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

Pierre COURBIN, Antoine PÉDRON, Tarik SAIDANI, Lionel LACASSAGNE

Institut d'Électronique Fondamentale – Digitéo Labs – Université Paris Sud
prenom.nom@u-psud.fr

Résumé – Cet article présente une évaluation des performances de huit architectures actuelles : processeur généraliste multicœur (GPP), carte graphique (GPU) et processeur Cell. L'algorithme retenu est l'opérateur de Harris. Le but est de guider l'utilisateur dans le choix d'une architecture parallèle et d'évaluer l'impact des différentes transformations afin d'avoir une implantation en adéquation avec l'architecture.

Abstract – This article deals with a performance analysis of eight current architectures : general purpose multi-core processors, graphic processing units and the Cell processor. The benched algorithm is the Harris operator. The main goal is to guide the user in chosing a parallel architecture and to assess how different transformations impact so as to get an architecture optimised implementation.

1 Introduction

Le but de cet article est de fournir des pistes pour guider les Traiteurs d'Images dans le choix d'une architecture pour l'implantation efficace d'opérateurs de TI. Face à l'évolution des processeurs généralistes (GPP), tels que l'ajout d'instructions SIMD ou la duplications des cœurs, et à l'apparition de nouvelles architectures proposant d'autres modèles de calcul (*Model of Computation*), il est nécessaire de se poser la question de leur efficacité respective lorsque les codes sont optimisés.

Lorsqu'une architecture spécialisée en TI (FPGA, SoC, rétine, ...) est développée, elle doit répondre classiquement à 2 critères antagonistes de puissance de calcul et de consommation électrique [3] [2]. Les personnes en charge d'un tel développement mettent en œuvre toutes leurs connaissances afin d'obtenir les meilleures performances possibles. L'idée est de faire de même pour des architectures logicielles, et que l'implantation optimisée d'algorithmes mette en évidence un certains nombre de points positifs ou négatifs (les problèmes classiques sont connus, mais pas précisément quantifiés). Cette analyse permettra d'alimenter une réflexion pour le choix d'un processeur spécialisé existant (Tile64 de Tilera, Imapcar de Nec ou le très attendu Larrabee d'Intel) ou pour la conception d'une nouvelle architecture.

L'algorithme choisi est l'opérateur de détection de points d'intérêt de Harris [1]. Il est représentatif du TI bas niveau car composé d'opérateurs ponctuels et de convolutions. Il est très utilisé dans le domaine de l'embarqué (stabilisation, tracking).

Les différentes transformations sont présentées dans la section 2. Les architectures évaluées sont les GPP (PowerPC et Intel), les GPU (Nvidia et ATI) et le Cell. Les résultats sont exposés dans la section 3.

2 Problématique et implantation

Nous décrivons dans cette section les transformations réalisées (algorithmiques, logicielles et architecturales), les outils utilisés et nous revenons sur l'intérêt général des multi-cœurs actuels.

2.1 Transformations

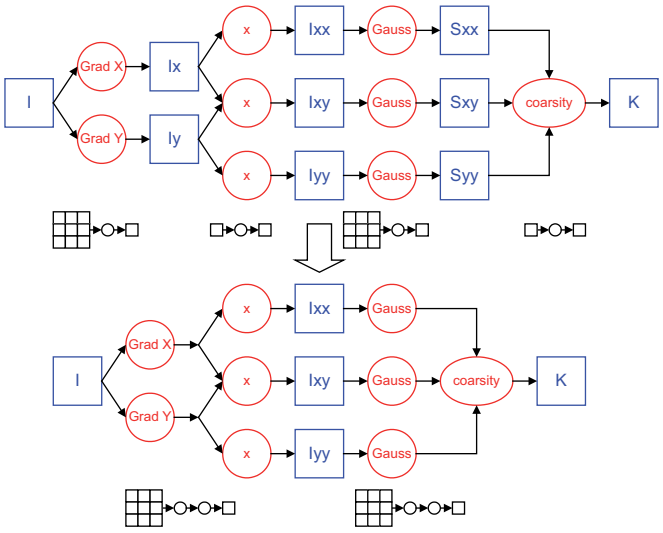

FIG. 1 – Transformation *Halfpipe* de l'opérateur de Harris

Les Transformations réalisées concernent différents niveaux d'abstraction et de connaissance de l'architecture.

– **Transformations logicielles** de type déroulage de boucle et entrelacement des calculs (*Unroll & Jam*) : pour augmenter la régularité des calculs dans le pipeline et « casser » les dépendances de données.

– **Transformations architecturales** :
  . utilisation d'instructions SIMD (SSE pour Intel et AMD, Altivec pour PowerPC, SPE pour Cell, et `float4` sur GPU) : pour augmenter la quantité de calculs réalisés par cycle,
  . entrelacement des données [8] : pour diminuer le nombre de références actives dans le cache (problème d'associativité) et donc les problèmes de défauts de cache.
– **Transformations algorithmiques** :
  . décomposition des filtres 2D (Sobel et Gauss) en filtres 1D : pour diminuer la quantité de calculs et d'accès mémoire.
  . **Chaînage d'opérateurs** : pour éliminer les accès (écriture puis lecture) à des zones mémoires intermédiaires, important pour les opérateurs dont la vitesse est limitée par la bande passante des bus (*memory bound problem*).

L'opérateur de Harris à été choisi car il met en évidence l'impact du chaînage d'opérateurs. étant composées de quatre phases successives de calculs (gradient, produits des dérivées premières, lissage des produits et coarsité), il implique dans son implantation classique, l'accès à huit mémoires intermédiaires (Fig. 1). Il est ainsi possible de chaîner les opérateurs Sobel et Mul d'une part et les opérateurs Gauss et Coarsité d'autre part, car leur motifs de consommation et production mémoire sont compatibles : $(3 \times 3) \rightarrow (1 \times 1)$ et $(1 \times 1) \rightarrow (1 \times 1)$. Cette transformation est appelée *Halfpipe*. Il est aussi possible de chaîner entièrement les opérateurs (Fig. 2) c'est la version *Fullpipe*. Dans ce cas il est alors nécessaire d'adapter les motifs de consommation et production : pour produire (1×1) point en sortie, il es nécessaire d'en consommer (5×5) et d'exécuter (3×3) fois l'ensemble Sobel+Mul. Notons aussi que la version *Fullpipe* est bien plus complexe que les autres : jusqu'à 9 fois plus de calculs.

2.2 Outils

Les outils utilisés pour les GPP sont d'une part les compilateur GCC 4.2 et ICC 11.0 pour les optimisations bas niveau, la bibliothèque *Pthreads* pour le *multi-threading* et OpenMP pour la parallélisation automatique. En combinant transformations et outils, il est possible d'avoir des versions *tout automatique* (utilisation d'OpenMP et activation des options de vectorisation) *semi-automatique* (codage en SIMD et parallélisation OpenMP) et *tout manuel* (codage SIMD et Pthreads). Pour le Cell un *middleware* d'encapsulation des transferts DMA nommé *Cell-MPI* a été développé [7]. Cet outil s'inspire des travaux réalisés pour le DSP C80 [4], les deux architectures étant très proches. Les GPU ont été programmé avec leur langage *ad hoc* : CUDA pour Nvidia et Brook+ pour ATI. Brook+ est particulièrement intéressant car propose un plus haut niveau d'abstraction (*Domain Specific Language*) que CUDA en proposant la notion de `kernel` et de `stream`.

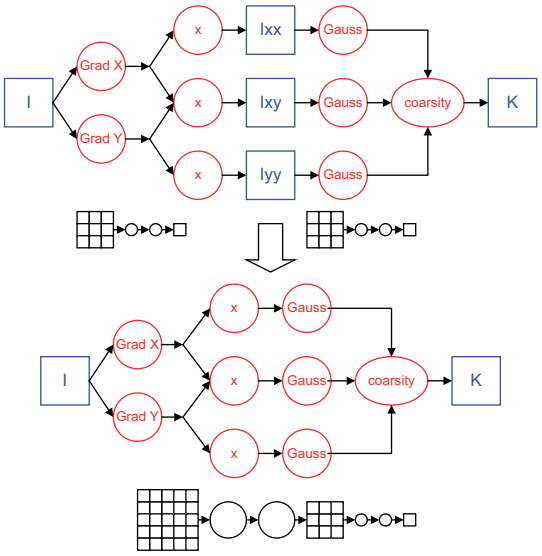

FIG. 2 – Transformation *Fullpipe* de l'opérateur de Harris

2.3 Intérêt des multi-cœurs

L'augmentation du nombre d'unités de calculs (cœurs, SPE) dans les processeurs s'est généralisée depuis quelques années. L'utilisation de la puissance ainsi mise à disposition peut apparaître comme un moyen relativement simple et rapide d'accélerer l'exécution d'un code. L'intérêt du multi-cœur est en fait double. Il permet évidemment d'améliorer les performances en diminuant le cpp. Si l'algorithme est « scalable » alors le cpp sera divisé par une valeur proche de p le nombre de processeurs (Tab. 2), c'est la loi d'Amhdal. Le second intérêt qui découle du premier est qu'en multipliant le nombre de processeur, la quantité de mémoire cache est aussi multipliée. Ainsi, l'utilisation conjointe de plusieurs cœurs ayant chacun un cache propre ou partagée peut permettre de repousser la limitation des performances due aux sorties de caches. (Fig. 3).

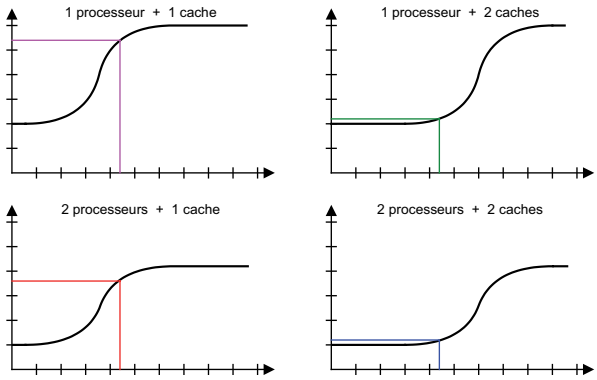

FIG. 3 – *évolution des sorties de cache en fonction de l'évolution du nombre de processeurs et de caches.*

3 Benchmarks et efficacités

3.1 Architectures testées

Nous avons effectué les benchmarks sur les processeurs suivants :

- PowerPC G4 à 1 GHz (PPC4470)
- Bi-dual cœurs PowerPC G5 à 2.5 GHz (PPC970MP)
- Dual cœurs Intel Conroe à 2.4 GHz (T7700)
- Quadri-cœurs Intel Penryn à 2.8 GHz (Q9550)
- Bi-quadri cœurs Intel Penryn à 3.0 GHz (X3370)
- Cell à 3.2 GHz
- Nvidia GeForce 8800 GTX à 575 MHz
- ATI HD4850 à 625 MHz.

3.2 Benchmarks & observations

TAB. 1 – Résultats en *cpp* sans / avec parallélisation, en scalaire et SIMD pour des images 512×512

archi	prog	*Planar*	*Halfpipe*	gain
G4	scalaire	518	248	×7
	SIMD	189	**73.4**	
G5	scalaire	254 / 79	35 / 15	×69
	SIMD	79 / 43	8.9 / **2.9**	
Conroe	scalaire	143.9 / 84.8	31.9 / 17.2	×12
	SIMD	52.9 / 52.3	13.5 / **11.8**	
Penryn	scalaire	140 / 38	45 / 11.0	×22
	SIMD	48.0 / 11.0	20.0 / **6.3**	
bi-Penryn	scalaire	145 / 16.9	32 / 4.3	×91
	SIMD	55.0 / 3.6	8.4 / **1.6**	
Cell	scalaire	857 / 402	199 / 140	×214
	SIMD	79.5 / 12.6	29.8 / **4.0**	
GeForce 8800GTX	scalaire	120	15	×8
ATI HD4850	scalaire	98.4	13.9	×7

Les performances ont été évaluées pour des tailles d'images allant de 128×128 à 2048×2048. Les résultats (Tab. 1) sont donnés en Cycle Par Point ou *cpp* ($cpp = t \times Freq/n^2$, avec n, le coté d'une image), en scalaire et en SIMD, sans et avec parallélisation pour chacune des architectures testées et pour des images 512×512. Le meilleur *cpp*, obtenu par combinaisons de toutes les optimisations, est indiqué en gras. « Planar » représente la version sans optimisations et « Halfpipe » la version associée à la Fig. 1.

Les résultats de parallélisation présentés ont été obtenus avec OpenMP. Le choix entre OpenMP et Pthread est présenté dans la section suivante.

3.2.1 Efficacité des transformations

Les transformations logicielles, architecturales et algorithmiques utilisées permettent de mettre en évidence les point forts et points faibles des diverses architectures étudiées. Les détails des transformations algorithmiques et logicielles ayant été abordées dans la partie 2, nous nous concentrerons ici sur les apports des transformations architecturales. Cependant, le tableau 1 permet d'ores et déjà de noter que les transformations algorithmiques et architecturales ont un impact majeur sur les performances (gains entre 2 et 8, suivant les architectures, entre la version planar et la version *halfpipe*) et restent, par nature, hors de portée d'un compilateur.

TAB. 2 – Impact de la vectorisation et du *multithreading* sur le bi-Penryn

	SIMD		OMP (8 threads)		OMP+SIMD	
Taille	512	1024	512	1024	512	1024
Planar	×2.69	×2.20	×8.76	×3.46	×41.11	×3.54
Halfpipe	×3.83	×1.17	×7.49	×9.19	×20.13	×24.13
Fullpipe	×3.52	×3.61	×7.92	×7.99	×27.44	×27.67

Le tableau 2 donne les gains apportés par les transformations architecturales (SIMD et *multithreading*) sur le bi-Penryn, en prenant chaque fois comme référence le code séquentiel scalaire de l'algorithme concerné. Les lignes représentent les différentes versions de l'algorithme.

Utilisation de la vectorisation
De manière générale, la vectorisation donne des gains significatifs. Cependant, cette optimisation doit être apportées de façon manuelle car les compilateurs actuels n'offrent pas des performances optimales. Sur les GPP testés, on peut constater ce bon comportement. En revanche, on est toujours face aux problèmes de sorties de cache qui, sur certaines architectures comme le Penryn, rendent les gains apportés médiocres.

Concernant le Cell, les gains apportés par la vectorisation par rapport au mode scalaire sont bien meilleurs que sur les GPP. Ceux-ci s'expliquent par les très mauvaises performances de ce mode. En effet, le jeu d'instructions SPU est exclusivement vectoriel. Exécuter une instruction scalaire induit l'utilisation de multiples instructions vectorielles ce qui implique forcément des temps de calculs plus important. Le Cell ne laisse donc pas le choix quant aux méthodes de développement à appliquer si l'on cherche la performance.

Nous pouvons remarquer qu'il y a une perte d'efficacité sur des images 1024×1024 pour les versions *memory bound* planar et *halfpipe*. L'efficacité de la vectoriation est donc liée à la taille des caches. Ainsi, au regard du temps de développement nécessaire, l'intérêt de cette optimisation se réduit pour des tailles d'images importantes.

Utilisation du multithreading
Pour rendre parallèle un code séquentiel existant, hormis quelques modifications éventuelles de l'algorithme du fait du découpage en bande, plusieurs options sont offertes au programmeur. Les notions suivantes ne sont applicables qu'aux cas des GPP et du Cell. 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'une interface de programmation rendant plus transparent la gestion des threads. L'interface que nous avons retenu est OpenMP, intégré aux compilateurs CBEXLC, ICC et GCC récents. 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.

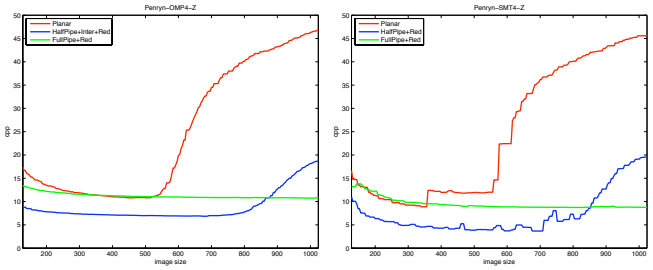

FIG. 4 – Comparaison des performances pour le *multithreading* entre l'utilisation d'OpenMP vs Pthreads

Les tests que nous avons mené montrent que les versions utilisant OpenMP restent plus stables que celles reposant sur la bibliothèque PThread (Fig. 4). Cependant, dans le cas du Cell, la version d'OpenMP intégrée à CBEXLC (compilateur *Single Source* pour le Cell) n'a pas permis d'utiliser conjointement les fonctions de vectorisation spécifiques aux SPEs. Ainsi, si le programmeur décide de vectoriser lui-même son code par l'utilisation de primitives spécifiques au Cell, il n'aura pas d'autre choix que de coder aussi lui-même la parallélisation à l'aide des Pthreads.

Finalement, pour les besoins les plus simples de parallélisation des instructions, il nous semble plus efficace de se tourner vers les outils existants tel qu'OpenMP. Les Pthreads sont à réserver aux cas particuliers tel que la parallélisation des codes SIMD sur le Cell.

3.2.2 Les GPP : *multithreading* et multi-cache

En accord avec les notions exposées dans la partie 2.3, le *multithreading*, de part son utilisation d'un plus grand nombre de caches, permet de retarder les sortie de cache. L'efficacité du code est ainsi prolongée pour de grande taille de données. Concernant le Penryn (Fig. 5), 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.

Les mêmes observations se retrouvent sur le processeur G5

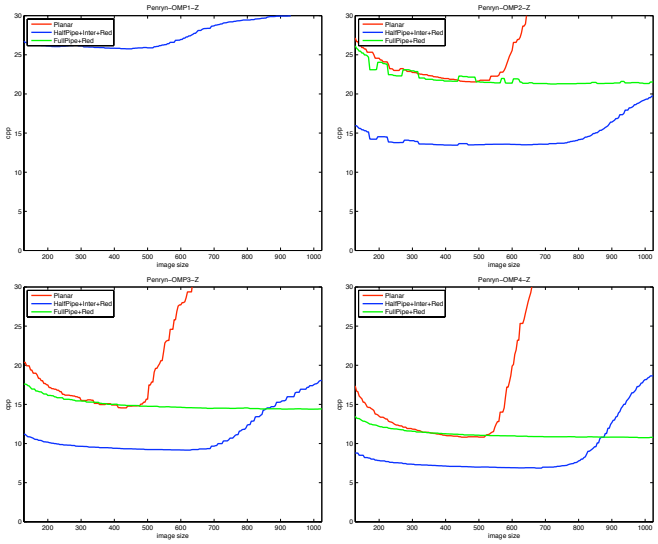

FIG. 5 – Sorties de caches sur le Penryn avec OpenMP

pour la version planar (Fig. 6). En revanche, la version *halfpipe* ne présente pas de sortie de cache et reste efficace pour toutes les tailles d'images (validé jusqu'à des images 2048×2048).

La version *fullpipe* est *computation bound* : son *cpp* est constant et indépendant de la taille des images traitées.

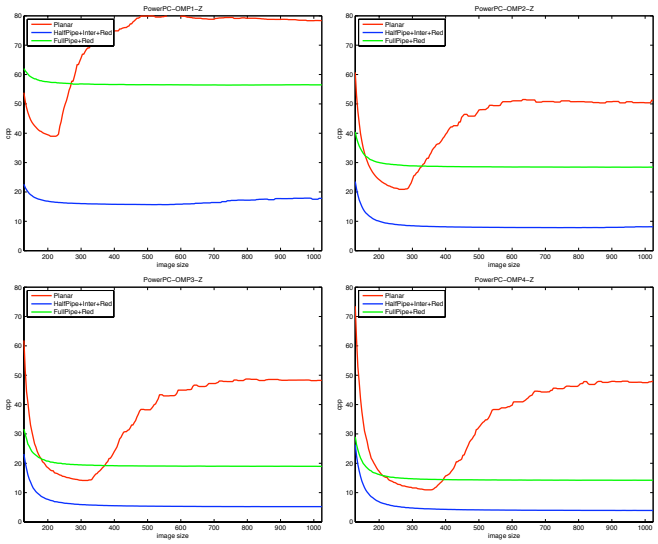

FIG. 6 – Sorties de caches sur le G5 avec OpenMP

3.2.3 Le Cell

Comme introduit dans la partie 3.2.1, la vectorisation manuelle dans les sections parallèles d'OpenMP n'est pas supportée par le compilateur CBEXLC, celui-ci proposant une vectorisation automatique. Les versions utilisant la parallélisation via OpenMP et la vectorisation automatique via CBEXLC ne permettent qu'une accélération de $\times 1.8$.

Les hautes performances ne sont atteignables que via une parallélisation manuelle et l'utilisation du SIMD. Dans ce cas, la scalabilité du Cell est très bonne. De plus, le Cell n'ayant pas de cache, les performances restent stables quelque soit la taille des images (validation jusqu'à des images 4096×4096).

3.2.4 Les GPU

TAB. 3 – Résultat en cpp de l'opérateur Harris sur 8800GTX.

Version	cpp calcul	cpp transfert	cpp total	ratio
Planar A/R RAM PC	2.92	54.69	57.61	$\times 18.7$
Planar RAM GPU	2.92	2.92	5.84	$\times 1.0$
Halfpipe	0.77	2.92	3.69	$\times 3.8$
Fullpipe	0.55	2.92	3.48	$\times 5.2$
Fullpipe + bords	1.10	2.92	4.03	$\times 2.6$

Si les GPU sont souvent considérées comme efficaces à travers de nombreux benchmarks, comme ceux compilés sur la page CUDA, il y a parfois des ambiguïtés sur les mesures réalisées : les performances prennent-elles en compte les temps de transferts entre le PC et la carte (dans un sens puis dans l'autre) ? Si on considère le cas du Traitement d'Images, les images fournies au GPU sont issues d'une caméra éventuellement rapide (les dernières caméras FireWire800 atteignent 205 images par seconde en 640×480), le GPU peut être considéré comme un accélérateur matériel pour une partie des calculs. Deux verrous technologiques ont été mis en évidence : les temps de transferts et la gestion des bords.

Dans la suite du développement, nous analysons les résultats du GPU de Nvidia. En effet, Brook+ s'est avéré plus simple à mettre en œuvre mais n'a pas permis autant d'optimisations que CUDA.

Le tableau 3 donne les temps de calculs et de transferts en *cpp* pour 5 implantations différentes (Planar avec / sans transfert pour chaque opérateur, *halfpipe*, *fullpipe* avec / sans traitement des bords). Ainsi que le ratio entre transferts et calculs. La première ligne (Planar avec transferts) indique les performances dans le cas où le GPU est considéré comme un accélérateur externe : chaque fois que le CPU rencontre un opérateur accélérable sur GPU, il envoie les données, lance les calculs et récupère le résultats. Il y a donc des transferts pour chaque opérateur (Sobel, puis Mul, Gauss et coarsité). Ce fonctionnement s'apparente à celui d'une FPU externe fonctionnant sous interruptions. Dans ce cas défavorable, les temps de transfert représentent 18 fois les temps de calcul.

Dans les autres implantations, la granularité augmente : Harris est considéré comme un et un seul opérateur atomique. Les données sont transférées avant le calcul du gradient, les données temporaires sont uniquement stockées sur la carte et seul le résultat final est renvoyé en mémoire centrale. Les temps de transferts étant identique pour ces versions et le temps de calcul diminuant avec les optimisations, le ratio atteint une valeur de $\times 5.3$.

Dans la conception actuelle (adressage circulaire via modulo pour l'utilisation de textures dans le jeux) la gestion des bords est problématique. Les différentes gestions classiques des bords peuvent être utilisées, mais aucune ne donne de bonnes performances. La simple gestion des bords provoque une augmentation du temps de calcul d'un facteur 2 !

Dans la perspective où ces deux problèmes seraient résolus, en ne prenant en compte que les calculs ($cpp = 0.55$) l'exécution ne prendrait plus que 0.25 ms. Les GPUs seraient alors plus rapides que le Cell (0.33 ms) et ferait jeu égal avec l'octo-cœur Penryn. Sur le plan énergétique, les GPU réduisent alors leur écart avec les GPP en atteignant 47.1 mJ.

3.3 Efficacité énergétique

TAB. 4 – Efficacité énergétique pour des images 512×512.

Archi	Techno (nm)	Puissance (W)	énergie (mJ)
G4	130	10	**192.4**
G5	90	2×70	44
Conroe	65	35	47.8
Penryn	45	95	56.0
bi-Penryn	45	2×95	**26.6**
Cell	65	70	**22.9**
8800 GTX	90	175	277.6
HD4850	55	110	641.3

Le tableau 4 présente l'efficacité énergétique des différentes architectures. Une précédente étude avait montré qu'il été possible de rendre les GPP compétitifs énergétiquement face aux systèmes spécialisés [5]. Dans cette nouvelle comparaison, les processeurs pour « serveurs » se révèlent plus efficaces que les processeurs pour portable (G4 et Conroe). Le Cell est le plus efficace, devant les quad et octo cœurs. Les GPU sont distancés. La bonne performance du Cell est semblable au classement énergétique *green* du Top500 : le Cell est actuellement le *Model of Computation* le plus efficace pour le calcul intensif.

3.4 De la relativité de la mesure

TAB. 5 – Efficacité énergétique pour des images 400×400

Archi	Conroe	bi-quad Penryn	Cell
énergie (mJ)	**10**	16.2	14

Jusqu'à maintenant, les performances des machines étaient évaluées pour des tailles fixes d'images, correspondant plus ou moins à la taille des capteurs existants. Il peut être intéressant de prendre le problème à l'envers et de s'interroger sur l'intervalle de taille d'image pour lequel les différentes architectures sont efficaces. Si l'efficacité du Cell est constante (pas 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. Une configuration de type « serveur » est plus efficace (bus rapide) plus longtemps (2 processeurs quadri-cœurs Penryn 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 les paramètres d'efficacité du Intel Conroe pour une taille d'image 400×400 au lieu de 512×512, le classement des processeurs « efficaces » est modifié (Tab 5). Le Conroe qui était 1.8 fois moins efficace que le bi-Penryn devient 1.6 fois plus efficace (amélioration d'un facteur 2.9). La taille des caches est un facteur primordial pour les GPP. Les optimisations présentées, en repoussant le moment des sorties de cache et en diminuant leur amplitude, sont nécessaires pour limiter l'accroissement de la taille des caches tout en ayant de bonnes performances pour de grandes tailles d'images.

4 Conclusions et perspectives

4.1 Conclusions :

Les développements précédents permettent de conclure quelques enseignements sur les diverses architectures testées :

- **Cell :** Il est très scalable car insensible à la taille des données. C'est actuellement le modèle de calcul le plus efficace. Cependant, pour obtenir une utilisation efficace du Cell, les optimisations manuelles (vectorisation et parallélisation) sont indispensables.
- **GPP :** Les facteurs d'accélérations observés (jusqu'à $\times 90$ pour les versions les plus optimisées) les rendent plus rapides et plus efficaces que les GPU.
- **GPU :** Les GPU souffrent de problèmes de bande passante et de gestion des bords. Ce type d'architecture deviendra compétitive lorsque ces problèmes seront résolus, par exemple sous forme de *streaming* SoC (bus capteur - GPU rapide).

De plus, pour le choix de l'architecture, il faudra aussi porter une attention particulière à la facilité de développement, aux outils disponibles et aux transformations que l'on est prêt à mettre en œuvre pour accélérer efficacement l'execution d'un code :

- **Transformations :** Les transformations architecturales et algorithmiques sont efficaces et nécessaires pour obtenir de bonnes performances, quelle que soit l'architecture.
- ***Multithreading* :** OpenMP est la façon la plus efficace et la plus simple pour déployer un code sur un multi-cœurs.
- **SIMD :** Avec l'utilisation des caches, le SIMD permet des accélérations très importantes. Cela reste une option incontournable en terme d'efficacité mais représente un temps et une difficulté de développement non négligeable.
- **Compilateurs :** Les options d'optimisation inclues dans les compilateurs ICC et XLC (vectorisation et déroulage de boucles par exemple) sont utiles mais les performances restent inférieures à ce que l'on peut obtenir dans le cadre d'optimisations manuelles.

Finalement, selon les ressources disponibles pour le développement et si la consommation énergétique est un point crucial, le Cell se révèle à nouveau être le *Model of Computation* le plus efficace.

4.2 Perspectives :

Il sera intéressant de suivre l'évolution des GPU (augmentation du nombre de cœur et norme PCI Express 3.0) et de les comparer à des version du Cell comprenant plus de cœurs (16 ou 32). Sachant que les futurs compétiteurs seront les Intel Sandy Bridge (AVX 256 bits) et le Larabee (32 cœurs et SIMD 512 bits). Les problèmes de transfert seront toujours critiques. En ce qui concerne les outils, l'intégration dans GCC 4.4 de nouvelles options telles que le scripting de graphite [6] ou la gestion de la version 3 d'OpenMP reste aussi à évaluer.

Remerciement

Les auteurs tiennent à remercier chaleureusement Loic Cloup et Dany Tello pour leur aide au codage des GPU et à la réalisation des nombreux benchmarks.

Références

[1] C. Harris and M. Stephens : *A combined corner and edge detector*. 4th ALVEY Vision Conference, 1988.

[2] J.-O. Klein, L.Lacassagne, H. Mathias, S. Moutault, A. Dupret : *Low Power Image Processing : Analog versus Digital Comparison*, IEEE CAMP 2005.

[3] L. Lacassagne, A. Manzanera, J. Denoulet, A. Mérigot : *High Performance Motion Detection : Some trends toward new embedded architectures for vision systems*. Journal of Real Time Image Processing 2008. Open Access : `http://www.springerlink.com/content/02264j1658w48641`.

[4] F. Lohier, L. Lacassagne, P. Garda : ICASSP 1999 *A generic methodology for the software managing of caches in multiprocessors DSP architectures - Application to the real time implementation of low level image processing on the TMS320C80*, ICASSP 1999.

[5] S. Piskorski , L. Lacassagne, S. Bouaziz, D. Etiemble : *Customizing CPU instructions for embedded vision systems*, IEEE CAMPS 2006.

[6] S. Pop, A. Cohen, C.Bastoul, S. Girbal : *Graphite : Polyhedral Analyses and Optimizations for GCC*. GCC 2008.

[7] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, S. Bouaziz : *Parallelization Schemes for Memory Optimization on the Cell Processor : A Case Study on the Harris Corner Detector*. HiPEAC Transactions on High-Performance Embedded Architectures and Compilers 2008.

[8] D. Truong, F. Bodin, A. Seznec : *Improving cache behavior of dynamically allocated data structures*, PACT 1998.