$$

où *c* représente le nombre de cycles d'horloge du processeur. *i/c* est le nombre moyen d'instructions exécutées par cycle processeur c'est à dire l'*IPC* (Instructions Per Cycle), soit l'indice révélateur du nombre d'instructions exécutées par cycle. Quant à *c/s*, le nombre de cycle par seconde est par définition la fréquence du processeur. La formule devient ainsi:

$$i/s = IPC * F$$

Cette formule très simple montre ainsi que l'IPC et la fréquence sont deux facteurs majeurs de performance des processeurs.

C'est par ce constat que sont nées les architectures *superscalaires* qui ont exploité cette propriété en enfreignant le modèle de Von Neumann. Les instructions ne sont alors plus exécutées séquentiellement mais lorsque leurs opérandes sont disponibles. Une nouvelle ressource qu'est la fenêtre d'instructions permet ainsi de sélectionner quelles instructions parmi les N instructions consécutives contenues dans la fenêtre pourront être exécutées à chaque cycle d'horloge. Ces architectures disposent alors de plusieurs unités d'exécution pouvant fonctionner en parallèle, permettant ainsi d'exécuter plus d'une instruction par cycle. Une autre ressource de ré-ordonnancement a pour tâche de s'assurer de la cohérence du programme en veillant à ce que l'écriture des résultats en mémoire suive bien l'ordre imposé par le modèle de Von Neumann.

Ces architectures sont donc capables d'exécuter plusieurs instructions simultanément. Le nombre maximum d'instructions simultanées s'appelle la *largeur du processeur*.

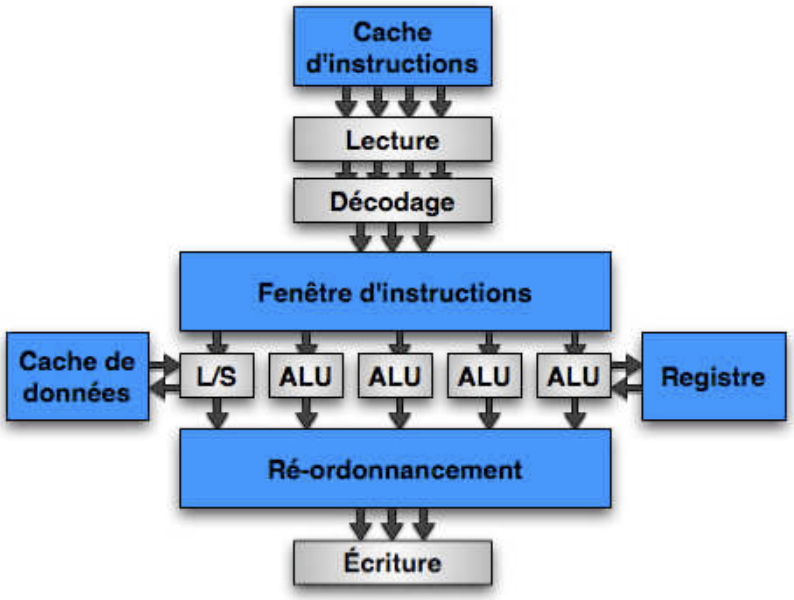

Figure 2.1.2: Exemple d'architecture superscalaire

Un autre type d'architecture plus exotique hérite aussi du modèle de Von Neumann ainsi que de cette approche superscalaire. Ce sont les architectures dites *VLIW* pour Very Long Instruction Word [8] où le parallélisme d'instructions n'est pas effectué dynamiquement par la fenêtre d'instructions mais par le compilateur. Le compilateur extrait du programme les instructions susceptibles d'être exécutées simultanément et regroupe ces dernières en longs mots [9]. Ces longs mots, ou instructions complexes correspondant à la juxtaposition de plusieurs instructions simples, sont alors exécutés selon le modèle de Von Neumann et d'un *pipeline* à plusieurs étages. Un exemple de processeur VLIW est le processeur Itanium d'Intel [10].

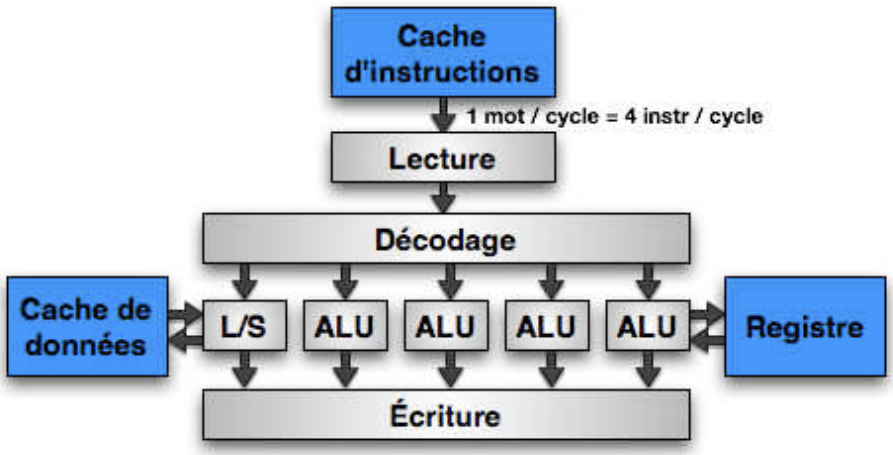

Figure 2.1.3: Exemple d'architecture VLIW

Aujourd'hui, les architectures superscalaires modernes peuvent posséder jusqu'à 12 unités d'exécution indépendantes et offrent de véritables gains en performance par rapport aux architectures précédentes. Cependant d'autres paramètres, comme le nombre d'instructions chargées ou décodées par cycle d'horloge, entraînent de nouveaux goulots d'étranglement dans l'architecture d'un processeur donné. De plus, il est montré que le parallélisme local d'instructions au sein d'un programme s'élève en moyenne autour de 5 instructions par cycle processeur.

2.1.2.4. Les architectures parallèles

Une architecture parallèle peut être définie comme un système de plusieurs éléments coopérant à l'exécution d'une ou plusieurs tâches. Ces architectures sont donc en contradiction avec le modèle de Von Neumann d'une architecture séquentielle où seul un processeur exécute la tâche. De part le nombre d'architectures et de solutions parallèles, une certaine forme de classification est donc requise.

La taxonomie de Flynn est une classification des architectures d'ordinateur proposées par Michael J. Flynn en 1966 [11]. La conception d'une architecture étant caractérisée par son flot d'instructions et son flot de données, la taxonomie de Flynn classifie les architectures par leur multiplicité de flux d'instructions ou de données, ce qui donne quatre classes possibles d'architectures:

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

Tableau 2.1.1: Taxonomie de Flynn

L'architecture SISD correspond à un ordinateur séquentiel qui n'exploite aucun parallélisme. Il exécute tour à tour une instruction et ne modifie qu'une donnée à la fois, typiquement le modèle de Von Neumann.

L'architecture MISD, quant à elle, considère le parallélisme d'exécution au niveau du flot d'instructions, ce que l'on peut aisément représenter par une architecture pipelinée [12].

L'architecture SIMD est la classe de processeurs la plus répandue et représente toutes les architectures exploitant le parallélisme de données dans l'exécution du programme. Elles sont aussi appelées architectures vectorielles et permettent d'exécuter la même instruction sur plusieurs données.

Une architecture SIMD consiste donc en un ensemble d'unités de calcul PE (Processing Element), coordonnées par une unique unité de contrôle

assurant un flux commun d'instructions à chacun des PE. Ces derniers opéreront chacun sur les données qui leur seront attribuées, créant un parallélisme d'exécution au niveau données. Ci-dessous, un exemple simple de calcul vectoriel:

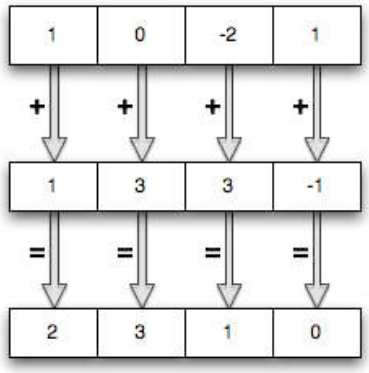

Figure 2.1.4: Exemple d'addition vectorielle

Le bénéfice d'une telle architecture n'est plus à démontrer, notamment dans les applications multimédia où les calculs sur les données sont massivement parallèles et sans grande dépendance entre les différentes données. Elles sont aussi bien adaptées pour des implémentations d'algorithmes de traitement de signal où les formules mathématiques sont souvent exprimées sous forme de vecteurs et de matrices. Ces architectures sont implantées sous forme d'unités vectorielles dans la plupart des processeurs généraux existants, rendant cette classe d'architectures quasiment incontournable.

L'architecture MIMD est une architecture plus complexe où contrairement à l'architecture SIMD, chaque PE dispose de son propre flot d'instructions venant de sa propre unité de contrôle. Les PE se synchronisent et s'échangent les données entre eux par le biais d'un réseau d'interconnexions. Il existe plusieurs types d'interconnexions possibles permettant de différencier plusieurs familles d'architecture MIMD. Parmi elles, se distinguent deux familles d'architectures parallèles qui sont:

- les architectures à mémoire partagée. Les processeurs ont accès à la mémoire comme à un espace d'adressage global. Toute modification dans la mémoire est vue par les autres processeurs. La communication entre processeurs se fait donc au travers de la mémoire globale.
- les architectures à mémoire distribuée. Chaque *CPU* dispose de sa propre mémoire et de son système d'exploitation. Un middleware est alors nécessaire à la communication et à la synchronisation des éléments processeur.

2.1.2.5. Les architectures SAXPY

Il ne s'agit pas vraiment d'une architecture mais plutôt d'une instruction de calcul disponible dans le jeu d'instructions de certaines architectures. *SAXPY* représente alors la capacité de fusionner sous forme d'une seule et unique instruction deux opérations flottantes que sont la multiplication et

l'addition. S pour simple précision et AXPY pour A X PLUS Y représentant l'équation mathématique $a.x+y$, c'est-à-dire la multiplication de la variable A par la variable X, le tout additionné à la variable Y. La version double-précision s'appelle par analogie *DAXPY*.

Cette multiplication addition fusionnée appelée aussi par contraction muladd (multiply and add) ou mulacc (multiply and accumulate), offre un gain double. D'une part, elle permet d'exécuter deux opérations par l'intermédiaire d'une seule instruction exécutée en un seul et unique cycle d'horloge. D'autre part, l'implémentation de la multiplication addition fusionnée accroît la précision du résultat.

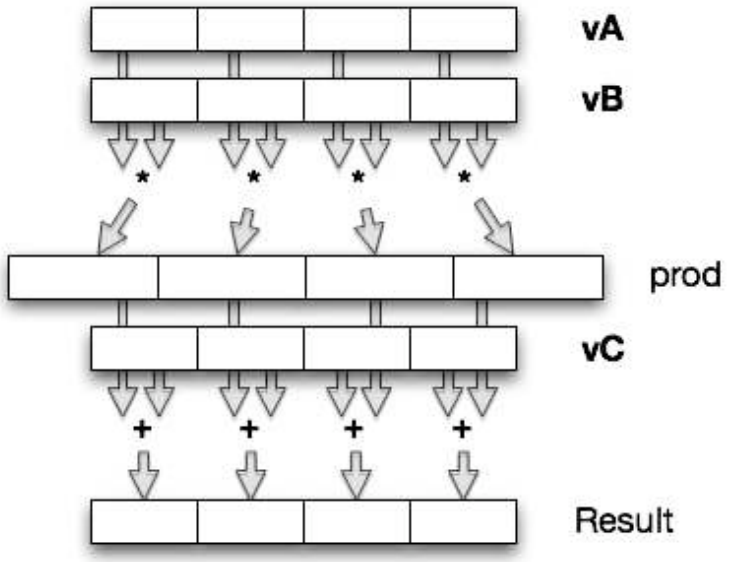

Figure 2.1.5: Exemple de multiplication addition fusionnée vectorielle

L'importance de cette instruction est telle qu'elle représente un atout lorsque cette dernière est présente dans le jeu d'instructions de l'architecture en question. À l'inverse, son absence peut être considérée comme un handicap à l'atteinte des performances optimales des algorithmes de calcul []. C'est donc une instruction indispensable qui sera considérée comme présente dans les jeux d'instructions standards des processeurs au risque de devoir la simuler dans les architectures qui en sont dépourvues.

2.1.2.6. Les architectures Simultane` us Multi Threading

Simultaneous Multi Threading (SMT) est une technique datant des années 1950. Comme le Symmetric Multi Processing (SMP), elle consiste à accélérer l'exécution parallèle des différents threads. Un thread est un processus ou un fil d'exécution. Il représente l'exécution d'un ensemble d'instructions.

Les systèmes d'exploitation actuels ne sont plus mono tâche et permettent l'exécution simultanée de plusieurs processus. Ceci est simulé par le

système d'exploitation par commutation de contexte sur un processeur mono cœur. La commutation de contexte (*context switching*) consiste à sauvegarde l'état d'un processus afin de partager les ressources du processeur entre les différents processus en cours d'exécution.

Les techniques SMT et SMP permettent de multiplier respectivement les cœurs logiques et physiques. L'exécution simultanée des processus n'est alors plus simulée.

Alors que le SMP est une duplication des ressources de calcul, le SMT est une technique implantée dans certains processeurs permettant le partage des mêmes ressources à deux processus en cours d'exécution. Le SMT permet d'exploiter au maximum l'ILP du processeur [7][14].

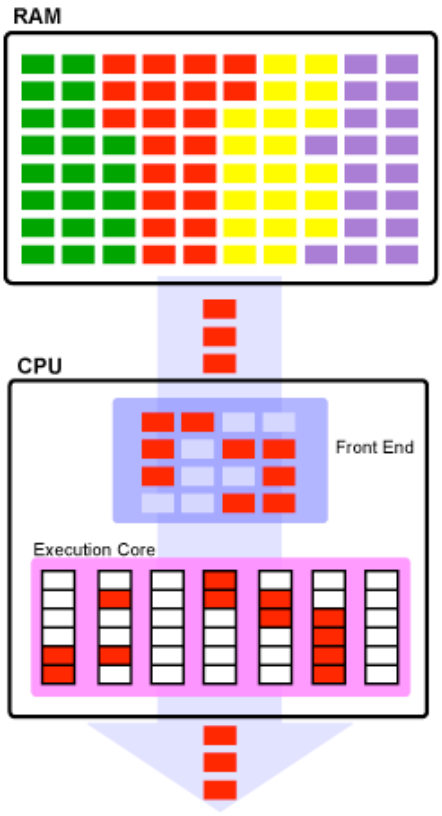

Figure 2.1.6: Exécution Mono Thread - Mono Cœur (Ars Technica)

La figure 2.1.6 montre l'intérêt des techniques SMT et SMP. Lors de l'exécution d'un processus pendant la mise en attente des autres fil d'exécution, les dépendances entre opérations et les latences de calcul font le CPU n'est pas exploité à 100%. À chaque cycle d'horloge, ce ne sont pas la totalité des unités de calcul qui sont mises au service de l'unique processus en cours.

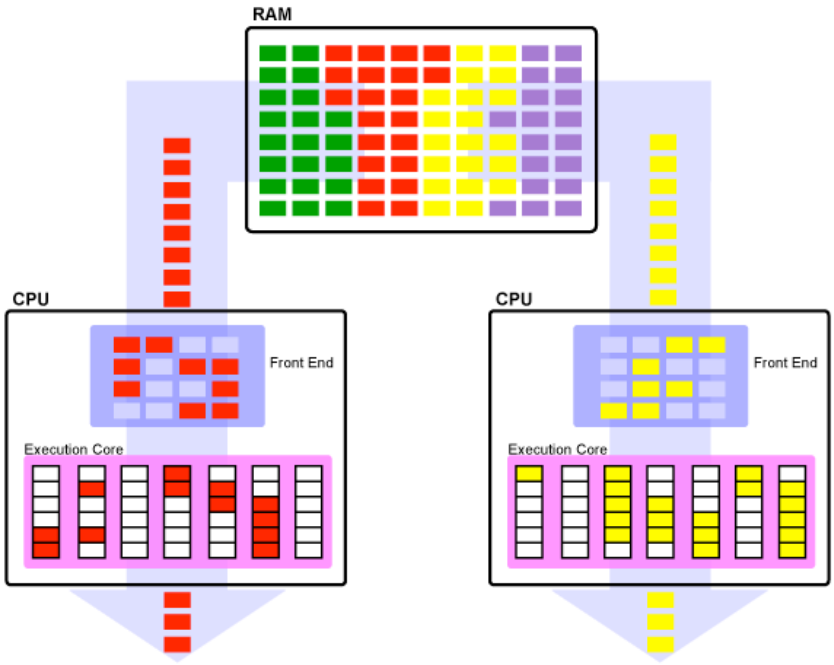

Figure 2.1.7: Exécution Mono Thread - Multi Cœurs (Ars Technica)

La figure 2.1.7 montre le gain apporté par la technique SMP. Les deux cœurs physiques se partagent la mémoire et permettent l'exécution simultanée de deux processus. Cette technique ne résout pas le taux d'occupation des cœurs de calcul.

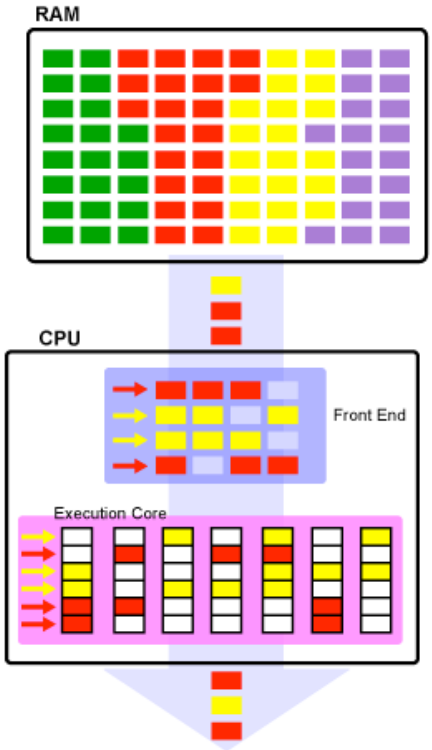

Figure 2.1.8: Exécution Multi Thread - Mono Cœur (Ars Technica)

La figure 2.1.8 illustre la technique SMT permettant d'augmenter cette fois-ci le taux d'occupation du CPU. Les unités de calcul sont partagées entre les deux processus. Cette technique permet d'alterner le traitement de deux processus sur les différentes unités de calcul constituant le processeur. Ceci permet de combler les bulles créées dans le pipeline par les dépendances et les latences de calcul entre opérations. Cette technique n'accélère pas le traitement d'un thread mais celui des deux threads dans leur ensemble. L'exploitation des ressources de calcul gagnent en efficacité.

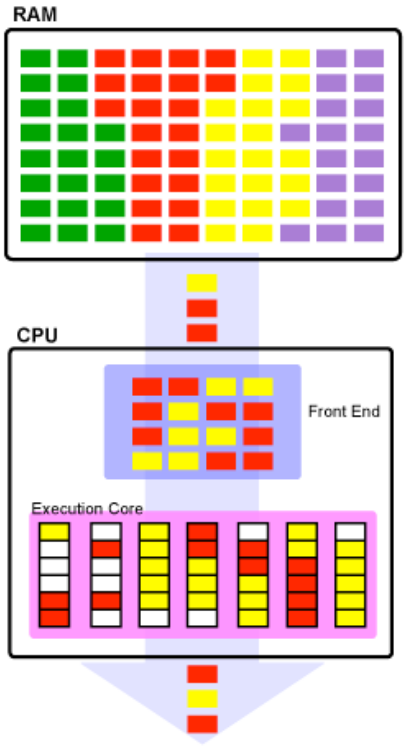

Figure 2.1.9: HyperThreading d'Intel (Ars Technica)

L'hyperthreading [13] est une technique implantée par Intel dans ses architectures, poussant plus loin le partage des ressources entre les deux processus en cours d'exécution. En effet, au cours du même cycle d'horloge deux instructions qui n'appartiennent pas au même processus peuvent être traiter simultanément.

2.1.3. L'architecture P` werPC

Historiquement, l'architecture PowerPC [1][2] est une référence dans le domaine de l'embarqué. Ce sont des architectures RISC dont les performances par watt ont toujours été les meilleures du marché. Aujourd'hui avec l'implication d'Intel dans les solutions basse consommation destinées aux ordinateurs portables, le PowerPC voit apparaître un sérieux concurrent.

2.1.3.1. P` werPC 750 (G3)

Apparue en 1997, cette architecture de Motorola et de IBM [2][16], âgée maintenant de plus de 10 ans, reste toujours d'actualité notamment pour des solutions basse consommation et faible dissipation exigées par les intégrateurs de l'aérospatial.

Ce processeur *superscalaire* est une évolution du PPC603/603e [2][15], lui même pensé pour une utilisation basse consommation avec le souhait d'Apple de l'intégrer dans leurs solutions mobiles, PowerBook, de première génération.

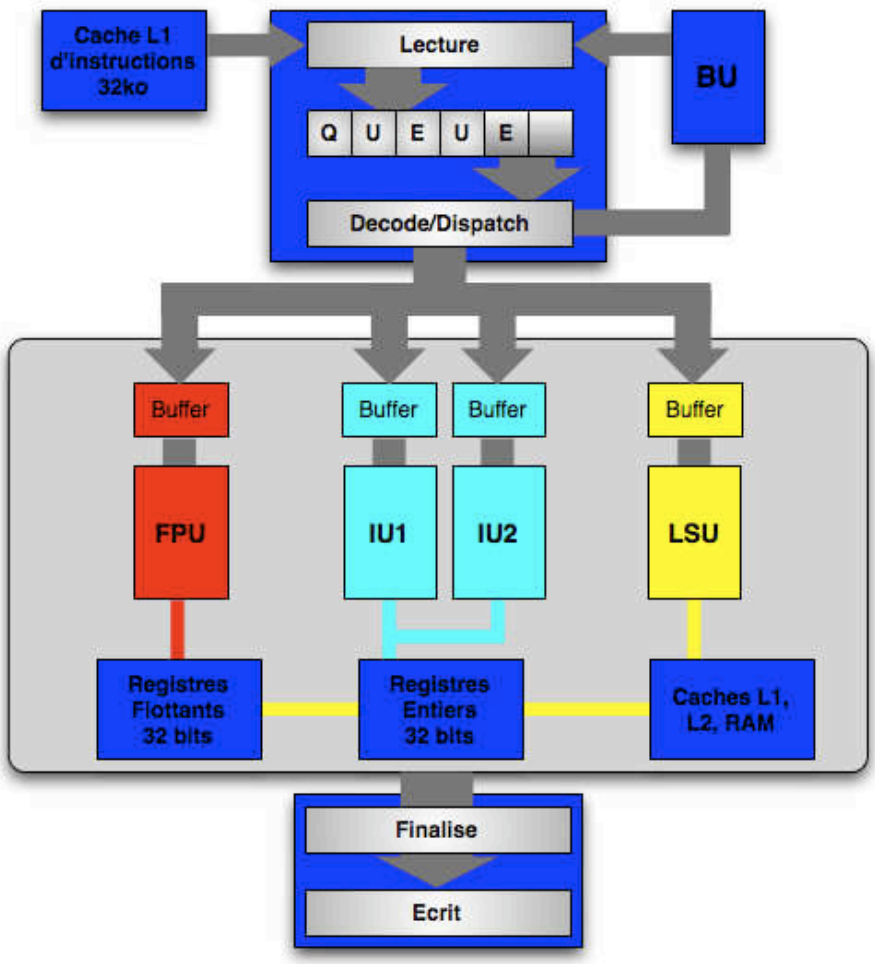

Figure 2.1.10: Architecture du G3

Le G3 dispose donc de deux niveaux de cache. Le cache L1 se divise en 2 caches indépendants de 32ko chacun, le premier dédié à l'acheminement des données et le second à celui des instructions. Le cache L2 est, quant à lui, unifié et accueille jusqu'à 512ko d'information.

Il a été conçu autour du *pipeline* classique en 4 étapes comme son prédécesseur:

1. Lecture de l'instruction
2. Décodage et distribution
3. Exécution
4. Écriture du résultat

Il peut lire jusqu'à 4 instructions par cycle qui iront vers une file d'attente contenant jusqu'à 6 instructions. De là, jusqu'à **2** instructions peuvent être dispatchées simultanément vers les différentes unités du processeur.

Concernant les unités d'exécution du processeur, le G3 dispose de **5** unités:

- une unité flottante (*Floating Point Unit - FPU*) pouvant exécuter toutes les instructions flottantes simple précision avec une latence de **3** cycles au rythme d'une instruction par cycle. La multiplication double précision et la multiplication addition fusionnée double précision demande un cycle supplémentaire d'exécution.
- deux unités entières dont l'une (*Complex Integer Unit - CIU*) peut exécuter toutes les instructions entières tandis que l'autre (*Simple Integer Unit - SIU*) ne traite pas la multiplication et la division. La majorité des instructions s'exécute en **1** cycle.
- une unité d'interface mémoire (*Load Store Unit - LSU*) gérant tous les calculs d'adressage en lecture et écriture.
- une unité de branchement (*Branch Unit - BU*).

Le G3 est donc une excellente architecture qui s'impose très bien sur le marché des processeurs basse consommation. Seul point manquant par rapport à la concurrence: l'absence d'un calculateur vectoriel.

2.1.3.2. P` werPC 7448 (G4e)

Le G4e est une évolution du G4 [2][17], lui-même pensé autour du G3 en y ajoutant les capacités vectorielles manquantes avec l'unité AltiVec [18][19]. Pour commencer, le G4e abandonne le *pipeline* classique en 4 étapes pour un *pipeline* en 7 étapes, lui permettant de monter plus haut en fréquence.

	G4		G4e	
Front End	1	Fetch	1	Fetch1
			2	Fetch2
	2	Decode / Dispatch	3	Decode / Dispatch
			4	Issue
Back End	3	Execute	5	Execute
	4	Complete / Write Back	6	Complete
			7	Write Back

Tableau 2.1.2: Détail du pipeline des deux architectures G4 et G4e

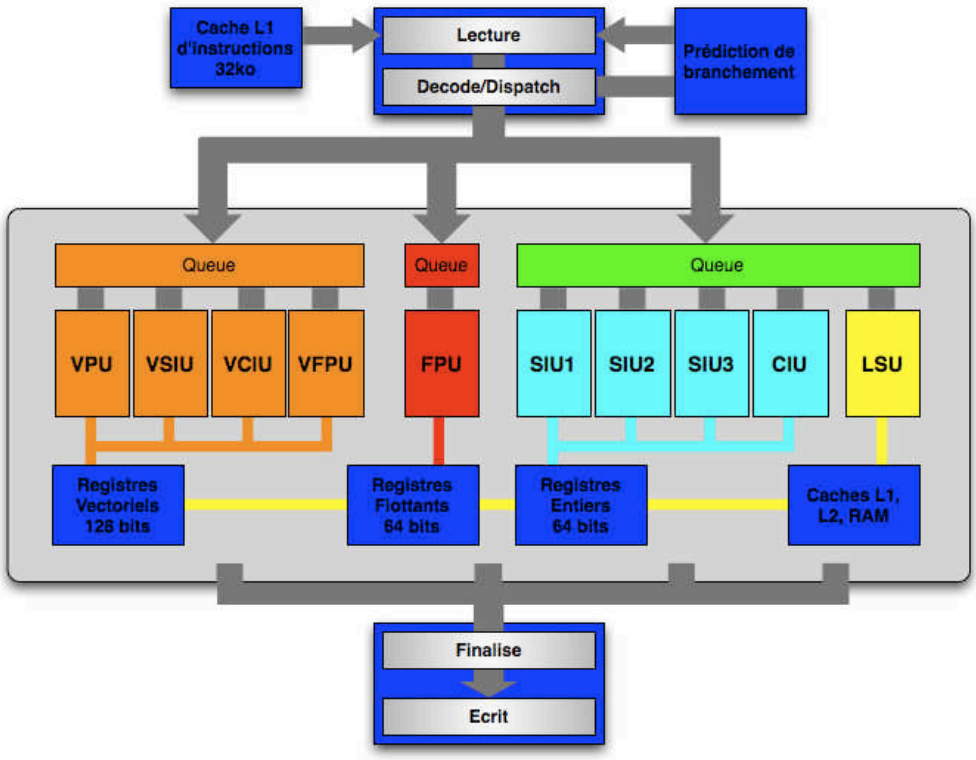

Figure 2.1.11: Architecture du G4e

Durant les étapes 1 et 2 de lecture des instructions, le G4e peut lire jusqu'à 4 instructions par cycle. Après décodage, au plus **3** instructions peuvent être envoyées vers les unités adéquates.

Une des améliorations fondamentales concerne le remplacement des buffers de réservation par des files d'attente d'instructions (issue queue) permettant de réduire les risques de *stall* lors du dispatch dû à l'occupation de l'unité concernée. Ainsi le GIQ (General Issue Queue) peut recevoir jusqu'à 3 instructions de l'unité de dispatch et en restituer jusqu'à 3 également aux 4

unités entières du G4e et à l'unité de load/store, toutes rattachées à cette file d'attente. Le VIQ (Vector Issue Queue) peut recevoir jusqu'à 2 instructions par cycle et les transmettre à 2 des 4 unités SIMD. Le FPIQ (Floating Point Issue Queue), quant à lui, ne peut recevoir qu'une seule instruction par cycle possible qu'elle enverra à l'unique unité flottante du G4e.

Au niveau des unités d'exécution du G4e, apparaissent 2 unités entières supplémentaires ne calculant que des opérations simples c'est-à-dire toutes les instructions entières hormis la multiplication et la division. Ensuite, le nouveau processeur se voit doté de l'AltiVec, unité vectorielle de Motorola avec au total 4 unités indépendantes dont une unité de permutation, une unité de calcul flottant et deux unités de calcul entier (une simple et une complexe).

2.1.3.3. P` werPC 970 (G5)

Le G5 [20][21] figure parmi les derniers PowerPC utilisés par Apple dans sa gamme de produits haute performance, il reste à ce jour le PowerPC le plus performant du marché. Développé par IBM, il offre une architecture 64 bits pouvant atteindre des fréquences jusqu'à 2,7 GHz et des performances flottantes crêtes atteignant les 16 GFlops. Contrairement au choix des concurrents dans cette course à la fréquence, ce processeur a été conçu de manière à être le plus efficace possible. Plutôt que de chercher à calculer une opération le plus vite possible, la philosophie du G5 est de calculer le plus d'opérations possible par cycle d'horloge.

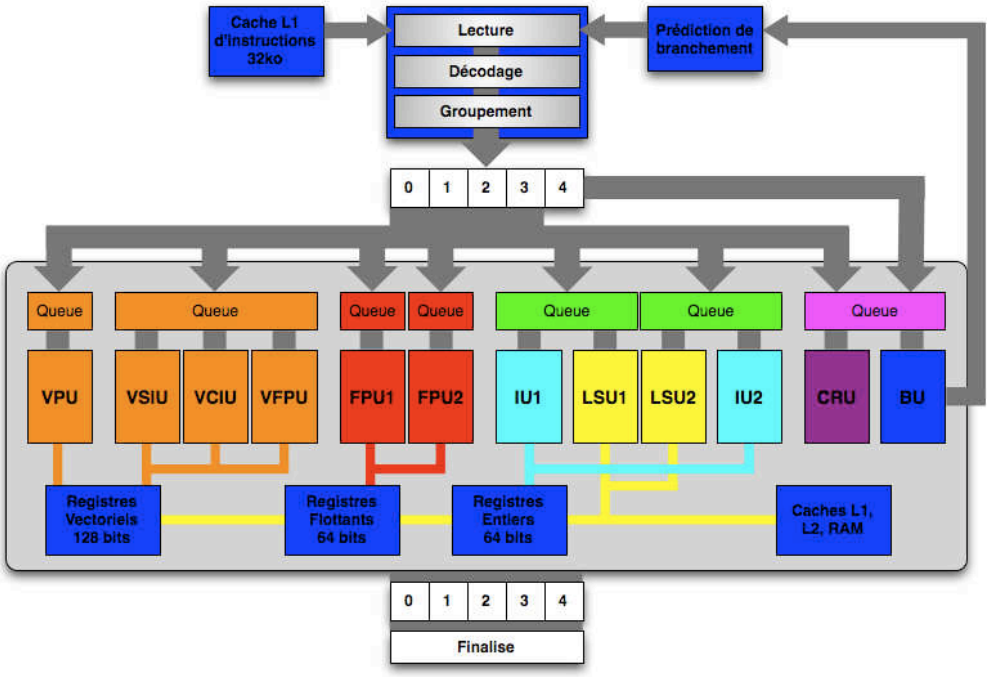

Figure 2.1.12: Architecture du G5

Le PowerPC 970 est donc une architecture *superscalaire* disposant d'un total de 12 unités d'exécution. Le processeur peut lire jusqu'à 8 instructions simultanément à partir du cache instructions L1. Ces 8 instructions sont ensuite prise en charge par l'unité de décodage.

La nouveauté du G5 se situe au niveau du dispatch. Il est capable de dispatcher au plus 5 instructions par cycle mais ceci au sein d'un groupe de 5 instructions préalablement formé par l'unité de groupement. Le contenu de ce groupe est donc soumis à certaines règles d'éligibilité d'une instruction pour un slot donné. Ainsi, par exemple, le cinquième slot est uniquement réservé à l'unité de branchement. C'est donc ce groupe de **5** instructions que le G5 distribue dans l'ordre aux six files d'attente qui alimentent les unités d'exécution du processeur. Au final jusqu'à 8 instructions par cycle arrivent, dans le désordre, des unités d'exécution vers l'étage de complétion qui s'occupe de reformer les groupes d'instructions dans l'ordre natif du programme.

Concernant le cœur d'exécution du G5, ce dernier dispose de:

- 4 unités vectorielles au fonctionnement identique à celles du G4e, à l'exception de la contrainte imposant que seule l'unité de permutation puisse être alimentée simultanément avec l'une des 3 autres unités vectorielles.
- 2 unités flottantes indépendantes simple et double précision.
- 2 unités entières indépendantes
- 2 unités de lecture / écriture mémoire
- 1 unité conditionnelle
- 1 unité de branchement

2.1.3.4. Freescale 8544 (e500)

Freescale se spécialise aujourd'hui sur le marché des System-on-Chip (SOC). Ce sont des solutions intégrant sur une puce un processeur simple ou multicœurs associé à son bridge proposant divers I/Os tel que l'ethernet, le PCI-Express ou encore l'USB. Freescale propose ainsi une gamme de SOCs sur catalogue. Il peut aussi proposer des SOCs personnalisés selon les besoins du client.

Le 8544 de Freescale est un SOC équipé d'un cœur e500 et d'un bridge interne proposant du gigabit ethernet, du PCI-Express et du DMA. Le cœur e500 [22] de Freescale est un processeur PowerPC destiné aux solutions à basse consommation. C'est un processeur 32 bits qui peut exécuter jusqu'à 2 instructions simultanément au travers d'un *pipeline* à 7 étages. Ce nouveau cœur est proposé à des fréquences allant de 533 MHz à 1,5GHz, ce qui en fait un excellent successeur au PowerPC G3.

2.1.3.5. Freescale 8641 (e600)

Avec sa famille des MPC86xx, Freescale vise le marché des "System-on-Chip" (SoC) haute performance. Ce sont des solutions principalement multicœurs intégrant le cœur e600 de Freescale et la gestion de différentes

interfaces telles que deux contrôleurs mémoire 64-bits DDR2, du PCI-Express, du Serial Rapid-IO et quatre liens gigabit ethernet. Le MPC8641D propose ainsi un double cœur pouvant atteindre 1,5GHz avec son bridge associé.

Le cœur e600 [23] de Freescale est une continuation du PowerPC G4. Ce processeur est capable de traiter jusqu'à 3 instructions simultanément à travers un *pipeline* de 7 étages. Tout comme le G4, il dispose de la technologie AltiVec et se montre particulièrement efficace face aux prédictions de branchement.

2.1.3.6. P.A. Semi PA6T

PA Semi est une start-up fondée en 2003. Leur principale contribution fut de proposer une version basse consommation et dual cœur du PowerPC 970, la consommation initiale de ce dernier étant le facteur contraignant à son intégration dans les systèmes embarqués. Les premiers masques réceptionnés furent très prometteurs. Ils présentaient des performances crêtes très proches du PowerPC 970 pour des fréquences atteignant les 2GHz et ceci pour une consommation tout à fait adaptée à une utilisation embarquée du processeur. Ils présentaient aussi la particularité d'intégrer le bridge gérant les différentes connexions tels que l'USB, l'Ethernet et l'IDE. Cependant le rachat soudain de la société par Apple, en début d'année 2008, a mis un terme au développement du micro-processeur.

2.1.3.7. IBM CELL Br` adband Engine

Dans cette course à la performance, IBM a choisi une toute autre solution en proposant une nouvelle architecture dénommée Cell Broadband Engine [24].

Cette architecture a la particularité de disposer, dans une seule puce, d'un processeur principal assisté d'un total de 8 co-processeurs pour le calcul intensif. Cette vision originale du multi-cœurs par IBM est soutenue par Sony et Toshiba qui ont collaboré à son développement pour la console de salon PS3. IBM espère que ce tremplin lui permettra d'obtenir des parts de marché dans le monde du multimédia. De plus, il dispose déjà d'une roadmap détaillée de ce que deviendra le Cell dans le futur dans les différentes solutions embarquées du processeur.

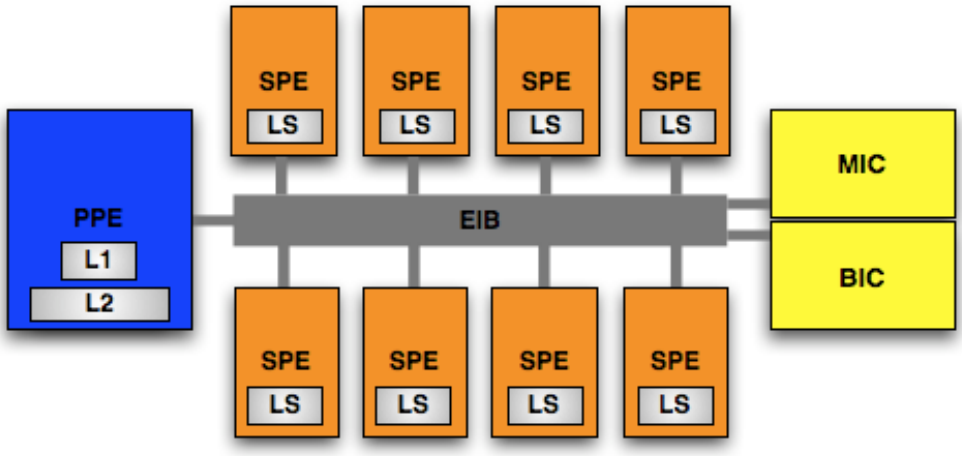

Figure 2.1.13: Vision d'ensemble du CELL BE

Ainsi le processeur dispose d'un cœur principal connecté à 8 cœurs DSP par l'intermédiaire d'un bus circulaire. Ce bus est capable de transférer jusqu'à 96 octets par cycle et est constitué de 4 anneaux indépendants. Deux des quatre anneaux font circuler les données dans le sens des aiguilles d'une montre à l'inverse des deux autres anneaux qui transitent les données dans le sens contraire.

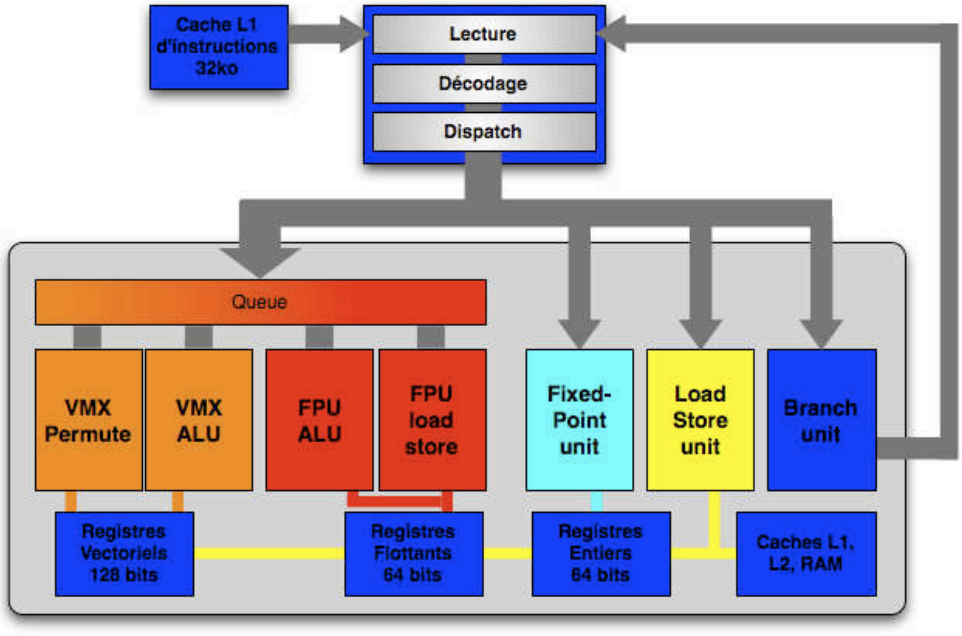

Figure 2.1.14: Détail du PPE

Le processeur principal dénommé PPE [24] par IBM pour Power Processing Element, est un processeur 64-bits dual thread totalement compatible avec le jeu d'instructions PowerPC.

Le cœur est nettement plus traditionnel que les architectures concurrentes. Il exécute les instructions dans l'ordre supprimant tout ré-ordonnancement dynamique des instructions et offrant une forme très limitée d'exécution superscalaire. L'architecture lit 4 instructions par cycle mais ne permet de distribuer que deux instructions par cycle aux unités d'exécution. Ces dernières se retrouvent aussi réduites à une unité flottante, une unité arithmétique et logique et une unité vectorielle. IBM ne reprend donc pas le parallélisme superscalaire qui a fait la force du PowerPC 970. En revanche, il privilégie une architecture impliquant une plus faible consommation et permettant de monter plus haut en fréquence. Le coût de réalisation est lui aussi plus faible.

IBM a doté le processeur d'un mécanisme de multi-threads permettant l'exécution d'un deuxième fil d'exécution du programme en parallèle de façon à exploiter les cycles d'horloge inutilisés par le premier fil d'exécution.

Première constatation, le compilateur jouera un rôle majeur dans les performances finales de l'application car en l'absence de re-ordonnancement dynamique dans l'architecture du processeur, il devra effectuer un re-ordonnancement statique qui réduira au maximum les dépendances entre opérations.

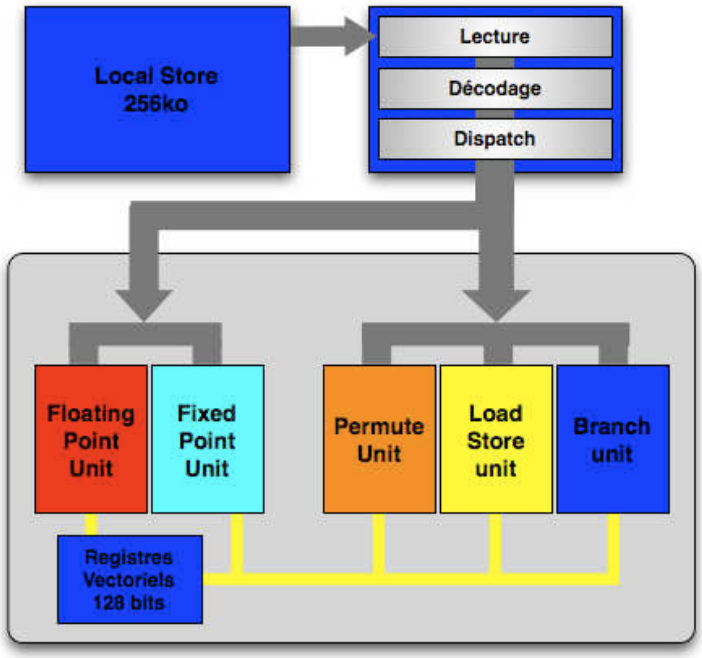

Figure 2.1.15: Détail d'un SPE

Le SPE, Synergistic Processing Element [24], est la principale nouveauté du CELL, offrant au PPE des co-processeurs performants permettant de faire du calcul intensif.

Premièrement les SPE sont des processeurs uniquement SIMD. Ils ne peuvent exécuter que des opérations vectorielles. Il existe des techniques permettant de simuler du calcul scalaire mais elles n'utilisent pas efficacement les performances propres du processeur.

Les SPE disposent de 2 unités:

- un moteur *DMA (Direct Memory Access)* [25]: le Memory Flow Controler (MFC)
- une unité vectorielle: Synergistic Processing Unit (SPU), d'une mémoire de 256 ko de SRAM appelée Local Store et d'un total de 128 registres 128-bits.

Les SPU sont des unités vectorielles permettant de faire du calcul SIMD. Elles disposent globalement des mêmes caractéristiques que leur confrère l'AltiVec mais utilisent leur propre jeu d'instructions, ce qui rend incompatible tout programme écrit en AltiVec. Un portage est donc nécessaire pour tout développement sur SPE.

Comme le PPE, le SPE peut dispatcher et exécuter jusqu'à deux instructions par cycle. Il dispose donc de deux *pipelines* indépendants dont l'un est dit pair et l'autre impair afin de les différencier. Le premier est responsable des opérations de lecture et écriture mémoire, de branchement et de permutation vectorielle. Le second concerne tous les calculs flottants, arithmétiques et logiques.

Le SPE est optimisé pour le calcul flottant simple-précision offrant une performance crête de 25,6GFlops à 3,2GHz au rythme d'une multiplication addition vectorielle par cycle d'horloge. Le Cell est d'ailleurs conçu pour atteindre des fréquences élevées jusqu'à 4GHz.

Le SPE est aussi capable de respecter la norme *IEEE* 754 en double précision, en proposant la prise en charge du calcul flottant double précision. Cependant ce dernier n'étant pas entièrement pipeliné, seule une opération est exécutée tous les 7 cycles.

En résumé, une métaphore d'IBM résume bien la philosophie du CELL en disant que les SPE constituent l'orchestre et le PPE le chef d'orchestre.

2.1.4. L'architecture Intel

Après s'être longtemps consacrée à la montée en fréquence de son processeur Pentium IV, l'équipe Intel a conçu une nouvelle architecture dans le but de répondre aux besoins des plate-formes mobiles. Ce nouveau processeur, le Pentium M s'est avéré être une belle réussite, présentant d'excellentes performances par watt et étant aujourd'hui la base du nouveau porte drapeau d'Intel, l'architecture Core.

2.1.4.1. Pentium M

En 2003 apparaît le Pentium M [26][27], processeur conçu spécialement pour le marché des plateformes mobiles d'Intel.

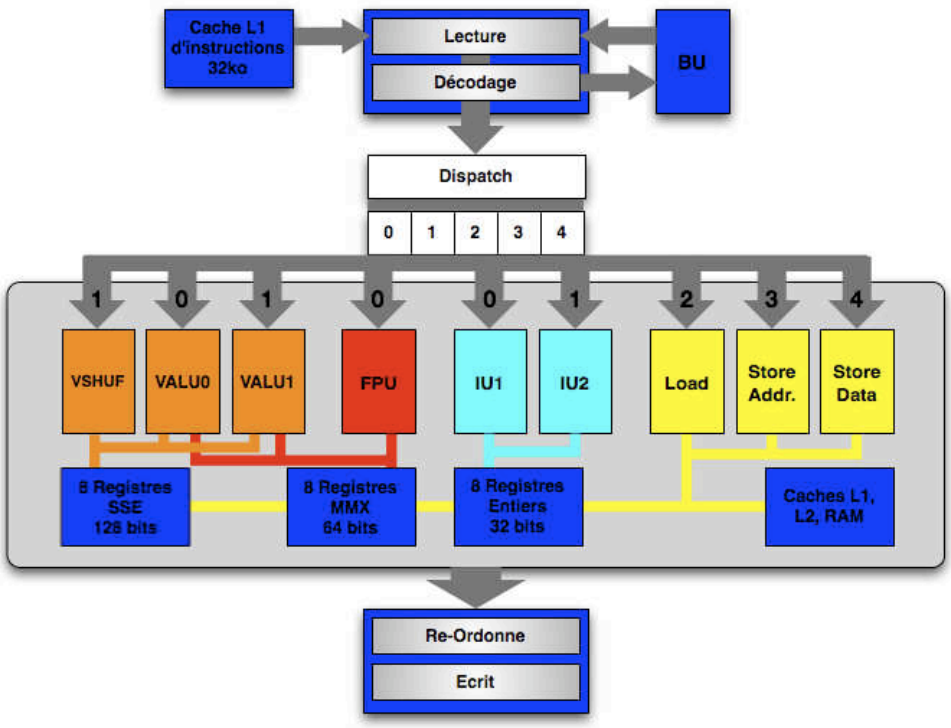

Figure 2.1.16: Architecture du Pentium M

Durant l'étape de lecture, le Pentium M charge une ligne d'instructions de 32 octets du cache L1 d'instructions dans un buffer de streaming qui transmettra les instructions vers le décodeur par ligne de 16 octets. L'unité de décodage est composée de 3 décodeurs indépendants dont deux ne gérant que les instructions se traduisant par une seule et unique micro-instruction tandis que le troisième décodeur peut traiter les cas de une à quatre micro-opérations.

C'est donc un total de 3 micro-instructions par cycle qui peut être envoyé vers l'unité de réservation qui s'occupe d'attribuer les instructions à leur port correspondant. Le Pentium M dispose de 5 ports au total, qui alimentent la totalité des unités d'exécution du processeur.

Parmi les unités du Pentium M, se trouvent:

- 2 unités entières
- 1 unité flottante
- 3 unités vectorielles, une spécialisée dans les permutations et les deux autres traitant tous les calculs *MMX, SSE* ou *SSE2*.
- 1 unité de lecture mémoire

- 2 unités d'écriture mémoire devant fonctionner simultanément lors d'une écriture en mémoire RAM. La première calculera l'adresse tandis que l'autre gérera l'écriture mémoire.

Les instructions résultantes sont ensuite renvoyées vers l'unité de réordonnancement qui rétablira l'ordre initial des instructions.

2.1.4.2. C` re Du`

Le Pentium M montrant d'excellentes performances, parfois supérieures à celles du Pentium IV, Intel lance le premier véritable processeur dual cœur avec un cache L2 partagé. Ce processeur est décliné en plusieurs versions allant de la plate-forme mobile aux gros calculateurs, preuve de l'efficacité de cette architecture. Cette nouvelle famille est nommée Core Duo [28] par Intel et se compose de processeurs simple et double cœur dont le fonctionnement est identique au Pentium M.

2.1.4.3. C` re 2 Du`

Le Core 2 Duo [28] est le nom commercial de la nouvelle architecture d'Intel faisant suite au Core Duo, l'architecture Core. Cette dernière est toujours dérivée de l'architecture du Pentium M dans le but d'accroître les performances par watt.

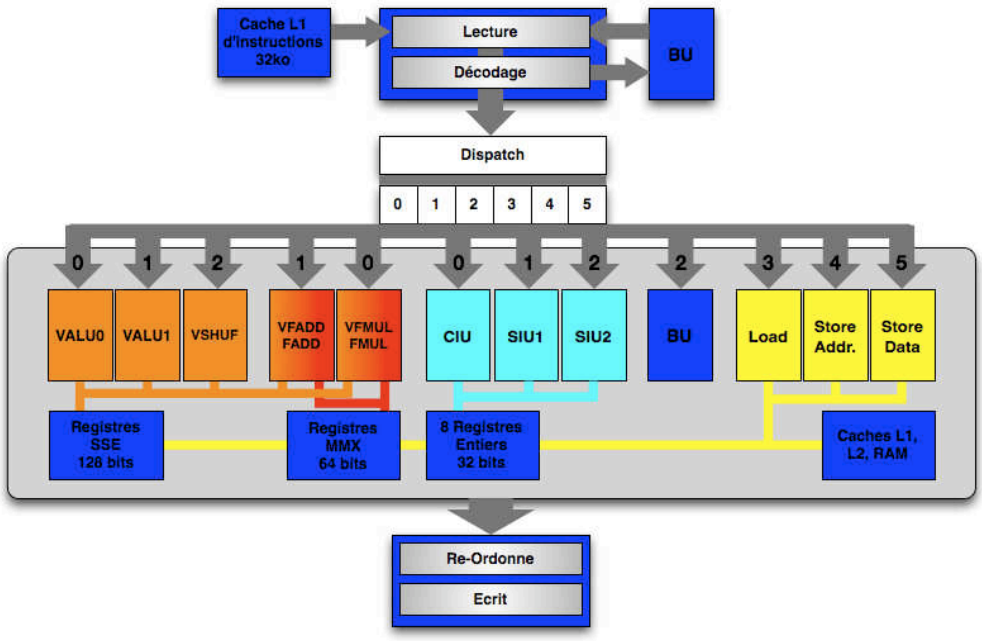

Figure 2.1.17: Architecture du Core

L'architecture Core [28][29] montre de nombreuses améliorations en comparaison de la précédente architecture P6. Premièrement, le bus de données menant aux décodeurs se trouve élargi et transmet maintenant 20 octets soit 5 instructions x86 dans le cas général.

L'unité de décodage est renforcée par un décodeur simple supplémentaire, ce qui permet de décoder jusqu'à 4 instructions par cycle.

Le Pentium M fut le processeur implantant pour la première fois la micro-fusion [35] qui permet de fusionner deux micro-opérations et de les décoder comme étant une seule micro-opération. Ceci permet d'utiliser efficacement les décodeurs de l'architecture et d'économiser de l'énergie en réduisant le trafic sur les bus. L'architecture Core, quant à elle, marque l'apparition de la macro-fusion [35] où il est toujours question de fusion de deux micro-opérations mais, cette fois ci, au niveau du dispatch et de l'éxécution, un véritable gain de performance. Ces techniques concernent essentiellement les opérations de comparaison et de branchement.

Au niveau de la distribution des instructions, un port supplémentaire est ajouté, ce qui fait un total de 6 ports dont 3 alimentant les unités d'exécution.

Les unités de calculs sont nettement plus nombreuses, on peut ainsi dénombrer:
- 2 unités arithmétiques et logiques vectorielles
- 1 unité de permutation.
- 2 unités flottantes qui prennent en charge aussi bien les opérations scalaires que vectorielles. Une des deux unités est spécialisée pour l'addition tandis que l'autre est spécialisée pour la multiplication.
- 3 unités entières 64-bits, une pour les calculs complexes et les deux autres pour les opérations simples.
- 3 unités destinées aux lectures et écritures mémoire.

Concernant les unités vectorielles, l'architecture Core permet pour la première fois de faire du calcul vectoriel à plein régime. Le bus de données ayant été élargi à 128 bits, l'architecture est donc capable d'exécuter une instruction vectorielle par cycle d'horloge au contraire des architectures précédentes qui avaient besoin de deux cycles telles que le Pentium M et le Core Duo.

L'unité entière permet aussi d'effectuer des opérations 64 bits avec une latence d'un seul cycle d'horloge, ce qui est une première pour Intel et permet même de se placer devant le PPC 970 qui présente une latence de 2 cycles.

2.1.4.4. Nehalem

Le Nehalem est le futur processeur Intel, héritant des évolutions des micro-architectures précédentes Core 2 Duo dont le Penryn.
Les principales évolutions sont:
- l'intégration du contrôleur mémoire voire même celle d'un processeur graphique ou d'un contrôleur PCI-Express sur certaines déclinaisons du processeur

- l'utilisation d'un nouveau bus de données système et inter-processeurs *QuickPath Interconnect*
- l'ajout de 7 instructions supplémentaires SSE4.2, au jeu d'instructions SSE4 déjà présent dans le Penryn
- le retour de l'Hyperthreading d'Intel permettant l'exécution de deux threads simultanés sur chaque cœur du processeur

L'évaluation du processeur en calcul vectoriel flottant a montré des performances jusqu'à 8% plus rapides que celles du Core 2 Duo.

2.2. C` mpilati` n et aide à la pr` grammati` n

Les processeurs modernes mettent à disposition un grand nombre de techniques permettant d'accélérer l'exécution du code. Ces techniques peuvent être implicites comme l'exécution dans le désordre ou la prédiction de branchement. D'autres sont à la charge du programmeur comme l'utilisation des unités vectorielles et l'exploitation du multi-threading lorsque plusieurs cœurs de calcul sont à disposition.

Il convient donc d'assister le travail du programmeur dans la programmation de ces applications et de lui permettre d'exploiter au maximum les performances du processeur.

2.2.1. Les c` mpilateurs

Le compilateur joue le rôle d'interprète entre le programmeur et le processeur. Il permet d'une part de traduire un langage de haut niveau, ou même assembleur, utilisé par le programmeur en langage machine ou assembleur interprétable par le processeur. Il permet, d'autre part, d'optimiser le code sur différents critères de choix comme la taille ou la performance du code généré.

Il existe un grand nombre de compilateurs dont le plus connu est le compilateur GCC. À l'origine, l'acronyme GCC signifiait GNU C Compiler. Il a été créé par le projet GNU, ce dernier étant le premier projet de production de logiciels libres. Aujourd'hui, le compilateur GCC a été étendu à d'autres langages de programmation que le langage C et signifie alors GNU Compiler Collection. D'autres compilateurs existent dont les compilateurs XLC d'IBM et ICC d'Intel, tous deux conçus dans le but de générer le meilleur programme possible pour leurs architectures respectives. Ainsi ICC ne cible que les architectures d'Intel comme l'architecture x86 tandis que le compilateur XLC ne génère que du code pour les architectures Power d'IBM tout en incluant le PowerPC.

Les compilateurs mettent à disposition plusieurs niveaux d'optimisation -O, -O2, -O3 ou -Os. Ce sont des packages d'options qui permettent d'effectuer un certain nombre d'opérations sur le code source afin d'optimiser le programme en performance ou en taille. Le bénéfice de ces optimisations n'est plus à démontrer. Cependant, elles n'ont pas permis d'obtenir les performances maximales du processeur bien que l'algorithme ait été conçu pour exploiter à 100% les ressources du processeur. Il convient toujours de dérouler manuellement les boucles et surtout d'ordonnancer les opérations convenablement afin de pouvoir maintenir le débit d'exécution

maximal du processeur. Concernant le cas du calcul vectoriel sur PowerPC 970, le débit maximal est d'une instruction de calcul par cycle d'horloge. Aucune des options d'optimisation n'a pu atteindre ces performances crêtes et seule l'aide d'un ordonnancement en amont de la compilation a permis de s'en approcher et ainsi d'améliorer les performances de l'algorithme de 20%.

Une autre facette de la compilation et de ces optimisations possibles est la *vectorisation*. La *vectorisation* représente le processus qui consiste à passer du code scalaire initial au code exploitant les ressources SIMD du processeur. C'est un travail de réflexion nécessitant de dévoiler le parallélisme d'exécution au sein de l'algorithme puis de programmer à l'aide des instructions vectorielles. Il existe, au sein de chaque compilateur, des options permettant d'automatiser ce processus:

- `-ftree-vectorize` pour GCC [36]
- `-vec` pour ICC
- `-qhot` pour XLC

D'autres types de logiciels permettent d'automatiser la *vectorisation* de boucles, comme par exemple VAST pour le langage AltiVec [37].

Tous ces compilateurs montrent de très bonnes performances. Cependant, la *vectorisation* automatique n'est pas encore une méthode maîtrisée et ne peut encore remplacer complètement l'intervention d'un œil expert. La *vectorisation* automatique d'un compilateur s'est montrée très performante sur des algorithmes de seuillage où les calculs sont identiques et les données indépendantes. Ce type d'optimisation se nomme *vectorisation de boucle* où le but est de traiter 4 itérations de boucle en une seule itération de calcul. Les gains apportés sur une telle optimisation sont de l'ordre de 4. Cependant cette amélioration ne devrait pas refléter seulement la vectorisation de l'algorithme mais aussi l'élimination des tests conditionnels grâce aux instructions SIMD de comparaison nécessaires à l'algorithme de seuillage.

2.2.2. Les l` giciels d'aide au pr` t` typage d'applicati` ns

Dans le même registre que la *vectorisation*, il existe aussi la *parallélisation* de l'algorithme. Face à la démocratisation des processeurs multi-cœurs et à la multiplication croissante du nombre de processeurs au sein des architectures de calcul, il convient d'exploiter la totalité des ressources mises à disposition afin d'optimiser au mieux l'application d'intérêt.

De nombreux outils sont mis à la disposition du programmeur afin de l'aider au développement d'applications pour des architectures multi-cœurs et d'en exploiter le maximum de performance. Il existe ainsi des méthodes implicites comme OpenMP [38], opérant avec l'aide de pragmas, sorte de commentaires à destination du compilateur, au sein d'un code écrit en langage haut niveau tel que le C/C++ ou Fortran afin de paralléliser automatiquement les boucles de calcul lors de la compilation. D'autres méthodes sont à l'inverse explicites telles que le standard MPI [39] où le programmeur contrôle explicitement le parallélisme de son application et les communications engendrées.

D'un point de vue plus haut niveau, il existe des ateliers de programmation permettant au développeur de modéliser son architecture parallèle et son algorithme afin de déployer son algorithme et d'optimiser la répartition des tâches qui seront allouées à chaque cœur de calcul. Ces ateliers permettent entre autre de générer automatiquement les communications inter processeurs causées par la parallélisation du code. De tels ateliers sont proposés par SPEAR de Thales Research Technology, SynDEx de l'INRIA basé sur la méthodologie d'Adéquation Algorithme-Architecture (AAA) [54], Gedae [55] et HMPP de CAPS entreprise [56].

De la même manière qu'OpenMP, Parallel Fortran [57] est un langage de programmation spécifique permettant de *paralléliser* le traitement des boucles de calcul.

SPIRAL est un générateur de code haute-performance spécialisé dans les transformées de traitement de signal linéaire (DSP) [58].

2.3. Les alg` rithmes de traitement de signal

Il existe de nombreux domaines où les performances d'une application sont déterminantes quant à son intégration sur un processeur donné. Parmi elles, les applications multimédia ont largement contribué à l'évolution rapide des processeurs. Dans le milieu de l'embarqué, ce sont les algorithmes de traitement de signal qui sont à l'honneur. Les applications RADAR et SONAR sont deux domaines exigeants en puissance de calcul. Plus particulièrement pour les traitements RADAR, les latences de calcul doivent être réduites au minimum sous peine de ne pas pouvoir détecter en temps réel les cibles en mouvement.

Ces applications font appel à des bibliothèques mathématiques contenant une liste exhaustive de fonctions élémentaires telles que des transformées, des filtres et des opérations matricielles. Ce sont des algorithmes de type flot de données opérant sur de larges tableaux d'une ou plusieurs dimensions. Parmi les différents algorithmes qui composent ces bibliothèques, un est sujet à diverses optimisations depuis les débuts de son utilisation. Il s'agit de la transformée de Fourier rapide.

2.3.1. Le traitement de signal et ses alg` rithmes

Il est difficile de donner une définition précise et succincte de ce qu'est le traitement du signal. Aussi pour bien comprendre, il faut définir ce qu'est un signal. Un signal est l'observation d'un phénomène physique dépendant d'une ou plusieurs variables. Il correspond au support physique d'une information. Par exemple, les signaux sonores correspondent aux fluctuations de la pression d'air apportant un message à nos tympans, les signaux lumineux sont constitués d'ondes lumineuses véhiculant une information à notre rétine. Mathématiquement, il s'agit d'une fonction d'une ou plusieurs variables [40]. La plus grande famille de signaux étudiés correspond aux signaux d'une seule variable, classiquement le temps. On parle de signaux monodimensionnels. Cependant depuis l'avènement des puissances de calculs, on s'intéresse également aux signaux dépendants de plusieurs variables comme une image caractérisée par une intensité lumineuse dépendante de ses coordonnées sur un plan ou un signal RADAR dépendants du temps et la direction de la propagation de

l'onde électromagnétique réceptionnée par l'antenne. On parle alors de signal spatio-temporel.

Il existe plusieurs classes de signaux. Lorsque la fonction, décrivant le signal, est parfaitement définie en tout point, le signal est dit déterministe. A l'inverse, ce dernier est dit aléatoire. Suivant que les variables régissant le signal soient continues ou discrètes, le signal est dit respectivement analogique ou discret. Un signal discret, dont l'amplitude est également discrète, est connu sous l'appellation de signal numérique.

Le traitement de signal est constitué de la réunion de méthodes et de techniques dont les finalités sont nombreuses et propres aux domaines auxquels sont appliquées ces dites méthodes. Parmi les principales applications, on cite les domaines de la télécommunication tels que les réseaux GSM ou encore l'ADSL, les domaines militaires avec la détection RADAR et SONAR et les domaines médicaux avec notamment les diverses formes d'imageries médicales.

L'outil le plus connu du traitement de signal est la transformée de Fourier. À un signal quelconque donné correspond deux représentations, une représentation temporelle et une représentation fréquentielle. Le passage d'une représentation à l'autre se fait par l'intermédiaire de cette transformée.

Il existe différentes transformées ayant chacune leurs propriétés uniques. Ainsi la transformée de Fourier permet de mettre en avant le spectre d'amplitude et de phase d'un signal. La transformée en cosinus est une transformée proche de la transformée de Fourier qui a la particularité de générer des coefficients réels contrairement la tranformée de Fourier. Elle est utilisée pour des problèmes de compression, notamment le format JPEG et MP3 du à son excellente propriété de regroupement de l'énergie. La transformée en ondelette est un autre exemple de transformées utilisée aussi dans la compression, cette fois-ci avec ou sans perte. Elle est à l'origine du format de compression JPEG2000.

La propagation d'un signal n'est pas sans perte. Ainsi un signal reçu ne coïncidera pas parfaitement au signal émis. Le signal reçu est une superposition du signal d'entrée avec un signal perturbateur généralement appelé bruit. Le traitement de signal apporte une série de techniques permettant de restituer le signal d'origine. Ces méthodes se font avec l'aide de filtres tels que les filtres ARMA (Auto-Régressive, Moyenne Mobile), RII (Réponse Impulsionnelle Infinie) et RIF (Réponse Impulsionnelle Finie).

Le traitement de signal trouve ainsi son application dans de nombreux domaines. Dans un soucis d'optimisation nous nous intéresserons à la transformée de Fourier, fonction utilisée dans la grande majorité des applications de traitement de signal.

2.3.2. La transf` rmée de F` urier

Le transformée de Fourier dans le domaine du traitement de signal, est une fonction permettant d'associer à un signal temporel, son spectre en fréquences. Le signal est représenté comme étant la somme des fonctions trigonométriques de toutes les fréquences qui forment son spectre.

La transformée de Fourier est une opération qui transforme une fonction intégrale sur $\mathbb{R}$ en une autre fonction, décrivant le spectre fréquentiel de cette dernière. Ainsi si une fonction *f* est intégrable sur $\mathbb{R}$, sa transformée de Fourier est la fonction *F* donnée par la formule suivante:

eq 1:
$$F(\nu) = \int_{-\infty}^{+\infty} f(t) e^{-i2\pi\nu t} dt$$

et son inverse:

eq 2:
$$f(t) = \int_{-\infty}^{+\infty} F(\nu) e^{+i2\pi\nu t} d\nu$$

2.3.3. La transf` rmée de F` urier discrète

L'implémentation des algorithmes de traitement de signal en informatique ne se fait malheureusement pas sur l'espace infini des réels $\mathbb{R}$, ce qui nécessiterait un temps de calcul infini. En revanche, la théorie du traitement de signal numérique permet d'adapter la transformée de Fourier sur un nombre fini d'échantillons donnant lieu à l'algorithme de transformée de Fourier discrète (DFT).

Soit un signal numérique de N échantillons, la définition mathématique de sa transformée de Fourier discrète est:

eq 3:
$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-2i\pi \frac{nk}{N}}$$

et son inverse:

eq 4:
$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{+2i\pi \frac{nk}{N}}$$

2.3.4. La transf` rmée de F` urier rapide

L'algorithme de transformée de Fourier discrète, décrit comme précédemment, requiert un total de N(N-1) additions et N² multiplications. Ce nombre d'opérations induit une complexité de calcul importante exprimée en O(N²).

Ainsi les transformées de Fourier rapides, plus communément appelées FFT pour Fast Fourier Transform, ont pour but d'accélérer le calcul de la transformée de Fourier. Ceci est possible grâce aux propriétés intrinsèques de la transformée de Fourier discrète qui, suivant un certain nombre d'hypothèses, peut s'exprimer avec un plus faible nombre d'opérations.

L'algorithme le plus connu est celui de J. W. Cooley et de J. W. Tukey [41] de part leur célèbre publication datant de 1965. Il est construit autour d'une propriété de la transformée de Fourier qui démontre que le calcul d'une transformée de Fourier sur N échantillons, sachant que N = r.n, peut être factorisé par le calcul de r transformées de Fourier sur n échantillons.

Il existe, à ce jour, un grand nombre d'algorithmes FFT, nécessitant chacun d'entre eux le respect d'hypothèses permettant d'exploiter les propriétés correspondantes et d'obtenir le maximum des performances de ces algorithmes FFT. Ainsi un algorithme peut être efficace dans une situation et inutile dans d'autres situations. Il est donc indispensable de bien choisir son algorithme avant de se lancer dans l'implémentation et l'optimisation de ce dernier.

2.3.5. L'alg` rithme FFT radix-2 à entrelacement temp` rel

Commençons tout d'abord par présenter l'algorithme le plus répandu, l'algorithme FFT radix-2 comme l'avait démontré Carl Friedrich Gauss en 1805.

Soit la transformée de Fourier discrète, définie comme précédemment:

eq 5:
$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-2i\pi \frac{nk}{N}}$$

Nommons par commodité les coefficients trigonométriques de la façon suivante:

eq 6:
$$\omega_N = e^{-2i\frac{\pi}{N}}$$

L'équation de la transformée de Fourier discrète devient alors:

eq 7:
$$X(k) = \sum_{n=0}^{N-1} x(n) \omega_N^{nk}$$

L'algorithme FFT radix-2, quant à lui est construit autour de deux propriétés de ce coefficient:

eq 8: $$\omega_N^{2nk} = \omega_{\frac{N}{2}}^{nk}$$

eq 9: $$\omega_N^{n(k+\frac{N}{2})} = (-1)^n \omega_N^{nk}$$

La première propriété met en évidence le lien entre les coefficients invoqués dans le calcul de la transformée de Fourier pour un signal de N échantillons avec ceux de la même transformée d'un signal de N/2 échantillons.

La deuxième propriété montre une certaine périodicité des coefficients de par leur nature trigonométrique.

Ainsi, dans le cas où N est un multiple de deux, si l'on réécrit la transformée de Fourier discrète en deux sommes indépendantes tout en distinguant les échantillons d'indice pair de ceux d'indice impair, nous obtenons:

eq 10: $$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)\omega_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\omega_N^{(2n+1)k}$$

eq 11: $$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)\omega_N^{2nk} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\omega_N^{2nk}$$

En réinjectant la première propriété dans l'équation précédente, celle-ci devient:

eq 12: $$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)\omega_{\frac{N}{2}}^{nk} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\omega_{\frac{N}{2}}^{nk}$$

Ainsi pour k ∈ [0, N/2-1], le calcul d'une transformée de Fourier de N échantillons revient à calculer deux transformées de Fourier sur N/2 échantillons:

eq 13: $$X(k) = DFT(x_{2n})(k) + \omega_N^k DFT(x_{2n+1})(k)$$

Concernant la deuxième moitié des calculs, c'est à dire pour k ∈ [N/2, N-1], ceci revient à écrire l'équation toujours pour k ∈ [0, N/2-1] de la façon suivante:

eq 14: $$X(k+\frac{N}{2}) = DFT(x_{2n})(k+\frac{N}{2}) + \omega_N^{k+\frac{N}{2}} DFT(x_{2n+1})(k+\frac{N}{2})$$

Le spectre d'un signal de N échantillons étant N-périodique et en appliquant la deuxième propriété énoncée concernant les coefficients, l'équation devient:

eq 15: $$X(k+\frac{N}{2}) = DFT(x_{2n})(k) - \omega_N^k DFT(x_{2n+1})(k)$$

En récapitulant, dans le cas où le nombre N d'échantillons est multiple de deux, calculer une transformée de Fourier d'un tel signal revient à calculer deux transformées de Fourier sur deux signaux de N/2 échantillons.

eq 16: $$X(k) = DFT(x_{2n})(k) + \omega_N^k DFT(x_{2n+1})(k)$$

eq 17: $$X(k+\frac{N}{2}) = DFT(x_{2n})(k) - \omega_N^k DFT(x_{2n+1})(k)$$

De cette écriture, est tiré le diagramme du célèbre opérateur papillon de l'algorithme FFT radix-2.

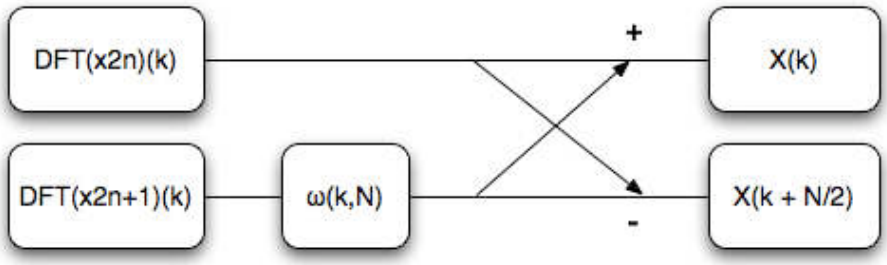

Figure 2.3.1: Diagramme de l'opération Papillon radix-2

L'intérêt de cette opération est de réduire le nombre d'opérations à effectuer pour le calcul de la transformée de Fourier. Le tableau suivant montre les gains apportés en fonction du nombre d'échantillons constituant le signal.

	DFT		Radix-2	
	Multiplica-tions	Additions	Multiplica-tions	Additions
N	N^2	N(N-1)	$(N^2 + N)/2$	$N^2/2$
16	256	240	136	128

Tableau 2.3.1: Gain apporté par l'écriture de la DFT sous forme radix-2

Cette propriété étant récursive, il en est de même pour les deux DFT de taille N/2. Ainsi, dans le cas où le nombre d'échantillons constituant le signal est égal à une puissance de deux, $N = 2^m$, cette propriété est applicable *m* fois jusqu'à obtenir des DFT sur 2 échantillons qui se traduisent par une addition complexe et une soustraction complexe. *m* étages de N/2 papillons sont alors à calculer, ce qui donne une complexité de calcul de:

- $m.N/2 = \log_2(N).N/2$ multiplications complexes
- $m.N = \log_2(N).N$ additions complexes

	DFT		FFT Radix-2	
	Multiplications	Additions	Multiplications	Additions
N	N^2	N(N-1)	$\log_2(N).N/2$	$\log_2(N).N$
16	256	240	32	64

Tableau 2.3.2: Dénombrement des opérations entre une DFT et une FFT radix-2

La multiplication complexe représente quatre multiplications flottantes et deux additions flottantes. L'addition complexe, quant à elle, représente deux additions flottantes. Ainsi cet algorithme diminue le nombre d'opérations flottantes à $5N.\log_2(N)$, soit une complexité de calcul en $O(N.\log_2(N))$.

En revanche, cet algorithme entraîne un ordonnancement des données en amont du calcul FFT, d'où son nom précisant l'entrelacement temporel des données. Le déploiement des opérations pour N=16 permet de remarquer qu'en entrée d'algorithme, l'ordre des données n'est pas l'ordre naturel du signal mais ce dernier est obtenu par *bit reversing* de la position de chaque échantillon.

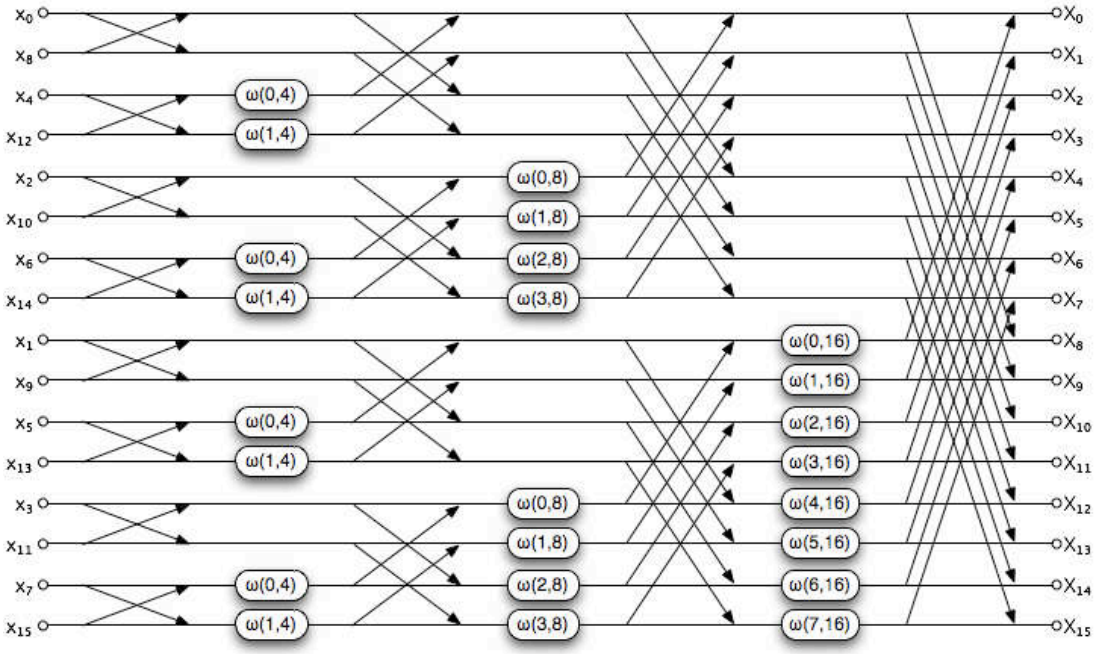

Figure 2.3.2: Diagramme papillon déployé pour une FFT N=16

2.3.6. L'alg` rithme FFT radix-2 à entrelacement fréquentiel

Contrairement à son homologue à entrelacement temporel, l'algorithme FFT radix-2 à entrelacement fréquentiel se caractérise par l'ordre résultant de la transformée sur le signal d'entrée. En effet, l'ordre obtenu après transformée est analogue à l'ordre nécessaire en entrée de l'algorithme à entrelacement temporel, faisant appel au *bit-reversing*. En revanche, l'ordre nécessaire en entrée de cet algorithme correspond à l'ordre naturel des échantillons.

Le opérateur papillon de cet algorithme devient:

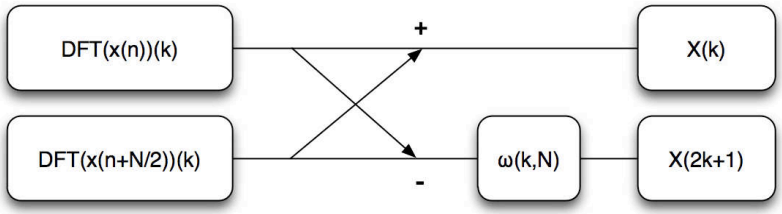

Figure 2.3.3: Diagramme papillon radix-2 à entrelacement fréquentiel

Le nombre d'opérations par papillon est identique à l'algorithme radix-2 à entrelacement temporel, 3 additions, 3 soustraction et 4 multiplications. L'algorithme se déroule aussi de la même façon avec $\log_2(N)$ étages de N/2 papillons où N est une puissance de 2.

Les deux algorithmes se distinguent ainsi par la localité de l'ordonnancement et de la nature du papillon.

2.3.7. L'alg` rithme FFT radix-3

Comme il est démontré dans la publication de J. W. Cooley et J. W. Tukey [41], il existe différentes factorisations possibles dans la répartition du calcul de la transformée de Fourier discrète. Cette décomposition donne alors la valeur du radix. Ainsi grâce aux propriétés des fonctions trigonométriques qui composent l'algorithme de la transformée de Fourier discrète, il est démontré que les trois radix les plus efficaces dans l'optimisation du calcul de la FFT sont les radix 2, 3 et 4. De la même manière que le radix 2, il est possible d'exprimer la transformée de Fourier comme étant le calcul de 3 ou 4 transformées de Fourier indépendantes suivant que le nombre d'échantillons constituant le signal soit multiple de 3 ou de 4.

L'algorithme radix-3 est présenté comme étant le radix présentant la plus faible complexité de calcul [41].

2.3.8. L'alg` rithme FFT radix-4

L'algorithme FFT radix-4 est une décomposition de l'algorithme de transformée de Fourier discrète en 4 transformées indépendantes de taille N/4. Ainsi le papillon caractéristique de cet algorithme se présente sous la forme:

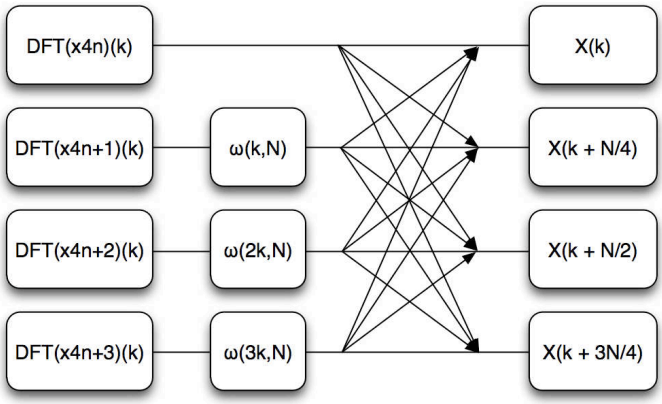

Figure 2.3.4: Diagramme papillon radix-4

En comparaison avec l'algorithme radix-2, l'algorithme radix-4 est une succession de $\log_4(N)$ étages de N/4 papillons où N est une puissance de 4 représentant le nombre d'échantillons constituant le signal à considérer.

2.3.9. L'alg` rithme FFT Split-Radix

L'algorithme FFT Split Radix est une combinaison des algorithmes radix-2 et radix-4 donnant à son papillon, sa forme caractéristique en L. Il divise récursivement le calcul de la FFT en une FFT de taille N/2 et deux FFT de taille N/4.

eq 18:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n).e^{-(i.\frac{2\pi(2n)k}{N})} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1).e^{-(i.\frac{2\pi(4n+1)k}{N})} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3).e^{-(i.\frac{2\pi(4n+3)k}{N})}$$

Après identification des trois sous-transformées, il ressort l'écriture suivante:

eq 19:

$$X(k) = DFT_{\frac{N}{2}}[x(2n)] + \omega_N^k.DFT_{\frac{N}{4}}[x(4n+1)] + \omega_N^{3k}.DFT_{\frac{N}{4}}[x(4n+3)]$$

Créé en 1984 par P. Duhamel et H. Hollmann [42], il représente l'algorithme possédant le plus faible nombre d'opérations flottantes, multiplications et additions confondues, sur des signaux de taille égale à une puissance de deux.

Plus récemment, en 2007, ce nombre d'opérations arithmétiques nécessaires au calcul de l'algorithme a encore été amélioré par S. G. Johnson et M. Frigo [43].

2.3.10. L'alg` rithme FFT de Win` grad

L'algorithme FFT de Winograd est un algorithme efficace connu pour sa faible complexité en multiplications [44]. L'auteur du même nom, Shmuel Winograd a en effet mis en évidence que pour un certain nombre de cas où la taille des signaux est égale à 2,3,4,5,7,8,11,13 et 16, l'écriture de la transformée de Fourier peut être exprimée avec un plus faible nombre de multiplications.

Cependant, cet algorithme a pour conséquence d'augmenter en contrepartie le nombre d'additions, ce qui ne permet pas de réduire la complexité globale de l'algorithme. Ceci avait un sens pour les architectures montrant de fortes latences pour le calcul d'une multiplication. En revanche, pour la totalité des architectures embarquées et étudiées dans ce mémoire, le temps d'exécution d'une multiplication flottante est identique à celui de l'addition flottante.

2.3.11. L'alg` rithme FFT de St` ckham

L'algorithme FFT de Stockham est né de l'idée de concevoir un algorithme pouvant s'affranchir du bit reversing [45]. Il est qualifié d'*autosort* par les ouvrages car l'ordonnancement se fait petit à petit entre chaque étage de calcul.

Le cœur de calcul de l'algorithme peut s'appuyer sur les radix-2 ou 4. La structure de l'algorithme est ainsi conforme aux algorithme radix-2 et 4. Dans le cas d'une implémentation radix-2, l'algorithme se compose de $\log_2(N)$ étages de N/2 papillons où N est une puissance de 2.

Ainsi la complexité de calcul entre un algorithme de Stockham et un algorithme de type radix est strictement identique.

L'algorithme FFT de Stockham figure dans de nombreux articles étudiant la programmation parallèle de l'algorithme [46][47] et plus récemment, il a été utilisé dans l'implémentation de la FFT par IBM pour son processeur Cell Broadband Engine [48][49].

En effet, malgré ses fortes capacités de *parallélisation* avec ses 8 cœurs de calcul SIMD, le CELL présente néanmoins un très faible parallélisme d'instructions par cœur. Seules deux instructions peuvent être distribuées par cycle et par cœur, soit une instruction de calcul en parallèle d'une instruction d'accès mémoire. Le calcul du bit-reverse est dans le cas présent à considérer dans la complexité de l'algorithme puisqu'il ne peut être traiter en parallèle du calcul de la transformée.

Lorsque le calcul du bit reverse vient interférer avec le calcul du papillon, il est alors judicieux de s'affranchir de cette étape de calcul en s'intéressant à des algorithmes *auto-sort* tels que l'algorithme FFT de Stockham. Le choix d'IBM s'est porté sur l'implémentation de l'algorithme de Stockham radix-4.

2.3.12. Le bit-reverse

Une grande partie des algorithmes de transformée de Fourier rapide présente une caractéristique commune, celle d'un ordonnancement des données.

Considérons l'ordonnancement de l'algorithme FFT radix-2 à entrelacement temporel, visible sur la figure 3.4.2. Cet ordre est obtenu par application de l'opération bit-reverse.

Chaque échantillon dispose d'un indice permettant de les différencier au sein du signal. Ainsi l'ordre naturel des échantillons va de l'indice 0 à N-1 où N représente la taille du signal. L'opération bit-reverse consiste à inverser l'écriture binaire de ces indices.

Le nombre de bits nécessaire à l'écriture binaire de l'indice dépend directement de la taille du signal. Ainsi un signal de 16 échantillons nécessitera 4 bits pour exprimer ses indices en écriture binaire tandis qu'un signal de 256 échantillons nécessitera quant à lui un total de 8 bits.

Ceci a un impact direct sur l'opération bit-reverse. L'échantillon d'indice 9 du premier signal ne sera pas permuté alors le même échantillon du deuxième signal sera permuté avec l'indice 144.

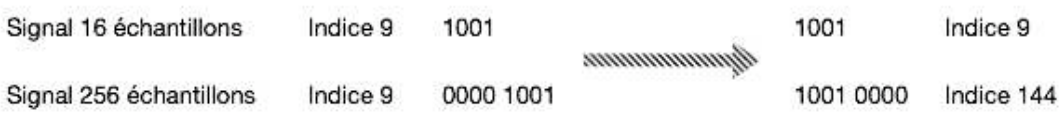

Figure 2.3.5: Exemple d'opération bit-reverse

2.4. C` nclusi` ns

L'optimisation est un vaste domaine dont les solutions dépendent de multiples facteurs. Les premiers facteurs proviennent de l'architecture. Nous avons vu qu'il existait un vaste choix d'architectures dans le domaine des processeurs embarqués qui diffèrent selon de nombreux critères. Néanmoins, ces dernières tendent vers des solutions communes telles que le calcul vectoriel et le fonctionnement superscalaire. D'autres facteurs proviennent quant à eux de l'algorithme.

L'optimisation peut être bidirectionnelle. En effet, le design du système peut être fait en fonction de l'application à porter et des contraintes de performances souhaitées. À l'inverse, notre cas d'étude positionne l'architecture comme une contrainte. L'optimisation consiste en une série d'opérations visant à maximiser les performances de l'algorithme choisi sur le processeur imposé.

L'étude qui va suivre concerne donc l'optimisation de la transformée de Fourier rapide sur le PowerPC 970 FX. Plus précisément cette étude se concentre sur l'algorithme FFT radix-2 à entrelacement temporel pour ses qualités d'adéquation aux unités SIMD. Le PowerPC 970 est, quant à lui, une référence dans le domaine du calcul intensif en milieu embarqué.

3. Optimisati` n de la transf` rmée de F` urier rapide

Le traitement de signal est à l'origine de nombreuses applications scientifiques dont certaines ont des contraintes temps réel incontournables. Dans le domaine aéronautique, la détection RADAR est un exemple concret d'application temps réel. Les latences de calcul se doivent d'être réduites au minimum.

L'optimisation des algorithmes de traitement de signal est un enjeu pratiqué depuis les débuts de l'informatique et ne cessent d'être d'actualité en dépit de la montée en puissance de calcul continue des processeurs.

En effet, d'une part, les méthodes d'optimisation choisies par les concepteurs de processeurs changent l'implémentation des algorithmes choisis. Le gain de puissance de calcul ne s'atteint plus avec la montée en fréquence. Cette dernière a tendance à stagner ces dernières, atteignant les limites de la physique. Les solutions s'orientent aujourd'hui vers le parallélisme et les programmeurs se doivent d'y faire face.

D'autre part, 10% de performances gagnés sur un algorithme tel que la FFT, qui sera exécuté infiniment, représente une diminution de 10% de la puissance de calcul employée dans le système donné. Ceci se traduit par la diminution du nombre de cartes de calcul utilisées dans le système, qui se traduit à son tour en:

- diminution du poids total du système
- diminution de la consommation totale du système
- diminution de la dissipation thermique totale du système
- diminution du prix total du système

À l'exception de la dernière conséquence énoncée qui permet de s'imposer au niveau concurrentiel, toutes agissent directement sur les contraintes capitales régissant l'intégration d'un système pour des solutions embarquées.

3.1. Les bibli` thèques de traitement de signal

Afin de simplifier le traitement de signal sur processeur, des APIs sont disponibles aux programmeurs. Ce sont des fonctions de haut niveau permettant le développement rapide d'applications. Il existe différents standards comme BLAS [61] et VSIPL [62]. Ces standards permettent d'unifier les diverses implémentations de librairies optimisées.

BLAS, Basic Linear Algebra Subprograms, est un standard pour générer, comme son nom l'indique des opérations basiques d'algèbre linéaire. Ce sont essentiellement des opérations de type multiplication vectorielle ou matricielle.

VSIPL, Vector Signal Image Processing Library, est, quant à lui, un standard dans le développement d'applications de traitement de signal. Ce sont aussi des opérations orientées vecteur et matrice de part la nature des signaux étudiés auxquelles sont ajoutées les opérations mathématiques essentielles au traitement du signal. Sont présents, les opérations basiques d'additions, de multiplications entre vecteurs ou matrices, le calcul de la racine carrée

ou de l'exponentielle d'un vecteur ou d'une matrice, le calcul de diverses fenêtres nécessaires aux filtrages des signaux et ainsi que différentes versions de transformée de Fourier, complexe vers complexe, complexe vers flottant, simple ou double précision, in-place ou out-of-place suivant si le résultat écrase ou non le signal d'entrée.

Il existe d'autres librairies répondant à leur propre standard comme la librairie IPP [65], Integrating Performance Primitives, d'Intel proposant tous les outils nécessaires au calcul scientifique mais aussi FFTW [66], Fastest Fourier Transform in the West, issu d'une équipe du MIT proposant une solution complète de calcul de transformée de Fourier optimisée. vDSP [67] est quant à lui une bibliothèque propriétaire d'Apple, à l'image d'IPP, proposant une liste exhaustive d'algorithmes de calcul scientifique optimisés.

Parmi les algorithmes de calcul, un d'entre eux est sujet à de multiples benchmarks, la transformée de Fourier rapide sous sa forme la plus utilisée, c'est à dire simple-précision, complexe vers complexe. Cet algorithme de calcul est non seulement un algorithme déterminant quant aux performances des diverses applications de traitement de signal mais aussi un algorithme permettant de juger de la performance d'une architecture. C'est pourquoi l'implémentation de la transformée de Fourier requiert une optimisation maximale sur l'architecture cible et fait appel à des choix d'optimisations dites fines dépendantes de l'architecture cible.

3.2. Etude d'adéquati` n alg` rithmes architectures

Parmi les algorithmes de transformée de Fourier rapide étudiés précédemment, un certain nombre d'entre eux ne s'adapte plus aux architectures processeurs actuelles, d'autres nécessitent plus de réflexion quant à leur efficacité. Ce chapitre vise donc à étudier les performances estimées de chaque algorithme sur les processeurs embarqués que nous avons étudiés.

Il existe un grand nombre d'algorithmes FFT, chacun étudié de façon à être optimisé suivant certaines conditions. Nous allons donc, dans ce chapitre, étudier le comportement de différents algorithmes sur les architectures d'intérêt, mettre en avant les points forts et les points faibles de chacun afin de sélectionner le ou les algorithmes présentant les plus grandes dispositions quant à l'implémentation sur processeurs embarqués.

Ainsi nous jugerons de la qualité d'un algorithme en fonction de plusieurs critères permettant de refléter les prédispositions de l'algorithme sur processeur parallèle. Les deux principaux critères seront:

- ***la complexité de calcul*** de l'algorithme. Le plus faible nombre d'opérations étant souhaitable, nous chercherons aussi à raisonner en nombre d'instructions de part la présence d'opérations fusionnées dans certaines architectures.
- ***l'indépendance des calculs entre données*** permettant de *vectoriser* l'algorithme compte tenu de la présence d'unités SIMD dans la majorité des processeurs embarqués.

Sachant que d'autres facteurs de performance peuvent entrer en compte comme:

- la pression de registres,
- le pipelining ou l'ordonnancement d'opérations,
- le déroulement des boucles d'itération,
- le nombre d'accès mémoire,

il convient de les garder en considération au cours de notre étude.

3.2.1. Alg` rithme FFT radix-2 à entrelacement temp` rel

Si l'on étudie le comportement de l'algorithme FFT radix-2 à entrelacement temporel, il apparaît que ce dernier présente de nombreux atouts.

Tout d'abord, les N/2 papillons indépendants, qui composent les $\log_2(N)$ étages de calcul de l'algorithme radix-2, offrent une excellente indépendance entre calculs. Ceci permet, d'une part, d'obtenir le parallélisme nécessaire à la *vectorisation* de l'algorithme, mais le déroulement des calculs afin de masquer les latences et dépendances de calculs inhérentes.

Ensuite le papillon radix-2 à entrelacement temporel présente une écriture particulière permettant une implémentation efficace en multiplications additions fusionnées.

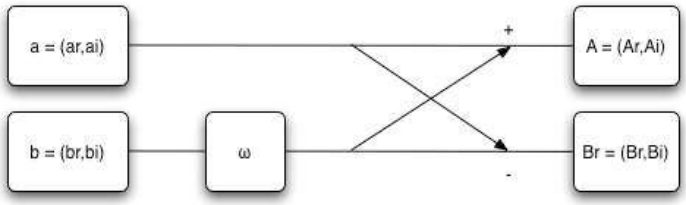

Figure 3.2.1: Papillon Radix-2 à entrelacement temporel

Le cœur de calcul de l'algorithme radix-2 à entrelacement temporel se compose d'une multiplication complexe et de deux additions complexes. Étudions, dans un premier temps, l'écriture classique du papillon tel qu'on le trouve dans la littérature.

eq 20:

$$A_r = a_r + (\omega_r.b_r - \omega_i.b_i) \quad B_r = a_r - (\omega_r.b_r - \omega_i.b_i)$$
$$A_i = a_i + (\omega_i.b_r + \omega_r.b_i) \quad B_i = a_i - (\omega_i.b_r + \omega_r.b_i)$$

Pour que cette écriture soit optimale, le calcul de la multiplication complexe se doit d'être fait une seule et unique fois. Cette écriture représente alors, en terme

d'opérations flottantes, un total de 4 multiplications flottantes et de 6 additions flottantes, soit 10 opérations flottantes.

L'algorithme FFT radix-2 est une succession de $\log_2(N)$ étages, tous composés de N/2 papillons. La complexité de l'algorithme s'élève donc à un total de $5N.\log_2(N)$ opérations flottantes.

L'implémentation avec l'aide des multiplications additions fusionnées n'est pas optimale suivant cette expression du papillon. Deux solutions s'offrent à nous en fonction de la prise en considération ou non du calcul préliminaire de la multiplication complexe.

Si oui, la multiplication complexe nécessite 2 FMA pour sa partie réelle et 2 autres pour sa partie imaginaire. Restent au calcul final du papillon, 2 additions et 2 soustractions, ce qui fait un total de 8 instructions de calcul.

Si non, chacune des quatre composante du résultat du papillon peut être exprimée avec 2 FMA, soit un total de 8 instructions de calcul.

Les deux solutions sont équivalentes en terme de complexité de calcul. Elles permettent de réduire la complexité de calcul du papillon à 8 instructions au lieu de 10 instructions initialement, soit une complexité totale de l'algorithme s'élevant à $4N.\log_2(N)$ instructions. Néanmoins, cette implémentation n'est pas optimale au sens de l'instruction FMA. Une implémentation optimale permettrait de réduire la complexité de calcul à 5 instructions.

Procédons maintenant à la factorisation des calculs par la partie imaginaire du coefficient complexe ω.

eq 21:

$$A_r = a_r - (b_i - \frac{\omega_r}{\omega_i}.b_r).\omega_i \qquad B_r = a_r + (b_i - \frac{\omega_r}{\omega_i}.b_r).\omega_i$$

$$A_i = a_i + (\frac{\omega_r}{\omega_i}.b_i + b_r).\omega_i \qquad B_i = a_i - (\frac{\omega_r}{\omega_i}.b_i + b_r).\omega_i$$

L'équation 21, résultat de la factorisation, est alors entièrement exprimée en multiplications additions fusionnées. Cette nouvelle écriture permet de réduire le calcul du papillon avec seulement 6 instructions de multiplication addition fusionnée. La complexité totale de l'algorithme ne s'élève plus qu'à $3N.\log_2(N)$ instructions.

Cette solution ne permet toujours pas de diviser par deux la complexité de calcul de l'algorithme avec l'aide de l'instruction de multiplication addition fusionnée. En revanche, elle est optimale dans le sens où le papillon radix-2 n'est pas équilibré dans le nombre de multiplications et d'additions qui composent ce dernier. Il est en effet constitué de 4 multiplications flottantes qui font face à 6 additions flottantes. Les 6 additions dimensionnent alors le nombre minimal d'instructions FMA permettant d'exprimer le papillon.

3.2.2. L'alg` rithme FFT Radix-2 à entrelacement fréquentiel

Tout comme son homologue à entrelacement temporel, l'algorithme radix-2 à entrelacement fréquentiel est une succession de $\log_2(N)$ étages de N/2 papillons. Seul diffère l'expression du papillon le constituant.

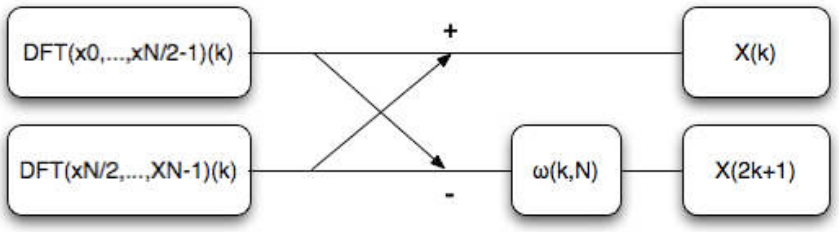

Figure 3.2.2: Papillon Radix-2 à entrelacement fréquentiel

Le papillon radix-2 à entrelacement fréquentiel est toujours composé d'une multiplication complexe et de deux additions complexes. Cependant, la multiplication complexe intervient après les opérations d'additions.

L'étude de l'écriture standard de ce papillon dénombre un total de 4 multiplications flottantes et de 6 additions flottantes, soit un total de 10 opérations flottantes. La complexité de calcul de cette algorithme est alors identique au précédent algorithme, soit de $5N.\log_2(N)$ opérations flottantes.

L'optimisation naïve avec l'aide des instructions de multiplications additions fusionnées permet de réduire cette complexité de calcul à 8 instructions par papillon soit une complexité totale s'élevant à $4N.\log_2(N)$.

Il n'existe cependant aucune factorisation possible permettant de réduire cette complexité ce qui en fait un algorithme moins efficace face à l'algorithme radix-2 à entrelacement temporel.

3.2.3. L'alg` rithme FFT Radix-3

L'algorithme radix-3 est présenté comme étant le plus efficace des radix [41]. Les deux radix suivant dans ce classement d'efficacité sont le radix-2 et le radix-4. La prédominance du radix-3 face aux deux radix cités précédemment, se mesure à 6%.

En revanche, les contraintes imposées sur la taille des signaux, qui doivent être des puissance de 3, ne permettent pas de le considérer plus efficace que les radix-2 et radix-4.

D'une part, le nombre d'échantillons utilisés dans le domaine du traitement du signal et d'image sont en majorité des puissances de 2. Ceci entraîne une adaptation de l'algorithme avec notamment une utilisation de padding.

D'autre part, l'algorithme radix-3 ne permet pas de s'adapter efficacement au parallélisme d'ordre 4 offert par les différentes unités vectorielles.

3.2.4. L'alg` rithme FFT Radix-4

L'algorithme radix-4 est une succession de $\log_4(N)$ étages, chacun constitué de N/4 papillons de la forme:

L'algorithme radix-4 présente l'avantage de s'adapter naturellement au parallélisme d'ordre 4 offert par les unités SIMD actuelles. Cependant, cette propriété existe aussi au sein de l'algorithme radix-2. En effet, ce dernier offre naturellement un parallélisme d'ordre 2. Ainsi, par récursivité, l'algorithme radix-2 offre à son tour ce parallélisme d'ordre 4 utile à l'implémentation vectorielle de l'algorithme.

Comparons alors la complexité de calcul des deux algorithmes. Le papillon radix-4 est composé de 3 multiplications complexes, suivies de 12 additions complexes. Ceci représente un total de 12 multiplications flottantes et de 30 additions flottantes, soit une complexité totale de $10{,}5N.\log_4(N)$ opérations flottantes.

Une factorisation en multiplications additions fusionnées similaire à celle de l'algorithme radix-2 à entrelacement temporel permet de réduire le nombre d'instructions à 30 FMA. La complexité finale de l'algorithme radix-4 s'élève ainsi à $7{,}5N.\log_4(N)$ instructions.

	64	256	1024	4096
radix-2	1152	6144	30720	147456
radix-4	1440	7680	38400	184320

Tableau 3.2.1: Nombre d'instructions en fonction de la taille FFT

L'implémentation radix-2 à entrelacement temporel implique un plus faible nombre d'instructions que le radix-4. De plus, l'algorithme radix-4 se limite aux tailles de signal égales à une puissance de 4. L'algorithme radix-2 à entrelacement temporel est un meilleur candidat de performances sur les architectures SIMD proposant la multiplication addition fusionnée.

3.2.5. L'alg` rithme FFT Split Radix

La forme du papillon Split Radix et l'irrégularité de l'algorithme en fait un mauvais candidat pour une implémentation sur des architectures parallèles telles que les unités SIMD. En effet, l'utilisation des unités vectorielles nécessite des algorithmes proposant un parallélisme d'exécution symétrique contrairement à l'algorithme Split-Radix.

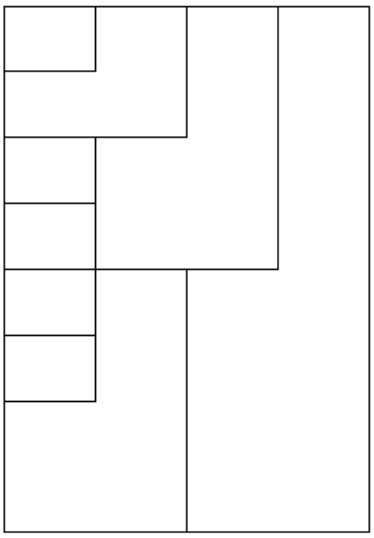

Figure 3.2.3: Structure de l'algorithme avec papillons en L

Néanmoins, une solution régulière de l'algorithme FFT Split Radix existe. Celle-ci nécessite un supplément de N-2 opérations supplémentaires [50].

3.2.6. L'alg` rithme FFT de St` ckham

L'algorithme FFT de Stockham est construit autour d'un papillon de type radix. Ainsi tout comme l'algorithme radix-2 à entrelacement temporel, l'algorithme FFT de Stockham radix-2 est une succession de $\log_2(N)$ étages de N/2 papillons.

L'algorithme de Stockham diffère de l'algorithme radix-2 à entrelacement temporel par son intégration du bit-reversing dans le calcul de la FFT.

Alors que l'algorithme radix-2 à entrelacement temporel nécessite une étape de bit-reversing préliminaire au calcul de la FFT, l'algorithme de Stockham est qualifié d'*autosort* car le réordonnancement des données est fait progressivement durant le calcul de la FFT.

Cet ordonnancement est permis par l'utilisation d'un tableau supplémentaire dans l'implémentation de l'algorithme. Ce tableau "*tampon*" permet de changer la localité des résultats après chaque étage de calcul de la FFT. Le tableau initial et le tableau "*tampon*" alternent leur rôle après chaque étape. Ceci a pour conséquence de ne pouvoir garantir de retourner les résultats de la FFT sur le tableau initial pour toutes les tailles d'échantillons. Lorsque $\log_2(N)$ n'est pas pair, il est nécessaire d'avoir recours à une copie de tableaux pour restituer les résultats de la FFT.

3.2.7. C` nclusi` n sur les alg` rithmes FFT

Parmi les algorithmes précédemment étudiés, deux d'entre eux présentent de bonnes prédispositions à une implémentation sur architectures parallèles. Ces deux algorithmes sont l'algorithme radix-2 à entrelacement temporel et l'algorithme radix-2 de Stockham.

Chacun dispose de ses avantages et de ses contraintes. Cependant, il n'est pas possible au stade actuel de définir la supériorité de l'un par rapport à l'autre. Dans le cas d'un processeur superscalaire possédant un large parallélisme d'exécution, le bit-reversing peut être masqué derrière le calcul du premier étage de calcul FFT. En revanche, cette solution pourrait être pénalisante pour une architecture ayant un parallélisme étroit d'exécution tel que le CELL. Dans quel cas, l'algorithme FFT de stockham semble mieux adapté. D'autant plus que la copie de tableau nécessaire au calcul des FFT dont les tailles sont égales à une puissance de deux impaire peut être contourné par une permutation de pointeur lors d'une communication de type DMA.

Face à la similitude des deux algorithmes, l'algorithme radix-2 à entrelacement temporel sera choisi comme exemple d'optimisation et d'implémentation d'algorithmes de traitement de signal sur architectures parallèles embarquées.

3.3. Optimisati` n de l'alg` rithme FFT radix-2 DIT sur P` werPC 970 FX

L'optimisation de l'algorithme FFT radix-2 à entrelacement sera décrit dans ce chapitre étape par étape en fonction des solutions offertes par le processeur PowerPC 970 FX. Ce processeur est une référence de performances dans le domaine de l'embarqué. D'autre part, il recouvre un grand nombre de solutions d'optimisation utiles à notre problématique.

Cette optimisation permettra de donner un exemple exhaustif d'optimisation d'algorithmes de traitement de signal sur architecture embarqué et de déduire un modèle de programmation optimisée multi-architectures.

L'algorithme FFT radix-2 à entrelacement temporel est composé de trois boucles imbriquées. La première permet de passer d'un étage de la FFT au suivant. La deuxième boucle permet de parcourir tous les coefficients d'un étage de calcul. La troisième et dernière boucle traite un par un l'intégralité des papillons d'un étage faisant intervenir le même coefficient. C'est ainsi que l'algorithme parcourt la totalité des papillons nécessaires au calcul de la transformée de Fourier.

Premières optimisations

Les premières optimisations effectuées font appel à des règles de bonne programmation permettant d'avoir une vue d'ensemble claire de l'algorithme.

Il est préférable, dans un premier temps, d'exprimer les boucles de calcul sous forme de *for* plutôt que de *while.* Les boucles *for* permettent d'avoir une vue plus déterministe de l'exécution des boucles aussi bien pour le programmeur que pour l'unité de prédiction de branchement.

Il existe de nombreux outils de *profiling* permettant de faire une analyse dynamique du code. Par le biais d'une étude statistique du temps d'exécution de l'application étudiée, ces outils sont en mesure de mettre en évidence les portions de code déterminant les performances de l'application et d'émettre des solutions d'optimisation. Les logiciels Shark d'Apple et VTune d'Intel sont de tels outils. Cependant, de tels logiciels trouvent leur intérêt dans le développement d'applications complexes faisant appel à de multiples fonctions et non dans l'optimisation d'une seule et unique fonction. Néanmoins, les fonctions de calcul du logarithme, du cosinus et du sinus sont identifiées comme étant responsables de plus de 90% du temps de calcul de la FFT. D'une part, ces fonctions sont à éviter dans une problématique d'optimisation. D'autre part, ces calculs ne correspondent pas au cœur de calcul de la FFT.

Une des propriétés de la transformée de Fourier est que les valeurs des coefficients ne dépendent que de la taille de la transformée calculée. Ceci permet de déporter le calcul de coefficients par l'intermédiaire d'une fonction d'initialisation. C'est ainsi que toutes les bibliothèques optimisées de traitement de signal proposent en parallèle d'une fonction de calcul de FFT, une fonction d'initialisation chargée de calculer toutes les valeurs indépendantes des données.

Réduction de la complexité de calcul

A ce stade d'analyse, le cœur de calcul de notre algorithme FFT est de la forme suivante:

```
tmp1 = Wre*Xre[j] - Wim*Xim[j];
tmp2 = Wim*Xre[j] + Wre*Xim[j];
Xre[j] = Xre[i] - tmp1;
Xim[j] = Xim[i] - tmp2;
Xre[i] = Xre[i] + tmp1;
Xim[i] = Xim[i] + tmp2;
```

Figure 3.3.1: Cœur de calcul FFT

Après transformation du cœur de calcul suivant l'optimisation décrite par l'équation 19, le cœur de calcul de l'algorithme devient:

```
tmp1 = Xim[j] - Wre*Xre[j];
tmp2 = Wre*Xim[j] + Xre[j];
Xre[j] = Xre[i] + tmp1 * Wim;
Xim[j] = Xim[i] - tmp2 * Wim;
Xre[i] = Xre[i] - tmp1 * Wim;
Xim[i] = Xim[i] + tmp2 * Wim;
```

Figure 3.3.2: Cœur de calcul FFT optimisé

Remarquons que les coefficients impliqués dans les deux codes écrits ci-dessus ne sont pas identiques. Dans la solution optimisée, le coefficient réel correspond à l'ancien divisé par le coefficient imaginaire.

Par ailleurs, la division n'est possible que lorsque que le coefficient imaginaire est différent de zéro. Or lorsque ce dernier est nul, le coefficient réel associé est égal à l'unité. L'expression du papillon se réduit à une addition et une soustraction complexe.

Vectorisation du cœur de calcul

Vient maintenant ce que l'on appelle la *vectorisation* de l'algorithme, c'est à dire l'utilisation des unités SIMD du processeur au sein de l'algorithme de calcul.

Cette étape consiste à identifier, dans un premier temps, l'indépendance des calculs entre les différentes données des vecteurs puis dans un second temps, d'exploiter cette indépendance d'exécution pour mettre en évidence le parallélisme natif des unités SIMD.

Les processeurs importants disposent tous d'une unité vectorielle permettant de faire du calcul flottant simple-précision sur un total de 4 valeurs flottantes simultanément. L'enjeu est donc d'identifier au sein de l'algorithme étudié ce parallélisme d'ordre 4.

L'étude de l'algorithme radix-2 a permis de mettre en évidence ce parallélisme d'ordre 4. Nous avons vu qu'il était construit autour du calcul de 2 FFT de taille N/2 et indépendantes. Par récursivité, les deux FFT de taille N/2 sont à leur tour chacune calculée par l'intervention de 2 FFT de taille N/4.

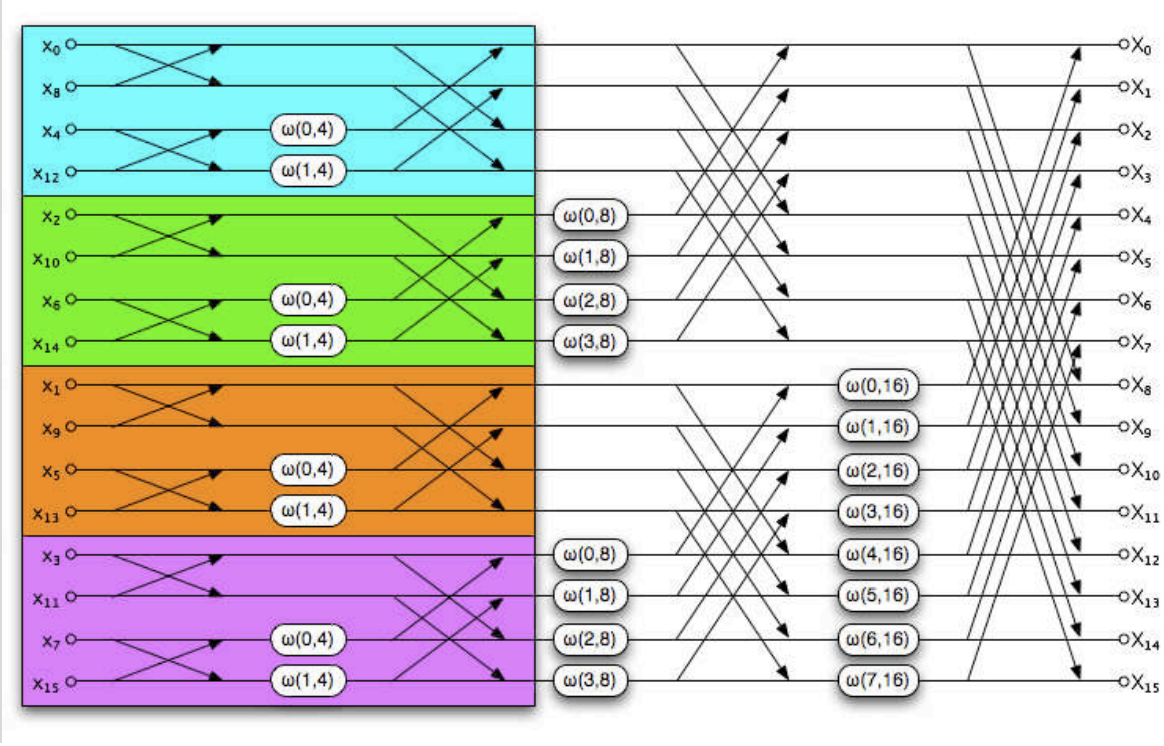

Figure 3.3.3: Quatre groupes de calcul indépendants et symétriques

Après observation des 4 FFT de taille N/4, il apparaît que chacune d'entre elles nécessite un groupe distinct. Ces derniers correspondent aux groupes d'échantillons d'indices 4n, 4n+1, 4n+2 et 4n+3.

De part l'entrelacement naturel des 4 groupes d'échantillons, le calcul simultané des 4 FFT identifiées avec des instructions de calcul SIMD ne nécessite aucune permutation de données. Les mêmes opérations sont alors appliquées aux quatre données contiguës. Le passage au papillon suivant se fait par translation des opérations sur le vecteur de données.

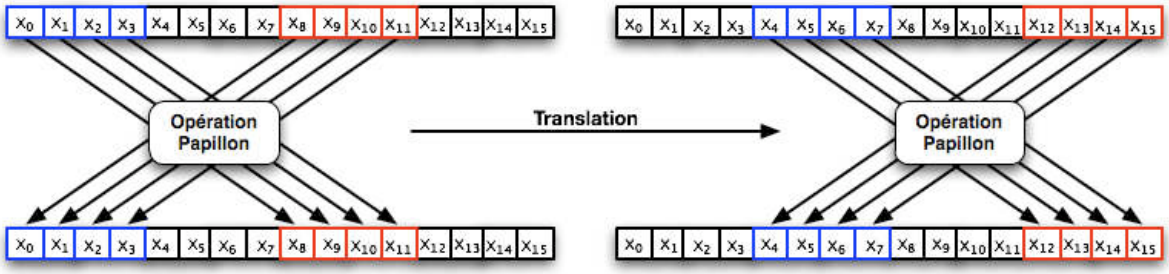

Figure 3.3.4: Vectorisation et translation de l'opération papillon

Une *vectorisation* efficace permet d'accélérer, d'un facteur 4, les performances d'un algorithme. Bien qu'il n'existe pas toujours de solution pour exploiter ce parallélisme de données dans tous les algorithmes, cette optimisation reste incontournable et montre d'excellents résultats dans l'implémentation de la FFT.

Seuls les deux derniers étages de calcul nécessitent un réordonnancement des données. En effet, les données, nécessaires aux papillons à ce stade du calcul de la FFT, sont contenues dans un seul et même vecteur. Ceci nécessite l'appel d'instructions de permutation afin rétablir l'ordonnancement souhaité.

Couplée à la factorisation en multiplications additions fusionnées présentée dans le *chapitre 3.5.1* (*équation 19*), la *vectorisation* permet même d'accélérer d'un facteur 8 les performances de l'algorithme. Le cœur de calcul vectorisé est de la forme suivante:

```
vAr = vec_ld(16*i,Xre);
vAi = vec_ld(16*i,Xim);
vBr = vec_ld(16*j,Xre);
vBi = vec_ld(16*j,Xim);
vtmp1 = vec_nmsub(vWr,vBr,vBi);
vtmp2 = vec_madd(vWr,vBi,vBr);
vBr = vec_madd(vWi,vtmp1,vAr);
vBi = vec_nmsub(vWi,vtmp2,vAi);
vAr = vec_nmsub(vWi,vtmp1,vAr);
vAi = vec_madd(vWi,vtmp2,vAi);
vec_st(vAr,16*i,Xre);
vec_st(vAi,16*i,Xim);
vec_st(vBr,16*j,Xre);
vec_st(vBi,16*j,Xim);
```

Figure 3.3.5: Cœur de calcul vectorisé

Masquage des opérations hors calcul

Afin d'optimiser notre algorithme FFT, nous avons cherché à diminuer la complexité de calcul de ce dernier. Cette complexité de calcul est directement liée aux opérations de calcul impliquées dans le traitement d'un papillon. Les performances souhaitées ne sont alors atteintes que si les opérations hors calcul sont traitées en parallèle, de sorte à être masquées derrière les opérations de calcul du papillon. Ces opérations hors calcul correspondent dans notre cas, aux opérations d'accès mémoire et d'incrément d'indice.

Une étude plus approfondie du nid de boucles montre que l'expression du cœur de calcul radix-2 est composée de 6 instructions de calcul et de 8 instructions d'accès mémoire. Même si un processeur superscalaire possédant un large parallélisme d'exécution, tel que le G5, est capable d'exécuter deux lectures mémoire simultanément, la majorité des processeurs montre un parallélisme moyen d'une instruction d'accès mémoire pour une instruction de calcul.

Notre algorithme n'est alors pas dimensionné par la complexité de calcul mais par le nombre d'accès mémoire à effectuer. Il faut donc réduire ce nombre d'accès mémoire par rapport aux calculs. Une solution est de ne pas écrire directement les résultats en mémoire après le calcul d'un papillon mais de les garder à disposition dans des registres afin de les réutiliser dans la suite des calculs. Ainsi après le calcul d'un papillon, il faut conserver les résultats et enchaîner sur le calcul du papillon suivant. Or dans le déroulement de l'algorithme FFT radix-2, le calcul d'un papillon dépend des résultats de deux papillons de l'étage de calcul précédent. Cette observation montre que au sein de deux étages consécutifs, les dépendances de calcul forment des groupes de 4 papillons.

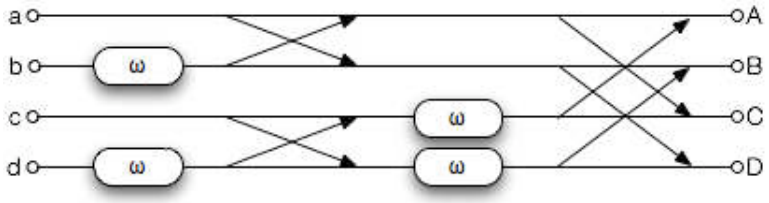

Figure 3.3.6: Groupement de quatre papillons dépendants

En traitant dans ce nouveau nid de boucle, non plus un seul papillon mais ce groupe de 4 papillons, le nombre d'instructions de calcul se trouve par conséquent multiplié par 4 et représente donc un total de 24 instructions. Concernant les accès mémoire, leur nombre n'augmente que d'un facteur 2 soit un total de 16 accès mémoire, lectures et écritures confondues.

Cette dernière optimisation assure que les performances de l'algorithme ne dépendent alors que de l'exécution des instructions de calcul. Connaissant le débit maximal de l'architecture PowerPC 970FX, une instruction de multiplication addition fusionnée par cycle d'horloge, il peut être envisagé d'atteinte les performances crêtes du processeur avec cet algorithme.

Respect des latences

L'étape précédente a permis de rendre faisable l'atteinte des performances crêtes du processeurs. Cependant pour ce faire, la cadence d'une instruction de multiplication addition fusionnée par cycle d'horloge doit être réalisée.

Afin d'atteindre cette cadence maximale, il est nécessaire que chaque latence de calcul entre deux opérations dépendantes soit respectée. Le traitement d'une opération suit diverses étapes dans le pipeline d'exécution. Le nombre de cycles entre la prise en charge de l'instruction par l'unité de calcul et la restitution du résultat correspond à la latence qui nous intéresse. Ces données sont disponibles au sein du manuel d'utilisateur du processeur [20]. Ainsi l'instruction *vec_ld* nécessite 6 cycles d'horloge à son exécution si les données sont en cache L1 et l'instruction vec_madd, quant à lui, 8 cycles d'horloge.

Une autre donnée est aussi primordiale, le *throughput* qui correspond au débit maximal d'exécution d'une instruction. Dans notre cas présent, toutes les instructions appelées ont un *throughput* d'un cycle soit un débit maximal d'une instruction par cycle d'horloge.

Le réordonnancement des instructions prend en compte deux facteurs, les latences d'exécution des instructions et le parallélisme d'exécution superscalaire. Son but est de masquer les cycles de latence entre deux instructions dépendantes en initiant d'autres opérations prêtes à être exécutées.

Prenons l'exemple d'un code non ordonnancé:

```
                                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
vAr = vec_ld(16*i,dataR);       S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vAi = vec_ld(16*i,dataI);       S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vBr = vec_ld(16*j,dataR);       .  S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vBi = vec_ld(16*j,dataI);       .  S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vtmp1 = vec_nmsub(vWr,vBr,vBi); .  .  S1 .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .  .  .  .  .  .  .
vtmp2 = vec_madd(vWr,vBi,vBr);  .  .  .  S1 .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .  .  .  .  .  .
vBr = vec_madd(vWi,vtmp1,vAr);  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .
vBi = vec_nmsub(vWi,vtmp2,vAi); .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .
vAr = vec_nmsub(vWi,vtmp1,vAr); .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .
vAi = vec_madd(vWi,vtmp2,vAi);  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .
vec_st(vAr,16*i,dataR);         .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  S2 S3 S4
vec_st(vAi,16*i,dataI);         .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  S2 S3
vec_st(vBr,16*j,dataR);         .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  S2 S3
vec_st(vBi,16*j,dataI);         .  .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  S2
```

Figure 3.3.7: Exécution d'un code non ordonnancé

Le code est constitué d'un total de 6 instructions de multiplication addition fusionnée et nécessite plus de 25 cycles pour être complètement exécuté. Ceci est dû aux bulles dans le pipeline qui correspondent aux cycles d'horloge perdus lors de l'attente d'une dépendance pas encore traitée.

Prenons maintenant le même code ordonnancé de manière à respecter au mieux les latences d'exécution.

```
                                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
vBr = vec_ld(16*j,dataR);       S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vBi = vec_ld(16*j,dataI);       S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vAr = vec_ld(16*i,dataR);       .  S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vAi = vec_ld(16*i,dataI);       .  S1 S2 S3 S4 S5 S6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
vtmp1 = vec_nmsub(vWr,vBr,vBi); .  .  S1 .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .  .  .  .  .  .  .  .
vtmp2 = vec_madd(vWr,vBi,vBr);  .  .  .  S1 .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .  .  .  .  .  .  .
vBr = vec_madd(vWi,vtmp1,vAr);  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .
vAr = vec_nmsub(vWi,vtmp1,vAr); .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .  .
vBi = vec_nmsub(vWi,vtmp2,vAi); .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .  .
vAi = vec_madd(vWi,vtmp2,vAi);  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  S2 S3 S4 S5 S6 S7 S8 .  .  .
vec_st(vBr,16*j,dataR);         .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6 .
vec_st(vAr,16*i,dataR);         .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  S2 S3 S4 S5 S6
vec_st(vBi,16*j,dataI);         .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  S2 S3 S4 S5
vec_st(vAi,16*i,dataI);         .  .  .  .  .  .  .  .  S1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  S2 S3 S4
```

Figure 3.3.8: Exécution d'un code ordonnancé

Certaines instructions ont pu être anticipées et gagner ainsi deux cycles d'horloge dans le traitement de code.

Déroulement de boucle

Le fait qu'il existe toujours des bulles dans le pipeline est dû au trop faible nombre d'opérations indépendantes permettant de masquer les cycles d'horloge inutilisés. Le déroulement de boucle permet non seulement de diminuer le nombre de sauts conditionnels dans le code mais aussi de multiplier les opérations indépendantes. Par contre, il augmente la taille du code généré.

Ainsi pour plus de liberté dans l'ordonnancement des instructions et si le nombre de registres disponibles le permet, il est recommandé de dérouler la boucle de calcul.

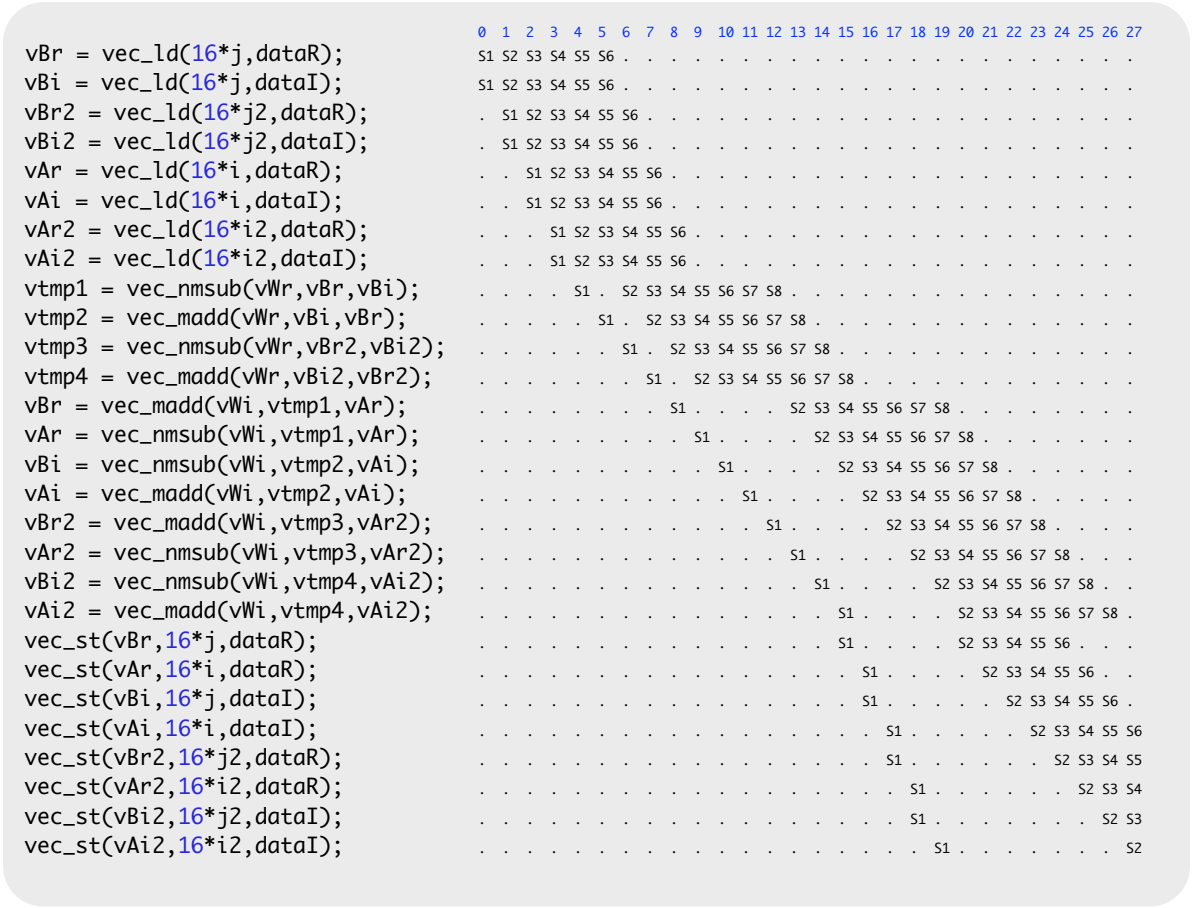

Figure 3.3.9: Déroulement de boucle

La figure 3.7.7, ci-dessus, met en évidence qu'un déroulement d'un facteur 2 d'une boucle de calcul ne demande qu'un très faible nombre de cycles de calcul supplémentaires pour l'exécution d'une itération, 4 dans le cas présent.

À noter qu'un déroulement de boucle ne se fait pas aisément suivant l'architecture étudiée. Dans le cas présent, celui du PowerPC 970FX, le processeur dispose d'un grand nombre de registres, 32 registres vectoriels, ce qui nous permet d'effectuer une telle opération. Ceci est valable pour le reste de la famille PowerPC et encore plus pour le CELL avec ses 128 registres adressables par SPE.

En revanche, les architectures Intel ne mettent à disposition qu'un faible nombre de registres, 8 en mode 32-bit et 16 en mode 64-bit. En contrepartie, leurs processeurs disposent d'un grand nombre de *rename registers*, registres internes servant à l'optimisation dynamique faite par le processeur lors de l'exécution du programme. Il est alors préconisé par Intel de ne pas dépasser ce nombre de registres. Les déroulements de boucle, quant à eux, sont toujours bénéfiques mais contrairement au PowerPC, il est recommandé de laisser les itérations de calcul disjointes [35].

Même si les marges de manoeuvre sont réduites, le déroulement de boucle permet toujours d'exposer le code à diverses optimisations telles que les lectures redondantes, l'élimination de sous-expression commune, etc.

Software Pipelining

Afin d'éviter toute rupture du pipeline entre deux itérations de boucle, une méthode existe, le Software Pipelining. Pour comprendre le pipeline logiciel, regardons une boucle générique exécutée sur un pipeline à trois étages: Lecture (L), Exécution (E) et Rangement (R).

Les itérations de la boucle sont représentées sur l'axe vertical et le temps sur l'axe horizontal. Nous supposons qu'une itération de la boucle consomme le même temps dans chacun des étages L, E et R de ce pipeline virtuel.

Maintenant, prenons un instantané du pipeline à l'instant 4 par exemple. Le pipeline effectue le rangement de l'itération 3, exécute l'itération 4 et procède à la lecture de l'itération 5.

Ainsi d'une manière générale, le Software Pipelining transforme la boucle de calcul de sorte qu'elle puisse, à chaque itération i, écrire les résultats de l'itération i-1, exécuter l'itération i et lire les données de l'itération i+1.

En contrepartie, cette méthode de programmation nécessite une phase d'initialisation et de finalisation appelé respectivement prologue et épilogue.

Le software pipelining permet ainsi d'éliminer les cycles d'horloge perdus lors du passage d'une itération de boucle à la suivante. L'anticipation des chargements permet d'éviter les blocages des instructions en attente des opérandes et les rangements tardifs tendent à minimiser les blocages dus à la non-disponibilité des résultats.

Cette optimisation est dite fine car elle fait référence à des notions de programmation de très bas niveau. Elle permet d'exploiter le maximum des capacités du processeur en saturant le *pipeline*. Cependant, d'une part c'est un travail manuel long. D'autre part, il n'assure pas de solution optimale. En effet, compte tenu du nombre élevé de solutions à étudier, il est difficile de considérer chacune des solutions existantes. De plus, le nombre de paramètres mis en jeu est élevé. Ces derniers vont du respect des latences à celui des dépendances entre les instructions, en passant par la prise en considération de l'exécution simultanée des instruction et la pression de registres.

Néanmoins, cette dernière opération a permis d'apporter jusqu'à 20% de gain de performance par rapport au code non ordonnancé. D'un point de vue global, l'optimisation de cet algorithme permet d'atteindre les 10GFlops sur PowerPC 970FX et de figurer parmi les meilleures FFT du marché, devançant même la librairie d'Apple jusqu'à présent leader sur le G5 [51]. Autre fait marquant, cette FFT a été jugée par des clients de Kontron Modular Computers SAS comme étant 10% plus rapide que leur librairie actuelle.

	64	256	1024	4096
FFT Kontron	8,1GFlops	9,8GFlops	9,8GFlops	8,2GFlops
VSIPro	7,9Gflops	8,1Gflops	8,9Gflops	7,6Gflops
vDSP Apple	7,5GFlops	9GFlops	9GFlops	7,7GFlops
FFTW	6,2GFlops	9,2GFlops	9,7GFlops	8GFlops

Tableau 3.3.1: Comparaison des performances FFT sur PPC970FX @ 2GHz

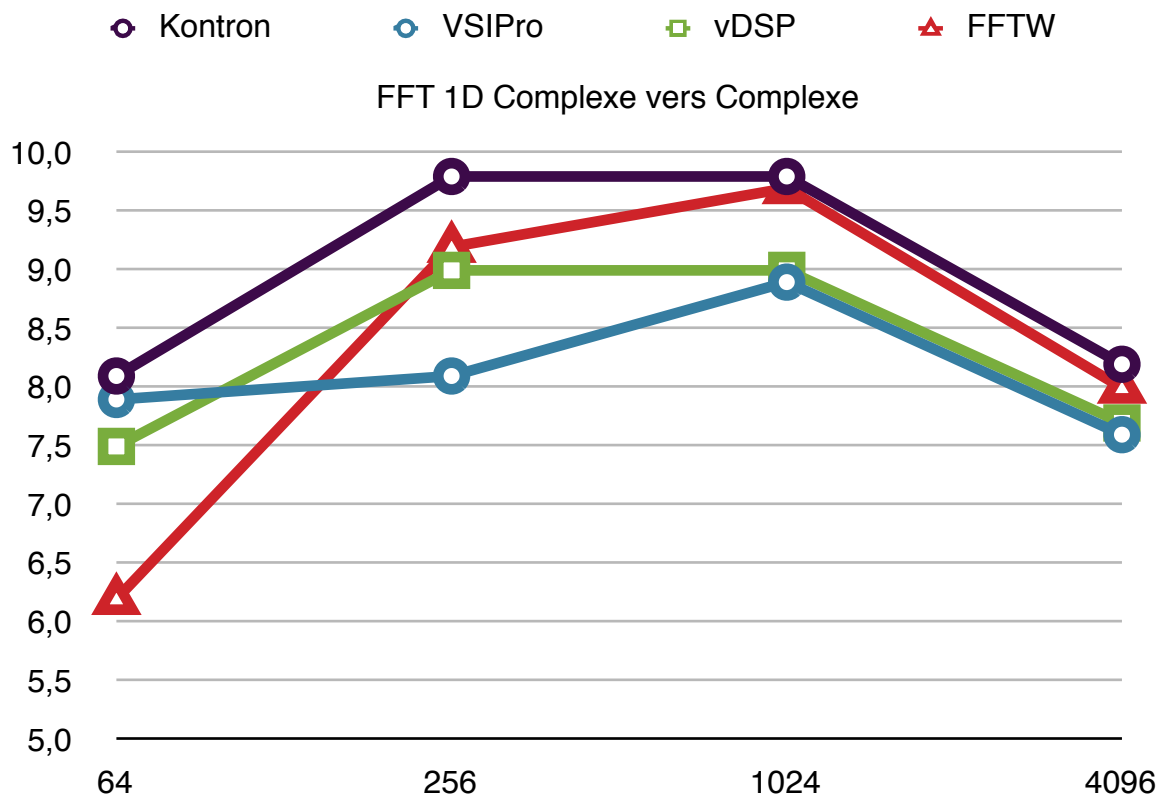

Figure 3.3.10: Comparatif avec les meilleures librairies FFT optimisées

Conclusions

L'implémentation de la transformée de Fourier rapide sur l'architecture PowerPC 970 FX est passée par diverses étapes. La première et la plus importante fut le choix de l'algorithme. Ceci a permis de mettre en évidence le fait qu'il n'existe pas d'optimisation générique. D'une part, les choix se sont faits en fonction de l'architecture ciblée. Une étude du fonctionnement interne du processeur est donc incontournable.

Dans notre cas, celui des architectures embarquées, l'accent a été mis sur le point d'entente entre tous ces processeurs, c'est à dire, les unités vectorielles. Leur efficacité n'est plus à démontrer mais nécessite une programmation spécifique et une modification de l'algorithme afin de prendre en compte ce parallélisme d'exécution.

Hormis les premières optimisations qui furent des règles classiques de bonne programmation, l'algorithme a été sujet à une série de choix d'optimisation et de transformations fonction du processeur cible.

A commencer par la *vectorisation* du code, le cœur de calcul fût écrit en langage AltiVec, jeu d'instructions SIMD du PowerPC 970. Ce langage rend, d'une part, le code incompatible avec les architectures hors PowerPC. D'autre part, il fait appel à des opérations uniques telles que la multiplication addition fusionnée.

Les opérations qui suivirent avait toutes un seul et unique but, atteindre le fonctionnement nominal du processeur, le rythme d'une instruction de calcul SIMD par cycle d'horloge.

Ainsi le regroupement de papillons, afin de supprimer les accès mémoire trop nombreux, n'a été possible que par analyse du code. Le déroulement de la boucle de calcul a permis de donner plus de liberté au re-ordonnancement des instructions.

Cette dernière étape est un jeu de permutations d'instructions visant à respecter toutes les latences de calcul tout en masquant les instructions hors calcul. Le retour sur expérience nous a permis de voir la difficulté d'une telle opération. Sa complexité augmente avec le nombre d'instructions impliquées. Les solutions sont aussi plus nombreuses. Enfin, cette optimisation dépend directement de l'architecture. Elle est donc à refaire pour toutes architectures susceptibles d'accueillir notre algorithme de transformée de Fourier rapide.

4. Générati` n de c` de ` ptimisé multi-architectures

L'optimisation de l'algorithme FFT radix-2 à entrelacement temporel a permis de mettre en avant les différentes étapes d'optimisation. Les différentes optimisations réalisées ont été faites en fonction de l'architecture considérée. Les principales différences entre le PowerPC 970 et les autres architectures telles que le PowerPC 7448, le CELL BE ou le Core 2 Duo résident dans le langage SIMD de l'architecture (AltiVec, SPU Intrinsic, SSE) et les latences d'exécution associées.

Ainsi, premièrement, les primitives fournies en langage C/C++ pour la programmation des unités vectorielles dépendent de l'architecture. En effet, il n'existe pas de solution de programmation SIMD commune entre les différentes architectures.

La programmation vectorielle n'est en aucun cas un standard de programmation mais une solution d'optimisation exploitée depuis les débuts des architectures parallèles. C'est ainsi que chaque fabricant de processeurs proposèrent leur solution intégrée. Chacun exploitèrent le vectoriel avec leur vision et leurs contraintes. Ce modèle d'exécution fut introduit dans le domaine grand public sous forme d'unités SIMD intégrées dans les processeurs généralistes.

Ces unités SIMD travaillent avec l'aide de registres dit vectoriels, capables de stocker plus d'une valeur. Elle dispose également de leur propre jeu d'instructions. Il existe ainsi un langage de programmation par jeu d'instructions.

Quelques solutions de portage existent, notamment pour traduire un code AltiVec en SSE. C'est le cas de SWAR [52] et plus récemment de NASoftware [53]. C'est derniers proposent des solutions de conversion entre langage AltiVec et langage SSE.

Le PowerPC a été pendant longtemps une référence dans le milieu embarqué. Depuis peu, Intel s'est ouvert au marché de l'embarqué. NASoftware propose ainsi un outil de conversion sous forme d'un fichier d'en-tête *altivec.h* à inclure dans le programme écrit en langage AltiVec mais à générer sur un architecture x86. Cette solution se limite à permettre la compilation d'un code AltiVec sur une architecture x86 par le biais d'une conversion du langage AltiVec en langage SSE.

Dernièrement, AMD souhaite affirmer sa vision du langage avec notamment le projet d'une nouvelle extension SSE5 incluant la multiplication addition fusionnée et la gestion d'instructions à plus de deux registres. D'autre part, le CELL marque l'apparition d'un nouveau langage de programmation pour ses coprocesseurs SPEs. La problématique de portage est donc une réalité à ce jour.

Enfin, le réordonnancement des instructions, ainsi que le software pipeline s'appuie sur la connaissance des latences d'exécution, le nombre d'unités d'exécution et d'autres caractéristiques intrinsèques au processeur étudié. Ce travail est donc à reproduire pour chaque portage effectué.

Ainsi, le portage d'un algorithme optimisé dispose de deux facettes, la traduction du langage et l'ordonnancement des instructions. Ceci nous mène au second thème de nos travaux, l'automatisation de la génération et de l'ordonnancement des instructions.

4.1. Les architectures pr` cesseurs embarqués

Au cours de notre étude sur les architectures processeurs embarqués, nous avons étudié un large choix de processeurs, permettant d'avoir une vue d'ensemble sur le fonctionnement interne de ces architectures.

Même si les architectures qui ont été présentées ci-dessus diffèrent en de nombreux points, leur fonctionnement reste néanmoins similaire et montre même une certaine convergence du point de vue dans leur conception. En effet, ce sont toutes des architectures superscalaires capables de traiter plusieurs instructions par cycle d'horloge dont les performances ne dépendent que d'un certain nombre de paramètres dont les latences d'exécution.

Ainsi, chacun des processeurs peut être décrit par:

- le nombre et la nature de ses unités d'exécution,
- ses capacités à lire, à décoder, à dispatcher et à exécuter plusieurs instructions simultanément,
- le nombre de registres disponibles,
- le jeu d'instructions,
- les latences d'exécution,
- etc.

Notre optimisation ne s'est basée que sur un certain nombre de paramètres, le nombre d'unité d'exécution, les règles de distribution d'instructions aux unités d'exécution, les latences d'exécution ainsi que le contenu du jeu d'instructions. Il est donc important à ce stade de proposer un modèle d'architecture superscalaire.

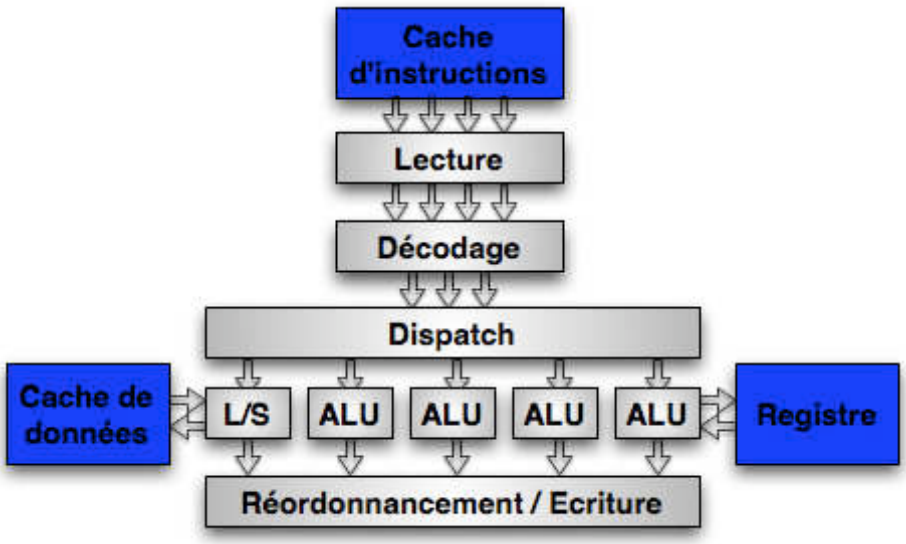

Figure 4.1.1: Modèle d'architecture supercalaire

Le principe fondamental des architectures superscalaires est de pouvoir exécuter plusieurs instructions du programme en parallèle. Le choix des instructions exécutées est réalisé dynamiquement, ce qui les différencie des architectures VLIW.

Une caractéristique fréquemment rencontrée dans les architectures processeurs modernes est que les instructions exécutées peuvent se terminer dans un ordre différent de celui du programme, ce qui permet au processeur de continuer l'exécution de la suite du programme même si une instruction bloque par l'indisponibilité d'une ou plusieurs ressources ou de ses opérandes. L'inconvénient majeur de cette innovation est que le comportement de ces processeurs devient beaucoup moins prédictif du point de vue du programmeur.

Nous allons émettre l'hypothèse que si le programme est déjà bien ordonnancé, l'exécution de ce dernier devrait se faire globalement dans l'ordre malgré l'exécution dans le désordre du processeur. Une telle hypothèse permet de mieux simuler le comportement du processeur et de pouvoir se fier à un modèle séquentiel de processeur plus déterministe aux yeux du programmeur.

Ainsi un processeur peut être décrit par le nombre maximal d'instructions par cycle qu'il est capable de distribuer aux unités d'exécution et par la nature et le nombre d'unités d'exécution, les latences des différentes opérations.

4.2. Le c` ncept

Devant le renouvellement continu des processeurs, il est important de pouvoir s'adapter rapidement à chaque nouvelle architecture. L'idée d'un générateur automatique de code optimisé vient de cette volonté d'adaptation à toute nouvelle architecture. Ce générateur s'appuiera sur un modèle d'architecture simple et suffisamment exhaustif, interprétera un langage unique et permettra de générer une ou plusieurs solutions à la fois optimisées et traduites dans le bon langage de programmation propre à l'architecture ciblée.

Ce générateur est donc bâti autour de trois facettes clés qui sont la modélisation des architectures processeurs, la définition d'un langage universel et l'optimisation de code. La modélisation fournit au générateur les informations nécessaires sur le fonctionnement du processeur. Le langage universel permet, quant à lui, de pouvoir utiliser un code unique et d'utiliser ce dernier sur plusieurs architectures. Enfin l'optimisation permet d'assurer une solution d'ordonnancement optimale.

4.3. Le langage universel

Le langage universel a pour but de faire abstraction de l'architecture sur laquelle le code source est généré. C'est, autrement dit, le rôle de tous les langages de programmation tel que le langage C/C++ ou le Fortran. Cependant, nous avons vu que chaque solution SIMD nécessite l'utilisation de son propre jeu d'instruction, limitant ainsi la *portabilité* du code développé dans un tel langage. Ce générateur n'a pas pour prétention de remplacer le rôle d'un compilateur tel que GCC qui est actuellement une des références du domaine de la compilation. Il a pour vocation de mettre l'accent sur ce manque d'aide à la programmation en langage SIMD. Ainsi, nous nous intéresserons plus particulièrement aux instructions de type SIMD et aux nids de boucles écrits en langage SIMD.

Si l'on considère les opérations effectuées au sein d'un nid de boucle comme celui du calcul de la transformée de Fourier, seul un certain nombre d'instructions sont impliquées. Parmi elles, figurent:

- les instructions de chargement et d'écriture indispensables pour l'accès aux données,
- les opérations arithmétiques utiles au calcul du papillon dont l'addition, la soustraction et spécialement la multiplication addition fusionnée,
- la permutation nécessaire à l'ordonnancement des données de la FFT,
- l'addition scalaire permettant d'incrémenter les indices afin de passer aux données suivantes.

Cette liste peut être considérée comme exhaustive dans le domaine du traitement du signal systématique vu que l'intégralité des fonctions constituant une librairie telle que la VSIPL, ne fait appel qu'à des algorithmes de type transformée ou filtre.

La librairie VSIPL est un standard libre utilisé pour des applications embarquées de traitement de signal ou d'image. Elle permet ainsi de développer des applications portables dans un langage proche du traitement du signal. L'optimisation de la transformée de Fourier vient de la volonté d'optimiser cette librairie constituée essentiellement de transformées et de filtres. De ce constat, seules ces opérations seront considérées dans notre langage haut niveau.

Concernant les jeux d'instructions étudiés et plus particulièrement les différents jeux d'instructions SIMD des différents processeurs dont l'AltiVec, le SSE et le SPU.

	AltiVec	SSE	SPU
load	`vec_ld`	`_mm_load_ps`	`si_lqx`
store	`vec_st`	`_mm_store_ps`	`si_stqx`
add	`vec_add`	`_mm_add_ps`	`si_fa`
sub	`vec_sub`	`_mm_sub_ps`	`si_fs`
multiply-add	`vec_madd`	X	`si_fma`
permut	`vec_perm`	`_mm_shuffle_ps`	`si_shufb`

Tableau 4.3.1: Correspondance d'instructions AltiVec, SSE et SPU

Première observation, la multiplication addition fusionnée est absente du jeu d'instructions SSE. Il conviendra alors de simuler cette opération avec les instructions SSE disponibles.

```
vD = vec_madd(vA, vB, vC);  <=>  vD = _mm_add_ps(_mm_mul_ps(vA, vB), vC);
```

Figure 4.3.1: Simulation de multiplication addition fusionnée en langage SSE

Cette simulation est exact au sens mathématique. Cependant, l'équivalence est fausse au sens informatique pour des calculs flottants simple précision. En effet, la multiplication addition fusionnée ne fait aucun arrondi entre la multiplication et l'addition. Ce n'est pas le cas pour l'opération imbriquée écrite en langage SSE.

Ensuite l'utilisation des opérandes des différentes instructions n'est pas systématiquement identique d'un jeu d'instructions à un autre. Ainsi les accès mémoire propres aux langages AltiVec et SPU permettent d'accéder à une adresse mémoire calculée par l'addition d'un pointeur à un offset de type entier, tandis que le SSE ne permet d'accéder qu'à l'adresse pointée par l'unique opérande d'adressage.

```
AltiVec:     vD = vec_ld(offset, ptr);        vec_st(vD, offset, ptr);
SPU:         vD = si_lqx(offset, ptr);        si_stqx(vD, offset, ptr);
SSE:         vD = _mm_load_ps(ptr + offset);  _mm_store_ps(ptr + offset, vD);
```

Figure 4.3.2: Détail des opérations d'accès mémoire AltiVec, SPU et SSE

Autre divergence, les jeux d'instructions AltiVec et SPU permettent d'écrire le résultat d'une opération dans un nouveau registre alors que, dans le cas du langage SSE, le registre résultat est systématiquement un des deux registres utilisés en tant qu'opérandes.

```
ADDPS xmm1, xmm2    <=>   vaddfp vA, vA, vB
```

Figure 4.3.3: Limitation du jeu d'instruction à deux registres

Ceci est contourné lors d'une écriture en langage C car le raisonnement ne se fait plus en fonction de registres mais de variables. Ainsi l'écriture à trois variables est tout à fait possible et fonctionnelle. Cependant il ne faudra pas oublier qu'une telle écriture impose à la compilation l'usage d'une opération de copie entre deux registres dans le cas du langage SSE.

Un problème subsiste avec l'instruction SSE de permutation puisque cette dernière diffère beaucoup de l'instruction AltiVec et SPU. Elle fait appel à une valeur immédiate plutôt qu'un registre vectoriel et entraîne une incompatibilité d'écriture entre le langage SSE et les langages AltiVec et SPU. En effet, non seulement elle se limite à des permutations de mots de 32-bit au sein de deux vecteurs, limitation sans importance dans le cas de données flottantes simple précision, mais elle ne permet pas de réaliser toutes les combinaisons de permutations possibles entre ces deux mêmes vecteurs. De ce fait, certaines permutations de type AltiVec ou SPU ne pourront être traduites en une seule et unique permutation SSE.

```
AltiVec:
vC = { 0,1,2,3, 16,17,18,19, 8,9,10,11, 24,25,26,27 };
vD = vec_perm(vA,vB,vC);

SSE:
#define vC 0x88
vD = _mm_shuffle_ps(vA, vB, vC);
vD = _mm_shuffle_ps(vD, vD, 0xD8);
```

Figure 4.3.4: Exemple de permutation impossible en langage SSE

Enfin, une dernière divergence de programmation concerne cette fois-ci la programmation d'un SPE. Ces co-processeurs disponibles dans l'architecture Cell d'IBM sont des processeurs strictement SIMD et ne font donc appel qu'à des opérations SIMD. De ce fait, l'usage des *offsets* et les calculs d'incréments associés se font aussi par l'intermédiaire d'opérations SIMD. Les indices et offsets utilisés ne seront plus déclarés en tant que nombre entier 32 bit mais en tant que *qword* représentant dans le cas de la programmation d'un SPE, un vecteur de type quad word soit 128 bits.

```
k = k + offset;    <=>    k = si_a(k, offset);
```

Figure 4.3.5: Traduction des opérations entières pour la programmation d'un SPE

Ainsi, toutes ces différences entre langages de programmation entraînent un remaniement du code source pour chaque architecture étudiée. Le but d'un modèle unique de programmation est donc de faire abstraction de l'architecture cible et de son langage SIMD associé.

Étant donné que le Cell est un cas particulier avec la programmation de ses co-processeurs vectoriels et que le langage SSE souffre d'un manque de fonctionnalités, le langage AltiVec est choisi comme modèle dans l'élaboration de ce langage universel.

Opération	Langage universel	AltiVec / C
Chargement Vectoriel	vload	vec_ld
Ecriture Vectorielle	vstore	vec_st
Addition Vectorielle	vadd	vec_add
Soustraction Vectorielle	vsub	vec_sub
Multiplication Addition Vectorielle	vmadd	vec_madd
Multiplication Soustraction Vectorielle	vnmsub	vec_nmsub
Permutation Vectorielle	vperm	vec_perm
Addition Scalaire	iadd	+

Tableau 4.3.2: Tableau de correspondance entre le langage universel et l'AltiVec

Langage AltiVec	Langage universel
`v1 = vec_ld(k, a);`	`vload v1 k a;`
`v2 = vec_ld(k, b);`	`vload v2 k b;`
`vr1 = vec_add(v1, v2);`	`vadd vr1 v1 v2;`
`v3 = vec_ld(k, c);`	`vload v3 k c;`
`v4 = vec_ld(k, d);`	`vload v4 k d;`
`vr2 = vec_add(v3, v4);`	`vadd vr2 v3 v4;`
`vr3 = vec_add(vr1, vr2);`	`vadd vr3 vr1 vr2;`
`vec_st(vr3, k, e);`	`vstore vr3 k e;`
`v5 = vec_ld(k2, a);`	`vload v5 k2 a;`
`v6 = vec_ld(k2, b);`	`vload v6 k2 b;`
`vr4 = vec_add(v5, v6);`	`vadd vr4 v5 v6;`
`v7 = vec_ld(k2, c);`	`vload v7 k2 c;`
`v8 = vec_ld(k2, d);`	`vload v8 k2 d;`
`vr5 = vec_add(v7, v8);`	`vadd vr5 v7 v8;`
`vr6 = vec_add(vr4, vr5);`	`vadd vr6 vr4 vr5;`
`vec_st(vr6, k2, e);`	`vstore vr6 k2 e;`
`k = k + offset;`	`iadd k k offset;`
`k2 = k2 + offset;`	`iadd k2 k2 offset;`

Figure 4.3.6: Exemple de programmation en langage universel

4.4. Le m` dèle d’exécuti` n pr` cesseur

Comme il a été présenté dans le chapitre 3.7 détaillant l’implémentation et l’optimisation d’un algorithme de transformée de Fourier rapide sur un processeur complexe tel que le PowerPC 970, une grande partie des performances d’un algorithme vient de son adaptation sur l’architecture cible en fonction des solutions d’optimisation mises à disposition par cette dernière.

En revanche, ce qui permet de gagner les 10 à 20% de performance et d’exploiter les ressources du processeur au plus près des 100% provient de cette méthode d’optimisation appelée “*pipelining logiciel*” consistant en une série d’optimisations dite “fines” car très proches de l’architecture et donc de très bas niveau. Il faut à la fois respecter les latences des instructions, gérer la pression de registres et faciliter l’exécution simultanée des instructions indépendantes, tout en respectant les dépendances d’exécution entre les opérations. C’est donc une étape longue et minutieuse, dépendante de l’architecture où il existe une infinité de solutions d’ordonnancement possible.

Dans le but d’assurer un ordonnancement automatisé au travers d’un générateur de code optimisé, il convient de définir une méthode de modélisation permettant de décrire le fonctionnement interne du processeur étudié à l’aide d’un certain nombre de paramètres.

L’ordonnancement manuel présenté lors de l’optimisation de la transformée de Fourier rapide a permis de montrer que cette optimisation repose sur la connaissance d’un nombre défini d’informations sur le fonctionnement du processeur. Ainsi, les performances nominales d’un processeur ne sont atteintes que si le programmeur a la connaissance :

- du nombre et de la nature des différentes unités d'exécution constituant le cœur de calcul
- du nombre maximal d'instructions par cycle pouvant être distribuées simultanément aux unités de calcul
- des latences d'exécution
- des différentes règles d'exécution ou de distribution

Ces informations permettent de synthétiser le fonctionnement du processeur et d'en prévoir les performances. Ainsi en croisant notre retour d'expérience sur l'optimisation de l'algorithme radix-2 sur PowerPC 970 avec notre étude sur les différentes architectures embarquées, nous déduisons le modèle d'exécution suivant.

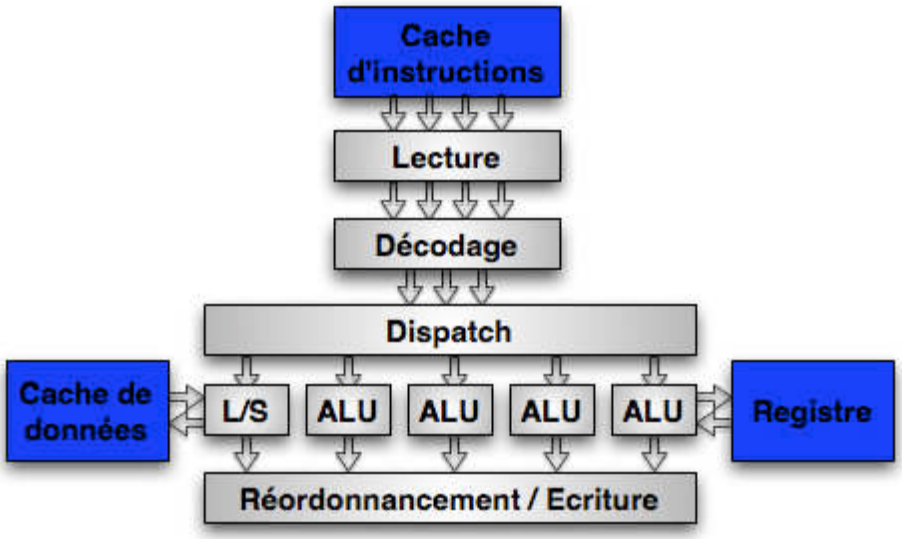

Figure 4.4.1: Modèle d'architecture supercalaire

Suivant cette modélisation, l'exemple ci-dessus décrit un processeur capable de lire simultanément, dans le cache L1 d'instructions, 4 instructions par cycle d'horloge. L'unité de décodage prend à son tour en charge les instructions lues au rythme de 4 par cycle d'horloge. Le processeur distribue ensuite 3 instructions par cycle aux unités d'exécution. Ces dernières sont au nombre de 5, une unité d'opération en mémoire et quatre unités arithmétiques et logiques.

En plus de ces informations, il est nécessaire de connaître les règles de distribution et d'exécution du processeur. En effet, il peut exister quelques restrictions dans le fonctionnement du processeur. Par exemple, dans le cas du PowerPC 970, au maximum deux instructions vectorielles peuvent être distribuées simultanément par cycle d'horloge si et seulement si l'une des deux est destinée à l'unité de permutation. Cette restriction n'existe pas dans le PowerPC 7448, génération précédente du PowerPC.

Afin de permettre au générateur d'être suffisamment modulaire et de pouvoir évoluer à chaque nouveau besoin, notre choix s'est porté vers un fichier de configuration que l'utilisateur pourra renseigner conformément au modèle suivant:

```
## Description Architecture

## Liste des unités
```

Lister les unités disponibles au sein du processeur.

```
## Dispatch et rules
## Règle 1:  Nombre d'instructions dispatchées par cycle processeur

MAX
```

Indiquer le nombre maximal d' instructions distribuées par cycle d' horloge.

```
## Règle 2: Attribution des slots pour chaque unité
```

Pour chaque unité, lister les numéros de ports de distribution autorisés (0 à MAX-1)
Une unité peut être attribuée à un ou plusieurs ports de distribution

```
## Règle 3: Exclusion mutuelle de ports d'instructions
```

Lister les incompatibilités de distribution deux à deux existantes entre les différents ports d' instructions

```
INSTRUCTIONS
## Liste des instructions et de leur format
## operation | format | nb instructions | instruction 1 | type (o=opera-
tion; f=fonction) | nb entrees | nb sorties | latence | instruction 2 |
type | nb .......
```

Pour chaque opération du langage modèle, indiquer la traduction en langage supportée par l' architecture

Figure 4.4.2: Patron de modélisation d'architecture

Cette modélisation du processeur simplifie le fonctionnement du *front end* (lecture, décodage, dispatch) en considérant seulement l'étape de distribution des instructions vers les unités d'exécution. L'hypothèse faite est celle que le processeur est conçu de sorte à permettre un tel débit. Toute limitation proviendrait d'un code mal optimisé.

Prenons comme exemple la modélisation du PowerPC 970. Le G5 dispose de 12 unités d'exécution au total. Un groupe d'au maximum 5 instructions est distribué à chaque cycle d'horloge. La formation de ce groupe est régie par un certain nombre de règles. L'une d'entre elles concerne la cinquième instruction du groupe qui ne peut être qu'une instruction de branchement. Une autre contrainte n'autorise qu'une seule des trois unités de calcul vectoriel à être alimentée par cycle d'horloge.

La traduction de ces informations dans le modèle fourni précédemment donne le fichier de configuration suivant:

```
## Description Architecture
## Liste des unités

VPU_PERM VPU_SI VPU_CI VPU_FP FPU_1 FPU_2 IU_1 IU_2 LSU_1 LSU_2 CRU BU

## Dispatch et rules
## Règle 1:   Nombre d'instructions dispatchées par cycle processeur

MAX  5

## Règle 2: Attribution des slots pour chaque unité

VPU_PERM      0 1 2 3
VPU_SI        0 1 2 3
VPU_CI        0 1 2 3
VPU_FP        0 1 2 3
FPU_1         0 3
FPU_2         1 2
IU_1          0 3
IU_2          1 2
LSU_1         0 3
LSU_2         1 2
CRU           0 1
BU            4

## Règle 3: Exclusion mutuelle de ports d'instructions

VPU_SI VPU_CI
VPU_SI VPU_FP
VPU_CI VPU_FP
IU_1 LSU_1
IU_2 LSU_2

INSTRUCTIONS
## Liste des instructions et de leur format
## operation | format | nb instructions | instruction 1 | type (o=operation;
f=fonction) | nb entrees | nb sorties | latence | instruction 2 | type | nb...

vload    0    vec_ld         f    2    1    6    LSU_1LSU_2
vstore   0    vec_st         f    3    0    1    LSU_1LSU_2
vadd     0    vec_add        f    2    1    8    VPU_FP
vsub     0    vec_sub        f    2    1    8    VPU_FP
vmadd    0    vec_madd       f    3    1    8    VPU_FP
vnmsub   0    vec_nmsub      f    3    1    8    VPU_FP
vperm    0    vec_perm       f    3    1    1    VPU_PERM
iadd     0    +              o    2    1    1    IU_1 IU_2
```

Figure 4.4.3: Modélisation de PowerPC 970 FX

Les douze unités constituant le G5 sont:

- l'unité de permutation vectorielle (VPU_PERM), l'unité vectorielle de calcul simple entier (VPU_SI), l'unité vectorielle de calcul complexe entier (VPU_CI) et l'unité vectorielle de calcul flottant (VPU_FP),
- les deux unités de calcul flottant (FPU_1 et FPU_2),
- les deux unités de calcul entier (IU_1 et IU_2),
- les deux unités d'accès mémoire (LSU_1 et LSU_2),
- l'unité de comparaison (CRU),
- et l'unité de branchement (BU).

A chaque cycle d'horloge, un groupe accueillant jusqu'à cinq instruction est distribué aux unités d'exécution. Chaque unité est attribuée à un ou plusieurs *slots* de ce groupe d'instructions. Par exemple, une instruction destinée à l'unité VPU_PERM peut être placée dans les quatre premiers slots. L'unité BU, quant à elle, dispose de son *slot* exclusif, le cinquième et dernier du groupe.

Au delà de ces restrictions, certaines règles peuvent intervenir. Les unités IU_1 et LSU_1 partageant la même file d'instruction, elles ne peuvent être alimentées simultanément. Ceci crée donc une incompatibilité entre elles traduites par l'exclusion mutuelle IU_1 LSU_1 de la modélisation.

Quant au renseignement de la liste d'instructions correspondant, le G5 n'est pas un exemple représentatif. En effet, le G5 ne fait intervenir qu'un seul cas de figure, celui où le nombre d'instructions est égal à 0. Ceci est dû à l'étroite similitude entre le langage AltiVec et le langage universel choisi. Retenons que l'équivalence entre le langage modèle et le langage lié à l'architecture se fait en indiquant pour chaque opération du modèle:

- le nombre d'instructions nécessaires à son calcul
- le modèle d'écriture si le nombre d'instructions est supérieur ou égal à 1
    - la première instruction à exécuter
        - le type de l'instruction, fonction ou opération
        - le nombre d'opérandes
        - la retour d'un résultat ou non
        - la latence de l'instruction
        - la liste des unités pouvant l'exécuter
    - la deuxième instruction si la première valeur est strictement supérieure à 1
        - ....
        - ....

Concernant la première information qui est le nombre d'instructions nécessaires au calcul de l'opération, ce besoin provient d'autres architectures telles que celles d'Intel où la traduction de notre langage *universel* peut nécessiter plusieurs instructions. Cette information est dans le cas du G5 toujours égale à 0. La valeur zéro permet, d'une part, la correspondance une à une entre l'opération du langage universelle et l'unique instruction associée. D'autre part, cette valeur permet d'indiquer que la correspondance des opérandes est strictement identique, ce qui permet de ne pas remplir le champs suivant, le modèle d'écriture indiquant l'ordre des opérandes et des instructions.

Le type de l'instruction permet de déterminer la syntaxe d'écriture qu'il faut adopter. Deux types sont utilisés, "f" pour fonction et "o" pour opération. Ainsi une instruction "op" de type fonction s'écrira de la manière suivante:

R = op(A, B)

tandis que la même instruction mais de type opération, s'écrira:

R = A op B

Ensuite, sont listés le nombre d'opérandes propre à l'instruction, le retour ou non d'un résultat, la latence d'exécution de l'instruction et l'appartenance à une unité d'exécution.

De la même manière que le PowerPC 970, la modélisation des co-processeurs du processeur CELL BE d'IBM se fait de la façon suivante:

```
## Description Architecture
## Liste des unités

PU FiPU FlPU LSU CU BU

## Dispatch et rules
## Règle 1:   Nombre d'instructions dispatchées par cycle processeur

MAX      2

## Règle 2: Attribution des slots pour chaque unité

PU       1
FiPU     1
FlPU     0
LSU      1
CU       1
BU       1

## Règle 3: Exclusion mutuelle de ports d'instructions

INSTRUCTIONS
## Liste des instructions et de leur format
## operation | format | nb instructions | instruction 1 | type (o=operation;
f=fonction) | nb entrees | nb sorties | latence | instruction 2 | type | nb...

vload    0    si_lqx          f    2    1    6    LSU
vstore   0    si_stqx         f    3    0    6    LSU
vmadd    0    si_fma          f    3    1    8    FlPU
vnmsub   0    si_fnms         f    3    1    8    FlPU
vadd     0    si_fa      f    2    1    8    FlPU
vsub     0    si_fs      f    2    1    8    FlPU
vperm    0    si_shufb        f    3    1    1    PU
iadd     0    si_a            f    2    1    8    FiPU
```

Figure 4.4.4: Modélisation du co-processeur SPE

Les SPUs sont des processeurs pouvant traiter jusqu'à 2 instructions simultanément par cycle à condition qu'une des instructions soit de type calcul flottant. C'est un processeur à architecture in-order qui se différencie du G5 de par sa nature SIMD ne lui permettant d'exécuter que des instructions vectorielles. Les calculs scalaires sont alors simulés au travers d'une opération vectorielle d'où le type fonction utilisé pour l'opération scalaire "iadd" qui fait référence à l'instruction vectorielle "si_a" où seul le premier mot du vecteur sera utilisé.

L'architecture Nehalem, dernière née d'Intel, se décrit, quant à elle de la façon suivante:

```
## Description Architecture
## Liste des unités

VALU0 VALU1 VSHUF VFADD VFMUL CIU SIU1 SIU2 BU ST LD

## Dispatch et rules
## Règle 1:   Nombre d'instructions dispatchées par cycle processeur

MAX  5

## Règle 2: Attribution des slots pour chaque unité

VALU0    0
VALU1    1
VSHUF    2
VFADD    1
VFMUL    0
CIU      0
SIU1     1
SIU2     2
BU       2
ST       3
LD       4

## Règle 3: Exclusion mutuelle de ports d'instructions

INSTRUCTIONS
## Liste des instructions et de leur format
## operation | format | nb instructions | instruction 1 | type (o=operation;
f=fonction) | nb entrees | nb sorties | latence | instruction 2 | type | nb ...

vload    2 ((b,a))      _mm_load_ps f 2 1 6 LD         + o 2 1 1 CIU SIU1 SIU2
vstore   2 ((c,b),a)    _mm_store_ps f 2 0 6 ST        + o 2 1 1 CIU SIU1 SIU2
vadd     0              _mm_add_ps f 2 1 8 VFADD
vsub     0              _mm_sub_ps f 2 8 VFADD
vmadd    2 ((a,b),c)    _mm_add_ps f 2 1 8 VFADD  _mm_mul_ps f 2 1 8 VFMUL
vnmsub   2 (c,(a,b))    _mm_sub_ps f 2 1 8 VFADD  _mm_mul_ps f 2 1 8 VFMUL
vperm    0              _mm_shuffle_ps f 3 1 1 VSHUF
iadd     0              + o 2 1 1 CIU SIU1 SIU2
```

Figure 4.4.5: Modélisation de l'architecture Core Nehalem

C'est ainsi qu'est décrit le processeur Nehalem. En réalité, ce sont six slots de dispatch qui sont mis à disposition mais deux d'entre eux sont utilisés simultanément par l'unité d'écriture en mémoire. Pour simplifier, un seul est retenu.

Revenons au détail de la liste d'instructions qui avait été laissée en suspens. Nous avons vu que lorsque le nombre d'instructions nécessaires à l'écriture de l'opération est égal à 0, ceci signifie d'une part que l'opération correspond à une seule et unique instruction et que, d'autre part, l'usage des opérandes est identique entre l'instruction du langage modèle et l'instruction correspondante. En revanche, si l'on considère le cas d'Intel, la multiplication addition fusionnée est absente du jeu d'instructions SSE. Il faut donc la simuler à l'aide de deux instructions élémentaires telles que l'addition et la multiplication.

```
vmadd   2 ((a,b),c)   _mm_add_ps f 2 1 8 VFADD    _mm_mul_ps f 2 1 8 VFMUL
```

Figure 4.4.6: Traduction de la multiplication addition fusionnée en langage SSE

Cette syntaxe permet d'indiquer lors de la modélisation de l'architecture que l'opération vmadd nécessite à sa traduction en langage SSE un total de "2" instructions. La génération de l'opération correspondante se fait alors par lecture des instructions de gauche à droite tout en se référant au modèle d'écriture ((a,b),c). Ainsi vmadd vR vA vB vC équivaut en langage SSE à:

```
vR = _mm_add_ps(_mm_mul_ps(vA, vB), vC)
```

Figure 4.4.7: Opération SSE générée suite à la modélisation

Certaines opérations nécessitent de changer l'ordre des opérandes afin de pouvoir restituer le bon résultat lors de sa traduction dans le langage approprié. C'est le cas de la multiplication soustraction fusionnée qui dans le cas de l'AltiVec inverse le signe du résultat par négation. Ainsi l'opération vnmsub, *-(ax-y)* peut se traduire en langage SSE de la façon suivante:

```
vR = _mm_sub_ps(vC, _mm_mul_ps(vA, vB))
```

Figure 4.4.8: Instruction nécessitant un ordre d'opérandes différent du modèle

d'où le modèle d'écriture suivant:

```
vnmsub  2 (c,(a,b))   _mm_sub_ps f 2 1 8 VFADD    _mm_mul_ps f 2 1 8 VFMUL
```

Figure 4.4.9: Modélisation d'une opération nécessitant un ordonnancement des opérandes

Le but de cette modélisation est d'une part de fournir les informations nécessaires au générateur pour traduire le code dans le langage propre à l'architecture ciblée; d'autre part de pouvoir reproduire le fonctionnement du processeur afin de simuler l'exécution du code généré au cycle prés. Le format "*fichier de configuration*" permet de pouvoir éditer et modéliser à volonté les architectures processeurs.

4.5. L'évaluation

Le critère de performance choisi est la minimisation du temps d'exécution du code généré. La méthode d'évaluation la plus évidente serait donc d'exécuter les solutions une à une et de mesurer leur performance. Cependant, c'est une méthode qui nécessite d'une part de devoir générer nativement le code sur l'architecture et d'autre part qui implique un temps d'évaluation conséquent, voire trop long face au nombre de solutions d'ordonnancement à étudier.

Nous préférons donc estimer les performances d'une solution générée plutôt que d'en effectuer les mesures réelles. L'estimation est faite par l'intermédiaire d'un simulateur d'exécution qui reproduira le comportement du processeur. Ce même simulateur s'appuie sur les informations fournies par la modélisation de l'architecture choisie.

Le rôle du simulateur est de prendre en entrée une solution d'ordonnancement du code initial, puis de simuler son exécution en fonction de la modélisation de l'architecture et de retourner le nombre de cycles d'horloge nécessaires. Un peu à la manière de simg4, simg5 [59] ou encore spu_timing [60], le simulateur reproduit l'exécution du code dans le pipeline du processeur.

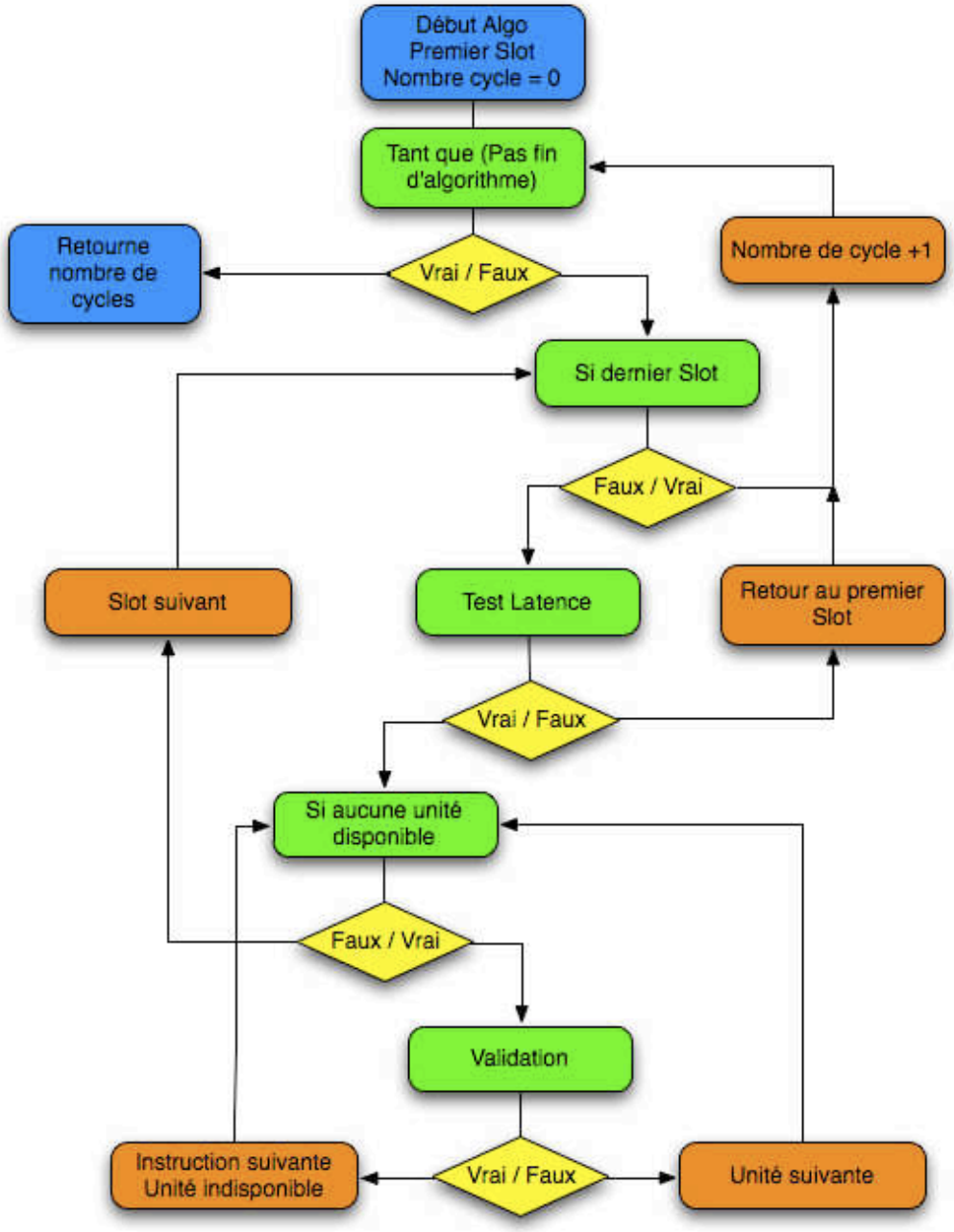

Figure 4.5.1: Diagramme fonctionnel du simulateur d'exécution

Le simulateur est une machine séquentielle qui traite le code d'entrée suivant l'ordre naturel de lecture, instruction par instruction. Pour chaque instruction, le simulateur teste:

- la disponibilité du port de distribution,
- l'accessibilité des unités impliquées, la première testée et disponible étant choisie par défaut,
- le respect des latences qui autorisera ou non la distribution de l'instruction vers la ou les unités de calcul correspondantes,
- la validation de la combinaison d'instructions distribuées au cours du cycle d'horloge.

Si au cours de l'étude d'une instruction, plus aucun port n'est susceptible de prendre en charge l'instruction ou si les latences d'exécution ne sont pas respectées, l'instruction est bloquée, le groupe d'instructions validées est distribué, le compteur de cycles incrémenté et l'étude est reprise avec un groupe vierge et les latences en cours décrémentées.

La validation des combinaisons de distribution d'instructions se fait par l'intermédiaire d'une matrice de conformité construite à la lecture du modèle d'exécution de l'architecture étudiée en fonction de sa configuration de ports de distribution et à des contraintes de distribution. Ainsi par simple lecture de la matrice, la faisabilité d'une combinaison de distributions simultanées entre différentes unités d'exécution peut être connue à tout moment.

4.6. L'optimisation

Dans notre problématique d'optimisation, l'accent est particulièrement mis sur les nids de boucles constituant un algorithme. Ce sont en effet ces boucles de calcul qui mettent les processeurs à contribution pendant la quasi totalité du temps d'exécution de l'algorithme. Ainsi un gain de performance acquis sur l'exécution d'une itération de la boucle de calcul se répercute immédiatement et proportionnellement au nombre d'itérations sur le temps de calcul global de l'algorithme.

La *vectorisation* et le déroulement de boucle faisant partie du design de l'algorithme et partie intégrante de la réflexion du programmeur, l'optimisation en question est un problème d'ordonnancement où chaque opération doit être faite au plus tôt lorsque tous ses opérandes sont disponibles. Il faut donc pouvoir respecter les latences et le parallélisme d'exécution de façon à exploiter au maximum les performances du processeur.

Or à chaque étape de l'ordonnancement, le programmeur se retrouve confronté à un choix. En effet, plusieurs possibilités peuvent s'avérer équivalentes à un temps t de l'exécution de l'algorithme. En revanche, un certain nombre d'entre elles mèneront vers des solutions non optimales. Cette prise de décision est arbitraire et les conséquences difficiles à prévoir.

Afin de retenir seulement la ou les solutions optimales, le générateur est capable de parcourir toutes les solutions possibles à notre problème d'ordonnancement. Bien que notre problème soit NP-complet, un code composé de n instructions engendrera jusqu'à n! combinaisons possibles d'instructions et donc autant de solutions à étudier, bon nombre d'entre elles ne seront pas valides car elles ne respectent pas les dépendances de calcul entre les différentes instructions constituant l'algorithme. Le générateur veille alors à n'étudier que les solutions respectant la cohérence de calcul de l'algorithme.

À partir d'une trentaine d'instructions, le temps de génération devient long et peut se compter en jours. Afin de converger plus rapidement vers des solutions optimisées, le générateur permet de rechercher la ou les solutions localement optimales. Ainsi par l'intermédiaire d'une fenêtre d'instructions, il est possible de rechercher la solution optimale localement en ne permutant que les instructions contenues au sein de la fenêtre. Afin de parcourir l'intégralité de l'algorithme, une translation de la fenêtre est faite jusqu'à atteindre les dernières instructions.

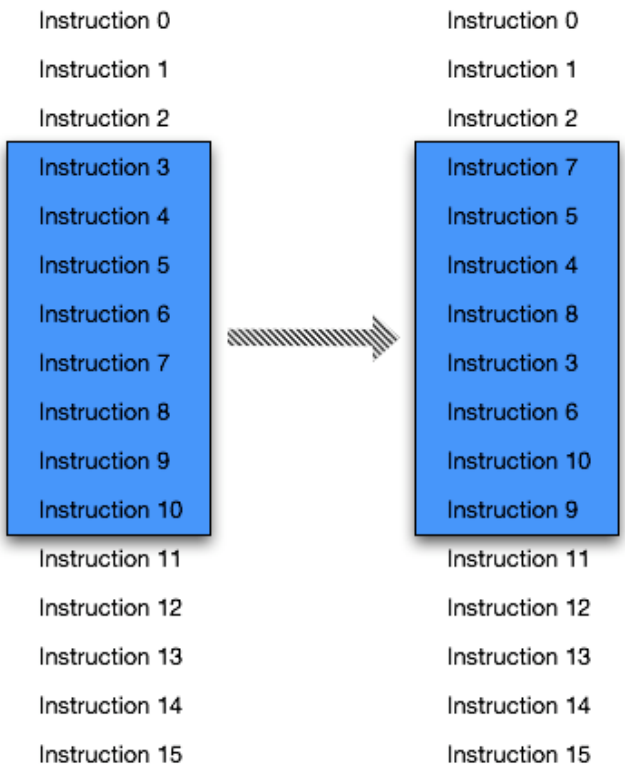

Figure 4.6.1: Exemple d'application de fenêtre de taille 8 sur un code à 16 instructions

Dans le but de devenir plus modulaire et plus efficace en toutes circonstances, le générateur dispose de plusieurs options de réglage telles que:

- la gestion de la taille de la fenêtre d'instructions,
- la gestion du pas de translation de la fenêtre d'instructions,
- la gestion du nombre d'itérations d'optimisation du code,
- la gestion du nombre de solutions retenues à chaque étape d'optimisation.

La taille de la fenêtre et le pas de translation sont des paramètres modifiables permettant à l'utilisateur d'influer sur la rapidité de la génération ou alors sur la qualité des résultats.

Le choix de la taille de la fenêtre influe directement sur la qualité d'optimisation et le temps de génération. Plus cette taille sera grande, plus les résultats seront proches de la solution optimale mais en contrepartie, le temps de génération augmentera de façon exponentielle.

Le pas de translation est un paramètre plus fin. Il permet de réduire la discontinuité lors du passage d'une fenêtre à la suivante. Ainsi des fenêtres qui se recouvrent entraîneront de meilleurs résultats tout en se rapprochant de la solution optimale.

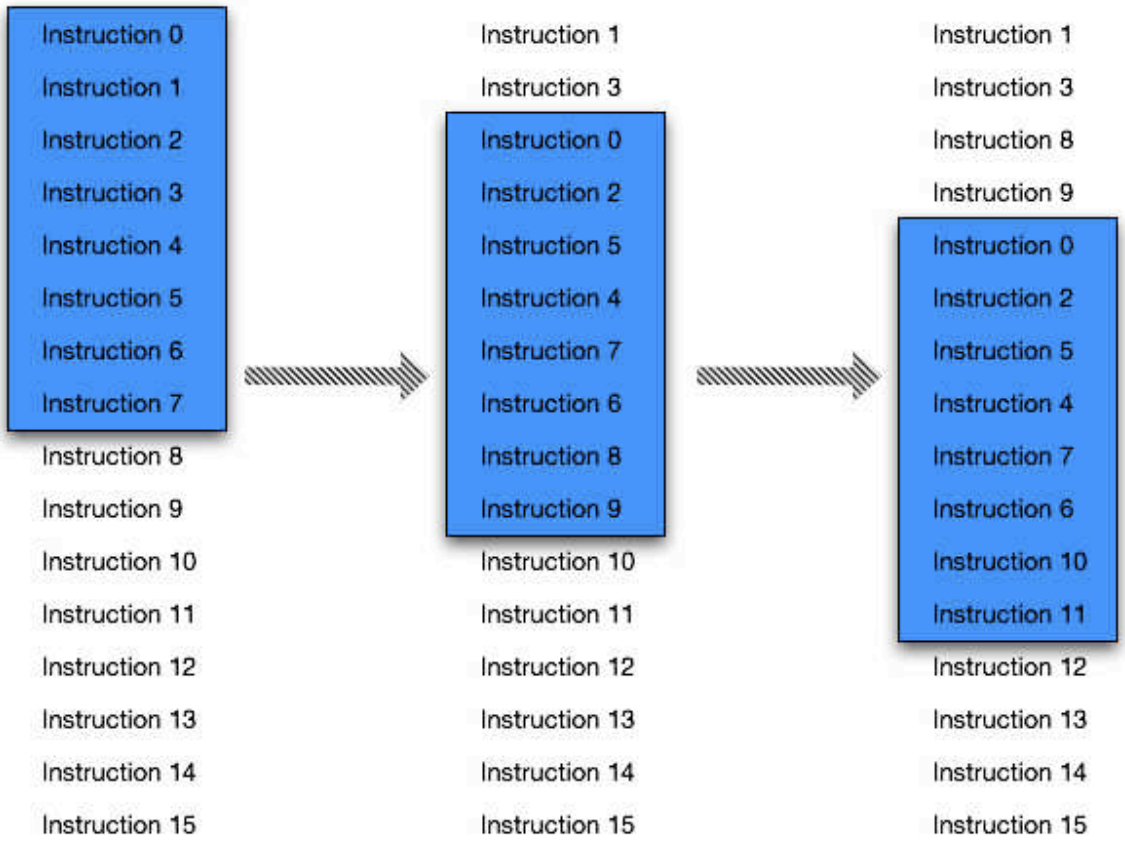

Figure 4.6.2: Recouvrement et translation des fenêtres

Le recouvrement des fenêtres permet de remonter, plus en amont dans l'exécution du code, les instructions initialement en fin d'algorithme et sans dépendances avec les instructions précédentes. Prenons l'exemple de la figure 4.6.2, la fenêtre d'instructions de taille 8 effectue une translation de 2 instructions par itération de recherche de l'optimum. On observe que les instructions 8 et 9, d'abord invisibles à la première itération puis visibles à la seconde, peut être anticipées bien en amont de l'algorithme. Ceci permet, d'autre part, de masquer les bulles dans le *pipeline* d'exécution causées par les latences de deux opérations dépendantes. Il est possible aussi d'effectuer plusieurs itérations consécutives d'optimisation sur le même code.

Le générateur donne la possibilité de retenir les *n* meilleures solutions. Lors de l'optimisation du code par glissement d'une fenêtre d'instructions, à chaque nouvelle étape, le générateur prend en considération les *n* solutions retenues comme nouveau point de départ. À partir d'un code initial unique, dès l'application de la première fenêtre d'instructions, le générateur traite alors ensuite une par une les *n* solutions retenues suite à l'application de la première fenêtre d'instructions. En effet, au cours de cette recherche de la solution optimale, un certain nombre de solutions sera équivalent au niveau performance alors que toutes ne mèneront pas vers la solution optimale. Notre cas d'optimisation est un problème de minimisation du temps d'exécution de l'algorithme.

Le risque est de rester coincé autour d'un minimum local. Le fait de retenir plusieurs solutions, même moins bonnes que la meilleure obtenue, permet d'augmenter les chances de sortir de ces minimums locaux et de converger plus facilement vers la solution optimale.

4.7. Application du générateur

4.7.1. Capacité d'optimisation du générateur

Nous disposons de résultats concrets de la transformée de Fourier rapide optimisée manuellement pour les architectures PowerPC 970 dont les résultats sont récapitulés ci-dessous:

Taille FFT	64	256	1024	4096
Temps de calcul de la FFT non ordonnancée	341 ns	1498 ns	7222 ns	39424 ns
Temps de calcul de la FFT optimisée	297 ns	1318 ns	6570 ns	37324 ns
Gain	12,90%	12%	9%	5,30%

Tableau 4.7.1: Performances FFT optimisée sur PowerPC 970 FX @ 1,6GHz

Les premiers résultats proviennent de notre première implémentation de l'algorithme FFT radix-2 à entrelacement temporel sur PowerPC 970 FX. Ce sont des résultats mesurés. On observe un gain de performances allant jusqu'à 12% entre l'implémentation naïve et la même ordonnancée manuellement.

Ce travail d'optimisation nous a coûté plusieurs semaines de développement. L'intérêt du générateur est donc d'automatiser cette étape d'optimisation fine. L'application du générateur sur le même code non ordonnancé montre les résultats suivants.

Taille FFT	64	256	1024	4096
Temps de calcul de la FFT non ordonnancée	341 ns	1498 ns	7222 ns	39424 ns
Temps de calcul de la FFT générée	307 ns	1324 ns	6465 ns	36774 ns
Gain	10%	11,60%	10,50%	6,70%

Tableau 4.7.2: Performances FFT générée sur PowerPC 970 FX @ 1,6GHz

En récapitulatif:

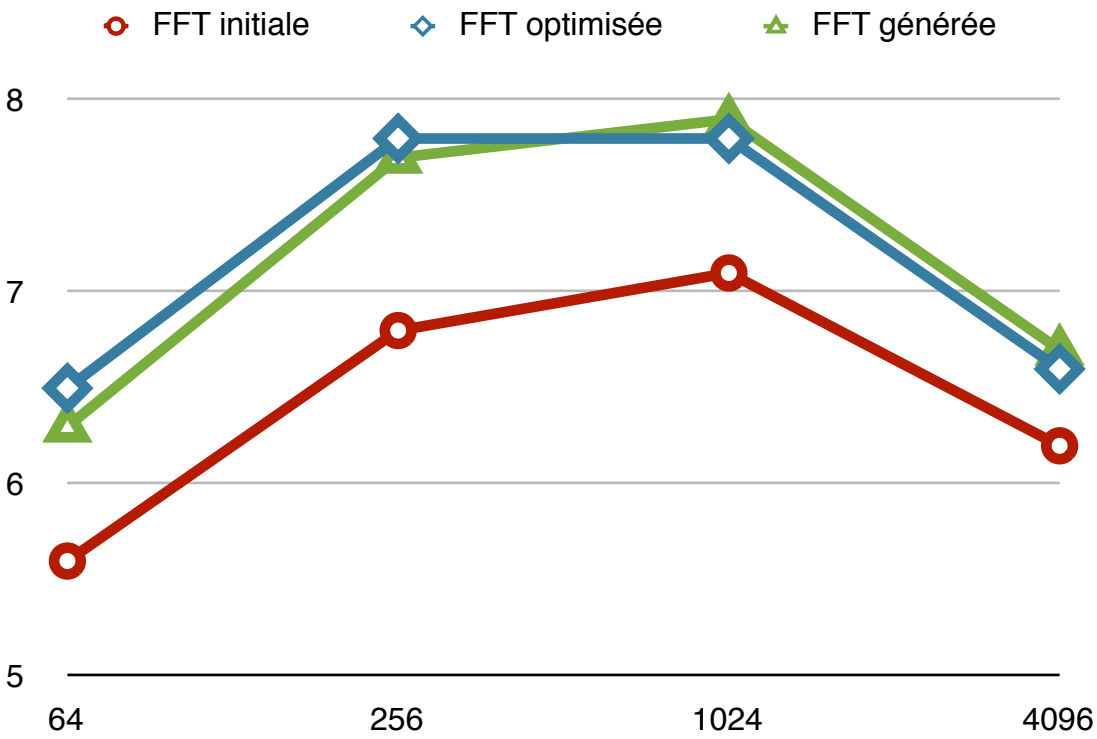

Figure 4.7.1: Comparaison entre optimisation manuelle et optimisation automatisée

L'algorithme FFT étudié est constitué de 3 nids de boucle différents et d'un total de 5 boucles de calcul indépendantes. Le premier nid de boucle traite le cas des papillons triviaux et n'implique que des opérations d'addition et de soustraction, le second est le cas général faisant intervenir les multiplications additions fusionnées, quant au troisième et dernier, il a pour rôle de restituer l'ordre naturel des données.

Si l'on étudie l'impact de l'optimisation sur chaque boucle isolée du reste de l'algorithme, il en résulte les informations suivantes:

	Code initial	Optimisation manuelle		Optimisation automatisée	
64	74 ns	77 ns	- 4%	72 ns	2,70%
256	239 ns	121 ns	11,30%	233 ns	2,50%
1024	873 ns	714 ns	18,20%	841 ns	3,70%
4096	3424 ns	2777 ns	18,90%	3286 ns	4%

Tableau 4.7.3: Gain apporté à l'optimisation du nid de boucle "papillon triviaux"

On peut voir que le générateur a eu très peu d'impact, montrant un gain de performances assez faible comparé à l'optimisation manuelle.

	Code initial	Optimisation manuelle		Optimisation automatisée	
64	106 ns	103 ns	2,80%	102 ns	3,80%
256	365 ns	371 ns	-1,60%	348 ns	4,70%
1024	1382 ns	1394 ns	-0,90%	1303 ns	5,70%
4096	7480 ns	7799 ns	-4,30%	7418 ns	0,80%

Tableau 4.7.4: Gain apporté à l'optimisation du nid de boucle "général"

Sur cette nouvelle boucle de calcul, le générateur a su optimiser le code alors que l'optimisation manuelle a eu l'effet inverse en dégradant les performances.

	Code initial	Optimisation manuelle		Optimisation automatisée	
64	174 ns	148 ns	14,90%	149 ns	14,40%
256	580 ns	473 ns	18,40%	476 ns	17,90%
1024	2187 ns	1744 ns	20,30%	1763 ns	19,40%
4096	9822 ns	8329 ns	15,20%	9090 ns	7,50%

Tableau 4.7.5: Gain apporté à l'optimisation du troisième nid de boucle "recombinaison"

Enfin les résultats sur le dernier nid de boucle mélangeant calcul et permutation, les résultats coïncident entre optimisation manuelle et automatisée.

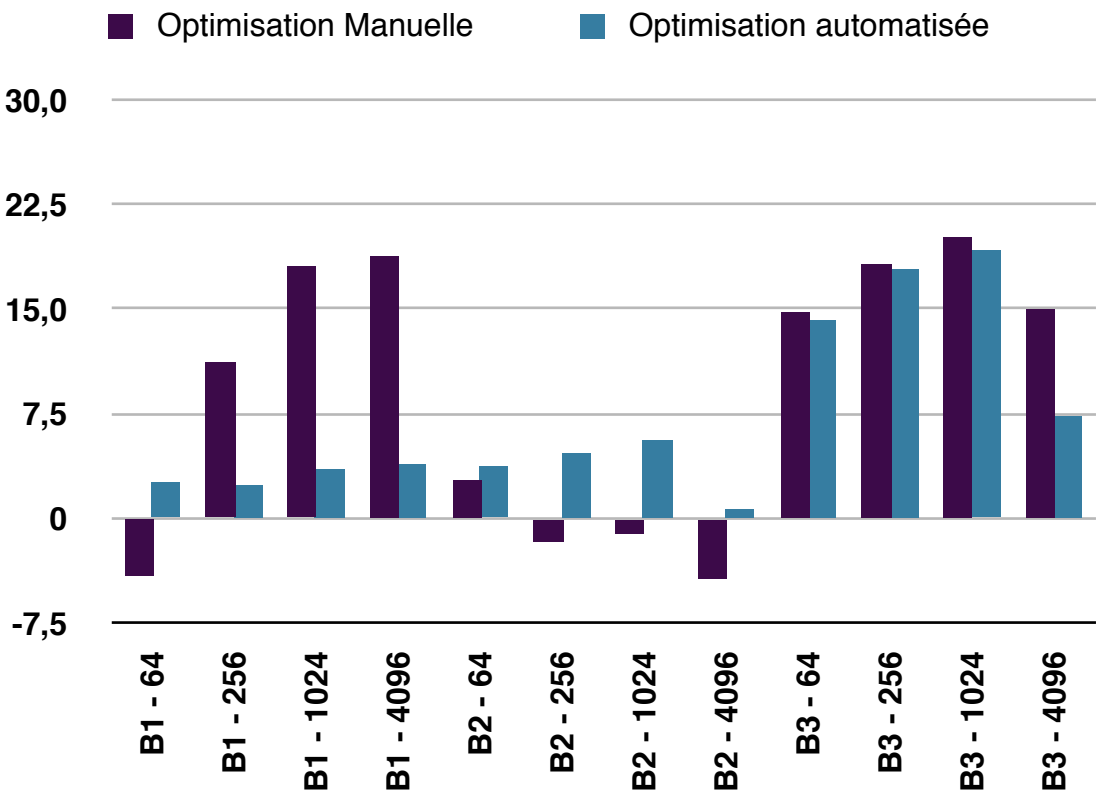

Figure 4.7.2: Comparaison entre optimisation manuelle et optimisation automatisée

La figure 4.7.2 illustre l'étude au cas par cas des trois nids de boucle constituant l'algorithme FFT radix-2 à entrelacement temporel. Le détail des résultats montre que le générateur a optimisé d'environ 5% les performances du deuxième nid de boucle. À l'inverse, l'optimisation manuelle a, quant à elle, dégradé les performances de l'algorithme.

L'optimisation du troisième nid de boucle est elle aussi positive. Les résultats obtenus par le générateur sont du même ordre de grandeur que ceux obtenus par optimisation manuelle.

En revanche, seul le cas du premier nid de boucle ternit les performances du générateur. Cependant, la comparaison est faussée. Ce nid de boucle est confondu avec le calcul du bit reverse dans l'implémentation de l'algorithme. C'est ce calcul qui a été ordonnancé lors de l'optimisation manuelle, contrairement à la génération automatique où seul le calcul des papillons triviaux a été pris en compte. L'écart de performances entre les deux méthodes correspond ainsi au masquage du bit-reverse faite par l'optimisation manuelle.

Intéressons-nous maintenant au cas de la transformée de Fourier rapide de Stockham présentée précédemment. Aucune comparaison ne peut être faite avec une optimisation manuelle. Cette dernière n'a pas été faite.

Taille FFT	64	256	1024	4096
Temps de calcul de la FFT non ordonnancée	591 ns	1805 ns	11087 ns	52325 ns
Temps de calcul de la FFT générée	571 ns	1670 ns	10539 ns	45259 ns
Gain	3,40%	7,50%	4,90%	15,60%

Tableau 4.7.6: Optimisation de l'algorithme FFT de Stockham sur PPC970FX @ 1,6GHz

Les résultats obtenus sont du même ordre de grandeur que ceux obtenus lors de l'optimisation de l'algorithme radix-2 à entrelacement temporel. Ceci valide la stabilité des résultats d'optimisation du générateur.

La variation des gains en fonction de la taille FFT que l'on peut observer sur le tableau précédent est du à l'écriture de l'algorithme de Stockham. Cet algorithme alterne lecture des données et écriture des résultats entre deux tableaux distincts. Ainsi suivant les tailles de FFT choisies, l'obtention des résultats nécessite une copie de tableau, ce qui dégrade les performances de l'algorithme.

4.7.2. Génération multi-architecture

L'algorithme de Stockham a été généré avec l'aide du générateur sur 3 architectures différentes:

- Le PowerPC 970 FX d'IBM
- Le Cell SPE d'IBM
- Le Core 2 Duo d'Intel

Concernant la génération de code sur l'architecture ciblée, cette dernière est entièrement fonctionnelle pour le PowerPC et le Cell. Quant à Intel, un problème subsiste, le cas de l'instruction de permutation SSE qui, suivant la configuration de permutation impliquée, peut ne pas être compatible. Une intervention manuelle est alors obligatoire après génération.

Les performances de l'algorithme sur les différentes architectures sont:

Taille FFT	64	256	1024	4096	Max Théorique
PPC970FX 1,6GHz	3,4Gflops	6,1Gflops	4,9Gflops	5,4Gflops	12,8Gflops
Cell SPE 3,2GHz	5,3Gflops	9,3Gflops	10,9Gflops	11,6Gflops	26,5Gflops
Core2Duo 2GHz	2,9Gflops	3,9Gflops	3,8Gflops	4,0Gflops	16Gflops

Tableau 4.7.7: Performances de l'algorithme FFT de Stockham

Comparé aux autres architectures, l'algorithme généré sur l'architecture Intel montre des performances très faibles. L'architecture x86 ne possède en effet qu'un total de 8 registres en mode 32-bit et de 16 en mode 64-bit. Les déroulements répétitifs effectués lors de l'optimisation de l'algorithme sur l'architecture PowerPC sont néfastes sur les architectures Intel.

À l'inverse, la génération de l'algorithme sur le Cell montre de bonnes performances qui, comparées aux différentes librairies étudiées par l'équipe FFTW [51], culminent parmi les meilleures actuelles.

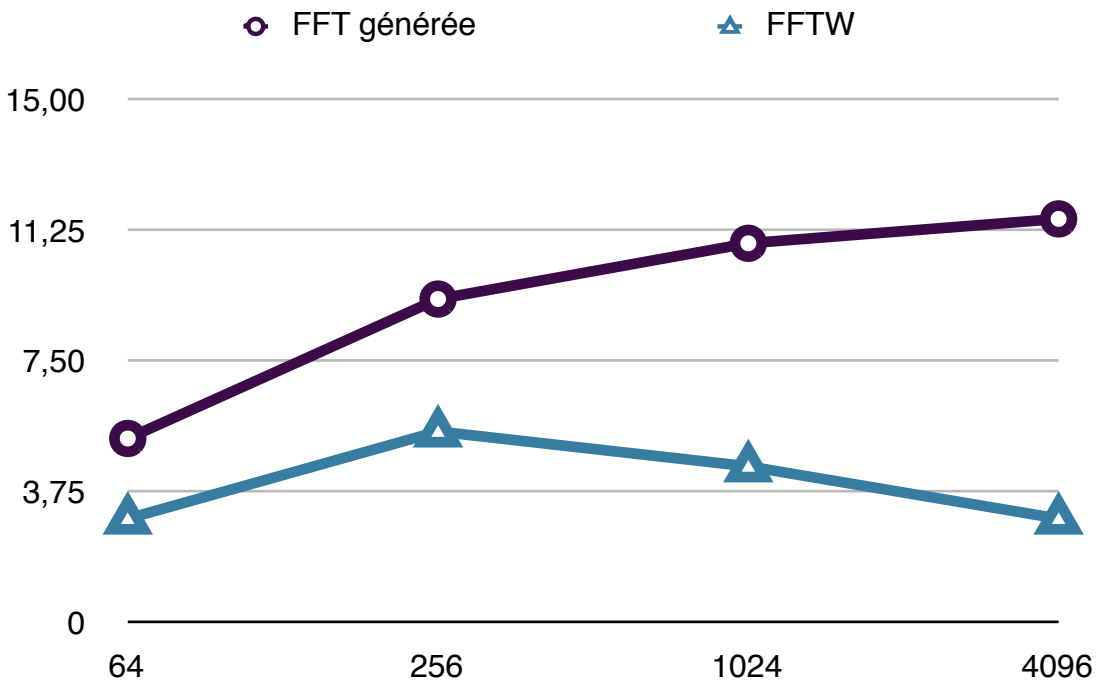

Figure 4.7.3: Performances FFT sur le Cell SPE

5. Conclusion et perspectives

Le déroulement de ce travail de recherche s'est fait en deux temps.

La première partie concerne l'optimisation de la transformée de Fourier rapide sur le processeur embarqué PowerPC 970 FX. Cette optimisation répond à un besoin de performances et de compétitivité dans le domaine de l'embarqué pour des applications de traitement de signal.

Cette volonté de posséder sa propre bibliothèque de traitement de signal nous permet d'une part de figurer parmi les meilleures FFT PowerPC du marché. L'importance d'exploiter le processeur au maximum de ses performances n'est plus à démontrer.

D'autre part, l'intérêt de maîtriser l'algorithme de transformée de Fourier rapide et ses sources est une porte ouverte à toute modification sur mesure de l'algorithme pour le client. En effet, pour gagner en efficacité, une technique d'optimisation nommée fusion de boucle permet de réunir en une seule boucle le traitement de deux boucles indépendantes. Ces situations sont fréquentes dans les applications de traitement de signal. Considérons qu'il faille multiplier le résultat de la transformée de Fourier par un coefficient multiplicateur, une normalisation par exemple. Cette opération peut être alors introduite au sein du calcul de la transformée. Une intégration intelligente pourrait rendre ce calcul transparent. Néanmoins elle sera toujours bénéfique. Ceci permet de ne pas rompre le déroulement du pipeline en ré-initialisant une nouvelle boucle de calcul.

La deuxième partie de la thèse concerne la conception d'un générateur de code optimisé. L'implémentation et l'optimisation d'une transformée de Fourier rapide est un travail long et coûteux. Face au renouvellement incessant des architectures processeur, le besoin d'un générateur permettant de réutiliser notre code optimisé pour de nouveaux processeurs s'est fait ressentir.

Sur le plan optimisation, cette deuxième étude s'est montrée concluante sur l'architecture PowerPC, parvenant à obtenir des performances égales ou meilleures que l'optimisation manuelle.

Sur le portage de la transformée de Fourier, l'expérience a été réussi dans le cas du processeur CELL BE. Les performances obtenues sont très prometteuses. Bien qu'elles ne représentent que 50% des performances maximales du processeur, notre algorithme atteint le double des performances des FFT concurrentes figurant sur FFTW.

En revanche, la génération sur architecture x86 d'Intel tel que le Core 2 Duo n'a pas atteint les performances espérées. D'une part, la génération n'est pas complètement automatisée. Une intervention manuelle est toujours nécessaire pour la gestion des permutations. Certaines d'entre elle ne peuvent être traduites par une seule permutation SSE, contrairement aux permutations AltiVec. D'autre part, un code déroulé comme celui écrit pour le PowerPC 970 n'est pas adapté pour une architecture x86 pauvre en registre.

L'optimisation d'un algorithme requiert toujours une réflexion en amont sur le fonctionnement de l'architecture. L'algorithme choisi reste une solution prometteuse sur l'architecture d'Intel mais nécessite des choix d'optimisation différents. Certaines optimisations telles le déroulement et la multiplication addition fusionnée sont destructrices sur l'architecture x86.

Concernant notre méthode d'exploration des solutions d'ordonnancement, une seule a été présentée. L'accent a été mis sur la méthodologie. Il existe différentes méthodes permettant d'explorer les solutions. Notre souhait d'obtenir la solution optimale nous a intuitivement orienté vers l'étude de toutes les solutions. Le coup d'une telle méthode est effectivement un temps de calcul considérable face aux multiples solutions d'ordonnancement existantes. Néanmoins le coût d'une semaine de génération automatique n'est pas le même qu'une semaine d'optimisation manuelle.

Les méthodes employées par les compilateurs ne permettent pas de parcourir toutes les solutions privilégiant le meilleur rapport rapidité de compilation / performances [68].

La programmation linéaire (PL) est un problème d'optimisation où la fonction objectif et les contraintes sont toutes linéaires. Lorsque les variables sont discrètes comme dans notre modélisation, on parle de programmation linéaire en nombres entiers (PLNE). La résolution de ce problème est nettement plus difficile et peut être NP-complet. Néanmoins certaines formulations ILP peuvent être résolues. Ces formulations sont appelées structurées [69]. Comparée à la méthode d'optimisation choisie, cette dernière est soit meilleure soit équivalente. Une évolution de la méthode de notre générateur vers cette méthode est donc conseillée.

Nous avons étudié l'approche stochastique ce qui nous à permis d'implémenter une génération aléatoire de solutions d'ordonnancement valides. Deux solutions d'exploration ont été testées, une itération de générations aléatoires successives et une itération de permutations d'instructions aléatoires. La rapidité de convergence était plus lente que la méthode retenue.

L'étude de faisabilité d'une approche par les algorithmes génétiques a été faite. Les algorithmes génétiques sont des méthodes d'optimisation s'inspirant directement de l'évolution naturelle. Un tel algorithme repose ainsi sur cinq critères différents:

- une représentation ou un principe de codage des individus de la population,
- un mécanisme de génération de la population initiale,
- une fonction d'évaluation de l'individu
- des opérateurs permettant de diversifier la population. Le croisement recompose les gênes de deux individus exhistants. La mutation est une transformation permettant de garantir l'exploration de toutes les solutions.
- des paramètres de dimensionnement, taille de la population, nombre de génération, condition d'arrêt, probabilité de croisement et de mutation.

Dans notre cas d'optimisation, les mutations pourraient être modélisées par des permutations aléatoires et les croisements par la construction d'une solution intermédiaire entre deux solutions de la population.

L'exemple ci-dessous (figure 5.1) expose le cas d'une mutation et d'un croisement. La permutation est représentée par une permutation aléatoire entre deux instructions du code. Le croisement, quant à lui, n'est possible que si le code étudié possède deux fils d'instructions indépendants issus d'un déroulement de boucle par exemple. Les instructions de 00 à 07 appartiendraient au premier fil tandis que les instructions 10 à 17 appartiendraient au second. Le croisement se traduirait par un moyennage des positions des instructions du premier fil entre deux solutions de la population. Le résultat du croisement est représenté au milieu des deux géniteurs.

Figure 5.1: Exemple de mutation et de croisement au sein d'une population

Enfin, les générations futures d'architectures promettent de nouvelles heures de réflexion. L'exemple de la mise à disposition des documents de référence du futur jeu d'instructions SIMD d'Intel, portant le nom AVX pour Advanced Vector Extensions marque l'arrivée d'un nouveau langage et d'un nouveau parallélisme. Ce dernier proposera une extension du SSE sur 256-bit soit deux fois plus de données que le SSE.

6. Annexes

6.1. Algorithmes FFT

Les deux algorithmes FFT retenus dans le but de donner les meilleures performances sur des processeurs embarqués tels que ceux étudiés sont l'algorithme radix-2 à entrelacement temporel et l'algorithme de Stockham.

```
void radix2(float *xRe, float *xIm, int n, int flag)
{
	int i,j,k,n1,n2;
	float wRe,wIm,e,a,t1,t2;

	/* Calcul du bit-reverse et permutation des données */
	j = 0;
	n2 = n/2;
	for (i=1; i < n - 1; i++)
	{
		n1 = n2;
		while ( j >= n1 )
		{
			j = j - n1;
			n1 = n1/2;
		}
		j = j + n1;
		if (i < j)
		{
			t1 = xRe[i];
			xRe[i] = xRe[j];
			xRe[j] = t1;
			t1 = xIm[i];
			xIm[i] = xIm[j];
			xIm[j] = t1;
		}
	}
	/* Calcul de la FFT */
	n1 = 0;
	n2 = 1;
	m = log2f(n);
	for (i=0; i < m; i++)
	{
		n1 = n2;
		n2 = n2 + n2;
		e = -6.283185307179586/n2;
		a = 0.0;
		for (j=0; j < n1; j++)
		{
			/* Calcul des coefficients */
			wRe = cos(a);
			wIm = flag * sin(a);
			a = a + e;
			for (k=j; k < n; k=k+n2)
			{
				/* Calcul du papillon */
				t1 = wRe*xRe[k+n1] - wIm*xIm[k+n1];
				t2 = wIm*xRe[k+n1] + wRe*xIm[k+n1];
				xRe[k+n1] = xRe[k] - t1;
```

```
                        xIm[k+n1] = xIm[k] - t2;
                        xRe[k] = xRe[k] + t1;
                        xIm[k] = xIm[k] + t2;
                    }
                }
            }
        }

        void stockham(float *xRe, float *xIm, int n, int flag, float *yRe, float *yIm)
        {
            float *y_orig, *tmp;
            int i, j, k, k2, Ls, r, jrs;
            int half, n2, m, m2;
            float wRe, wIm, tr, ti;

            y_orig = yRe;
            r = half = n >> 1;
            n2 = Ls = 1;

            while(r >= n2) {          /* log2(n/n2) itérations */
                tmp = xRe;            /* y est toujours l'ancien pointeur */
                xRe = yRe;            /* x les nouvelles données */
                yRe = tmp;
                tmp = xIm;
                xIm = yIm;
                yIm = tmp;
                m = 0;                /* m parcourt la première moitié du tableau */
                m2 = half;            /* m2 la deuxième moitié */
                for(j = 0; j < Ls; ++j) {
                    wRe = cosf(M_PI*j/Ls);    /* calcul des coefficients */
                    wIm = flag * sinf(M_PI*j/Ls);
                    jrs = j*(r+r);
                    for(k = jrs; k < jrs+r; ++k) {
                        k2 = k + r;
                        tr =  wRe*yRe[k2] - wIm*yIm[k2];
                        ti =  wRe*yIm[k2] + wIm*yRe[k2];
                        xRe[m] = yRe[k] + tr;
                        xIm[m] = yIm[k] + ti;
                        xRe[m2] = yRe[k] - tr;
                        xIm[m2] = yIm[k] - ti;
                        ++m;
                        ++m2;
                    }
                }
                r  >>= 1;
                Ls <<= 1;
            };

            if (yRe != y_orig) {            /* si y diffère de l'original */
                for(i = 0; i < n; ++i)
                    yRe[i] = xRe[i];    /* seulement quand log2(n/n2) est impair */
                    yIm[i] = xIm[i];
                }
            }
        }
```

6.2. CELL Programming Guidelines

La programmation du Cell nécessite un bon apprentissage du fonctionnement de l'architecture. En effet, le processeur requiert une programmation d'assez bas niveau, invoquant l'utilisation d'instructions SIMD très proches de l'AltiVec puisque les huit coprocesseurs constituant le Cell sont exclusivement vectoriels.

Le but de ce document est de fournir un guide de programmation à tout programmeur souhaitant développer une application sur le Cell. L'exemple choisi est celui de la multiplication matricielle et l'enjeu est d'exploiter le maximum des performances du processeur. Cadencé à 3,2GHz, un SPE est capable d'atteindre des performances crêtes de 25,6 GFlops.

Les outils de développement sont l'association d'Eclipse et du SDK 2.1 d'IBM.

Etape 1: Ecrire le code scalaire sur le PPE

Cette étape consiste en une programmation en langage C standard.
Pour simplifier le cas étudier, on se limite à une taille de matrice fixe 24x24.

```
#include<stdio.h>
#include<stdlib.h>
#define  N 24

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

    /* initialisation des matrices */
    ...

    /* multiplication matricielle */
    for(i=0;i<N;i++)
    {
         for(j=0;j<N;j++)
         {
              c[i][j]=0;
              for(k=0;k<N;k++)
              {
                   c[i][j]+=b[i][k]*a[k][j];
              }
         }
    }

    /* verification des resultats */
    ...

    return(0);
}
```

Etape 2: *Vectoriser* le code sur PPE

Les coprocesseurs SPE étant des processeurs vectoriels ne pouvant exécuter que des opérations SIMD, il convient de *vectoriser* le code initial. Bien qu'il soit possible de compiler du code scalaire sur les SPEs, ce n'est en aucun cas une solution à retenir puisqu'elle ne fera que simuler une exécution scalaire sur un processeur SIMD. D'une part, elle n'exploitera qu'un quart des performances du SPE mais entraînera l'ajout d'opérations supplémentaires nécessaires à cette simulation.

Il est donc recommandé de programmer le SPE en langage SIMD. Cependant, le jeu d'instructions utilisé est unique et ne fait aucunement référence à l'AltiVec utilisé par les PowerPC et le PPE du Cell. Il reste cependant très proche de l'AltiVec dans le contenu d'opérations possibles et de son fonctionnement, il est donc plus facile de passer par une étape intermédiaire de programmation AltiVec avant de porter l'application sur un SPE.

Pour la programmation AltiVec, se référer au Programming Interface Manual du PowerPC:
http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

```
#include<stdio.h>
#include<stdlib.h>
#define  N 24

int main()
{
    int i, j, k, k1;
    float a[N][N] __attribute__ ((aligned (16)));
    float b[N][N] __attribute__ ((aligned (16)));
    float c[N][N] __attribute__ ((aligned (16)));
    vector float vA, vB, vC, vtmp;

    /* initialisation des matrices */
    ...

    /* multiplication matricielle */
    for(i=0;i<N;i++)
    {
         for(j=0;j<N;j+=4)
         {
              vC = (vector float)vec_splat_s32(0);
              for(k=0;k<N;k+=4)
              {
                   k1 = k/4;
                   vtmp = vec_ld(0,&(b[i][k1]));

                   vB = vec_splat(vtmp,0);
                   vA = vec_ld(0,&(a[k][j]));
                   vC = vec_madd(vB,vA,vC);

                   vB = vec_splat(vtmp,1);
                   vA = vec_ld(0,&(a[k+1][j]));
                   vC = vec_madd(vB,vA,vC);

                   vB = vec_splat(vtmp,2);
                   vA = vec_ld(0,&(a[k+2][j]));
                   vC = vec_madd(vB,vA,vC);
```

```
                vB = vec_splat(vtmp,3);
                vA = vec_ld(0,&(a[k+3][j]));
                vC = vec_madd(vB,vA,vC);
            }
            vec_st(vC,0,&(c[i][j]));
        }
    }

    /* verification des resultats */
    ...

    return(0);
}
```

Il faut bien veiller à l'alignement des données. L'AltiVec ne peut charger que des données sur des frontières de 16 octets. Si l'adresse fournie en paramètre n'est pas alignée sur 16 octets, c'est à dire que la valeur en hexadécimal de l'adresse ne finit pas par 0, l'opération sera tout de même réalisée mais sur une autre zone mémoire correspondant l'adresse demandée avec l'alignement forcé sur 16 octets.

Etape 3: Porter le code PPE vectorisé sur SPE

Le SPE est donc un processeur SIMD utilisant son propre langage. Le jeu d'instructions est détaillé dans le document Synergistic Processor Unit Instruction Set Architecture v1.2 disponible à l'adresse suivante:
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44

L'utilisation en langage C/C++ est possible tout comme l'AltiVec. Il existe cependant deux langages possibles:

- Specific Intrinsics
- SPU Intrinsics

Les Specific Intrinsics ont la particularité d'être très proche du langage assembleur puisque chaque opération se traduit par une seule et unique instruction. Elles sont désignés par le préfixe *si_*. Par exemple, l'opération d'addition se nomme *si_a*. Le type de données utilisé est unique et se nomme *qword* pour quad word en référence au vecteur contenant quatre mots de 32 bits. Le nom de l'opération suffit à différencier les types de données employés. Il y a donc autant d'instructions que de types de données. Ainsi l'addition entière se fait à l'aide de l'opération *si_a* tandis que l'addition flottante se fait avec *si_fa*.

Les SPU Intrinsics constituent un langage très proche de l'AltiVec, ce qui permet une plus grande facilité de portage. Cependant une opération peut invoquer plus d'une instruction. Les opérations sont ici désignées par le préfixe *spu_*. L'opération d'addition est ici représentée par l'unique instruction *spu_add*. Le type de données est ici précisé lors de la déclaration des variables comme pour l'AltiVec.

La programmation d'un SPE peut mener à combiner les deux langages.

Une liste de *Specific Casting* permet aussi d'effectuer tous les changements de types de données possibles.

Le détail des deux langages est fait dans le document C/C++ Language Extensions for Cell Broadband Engine Architecture v2.5:
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E

Le modèle de programmation du Cell peut être imagé par un processeur chef d'orchestre qui dirige les processeurs SPE. Le PPE est donc responsable de la création des threads SPE. Le programme se compose de 2 codes source distincts, un pour le PPE et un pour le SPE, tous les deux faisant référence à unique fichier d'en-tête.

Le fichier d'en-tête:

matmul.h

```
#include <stdio.h>

#define  N          24
#define  N_4        6
#define  SPE_THREADS     1

typedef struct {
    float* A __attribute__ ((aligned (16)));
    float* B __attribute__ ((aligned (16)));
    float* C __attribute__ ((aligned (16)));
} context;
```

La structure *context* permet au PPE de transmettre les données nécessaires aux SPE chargé du calcul. Le PPE envoie ainsi le pointeur désignant cette structure permettant au SPE de récupérer les données.

Attention, il est important que chaque adresse soit bien alignée sur 16 octets.

Le code PPE:

```
#include <stdlib.h>
#include <libspe.h>
#include "matmul.h"

extern spe_program_handle_t matmul;

int main()
{
    int i, j, status;
    float A[N*N] __attribute__ ((aligned (16)));
    float B[N*N] __attribute__ ((aligned (16)));
    float C[N*N] __attribute__ ((aligned (16)));
    context ctxs __attribute__ ((aligned (16)));
```

```
    speid_t spe_ids;

    /* initialisation des matrices */
    ...

    ctxs.A = A;
    ctxs.B = B;
    ctxs.C = C;

    // Creation du thread SPE en passant le context en parametre.
    spe_ids = spe_create_thread(0, &matmul, &ctxs, NULL, -1, 0);
    if (spe_ids == -1) {
         perror("Unable to create SPE thread");
         return (1);
    }
    // Attente de la fin du thread.
    (void)spe_wait(spe_ids, &status, 0);

    /* verification des resultats */
    ...

    return(0);
}
```

Le code SPE:

```
#include <spu_intrinsics.h>
#include <cbe_mfc.h>
#include "matmul.h"

// Local store structures and buffers.
volatile context ctx;
volatile vector float matA[N*N_4];
volatile vector float matB[N*N_4];
volatile vector float matC[N*N_4];

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i,j,k,k1;
    float val;
    unsigned int tag_id = 0;
    vec_float4 vA,vB,vC,vtmp;

    spu_writech(MFC_WrTagMask, -1);

    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), tag_id,
    MFC_GET_CMD);
    (void)spu_mfcstat(2);
    spu_mfcdma32((void *)(matA), (unsigned int)(ctx.A), N*N_4*sizeof(vector
    float), tag_id, MFC_GETB_CMD);
    spu_mfcdma32((void *)(matB), (unsigned int)(ctx.B), N*N_4*sizeof(vector
    float), tag_id, MFC_GETB_CMD);
    (void)spu_mfcstat(2);

    for(i=0;i<N;i++)
    {
         for(j=0;j<N_4;j++)
         {
              vC = (vec_float4)spu_splats(0);
```

```
                for(k=0;k<N;k+=4)
                {
                      k1 = k/4;
                      vtmp = matB[i*N_4+k1];

                      val = spu_extract(vtmp,0);
                      vB = spu_splats(val);
                      vA = matA[k*N_4+j];
                      vC = spu_madd(vB,vA,vC);

                      val = spu_extract(vtmp,1);
                      vB = spu_splats(val);
                      vA = matA[(k+1)*N_4+j];
                      vC = spu_madd(vB,vA,vC);

                      val = spu_extract(vtmp,2);
                      vB = spu_splats(val);
                      vA = matA[(k+2)*N_4+j];
                      vC = spu_madd(vB,vA,vC);

                      val = spu_extract(vtmp,3);
                      vB = spu_splats(val);
                      vA = matA[(k+3)*N_4+j];
                      vC = spu_madd(vB,vA,vC);
                }
                matC[i*N_4+j] = vC;
          }
    }

    spu_mfcdma32((void *)(matC), (unsigned int)(ctx.C), N*N*sizeof(float), tag_id,
    MFC_PUT_CMD);
    (void)spu_mfcstat(2);

    return (0);
}
```

Ce programme est la traduction en langage SPU de la multiplication vectorielle. Les seules différences notables sont l'ajout de requêtes DMA nécessaire à la lecture et écriture des données se trouvant en mémoire centrale puisque le SPE n'opère qu'avec sa mémoire privée, le Local Store.

Etape 4: Optimisation

On peut maintenant optimiser le code SPE de sorte à exploiter au maximum des ressources du processeur Cell. On peut écrire le code sous forme de plusieurs thread permettant d'utiliser les huit SPEs du Cell, faisant référence aux notions de programmation multi-thread bien connues.

Cependant, on voudrait pouvoir utiliser le maximum des performances d'un SPE. Il est à noter qu'un SPE dispose d'un total de 128 registres vectoriels, soit quatre fois plus qu'un PowerPC. De plus, ne possédant aucun mécanisme évolué de prédiction de branchement, ce processeur présente d'énormes aptitudes aux déroulements de boucles.

Le déroulement complet des deux boucles internes de calcul de la multiplication matricielle donne lieu au code suivant:

```
indexA = si_from_uint((unsigned int)matA);
offsetBC = si_from_uint(16*N_4);
indexB = si_from_uint((unsigned int)matB);
indexC = si_from_uint((unsigned int)matC);
for(i=0;i<N;i++)
{
     vC0 = vzero; vC1 = vzero; vC2 = vzero; vC3 = vzero; vC4 = vzero; vC5 = vzero;
     vtmp0 = (vector float)si_lqd(indexB,0);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
     vA0 = (vector float)si_lqd(indexA,0);    vA1 = (vector float)si_lqd(indexA,16);
     vA2 = (vector float)si_lqd(indexA,32);   vA3 = (vector float)si_lqd(indexA,48);
     vA4 = (vector float)si_lqd(indexA,64);   vA5 = (vector float)si_lqd(indexA,80);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
     vA0 = (vector float)si_lqd(indexA,96);   vA1 = (vector float)si_lqd(indexA,112);
     vA2 = (vector float)si_lqd(indexA,128);  vA3 = (vector float)si_lqd(indexA,144);
     vA4 = (vector float)si_lqd(indexA,160);  vA5 = (vector float)si_lqd(indexA,176);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
     vA0 = (vector float)si_lqd(indexA,192);  vA1 = (vector float)si_lqd(indexA,208);
     vA2 = (vector float)si_lqd(indexA,224);  vA3 = (vector float)si_lqd(indexA,240);
     vA4 = (vector float)si_lqd(indexA,256);  vA5 = (vector float)si_lqd(indexA,272);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
     vA0 = (vector float)si_lqd(indexA,288);  vA1 = (vector float)si_lqd(indexA,304);
     vA2 = (vector float)si_lqd(indexA,320);  vA3 = (vector float)si_lqd(indexA,336);
     vA4 = (vector float)si_lqd(indexA,352);  vA5 = (vector float)si_lqd(indexA,368);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);

     vtmp0 = (vector float)si_lqd(indexB,16);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
     vA0 = (vector float)si_lqd(indexA,384);  vA1 = (vector float)si_lqd(indexA,400);
     vA2 = (vector float)si_lqd(indexA,416);  vA3 = (vector float)si_lqd(indexA,432);
     vA4 = (vector float)si_lqd(indexA,448);  vA5 = (vector float)si_lqd(indexA,464);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
     vA0 = (vector float)si_lqd(indexA,480);  vA1 = (vector float)si_lqd(indexA,496);
     vA2 = (vector float)si_lqd(indexA,512);  vA3 = (vector float)si_lqd(indexA,528);
     vA4 = (vector float)si_lqd(indexA,544);  vA5 = (vector float)si_lqd(indexA,560);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
     vA0 = (vector float)si_lqd(indexA,576);  vA1 = (vector float)si_lqd(indexA,592);
     vA2 = (vector float)si_lqd(indexA,608);  vA3 = (vector float)si_lqd(indexA,624);
     vA4 = (vector float)si_lqd(indexA,640);  vA5 = (vector float)si_lqd(indexA,656);
     vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
     vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
     vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
     vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
     vA0 = (vector float)si_lqd(indexA,672);  vA1 = (vector float)si_lqd(indexA,688);
```

```
vA2 = (vector float)si_lqd(indexA,704);  vA3 = (vector float)si_lqd(indexA,720);
vA4 = (vector float)si_lqd(indexA,736);  vA5 = (vector float)si_lqd(indexA,752);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);

vtmp0 = (vector float)si_lqd(indexB,32);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
vA0 = (vector float)si_lqd(indexA,768);  vA1 = (vector float)si_lqd(indexA,784);
vA2 = (vector float)si_lqd(indexA,800);  vA3 = (vector float)si_lqd(indexA,816);
vA4 = (vector float)si_lqd(indexA,832);  vA5 = (vector float)si_lqd(indexA,848);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
vA0 = (vector float)si_lqd(indexA,864);  vA1 = (vector float)si_lqd(indexA,880);
vA2 = (vector float)si_lqd(indexA,896);  vA3 = (vector float)si_lqd(indexA,912);
vA4 = (vector float)si_lqd(indexA,928);  vA5 = (vector float)si_lqd(indexA,944);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
vA0 = (vector float)si_lqd(indexA,960);  vA1 = (vector float)si_lqd(indexA,976);
vA2 = (vector float)si_lqd(indexA,992);  vA3 = (vector float)si_lqd(indexA,1008);
vA4 = (vector float)si_lqd(indexA,1024); vA5 = (vector float)si_lqd(indexA,1040);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
vA0 = (vector float)si_lqd(indexA,1056); vA1 = (vector float)si_lqd(indexA,1072);
vA2 = (vector float)si_lqd(indexA,1088); vA3 = (vector float)si_lqd(indexA,1104);
vA4 = (vector float)si_lqd(indexA,1120); vA5 = (vector float)si_lqd(indexA,1136);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);

vtmp0 = (vector float)si_lqd(indexB,48);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
vA0 = (vector float)si_lqd(indexA,1152); vA1 = (vector float)si_lqd(indexA,1168);
vA2 = (vector float)si_lqd(indexA,1184); vA3 = (vector float)si_lqd(indexA,1200);
vA4 = (vector float)si_lqd(indexA,1216); vA5 = (vector float)si_lqd(indexA,1232);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
vA0 = (vector float)si_lqd(indexA,1248); vA1 = (vector float)si_lqd(indexA,1264);
vA2 = (vector float)si_lqd(indexA,1280); vA3 = (vector float)si_lqd(indexA,1296);
vA4 = (vector float)si_lqd(indexA,1312); vA5 = (vector float)si_lqd(indexA,1328);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
vA0 = (vector float)si_lqd(indexA,1344); vA1 = (vector float)si_lqd(indexA,1360);
vA2 = (vector float)si_lqd(indexA,1376); vA3 = (vector float)si_lqd(indexA,1392);
vA4 = (vector float)si_lqd(indexA,1408); vA5 = (vector float)si_lqd(indexA,1424);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
vA0 = (vector float)si_lqd(indexA,1440); vA1 = (vector float)si_lqd(indexA,1456);
vA2 = (vector float)si_lqd(indexA,1472); vA3 = (vector float)si_lqd(indexA,1488);
vA4 = (vector float)si_lqd(indexA,1504); vA5 = (vector float)si_lqd(indexA,1520);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
```

```
vtmp0 = (vector float)si_lqd(indexB,64);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
vA0 = (vector float)si_lqd(indexA,1536); vA1 = (vector float)si_lqd(indexA,1552);
vA2 = (vector float)si_lqd(indexA,1568); vA3 = (vector float)si_lqd(indexA,1584);
vA4 = (vector float)si_lqd(indexA,1600); vA5 = (vector float)si_lqd(indexA,1616);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
vA0 = (vector float)si_lqd(indexA,1632); vA1 = (vector float)si_lqd(indexA,1648);
vA2 = (vector float)si_lqd(indexA,1664); vA3 = (vector float)si_lqd(indexA,1680);
vA4 = (vector float)si_lqd(indexA,1696); vA5 = (vector float)si_lqd(indexA,1712);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
vA0 = (vector float)si_lqd(indexA,1728); vA1 = (vector float)si_lqd(indexA,1744);
vA2 = (vector float)si_lqd(indexA,1760); vA3 = (vector float)si_lqd(indexA,1776);
vA4 = (vector float)si_lqd(indexA,1792); vA5 = (vector float)si_lqd(indexA,1808);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
vA0 = (vector float)si_lqd(indexA,1824); vA1 = (vector float)si_lqd(indexA,1840);
vA2 = (vector float)si_lqd(indexA,1856); vA3 = (vector float)si_lqd(indexA,1872);
vA4 = (vector float)si_lqd(indexA,1888); vA5 = (vector float)si_lqd(indexA,1904);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);

vtmp0 = (vector float)si_lqd(indexB,80);
indexB = si_a(indexB,offsetBC);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle0);
vA0 = (vector float)si_lqd(indexA,1920); vA1 = (vector float)si_lqd(indexA,1936);
vA2 = (vector float)si_lqd(indexA,1952); vA3 = (vector float)si_lqd(indexA,1968);
vA4 = (vector float)si_lqd(indexA,1984); vA5 = (vector float)si_lqd(indexA,2000);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle1);
vA0 = (vector float)si_lqd(indexA,2016); vA1 = (vector float)si_lqd(indexA,2032);
vA2 = (vector float)si_lqd(indexA,2048); vA3 = (vector float)si_lqd(indexA,2064);
vA4 = (vector float)si_lqd(indexA,2080); vA5 = (vector float)si_lqd(indexA,2096);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle2);
vA0 = (vector float)si_lqd(indexA,2112); vA1 = (vector float)si_lqd(indexA,2128);
vA2 = (vector float)si_lqd(indexA,2144); vA3 = (vector float)si_lqd(indexA,2160);
vA4 = (vector float)si_lqd(indexA,2176); vA5 = (vector float)si_lqd(indexA,2192);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);
vB0 = spu_shuffle(vtmp0,vtmp0,ctx.vshuffle3);
vA0 = (vector float)si_lqd(indexA,2208); vA1 = (vector float)si_lqd(indexA,2224);
vA2 = (vector float)si_lqd(indexA,2240); vA3 = (vector float)si_lqd(indexA,2256);
vA4 = (vector float)si_lqd(indexA,2272); vA5 = (vector float)si_lqd(indexA,2288);
vC0 = spu_madd(vB0,vA0,vC0); vC1 = spu_madd(vB0,vA1,vC1);
vC2 = spu_madd(vB0,vA2,vC2); vC3 = spu_madd(vB0,vA3,vC3);
vC4 = spu_madd(vB0,vA4,vC4); vC5 = spu_madd(vB0,vA5,vC5);

si_stqd((qword)vC0,indexC,0);
si_stqd((qword)vC1,indexC,16);
si_stqd((qword)vC2,indexC,32);
```

```
    si_stqd((qword)vC3,indexC,48);
    si_stqd((qword)vC4,indexC,64);
    si_stqd((qword)vC5,indexC,80);
    indexC = si_a(indexC,offsetBC);
}
```

L'optimisation à titre d'exemple de cette multiplication matricielle a permis d'atteindre assez rapidement les 18,5 Gflops, soit les deux tiers des performances maximales du processeur.

Remarques complémentaires

Le SPE, tout comme le PPE ne dispose que de deux pipelines d'instructions, ce qui ne lui permet d'exécuter que deux instructions par cycles. Les pipelines sont nommés *odd* (impair) et *even* (pair) dans les documentations. L'un exécute les instructions d'accès mémoire et les permutations et l'autre exécute toute opération de calcul.

Dans le cas étudié, un *profiling* du code permet de mettre évidence qu'on dénombre au total par boucle:

- 150 lectures (pipeline odd)
- 6 écritures (pipeline odd)
- 24 permutations (pipeline odd)
- 144 multiplications additions fusionnées (pipeline even)
- 2 additions (pipeline even)

On remarque que le pipeline chargé des accès mémoire est plus sollicité que le pipeline de calcul et donc que les performances maximales du SPE en GFlops ne sont pas atteignables avec ce code. On dénombre 180x24 instructions exécutées dans le pipeline *odd* contre 146x24 instructions dans le pipeline *even*. Dans des conditions optimales de fonctionnement, ce dénombrement entraînerait un total de 4320 cycles d'exécution, soit un temps d'exécution de 1350 ns à 3,2GHz contre les 1495 ns mesuré. L'écart de performances entre le maximum théorique et les mesures obtenues ne s'élève qu'à seulement 10%.

Conclusion

En résumé, le schéma de programmation préconisé est:

1. PPE: Programmation scalaire de l'application
2. PPE: *Vectorisation* de l'application en langage AltiVec
3. SPE: Portage brut de l'application en langage SPU
4. SPE: Optimisation fine de l'application

Bibliographie

(1) Weiss S., Smith J.E.. POWER et POWERPC, Principes, Architecture. *Morgan Kaufmann Publishers Inc. 1994.*

(2) Stokes J.. PowerPC on Apple: An architectural history, Part 1 & 2. *ArsTechnica.com. 2004.*

(3) IBM®, Power®. Power ISA® Version 2.06. *Softcopy Distribution. 2009.*

(4) Intel Corporation. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. *Order Number 243191. 1999.*

(5) Stokes J..The Pentium: An Architectural history of the World's Famous Desktop Processor, Part 1 & 2. *ArsTechnica.com. 2004.*

(6) Apple Developer Connection. "Software Pipelining". Velocity Engine. Apple Computer, Inc. 2008.

(7) Hennessy J.L., Patterson D.A.. Computer Architecture: A Quantitative Approach, Third Edition. *Morgan Kaufmann Publishers Inc. 2002.*

(8) Philips Semiconductors. Introduction to VLIW Computer Architecture. *Pub#: 9397-750-01759.*

(9) Pokam G., Bodin F.. Energy-Delay Tradeoff Analysis of ILP-based Compilation Techniques on a VLIW Architecture. *Rapport de Recherche INRIA Rennes n°5026. 2003.*

(10) A. Darte, Y. Robert et F. Vivien. Scheduling and Automatic Parallelization. *Birhäuser, 2000.*

(11) Flynn M.J.. Very high-speed Computing systems. *Proceedings of the IEEE, 54:1901-1909. 1966.*

(12) Cosnard M., Trystram D. Parallel Algorithms and Architectures. *Thomson Computer Press. 1995.*

(13) Marr D.T., Binns F., Hill D.L.,Hinton G., Koufaty D.A., Miller J.A., Upton M.. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Corp. 2002.*

(14) Stokes J.. Introduction to the Multithreading, Superthreading and Hyperthreading. *ArsTechnica.com. 2002.*

(15) PowerPC 603® RISC Microprocessor Technical Summary. *Motorola Inc. 1994.*

(16) PowerPC 740/750 RISC Microprocessor User's Manual. *IBM. 1999.*

(17) MPC7448 RISC Microprocessor Hardware Specifications. *Freescale Semiconductor. 2007.*

(18) Freescale Semiconductor. AltiVec® Technology Programming Interface Manual. *1999.*

(19) Freescale Semiconductor. AltiVec® Technology Programming Environments Manual. *rev3. 2006.*

(20) IBM. IBM PowerPC 970FX RISC Microprocessor User's Manual. *version 2.3. 2008.*

(21) Stokes J.. Inside the IBM PowerPC 970, Part 1 & 2. *Arstechnica.com. 2003.*

(22) Freescale Semiconductor. PowerPC e500 Core Family Reference Manual. *Rev1. 2005.*

(23) Freescale Semiconductor. e600 PowerPC Core Reference Manual. *Rev0. 2006.*

(24) Kahne J.A., Day M.N., Hofstee H.P., Johns C.R., Maeurer T.R., Shippy D.. Introduction to the Cell Multiprocessor. *IBM J. RES. & DEV. VOL. 49. NO.4/5. 2005.*

(25) Kistler M., Perrone M., Petrini F.. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro. VOL. 26, Issue 3. p10-23. 2006.*

(26) Torres G.. Inside Pentium M architecture. *Hardwaresecrets.com. 2006.*

(27) Stokes J.. A look at Centrino's Core: The Pentium M. *Arstechnica.com. 2004.*

(28) Kanter D.. Intel's Next Architecture Microarchitecture Unveiled. *Readworldtech.com. 2006.*

(29) Stokes J.. Into the Core: Intel's next-generation microarchitecure. *Arstechnica.com. 2006*

(30) Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture. *Order Number 253665. 2009.*

(31) Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M. *Order Number 253666. 2009.*

(32) Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z. *Order Number 253667. 2009.*

(33) Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1. *Order Number 253668. 2009.*

(34) Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 1. *Order Number 253669. 2009.*

(35) Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. *Order Number 248966. 2009.*

(36) Auto-vectorization in GCC, *http://gcc.gnu.org/projects/tree-ssa/vectorization.html*

(37) VAST-C/AltiVec, Automatic C/C++ Vectorizer for PowerPC AltiVec/VMX, Crescent Bay Software, *v1.6, 2004.*

(38) OpenMP, *http://openmp.org/wp/*

(39) The Message Passing Interface (MPI) standard, *http://www-unix.mcs.anl.gov/mpi/*

(40) Courmontagne P.. Une approche synthétique du traitement du signal déterministe et aléatoire. *Cours ISEN. 2004.*

(41) Cooley J.W., Tukey W.. An algorithm for the machine calculation of complex Fourier series. *Math. Computat., vol.19, pp.297-301, 1965.*

(42) Duhamel P., Hollmann H.. Split-Radix FFT Algorithm. *1984.*

(43) Johnson S.G., Frigo M.. A modified Split Radix FFT with fewer arithmetic operation. *IEEE Signal Processing 55(1) 111-119. 2007.*

(44) Winograd S.. On computing the discrete Fourier Transform. *Math. Computation 32: 175-199, 1978.*

(45) Van Loan C.. Computational Frameworks for the Fast Fourier Transform. *1992.*

(46) Swarstrauber P.N.. Multiprocessor FFTs. *Parallel Computing, 5 p197-210, 1987.*

(47) Bailey D.H.. A High-Performance FFT Algorithm for Vector Supercomputers. *Intl. Journal of Supercomputer Applications, vol. 2, no. 1, p82-87, 1988.*

(48) Chow A.C., Fossum G.C., Brokenshire D.A.. A Programming Example: Large FFT on the Cell Broadband Engine. *2005.*

(49) Chow A.C., Fossum G.C., Brokenshire D.A.. Unleashing the Power: A programming example of large FFTs on Cell. *2005.*

(50) Bruun G.. Z-Transform DFT Filters and FFTs. *IEEE Transactions on Signal Processing, 26, 56-63. 1978.*

(51) FFT Benchmark Results. FFTW.org

(52) SWAR, SIMD Within A Register, *http://aggregate.org/SWAR/*

(53) NASoftware, Conversion Tools, *http://www.nasoftware.co.uk/products/conversiontools.html*

(54) SynDEx, *http://www-rocq.inria.fr/syndex/*

(55) Gedae, *http://www.gedae.com/*

(56) Dolbeau R., Bihan S., Bodin F.. HMPP: A Hybrid Multi-Core Parallel Programming Environment. *CAPS Entreprise, 2007.*

(57) Toomey L.J., Plachy E.C., Scarborough R.G., Sahulka R.J., Shaw J.F., Shannon A.W.. IBM Parallel Fortran. *IBM Systems Journal, vol. 27, NO. 4 1968.*

(58) Puschel M.. SPIRAL: Code Generation for DSP Transforms. *IEEE Vol 93, Issue 2 p235-275, 2005.*

(59) G5 Performance Programming, *http://developer.apple.com/hardwaredrivers/ve/g5.html*

(60) Curry S.. An introduction to the IDE for the Cell Broadband Engine SDK, *IBM, 2007.*

(61) L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Soft. 28-2, p135-151, 2002.*

(62) D.A. Schwartz, R.R. Judd, W.J. Harrod, D.P. Manley. VSIPL 1.3 API. *2008.*

(63) A. Aha, R. Sethi et J. Ullman. Compilateurs, Principes, techniques et outils. *InterEditions, 1991.*

(64) O. Sinnen. Task Scheduling for Parallel Systems, *Wiley, 2007.*

(65) Intel Corporation. Intel Integrated Performance Primitives (Intel IPP) 6.0 In-Depth. *2009.*

(66) M. Frigo, S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE 93 (2), 216–231. Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation. 2005.*

(67) Apple Computer. vDSP Library. *developer.apple.com. 2008.*

(68) R. Allen, K. Kennedy. Advanced Compilation for High Performance Computing. *Edition Morgan-Kauffman. 2005.*

(69) D. Kästner, S. Winkel. ILP-based Instruction Scheduling for IA-64. *ACM Sigplan Notices. Vol 36, Issue 8, p145-154. 2001*

(70) X. Li, M.J. Garzaran, D. Padua. Optimizing Sorting with Genetic Algorithms. *Proceedings of the International Symposium on Code Generation and Optimization. 2005.*

Lexique

A

API
Application Programming Interface

B

Bit Reversing
Opération qui consiste à inverser l'ordre binaire d'une valeur
5 = 0101 après bit reversing 1010 = 9

BU
Branch Unit

C

CISC
Complex Intruction Set Computer

CIU
Complex Integer Unit

CPU
Central Processing Unit ou Processeur.

D

DMA
Direct Memory Access

E

F

FLOPS
Floating-Point Operation Per Second

FPU
Floating-Point Unit

G

GFLOPS
Voir définition de FLOPS. Dans le cas, d'un algorithme FFT et conformément à l'unité de mesure utilisée par FFTW, les performances FFT sont calculées:
$5N\text{Log}_2(N)/\text{Time}$ où N est la taille de la FFT calculée et Time le temps d'exécution d'une FFT en nanoseconde.

H

I

IEEE
Institute of Electrical and Electronics Engineers

ILP
Instruction Level Parallelism

IPC
Instruction Per Cycle

J
K
L

LSU
Load Store Unit

M

MMX
MultiMedia eXtensions

N
O
P
Q
R

RISC
Reduced Instruction Set Computer

S

SIU
Simple Integer Unit

SMP
Symmetric Multi Processing

SMT
Simultaneous Multi Threading

SOC
System-On-Chip

SSE
Streaming SIMD Extensions

Stall
Interruption du flot d'instructions dans le pipeline

Superscalaire
Capacité d'un processeur à exécuter plusieurs instructions simultanément

T
U
V

VLIW
Very Long Instruction Word

W
X
Y
Z