# Parallelization Schemes for Memory Optimization on the Cell Processor : A Case Study on the Harris Corner Detector

Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz

Institut d'Electronique Fondamentale
Université de Paris Sud
91405 Orsay Cedex
France

**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. Reducing the gap between peak performance and effective performance is the challenge for software tool developers and the application developers. Image processing and media applications are typical "main stream" applications. We use the Harris algorithm for the detection of interest points 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. The scalability of the architecture is modeled and evaluated.

## 1 Introduction

Image processing applications are generally composed of a set of basic operators. These components can be point to point operators or convolution kernels. Due to both computation and memory complexity, real-time execution of image processing algorithms has historically not been easily done efficiently. Multi-core processors family appeared to respond to an increasing demand of processing power that single-task scalar systems, which raised computing and energy efficiency problems, could not satisfy. Furthermore, computing and transfer workloads can be distributed on the multiple processing units to reduce the processing time, in particular for media processing application which are well suited for the multiple levels of parallelism provided by parallel architectures.

The Cell processor [16] is a good example of a heterogeneous multi-processor (Fig. 1). Composed of a 64-bit power processor element (PPE), eight specialized units called synergistic processors (SPE) and a high bandwidth bus called Element Interconnect Bus (EIB), that allows communications between the different components [9], The Cell is a heterogeneous, multi-core chip containing several levels of parallelism that can be exploited to reach high peak performances. Assuming a clock speed of 3.2 Ghz, the Cell processor has a theoretical peak performance of 204.8 GFlops/s in single precision. The EIB is composed of four 128-bit rings, each ring can handle up to three concurrent transfers. The theoretical peak bandwidth of the bus is 204.8 GBytes/s for internal transfers, when performing 8 simultaneous non-colliding 25.6 GB/s transfers (the Cell network topology allows only 8 transfers to be done in parallel without collision). The PPE unit is a traditional 64-bit PowerPC Processor with a vector multimedia extension (VMX aka Altivec). 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 (LS) 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 LS of each SPE, by issuing DMA transfers from or toward main storage (MS).

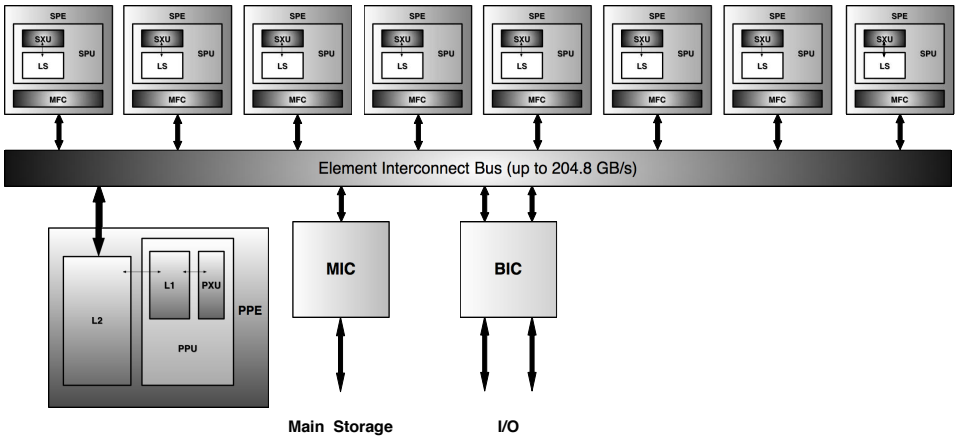


**Fig. 1.** Detailed view of the Cell Broadband Engine Architecture

However, some specific programming aspects – namely Direct Memory Access (DMA) controlled transfers – makes it hard for developers to code and debug their applications quickly on the Cell processor. Therefore, it is necessary to develop software tools that can make the programming process less painful and the most suitable with the target architecture. Since the release of the first prototypes and simulator, there were several examples of application porting on the Cell BE, with various application domains like bio-informatics [19, 18], graphics rendering [3] and other scientific computing kernels [20, 7, 15]. In [19] the authors adopted a progressive optimization strategy where a PPU version of the "Clustal W" applications was tuned so that the code matches the capabilities of the SPU cores. Various implementation strategies (MPI, OpenMP, SIMD) were tested and compared in [18] for a protein docking application, the influence of DMA transfers on performance was also discussed. On the other hand, various programming models was mapped on the Cell processor in the form of tools, programming languages and compilers [6, 12, 5, 14]. In [5] the Cell BE is viewed as a shared memory SMP (Symetric Multi-Processor) where the compiler performs the task of distributing work over the SPEs via OpenMP parallelization directives and where data transfers to the local stores are handled implicitly using software caches. The CellSs[2] programming model is somewhat similar to OpenMP as it relies on code annotation to offload a part of the work to the SPEs except that it relies on a source to source C compiler and a locality-aware task scheduler optimizes data transfers at runtime. Message passing programming model is treated in [14]: applications are partitioned into MPI micro-tasks and a static scheduler performs the task of optimizing their execution on the parallel cores. Finally, the RapidMind [12] tool relies on a model where data transfers and computation kernels are completely decoupled so that optimizations like inter-kernel data access elimination and transfer/computation overlapping can be performed.

The goal of this paper is to evaluate the performance of some computation models relying on various communication and mapping strategies on the Cell BE processor for a representative image processing algorithm.

- The implementation and evaluation of several parallelization schemes for the Harris interest point detection algorithm are performed.
- The influence of DMA transfer size on the performance of each model is demonstrated.
- The impact of chaining technique to boost the performance on the Cell is exposed.
- A comparison between the Cell SPU unit and other cache-based SIMD extensions is provided.
- The scalability of the Cell processors is modeled and measured via efficiency and speedup metrics.

The paper is organized as follows. In Section 2, we describe our image processing algorithm: the Harris and Stephen corner detector. Section 3 describes the implementation models and their

comparative performances. In Section 4 we model and evaluate the Cell scalability. Finally, we sum up our main contributions and discuss future work in Section 5.

## 2 The Harris Interest Point Detection Algorithm

Harris and Stephen [8] interest point detection algorithm is used in computer vision systems for feature extraction like motion detection, image matching, tracking, 3D reconstruction and object recognition. This algorithm was proposed to address the limitations of the Moravec corner detector [13] which was sensitive to noise and not rotationally invariant. A corner can be defined as the intersection of two edges when an interest point can be defined as a point which has a well defined position and can be robustly detected. Hence, the interest point can be a corner but also an isolated point of local intensity maximum or minimum, a line ending, or a point on a curve where the curvature is locally maximal.

### 2.1 Algorithm description

Assuming image patches of dimensions $u \times v$ (in our case $3 \times 3$) in a grayscale 2-dimensional image $I$ and shifting it by $(x, y)$, the Harris operator is based on the estimation of local autocorrelation $S$ for which the expression is:

$$S(x,y) = \sum_u \sum_v w(u,v) \left(I(u,v) - I(u-x, v-y)\right)^2 \tag{1}$$

By approximating S with a second order Taylor series expansion the Harris matrix $M$ is given by:

$$M = \sum_u \sum_v w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{2}$$

An interest point is characterized by a large variation of $S$ in all directions of the vector $(x, y)$. By analyzing the eigenvalues of $M$, this characterization can be expressed in the following way. Let $\lambda_1$, $\lambda_2$ be the eigenvalues on the matrix $M$:

1. If $\lambda_1 \approx 0$ and $\lambda_2 \approx 0$ then there are no features of interest at this pixel $(x, y)$.
2. If $\lambda_1 \approx 0$ and $\lambda_2$ is some large positive value, then an edge is found.
3. If $\lambda_1$ and $\lambda_2$ are both large, distinct positive values, then a corner is found.

Harris and Stephens note that eigenvalues computation is expensive, since it requires the computation of a square root, and instead suggest the following algorithm:

1. For each pixel $(x, y)$ in the image compute the autocorrelation matrix $M$:

$$M = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}; \text{where:} S_{xx} = \left(\frac{\partial I}{\partial x}\right)^2 \otimes w, S_{yy} = \left(\frac{\partial I}{\partial y}\right)^2 \otimes w, S_{xy} = \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w \tag{3}$$

   Where $\otimes$ is the convolution operator and $w$ a Gaussian kernel.

2. Construct the coarsity map by calculating the coarsity measure $C(x, y)$ for each pixel $(x, y)$, with $k$ being a empirically determined constant:

$$\begin{aligned} C(x,y) &= det(M) - k(trace(M))^2 \\ det(M) &= S_{xx} \cdot S_{yy} - S_{xy}^2 \\ trace(M) &= S_{xx} + S_{yy} \end{aligned}$$

An illustration of an input 512×512 grayscale image and a interest point detection on it are given in Fig. 2. To obtain this result, two additional steps are performed in order to extract visually appealing information from the dense coarsity matrix[1]. Those steps are:

1. Threshold the interest map by setting all $C(x, y)$ below a given threshold $T$ to zero.
2. Local maxima are then extracted by filtering points which are greater than all the points in a $3 \times 3$ neighborhood.

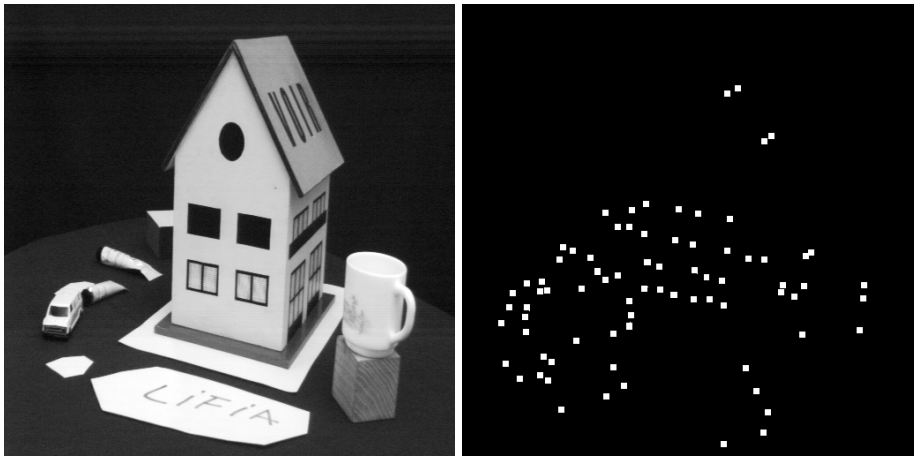

**Fig. 2.** Illustration of the interest point detection on a grayscale 512×512 image

### 2.2 Implementation Details

Grayscale 2-dimensional image pixels are typically 8-bit unsigned integers and the Harris algorithm output is in this case a 32-bit signed integer. However, because of the limitations of the Cell SPU ISA, and in order to guarantee a fair comparison between Altivec, SSE and Cell SPU, we chose the single precision floating point format for both input pixels and the output of the Harris operator. In our implementation of the Harris operator we divided the algorithm into four computation kernels: a Sobel operator representing the derivative in the horizontal and vertical directions, a multiplication operator, a Gaussian smoothing operator ($w$ in Eq. 2) followed by a coarsity computation. In our implementation the $k$ constant in Eq. 4 is fixed to 0 as it does not change the qualitative results. This leads to the data flow graph given in Fig. 3 which is representative of typical image processing algorithms as it includes convolution kernels and point to point operators and in which the Sobel operator convolution kernels (one for horizontal gradient ($GradX$) and one for the vertical gradient ($GradY$)) and the Gauss smoothing kernel $Gauss$ are defined by :

$$GradX = \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}; GradY = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}; Gauss = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The convolution kernels computation consist in centering the kernel on a pixel and computing the cumulated sum of the point to point product of the kernel elements with the image patch surrounding the central pixel. Hence, the Harris algorithm can be considered as a memory bounded problem since this kind of operators are great bandwidth consumers as they consume more elements than they produce. For this reason we chose to perform memory access optimizations at several levels of the Cell processor memory hierarchy.

[1] As those steps are merely cosmetic, we will not consider them as part of the algorithmic chain.

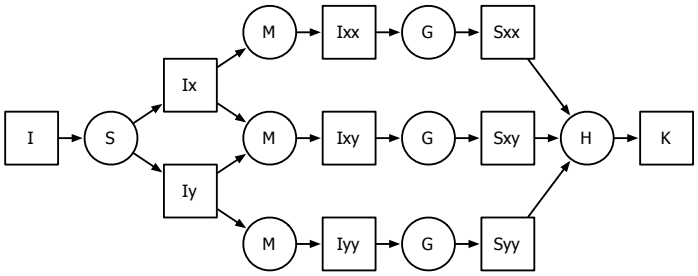


**Fig. 3.** Harris algorithm dataflow graph

## 3 Optimizations and Parallelization Strategies

Implementing a given image processing application on the Cell is not a trivial task as various level of optimization are available. We focus on two kind of optimizations : optimizations driven by the application domain and optimizations driven by the underlying architecture. We detail how those optimization techniques can be applied to the Harris algorithm and how they drive the parallelization strategy.

### 3.1 Signal Processing Optimization

The first optimizations to be applied are *Domain Specific*. Those optimizations include kernel separability, kernel overlapping and computation factorization.

**Kernel Separability** In our case, we will take advantages of the fact that 2D convolution kernels used by the Gauss and Sobel operators are separable. A 2D convolution kernel is said to be separable if it can be expressed as the outer product of two vectors Eq. 4, 5 and 6).

$$Gauss = \frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} = \frac{1}{16}\begin{bmatrix}1\\2\\1\end{bmatrix} * \begin{bmatrix}1&2&1\end{bmatrix} \tag{4}$$

$$Grad_X = \frac{1}{8}\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix} = \frac{1}{8}\begin{bmatrix}1\\2\\1\end{bmatrix} * \begin{bmatrix}-1&0&+1\end{bmatrix} \tag{5}$$

$$Grad_Y = \frac{1}{8}\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix} = \frac{1}{8}\begin{bmatrix}-1\\0\\+1\end{bmatrix} * \begin{bmatrix}1&2&1\end{bmatrix} \tag{6}$$

This reformulation reduces both the number of memory accesses and arithmetic complexity (see Table 1).

**Convolution kernel overlapping** The second step is to take into account how kernels are applied. Due to overlapping (Fig. 4), there is only one new column of pixels to load from the memory at each iteration. Thanks to kernels separability, they are first applied column-wise by computing the vertical filtering. Temporary results are saved into registers and convolved with the horizontal filter. The typical loops transformation are *Register Rotation* and *Loop Unrolling* (an example is given in the next section). The *Register Rotation* is preferred because it does not increase the loop body and because no prolog neither epilogue are required.

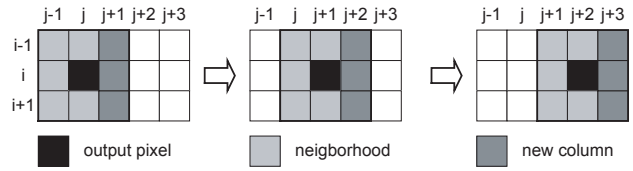


**Fig. 4.** convolution kernel overlaping

**Reduction and computation factorization** Once the 2D convolution kernels are split into two 1D convolution kernels and the kernel overlapping has been taken into account, *reduction* by column is applied to take advantage of the column reuse. Let us consider the convolution of the Gauss kernel[2] with a $3 \times 4$ pixels matrix (Eq. 7)[3].

$$\begin{bmatrix} r_0 \; r_1 \end{bmatrix} = \begin{bmatrix} 1\,2\,1 \\ 2\,4\,2 \\ 1\,2\,1 \end{bmatrix} * \begin{bmatrix} a_0 \; b_0 \; c_0 \; d_0 \\ a_1 \; b_1 \; c_1 \; d_1 \\ a_2 \; b_2 \; c_2 \; d_2 \end{bmatrix} \tag{7}$$

We have:

$$r_0 = \begin{bmatrix} 1\,2\,1 \end{bmatrix} * \left( \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \right) \tag{8}$$

Let $r_a$, $r_b$ and $r_c$ be the *reduced* registers by column:

$$r_a = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \quad r_b \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}, \quad r_c \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \tag{9}$$

The output $r_0$ can be expressed as:

$$r_0 = \begin{bmatrix} 1\,2\,1 \end{bmatrix} * \begin{bmatrix} r_a \; r_b \; r_c \end{bmatrix} = r_a + 2r_b + r_c \tag{10}$$

In order to compute $r_1$, the first column is recycled by loading three data (column $d$: $d_0$, $d_1$, $d_2$ ), and applying the 1D kernel to get a new *reduced* register $r_a$ (Eq. 11). Thus $r_1$ can benefit of the previous computations (Eq. 12).

$$r_a = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}, \tag{11}$$

$$r_1 = \begin{bmatrix} 1\,2\,1 \end{bmatrix} * \begin{bmatrix} r_b \; r_c \; r_a \end{bmatrix} = r_b + 2r_c + r_a \tag{12}$$

Each *reduced* register is used thrice, thus saving memory accesses and computations.

The arithmetic complexity of the Harris operators are given in table 1, where *Number* indicates the number of calls to each operator when no optimizations are performed and when kernel separability and overlapping are exploited. We notice that those simple optimizations reduce the global complexity by 46%.

[2] the same technique is also applied to the Sobel operator
[3] fractions have been removed to simplify the notation.

| *Operator* | Number | `MUL` | `ADD` | Total |
|---|---|---|---|---|
| *Complexity without optimization* | | | | |
| Sobel | 2 | 3 | 5 | 16 |
| Mul | 3 | 1 | 0 | 3 |
| Gauss | 3 | 6 | 8 | 42 |
| Coarsity | 1 | 2 | 1 | 3 |
| Total | - | 29 | 35 | **64** |
| *Complexity with optimizations* | | | | |
| Sobel | 2 | 0 | 5 | 10 |
| Mul | 3 | 1 | 0 | 3 |
| Gauss | 3 | 0 | 6 | 18 |
| Coarsity | 1 | 2 | 1 | 3 |
| Total | - | 4 | 30 | **34** |

**Table 1.** arithmetic operator complexity with/without optmizations

**Temporal Pipelining** In a *producer-consumer* point of view, there are actually two kind of operators in the Harris operator, each having a specific memory access pattern:

- point to point operators, like the Multiplication and the Coarsity operators, that consume a $1 \times 1$ data to produce a $1 \times 1$ data.
- $3 \times 3$ convolution kernels, like the Sobel gradients ($Grad_X$ and $Grad_Y$) and the Gauss smoother that consumes $3 \times 3$ data and produces a $1 \times 1$ data (Fig. 5)

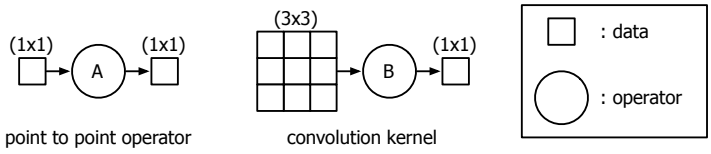


**Fig. 5.** producer-consumer model: memory access pattern for point operator and convolution kernel

The *Temporal pipelining* optimization consists in chaining operators together by adapting their memory access patterns in order to remove the intermediate `LOAD/STORE` instructions.

Figure 6 sums up the various pipelining rules. In rule 1, the output pattern of the first operator already fit the input pattern. No pattern adaptation is required before removing the intermediate memory access. For rule 2 : there is nothing to do for pipelining a $3 \times 3$ convolution kernel with a point operator. But permuting point operator and convolution kernel (rule 3) requires unrolling the first operator in order to adapt the pattern. In that case the first operator should be unrolled thrice in both dimensions. The last rule is the pipelining of two $3 \times 3$ convolution kernels (rule 4). As for the third rule, the first operator should be unrolled $3 \times 3$ times. The big difference in that case is that the new input pattern is $5 \times 5$, that requires 25 registers just to hold the loaded data. One possible drawback of this pipelining is *spill code* if the compiler runs out of register. The last point about pipelining is to see the Sobel gradient operator as one operator instead of two: $3 \times 3$ points are loaded only one time but consumed twice to produce two points, one for $Grad_X$ and one for $Grad_Y$.

**Benefits of Domain Specific Optimizations** The leading idea of all those optimizations is to reduce complexity. This can be done by both the reduction of the number of computations per

point (arithmetic transformation like reduction) and the amount of memory accesses (temporal pipelining).

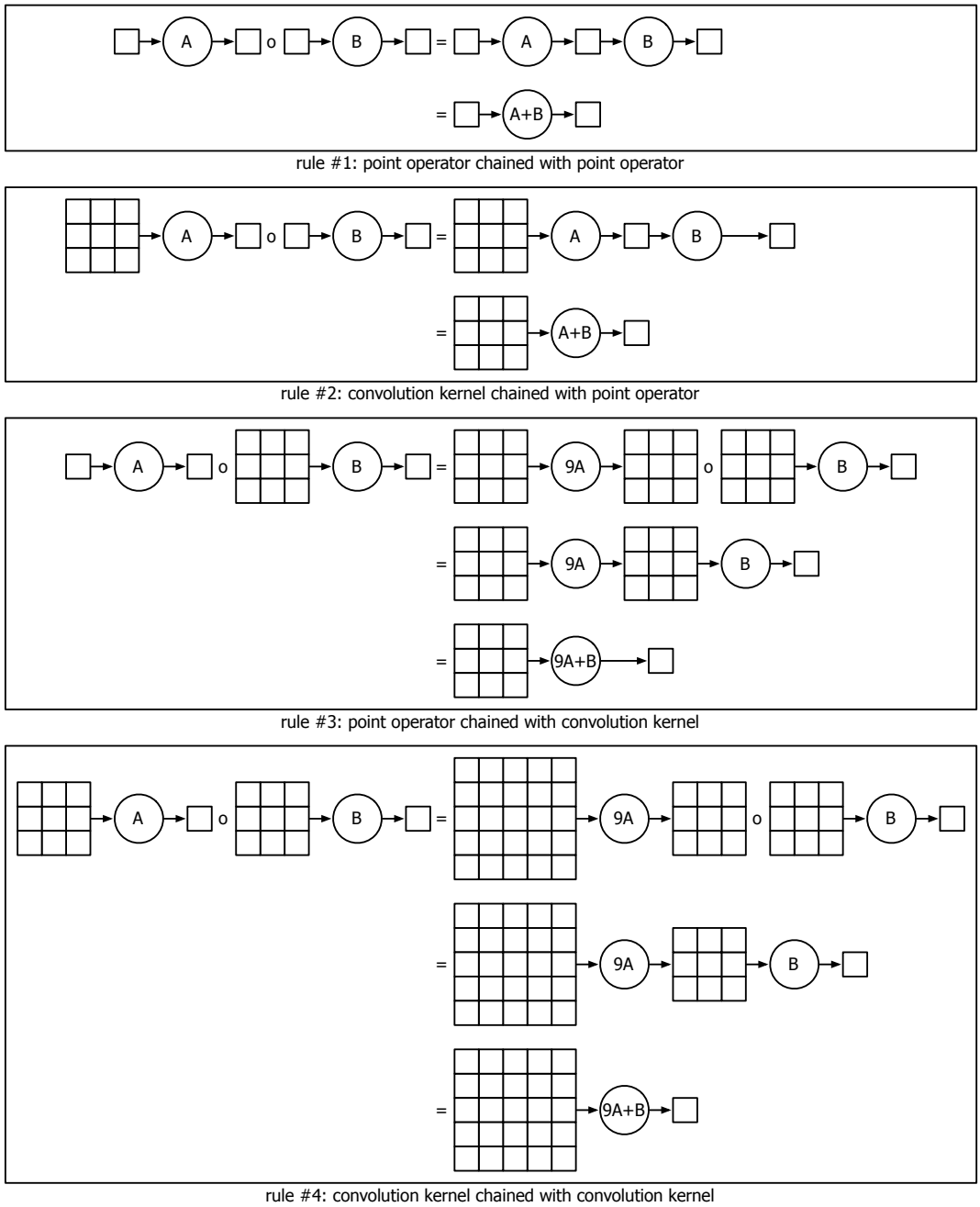


**Fig. 6.** Pattern fitting pipelining rules

By combining these two kind of optimizations, there are four versions of the Harris operator: the basic implementation of Harris with or without arithmetic optimizations, and the HalfPipe implementation of Harris with or without arithmetic optimizations. The HalfPipe optimization consists in applying rule #2 to Harris: pipelining a convolution kernel with a point to point

operator: Sobel is pipelined with Mul and Gauss is pipelined with Coarsity. Finally, one can remove too the normalization coefficients of Gauss and Sobel 1/16 and 1/8 that are usually used in image processing to normalize the output of these kernels to get the same magnitude as the input, but that is, in our case, useless as the threshold performed is relative to the maximum extracted value. The memory complexity of the operator is given in table 2 where *Number* indicates the number of *Input Pattern* and of *Output Pattern* of each operator.

| Operator | Number | Input Pattern | Number | Output Pattern | Total |
|---|---|---|---|---|---|
| *NoPipe version* | | | | | |
| Sobel | 2 | $3 \times 3$ | 2 | $1 \times 1$ | 20 |
| Mul | 3 | $1 \times 1$ | 3 | $1 \times 1$ | 6 |
| Gauss | 3 | $3 \times 3$ | 3 | $1 \times 1$ | 30 |
| Coarsity | 3 | $1 \times 1$ | 1 | $1 \times 1$ | 4 |
| Total | | 51 | | 9 | **60** |
| ***NoPipe with Register Rotation and Reduction*** | | | | | |
| Sobel | 1 | $3 \times 1$ | 2 | $1 \times 1$ | 5 |
| Mul | 2 | $1 \times 1$ | 3 | $1 \times 1$ | 5 |
| Gauss | 3 | $3 \times 1$ | 3 | $1 \times 1$ | 12 |
| Coarsity | 3 | $1 \times 1$ | 1 | $1 \times 1$ | 4 |
| Total | | 17 | | 9 | **26** |
| *HalfPipe version* | | | | | |
| Sobel+Mul | 1 | $3 \times 3$ | 3 | $1 \times 1$ | 12 |
| Gauss+Coarsity | 3 | $3 \times 3$ | 1 | $1 \times 1$ | 28 |
| Total | | 36 | | 4 | **40** |
| ***HalfPipe version with Register Rotation and Reduction*** | | | | | |
| Sobel+Mul | 1 | $3 \times 1$ | 3 | $1 \times 1$ | 6 |
| Gauss+Coarsity | 3 | $3 \times 1$ | 1 | $1 \times 1$ | 10 |
| total | | 12 | | 4 | **16** |
| *FullPipe version* | | | | | |
| Sobel+Mul+Gauss+Coarsity | 1 | $5 \times 5$ | 1 | $1 \times 1$ | 26 |
| total | | 25 | | 1 | **26** |
| ***FullPipe version with Register Rotation and Reduction*** | | | | | |
| Sobel+Mul+Gauss+Coarsity | 1 | $5 \times 1$ | 1 | $1 \times 1$ | 6 |
| total | | 5 | | 1 | **6** |

**Table 2.** Complexity of memory accesses pattern, with/without optimizations

### 3.2 DMA related Optimizations

DMA transfers are the main issue when developing image processing applications on the Cell processor. The developer must care about certain considerations when performing data transfers from main storage to local stores, or betweens local stores. The first parameter to consider when transferring data to the SPE is the size of the transfers. We measured the bandwidth of data transfer size varying from 8 to 16384 bytes which is the maximum size that can be issued by a single DMA request for the Cell MFC. The data must be transferred by 16 KB chunks to have a full bandwidth on the EIB, smaller transfers are done with a reduced bandwidth (Tab. 3). The internal bus bandwidth is also related to the number of concurrent transfers, as that the EIB can handle up to 12 parallel transfers (3 per ring but only 8 without potential collision (as explained in section 3). The second parameter to consider is the physical proximity (aka SPE affinity), between SPEs when performing an inter-SPE transfers. This parameter can not be controlled by the user as

the task of scheduling tasks on SPEs is done by the kernel scheduler. Finally, data being transfered must be contiguous in memory[4]. Those limitations have a big impact on the tiling strategy. As local store has only 256 KB to hold both code and data, large amounts of data being processed have to be split into *tiles* which size is compatible with the memory available on the SPEs and compatible with the maximal size of a DMA transfer.

| Size (B) | Agg BW (GB/s) | Size (B) | Agg BW (GB/s) |
|---|---|---|---|
| 8 | 0.92 | 512 | 47.92 |
| 16 | 1.86 | 1024 | 72.57 |
| 32 | 3.72 | 2048 | 86.48 |
| 64 | 7.42 | 4096 | 94.09 |
| 128 | 14.87 | 8192 | 97.72 |
| 256 | 27.37 | 16384 | 104.10 |

**Table 3.** Aggregate bandwidth for inter-SPE DMA transfers with 8 SPE transferring concurrently on a QS20 Blade

### 3.3 Parallel Implementations

In all the following figures, S refers to the Sobel operator, M to the multiplication, G to Gauss and H to Harris. The gray rectangles represents an SPE unit. The source image is divided into $p$ regions of processing (RoPs), $p$ being the number of available SPEs. Each Rop is then split into tiles. The operators consumes input tiles and produces output tiles. We assume that tile width equals RoP and image widths in order to avoid transfers of non contiguous memory regions.

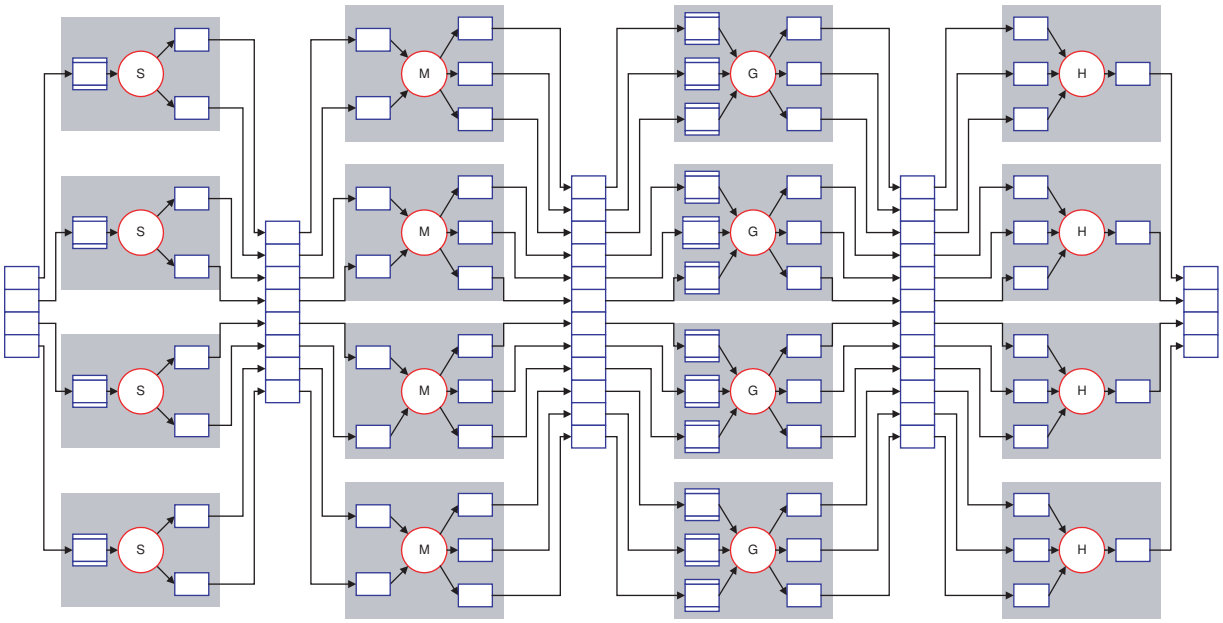


**Fig. 7.** Conventional SPMD model.

[4] Such DMA transfer are said to be mono-dimensional or *1D*

**Conventional SPMD** The conventional SPMD programming model (Fig. 7) equally divides the image into 8 RoPs, mapped on the SPEs (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 SPEs have finished performing the Sobel operator and the whole of the image has been transfered back into the MS.

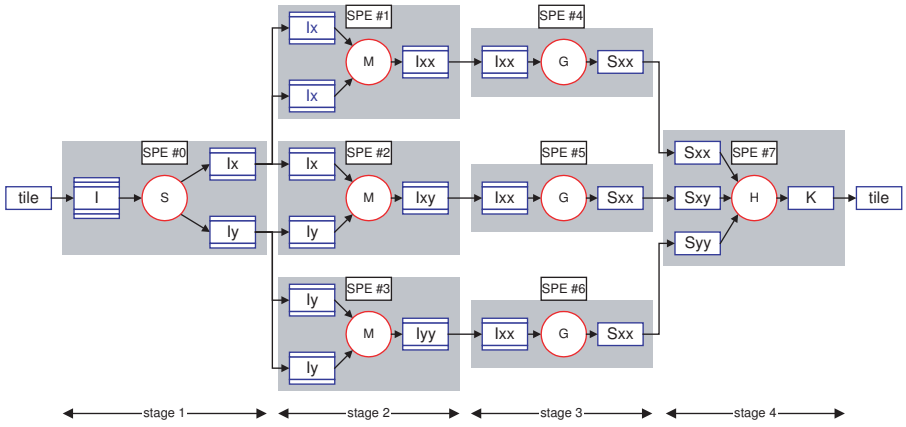


**Fig. 8.** Conventional pipeline model.

**Conventional Pipeline** This implementation of the algorithm (Fig. 8) consists in mapping the graph in pipeline fashion, where the RoP consists in the entire image. This way, we considerably 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 (neighborhood pixels). Left and right borders where removed by performing registers rotation.

**Half Chaining: 2 SPEs** In this version (Fig. 9), we merge 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.

**Half Chaining + Half Pipeline** The difference that we can note here is that in opposition to the previous model, the Sobel and Mul operators are chained in order to avoid the time loss in `LOAD` and in `STORE` instructions residing between these two steps. Therefore, the number of cycles per pixel can be considerably improved since we know that the memory instruction latency equals 6 in the SPU [9].

**Full Chaining + Half Pipeline : 1 SPE** By chaining all the operators into the same SPE, this implementation not only allows removing `LOAD/STORE` instructions between operators, but also improves the parallelism rate of the algorithm, since we can split the input image into 8 slices and use one SPU to perform all the operations.

### 3.4 Models Comparison

Fig. 12 gives the comparison between the different implementation models of the Harris algorithm on the Cell processor. The first observation that can be made is that the conventional pipeline

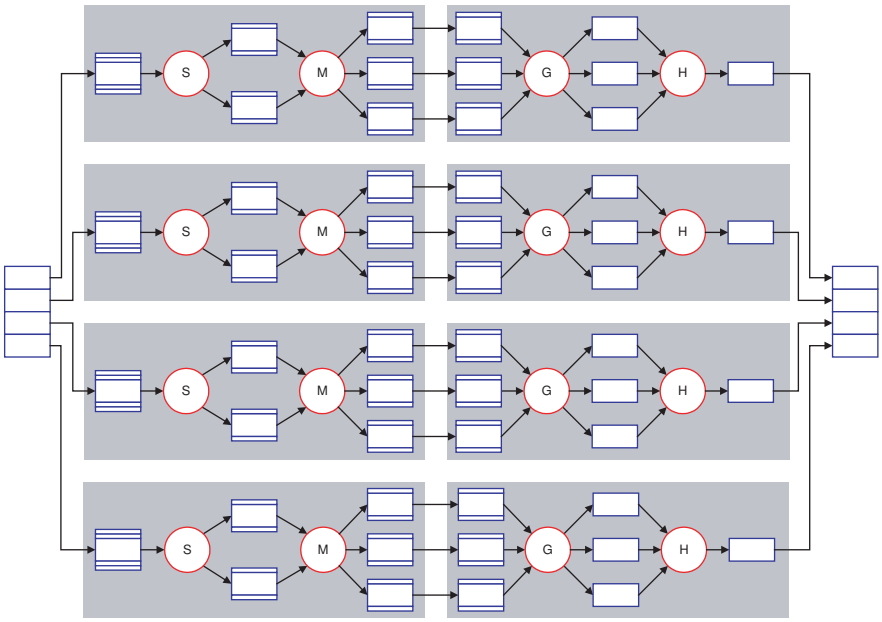


**Fig. 9.** Half chaining model on 2 SPEs.

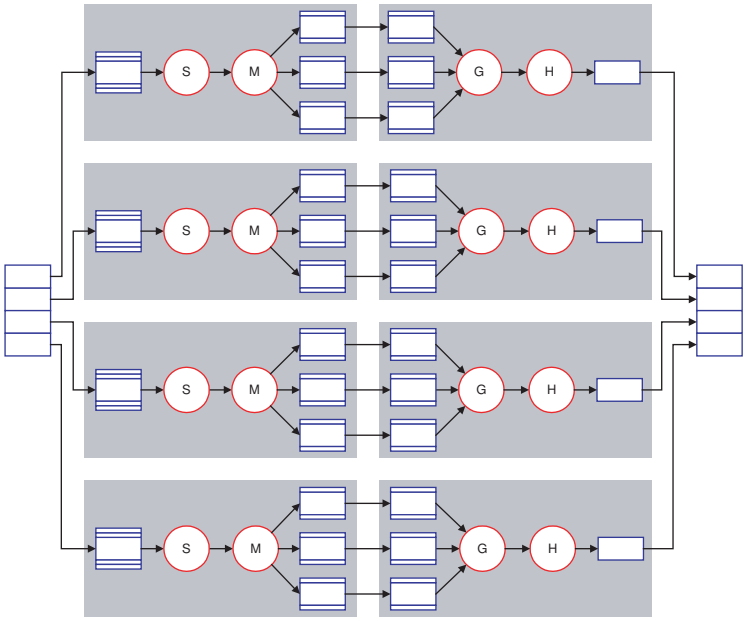


**Fig. 10.** Half chaining + half pipeline on 2 SPEs.

version gives the worst performances, which was expected: this version is deliberately serialized and does not fully exploit the TLP (Thread Level Parallelism) offered by the target architecture. The other observations match our expectations:

– Our memory optimization techniques improve global performances as the fastest implementation is the *Half pipeline+Full chaining* version where operators are pipelined and chained inside an SPE.

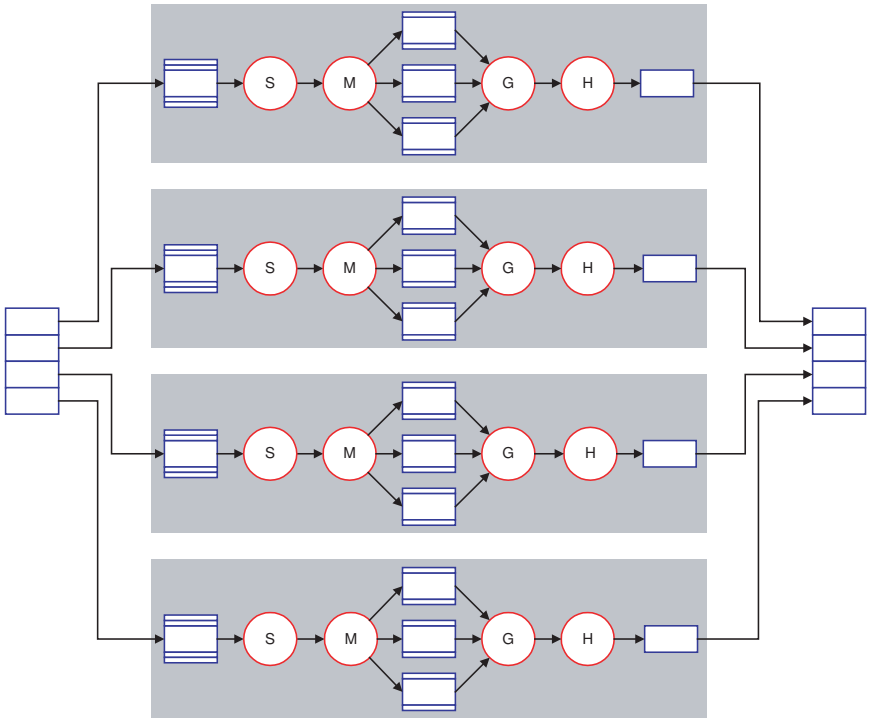


**Fig. 11.** Half chaining + half pipeline model on 1 SPE

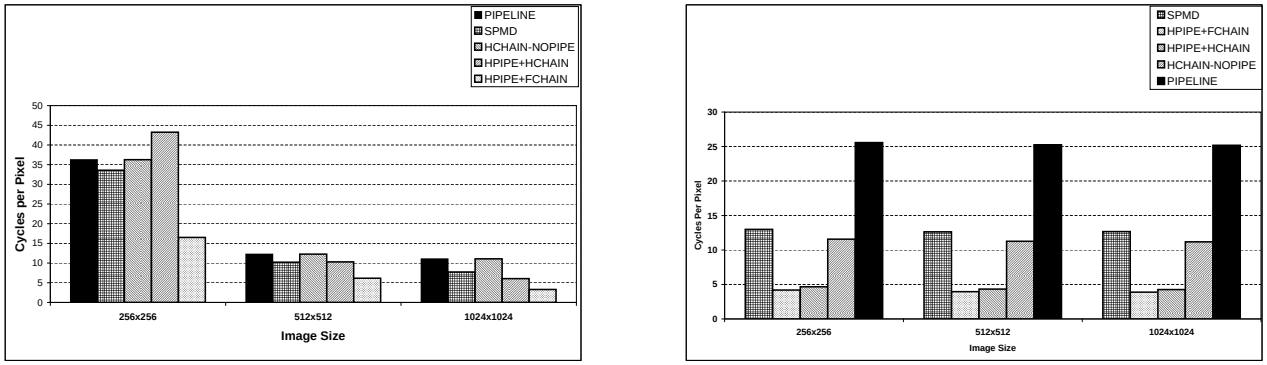


**Fig. 12.** Comparison benchmark for implementation models: The left figure shows results obtained in [17]

– The versions where inter-SPE transfers are used, have good performances as the *No pipeline+Half chaining* model runs faster than the *SPMD* model where data transits only on the external memory bus. In addition *Half pipeline+Half chaining* is almost as fast as the best version, which proves that inter-SPE bandwidth is comparable to local `LOAD/STORE` bandwidth.

### 3.5 Tile Size Influence

As stated in [11, 10] the size of transfered data blocks has an influence on the bandwidth on the EIB. In our application domain bandwidth performance is critical for the overall performance of

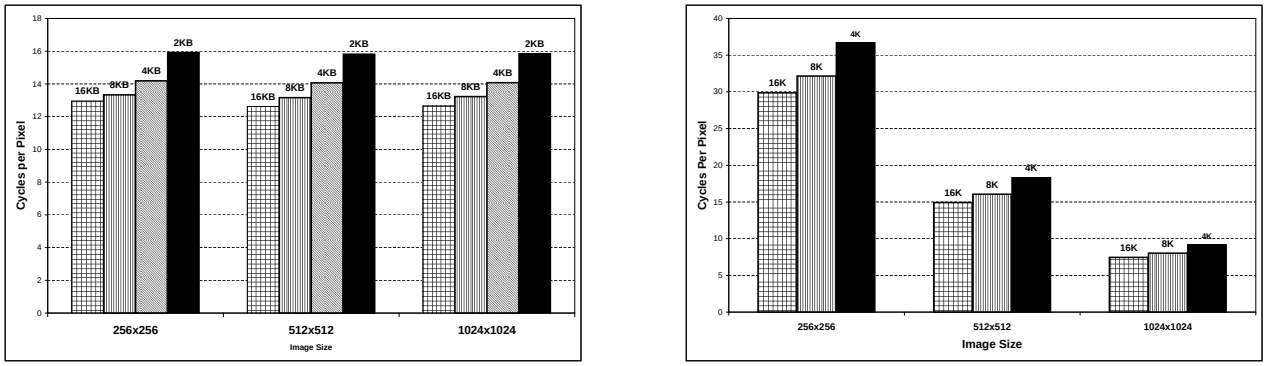


**Fig. 13.** Influence of transfer size on the performance, full pipeline model 1 SPU : The left figure shows results obtained in [17]

an algorithm since they are characterized by a large transfer/computation ratio. Fig. 13 states that the best *cpp* is reached when transferring 16 KB tiles which can be explained as follows:

- 16 KB is the transfer size that guarantees a maximum bandwidth on the EIB .
- Big tile size reduces the amount of reloaded data when performing convolution kernels.

### 3.6 Performance Analysis

The comparison of global performance of the different implementation models is not sufficient to prove that memory optimizations are the main factor for performance improvement. In order to give a more accurate analysis, we performed time measurements at the SPU level where we evaluated the gain provided by merging computation kernels and performing inter-SPE communication. One can note that the *Conventional Pipeline* and *SPMD* models are not considered because their were implemented just to serve as a reference and can not be compared with the other models as it does not benefit of most of the optimizations techniques that we cited above.
Table 4 shows on the one hand the clock cycle count for the Halfchain version where two kernels are decoupled and an intermediate `LOAD/STORE` operations are required between them and on the other hand the Halfchain+Halfpipe version where the two kernels are merged (the **o** operator denotes the function composition (merging) operator) and the intermediate `LOAD/STORE` are removed. As we see in Tab. 4 the speedup provided by this code transformation reaches $\times 7.2$. The other optimization that we performed which consists in replacing inter-SPE DMA by local

| Model | Operator | Cycles Count | Speedup |
|---|---|---|---|
| **Halfchain** | **Sobel+Mul+**`LOAD/STORE` | 119346 | x |
| **Halfchain** | **Gauss+Coarsity+**`LOAD/STORE` | 188630 | x |
| **Halfchain+Halfpipe** | **(Sobel o Mul)+**`LOAD/STORE` | 16539 | 7.2 |
| **Halfchain+Halfpipe** | **(Gauss o Coarsity)+**`LOAD/STORE` | 504309 | 3.5 |

**Table 4.** Operator fusion comparison

`LOAD/STORE` instructions aims to demonstrate the benefit of keeping data inside the local store as the maximum theoretical bandwidth is 51.6 GB/s for a `LOAD/STORE` in the LS and 25.6 GB/s for an inter-SPE GET/PUT DMA operation. One can note the the maximum bandwidth for `LOAD/STORE`

in Tab. 5 computed assuming that there is 1 instruction issued each cycle (pipelined execution) for a clock frequency of 3.2 Ghz. The same assumption was made in [10] for the LS bandwidth measurement. On the other hand, we used the IBM Performance Debugging Tool (PDT) to measure the average effective DMA bandwidth the result is given in Tab. 5. We observe that there is an order of magnitude between the two transfer rates which explains the difference in global performance between *Halfchain* and *Fullchain* versions. The measurements that we performed give

| Model | Communication Type | Average Bandwidth GB/s |
|---|---|---|
| **Halfchain** | inter-SPE DMA | 6.95 |
| **Halfchain+Halfpipe** | `LOAD/STORE` | 51.6 |

**Table 5.** average bandwidth comparison between models

a more precise view of the factors that influence the global performance of the different implementations. Merging computation kernels, reduces the memory complexity by maximizing the reuse of register to perform intermediate computations. However, this optimization should be used with care as there is a limited amount of available register to store intermediate results (128 for the SPU). On the other hand, keeping data in the local store whenever it is possible is a good practice as the `LOAD/STORE` bandwidth is higher than inter-SPE DMA bandwidth. This last optimization increases data locality and thus global performance but is limited by the available memory space (256 KB for the SPE local store).

### 3.7 Comparison between the SPU and General Purpose Processors (GPP) with SIMD extensions

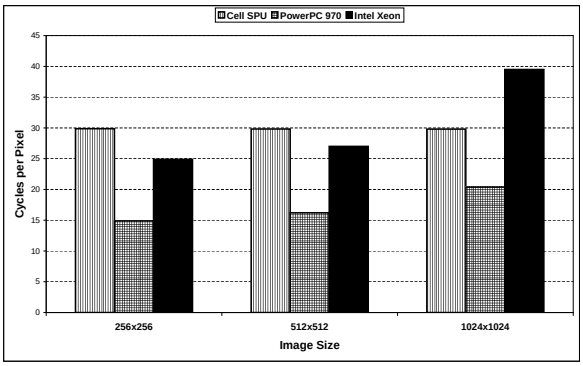


**Fig. 14.** Performance comparison between SPU, Intel Xeon and PowerPC 970

Comparing Cell performance with GPP implementation is interesting as the main difference is the memory management DMA versus cache. Fig. 14 provides the *cpp* of Harris half-pipe version running on 1 thread on 1 core on a PowerPC 970 running at 2.5 Ghz and a Core2Duo Xeon 2.5 Ghz at , with Altivec or SSE SIMD instructions. We can note that GPPs become inefficient when data can not fit in the cache for large image size due to cache replacement policy. Besides, this phenomenon does not occur with DMA controlled transfers : the image size has no impact on performance. This last characteristic is very interesting for computer vision systems as the execution time can be predicted accurately and therefore the system can satisfy a real-time constraint.

### 3.8 Discussion on Benchmarking Methodology

There are several reasons that justify the difference in performance results obtained above and those in [17]:

1. Since we adopted a separated source compilation (one source for the PPU and one for the SPU), we were forced to change the compiler from IBM XLC (ppu-xlc, spu-xlc) to GCC (ppu-gcc, spu-gcc) as in the last release of the Cell SDK (3.0), the XLC compiler became exclusively single-source. This first point can explain the difference in global performance as some compiler optimizations can be performed by XLC and not by GCC and vice versa.
2. The second reason concerns measurement methodology. In the [17] we measured the cycles count in the PPE using the time base counter, with two versions of the code: the first one including computation and the overhead related to thread creation and synchronization and the other one without computation. Then we subtracted the second from the first to get the computation duration. These measures were performed over several runs, and the mean value was taken. However, data presented a great sensitivity to image size. In this paper we took a more representative case were we consider the input data coming from a continuous stream and we make the measure on the PPE with an additional outer loop in the SPEs to process more than one image. This leads to a thread overhead becoming negligible when compared to the pure processing time. As this time was subject to big variation in the [17] we chose this method to attenuate its effect.
3. The last reason is about the computation tiles. In [17], the tile size is fixed to $16K$ and its width $w$ always equals the image width $W$ but its height $h$ varies with $W$, typically $h = \frac{16K}{w}$. As stated in [1] $h$ has an influence on the amount of reloaded data for a tile (when $h$ increases this amount decreases), we decided to adopt a new measurement methodology where we consider a $16K$ tile with $h$ and and $w$ being constants (in our case $h = 16$ and $w = 256$) in order to eliminate this perturbation. Hence, the *cpp* is less sensitive to image size, which is more coherent as the tile size is a constant whatever the image size is. Table 6 gives the percentages of reloaded data with different couples of $(h, w)$. From these values, we conclude that the amount of reloaded data explains the great sensitivity to image size for the left histograms in Fig. 12 and Fig. 13.

| **Image Size $H \times W$** | **Tile Size $h \times w$** | **Total Reload Ratio %** |
|---|---|---|
| $256 \times 256$ | $16 \times 256$ | 11 |
| $512 \times 512$ | $8 \times 512$ | 20 |
| $1024 \times 1024$ | $4 \times 1024$ | 33 |

**Table 6.** Illustration of the difference of the amount of reloaded data depending on the tile height for the Halfpipe+Halfchain model

## 4 Scalability Measure on the Cell Processor

In this section, we evaluate the scalability of the Cell processor by both measuring and modeling *Speedup* and *Efficiency* metrics. The measurements provide informations on how the Cell architecture scales to the Harris algorithms when varying the number of used SPUs. Modeling those metrics allows the prediction of scalability when considering future Cell generations with more accelerator units (SPUs).

**Amdahl's Law** In the basic formulation of the Amdahl's law the execution time of any algorithm on a sequential machine is divided into two parts: the time to execute the pure sequential part

of the code $Seq_0$ and the time to execute the part of the code that is parallelizable $Par_0$. For a machine containing $p$ processors the execution time will have the following expression:

$$T(p) = Seq_0 + \frac{Par_0}{p} \tag{13}$$

**Driscoll and Daasch Reformulation** The last formulation is not appropriate to predict performance on the Cell processor. The sequential portion of the code consisting in the cost of the thread creation, communications and synchronization of the threads. These parameters depend on the number of processors, namely they increase when the number of processors increases. The parallel portion of the code is also a function of p. These assumptions was made in [4] by Driscoll et *al.* We concluded after measurements that :

$$Seq(p) = a_s p + b_s \tag{14}$$

and

$$Par(p) = a_p p + b_p \tag{15}$$

Where $a_s$, $b_s$, $a_p$, $b_p$ are constants that we measured in our experiments. This leads to the following expression of the execution time:

$$T(p) = a_s p + b_s + \frac{a_p p + b_p}{p} = a_s p + b_s + a_p + \frac{b_p}{p} \tag{16}$$

One of the main characteristics of parallel architectures that reflect there performance is their scalability. Scalability metrics show how the system adapt to a certain workload when increasing the number of processing units. *Efficiency* and *Speedup* are one of the basic metrics of scalability, they are defined by the following expressions:

$$E = \frac{T_s}{pT_p} \tag{17}$$

$$S = \frac{T_s}{T_p} \tag{18}$$

Where $T_s$ is the execution time on one processor , $p$ the number of processors and $T_p$ is the execution time on $p$ processors. If we replace $T_p$ and $T_s$ by the expression in Eq. 16 we find the following expression of the efficiency:

$$E = \frac{a_s + b_s + a_p + b_p}{a_s p^2 + (a_s + a_p)p + b_p} \tag{19}$$

This leads to an efficiency decreasing when $p$ increases. The expression of the speedup is of the form :

$$S = \frac{(a_s + b_s + a_p + b_p)p}{a_s p^2 + (a_s + a_p)p + b_p} \tag{20}$$

Which is also a decreasing function of $p$. One must know that that the expressions above was evaluated when considering one input image for the algorithm.
When processing an image stream which is typically captured by a video camera, we can consider the cost of thread creation and synchronization ($Seq(p)$ in the formulas) negligible comparing to the parallel time (computation). We derive equations 21, 22 and 23 assuming that $Seq(p) = 0$

$$T(p) = \frac{Par(p)}{p} = a_p + \frac{b_p}{p} \tag{21}$$

$$E = \frac{a_p + b_p}{a_p p + b_p} \tag{22}$$

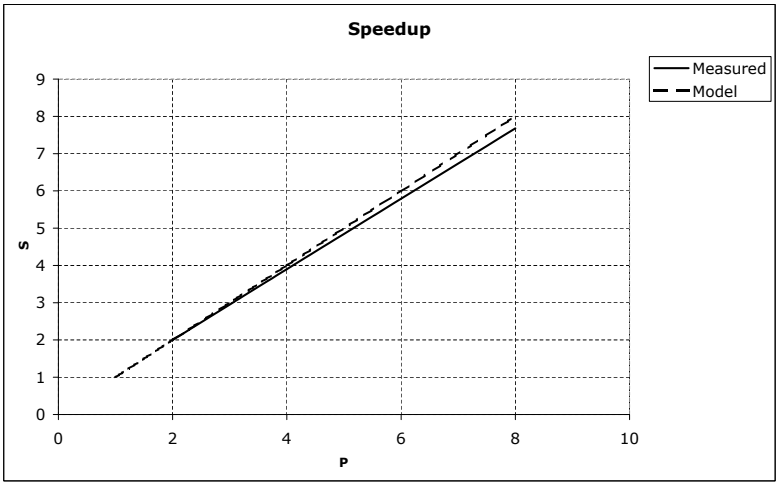


**Fig. 15.** Speedup measure for a stream of 1000 images of size$1024 \times 1024$

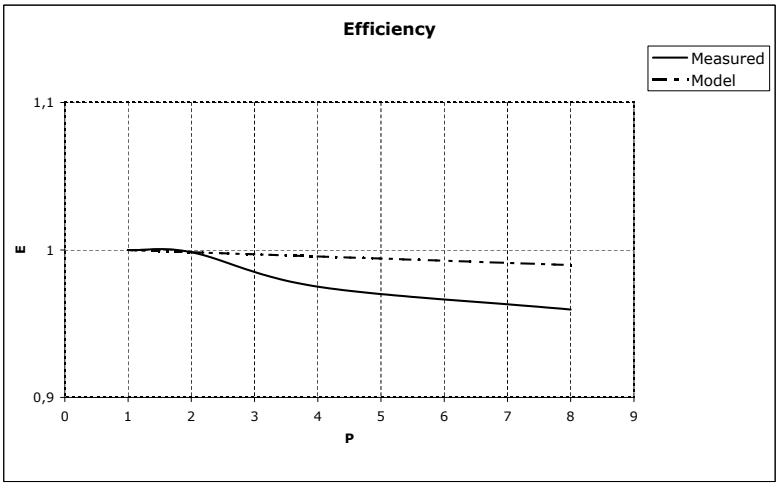


**Fig. 16.** Efficiency measure for a stream of 1000 images of size$1024 \times 1024$

$$S = \frac{(a_p + b_p)p}{a_p p + b_p} \tag{23}$$

These expressions gives a speedup increasing with $p$ until a saturation value of $(1 + \frac{b_p}{a_p})$ and an efficiency decreasing with $p$ but slower than in the previous case. As we observe in Fig. 15 and Fig. 16 the measured scalability metrics matches our model as speedup increases with $p$ and efficiency decreases with $p$. The measurements were performed with varying the number of used SPUs and

the execution time one SPU served as the sequential time $T_s$ ($T(p = 1)$).
From the experiments above we can conclude on how the Cell processor scales to our application which is a data-parallel/memory-bounded problem by making this two assertions:

- The current Cell processor with eight SPUs has a good scalability as the *Speedup* is close to the number of the working SPUs and *Efficiency* is close to 1.
- In the future releases of the Cell where there would be more SPUs, scalability will not be as good as we measured because an increasing number of cores leads to the decrease of *Efficiency* and thus to a saturating *Speedup*.

## 5 Conclusion and Future Work

In this paper, we investigated how a sample image processing algorithm - the Harris corner detector - can be efficiently implemented while taking into account the various architectural particularities of this processor. We explore, contrary to previous works, other models than the simple SPMD parallelization technique. We explored a variety of parallelization schemes that took advantage of the main architectural features of the Cell: a DMA based, distributed memory. The different optimization techniques, *Domain Specific* or those related to the Cell architectures were analyzed. By combining each step of our algorithm in various manners, we demonstrated that chaining and pipelining operators had a large impact on global performance of the application. Our schemes were benchmarked on the Harris corner detector because it features both point-to-point and convolution operations, making it a realistic sample of more complex image processing library. We also proposed a model of efficiency and scalability for the Cell processor in order to be able to predict performance of future releases of the machine with a greater number of SPEs. Future works includes : a deeper analysis of the relation between tiles shape and size and the overall algorithm performance. Other optimization techniques such as multi-buffering will be explored in the future.

## References


1. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*. IEEE Computer Society, 2006.
2. P. Bellens, J.M. Perez, , R.M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *in Proceedings of the ACM/IEEE Supercomputing Conference*, 2006.
3. Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing*, 2006.
4. Michael A. Driscoll and W. Robert Daasch. Accurate predictions of parallel program execution time. *J. Parallel Distrib. Comput.*, 25(1):16–30, 1995.
5. A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engineTM architecture. *IBM Syst. J.*, 45(1):59–84, 2006.
6. Kayvon Fatahalian, Timothy J. Knight, and Mike Houston. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.
7. J. Greene and R Cooper. A parallel 64k complex fft algorithm for the ibm/sony/toshiba cell broadband engine processor. In *Global Signal Processing Expo*, 2005.
8. C. Harris and M. Stephens. A combined corner and edge detector. In *4th ALVEY Vision Conference*, 1988.
9. IBM. *Cell Broadband Engine Programming Handbook*. IBM, version 1.0 edition, 2006.
10. X. Ramirez A. Jimenez-Gonzalez, D. Martorell. Performance analysis of cell broadband engine for high memory bandwidth applications. In *in Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, 2007.
11. Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.
12. Michael D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.

13. Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University*. September 1980. Available as Stanford AIM-340, CS-80-813 and republished as a Carnegie Mellon University Robotics Institue Technical Report to increase availability.
14. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1):85–102, 2006.
15. Fabrizio Petrini, Gordon Fossum, Juan Fernandez, Ana Lucia Varbanescu, Mike Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *IEEE/ACM International Parallel and Distributed Processing Symposium*, march 2007.
16. D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, January 2006.
17. Tarik Saidani, Stéphane Piskorski, Lionel Lacassagne, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor: a case study of image processing algorithm. In *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance*, pages 9–16, New York, NY, USA, 2007. ACM.
18. Harald Servat, Cecilia González-Alvarez, Xavier Aguilar, Daniel Cabrera-Benitez, and Daniel Jiménez-González. Drug design issues on the cell be. In *HiPEAC*, pages 176–190, 2008.
19. Hans Vandierendonck, Sean Rul, Michiel Questier, and Koen De Bosschere. Experiences with parallelizing a bio-informatics program on the cell be. In *HiPEAC*, pages 161–175, 2008.
20. Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.