Parallélisation d'opérateurs de TI : multi-cœurs, Cell ou GPU ?

Antoine Pédron[1], Florence Laguzet[2], Tarik Saidani[3], Pierre Courbin[3], Lionel Lacassagne[1,3], Michèle Gouiffès[3]

1. *CEA, LIST, Gif sur Yvette F-91191*
   antoine.pedron@cea.fr, lionel.lacassagne@cea.fr
2. *Laboratoire de Recherche en Informatique, Université Paris Sud, Orsay, F-91405*
   florence.laguzet@lri.fr
3. *Institut d'Électronique Fondamentale, Université Paris Sud, Orsay, F-91405*
   tarik.saidani@u-psud.fr, pierre.courbin@u-psud.fr, michele.gouiffes@u-psud.fr

RÉSUMÉ. *Cet article présente une évaluation des performances de 14 architectures actuelles : 8 processeurs généralistes, 5 GPU ainsi que le processeur Cell. L'algorithme retenu est l'opérateur de détection de points d'intérêt de Harris car représentatif des algorithmes de traitement d'images régulier bas niveau. À travers différentes transformations algorithmiques et optimisations logicielles adaptées à chaque architecture, cet article guide l'utilisateur dans le choix d'une architecture parallèle et évalue l'impact des optimisations afin d'avoir une implantation en adéquation avec l'architecture selon deux critères : le temps de calcul et la consommation énergétique.*

ABSTRACT. *This article deals with a performance analysis of current architectures : 8 general purpose processors, 5 graphic processing units and the Cell processor. The Harris salient points detection operator has been selected as representative of low level image processing. Through the presentation of algorithmic transforms and software optimization tuned to each architecture, the goal of this paper is to guide the user in chosing a parallel architecture and to assess how different transformations impact so as to get an architecture optimized implementation for two constraints: execution time and power consumption.*

MOTS-CLÉS : *parallélisation/vectorisation d'opérateur, processeurs généralistes, Cell, GPU, transformations algorithmiques, adéquation algorithme-architecture, opérateur de Harris.*

KEYWORDS: *parallelisation/vectorization, general purpose processors, Cell, GPU, algorithmic transforms, algorithm-architecture adequacy, Harris point operator*

DOI:10.3166/TS.27.161-187

Extended abstract

The goal of this article is to evaluate and compare the performance of current High Performance platforms through the optimization of a representative image processing algorithm and to guide the user in choosing an architecture to asses how different transformations impact so as to get an architecture optimized implementation for two constraints: execution time and power consumption. Eight SIMD multi-core processors, five mobile and desktop GPUs and the Cell processor have been evaluated. The Harris salient points detection operator has been selected as it is composed of point-to-point operator and convolution kernels. Classical software optimizations (loop unrolling, register rotation, register scalarization, circular buffers, cache optimization) and algorithmic transforms (convolution separation, factorization, operator fusion) are combined and tuned to each architecture. The resulting speedup can reach a factor 90 for a State-of-the-Art 8-core SIMD processor and the execution time can be as low as 0,11 ms (about 9000 images per second). Considering the power consumption, the article shows that generalist processors are more efficient than GPU.

1. Introduction

Le but de cet article est de fournir des pistes pour guider le concepteur d'algorithmes de traitement d'images dans le choix d'une architecture et de son implémentation efficace. Face à l'évolution des processeurs généralistes (GPP) – duplication des cœurs d'une part et généralisation du SIMD d'autre part – et face à l'apparition de nouvelles architectures proposant d'autres modèles de calcul (Cell, GPU), il est nécessaire de se poser la question de leur efficacité respective. La multitude d'architectures logicielles performantes nécessite de les comparer entre elles et de réaliser une adéquation algorithme-architecture avant d'envisager le recours à une architecture spécialisée ou dédiée (Zhang *et al.*, 2008).

Lorsqu'une architecture spécialisée (FPGA, SoC, rétine, ...) est développée en traitement d'images, elle doit répondre à des critères antagonistes de puissance de calcul et de consommation électrique (Lacassagne *et al.*, 2008) ; (Klein *et al.*, 2005). Les personnes en charge d'un tel développement mettent en œuvre toutes leurs connaissances afin d'obtenir les meilleures performances possibles. Notre contribution dans cet article est de faire de même pour des architectures logicielles en présentant les techniques d'optimisations les plus efficaces pour différentes classes architecturales et de mettre en évidence un certain nombre de points positifs ou négatifs des architectures utilisées.

Cette analyse pourra aussi servir à alimenter une réflexion portant sur le choix d'un processeur embarqué. Leur architecture étant très proche (pouvant être multicœur ou SIMD), les mêmes techniques d'optimisation peuvent leur être appliquées. Une synthèse des processeurs embarqués est disponible dans (Ventroux *et al.*, 2010).

Afin de réaliser cette analyse, l'algorithme choisi est l'opérateur de détection de points d'intérêt de Harris. Très utilisé dans le domaine de l'embarqué (stabilisation,

tracking), il est représentatif du traitement d'image régulier bas niveau. La première section détaille l'ensemble des transformations algorithmiques et optimisations logicielles appliquées à l'opérateur de Harris. La seconde présente les architectures cibles et les outils associés. La troisième section présente un *benchmark* évaluant l'impact des optimisations et la dernière section compare les architectures d'un point de vue énergétique.

2. Algorithme de Harris, transformations algorithmiques et optimisations logicielles

L'opérateur de Harris (Harris *et al.*, 1988) est représentatif du traitement d'images bas niveau : il est composé d'un ensemble d'opérateurs ponctuels (multiplications et additions point-à-point) et de convolutions (gradients de Sobel et lisseurs gaussien). De plus, les quatre phases de calculs qui le composent dans son implantation classique dite *Planar* en font un bon candidat à des optimisation de plus haut niveau (fusion d'opérateurs).

Cette section décrit les optimisations classiques en compilation (Allen *et al.*, 2002) pour le Calcul Haute Performance. Ces techniques sont ensuite adaptées aux opérateurs de traitement d'images bas niveau en prenant en compte leur spécificité comme la séparabilité des filtres 2D et le recouvrement des convolutions. Enfin leur combinaison est évaluée en termes de complexité.

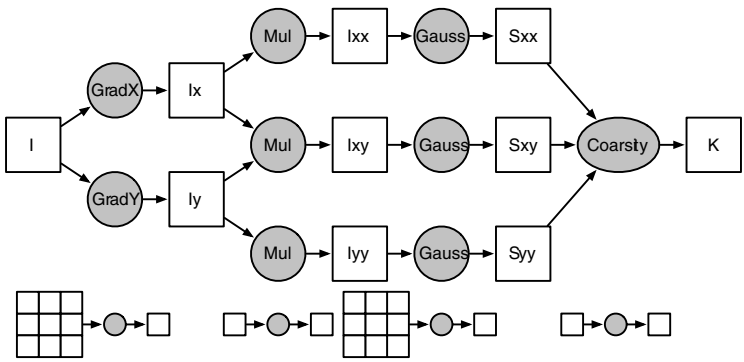

Figure 1. Opérateur de Harris, version Planar

2.1. Transformations et optimisations classiques

Le déroulage de boucle (*loop unrolling*) est une transformation classique pour les compilateurs optimisants (Allen *et al.*, 2002) qui a pour but principal d'améliorer le fonctionnement du pipeline du processeur. Il en va de même pour le déroulage avec mélange (*unroll and jam*) qui s'attaque au problème de la dépendance de données. Si l'ordre de déroulage est quelconque dans le cas d'un code généraliste et peut être choisi par l'utilisateur ou le compilateur, en traitement du signal et des images, l'ordre optimal de déroulage est égal à la taille de l'opérateur. C'est le seul à

permettre une mise en registres (*scalarisation*) efficace qui va permettre d'éviter le rechargement des données à la prochaine itération du filtre (Demigny, 2001).

Cette technique peut être efficacement combinée à une autre qui est propre à ce domaine applicatif : la factorisation des calculs *via* la séparabilité des filtres. Il est fréquent que les filtres utilisés soient des filtres 2D séparables en deux filtres 1D – un vertical et un horizontal. Par exemple le filtre binomial 3×3 B_3 est séparable et peut s'écrire comme la convolution d'un filtre 3×1 par un filtre 1×3 (équation 1). Dans le cas général, une convolution 3×3 comportant 9 multiplications et 8 additions est remplacée par deux convolutions de 3 multiplications et 2 additions.

$$B_3 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \qquad [1]$$

Algorithme 1. Implantation optimisée du filtre binomial 3×3 *(sans gestion des bords)*

Entrées : image $\mathbf{X}$ de taille $n \times n$
Sortie : image $\mathbf{Y}$ de taille $n \times n$

1 **for** $i = 1$ **to** $n - 1$ **do**
2 $\quad x_{a0} \leftarrow X(i-1,0), x_{a1} \leftarrow X(i,0), x_{a2} \leftarrow X(i+1,0), \Rightarrow r_a \leftarrow 1x_{a0} + 2x_{a1} + 1x_{a2}$
3 $\quad x_{b0} \leftarrow X(i-1,1), x_{b1} \leftarrow X(i,1), x_{b2} \leftarrow X(i+1,1), \Rightarrow r_b \leftarrow 1x_{b0} + 2x_{b1} + 1x_{b2}$
4 $\quad$ **for** $j = 1$ **to** $n - 1$ **step 3**
5 $\quad\quad x_{c0} \leftarrow X(i-1,j+1), x_{c1} \leftarrow X(i,j+1), x_{c2} \leftarrow X(i+1,j+1) \Rightarrow r_c \leftarrow 1x_{c0} + 2x_{c1} + 1x_{c2}$
6 $\quad\quad \mathbf{Y(i,j+0)} \leftarrow \mathbf{1r_a + 2r_b + 1r_c}$
7 $\quad\quad x_{a0} \leftarrow X(i-1,j+2), x_{a1}, \leftarrow X(i,j+2), x_{a2} \leftarrow X(i+1,j+2) \Rightarrow r_a \leftarrow 1x_{a0} + 2x_{a1} + 1x_{a2}$
8 $\quad\quad \mathbf{Y(i,j+1)} \leftarrow \mathbf{1r_b + 2r_c + 1r_a}$
9 $\quad\quad x_{b0} \leftarrow X(i-1,j+3), x_{b1}, \leftarrow X(i,j+3), x_{b2} \leftarrow X(i+1,j+3) \Rightarrow r_b \leftarrow 1x_{b0} + 2x_{b1} + 1x_{b2}$
10 $\quad\quad \mathbf{Y(i,j+2)} \leftarrow \mathbf{1r_c + 2r_a + 1r_b}$

L'étape suivante consiste à optimiser l'application des deux convolutions 1D. Plutôt que de les appliquer en deux passes sur l'image, elles sont appliquées en une seule passe. Le résultat de la première convolution est mémorisé dans un registre (*scalarisation*), c'est ce qu'on appelle une *réduction*. Un déroulage de boucle d'ordre k permet d'obtenir k valeurs réduites et d'enchaîner, pour chaque nouvelle valeur réduite, l'application de la seconde convolution. L'implantation optimisée (déroulage d'ordre 3 + réduction par colonne) d'un filtre gaussien (binomial) 3×3 est décrite dans l'algorithme 1. On notera la rotation des coefficients r_a, r_b et r_c (Algo. 1, lignes 6, 8 et 10) qui jouent chacun alternativement le rôle de colonne de gauche, colonne centrale et colonne de droite. Grâce à la *scalarisation* et à la *réduction par colonne* il est possible de diminuer d'un facteur 2, à la fois, le nombre d'accès mémoire (passant de 10 à 4) et le nombre de calculs par point (passant de 14 à 7) (tableau 1,

lignes Gauss et Gauss + réduction). À noter que dans l'algorithme 1, les multiplications par 1 sont volontairement laissées afin de faire apparaître les coefficients des filtres horizontaux et verticaux. Dans une implantation réelle, les multiplications par 1 sont supprimées et les multiplications par des puissances de 2 remplacées par des décalages en calcul entier et par des additions en calcul flottant.

2.2. Fusion d'opérateurs

La fusion d'opérateurs est une transformation qui va au delà de la fusion de boucle (pour améliorer la localité des données) et permet en pipelinant des opérateurs et en stockant les résultats du premier opérateur dans des registres (et non en

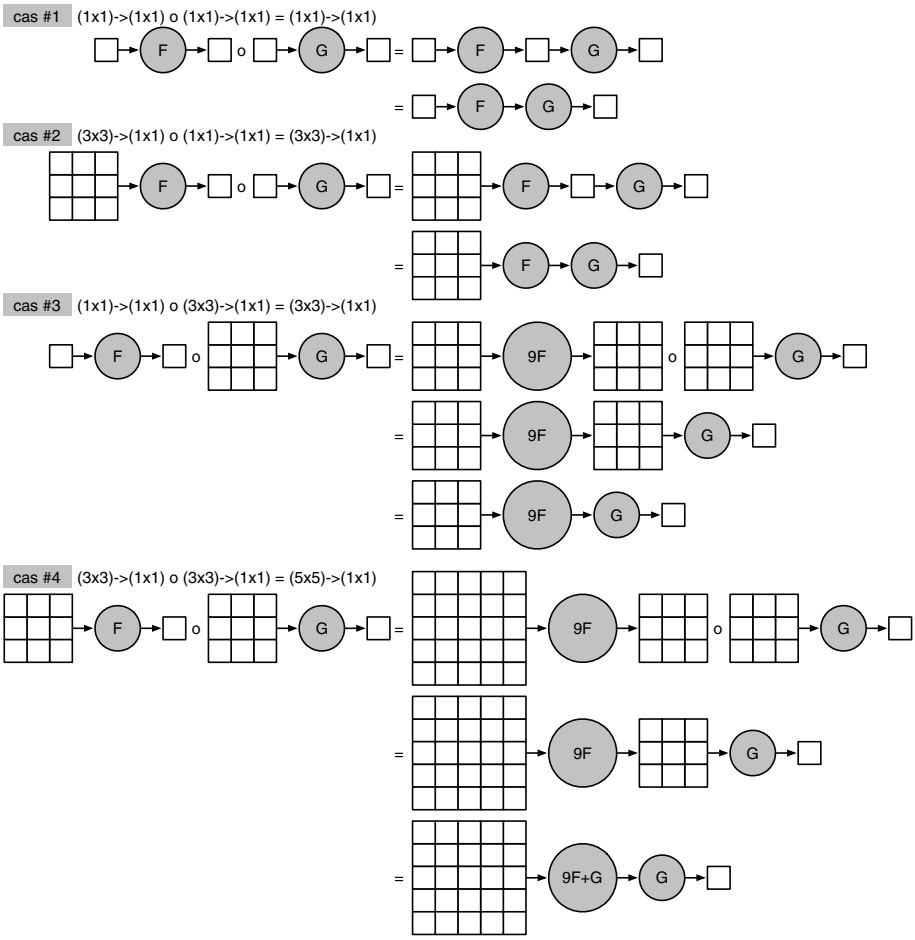

Figure 2. Quatre cas de fusion d'opérateurs ponctuels et de convolutions

mémoire) de complètement faire disparaître les accès mémoire intermédiaires. Pour cela, chaque opérateur est décrit par le modèle *producteur-consommateur* avec un motif de consommation de données et un motif de production. Ce type de modèle s'inspire des travaux des équipes Array-OL (Demeure *et al.*, 1995) et SDF (Lee *et al.*, 1987) (Lee, 1993). La fusion d'opérateurs peut être vue comme un pipeline temporel dans des registres. Pour fusionner deux opérateurs (au sens de la composition mathématique de fonctions $f \circ g$) il suffit que les motifs soient identiques ou adaptables. La figure 2 représente les quatre principaux cas de fusion d'opérateurs en traitement d'images. À noter la profonde différence entre la seconde et la troisième fusion : bien que les motifs d'E/S soient identiques, il y a 9 duplications de F dans le second cas (liées à l'adaptation du motif).

2.3. Opérateur de Harris, transformations Halfpipe *et* Fullpipe

Dans sa version *Planar*, l'opérateur de Harris est composée de quatre phases distinctes de calculs : calcul des gradients, produits des gradients, lissage des produits et finalement calcul de la *coarsité* (granularité du point).Cet enchaînement des calculs implique l'accès à huit mémoires intermédiaires (figure 1).

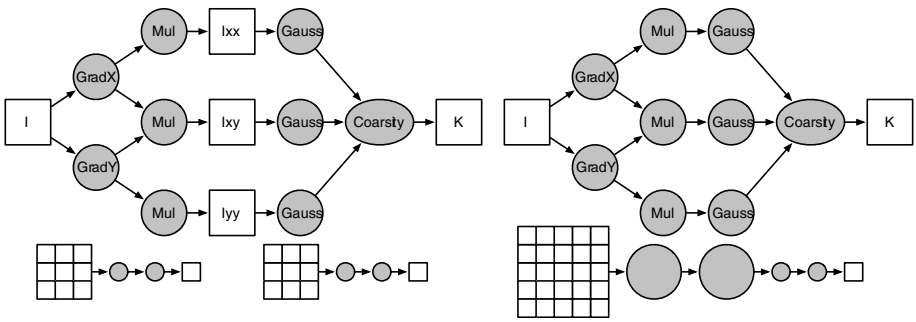

Figure 3. Transformations Halfpipe et Fullpipe de l'opérateur de Harris

Il est ainsi possible de pipeliner les opérateurs Sobel et Mul d'une part et les opérateurs Gauss et coarsité d'autre part, car leurs motifs de consommation et production mémoire sont compatibles. Cette transformation est appelée *Halfpipe* (figure 3) et permet de diminuer le nombre d'accès mémoire (tableau 1). Il est aussi possible de pipeliner entièrement les opérateurs et d'éliminer totalement les accès mémoires intermédiaires : c'est la version *Fullpipe*. Dans ce cas il est alors nécessaire d'adapter les motifs de consommation et production (figure 2, quatrième partie) : pour produire (1×1) point en sortie, il est nécessaire d'en consommer (5×5) et d'exécuter (3×3) fois l'ensemble Sobel+Mul. Une *réduction* par colonne est alors appliquée pour optimiser à la fois la quantité d'accès mémoire et le nombre de calcul.

Notons aussi que la version *Fullpipe* est bien plus complexe que les autres : il y a jusqu'à 9 fois plus de calculs que dans la version *Planar* car sans les images intermédiaires qui servent à mémoriser des calculs communs (qui étaient réutilisés

9 fois par les opérateurs gaussien), ces calculs doivent être refaits. Par contre, une fois optimisées (réduction par colonne), les versions *Halfpipe* et *Fullpipe* ont des complexités de calcul très proches (tableau 1), tandis que la version *Fullpipe* a un nombre d'accès mémoire bien moindre. La version *Fullpipe* est particulièrement intéressante d'un point de vue parallélisation car son ratio calculs / accès mémoire en fait un bon candidat *computation-bound* alors que les deux autres versions sont de type *memory-bound*.

Tableau 1. Complexité de l'opérateur de Harris : nombre d'opérations et d'accès mémoire par point pour les versions Planar*,* Halfpipe *et* Fullpipe*, avec et sans réduction par colonne*

opérateur	consommation → production	LOAD + STORE	MUL + ADD
Sobel	$2 \times (3 \times 3) \rightarrow 2 \times (1 \times 1)$	18 + 2	6 + 10
Mul	$3 \times (2 \times 1) \rightarrow 3 \times (1 \times 1)$	6 + 3	3 + 0
Gauss	$3 \times (3 \times 3) \rightarrow 3 \times (1 \times 1)$	27 + 3	18 + 24
Coarsité	$3 \times (1 \times 1) \rightarrow 1 \times (1 \times 1)$	3 + 1	2 + 1
Planar		**54 + 9**	**29 + 35**
Sobel *réd*	$1 \times (3 \times 1) \rightarrow 2 \times (1 \times 1)$	3 + 6	4 + 6
Mul *réd*	$3 \times (2 \times 1) \rightarrow 3 \times (1 \times 1)$	6 + 3	3 + 0
Gauss *réd*	$3 \times (3 \times 1) \rightarrow 3 \times (1 \times 1)$	9 + 3	9 + 12
Coarsité *réd*	$3 \times (1 \times 1) \rightarrow 1 \times (1 \times 1)$	3 + 1	2 + 1
Planar réd		21 + 9	18 + 19
Sobel+Mul	$1 \times (3 \times 3) \rightarrow 3 \times (1 \times 1)$	9 + 3	9 + 10
Gauss+Coarsité	$3 \times (3 \times 3) \rightarrow 1 \times (1 \times 1)$	27 + 1	20 + 25
Halfpipe		36 + 4	29 + 35
Fullpipe	$1 \times (5 \times 5) \rightarrow 1 \times (1 \times 1)$	25 + 1	101 + 115
Sobel+Mul *réd*	$1 \times (3 \times 1) \rightarrow 3 \times (1 \times 1)$	3 + 3	7 + 6
Gauss+Coarsité *réd*	$3 \times (3 \times 1) \rightarrow 1 \times (1 \times 1)$	9 + 1	11 + 13
Halfpipe réd		**12 + 4**	**18 + 19**
Fullpipe réd	$1 \times (5 \times 1) \rightarrow 1 \times (1 \times 1)$	**5 + 1**	**32 + 31**

2.4. Optimisations mémoires : entrelacement des données et buffers circulaires

L'entrelacement des données consiste à remplacer les accès à plusieurs tableaux différents par des accès à un tableau unique contenant les données des tableaux précédents (Truong *et al.*, 1998). Dans le cas de l'opérateur de Harris, cela consiste à entrelacer les tableaux produits aux mêmes étapes de calcul à savoir I_x et I_y pour l'opérateur Sobel, I_{xx}, I_{xy} et I_{yy} pour les opérateurs Mul et S_{xx}, S_{xy} et S_{yy} pour les

opérateurs gaussiens. Cette optimisation est particulièrement efficace lorsqu'elle permet de manipuler un nombre de tableaux inférieur à l'associativité du cache, évitant ainsi des défauts de cache systématiques. Ces défauts de cache sont dus au fait que les tableaux ayant la même taille et étant alignés en mémoire, les cases (i, j) de deux tableaux ont la même adresse en cache (problème de modulo).

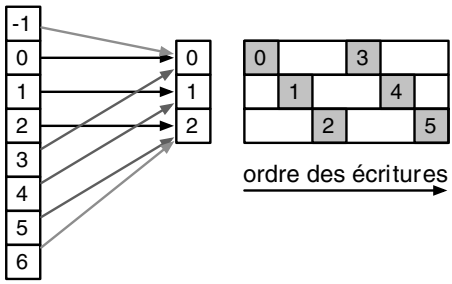

Figure 4. Ensemble de 3 buffers circulaires pour une image de 6 lignes et un bord d'image de taille 1, pour une convolution 3×3

Les buffers circulaires sont très utilisés dans le monde de l'embarqué (RISC, DSP) car ils permettent à la fois de diminuer la quantité de mémoire (Mamalet *et al.*, 2007) nécessaire pour stocker des résultats intermédiaires et d'améliorer le fonctionnement des caches. Le principe est, pour une chaîne de traitement de type flots de données, d'enchaîner les différents opérateurs en ne stockant en mémoire que les données intermédiaires nécessaires à l'exécution en séquence de ces opérateurs. Dans le cas de la version *Halfpipe*, plutôt que d'appliquer les opérateurs Sobel et Mul à l'image entière, ces opérateurs sont d'abord appliqués à 2 lignes de l'image (prologue). Puis, à partir de la troisième ligne, les deux premiers opérateurs ont produit suffisamment de données pour que les deux derniers puissent commencer à consommer des données : les deux groupes de deux opérateurs sont alors enchainés. Ce mécanisme est implanté sous forme d'un ensemble circulaire de 3 buffers de ligne (figure 4). La ligne i, résultat d'une convolution 3×3 appliquée aux lignes $(i - 1)$, i et $(i + 1)$ est stockée dans la ligne $(i \bmod 3)$ du buffer.

3. Architectures cibles et outils associés

Trois classes architecturales de machines sont testées (tableau 2) : les GPP (*General Purpose Processor*) multicœurs SIMD PowerPC et x86, les GPU Nvidia et le Cell pour les processeurs hétérogènes à mémoire distribuée. Au total, ce sont 14 architectures qui ont été évaluées. La consommation électrique indiquée est celle fournie par le constructeur (*Thermal Dissipation Power*). Pour chaque classe d'architecture, nous présentons les outils utilisés et l'impact attendu des optimisées de la section précédente.

Tableau 2. Principales caractéristiques des machines évaluées

processeur	référence	nb cœurs	techno (nm)	fréq (GHz)	cache (Mo)	TDP (W)
PowerPC G4	PPC4470	1	130	1,0	512 Ko	10
bi PowerPC G5	PPC970MP	2×2	90	2,5	2×512 Ko	70
Penryn	U9300	2	45	1,0	3	10
Penryn	P8700	2	45	2,53	3	25
Penryn	T9600	2	45	2,8	6	35
Yorkfield	Q9550	4	45	2,8	12	95
bi Yorkfield	X3370	2×4	45	3,0	2×12	190
bi Nehalem	X5550	2×4	45	2,67	2×8	260
Cell	CellXR8i	8	65	3,2	8×256 Ko	70
GeForce	9400M	16	55	1,110	-	12
GeForce	8600M GT	32	65	0,900	-	60
GeForce	120 GT	32	55	1,400	-	50
GeForce	FX 4600	112	65	1,400	-	155
GeForce	285 GTX	240	55	1,460	-	183

3.1. Processeurs généralistes (GPP)

Deux familles de processeurs ont été évaluées : PowerPC et Intel, regroupant des processeurs pour portables, station de travail et serveur de calcul. Les compilateurs utilisés sont ICC 11.0 et XLC 9.0.

Pour une machine multicœur SIMD, il y a deux niveaux de parallélisme. Tout d'abord au sein d'un processeur c'est le parallélisme de type SIMD ou plus précisément SWAR (*SIMD with A Register*) qui est présent dans tous les processeurs GPP ainsi que dans les processeurs embarqués haute performance. Les calculs étant faits en flottants 32 bits, il y a un parallélisme de 4 dans l'utilisation de registres 128 bits. Ce parallélisme est de mieux en mieux pris en compte par les compilateurs optimisants. La raison est qu'il est plus facile de vectoriser des codes flottants que des codes entiers, car les opérations flottantes d'addition et de multiplication ne nécessitent pas de changement de format en flottant (32 bits $\times$ 32 bits $\rightarrow$ 32 bits, 32 bits $+$ 32 bits $\rightarrow$ 32 bits). Pour être optimale, la vectorisation automatique doit être combinée avec un déroulage de boucle d'ordre égal à celui du filtre, ce qui n'est pas toujours réalisé. Le second niveau est l'aspect multicœur (et bientôt *many-cores*) des processeurs. Le modèle de programmation associé est un parallélisme de threads.

Pour paralléliser un code séquentiel existant, hormis quelques modifications éventuelles de l'algorithme du fait du découpage en bande, plusieurs options sont offertes au programmeur. Une première solution consiste à découper, soi-même, le travail qui devra être réalisé par chaque cœur. La bibliothèque Pthreads est une référence pour cela. Le programmeur a ainsi la possibilité de gérer lui même chaque création, destruction, lancement et synchronisation des threads qu'il utilise. Cette méthode demande un temps conséquent et une modification assez profonde du code séquentiel d'origine. La seconde solution est l'utilisation d'OpenMP qui rend plus transparente la gestion des threads. OpenMP a l'avantage d'être peu destructif pour le code séquentiel. Il se compose principalement de *pragmas* placés avant les parties à paralléliser. Cette facilité de programmation représente un gain de temps très important. Dans le cas d'une parallélisation SPMD et d'un découpage en bande de calcul, les modifications consistent à paralléliser les boucles externes et à privatiser les variables (pour éviter la sérialisation des accès). Dans le cas le plus simple, la parallélisation d'une boucle `for` utilise une directive (`#pragma omp parallel for`) non destructive pour le code séquentiel : si le code est compilé avec OpenMP la boucle sera parallélisée, mais la directive sera simplement ignorée comme un commentaire si le code est compilé sans.

En utilisant OpenMP, il n'est pas possible d'avoir une implantation simple de p ensembles de k buffers circulaires. La raison est que sur les bords de chaque bande de calcul, $k-1$ lignes sont calculées deux fois. Si la ligne i participe aux calculs de 2 bandes, elle doit être mémorisée dans autant de buffers différents, ce qui est impossible par un système d'indexation simple. Cela implique de profondes transformations de code allant bien au delà de l'ajout d'un `#pragma`. Le double pointeur sur chaque image doit être remplacé par p structures (préalablement initialisées) contenant le contexte de chaque thread (l'espace d'itération des boucles) et un ensemble de k buffers circulaires. La facilité et la souplesse d'utilisation d'OpenMP sont perdues et la complexité du code s'approche alors de celle d'un code à base de threads où les découpages en bandes et où les synchronisations sont explicites et réalisés par l'utilisateur.

L'impact attendu de ces transformations est le suivant. Soit une courbe de fonctionnement en *cpp* (figure 5a) et un point de fonctionnement correspondant à la taille des données à traiter. Comme le *cpp* est une métrique normalisée par la taille des données, un *cpp* horizontal indique que le temps d'exécution (comprenant les calculs et les accès mémoire) est strictement proportionnel à la taille des données. Une augmentation du *cpp* traduit donc des accès mémoire plus lents. Un phénomène transitoire apparaît sur la courbe (avant un retour à un *cpp* plus ou moins horizontal) lorsqu'on passe d'un état où les données tenaient dans les caches à un état où les données n'y tiennent plus. Nous appelons ce phénomène une *sortie de cache*.

La loi d'Amdhal exprime qu'en dupliquant d'un facteur p le nombre de processeurs, l'accélération globale sp sera proportionnelle à p et à τ, la fraction de code séquentielle non parallélisable (figure 5c). Dupliquer le nombre de caches revient à décaler vers la droite la courbe de fonctionnement. Les performances sont meilleures (figure 5b). Bien évidemment, caches et processeurs sont dupliqués en même quantité et c'est la combinaison de ces deux effets qui est observée (figure 5d) pour les processeurs multicœurs. Le *multi-threading*, de par son utilisation d'une plus

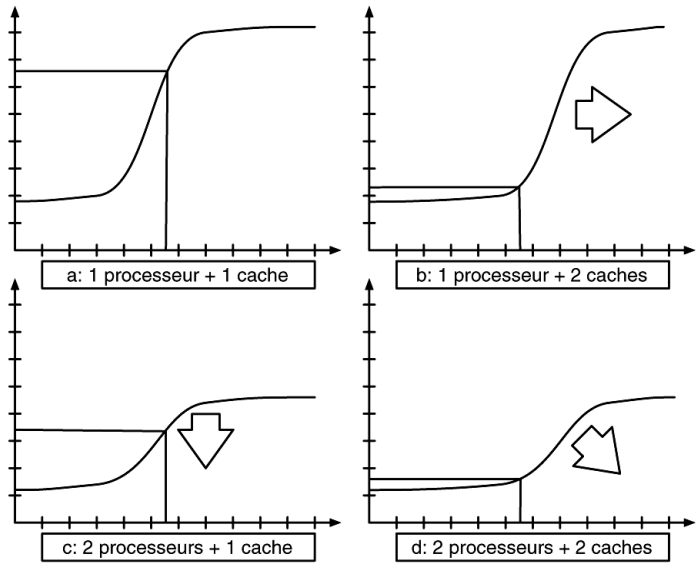

Figure 5. Impact attendu des transformations : évolution du point de fonctionnement en fonction du nombre de cache, des sorties de cache et du nombre de processeurs

grande quantité de mémoires caches, peut être vu comme un moyen de retarder les sorties de caches et donc de repousser la chute des performances.

À noter que les optimisations mémoires peuvent conduire à des accélérations sur-linéaires, supérieures au maximum théorique (fixé par les différents parallélismes), lorsque, pour un point de fonctionnement donné, la version basique ne tient plus en cache (la sortie de cache ayant eu lieu) et que la version optimisée y tient toujours.

3.2. Cell

Le processeur Cell peut être vu comme un processeur multicœur SIMD (nommés SPE) où les mémoires caches sont remplacées par des mémoires privées de 256 Ko (*scratchpad*) alimentées par un contrôleur DMA dont la bande passante totale est de 205 Go/s en interne entre les SPEs et de 25 Go/s en externe. Deux compilateurs sont disponibles : GCC qui accepte les instructions SIMD et la programmation par threads et CBEXLC qui supporte la parallélisation *via* OpenMP, mais pas la programmation SIMD. Afin d'utiliser les deux sources de parallélisme, c'est donc GCC qui est utilisé, la vectorisation et la parallélisation étant réalisées manuellement.

Le parallélisme mis en œuvre est de type SPMD : l'image est découpée en 8 bandes (une par SPE), chacune d'elles étant découpée en tuiles. Afin de ne pas avoir à synchroniser les SPE durant les calculs et d'éviter les transferts entre SPE, chaque SPE exécute l'ensemble des quatre opérateurs composant Harris. Afin de produire des tuiles de bonne taille en sortie du SPE, les tuiles consommées en entrée sont dotées de bords de taille 2 qui seront consommées par les deux convolutions 3×3

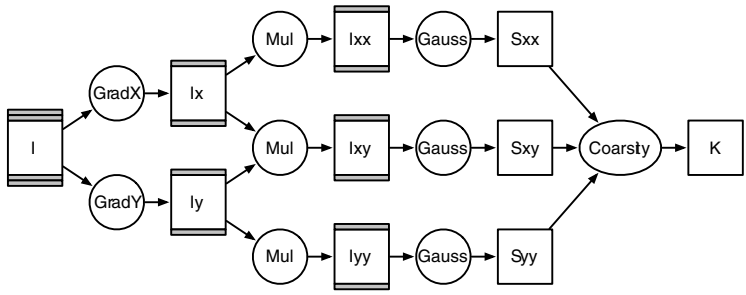

Figure 6. Harris sur le processeur Cell : convolutions et gestion des bords

(comme pour la version *Fullpipe* sur multi-cœurs). La programmation du Cell est toutefois simplifiée par le développement d'un un *middleware* d'encapsulation des transferts DMA nommé *Cell-MPI* (Saidini *et al.*, 2008).

3.3. GPU

Si les GPU sont souvent considérés comme rapides dans de nombreux benchmarks, notamment ceux présentés sur la page web CUDA de Nvidia, il y a parfois des ambiguïtés sur les mesures réalisées : les performances prennent-elles en compte les temps de transferts entre la machine hôte CPU et la carte (dans un sens puis dans l'autre) ? Et la comparaison est-elle faite avec un code scalaire non multi-threadé ? Si les GPU sont très performants dans le domaine du calcul scientifique (Volkov *et al.*, 2008a) où la nature des calculs se prête bien à une parallélisation massive (Garland *et al.*, 2008), il est important et il reste nécessaire d'évaluer objectivement leurs performances en traitement d'images et de les comparer à celles des autres classes d'architecture.

En CUDA, un *kernel* est une fonction de calcul exécutée sur le GPU. Il correspond à un ensemble de threads organisés sous la forme d'une grille de blocs. Les blocs de threads sont distribués dans les multiprocesseurs (*streaming multiprocessors*) à la manière d'une liste FIFO. Chaque multiprocesseur a ses propres ressources et les blocs ne peuvent communiquer entre eux. Un bloc est lui-même décomposé en groupes de 32 threads (appelés *warps*) qui sont exécutés de manière séquentielle au sein du multiprocesseur. De par son fonctionnement, un multiprocesseur s'apparente à une unité SIMD sur GPP. L'*overhead* lié à la création des threads ne dépasse pas 15 μs, synchronisation de fin comprise (Volkov *et al.*, 2008b).

La programmation efficace d'un GPU nécessite la compréhension fine de l'exécution des threads sur un multiprocesseur et du type d'accès aux différentes mémoires disponibles (tableau 3). La mémoire partagée sera utilisée pour les opérations ponctuelles, les convolutions nécessitant de la mémoire de texture. La mémoire globale est utilisée pour toutes les autres opérations car c'est la seule mémoire visible depuis le CPU et le kernel, en lecture et en écriture.

Tableau 3. Caractéristiques des mémoires des GPU Nvidia G80 et GT200

Mémoire	Emplacement	Cachée	Accès	Visibilité
Registre	on-chip	non	lecture/écriture	un thread
Locale	off-chip	non	lecture/écriture	un thread
Partagée	on-chip	n/a	lecture/écriture	tous les threads d'un block
Globale	off-chip	non	lecture/écriture	tous les threads + hôte
Constante	off-chip	oui	lecture/écriture	tous les threads + hôte
Texture	off-chip	oui	lecture/écriture	tous les threads + hôte

La mémoire globale est disponible en grande quantité. Elle est accessible depuis le CPU ainsi que depuis les kernels. Afin d'utiliser au maximum la bande passante, les accès doivent respecter des contraintes d'alignement et de voisinage. Le *scatter/gather* est supporté uniquement en enchainant séquentiellement les accès non voisins. Chaque accès en mémoire globale coûte de 400 à 800 cycles mémoire. Si l'ensemble des threads d'un demi *warp* accèdent à un voisinage et respectent les règles d'alignement le coût sera le même que pour un seul accès. Un accès respectant ces règles est appelé coalescent. Sachant que c'est la seule mémoire dans laquelle on puisse écrire en sortie de kernel, il est impératif de respecter ces règles d'accès afin d'éviter de trop grandes latences d'accès.

Chaque multiprocesseur possède 16 Ko de mémoire partagée (visible par les 8 PE le composant mais privée vis-à-vis de l'extérieur du kernel). Elle n'est visible que pour un bloc de threads et permet d'effectuer des opérations locales en profitant d'une latence d'accès proche de celle des registres. Son utilisation pour le calcul d'un opérateur de convolution est très intéressante si l'on gère correctement les bords et que l'on respecte les règles d'accès. L'accès se faisant *via* 16 bancs mémoire distincts, il est nécessaire que chaque thread d'un demi *warp* adresse des bancs différents afin d'exécuter l'accès en un seul cycle. À noter que la mémoire locale n'est qu'une abstraction mémoire exprimant la visibilité locale d'un thread avec ses voisins du même bloc.

Enfin, la mémoire de texture est accessible par un kernel uniquement en lecture et passe par un cache (8 Ko par multiprocesseur). L'hôte doit préalablement lier la texture (*bind*) à une zone de mémoire globale. Sur le GT200, il est possible de lier directement une texture à une zone mémoire standard, mais il est recommandé d'utiliser les `cudaArray` qui permettent d'optimiser la cohérence des caches et donc d'obtenir de meilleures performances. La contrepartie est qu'il faut allouer de la mémoire pour ce tableau et copier la zone mémoire *via* une primitive dédiée, ce qui induit des transferts supplémentaires. La mémoire de texture est la plus adaptée aux convolutions (Podlozhnyuk, 2007) car elle possède un mécanisme matériel d'interpolation bilinéaire qui autorise l'utilisation de nombre flottant comme index. De plus elle possède deux modes d'adressages : circulaire (pour le plaquage de texture en infographie) et *clamping* très utile pour éviter la duplication manuelle des bords de l'image.

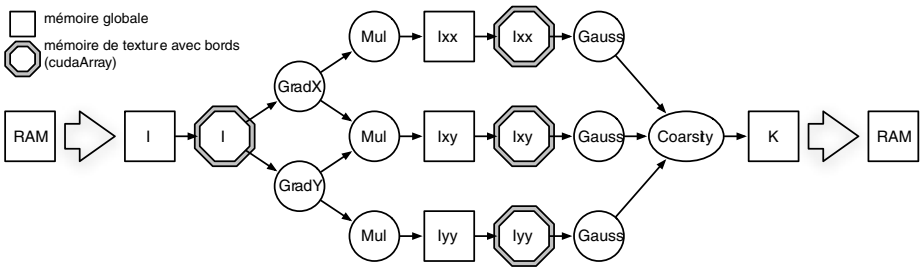

Figure 7. Halfpipe2 *sur GPU : mémoire globale, mémoire de texture et gestion des bords*

4. Benchmark #1 : vitesse

L'unité de mesure utilisée pour ce benchmark est le *cpp* (Cycle par Pixel), c'est le temps de calcul ramené en cycles et normalisé par le nombre de points traités : $cpp = t \times F/n^2$. Dans le cas des images, n représente le coté d'une image. Les performances ont été évaluées pour des tailles d'images allant de 128×128 à 2048×2048. Afin d'évaluer l'impact des différentes optimisations dans le gain total, plusieurs mesures ont été réalisées (tableau 4), en scalaire et en SIMD, avec ou sans parallélisation, avec ou sans optimisation mémoire. Les résultats sont donnés en *cpp* pour des images 512×512. Le meilleur *cpp*, obtenu par combinaison de toutes les optimisations est indiqué en gras ainsi que le gain total associé. Ces résultats sont ensuite analysés dans les sections suivants qui détaillent les performances et les gains des transformations pour chaque classe d'architecture.

4.1. Processeurs généralistes GPP

4.1.1. Impact des optimisations mémoire

La première étape a consisté à évaluer l'impact des optimisations mémoire. L'entrelacement des données n'a pas d'impact majeur sur les performances à moins d'être utilisé seul. De plus il nécessite une modification des allocations mémoires, ce qui peut être problématique dans le cas d'applications industrielles existantes (*code legacy*) où seul le code peut être modifié.

L'implantation de la version *Halfpipe* mono-thread (sans réduction par colonne) met en évidence que **les buffers circulaires permettent de diminuer fortement l'amplitude des sorties de cache**. Le gain est d'autant plus grand que le débit des caches est « faible ». Il est d'environ $\times 1{,}5$ pour le Yorkfield et de $\times 1{,}2$ pour le Nehalem (figure 8). De plus le *cpp* de la version avec buffers après la sortie de cache est approximativement égal au *cpp* de la version sans buffer avant la sortie de cache. Cette observation permet d'extrapoler l'impact des buffers circulaires combinés à une parallélisation automatique avec OpenMP. Les mesures suivantes sont donc réalisées sans buffer circulaire, une estimation de leur impact étant fournie à titre indicatif.

Tableau 4. Résultats en cpp *sans et avec parallélisation (multi-threading), en scalaire et SIMD pour des images* 512×512

archi	prog	*Planar* mono	*Planar* multi	*Halfpipe* mono	*Halfpipe* multi	gain	temps (ms)
G4	scalaire	518		248		×7,1	19,2
	SIMD	189		**73,4**			
G5	scalaire	254	79	35	15	×87,6	0,32
	SIMD	79	43	8,9	**2,9**		
Penryn U9300	scalaire	152,0	94,0	40,2	23,6	×12,9	2,58
	SIMD	35,6	29,4	13,8	**11,8**		
Penryn P8700	scalaire	151,0	88,3	34,8	22,6	×6,8	2,30
	SIMD	56,8	56,6	24,0	**22,2**		
Penryn T9600	scalaire	140,7	81,6	32,0	16,1	×35,0	0,44
	SIMD	53,7	51,6	8,4	**4,7**		
Yorkfield	scalaire	140	38	45	11,0	×22,2	0,59
	SIMD	48,0	11,0	20,0	**6,3**		
bi-Yorkfield	scalaire	145	16,9	32	4,3	×90,6	0,14
	SIMD	55,0	3,6	8,4	**1,6**		
bi-Nehalem	scalaire	49	7,2	24,5	3,4	×35,0	0,11
	SIMD	18,6	3,3	6,0	**1,4**		
Cell	scalaire	857	402	199	140	×214,3	0,33
	SIMD	79,5	12,6	29,8	**4,0**		
GeForce 9400M	scalaire	27,6		21,9		×1,3	5,18
GeForce 8600M	scalaire	11,7		7,7		×1,5	2,23
GeForce GT120	scalaire	10,2		6,9		×1,5	1,30
Quadro FX4600	scalaire	5,9		4,3		×1,4	0,81
GeForce GTX285	scalaire	2,2		1,5		×1,5	0,26

4.1.2. Impact de la taille des caches

Le second point à été d'évaluer l'impact de l'augmentation de la taille des caches. Concernant le Yorkfield (figure 9, les graphes sont à la même échelle permettant une comparaison directe), les pertes d'efficacité qui, pour un thread, débutaient pour des images de taille 380×380 ne commencent qu'à partir de 550×550 avec 4 threads pour la version *Planar*. Pour la version *Halfpipe*, la sortie de cache glisse de 500×500 à 800×800. Cette version est plus efficace (et plus longtemps) que la version *Planar*. Grâce à son plus petit nombre d'accès mémoire, la transformation *Halfpipe* peut être vue comme une façon de prolonger les performances des caches. Idem pour le bi-Yorkfield : les sorties de caches des courbes *Planar* et *Halfpipe* ont

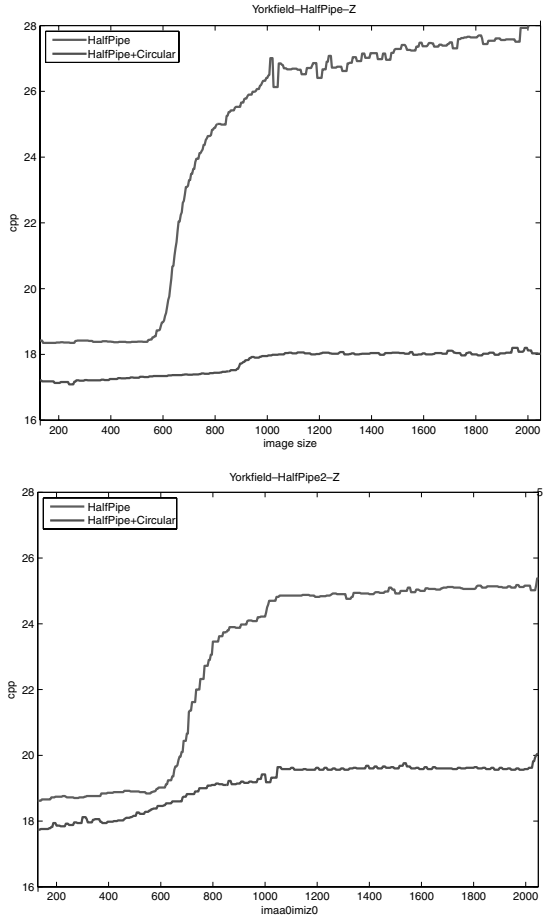

Figure 8. Diminution de l'amplitude des sorties de cache grâce aux buffers circulaires, version Halfpipe SIMD mono-thread, Yorkfield et Nehalem

lieu en 650 × 650 et 800 × 800 en monocœur et en 1000 × 1000 et 1400 × 1400 en octocœurs (figure 10). Pour le Nehalem, les sorties de caches sont respectivement en 400 × 400 et 512 × 512 en monocœur et en 600 × 600 et 850 × 850 en multicœurs. Les différences sont dues à la quantité de mémoire cache totale de ces machines (24 Mo pour le Yorkfield contre *seulement* 16 pour le Nehalem). **Les multicœurs permettent, par l'augmentation de la quantité de mémoire cache utilisée, une amélioration significative des performances pour les images de grande taille.**

4.1.3. Impact de la vectorisation SIMD et de la parallélisation

De manière générale, **la vectorisation et la parallélisation donnent des gains significatifs proches du maximum** théorique (respectivement ×4 pour le SIMD et ×4 ou ×8 pour OpenMP, en fonction du nombre de cœurs, tableau 5). Le gain est

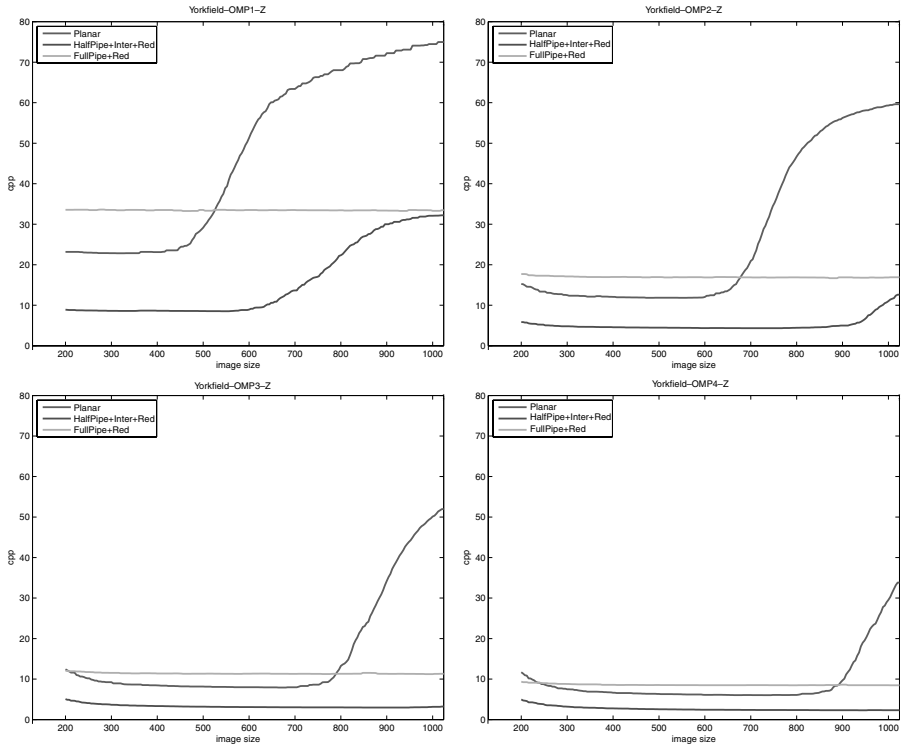

Figure 9. Décalage vers la droite des sorties de cache grâce à l'augmentation de la quantité de mémoire cache utilisée, Yorkfield SIMD+OpenMP, 1 à 4 cœurs

Tableau 5. Gains de la vectorisation SIMD et de la parallélisation OpenMP

	SIMD			OpenMP			SIMD + OpenMP		
taille	512	1024	2048	512	1024	2048	512	1024	2048
bi-Yorkfield (octocœur)									
Planar	×2,8	×2,8	×2,3	×8,8	×8,8	×2,8	×41,6	×41,6	×2,8
Halfpipe	×4,4	×4,4	×1,1	×7,4	×7,4	×1,5	×20,0	×20,0	×25,2
Fullpipe	×4,0	×4,0	×3,8	×8,0	×8,0	×8,1	×32,1	×32,1	×32,3
bi-Nehalem (octocœur)									
Planar	×2,4	×2,0	×2,0	×7,0	×5,2	×4,8	×14,3	×5,9	×5,0
Halfpipe	×4,2	×2,7	×2,7	×7,6	×6,7	×5,9	×20,8	×20,9	×21,1
Fullpipe	×3,6	×3,6	×3,6	×7,7	×7,6	×7,6	×27,3	×27,6	×27,0

même sur-linéaire dans certains cas, lorsque la version de base ne tient plus dans le cache et que la version optimisée y tient toujours (*Halfpipe*+OpenMP en 1024 sur Yorkfield).

Concernant les versions *Planar* et *Halfpipe*, le niveau de performance reste lié à celui des caches : lorsque les images sont trop grandes, les performances chutent. La sortie de cache ne dépendant que de la taille des caches de la machine. Par contre, comme indiqué précédemment, l'amplitude de la chute dépend de la bande passante du cache. Ainsi pour le Yorkfield, dans la configuration SIMD+OpenMP, il y a un facteur de décélération supérieur à 10 (17,4 pour la version *Planar* et 16,4 pour la version *Halfpipe*) lorsqu'on passe d'une image 512×512 à 2048×2048. Ce facteur n'est seulement que de 3 pour le Nehalem.

4.1.4. Impact des transformations haut niveau Halfpipe *et* Fullpipe

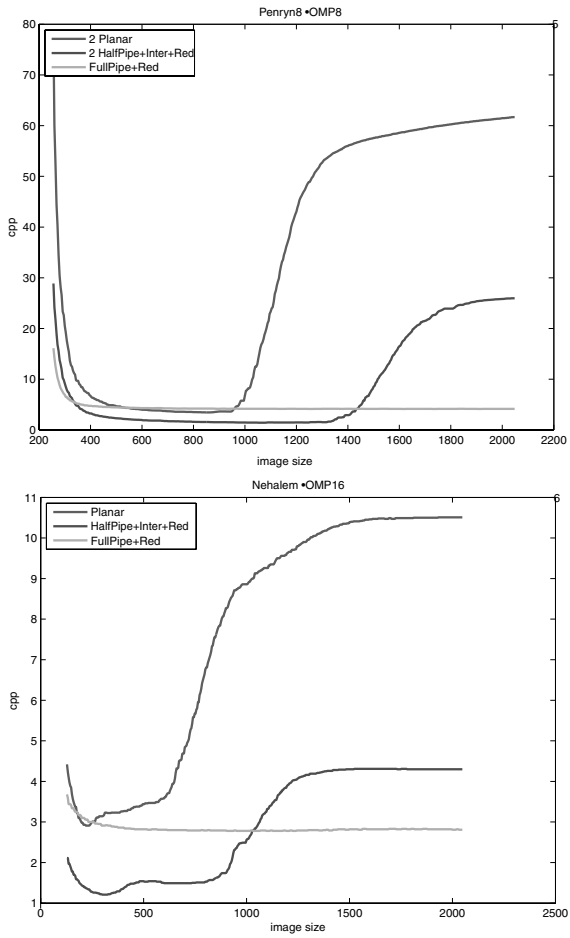

Figure 10. Prédominance de la version Halfpipe *puis* Fullpipe*, cpp des versions octocœurs SIMD, Yorkfield et Nehalem*

Tableau 6. Gains des différentes versions Halfpipe *et* Fullpipe *avec SIMD+OpenMP, Yorkfield et Nehalem*

processeur	Yorkfield			Nehalem		
taille	512	1024	2048	512	1024	2048
cpp des versions sans buffer circulaire						
Planar	3,6	49,3	62,5	3,3	8,9	10,5
Halfpipe	1,6	1,6	26,2	1,4	2,4	4,3
Fullpipe	4,2	4,2	4,2	2,7	2,7	2,8
gains associés						
Halfpipe	×2,3	×**30**,**8**	×2,4	×2,4	×**3**,**7**	×2,4
Fullpipe	×0,9	×11,7	×**19**.**9**	×1,2	×3,3	×**3**,**8**
cpp et gain extrapolés dune version *Halfpipe* avec buffers circulaires						
Halfpipe	1,6	1,6	1,6	1,4	1,4	1,4
Halfpipe	×2,25	×30,0	×**39**,**9**	×2,4	×6,4	×**7**,**5**

Les transformations algorithmiques *Halfpipe* et *Fullpipe* ont un impact majeur sur les performances (tableau 6). Sur le Yorkfield, le gain en (1024×1024) est particulièrement grand (sur-linéaire) et s'explique par le fait que la version *Halfpipe* tient toujours en cache contrairement à la version *Planar* (figure 10). Sur le Nehalem, le gain est plus faible car la sortie de cache de la version *Halfpipe* a déjà eu lieu (en 800×800).

Puisque la bande passante totale (celle des deux processeurs) est partagée par l'ensemble des cœurs, diminuer la quantité de données transférée peut être efficace, quitte à augmenter la quantité de calcul. C'est ce que fait la version *Fullpipe* qui est *computation-bound* (figure 10, *cpp* constant). Les calculs n'étant plus ralentis par les transferts de données, à partir d'un *multi-threading* de 2 sur Yorkfield (et de 4 sur Nehalem), la version *Fullpipe* devient plus rapide que la version *Halfpipe* pour les images de grande taille – après la sortie de cache de la version *Halfpipe*. Ainsi, sur le Yorkfield (tableau 6), la version *Fullpipe* qui était 2.6 fois plus lente que la version *Halfpipe* en 1024×1024 devient $\times 6.25$ fois plus rapide en 2048×2048. Sur le Nehalem, le gain est plus faible – seulement $\times 1{,}5$ – car la plus grande bande passante de la machine rend la version *Halfpipe* moins lente. À noter que des résultats similaires seraient *a priori* aussi obtenus avec une version *Halfpipe* combinée à des buffers circulaires (tableau 6, valeurs extrapolées en italique), les accélérations seraient plus grandes (car la version *Halfpipe* est plus rapide que la version *Fullpipe* avant la sortie de cache) et croissantes.

4.1.5. Conclusions pour les multicœurs SIMD

Ce benchmark a mis en lumière plusieurs points importants. Tout d'abord, la vectorisation et la parallélisation sont efficaces sur les machines multicœurs SIMD et

s'il est encore nécessaire d'écrire manuellement du code SIMD, la parallélisation *via* OpenMP est simple à mettre en œuvre tout en étant performante. Concernant la fusion d'opérateurs, les transformations *Halfpipe* et *Fullpipe* apportent des gains supplémentaires conséquents. Comme ces transformations restent hors de portée des meilleurs compilateurs actuels (Pouchet *et al.*, 2008), leur codage manuel est justifié. Une alternative à la version *Fullpipe* serait l'utilisation de multi-buffers circulaires compatibles avec OpenMP dont la génération automatique est un enjeu important (Han *et al.*, 2006). Plus généralement, ces résultats renforcent notre stratégie de benchmark systématique d'un certain nombre d'implantations potentiellement efficaces d'un algorithme donné pour *toutes* les tailles de données possibles (dans ce cas, de 128×128 à 2048×2048). À titre d'information, 11 implantations différentes furent évaluées en scalaire et 21 en SIMD et ce, sur 8 machines différentes.

4.2. Le Cell

Différentes implantations ont été évaluées en détail dans (Saidani *et al.*, 2008), seuls les résultats de la version *Halfpipe* sont analysés ici. Les SPE du Cell ayant été conçus pour faire du calcul SIMD et non du calcul scalaire, les versions scalaires sont toutes inefficaces, le gain global $\times 214$ n'est donc pas significatif. La version *Halfpipe* est plus rapide car il y a, à la fois, moins de calculs (réduction par colonne), moins de transferts et moins d'accès mémoire (en mémoire privée *local store*). L'aspect *memory bound* s'observe par le mauvais passage à l'échelle de la version *Planar* par rapport à la version *Halfpipe* : $\times 3{,}6$ contre $\times 7{,}5$. Le point fort du Cell réside dans son absence de cache, qui permet de maintenir le même niveau de performances, quelle que soit la taille des images (validé jusqu'à des images 4096×4096). Ces résultats, sont toutefois en retrait par rapport à une machine généraliste octocœurs et s'expliquent en partie par les limitations du contrôleur DMA qui ne peut transférer rapidement que des tuiles très allongées ce qui conduit à une grande quantité de rechargement mémoire pour gérer les bords.

4.3. Les GPU

Trois versions ont été implantées. La première est une version *Planar* et se décompose en quatre étapes (comme pour les GPP) et en quatre kernels CUDA. Comme indiqué, les convolutions (Sobel et Gauss) se font – à cause des accès au voisinage du point considéré – depuis la mémoire de texture vers la mémoire globale. Des copies mémoires dans des `cudaArray` sont donc réalisées. Pour les opérateurs ponctuels (Mul et Coarsité), les calculs se font en mémoire globale *via* des accès rapides coalescents. La seconde version, *Halfpipe2*, regroupe les calculs en deux étapes et en deux kernels CUDA, ce qui permet d'éviter les accès (écriture puis lecture) en mémoire globale se situant entre les convolutions et les opérateurs ponctuels, mais qui ne permet toujours pas d'éviter les deux recopies en mémoire de texture (figure 7).

Afin d'évaluer l'impact des recopies en mémoire de texture des deux premières versions, une troisième version – *Halfpipe1* – qui est fausse a été codée. Elle regroupe en un seul kernel les quatre opérateurs et n'utilise pas de mémoire de

texture. Au sein d'un bloc de threads, les deux premiers opérateurs produisent les résultats intermédiaires I_{xx}, I_{xy} et I_{yy} qui sont écrits en mémoire partagée. Après synchronisation, ces résultats sont rechargés par les mêmes threads pour produire K. Les blocs initiaux ne contenant pas de bord, il manque deux lignes et deux colonnes aux résultats intermédiaires et quatre au résultat final. Il est possible d'ajouter une gestion particulière des bords, mais cela diminue tellement la vitesse de traitement que cela rend cette version, initialement la plus rapide des trois, la plus lente de toutes. Grâce à sa gestion transparente des bords, la mémoire de texture devient alors plus efficace (Podlozhnyuk, 2007).

Tableau 7. cpp de calcul des GPU pour des images 512×512

version	*Planar*			*Halfpipe2*			*Halfpipe1*		
T = Transferts; Calc = Calculs; S = Somme T+C									
GPU	T	**Calc**	S	T	**Calc**	S	T	**Calc**	S
9400M	11,5	**27,7**	39,2	11,5	**21,9**	33,4	5,4	**7,9**	13,3
8600M	11,0	**12,4**	23,3	11,0	**8,1**	19,1	7,1	**4,6**	11,6
GT120	7,9	**10,2**	18,1	7,9	**6,9**	14,9	4,9	**3,6**	8,5
FX4600	9,5	**5,9**	15,4	9,6	**4,3**	13,9	8,3	**1,6**	9,9
GTX 285	10,5	**2,2**	12,7	10,2	**1,5**	11,7	9,6	**0,6**	10,2

Du point de vue de la vitesse (en *cpp*), les GPU peuvent être classés en deux groupes (tableau 7). D'un côté il y a les GPU avec un petit nombre de PE pour lesquels le temps de calcul est supérieur ou égal au temps de transfert. Un mécanisme de double tampon pourrait permettre de masquer les transferts et donc de doubler les performances de ces GPU. De l'autre, il y a les GPU (*many-cores*) avec un grand nombre de PE qui représentent la tendance *Calcul Haute Performanc*e. Pour ceux là (GTX 285 en *Halfpipe2*), les calculs sont jusqu'à 7 fois plus rapides que les transferts. Ce ratio atteint un facteur $\times 16$ pour la version *Halfpipe1*. Le bus PCIe est donc un goulet d'étranglement pire que le bus FSB des anciens processeurs Intel. On peut observer que les transferts sont plus rapides pour le GT120 qui est la seule carte interfacée avec un Macintosh doté d'un port PCIe 2.0. Ce phénomène empirera avec l'augmentation du nombre de cœurs (les Fermi seront dotés de 480 cœurs), le passage au PCIe 2.0 puis 3.0 ne permettra de compenser ce ratio, que d'un facteur $\times 2$ (respectivement $\times 4$). Deux alternatives sont possibles. La première est de considérer des algorithmes plus complexes, avec une plus grande quantité de calculs par point, ou d'enchaîner l'exécution de plusieurs algorithmes. Cela n'est pas simple car tous les algorithmes d'une chaîne de traitement peuvent ne pas être massivement parallélisables sur GPU. Les algorithmes de traitement d'images de moyen niveau sont par définition irréguliers et sont rarement de bons candidats à la parallélisation. La seconde alternative consiste à court-circuiter le PCI et à rapprocher le GPU de la RAM et du CPU. C'est ce qui est déjà fait sur certaines consoles de jeux. Pour les systèmes embarqués ce serait faire un *streaming SoC*, le GPU étant au plus proche d'un imageur rapide.

4.4. Comparaison et analyse globale des résultats

Il apparaît que les GPP, grâce à l'amplitude des gains (l'efficacité cumulée des transformations algorithmes et optimisations logicielles) sont les plus rapides. Les deux premières places reviennent aux octocœurs Yorkfield et Nehalem. Le Cell se classe en quatrième position, en étant 2,9 fois plus lent que le Nehalem. En ne prenant en compte que le temps de calcul, le plus rapide des GPU arrive en troisième position. Mais en prenant en compte le temps de transfert, le temps total pour le GTX 285 est multiplié par 8 et passe à 2,1 ms. Cela ramène les performances d'une machine *many-cores* à celle d'un processeur *dual core*. Concernant les GPU mobiles, l'écart se creuse, car même en ne prenant en compte que le temps de calcul, le 8600M GT et ses 32 PE n'est qu'au niveau du Penryn P8700 qui n'est plus dans sa zone d'efficacité (sortie de cache). Comparé au Penryn T9600 qui n'a pas encore eu de sortie de cache, il est alors 5 fois plus lent.

5. Benchmarks #2 : efficacité énergétique

Tableau 8. Efficacité énergétique pour des images 512×512

Archi	Techno (nm)	Puissance (W)	Temps (ms)	Énergie (mJ)
PowerPC G4	130	10	19,24	192,4
PowerPC G5	90	2×70	0,32	44,0
Penryn U9300	45	10	2,58	25,8
Penryn P8700	45	25	2,30	57,5
Penryn T9600	45	35	0,44	**15,4**
Yorkfield	45	95	0,59	56,0
bi-Yorkfield	45	2×95	0,14	**26,6**
bi-Nehalem	45	2×130	0,11	29,7
Cell	65	70	0,33	**22,9**
GeForce 9400M	55	12	5,12	**62,2**
GeForce 8600M GT	80	60	2,11	126,6
GeForce GT120	55	50	1,30	64,8
Quadro FX4600	90	134	0,81	108,6
GeForce GTX285	55	183	0,26	**47,6**

Le second benchmark consiste à estimer l'énergie consommée ($E = t \times P$) en se basant le TDP constructeur. Les résultats sont intéressants et paradoxaux. Si c'est effectivement un processeur pour portable qui est le plus efficace (Penryn T9600), ce n'est pas celui qui consomme le moins (Penryn U9300). Ce dernier, avec un TDP de 10 W, fait jeu égal avec l'octo-processeur Yorkfield et un TDP de 190 W ! La seconde

meilleure performance revient au Cell. La performance du Cell vis-à-vis des processeurs généralistes (Intel, AMD et IBM) avait déjà été observée lors de précédents congrès *Super Computing*/Top500 avec l'apparition d'un classement de *Green Computing* : pour les grands besoins en puissance de calcul, le Cell est le modèle de calcul le plus efficace énergétiquement : les machines *Roadrunner*, composés de Cell, de GPP et de GPU sont actuellement parmi les plus efficaces (en MFlops/Watt). Concernant les GPU, ils arrivent tous en fin de classement, même si, seul le *cpp* de calcul est pris en compte. Le plus efficace étant paradoxalement celui qui consomme le plus (GTX 285).

5.1. De la relativité de la mesure

Tableau 9. Efficacité énergétique pour des images 300×300

Archi	Techno (nm)	Puissance (W)	Temps (ms)	Énergie (mJ)
Penryn U9300	45	10	0,360	**3,6**
Penryn P8700	45	25	0,157	3,9
Penryn T9600	45	35	0,148	5,2
bi-Yorkfield	45	2×95	0,048	9,1
bi-Nehalem	45	2×130	0,038	9,8
Cell	65	70	0,113	**7,9**

Il est intéressant d'observer l'efficacité énergétique de ces machines pour des images plus petites : 300×300. Dans ce cas, toutes les machines sont surdimensionnées : le bi-Yorkfield affichant une cadence de traitement de 26 315 images/sec et le « petit » Penryn U9300 une cadence de 2 777 images/sec. L'ordre de performance est maintenant respecté : le U9300 est le plus efficace et le Cell se positionnent devant les processeurs octocœurs.

Jusqu'à maintenant, les performances des machines étaient évaluées pour des tailles fixes d'images. Il peut être intéressant de prendre le problème à l'envers et de s'interroger sur l'intervalle de taille d'image pour lequel ces processeurs sont efficaces. Si l'efficacité du Cell est constante (absence de cache) et celle des GPU croît avec la taille des images (problème de l'alimentation des données), les GPP ne sont efficaces que tant que les données tiennent dans les caches (à moins d'implanter des buffers circulaires). Une configuration de type *serveur* est plus efficace (bus rapide) plus longtemps (2 processeurs quadricœurs Yorkfield ont un total de 24 Mo de cache L2) qu'une configuration *portable* (les dual cœurs ont entre 4 et 6 Mo de cache L2). En recalculant l'efficacité des Penryn pour une taille d'image 300×300 au lieu de 512×512, le classement des processeurs *efficaces* est modifié (Tab 9). Le Penryn U9300 qui était aussi efficace que le bi-Yorkfield devient alors 2,5 fois plus efficace.

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

6. Conclusions et perspectives

Ce article a présenté deux benchmarks comparant les performances de GPP, de GPU et du Cell pour du traitement d'images bas niveau régulier. À travers les métriques utilisées, il a été mis en évidence l'importance des transformations algorithmiques qui combinées aux instructions SIMD et à la parallélisation multicœur font que les processeurs généralistes restent compétitifs face aux nouvelles architectures (Cell et GPU) un facteur $\times 90$ a été atteint. Grâce à cela, ils dépassent les performances brutes du Cell. De plus, et contrairement au Cell, le nombre de cœurs des machines généralistes a réussi à croître rapidement : Intel et AMD annoncent des processeurs octocœurs et des machines bi ou quadri processeurs (soit un maximum de 32 cœurs). Certains processeurs pour serveur sont déclinés en version *basse-consommation*, ce qui les rend plus compétitifs encore, tout en maintenant un modèle de programmation simple et des temps de développement rapides.

Évaluer les performances des processeurs généralistes reste donc d'actualité et il sera intéressant de mesurer l'augmentation des performances due à l'augmentation de la taille des registres SIMD. Le processeur Sandy Bridge 256 bits et surtout le processeur Larrabee composé de 80 cœurs SIMD 512 bits (dont la sortie est retardée) sont très prometteurs. L'ère du *many-cores* généraliste aura alors commencé et la paire vectorisation/parallélisation sera la clé de la performance de ces machines. Un point important dans le choix d'une architecture est la facilité de développement et les outils disponibles. Cela est d'autant plus justifié, de notre point de vue, qu'IBM vient d'annoncer l'arrêt du Cell en commettant les mêmes erreurs que Texas Instrument pour le C80 : complexité de la programmation, manque d'outils et absence de couche d'abstraction efficace. Les outils et la facilité de programmation sont d'ailleurs les points forts des GPP : la programmation SIMD (Altivec ou SSE) est relativement simple et la parallélisation *via* OpenMP qui est à la fois simple et extrêmement efficace devrait permettre à cette API de se démocratiser pour le traitement d'images.

La prise en main des GPU et leur maîtrise (modèle de calcul, modèle mémoire) est relativement rapide et l'investissement plus léger que pour le Cell. Leur principal défaut est la faible bande passante du port PCIe et la gestion inefficace des bords (inutile en HPC mais nécessaire en traitement d'images). Ce type d'architecture deviendra compétitive lorsque ces problèmes seront résolus, par exemple sous forme de *streaming* SoC (bus rapide entre mémoire et GPU), l'interfaçage avec un bus PCIe 3.0 ne faisant que maintenir à l'identique le déséquilibre entre transferts et calculs. Il serait intéressant d'évaluer les performances des GPU sur des problèmes plus complexes. À défaut, en tant qu'accélérateur matériel pouvant être placé en quadruple exemplaire à l'intérieur d'un PC, ils se révèlent très utiles.

Les prospectives de recherches concernent l'implantation de versions *Halfpipe* avec buffers circulaires, l'évolution de Cell-MPI et l'évaluation des processeurs généralistes pour l'embarqué.

Bibliographie

Allen R., Kennedy K. (eds) (2002). *Optimizing compilers for modern architectures : a dependence-based approach*, Morgan Kaufmann, chapter 8,9,11.

Demeure A., Lafarge A., Boutillon E., Rozzonelli D., Dufourd J., Marro J. L. (1995). « Array-OL : Proposition d'un formalisme tableau pourle traitement de signal multidimensionnel », *GRETSI*.

Demigny D. (ed.) (2001). *Méthodes et Architectures pour le TSI en temps réel*, Hermes, chapter 10 : Optimisation logicielle pour processeurs VLIW.

Garland M., Grand S. L., Nickolls J., Anderson J., Hardwick J., Morton S., Phillips E., Zhang Y., Volkov V. (2008). « Parallel Computing Experiences with CUDA», *IEEE Micro*, vol. 28, p. 13-27.

Han S., Guerin X., Chaee S., Jerraya A. (2006). « Buffer memory optimization for video codec application modeled in Simulink », *Design Automation Conference*, p. 689-694.

Harris C., Stephens M. (1988). « A combined corner and edge detector », *4th ALVEY Vision Conference*, Éditions Hermès, Paris, p. 147-151.

Klein J.-O., L. Lacassagne H. M., Moutault S., Dupret A. (2005). « Low Power Image Processing : Analog versus Digital Comparison », *Computer Architecture Machine Perception*, IEEE.

Lacassagne L., Manzanera A., Denoulet J., Mérigot A. (2008). « High Performance Motion Detection : Some trends toward new embedded architectures for vision systems », *Journal of Real Time Image Processing*, october.

Lee E. (1993). « Multidimensional streams rooted in dataflow », *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism.*

Lee E., Messerschmitt D. G. (1987). « Synchronous Data Flow », Proceedings of the IEEE. Mamalet F., Roux S., Garcia C.(2007). « Real-time video convolutional face finder on embedded platforms », *Journal on Embedded Systems*, vol. 1, p. 22-29, January.

Podlozhnyuk V. (2007). Image Convolution with CUDA, technical report, NVIDIA, June.

Pouchet L.-N., Bastoul C., Cohen A., Cavazos J. (2008). « Iterative Optimization in the Polyhedral model : part II, multidimensional time », *ACM SIGPLAN Conference on Programming Lan-guage Design and Implementation (PLDI'08)*, p. 90-100.

Saidani T., Lacassagne L., Falcou J., Tadonki C., Bouaziz S. (2008). « Parallelization Schemes for Memory Optimization on the Cell Processor: A Case Study on the Harris Corner Detector », *HiPEAC journal.*

Truong D. N., Bodin F., Seznec A. (1998). « Improving cache behavior of dynamically allocated data structures », *Parallel Architectures and Compilation Techniques (PACT)*, ACM, p. 322-329.

Ventroux N., David R. (2010). « Les architectures parallèles sur puce : synthèse des architectures multitâches pour les systèmes embarqués », *Technique et Science Informatique*, vol. 29,3, p. 345-378.

Volkov V., Demmel J. (2008a). LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, May.

Volkov V., Demmel J. W. (2008b). « Benchmarking GPUs to tune dense linear algebra », *SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, Piscataway, NJ, USA, p. 1-11.

Zhang Q., Chen Y., Zhang Y., Xu Y. (2008). « SIFT implementation and optimization for multi-core systems », *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, p. 1-8.

Article reçu le 1/10/2009
Version révisée le 15/05/2010

Antoine Pédron *est un doctorant du CEA LIST/DISC de Saclay depuis 2009. Il a reçu son Master Professionnel en informatique de l'Université Denis Diderot en 2008. Il a ensuite effectué un pré-doc sur à l'Institut d'Électronique Fondamentale (IEF) de l'Université Paris-Sud. Son domaine de recherche principal est la parallélisation de code sur processeurs généralistes multicœurs et sur GPU. Ses travaux de thèse portent sur le développement d'algorithmes d'imagerie et de reconstruction pour des applications en contrôle non destructif.*

Tarik Saidani *est doctorant à l'Institut d'Électronique Fondamentale (IEF) de l'Université Paris Sud. Il a reçu son Master Recherche « Systèmes Embarqués et Traitement de l'Information » à l'Université de Paris-Sud en 2006. Ses thématiques de recherche portent sur le calcul hautes-performances et les architectures parallèles. Ses travaux de thèse ont porté sur les méthodologies d'optimisation d'algorithmes de traitement d'images pour les architectures parallèles : IBM Cell, Multi-core et GPU.*

Pierre Courbin *est doctorant au LISSI de l'Université Paris-Est, en association avec le laboratoire LACSC de l'ECE. Il a reçu son diplôme d'ingénieur informatique de l'ECE et son Master Recherche « Systèmes Embarqués et Traitement de l'Information » à l'Université Paris-Sud en 2009. Ses thématiques de recherche sont le calcul parallèle et l'ordonnancement temps-réel pour les multiprocesseurs et l'analyse de sensibilité.*

Florence Laguzet *est doctorante au Laboratoire de Recherche en Informatique (LRI) et à l'Institut d'Électronique Fondamentale de l'Université Paris-Sud. Elle a reçu son diplôme d'ingénieur en informatique de Polyech' Paris-Sud en 2009 ainsi que son Master Recherche en informatique. Son domaine de recherche est l'adéquation algorithme architecture appliquée à la parallélisation de codes irréguliers de traitement d'image et aux algorithmes de vision en temps réel. Ses recherches sont menées dans l'équipe Parall du LRI et dans le département AXIS (Architecture, Control, Communication, Vision, Systèmes) de l'IEF.*

Lionel Lacassagne *est ingénieur de recherche au CEA LIST dans le Laboratoire Calcul Embarqué du Département Architecture & Conception de circuits, Logiciels Embarqués (DACLE). Maître de Conférences à l'Université Paris Sud, dans l'équipe AXIS de l'Institut d'Électronique Fondamentale (IEF). Il a reçu son diplôme d'ingénieur en informatique industrielle et intelligence artificielle de l'EPITA en 1995, son DEA et son doctorat en robotique de l'Université Pierre et Marie Curie en 1996 et en 2000. Il est habilité à diriger des recherches depuis 2010. Ses thématiques de recherche sont l'Adéquation Algorithme Architecture appliquée au traitement d'images pour les systèmes embarqués ainsi que la conception d'algorithmes irréguliers pour les architectures haute performance. Il est expert en optimisation de codes pour DSP VLIW, architectures multicœurs SIMD, processeurs Cell et GPU.*

Michèle Gouiffès *est Maître de Conférence à l'Université Paris Sud depuis 2006. Elle a reçu son diplôme d'ingénieur en électronique et en informatique industrielle de l'INSA Rennes (Institut National des Sciences Appliquées) en 2002 et en 2005 son doctorat en traitement du signal de l'Université de Poitiers. Son domaine de recherche est la vision par ordinateur et l'analyse d'image. Elle s'intéresse particulièrement à l'extraction de primitives couleurs (points et lignes) ainsi qu'à leur robustesse vis-à-vis des changements d'illumination. Ses recherches portent aussi sur le matching de points d'intérêt pour le tracking et la stéréovision.*