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

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]. L'idée est de faire de même pour des architectures logicielles afin 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).

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. Une grande expertise est nécessaire afin d'obtenir les meilleures performances possibles. 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.

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) et les outils associés.

2.1 Transformations

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 [6] : 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. 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 : 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. La version *Fullpipe* nécessite moins de transferts mais 9 fois plus de calcul que la version *Halfpipe*.

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 *multithreading* 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, les compilateurs CBEXLC, PPU-GCC et SPU-GCC ont été utilisés. Les GPU ont été programmé avec leur langage *ad hoc* : CUDA pour Nvidia et Brook+ pour ATI.

3 Benchmarks et efficacités

3.1 Architectures testées

Nous avons effectué les benchmarks sur les processeurs suivants :

- Bi-dual cœurs PowerPC G5 à 2.5 GHz (PPC970MP)
- Dual cœurs Intel Conroe à 2.4 GHz (T7700)
- Quadricœurs Intel Penryn à 2.8 GHz (Q9550)
- Biquadri 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
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. La version *Fullpipe* n'est pas présente dans ce tableau car sont intérêt ne réside pas dans sa performance mais dans son model de consommation de données qui le rend indépendant de la taille des images.

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

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

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. Ces transformations restent, par nature, hors de portée d'un compilateur.

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 apportée 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 sont bien meilleurs que sur les GPP par rapport au mode scalaire. 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 importants. Le Cell ne

laisse donc pas le choix quant aux méthodes de développement à appliquer si l'on cherche la performance.

L'efficacité de la vectoriation étant toujours 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, 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. 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.

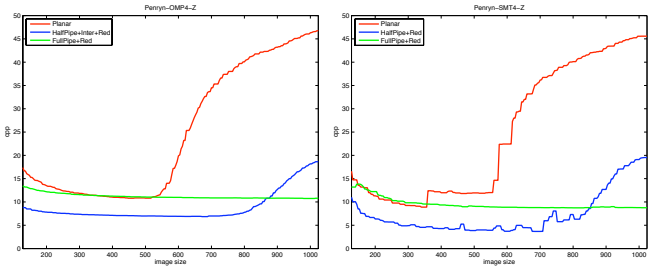

FIG. 1 – Comparaison des performances pour le *multithreading* entre l'utilisation d'OpenMP vs Pthreads

Les tests que nous avons mené montrent que les versions utilisant OpenMP restent plus stables que celles reposant sur la bibliothèque Pthread (Fig. 1). 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.

Finalement, pour les besoins les plus simples de parallélisation des opérateurs, 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

L'augmentation du nombre d'unités de calculs (cœurs, SPE) dans les processeurs s'est généralisée depuis quelques années. L'intérêt du multi-cœurs 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

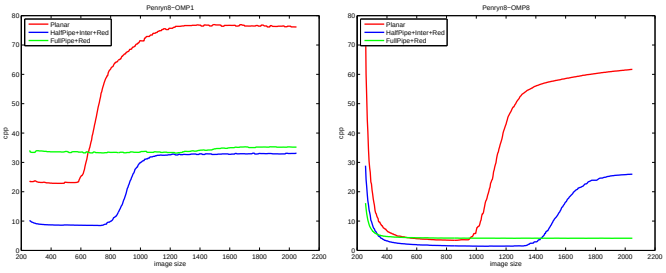

FIG. 2 – Sorties de caches sur le Biquad Penrynn avec OpenMP

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.

L'efficacité du code est ainsi prolongée pour de grande taille de données. Ainsi pour le processeur G5 (Fig. 2), les pertes d'efficacité qui, pour un thread, débutaient pour des images de taille 230×230 ne commencent qu'à partir de 380×380 avec 4 threads pour la version *Planar*. On constate que la version *Fullpipe* ne présente pas de sortie de cache et reste efficace pour toutes les tailles d'images.

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.

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 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	$\times 2.6$

Nous allons analyser 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.

Deux verrous technologiques ont été mis en évidence : les temps de transferts et la gestion des bords. Le tableau 3 donne les temps de calculs et de transferts en *cpp* pour 4 implantations différentes (Planar, *halfpipe*, *fullpipe* avec / sans traitement des bords). Ainsi que le ratio entre transferts et calculs.

Harris est considéré comme un 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 gain atteint une valeur de $\times 5.3$. Aussi, on constate que la gestion des bords augmente le 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)
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 [4]. Dans cette nouvelle comparaison, les processeurs pour « serveurs » se révèlent plus efficaces que les processeurs pour portable (i.e. 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.

4 Conclusions et perspectives

4.1 Conclusions :

Ce document apporte plusieurs arguments vis-à-vis du choix d'une architecture efficace utilisable en Traitement d'Image.

Le Cell est très scalable car insensible à la taille des données. C'est actuellement le modèle de calcul le plus efficace. Cependant, les optimisations manuelles (vectorisation et parallélisation) sont complexes et indispensables. Les optimisations effectuées sur les GPP les rendent plus rapides et plus efficaces que les GPU, ces derniers souffrant de problèmes de bande passante et de gestion des bords.

Aussi, il faudra 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 un code. Les transformations architecturales et algorithmiques sont efficaces et nécessaires pour obtenir de bonnes performances et ce, quelque soit l'architecture. De plus, la standardisation des GPP multi-cœurs implique l'utilisation d'outils tel qu'OpenMP qui permet de deployer un code simplement et automatiquement. Un code multithreadé permet d'utiliser au mieux les caches disponibles. L'utilisation du SIMD permet des accélération très importantes mais représente un temps et une difficulté de développement non négligeable. Les compilateurs proposent de nombreuses options d'optimisation, telle que la vectorisation et le déroulage de boucles pour ICC et XLC, mais les gains restent inférieurs à ce que l'ont peut obtenir avec des 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 [5] ou la gestion de la version 3 d'OpenMP reste aussi à évaluer.

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

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

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