Des flottants 16 bits sur microprocesseurs d'usage général pour images et multimédia

Daniel Etiemble* — **Lionel Lacassagne****

* *LRI, Université Paris Sud*
F-91405 Orsay cedex
de@lri.fr

** *IEF, Université Paris Sud*
F-91405 Orsay cedex
lionel.lacassagne@ief.u-psud.fr

RÉSUMÉ. *Nous évaluons l'implémentation d'instructions flottantes 16 bits sur un Pentium 4 et un PowerPC G5 pour le traitement d'images et le multimédia. En mesurant le temps d'exécution de benchmarks dans lesquels ces nouvelles instructions sont simulées, nous montrons que des accélérations significatives peuvent être obtenues par rapport aux versions flottantes 32 bits. Pour le traitement d'images, l'accélération vient du doublement du nombre d'opérations par instruction SIMD et de la réduction du nombre d'accès mémoire avec un stockage d'octets. Pour le traitement de flots de données avec des tableaux de structures, l'accélération provient des instructions SIMD plus larges.*

ABSTRACT. *We consider the implementation of 16-bit floating point instructions on a Pentium 4 and a PowerPC G5 for image and media processing. By measuring the execution time of benchmarks with these new simulated instructions, we show that significant speed-ups are obtained compared to 32-bit FP versions. For image processing, the speed-up both comes from doubling the number of operations per SIMD instruction and the better cache behavior with byte storage. For data stream processing with arrays of structures, the speed-up comes from the wider SIMD instructions.*

MOTS-CLÉS : *formats flottants 16 bits, traitement d'images, traitement multimédia, processeurs généralistes, Pentium 4, PowerPC.*

KEYWORDS: *16-bit floating-point formats, image processing, media processing SIMD instructions, general-purpose microprocessor, Pentium 4, Power PC.*

1. Introduction

Les applications graphiques et multimédia jouent maintenant un rôle central dans l'utilisation des microprocesseurs d'usage général. Elles ont conduit à l'introduction des extensions SIMD, aussi appelées multimédia, dans la plupart des jeux d'instructions actuellement utilisés. Dans cette introduction, nous discutons la motivation pour introduire des opérations flottantes sur 16 bits et les instructions SIMD correspondantes dans les microprocesseurs généralistes.

1.1. *Des formats flottants 16 bits*

Des formats flottants 16 bits existent depuis longtemps dans certains processeurs de traitement du signal. Par exemple, le TMS 320C32 de Texas dispose d'un format flottant interne sur 16 bits avec 1 bit de signe, un exposant sur 4 bits et une mantisse sur 11 bits qui peut être utilisé comme opérande immédiat par les instructions flottantes et d'un format externe 16 bits avec 1 bit de signe, un exposant sur 8 bits et une mantisse sur 7 bits qui est utilisé pour le stockage mémoire (TI, 1997). Ces formats sont pour la plupart inconnus et peu utilisés. Récemment, un autre format flottant 16 bits a été introduit dans le format graphique OpenEXR (OpenEXR, 2003) et dans le langage graphique Cg (NVdia, 2003; Mark *et al.*, 2003) défini par NVidia. Ce format, appelé *half*, est présenté en figure 1. Un nombre est interprété exactement comme dans les formats flottants IEEE simple ou double précision. L'exposant est biaisé, avec un excès de 15. La valeur 0 de cet exposant biaisé est réservée pour la représentation de 0 lorsque la partie fractionnaire est nulle, et des nombres dénormalisés lorsqu'elle est différente de zéro. La valeur 31 de l'exposant est réservée pour représenter l'infini lorsque la partie fractionnaire est différente de 0, et NaN lorsqu'elle est nulle. Pour 0<E<31, l'expression pour calculer la valeur d'un nombre dans cette représentation flottante est $(-1)^S$ x $(1.fraction)$ x $2^{exposant-15}$. L'étendue des nombres est comprise entre 2^{-24} = 6 x 10^{-8} (dénormalisés) ou 2^{-14} = 6 x 10^{-5} (normalisés) et $(2^{16} - 2^5) = 65504$. Dans le reste de cet article, nous appelons ce format F16.

Le format F16 est justifié par la société ILM, qui a développé le format graphique OpenEXR, comme une réponse aux besoins d'une plus grande précision dans les couleurs pour la visualisation : « Les formats entiers 16 bits représentent les niveaux de gris de 0 (noir) à 1 (blanc), mais ne prennent pas en compte les valeurs qui débordent (par exemple la surbrillance chromatique) qui peut être obtenue sur un négatif ou d'autres afficheurs HDR (High Dynamic Range). A l'opposé, le format TIFF (Tagged Image File Format) avec des flottants 32 bits est souvent trop puissant pour les besoins de visualisation. Le format TIFF avec les flottants 32 bits fournit plus que la précision et la dynamique nécessaire pour les images VFX (fichiers Ulead Photo Express) et augmente le coût de mémorisation, à la fois en mémoire principale et sur disque. » Des arguments semblables sont utilisés pour justifier le format *half* dans le langage Cg. Il est essentiellement utilisé pour réduire les coûts de mémorisation,

Figure 1. *Format half de NVidia, appelé F16 dans cet article*

même si certaines cartes graphiques disposent déjà du format *half* dans le processeur de fragments.

1.1.1. *Formats de données pour le traitement d'images et le multimédia*

Le traitement d'images nécessite généralement à la fois des formats entiers et flottants. Par exemple, vImage (Apple1, 2003), qui est la bibliothèque d'Apple pour le traitement d'images, propose quatre types d'images avec quatre types de pixels. Les deux premiers types de données pixel sont les octets non signés (de 0 à 255) et les flottants (de 0,0 à 1,0) pour une valeur de couleur ou alpha et les deux autres types de données pixel sont des vecteurs de quatre valeurs octets ou flottantes pour alpha, rouge, vert et bleu. Un certain nombre de traitements d'images (estimation de mouvement, détection de contour, lissage, etc.) présentent des caractéristiques semblables : ils travaillent sur des données 8 bits (pixels), effectuent des calculs conduisant à des données sur plus de 8 bits, généralement traités avec le format entiers 32 bits des langages évolués pour éventuellement revenir à des données 8 bits. Les opérations géométriques nécessitent le format flottant. Dans beaucoup de cas, le format F16 serait un bon compromis : la précision et la dynamique des flottants 32 bits n'est pas toujours nécessaire, et les calculs sur des flottants 16 bits est compatible avec un stockage octet des images si des conversions efficaces entre les deux formats sont disponibles. Nous aborderons plus loin l'intérêt des formats flottants pour la compilation.

Pour le traitement multimédia, il y a un réel débat entre l'utilisation des entiers et l'utilisation des flottants. Dans (Kolli, 2003), G Kolli justifie « l'utilisation de la virgule fixe au lieu du flottant pour de meilleures performances 3D » dans la bibliothèque GPP d'Intel (Graphics Performance Primitive). Des techniques pour la conversion automatique des formats flottants en virgule fixe ont été proposés (Menard *et al.*, 2002). D'un autre côté, une arithmétique flottante « allégée » a été proposée pour les applications de traitement du signal dans les applications mobiles faible consommation (Fang *et al.*, 2002). En utilisant la transformée en cosinus discrète inverse comme benchmark, les auteurs ont montré que des nombres flottants avec 5 bits d'exposant et 8 bits de mantisse sont suffisants pour obtenir un rapport signal sur bruit (PSNR) semblable à celui des flottants 32 bits. Cet exemple illustre un cas où le format *half* est adéquat. Si le débat concerne en premier lieu les applications enfouies ou embarquées, il est intéressant d'évaluer l'intérêt du format *half* pour les microprocesseurs généralistes comme une extension des instructions SIMD. En conservant les mêmes registres SIMD, l'utilisation de flottants 16 bits permet de doubler le nombres d'opérations effectuées en parallèle par rapport aux flottants 32 bits ou d'obtenir le même nombre d'opérations qu'avec des entiers 16 bits qui ont une dynamique beaucoup plus réduite.

1.1.2. *Entiers, flottants et instructions SIMD*

La capacité du compilateur à vectoriser, c'est-à-dire à utiliser les instructions SIMD, est un facteur clé dans l'optimisation des performances des benchmarks. De ce point de vue, entiers et flottants ne présentent pas les mêmes caractéristiques. De manière intrinsèque, les opérations arithmétiques entières ont des formats d'entrée et de sortie différents. L'addition de deux nombres de N bits donne un résultat sur N+1 bits, alors que la multiplication de deux nombres de N bits donne un résultat sur 2N bits. Ceci conduit à des spécificités au niveau des instructions SIMD entières implantées dans les jeux d'instructions. L'arithmétique saturée (par définition) et l'arithmétique sur les entiers signés ou non signés ignorent les retenues et ne peuvent être utilisées pour tout calcul dont la dynamique dépasse celle du format d'entrée. La multiplication SIMD a généralement deux instructions différentes pour obtenir soit la partie haute soit la partie basse des 2N bits du résultat pour des entrées sur N bits. Dans le jeu d'instruction IA-32, la seule instruction qui contourne le problème est la multiplication addition (PMADDWD) qui multiplie 4 ou 8 entiers signés 16 bits avec résultat 32 bits et effectue la somme horizontale de deux de ces produits pour délivrer 2 ou 4 entiers 32 bits. C'est une des rares opérations entières avec des formats d'entrée et de sortie différents. Il est clair qu'une telle opération est difficilement utilisable par un compilateur, excepté pour le cas particulier du produit scalaire sur des données 16 bits en traitement du signal pour laquelle elle a été définie. En fait, les opérations avec formats entiers différents en entrée et en sortie sont des instructions *ad-hoc*. L'instruction PSABDW en est l'exemple type. Calculant la somme des valeurs absolues des différences entre octets, elle n'est utilisable que pour l'estimation de mouvement pour laquelle elle a été définie. Au contraire, les instructions SIMD flottantes ont par définition les mêmes formats en entrée et en sortie et sont facilement utilisables par les compilateurs. Pour l'optimisation de programmes portables (n'utilisant pas la programmation au niveau assembleur), l'utilisation de formats flottants présente un avantage significatif. Lorsque le compilateur ne peut vectoriser et qu'il n'y a pas d'obstacle fondamental à la vectorisation comme des accès avec pas non unitaire, des dépendances entre itérations, etc., la seule solution est la vectorisation manuelle au niveau assembleur. Dans le cas du jeu d'instructions IA-32, il y a possibilité d'utiliser les , qui sont l'équivalent des instructions assembleur avec des registres virtuels, le compilateur-assembleur se chargeant de l'allocation des registres.

1.2. *Organisation de cette présentation*

Dans cet article, nous nous concentrons sur l'évaluation de performance des opérations et instructions flottantes 16 bits. Nous n'abordons pas les questions de précision et de dynamique pour les applications graphiques et multimédia, qui sont de la responsabilité du programmeur. C'est à lui de décider d'utiliser des flottants 16 bits avec les instructions correspondantes ou le format flottant 32 bits en fonction des besoins de l'application. Après cette introduction justifiant l'évaluation du flottant 16 bits sur processeurs généralistes, la section II présente la méthodologie qui a été utilisée.

Nous présentons les benchmarks utilisés et la technique pour « simuler » l'exécution d'opérations flottantes 16 bits sur des microprocesseurs existants, le Pentium 4 et le PowerPC G5. La section III présente les hypothèses architecturales et les instructions flottantes 16 bits définies sur le Pentium 4 et le PowerPC G5. La section IV présente l'évaluation de performance sur les différents benchmarks pour le Pentium 4 et le PowerPC G5. La section V présente une évaluation de la surface des opérateurs flottants 16 bits par rapport à la surface correspondante des opérateurs flottants 64 bits utilisés dans les microprocesseurs actuels.

2. Méthodologie

Pour évaluer les performances des instructions F16, nous devons simuler leur temps d'exécution sur des benchmarks significatifs. Dans cette section, nous décrivons les benchmarks et la méthodologie utilisés.

2.1. *Description des benchmarks*

La version C naïve des différents benchmarks est présentée en annexe. Nous ne considérons que des benchmarks qui nécessitent une dynamique supérieure à celle des entiers 16 bits. Dans le cas contraire, aucun avantage ne peut être espéré de l'utilisation des flottants 16 bits.

Pour le traitement d'images, nous considérons d'abord des opérateurs de convolution : les versions « vectorisables » des lisseurs et gradient de Deriche (Deriche, 1987). Ces opérateurs ont été utilisés pour mesurer la performance de détecteurs de contour optimaux sur processeurs RISC et DSP (Ea *et al.*, 1998; Lacassagne *et al.*, 1998; Demigny, 2001). Ils sont représentatifs des filtres spatiaux et ont des temps de calcul significatifs par rapport aux temps d'accès mémoire. Les deux autres benchmarks sont des variantes d'algorithmes de scan. Les opérateurs scan ont été introduit par Blelloch (Blelloch, 1990) pour le calcul parallèle. Etant donné un opérateur associatif o et un vecteur v(x), l'opération scan produit un vecteur w(x) tel que w(x) = v(0) o v(1) o . . . o v(x). Par exemple, le scan+ fournit pour chaque pixel la somme des pixels précédents dans l'ordre de balayage d'une image. Ce benchmark implémente une accumulation 2D et met en évidence les limitations de débit mémoire. Le scan+* est utilisé dans un nouvel algorithme de segmentation d'images proposé par Mérigot (Merigot, 2003). Alors que l'algorithme classique utilise le calcul d'un arbre quaternaire s'appuyant sur la variance d'une région, Mérigot utilise l'opérateur scan+* pour optimiser à la fois la vitesse et la qualité de la segmentation. Cet opérateur accumule la somme et la somme des carrés des pixels. Ces deux derniers benchmarks ont un ratio calcul/accès mémoire moins important que la première série de benchmarks. Ils ont besoin pour les résultats intermédiaires d'une dynamique qui implique des formats flottants.

Pour le traitement multimédia, nous considérons l'étude de cas sur un flot de données OpenGL présenté par Intel (Kumar, 1998). Le benchmark considère un ensemble

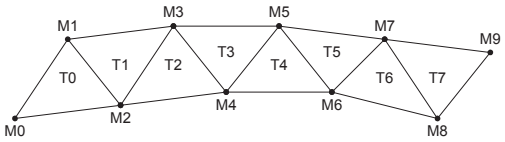

Figure 2. *Triangles en format tri strip*

OpenGL de triangles et calcule la plus petite boîte englobant chaque triangle. On ne considère que la version *tri strip* dans laquelle le premier triangle est représenté par ses trois sommets, mais où chaque triangle additionnel partage un côté avec le triangle précédent (figure 2). Pour chaque nouveau triangle, seul le troisième sommet est mémorisé : pour N triangles, N+2 sommets sont mémorisés. Le code original en assembleur d'Intel a été traduit en code avec « intrinsics » pour la version de référence (flottants 32 bits) à comparer à la version F16.

2.2. *Technique de simulation*

Au lieu d'utiliser un simulateur logiciel, nous avons directement mesuré les temps d'exécution sur des processeurs réels (Pentium 4 et PowerPC G5) en mesurant les temps d'exécution de programmes qui utilisent des instructions du processeur simulant les nouvelles instructions flottantes 16 bits. Cette technique est possible compte tenu des propriétés des benchmarks graphiques et multimédia que nous utilisons : le noyau de calcul est constitué de nids de boucles dont les bornes sont connues à la compilation. En d'autres termes, le comportement des branchements ne dépend pas des données et est complètement déterministe. Dans cette situation, toute instruction à simuler peut être remplacée par n'importe quelle instruction réelle du processeur ayant une latence et un débit de démarrage donné, à condition que les dépendances de données soient les mêmes avec les instructions simulées et avec les instructions réelles.

Cette technique impose de respecter trois contraintes : a) les accès cache doivent être exactement les mêmes pour les instructions simulées et les instructions qui les simulent ; b) les dépendances de données doivent être strictement respectées ; c) comme il n'est plus possible de vérifier la validité des résultats obtenus au niveau du traitement de l'image ou multimédia, il faut soigneusement vérifier que le compilateur génère toutes les instructions nécessaires en fonction du flot de données du programme. Il faut en particulier s'assurer que le compilateur n'optimise pas en supprimant du code jugé inutile. Avec un Pentium 4, l'instruction SIMD MULPS (multiplication flottante simple précision) peut être utilisée pour simuler une instruction MULF16 (multiplication SIMD F16) ayant la même latence (6 cycles) et la même fréquence de démarrage (1 fois tous les 2 cycles). La situation est la même pour l'extension Altivec du PowerPC.

Un inconvénient de cette technique est qu'il est moins facile de faire varier les latences et les débits de démarrage qu'avec un simulateur logiciel. L'avantage est que le meilleur simulateur d'un Pentium 4 ou d'un PowerPC G5 est le processeur réel. On s'abstrait ainsi de tous les problèmes de précision de modélisation dans les simulateurs logiciels, sans parler de la question importante des bogues des simulateurs. En utilisant les mêmes valeurs de latence et de débit de démarrage pour les instructions F16 que pour les instructions flottantes simple et double précisions du Pentium 4, on obtient une borne supérieure du temps d'exécution de ces instructions. En utilisant des latences légèrement réduites, on peut avoir une bonne approximation de réductions possibles des temps d'exécution.

2.3. *Mesures*

Pour chaque benchmark, le temps d'exécution a été mesuré au moins 10 fois et nous avons pris la valeur moyenne. Pour le Pentium 4, nous avons utilisé un processeur Pentium 4 2,4 GHz avec 768 Mo de mémoire exécutant Windows 2000. Le compilateur Intel C++ version 8 a été utilisé avec l'option QxW qui génère le code spécialisé et vectorisé (lorsque c'est possible) pour le Pentium 4. Les temps d'exécution ont été mesurés avec l'instruction RDSTC (read time stamp counter). Toutes les mesures ont été effectuées avec une seule application en cours d'exécution (Visual C++). Pour le PowerPC, nous avons utilisé un G5 1,6 GHz avec 768 Mo de DDR400 exécutant Mac OS X.3. Les programmes ont été compilés avec GCC3.3 et le support Altivec sous l'environnement de développement XCode.

Pour tous les benchmarks de traitement d'images, les résultats sont présentés en nombre de cycles par pixel (CPP), qui est le le nombre total de cycles d'horloge pour exécuter le benchmark divisé par le nombre de pixels. Pour tous les benchmarks, nous avons suivi la même méthodologie. La version entière naïve a été transformée en une version entière SIMD optimisée lorsque c'est possible. Autrement, elle a été transformée en une version flottante où chaque pixel est représenté par un flottant 32 bits. Cette version est généralement vectorisée par le compilateur. Nous avons ensuite écrit une autre version flottante avec les intrinsics d'Intel ou en assembleur Altivec en s'assurant que la performance de cette version avec intrinsics ou assembleur est équivalente ou meilleure que la version vectorisée par le compilateur. La version SIMD entière ou la meilleure version SIMD flottante 32 bits est alors convertie en une version F16 qui opère sur les flottants 16 bits après conversion à partir du format original de l'image (pixels stockés sous forme d'octets non signés). Pour l'étude de cas OpenGL, les résultats sont présentés sous forme de nombre de cycles par triangle.

3. Hypothèses sur le jeu d'instructions et la micro-architecture

3.1. *Pentium 4*

Comme processeur de référence, nous considérons la version disponible en 2003-2004 du Pentium 4 (sans la technologie hyperthreading) avec en plus les instructions horizontales SSE3 Prescott qui seront disponibles dans la prochaine version du Pentium4 (Prescott, 2003). Les instructions Prescott sont simulées avec la même technique que les instructions F16. Nous n'avons considéré que les instructions flottantes et entières 128 bits et les registres XMM 128 bits. On peut en effet considérer que les instructions MMX qui partagent les registres MMX 64 bits avec les instructions flottantes x87 n'ont été qu'une solution transitoire qui n'a plus lieu d'être considérée depuis qu'une solution plus efficace et plus propre existe avec des registres 128 bits indépendants. On se limite aux registres XMM actuellement disponibles en terme de taille (128 bits) et de nombre de registres (8). Proposer une extension à 256 bits signifierait une modification très importante de la microarchitecture en termes d'accès au cache données, de nombre d'unités fonctionnelles pour les unités SIMD (doublement).

Avec des registres et un chemin de données 128 bits, le nombre d'unités fonctionnelles pour chaque opération SIMD F16 est de 8. Les problèmes à résoudre sont : les opérations de conversion entre octets ou entiers 16 bits et format F16, les opérateurs flottants disponibles et les opérations de manipulation de données (permutation, formatage) qui doivent être ajoutées pour le format F16. La conversion d'octets en format F16 implique de transformer huit octets parallèles en huit *half* parallèles. Avec le jeu d'instructions IA-32 qui a un format (2,1), l'opérande source peut être soit un registre, soit un opérande mémoire. On pourrait définir une conversion à partir d'un opérande mémoire 64 bits (ou de la partie basse d'un registre XMM). L'autre option consiste à définir les conversions uniquement sur des registres XMM, ce qui signifie qu'un registre XMM est d'abord chargé à l'aide d'un accès mémoire aligné de 16 octets. On utilise alors deux instructions différentes de conversion. La première convertit les huit octets de la partie basse du registre XMM en huit flottants F16 dans un autre registre XMM. La seconde convertit les huit octets de la partie haute du registre XMM en huit flottants F16 dans un troisième registre XMM. Cette solution conduit à un déroulage implicite d'un facteur deux de toute boucle, les huit octets de poids faible étant traités d'abord, et les huit octets de poids fort ensuite. La seconde opération de conversion n'est pas absolument nécessaire, mais elle évite une instruction supplémentaire pour décaler la partie haute d'un registre XMM dans la partie basse d'un autre registre. Cette option, qui consiste à convertir uniquement à partir de registres, est la plus efficace. Elle évite notamment tous les problèmes d'alignement lors d'accès à des octets voisins comme X[i][j], X[i][j-1] et X[i][j+1] qui sont plus facile à traiter au niveau de registres XMM. Les instructions de conversion inverses de flottants F16 à octets et entiers 16 bits sont aussi nécessaires.

Pour traiter complètement le format F16, les opérateurs F16 nécessaires sont les mêmes que ceux qui sont disponibles dans les formats flottants simple et double précisions, à savoir les opérations SIMD d'addition/soustraction, multiplication, division

et racine carrée. Les opérations logiques bit à bit sont les mêmes que celles déjà disponibles pour les autres formats. L'instruction d'addition horizontale des différents éléments d'un registre SIMD qui a été introduite dans l'extension Prescott est nécessaire : en utilisant exactement le même principe, trois étapes d'additions successives fournissent la somme des huit flottants F16 dans chaque élément du registre SIMD. Dans le cas de tableaux stockés sous forme d'octets, il vaut mieux utiliser directement les instructions de permutation et de compactage/décompactage définies pour les octets avant la conversion en flottants F16. Quand les données sont stockées sous forme de flottants F16, les instructions actuellement disponibles pour les entiers 16 bits peuvent être utilisées, mais elles doivent être étendues aux huit emplacements d'un registre. Ceci pose un petit problème actuellement, car les opérations de permutation ou de compactage/décompactage sont définies par un immédiat sur huit bits dans le jeu d'instructions IA-32. Cette solution est satisfaisante avec quatre emplacements : chaque groupe de deux bits spécifie un des quatre emplacements dans le registre destination. Augmenter le nombre de bits de l'immédiat semble difficile, car les immédiats IA-32 ont 8 ou 32 bits et utiliser 32 bits pour spécifier huit emplacement est peu efficace. Conserver un immédiat huit bits avec huit emplacement semble possible en codant les opérations possibles sur les huit emplacements.

Le tableau 1 présente les différentes instructions F16 que nous avons rajoutées au jeu d'instructions IA-32 et simulées dans nos benchmarks. La latence de ces instructions est spécifiée dans le tableau. Toutes ces instructions ne peuvent démarrer que deux cycles après la précédente de même type. Les latences et débits de démarrage sont les mêmes que celles des instructions actuelles comparables du Pentium 4. Nous n'avons inclus dans le tableau ni les instructions de conversion entre entiers 16 bits et flottants F16, ni les instructions de chargement et de rangement des données *half* (qui sont similaires aux instructions de chargement et rangement des instructions entières 16 bits).

Instruction	Latence (max)	Signification
ADDF16	4	Xmmd := Xmmd + Xmms
HADF16	4	Addition horizontale (Prescott, 2003)
SUBF16	4	Xmmd := Xmmd - Xmms
MULDF16	6	Xmmd :=Xmmd * Xmms
MAXF16	4	Xmmd := Xmmd max Xmms
MINF16	4	Xmmd := Xmmd min Xmms
CBL2F16	4	Xmmd :=BytetoF16(Xmms bas)
CBH2F16	4	Xmmd :=BytetoF16(Xmms haut)
CF162BL	4	Xmmd bas :=F16toByte(Xmms)
CF162BH	4	Xmmd bas :=F16toByte(Xmms)
SHUFF16	4	Xmmd :=shuffle (8 slots) Xmms

Tableau 1. *Instructions flottantes 16 bits sur un Pentium 4*

3.2. *PowerPC G5*

Nous considérons l'implémentation actuelle du processeur G5 (Halfhill, 2002) et les latences d'instructions (Apple2, 2004) données dans le tableau 2. Comme l'extension Altivec est relativement complète, les seules instructions F16 supplémentaires nécessaires sont les versions F16 des instructions flottantes vectorielles et les instructions de conversion entre octets et flottants F16. Toutes les instructions de compactage, décompactage et de permutation nécessaires sont déjà disponibles pour les opérandes entiers 16 bits. Les instructions de conversion simulées ont une latence de 2, ce qui peut être légèrement optimiste. La multiplication accumulation F16 qui est utilisée pour les instructions d'addition, multiplication et multiplication-addition F16 a une latence de 5. Par rapport aux simulations des instructions F16 sur Pentium 4 qui étaient pessimistes, nos simulations sur G5 sont légèrement optimistes.

Unité d'exécution	Cycles
IU (+, -, logique, décalage	2-3
IU (multiplication)	5-7
FPU (+, -, *, MAC)	6
LSU (succès L1) vers GPR, FPR, VR	3, 5, 4-5
LSU (succès L2, chargements seuls)	11
VPERM	2
VSIU (partie de VALU)	2
VCIU (partie de VALU)	5
VFPU (partie de VALU)	8

Tableau 2. *Latences des instructions G5*

4. Résultats mesurés

4.1. *Les benchmarks de Deriche*

4.1.1. *Les problèmes de vectorisation des benchmarks de Deriche*

Le code C des versions initiales non optimisées des lisseurs H et HV des lisseurs de Deriche et du gradient de Deriche est donné en annexe. Pour ces benchmarks, le traitement peut être fait sur place (le tableau destination remplace le tableau initial) ou avec deux tableaux différents pour la source et la destination. Le comportement des caches n'est évidemment pas le même dans les deux cas. Ces benchmarks permettent d'illustrer les problèmes d'utilisation des instructions SIMD.

Le premier problème rencontré est celui des obstacles intrinsèques à la vectorisation, illustré par la version H du lisseur qui possède une dépendance entre itérations car Y[i][j] d'une itération dépend de Y[i][j-1] et Y[i][j-2] pour j croissant. La dépendance peut être résolue en transposant la matrice initiale avant application du lisseur

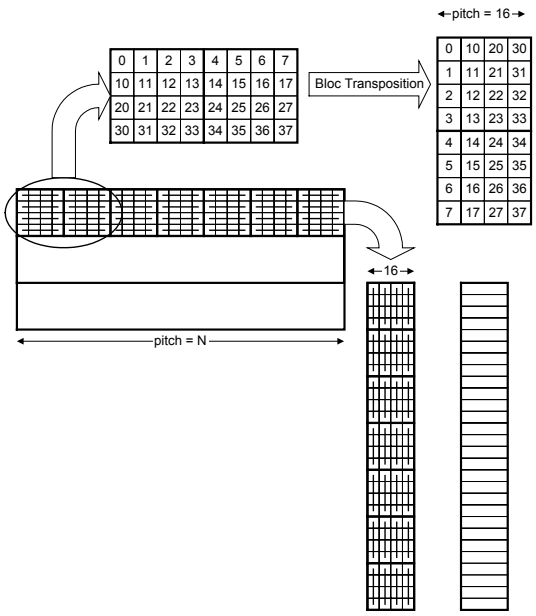

Figure 3. *Principe de la transposition avec colonne tampon*

puis en retransposant après application du lisseur. La transposition peut être effectuée manuellement à l'aide d'une fonction de transposition sur des blocs 16 x 16 en utilisant des instructions SIMD de décompactage. En fait, comme le montre la figure 3 avec des blocs 4 x 4, on peut transposer une suite horizontale de blocs 16 x 16 de l'image initiale dans une colonne verticale de blocs 16 x 16, appliquer le lisseur sur cette colonne tampon, puis retransposer la colonne tampon résultat dans la matrice initiale. Cette approche minimise les défauts de cache. Avec cette double transposition, on est ramené à une situation proche de celle du lisseur HV, ce qui permet d'évaluer le surcoût lié aux transpositions.

Après résolution du problème du lisseur H, on est confronté pour les trois benchmarks au problème du format des données d'un double point de vue : précision des calculs et vectorisation automatique (compilateur) ou manuelle (intrinsics ou assembleurs). En utilisant des flottants 32 bits ou 16 bits, l'amplitude des nombres représentables est suffisante pour qu'il n'y ait aucun problème de calcul, et il n'y a également aucun problème de vectorisation. De plus, les valeurs étant représentées entre 0.0 et 1.0, la division par 256 (opération »8) qui intervient dans les versions entières n'est pas nécessaire. En utilisant des entiers, le seul format réellement utilisable avec les instructions SIMD est le format 16 bits. Une vectorisation manuelle n'est alors possible que si l'on connaît au préalable de manière précise les valeurs des coefficients utilisés. Par exemple, pour les lisseurs de Deriche, si les coefficients b0, a1 et a2 ont tous une valeur inférieure à 256, le produit 8 bits par 8 bits tient sur 16 bits et les

multiplications SIMD sur des entiers de 16 bits avec résultat dans les 16 bits de poids faible peuvent être utilisées. Dans le cas où un coefficient est supérieur à 256, on peut modifier l'expression correspondant à la boucle interne de façon à se ramener au cas précédent. Par exemple, si 256<b0 <512, alors b0*X[i][j] » 8 = X[i][j] + ((b0-256) * X[i][j] » 8. Cet exemple montre clairement la différence essentielle du point de vue vectorisation entre utilisation de formats entiers et de formats flottants.

4.1.2. *CPP pour les benchmarks de Deriche*

Pour les trois benchmarks, les résultats présentés dans le tableau 3 et le tableau 4 correspondent au traitement sur place avec un seul tableau. Les deux tableaux donnent les temps d'exécution des meilleures versions scalaires (entiers et flottants 32 bits) et SIMD pour des entiers 16 bits, des flottants 32 bits et des flottants 16 bits. Les valeurs en italique correspondent aux versions SIMD avec entiers 16 bits pour lesquelles soit il faut vérifier au préalable les valeurs de coefficients (les lisseurs de Deriche) pour éviter de dépasser l'étendue des nombres représentables soit il y a possibilité de débordement (gradient). Nous présentons cette version pour deux raisons : premièrement, elle est implantée avec des instructions réelles et non simulées comme la version F16 ; ensuite, elle est très proche de la version F16 pour laquelle elle représente souvent une borne inférieure des temps d'exécution possibles. La version flottante F16 suppose les latences d'instruction définies dans le tableau 1 pour le Pentium 4 et les latences du G5 définies dans le tableau 2 .

Benchmark	Entiers scalaires (32 bits)	F32 scalaires	Entiers SIMD (16 bits)	F32 SIMD	F16 SIMD
Lisseur H	35,5	30	*9,1*	19,7	9,3
Lisseur HV	33,3	17	*6,1*	16,9	7
Gradient	17	11	*4,1*	6,8	3,5/5,3

Tableau 3. *Nombre de cycles par pixel (CPP) des benchmarks de Deriche pour une image 512 x 512 sur un Pentium 4*

Benchmark	Entiers scalaires (32 bits)	F32 scalaires	Entiers SIMD (16 bits)	F32 SIMD	F16 SIMD
Lisseur H	27,7	10,3	*4,2*	13,8	5
Lisseur HV	25	23	*2,2*	11,1	2,6
Gradient	17,6	73,6	*2,4*	5,7	2,5

Tableau 4. *Nombre de cycles par pixel (CPP) des benchmarks de Deriche pour une image 512 x 512 sur un Power PC G5*

Pour les deux processeurs, les versions SIMD 16 bits entières et F16 pour les lisseurs et le gradient surpassent les versions scalaires entières et flottantes 32 bits. Pour les processeurs, les versions entière et flottante 16 bits ont des performances proches car elles utilisent les mêmes accès mémoire et les mêmes types d'opérations de conversion (octets vers entiers 16 bits par décompactage avec ajout de zéros, octets vers F16 par véritable conversion). Les différences de performance entre elles proviennent des différences entre les latences des opérations de calcul entières ou F16 utilisées.

Pour le Pentium, la version F16 est légèrement plus lente que la version entière 16 bits parce que les additions F16, avec une latence de 4 cycles contre 2 pour les additions entières, sont plus fréquentes que les multiplications F16, dont la latence de 6 cycles est inférieure aux 8 cycles de latence de la version entière 16 bits. Cependant, il ne faut pas oublier que les versions entières dépendent des valeurs des coefficients. Le programmeur doit prévoir les différentes versions du programme en fonction des valeurs des coefficients, alors que la version F16 est générique. Le gain de la version F16 par rapport la version SIMD flottante 32 bits a deux causes : il y a deux fois plus d'opérations par instruction SIMD et les données stockées (octets) occupent quatre fois moins de place mémoire. Il y a deux valeurs pour la version F16, selon que l'on implante ou non l'instruction FABS16 (valeur absolue) qui n'existe pas dans la version flottante simple précision de IA-32.

Pour le G5, l'accélération de la version F16 par rapport à la version flottante SIMD 32 bits varie de 2,2 à 4,2, accélération qui provient des instructions SIMD avec deux fois plus d'opérandes, de la différence dans le comportement des caches et des latences réduites des instructions (5 pour les opérations arithmétiques F16 au lieu de 8, et la latence de 2 pour les instructions de conversion est moindre que les instructions de conversion entier à flottants 32 bits du G5).

La différence essentielle entre les versions entière et flottante 16 bits vient du fait que la version F16 serait facilement vectorisable par un compilateur à cause de l'homogénéité des formats d'entrée et de sortie, alors que la version Int16 l'est très difficilement. Dans le cas des multiplications (lissage), la version F16 supprime le problème du contrôle des dynamiques des résultats intermédiaires. L'avantage de ces deux versions 16 bits par rapport au flottant 32 bits est qu'elles sont compatibles avec une mémorisation sous forme d'octets alors que les flottants 32 bits impliquent quasi automatiquement une mémorisation sous forme de flottants 32 bits et qu'elles utilisent des opérations SIMD à 8 opérandes au lieu de 4. A la fois le nombre de transferts mémoire et le nombre de calculs sont réduits.

4.2. *Performance sur les benchmarks scan*

Nous considérons maintenant des benchmarks qui ont un rapport calcul/accès mémoire moins important : le scan+ a deux additions pour trois accès mémoire. A cause de l'accumulation, l'accumulateur doit être suffisamment grand pour éviter tout débordement : des entiers 32 bits ou des flottants 32 bits sont utilisés généralement pour de grandes images et des entiers 16 bits pour de petites images. Le scan peut être implanté en deux passes, l'une pour l'accumulation horizontale et l'autre pour l'accumulation verticale, ou en une seule passe combinant les deux balayages. Nous ne considérons ici que la version la plus performante en une passe. Contrairement aux benchmarks de Deriche, le résultat du scan est dans un tableau différent du tableau initial.

Pour le scan+*, le format entier 32 bits n'a pas une dynamique suffisante : une dynamique de 2^{34} est nécessaire pour une image 512x512 et 2^{36} pour une image

1024 x 1024. Des flottants 16 bits ou 32 bits sont nécessaires. Les sorties peuvent être rangées dans deux matrices différentes de flottants 32 bits contenant respectivement le scan+ et scan+* le ou les deux scans peuvent être entrelacés dans une seule matrice améliorant ainsi la localité spatiale des données. La version F16 a été simulée en partant de la version Int16 précédente et en ajoutant le code pour les calculs de carrés et les accumulations. Le scan+ (somme horizontale) dans un registre de 128 bits a été implémenté par trois paires d'instructions d'addition/décalage.

Les temps d'exécution des opérateurs scan+ et scan+* sont donnés dans le tableau 5 pour le Pentium 4 et le tableau 6 pour le G5 en même temps que le résultat de la copie, qui correspond au simple transfert des données d'un tableau dans un autre avec changement (ou non) de format. La copie donne la borne inférieure du CPP que l'on peut obtenir, puisqu'il n'y a pas d'opérations effectuées. Les valeurs en italique correspondent aux versions entières 16 bits pour le scan+ et 32 bits pour le scan+* pour lesquelles la dynamique n'est pas suffisante pour avoir des résultats corrects.

Données	Copie scalaire	Copie SIMD	Scan+ scalaire	Scan+ SIMD	Scan+* scalaire	Scan+* SIMD
Int8 - Int16	4,9	4,7	*5,6*	*7,2*		*9,9*
Int8 - Int32	9,4	9,2	9,6	10,5	*18,8*	*18,7*
Int8 - F32	9,5	9,4	10	10,6	19	18,9
F32 - F32	13,6	12,5	15,3	17,5	17	21,3
Int8 - F16				7,8		10,5

Tableau 5. *Nombre de cycles par pixel (CPP) pour la copie, le scan+ et le scan+* pour une image 512 x 512 sur un Pentium 4*

Données	Copie scalaire	Copie SIMD	Scan+ scalaire	Scan+ SIMD	Scan+* scalaire	Scan+* SIMD
Int8 - Int16	5,5	4,5	*24,3*	*5,1*		*7,8*
Int8 - Int32	9,3	6,7	10,4	7	*17,7*	*13*
Int8 - F32	62,4	7	95	7,7	26,7	15
F32 - F32	10,4	7,6	18	15	18,5	15,8
Int8 - F16				5,1		7,8

Tableau 6. *Nombre de cycles par pixel (CPP) pour la copie, le scan+ et le scan+* pour une image 512 x 512 sur un Power PC G5*

Les performances du scan+ et de la copie sont très proches, ce qui montre que ce benchmark est clairement limité par les accès mémoire. Dans cette situation, l'utilisation de formats flottants (dont F16) ne peut apporter aucun avantage et la meilleure solution consiste à utiliser des entiers 32 bits.

Pour le scan+*, la version F16 sur Pentium a une accélération légèrement inférieure à 2 par rapport à la version flottante scalaire 32 bits qui provient à la fois des

instructions F16 avec deux fois plus d'opérandes et du meilleur comportement de la hiérarchie mémoire (le tableau de résultats avec des flottants F16 occupe deux fois moins de place mémoire que les flottants 32 bits). La meilleure version entière 32 bits, qui n'a pas assez de dynamique, nécessiterait 12,1 CPP au lieu de 7 pour la version F16. On peut remarquer que les opérations scan+ et scan+* sont très défavorables à l'utilisation des instructions SIMD car chaque élément est la somme (ou la somme des carrés) des pixels précédents, ce qui correspond au cas typique des récurrences interdisant la vectorisation. Il faut un nombre significatif d'instructions SIMD de manipulation des données pour obtenir l'accumulation des sommes précédentes dans les différentes parties d'un registre SIMD. Avec le Pentium 4, la version « manuelle » SIMD est plus lente que la version scalaire à cause des nombreuses manipulations de données. Seule la version SIMD F16 présente un avantage par rapport aux versions flottantes 32 bits. La version F16 du scan+* a un temps d'exécution qui est 1,3 fois celui de la copie la plus rapide avec des entiers 16 bits, qui correspond à la meilleure performance possible. Le surcoût provient des instructions de gestion de boucle et du nombre significatif d'instructions de copie de registre à registre qui résultent du format à deux opérandes des instructions IA-32 (quand l'opérande destination doit être conservé, il doit d'abord être copié dans un autre registre). Pour améliorer encore la performance, il faudrait améliorer le débit mémoire et/ou réduire les instructions de gestion des boucles imbriquées, comme le suggère l'architecture MediaBreeze (Talla *et al.*, 2003).

Pour le PowerPC G5, le format F16 fournit une accélération de 1,5 par rapport au format flottant simple précision et de 1,4 par rapport à la version entière sur le scan+. Pour le scan+*, qui nécessite absolument un format flottant, F16 apporte une accélération de 1,9 par rapport au format flottant simple précision. La grande richesse de l'extension SIMD Altivec par rapport à l'extension SSE/SSE2 rend plus efficace la vectorisation manuelle du scan+ et du scan+*, ce qui conduit à des performances SIMD meilleures que les performances scalaires, contrairement au cas du Pentium 4.

4.3. *Performance sur un benchmark OpenGL*

Pour l'étude de cas de flot OpenGL, la version de référence a des sommets avec des coordonnées représentées par des flottants 32 bits et range les coordonnées de chaque boîte englobante dans un entier 32 bits contenant 3 valeurs de 10 bits. La version F16 a des coordonnées représentées par des flottants 16 bits et range les coordonnées des boîtes englobantes dans un mot de 64 bits (3 valeurs F16 + padding).

Les performances des versions F32 et F16 pour les deux processeurs, exprimées en nombre de cycles par triangle et en nombre de cycles par instruction (CPI) sont données dans le tableau 7. L'accélération de 1,8 avec le Pentium 4 provient des huit opérations parallèles par instruction SIMD. L'accélération pour le G5 est d'environ 2, mais il utilise environ 9 fois moins de cycles par triangle. Cette différence a deux causes : il utilise environ deux fois moins d'instructions pour transformer la structure de données initiale (tableau de structures) en structure de données adéquate pour les

opérations SIMD et le CPI est bien meilleur (CPI=1,6 pour le G5 contre 7,1 pour le Pentium pour la version 32 bits). L'impact de l'unité de permutation vectorielle (VPU) d'Altivec, qui se comporte comme un cross-bar complet entre deux registres source et un registre destination, est ici plus grand que pour les filtres de Deriche, car il y a un très grand nombre de reformatages de données à effectuer : désentrelacement des coordonnées (x, y, z) pour augmenter le parallélisme de calcul, puis entrelacement des coordonnées min/max de chaque boîte englobante.

Performance Processeur	Cycles par triangle Pentium 4	Cycles par instructions Pentium 4	Cycles par triangle PowerPC G5	Cycles par instruction PowerPC G5
F32	195	7,1	21,5	1,6
F16	107,5	5	10,5	1,3

Tableau 7. *Performances sur le benchmark OpenGL pour le Pentium 4 et le PowerPC G5*

5. Evaluation de surface des unités fonctionnelles flottantes 16 bits

Dans les microprocesseurs actuels, pour minimiser la surface, les mêmes opérateurs flottants sont utilisés pour les opérations flottantes en simple et double précision, comme l'indique le fait que les opérations ont les mêmes latences pour les opérateurs pipelinés comme l'addition et la multiplication. On pourrait penser utiliser ces mêmes opérateurs flottants pour le format F16. Cependant, cela obligerait à doubler le nombre d'opérateurs flottants 64 bits (et donc la surface) par rapport à ce qui existe actuellement. Envisager ce doublement dans l'hypothèse d'un passage à des registres 256 bits est possible, mais laisse entier le problème pour les opérateurs flottants F16. Il faut toujours deux fois plus d'opérateurs F16 que d'opérateurs simple et double précision pour que les opérateurs F16 présente un intérêt.

Seule une implémentation des opérateurs flottants 16 bits dans le microprocesseur complet (Pentium 4 ou G5) pourrait fournir des valeurs significatives de la surface, de la puissance dissipée et des performances dynamiques des unités fonctionnelles 16 bits. Pour obtenir un ordre de grandeur grossier, nous avons utilisé des modèles VHDL d'opérateurs flottants et une bibliothèque de cellules $0,18\ \mu m$ de ST (technologie HCMOS8D)(CMP, 2004) . La même approche a été utilisée par Talla et al (Talla *et al.*, 2003) pour évaluer le coût matériel de l'architecture MediaBreeze. Les modèles VHDL ont été développés par J. Detrey et F. De Dinechin (Detrey *et al.*, 2003) : ils comprennent des versions non pipelinées et des versions pipelinées pour les opérations d'addition, multiplication, division et racine carrée. L'additionneur utilise deux chemins de données différents : un chemin « proche » quand les valeurs d'exposant sont proches et un chemin « large » quand leur différence est importante. Le diviseur utilise un algorithme SRT à base 4 (Ercegovac *et al.*, 1994) alors que l'opérateur racine carrée utilise un algorithme SRT à base 2.

Dans le tableau 8, nous donnons la surface des différents opérateurs non pipelinés estimée par l'outil de synthèse Cadence 4.4.3 avant placement et routage. Pour

huit unités fonctionnelles F16 comprenant les quatre opérateurs cités, la surface serait inférieure à 11% de la surface des quatre unités fonctionnelles 64 bits qui sont implantés dans les microprocesseurs d'usage général (nous supposons que les mêmes unités fonctionnelles sont utilisées pour les opérations en simple et double précision, comme l'indique le fait qu'ils ont la même latence.

Opérateur	16 bits	64 bits	Rapport
Additionneur	0,019	0,097	19,9 %
Multiplieur	0,016	0,276	5,91 %
Diviseur	0,047	1,008	4,64 %
Racine carrée	0,027	0,679	4,04 %
Total	0,110	2,059	5,33 %

Tableau 8. *Estimation de surface en mm^2 pour des opérateurs flottants non pipelinés dans une technologie CMOS 0,18 μm*

Comme déjà indiqué, cette estimation est très grossière et n'a pour but que donner un ordre de grandeur. Une première raison est qu'utiliser des modèles VHDL écrits pour des implantations sur FPGA est plus que discutable pour évaluer des surfaces pour une implantation VLSI. Une seconde raison est que nous avons utilisé exactement les mêmes modèles pour évaluer des opérateurs 16 bits et 64 bits alors que les contraintes de temps de propagation ne sont pas fondamentalement différentes (nous avons dans nos simulations estimé que les latences des opérateurs F32 et F16 étaient les mêmes). Les opérateurs F16 peuvent certainement utiliser des structures plus simples, utilisant relativement moins de composants que les opérateurs calculant des flottants 64 bits. Par exemple, dans notre évaluation, l'additionneur F16 est relativement gros par rapport aux autres opérateurs. L'approche à deux chemins qui donne les meilleurs résultats pour l'additionneur 64 bits est trop luxueuse pour un additionneur 16 bits et un schéma plus simple pourrait être utilisé, comme indiqué dans (Fang *et al.*, 2002). Ensuite, comme le montre l'article (Fetzer *et al.*, 2002), les performances des opérateurs arithmétiques dépendent plus de la manière dont ils accèdent à leurs opérandes et fournissent leur résultat dans les bancs de registres que des temps de propagation propres à l'opérateur arithmétique lui-même. Enfin, la comparaison d'opérateurs flottants 16 bits avec des opérateurs 64 bits est possible pour le G5 avec les réserves indiquées ci-dessus. Pour le Pentium qui utilise la précision étendue, il faudrait comparer des opérateurs 16 bits avec des opérateurs 82 bits.

Seuls Intel et IBM possèdent les données qui permettent de traiter des problèmes comme l'implantation réelle des opérateurs dans les différents chemins de données, la question des codes opération, etc.

6. Conclusion

Pour les opérations graphiques effectuées par les processeurs et les cartes graphiques, il y a un compromis à faire entre la précision et la dynamique des calculs d'une part, et le coût de mémorisation d'autre part. Beaucoup d'applications graphiques ont de meilleures performances avec des formats flottants qu'avec des formats entiers. L'une des raisons est qu'avec les formats flottants, la vectorisation manuelle ou par le compilateur est plus facile car les opérations flottantes ont le même format en entrée et en sortie. Toutefois, le format flottant simple précision utilise quatre fois plus de mémoire que le format octet. Quand il fournit suffisamment de précision et de dynamique, le format flottant 16 bits défini par ILM pour le format OpenEXR et NVidia semble un bon compromis.

Dans cet article, nous avons considéré un ensemble limité d'opérations flottantes 16 bits et un ensemble d'instructions de conversion entre les octets et les flottants 16 bits dans le cadre de deux microprocesseurs d'usage général : le Pentium 4 et le PowerPC G5. Nous avons mesuré les temps d'exécution de différentes versions de benchmarks graphiques typiques (lisseurs et gradient de Deriche, scans) avec des formats entiers, flottants 32 bits et flottants 16 bits. Pour ce dernier format, nous avons simulé les instructions F16 en utilisant des instructions existantes du Pentium 4 ou du PowerPC G5 ayant la même latence et le même débit de démarrage que les instructions simulées.

Pour les benchmarks graphiques ou multimédia limités par le calcul, le format F16 fournit une accélération qui peut dépasser 2 par rapport aux utilisations de formats flottants 32 bits. Pour les applications graphiques travaillant sur des données rangées sous forme d'octets, la version F16 obtient des performances semblables à celles que l'on peut obtenir avec une vectorisation manuelle sur des entiers 16 bits, mais avec un double avantage : la vectorisation est beaucoup plus facile, notamment pour le compilateur et elle permet de travailler sur une dynamique beaucoup plus importante au niveau des résultats intermédiaires. Enfin, les différences significatives dans les extensions SIMD liées aux différences entre les caractéristiques essentielles des jeux d'instructions IA-32 (CISC) et PowerPC (RISC) peuvent conduire dans certains cas à des différences très significatives de performance (exprimées en cycles d'horloge), mais les gains de performance des flottants 16 bits par rapport aux flottants 32 bits pour les deux processeurs restent semblables pour l'ensemble des benchmarks considérés.

Une évaluation grossière de la surface de silicium montre que pour huit unités fonctionnelles flottantes 16 bits, la surface ne devrait pas dépasser environ 11% de la surface actuellement attribuée aux unités fonctionnelles flottantes dans un Pentium 4 ou un G5.

Ce travail doit être complétée en considérant un ensemble plus significatif d'applications graphiques et multimédia. Nous ne pensons pas que les résultats complémentaires modifieront la conclusion générale de cet article.

Remerciements

A. Dupret et J. O. Klein, de l'IEF (UMR du CNRS et Université Paris Sud) nous ont fourni les évaluations de surface de silicium en utilisant l'outil de synthèse de Cadence avec la technogie $0,18\mu m$ de ST. Leur aide a été extrêmement précieuse.

F. Tourant, d'Apple France Education, a mis à notre disposition pendant deux semaines un biprocesseur PowerPC G5 à 2 GHz, qui nous a permis de commencer les expériences et les mesures sur le PowerPC.

7. Bibliographie

Apple1, Introduction to vImage, Rapport technique, http ://developer.apple.com/documentation/Performance/Conceptual/vImage, 2003.

Apple2, G5 Performance Programming, Rapport technique, http ://developer.apple.com/hardware/ve/g5.html, 2004.

Blelloch G. E., *Vector Model for Data-Parallel Computing*, The MIT Press, 1990.

CMP, Multi-Chip projects, Design kits, Rapport technique, http ://cmp.imag.fr/ManChap4.html, 2004.

Demigny D., *Méthodes et architectures pour le TSI en temps réels*, Lavoisier, 2001.

Deriche R., « Using Canny's criteria to derive a recursively implemented optimal edge detector », *The International Journal of Computer Vision*, p. 167-187, May, 1987.

Detrey J., DeDinechin F., A VHDL Library of Parametrisable Floating Point and LSN Operators for FPGA, Rapport technique, http ://www.ens-lyon.fr/ jdetrey/FPLibrary, 2003.

Ea T., Lacassagne L., Garda P., « Exécution temps réel de contours de Deriche par des processeurs RISC », *Proceedings Conférence Adéquation AlgorithmesArchitecture'98*, 1998.

Ercegovac M. D., Lang T., *Division and square root : Digit-Recurrence Algorithms and Implementations*, Kluwer Academics Publishers, Boston, 1994.

Fang F., Chen T., Rutenbar R. A., « Lightweight Floating-Point Arithmetic : Case Study of Inverse Discrete Cosine Transform », *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication Systems*, 2002.

Fetzer E. S., Gibson M., Klein A., Calik N., Zhu C., Busta E., Mohammad B., « A Fully Bypassed Six-Issue Integer Datapath and Register File on the Itanium-2 Microprocessor », *IEEE Journal of Solid State Circuits*, 2002.

Halfhill T. R., « IBM trims Power4, adds Altivec », *Microprocessor Report*, 2002.

Horowitz S., Pavlidis T., « Picture segmentation by a tree traversal algorithm », *Journal of the ACM*, vol. , p. 368-388, 1976.

Intel1, IA-32 Intel Architecture Software Developer's Manual - Volume 2 : Instruction Set Reference, Rapport technique, http ://www.intel. com, 2001.

Intel2, Intel Pentium 4 and Intel XeonTM Processor Optimization - Reference Manual, Rapport technique, http ://www.intel. com, 2002.

Kolli G., Using Fixed-Point Instead of Floating Point for Better 3D Performance, Rapport technique, http ://www.devx.com/Intel/article/16478, 2003.

Kumar A., SSE2 Optimization - OpenGL Data Stream Case Study - Intel application notes, Rapport technique, www.intel.com/ cd/ids/developer/ asmo-na/eng/segments/ games/resources/graphics/19224.htm, 1998.

Lacassagne L., Lohier F., Garda P., « Real time execution of optimal edge detectors on RISC and DSP processors », *ICASSP'98*, 1998.

Mark W. R., R.S.Glanville, Akeley K., Kilgard M., « Cg : A system for programming graphics hardware in a C-like language », *ACM Transactions on Graphics (TOG)*, ACM, July, 2003.

Menard D., Chillet D., Charot F., Sentieys O., « Automatic Floating-point to Fixed-point Conversion for DSP Code Generation », *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2002.

Merigot A., « Revisiting image splitting », *International Conference on Image Analysing and Processing (ICIAP)*, p. 314-319, 2003.

NVdia, Cg User's manual, Rapport technique, http ://developer.nvidia.com/ object/ view.asp ?IO=cg_toolkit, 2003.

OpenEXR, OpenEXR, Rapport technique, www.openexr.org/details.html, 2003.

Prescott, Prescott New Instructions Software Developer's Guide, Rapport technique, http ://www.intel. com, 2003.

Talla D., John L. K., Burger D., « Bottlenecks in Multimedia Processing with SIMD style extensions and Architectural Enhancements », *IEEE Transactions on Computers*, vol. 52, n° 8, p. 1015-1031, 2003.

TI, TMS 320C3x User's guide, Rapport technique, http ://focus.ti.com/lit/ug/spru031e/spru031e.pdf, 1997.

Article reçu le 10 février 2004
Version révisée le 20 avril 2005

Daniel Etiemble *est professeur au Laboratoire de Recherche en Informatique de l'Université Paris-Sud. Ses thèmes de recherche sont le calcul parallèle et l'architecture des ordinateurs. Il s'intéresse aux applications multimédia embarquées.*

Lionel Lacassagne *est maître de conférences à l'Institut d'Electronique Fondamentale de l'Université Paris-Sud. Ses thèmes de recherche concernent le traitement d'images et l'adéquation algorithme architecture. Il s'intéresse particulièrement aux systèmes embarqués.*

8. Annexe : benchmarks

```
// define byte unsigned char;
// Deriche Lissage horizontal
INT_Lissage2_H_C_B(byte **m, int size) {
  int i,j, b0, a1, a2; byte **X, **Y;
  X = m;Y = m;
  for(i=0; i<size; i++) {
```

```
    for(j=0; j<size; j++) {
      Y[i][j] = (byte)((b0*X[i][j] + a1*Y[i][j-1] +
                a2*Y[i][j-2]) >> 8);}
    for (j=size-1;j>=0;j--) {
      Y[i][j] = (byte)((b0* X[i][j] + a1*Y[i][j+1] +
                a2*Y[i][j+2]) >> 8);}}}

// Deriche  Lissage vertical et balayage horizontal
 for(i=0; i<size; i++) {
    for(j=0; j<size; j++) {
      Y[i][j] = (byte)((b0*X[i][j] + a1*Y[i-1][j] +
                a2*Y[i-2][j]) >> 8);}}
 for (i=size-1;i>=0;i--) {
    for(j=0; j<size; j++) {
      Y[i][j] = (byte)((b0*X[i][j] + a1*Y[i+1][j] +
                a2*Y[i+2][j]) >> 8);}}

// Deriche  Gradient
void INT_Gradient_C_B(byte **m, int size) {
  int i, j; byte **X, **G;
  X = m; G = m;
  for(i=0; i<size-1; i++) {
    for(j=0; j<size-1; j++) {
      G[i][j] = (byte) (abs(-X[i][j]+X[i][j+1]-X[i+1][j]+ X[i+1][j+1])
            + abs(-X[i][j]-X[i][j+1] +X[i+1][j] + X[i+1][j+1]));}}

// Scan+ (1 passe)
byte X[height][width]; short S [height][width], sx;
 Y[0][0] = sx =X[0][0];
 for(j=1; j<width; j++) {Y[0][j] = sx += X[0][j];}
 for(i=1; i<height; i++) {
        sx = X[i][0]; Y[i][0] = sx + Y[i-1][0];
        for(j=1; j<width; j++) {
              sx += X[i][j]; Y[i][j] = sx + Y[i-1][j];}}

// Scan+* (1 passe, résultats entrelacés)
  byte I[height][width], f; int C[height][2*width], sf,sff;
  f   = I[0][0]; sf  = f;  C[0][0] = sf;
  sff = f*f;  C[0][1] = sff;
  for(j=1; j<width; j++) {
    jj = j+j; f = I[0][j]; sf += f; sff += f*f;
    C[0][jj] = sf; C[0][jj+1] = sff;}
  for(i=1; i<height; i++) {
    f   = I[i][0]; sf  = f; sff = f*f;
    C[i][0] = C[i-1][0] + sf ; C[i][1] = C[i-1][1] + sff;
    for(j=1; j<width; j++) {
      jj = j+j; f   = I[i][0]; sf  = f; sff = f*f;
      C[i][jj  ] = C[i-1][jj  ] + sf ;
      C[i][jj+1] = C[i-1][jj+1] + sff; }}
```