# Parallelization Strategies for the Points of Interests Algorithm on the Cell Processor

Tarik Saidani, Lionel Lacassagne, Samir Bouaziz, and Taj Muhammad Khan

Université de Paris-Sud, Institut d'Electronique Fondamentale, Bâtiment 220,
F-91405 Orsay Cedex, France
firstname.name@ief.u-psud.fr

**Abstract.** The Cell processor is a typical example of a heterogeneous multiprocessor-on-chip architecture that uses several levels of parallelism to deliver high performance. Closing the gap between peak performance and sustained performance is the challenge for software tool developers and the application developers. Image processing and media applications are typical "main stream" applications. In this paper, we use the Harris algorithm for detection of points of interest (PoI) in an image as a benchmark to compare the performance of several parallel schemes on a Cell processor. The impact of the DMA controlled data transfers and the synchronizations between SPEs explains the differences between the performance of the different parallelization schemes. These results will be used to design a tool for an efficient mapping of image processing applications on multi-core architectures.

## 1 Introduction

Recent trends in designing general purpose processors have focused on increasing Thread Level Parallelism - for example through Chip Multiprocessing architectures - rather than improving clock frequencies of single-task scalar systems which posed computing and energy efficiency problems. One way of facing these problems was found in SIMD architectures, either by duplicating complete processing units (according to Flynn's definition), or by inserting dedicated SIMD instructions inside existing CPUs (aka SWAR for SIMD Within A Register) - like Altivec or SSE. A good example can be found in the IBM Cell Processor[1]. However, developers are now faced with implementation problems, such as efficiently distributing code among the processing elements and generating DMA (Direct Memory Access) requests required by data flows. This paper is organized as follows. A presentation of the Cell Broadband Engine and applications that have already been benchmarked on the Cell are given, followed by a description of the Harris points of interest algorithms and its different implementation models. The influence of transfers on performance is then discussed. Finally, effective performance results are provided and discussed. The final part gives conclusions about our experiments and gives perspectives for future works.

## 2 The Cell Processor

The Cell processor is a heterogeneous, multi-core chip consisting of one 64-bit power processor element (PPE), eight specialized units called synergistic processors (SPE) [2],

I. Stojmenovic et al. (Eds.): ISPA 2007, LNCS 4742, pp. 104–112, 2007.

other interfacing units and a high bandwidth bus called Element Interconnect Bus (EIB), that allows communications between the different components. Assuming a clock speed of 3.2 Ghz, the Cell processor has a theoretical peak performance of 204.8 Gflops/s in single precision and 14.6 Gflops in double precision. The EIB supports a peak bandwidth of 204.8 Gbytes/s for internal transfers (when performing 8 simultaneous non-colliding 25GB/s transfers). The PPE unit is a traditional 64-bit PowerPC Processor with a vector multimedia extension (VMX). This Cell's main processor is in charge of running the OS, and coordinating the SPEs. Each SPE consists in a synergistic processor unit (SPU) and a Memory Flow Controller (MFC). The SPE holds a local storage of 256 KB, and a 128-bit SWAR (very close to Altivec) unit dedicated to high-performance data-intensive computation. The MFC holds a 1D DMA controller, that is in charge of transferring data from external devices to the local store, or writing back computation results to main memory. One of the main characteristics of the Cell processor is its distributed memory hierarchy. The main drawback of this kind of memory, is that the software must handle the limited size of the local storage of each SPE, by issuing DMA transfers from or toward main storage.

## 3 Related Work

The different examples of implementations on the Cell processor ([3], [4], [5] and [6]) consider a SPMD (Single Program Multiple Data) parallelization model on micro-benchmarks (few operators with a reduced amount of transfers), and do not explore other models. This implementation strategy is the most obvious one, and does not require any complex synchronization mechanism to work. In our paper, we present the Harris points of interest (POI) algorithm which is representative of several image processing algorithms, since it includes multiplication, thresholding and convolution kernels. It is also an interesting case study because its data flow graph allows different mapping strategies on the Cell processor. By exploring different parallelization schemes, we can show different aspects of the influence of DMA transfers on the performance, and compare practical results with the expectations.

## 4 The Harris Points of Interests Algorithm

In our paper we use a common image processing algorithm known as Harris points of interest. This application was chosen because its operator graph allows different implementation models. The Harris algorithm 4 is a mix of operators on pixels and convolution kernels. It is composed of four steps of basic operators, with up to three parallel operations, and many transfers with or without borders, in details we have:

1. a gradient computation (usually a Sobel X & Y operators),
2. three parallel multiplications, to combine first derivatives together,
3. three parallel Smoothing operations (usually a $3 \times 3$ Gauss Kernel),
4. a coarsity computation (Harris point operator) : $K = S_{xx}S_{yy} - S_{xy}^2$

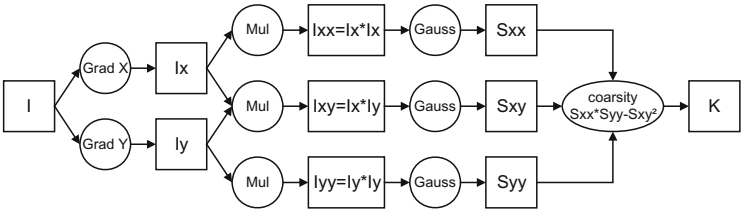


**Fig. 1.** Harris algorithm

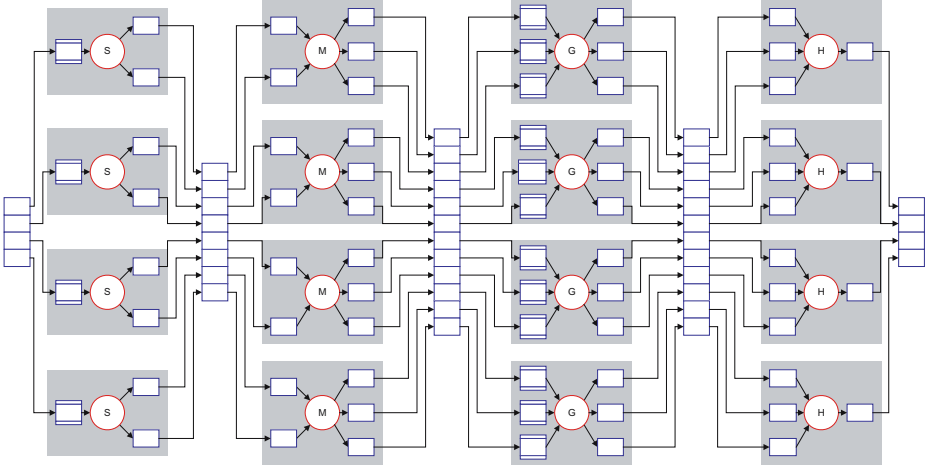


**Fig. 2.** POI SPMD model

## 5 Implementation Models for the POI Algorithm

### 5.1 SPMD Model

In all the following figures, S stands for the Sobel operator, M for the multiplication, G for Gauss and H for Harris. The gray rectangles represents and SPE. The SPMD programming model (Fig. 2) equally divides the image into eight regions of processing (RoPs) blocks, mapped on the SPUs (in the figure, only 4 SPEs are drawn to get a smaller figure, but 8 SPEs are actually used). All SPUs execute the same program/code.The PPU lets the SPUs run one operator on the whole image before proceeding with the next operator. For example, it will not issue the command for Multiplication operator until all the SPUs have finished performing the Sobel operator and the whole of the image has been transfered back into the XDR.

### 5.2 Conventionnal Pipeline Model

This implementation of the POI algorithm (Fig. 3) consists in mapping the graph in pipeline fashion, where the RoP consist in the entire image. This way, we considerably

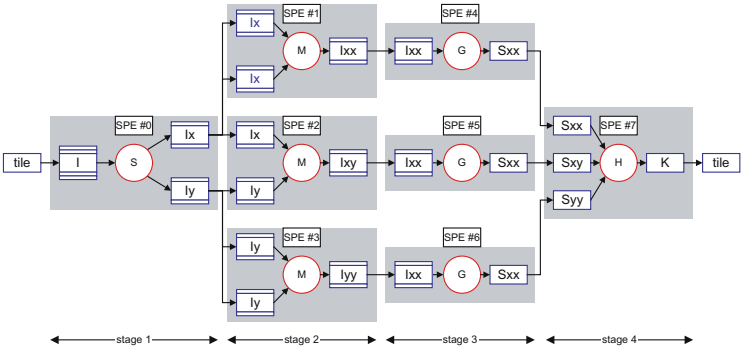


**Fig. 3.** POI simple pipeline model

serialize the algorithm, and maximize the amount of transfers between SPEs. Assuming that most of the transfers are performed serially, the contention rate on the bus is minimized. The transfers in this version are characterized by top and bottom borders, added for the convolution kernels. Left and right borders where removed by performing registers renaming.

### 5.3 Half Pipeline Model

In this version (Fig. 4), we rely on the TLP offered by the Cell processor by merging two successive operators in pairs, the Sobel with the Multiplication, and Gauss with Harris. Thus, we divide the graph into two threads, that can be duplicated four times to fill in the entire set of SPEs. Unlike the previous version, and considering that there are four threads running concurrently in each step, the EIB bandwidth can be considerably affected because of the important amount of concurrent transfers.

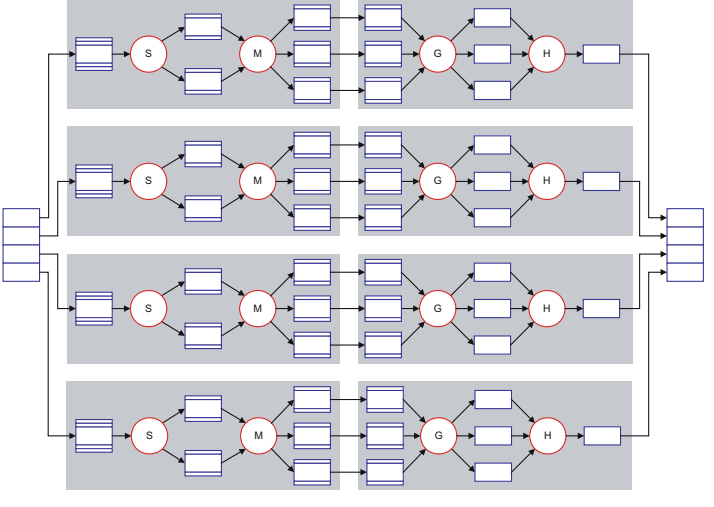


**Fig. 4.** Half Pipeline model

### 5.4 Models Comparison

The SPMD model is the easiest one to implement. Since every SPE performs the same computations on different input data, a single piece of code is developed and sent to each processing element. However, doing so implies storing the intermediate results either in the processor local memory (thus drastically restraining the size of the processed tile), or in the central memory, which leads to many transfers and thus communication bus contentions. However, if the simple pipeline model fully benefits from inter-SPEs communication facilities, it implies many transfers per computation and does not take advantage from the TLP offered by the Cell processor. The half pipeline model which can be considered as a compromise between the two precedent versions, exploits the fast communications between SPEs and the TLP. In the next section, we will verify if implementation results agree with our expectations.

### 5.5 Considerations on the DMA Transfers

DMA transfers are the main issue when developing image processing applications on the Cell processor. The size of computation tiles is the first parameter to consider. On the one hand, these pieces of data must be as large as possible (up to the LS). Transferring large tiles, reduces the amount of reloads when performing convolution kernels. On the other hand DMA requests must be performed in 16 KB chunks. Otherwise EIB bandwidth is seriously affected [7]. The internal bus bandwidth is also related to the number of concurrent transfers, as that the EIB can handle up to 12 non-colliding parallel transfers (3 per node). The last constraint on DMA requests concerns inter-SPE transfers, that needs a synchronization mechanism (Fig. 5). This process aims to ensure data coherence, but can causes SPE stalls when signaling is quite slow.

### 5.6 Benchmark Results

In this part of the paper, we present some preliminary results of our different models. In our experiments, we considered different image sizes. Pixels are coded in single precision floats format in order to have a simplified transfer management, and also because of the limitations of the SPU instruction set. We measured the execution time (CPP) with 4KB, 8KB and 16KB tiles to evaluate the impact of the tile size on it. The metric used is $cpp$, which is the number of clock cycles per pixel.

$$cpp = \frac{\text{cpu cycles}}{N^2} \tag{1}$$

Image size is N $\times$ N pixels.The $cpp$ is more representative than $cpi$ (cycles per instruction) to compare the complexity per pixel of different algorithms. It also shows the influence of memory transfers on computation and allows a fair comparison between the architectures,as this measure is independent from the clock frequency. According to the curves in Fig. 6, Fig. 7 and Fig. 8, one can note that $cpp$ decreases when tile size increases for all versions. This is due to the difference in the amount of transferred data. One can also note that 16KB is the optimal value of DMA message, that guarantees a maximum bandwidth on the EIB [7]. This is why we obtain an optimal cpp with a 16KB tile. Although it is theoretically the best one, the half pipeline does not reach satisfying

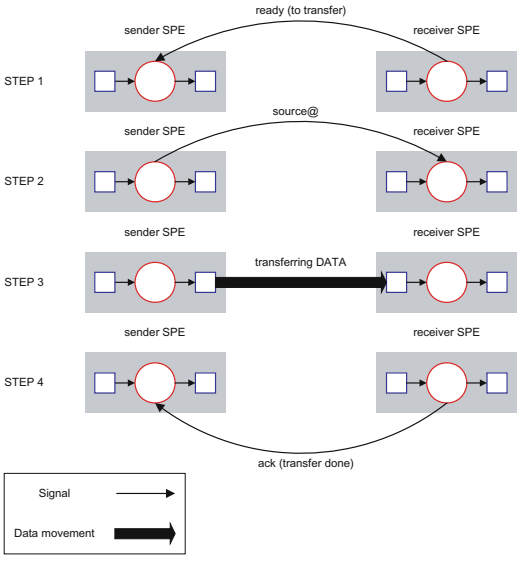


**Fig. 5.** Example of synchronization steps between two SPEs when transferring Data from one SPE to another SPE

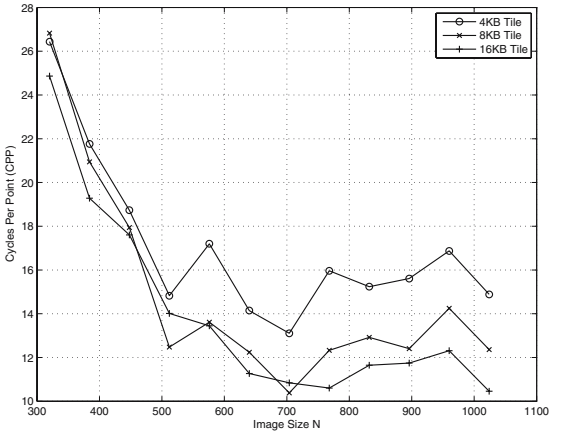


**Fig. 6.** Cycles per pixel for the SPMD Implementation

performances. This comes from the big contention on the bus caused by the collision of the numerous transfers running concurrently, and the effect of synchronizations that causes SPE stalls. The comparison between different implementations (Fig. 9) shows that the SPMD version has the best *cpp*.

**Comparison with a single core Implementation.** To get a better idea of the gain provided by the Cell processor, we compare the SPMD version with an equivalent implementation on a mono-core PowerPC G5 using the Altivec ISA (Instruction Set

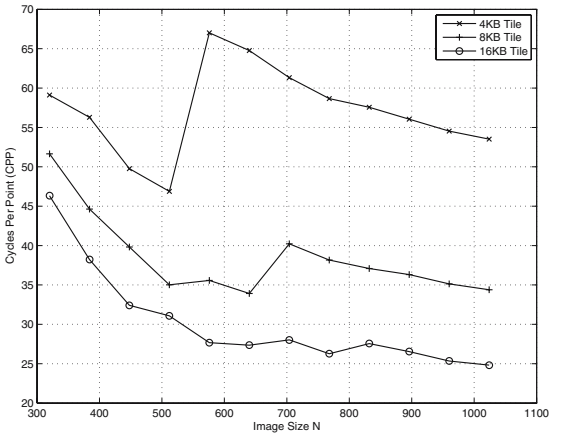


**Fig. 7.** Cycles per pixel for the Pipeline Implementation

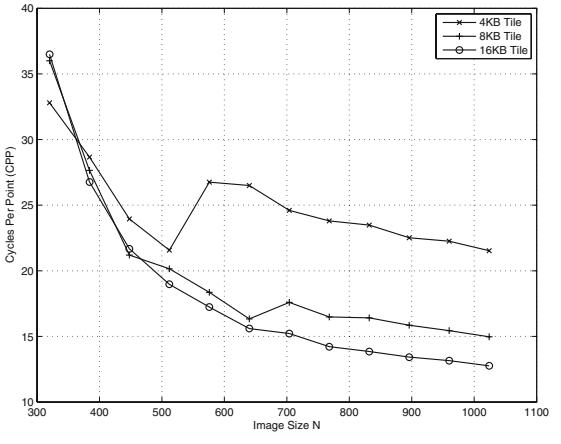


**Fig. 8.** Cycles per pixel for the Half Pipeline Implementation

Architecture), knowing that both of the Altivec and SPU units are in-order processors, and all of the instructions used to code the POI algorithm on the SPU have their corresponding instructions on the Altivec extension. The main characteristic of the G5, is that performance is limited by the cache size, which is used to minimize the bottleneck due to memory transfers : the *cpp* increases dramatically when the processed data do not fit in the cache. This handover is amplified by the cache misses, and can not be avoided but just deferred by increasing the cache size. Unlike the G5 processor, the SPU does not use the cache mechanism to improve memory transfers. The size of processed data (tile) is fixed by the user with taking into account the size of the local storage. The *cpp* decreases while the data size increases and reaches a constant value for large image sizes. With the SPMD version using 8 SPUs and large image sizes, the speed-up is approximately x7, which is close to the x8 maximum speed-up.

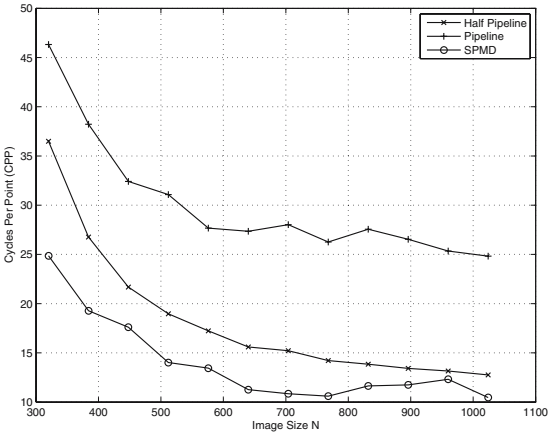


**Fig. 9.** Comparison between implementations for 16KB Tiles

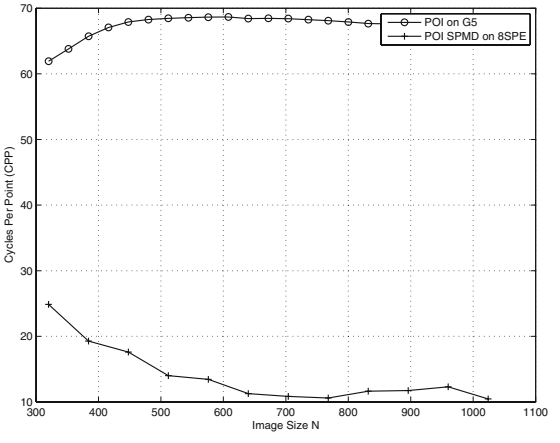


**Fig. 10.** Comparison Between G5 and Cell SPMD version

## 6 Conclusion and Future Work

We introduced different distribution models for the Harris POI algorithm on the Cell processor. This application was chosen for its variant schemes of parallelization and also because it includes different kinds of computation kernels. The influence of the size of memory transfers and the number of concurrent ones on the performance decrease is noted. The performances reached by pipeline versions do not match our expectations. This is due to costly synchronizations between SPEs that increases the number of stalls in the threads and numerous concurrent transfers that causes contention on the EIB. Better handling of synchronizations is one of the projected improvements. The obtained results will be exploited to design a tool for efficient image processing code distribution on the Cell processor and other multi-core platforms. Examples of parallelization methods can be found in [5], [8] and [9]. Tiling is the better strategy for distributing code on such architectures, since memory transfers are the bottleneck that limits performance.

## References

1. Pham, D., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Harvey, P., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Pham, M., Pille, J., Posluszny, S., Riley, M., Stasiak, D., Suzuoki, M., Takahashi, O., Warnock, J., Weitzel, S., Wendel, D., Yazawa, K.: Overview of the Architecture, Circuit Design, and Physical Implementation of a First-generation Cell Processor. IEEE Journal of Solid-State Circuits 41, 179–196 (2006)
2. IBM: Cell Broadband Engine Programming Handbook. Version 1.0 edn. IBM (2006)
3. Petrini, F., Fossum, G., Fernández, J., Varbanescu, A.L., Kistler, M., Perrone, M.: Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In: IEEE/ACM International Parallel and Distributed Processing Symposium (2007)
4. Greene, J., Cooper, R.: A Parallel 64k Complex fft Algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor. In: Global Signal Processing Expo. (2005)
5. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Zhang, A.W.T., Zhao, P., Gschwind, M.K., Archambault, R., Gao, Y., Koo, R.: Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. IBM Ssystems Journal 45 (2006)
6. Benthin, C., Wald, I., Scherbaum, M., Friedrich, H.: Ray Tracing on the CELL Processor. In: IEEE Symposium on Interactive Ray Tracing (2006)
7. Kistler, M., Perrone, M., Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed. IEEE Micro 26, 10–23 (2006)
8. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the Memory Hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (2006)
9. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for Explicitly Managed Memory Hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)