

The Performance of Gridding/Degridding on the Cell/B.E.

A.L. Varbanescu A. van Amesfoort H.J. Sips T. Cornwell
B. Elmegreen A. Mattingly G. van Diepen R. van Nieuwpoort

Abstract

In this report we present our experience with porting two very time consuming kernels from radioastronomy imaging, i.e. the gridding/degridding operations. Both these procedures are implemented using convolutional resampling, a bandwidth-limited application which is not trivial to parallelize on a hybrid memory architecture. We briefly discuss the role of the application in the radioastronomy context, and we present the Cell/B.E. multi-core processor as an interesting target for this application. Further, we show how the original reference implementation (sequential, C++ code) was not “Cell-friendly”, which is also why our first parallel version had poor results. Next, we show how the reference algorithm was tuned to become more suitable for Cell parallelization. We discuss how we have improved our parallelization strategy and we present our performance results with the new tuned algorithm running on the Cell/B.E. processor, showing a speed-up factor of over 20 when compared with the reference implementation. We conclude that a single Cell/B.E. can provide good results for a small-scale version of the radioastronomy imaging problem, and we sketch our future work directions towards using a parallel machine with multiple Cell/B.E. processors to address the real scale of the of the gridding/degridding applications.

1 Introduction

Due to large radioastronomy projects like ASKAP or LOFAR, followed by even larger multi-national projects like SKA, radioastronomy has become one of the very active high performance computing fields. Significantly increased data volumes, combined with streaming requirements and complex algorithms, represent interesting challenges for supercomputers and new opportunities for multi-core processors.

In the work reported here, we focus on a specific radioastronomy imaging application and its implementation on the Cell/B.E., a heterogeneous multi-core processor jointly built by IBM, Sony and Toshiba. Despite the complex programming effort, described in the following sections, the performance we have obtained (a speed-up factor of over 20 for one Cell/B.E. when compared with a commodity machine) is encouraging. Furthermore, the predicted application scalability towards larger problem sizes is an additional incentive for using the Cell/B.E. as the next generation HPC hardware for radioastronomy.

The rest of this report is structured as follows. Section 2 briefly discusses our target application and its role in the radioastronomy field, while Section 3 presents the Cell/B.E. processor, emphasizing the features that make it both very performant and difficult to program. Next, in Section 4 we discuss the reference application in detail, focusing on showing its main challenges for an efficient Cell/B.E. parallelization. We present our first parallelization attempt and its results on the Cell/B.E. Based on our empirical analysis, backed by the results of this first parallel version,

we show why and how the algorithm requires data-arrangement tuning to become efficient on a hybrid memory multi-core like the Cell. Section 5 presents the performance of this new, tuned reference code, for both the sequential and parallel versions. Finally, Section 6 summarizes our findings and concludes our report.

2 Radioastronomy - a primer

The history of radio astronomy has been one of solving engineering problems to construct radio telescopes of continually increasing angular resolution. Currently, this is achieved by using the principles of interferometry to synthesize a very large effective aperture from a number of small antennas. Each pair of such antennas measures one "Fourier component" of the brightness distribution of the radio source. A sufficient number of Fourier components allows the synthesis of a high-resolution image of the radio sky. The laborious computational task of doing Fourier transforms to obtain images from the interferometer data is accomplished with high-speed computers and specific numeric techniques (like FFT, for example). We focus further on this computation-intensive stage and we discuss the place of convolutional resampling in the imaging processing chain.

Radio synthesis telescopes [5] image celestial objects using measurements of the spatial coherence function of the electric field. The spatial coherence function, also called the visibility function, is the time-averaged cross-correlation of the electric fields as measured at two spatially separated locations. This measured quantity $V(u, v)$ is the two dimensional Fourier transform of the sky brightness $I(l, m)$ where the spatial frequencies (u, v) measure the vector between the two locations (in wavelengths) as projected onto a plane perpendicular to the line of sight, and the (l, m) are direction cosines relative to the line of sight. For small fields of view, the relationship is described by Equation 1.

$$V(u, v) = \int I(l, m) \exp^{2\pi j(ul + vm)} dl dm \quad (1)$$

An image can thus be reconstructed by the inversion of this equation, requiring an inverse transform of visibility samples. The samples are obtained by pairs of antennas observing a source over many hours. The distance between the two antennas is called a *baseline*. For a system comprising of A antennas, measurements are taken from all possible baselines (in total, $B = A \times (A - 1)/2$), and image reconstruction uses all this data to achieve high accuracy. Due to the large amounts of data, the use of a Fast Fourier Transform (FFT) method becomes mandatory for computation efficiency. In turn, FFT requires the irregularly visibility samples to be transferred to a regularly-spaced grid. This is usually accomplished using the convolutional resampling algorithm [5], which deposits each sample onto a 2D grid using a weighting function such as a Gaussian or, more recently, prolate spheroidal function, with width (typically seven or nine pixels) and oversampling (up to 100 subsamples per pixel) chosen to minimize aliasing. This algorithm is efficient and has well understood performance. In recent years, the choice of the convolution function has expanded to allow correlation of wide-field imaging artifacts [2], in which case the convolution function support can reach 100 pixels or more.

3 Cell/B.E. Overview

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor, initially designed by Sony, IBM and Toshiba for the Playstation 3 (PS3) game console. Given the Cell/B.E.'s peak

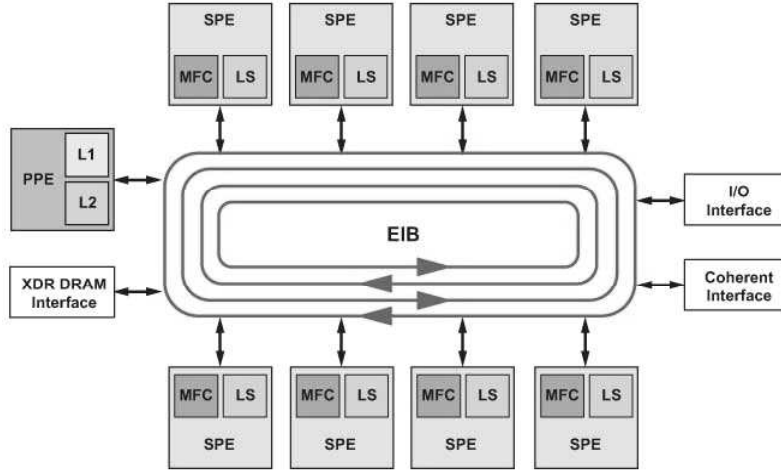


Figure 1: The Cell Broadband Engine.

performance of 230 single precision GFLOPS [4], it was quickly considered a good target platform for HPC applications.

The Cell/B.E. has nine cores: the Power Processing Element (PPE), acting as a main processor, and eight Synergistic Processing Elements (SPEs), acting as co-processors. All cores, the main memory, and the external I/O are connected by a high-bandwidth Element Interconnection Bus (EIB). The maximum data bandwidth of the EIB is 204.8 GB/s. A block diagram of the Cell processor is presented in Figure 1.

The PPE contains the Power Processing Unit (PPU), a 64-bit PowerPC core with a VMX/Altivec unit, separated L1 caches (32KB for data and 32KB for instructions), and 512KB of L2 Cache. The PPE's main role is to run the operating system and to coordinate the SPEs.

An SPE contains a RISC-core (the SPU), a 256KB Local Storage (LS), and a Memory Flow Controller (MFC). The LS is used as local memory for both code and data and is managed *entirely* by the application. The MFC contains separate modules for DMA, memory management, bus interfacing, and synchronization with other cores. All SPU instructions are 128-bit SIMD instructions, and all 128 SPU registers are 128-bit wide. The integer and single precision floating point operations are issued at a rate of 8, 16, or 32 operations per cycle for 32-bit, 16-bit and 8-bit numbers respectively. The double precision 64-bit floating point operations, however, are issued at the lower rate of two double-precision operations every seven SPU clock cycles.

The Cell/B.E. cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [6] to image processing applications [1] and games. The basic Cell/B.E. programming is based on a simple multi-threading model: the PPE spawns threads that execute asynchronously on SPEs, until interaction and/or synchronization is required. The SPEs can communicate with the PPE using simple mechanisms like signals and mailboxes for small amounts of data, or DMA transfers via the main memory for larger data.

4 The Application

In this section we analyze the computation and data access patterns for the original convolutional resampling kernel (used for both gridding and degridding) and we discuss its parallelization on the Cell/B.E. architecture.

4.1 Application Analysis

Figure 2 presents a schematic picture of the typical deconvolution phases. Note that the gridding and degridding operations, both performed by convolutional resampling, are the main targets for our parallelization, due to their long execution times.

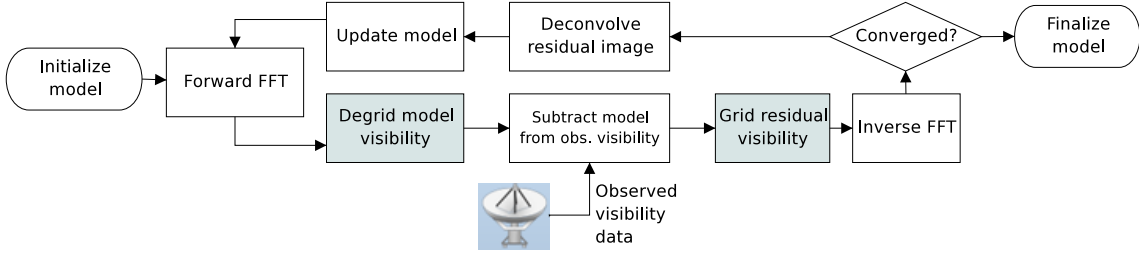


Figure 2: A diagram of the typical deconvolution process in which a model is iteratively refined by multiple passes. The shaded blocks (gridding and degridding) are both performed by convolutional resampling.

Gridding is the transformation from the irregularly sampled measured visibilities, $V(u_j, v_j)$, to “interpolated” gridded visibilities, $G(u_k, v_k)$, which are placed at the points of a $2^g \times 2^g$ regular grid - see equation 2 for a formal expression. Degridding is the reverse operation, used to obtain estimates of the *corrected* data, $V'(u_j, v_j)$ from the gridded image - see equation 3 for a formal expression.

$$G(u_k, v_k) = \sum_{j=1}^{M \cdot OS} (\mathbf{C}_{2D}(u_k - u_j, v_k - v_j) \times V(u_j, v_j)) \quad (2)$$

$$V'(u_j, v_j) = \sum_{k=1}^{M \cdot OS} ((\mathbf{C}_{2D}(u_j - u_k, v_j - v_k) \times G(u_k, v_k)) \quad (3)$$

Both these operations are performed by convolution with sub-blocks from $\mathbf{C}_{2D}$, a large matrix that contains all potential coefficients for the chosen convolution function; $\mathbf{C}_{2D}$ is also oversampled, i.e., the convolving sub-block is finer than the resulting grid by a factor of $OS = os \times os$. A convolution sub-block, SK_M of $M = m \times m$ elements is called a *support kernel*. Typical sizes for M are 7×7 to 9×9 , depending on the required accuracy level. Note that for two consecutive grid points, $G(u_k, v_k)$ and $G(u_{k+1}, v_{k+1})$, the corresponding SK_M^k and SK_M^{k+1} may not be adjacent.

The $\mathbf{C}_{2D}$ matrix is defined for coplanar baselines (i.e., for narrow fields of view). In the more common case of non-coplanar baselines (the one we discuss further in this paper), a third spatial frequency, w , is necessarily taken into account to preserve accuracy. Processing $V(u, v, w)$ requires an additional w -correction on the visibility data. For this correction we use the W-projection algorithm, which gives a method to generate a new convolution matrix $\mathbf{C} = \mathbf{C}_{2D} \otimes \mathbf{Wp}$ by taking into account projections of the w coordinate on P_w predetermined planes ($\mathbf{Wp}$ accounts for the

Table 1: The main data structures of the convolutional resampling and their characteristics

Name	Symbol	Cardinality	Type	Access pattern
Coordinates/baseline	u, v, w	$N_{samples}$	Real	Linear
Visibility data	V	$N_{samples} \cdot N_{freq}$	Complex	Linear
Convolution matrix	$\mathbf{C}$	$M \cdot os^2 \cdot P_w$	Complex	Irregular
Grid	$\mathbf{G}$	512 x 512	Complex	Irregular

Fresnel diffraction term [3] on each one of the projection planes). $\mathbf{C}$ is P_w -times larger than $\mathbf{C}_{2D}$ (in fact, there is one $\mathbf{C}_{2D}$ for each w plane). The size of SK_M also increases to values as large as $M = 129 \times 129$.

The Data

The essential application data structures are summarized in Table 1. Data is collected from A antennas (i.e., $B = A \cdot (A - 1)/2$ baselines); in one observation session, each baseline produces $N_{samples}$ for each one of the N_{freq} channels. For typical values like $M = 45 \times 45$, $OS = 8 \times 8$ and $P_w = 33$, $\mathbf{C}$ has more than 4 million elements. The size of the support kernel (SK_M) varies between $M = 7 \times 7$ and $M = 83 \times 83$. For a typical sampling rate of 1 sample/10 s, $N_{samples}$ varies between 2880 for an 8 hours-observation and 4320 for a 12 ours one.

We run all the experiments presented in this paper with three types of input data: (1) random data, (2) real data collected by one baseline (one pair of antennas), and (3) data collected by ten baselines, shuffled in order to address the case when ten data streams have to be concurrently imaged. We use samples for 8 hours measurements (i.e., one baseline produces 2880 samples). The random data has no radioastronomy-based constraints. The measured data from one baseline has the following property: $(u, v, w)_{t+1} = f((u, v, w)_t)$. Furthermore, the variation between the two consecutive coordinates is quite small, which in turn leads to very good temporal locality whne accessing both $\mathbf{C}$ and $\mathbf{G}$. Thus, having all baselines separated and processed individually in chronological order provides the highest data locality, but may also lead to resource wasting. Finally, for the case of ten shuffled baselines, some of the data locality may be preserved, but the shuffling will definitely diminish it. Please note that 990 shuffled baselines will be much closer, in terms of data locality, to the case of random data than to the case of single, sorted, baseline data.

The Computation

There are three main time-consuming blocks in our reference implementation: (1) the computation of $\mathbf{C}$ (12.8% of the overall execution time), (2) the gridding (35.4%), and (3) the degridding (48.74%). The rest of stages present in Figure 2 are not part of this parallelization effort.

The computation of $\mathbf{C}$ consists of a series of large nested loops that iterate over the entire matrix; for each element, one relatively simple computation (i.e., one exponential or one cosine) is performed. Because $\mathbf{C}$ is computed only once for a set of visibilities (the matrix is being re-used to process all data from all baselines), an optimized calculation of $\mathbf{C}$ has little impact on the overall application performance. Thus, we focus on parallelizing the gridding and degridding operations for one baseline. This computation is presented in Listing 1. Note that the effective computation for both operations is only one multiply-add operation. The large execution time of this kernel is the result of two causes: the large iteration space ($N_{freq} \times N_{samples} \times M$), and the irregular access patterns in both $\mathbf{C}$ and $\mathbf{G}$ matrices, which leads to very poor cache efficiency.

Listing 1: The core of the convolutional resampling kernel

```

1  forall (i=0..N_freq; j=0..N_samples-1) // for all samples
2      compute g_index=func((u,v,w)[j], freq[i]);
3      compute c_index=func((u,v,w)[j], freq[i]);
4      for (x=0; x<M; x++) //sweep the convolution kernel
5          if (gridding)    G[g_index+x] += C[c_index+x]*V[i,j];
6          if (degridding) V'[i,j] += G[g_index+x]*C[c_index+x];

```

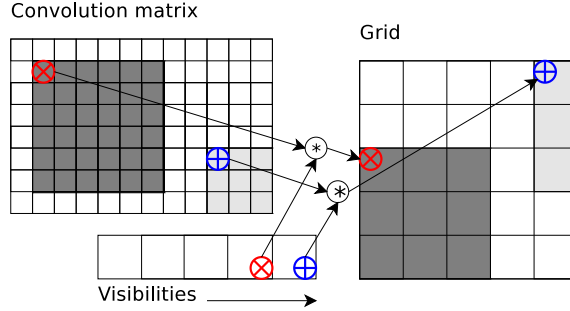
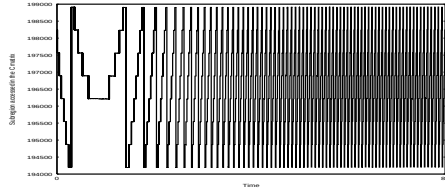


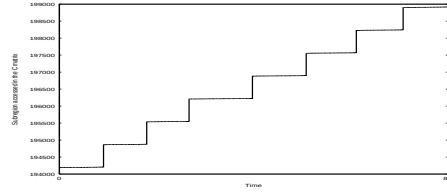
Figure 3: The data access patterns of the gridding/degridding operations. $\oplus$ and $\otimes$ mark items used in the same computation iteration; $*$ shows multiplication; new entries in the grid are always added on the existing values.

The Access Patterns

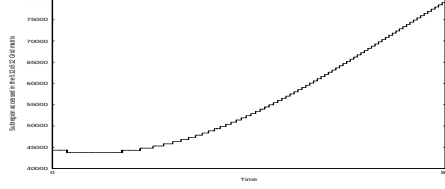
Each visibility sample uses a specific support kernel from $\mathbf{C}$ to contribute to a sub-grid from $\mathbf{G}$. However, the subregions accessed in consecutive steps are not adjacent - see Figure 3. These irregular accesses to $\mathbf{C}$ and $\mathbf{G}$, generated because $\mathbf{c_index}$ and $\mathbf{g_index}$ are computed as functions of the $(u, v, w)_j$ coordinates, lead to a data-dependent behaviour of the application. In turn, due to this dynamic behaviour, (1) temporal data locality is very low, and (2) a parallel data layout of the application is difficult to design. Figure 4 gives a measure of how irregular these accesses are by plotting the positions in $\mathbf{C}$ and $\mathbf{G}$ from where consecutive accesses are requested for the three input data sets: one baseline, ten shuffled baselines, and random data equivalent with ten baselines. On the left-hand side of the graphs, one can see the accesses in the execution order; the right-hand side graphs show the results for the same data, only sorted. While the original access patterns are very sparse, thus providing low temporal and spatial locality, the sorted graphs show that data locality does exist in the application, but a way to sort the accesses must be found. Note that radioastronomers consider pre-sorting is not an option (especially in a streaming environment).



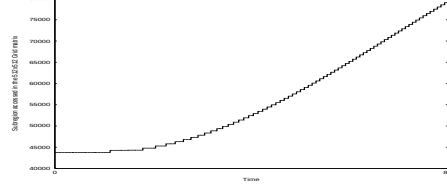
(a) C, 1 baseline, un-sorted



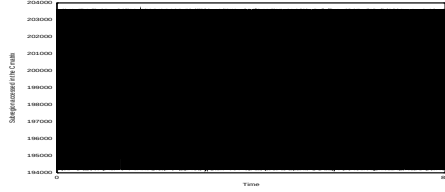
(b) C, 1 baseline, sorted



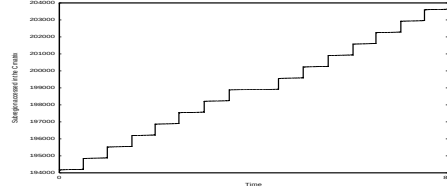
(c) G, 1 baseline, un-sorted



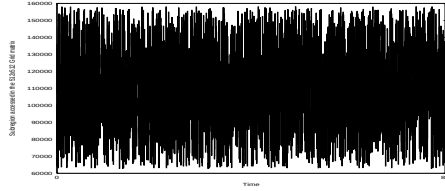
(d) G, 1 baseline, sorted



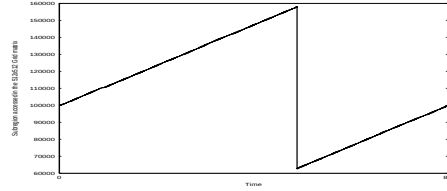
(e) C, 10 baselines data, un-sorted



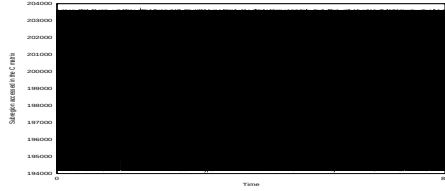
(f) C, 10 baselines data, sorted



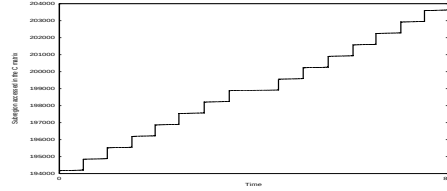
(g) G, 10 baselines data, un-sorted



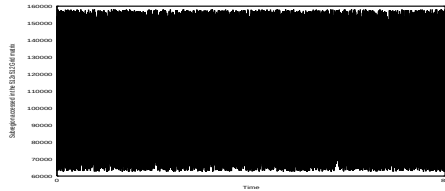
(h) G, 10 baselines data, sorted



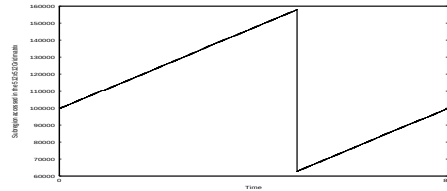
(i) C, 10 baselines, random, un-sorted



(j) C, 10 baselines, random, sorted



(k) G, 10 baselines, random, un-sorted



(l) G, 10 baselines, random, sorted

Figure 4: Access patterns in **C** and **G**, with various data and data-sizes, original and sorted.

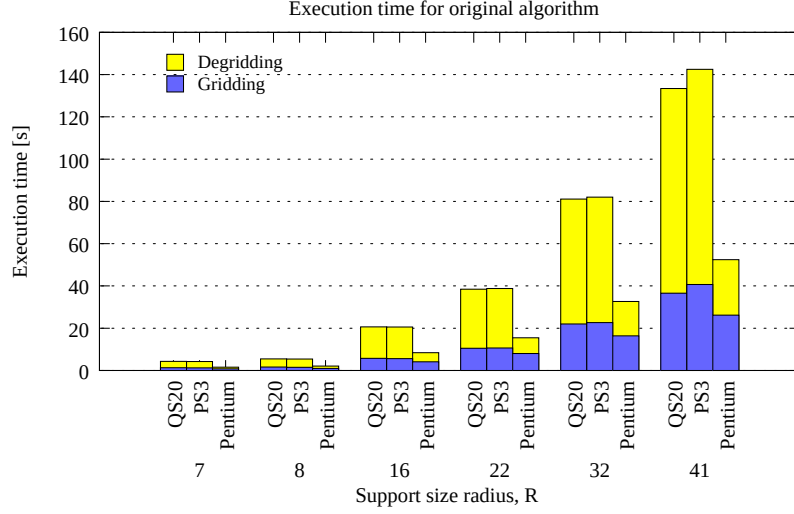


Figure 5: Execution time of the original application

4.2 Sequential performance

We have measured the execution time of the sequential application on a PentiumD, a PS3 and a Cell blade. We have used the PentiumD processor as an instance of a general purpose processor; the choice for this particular machine was only due to availability, and we use the execution time on PentiumD only as a measure of the performance the reference sequential code ¹, not as a measure of the potential performance of the processor. Figure 5 presents a comparison of the execution times of the original application on these three platforms, for six different support kernel sizes (note that $SK_M = m \times m$). Further, we also plot the execution time per data sample, as well as the execution time per grid-addition (significant metrics for the radioastronomers), for the same six different sizes of the support kernel. The graphs are presetned in Figure 6 and Figure 7.

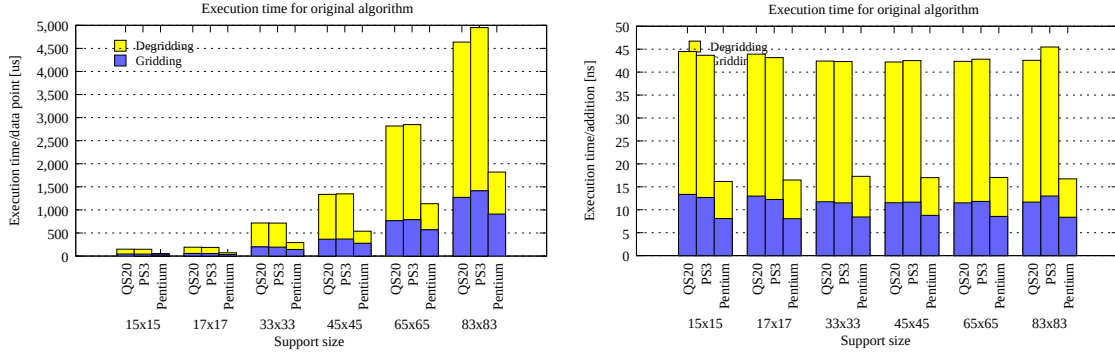


Figure 6: Execution time for each data point for Figure 7: Execution time for each matrix addition for various support kernel sizes: $SK_M = m \times m$ elements

¹The execution time of the PentiumD can also be reduced by applying similar optimizations.

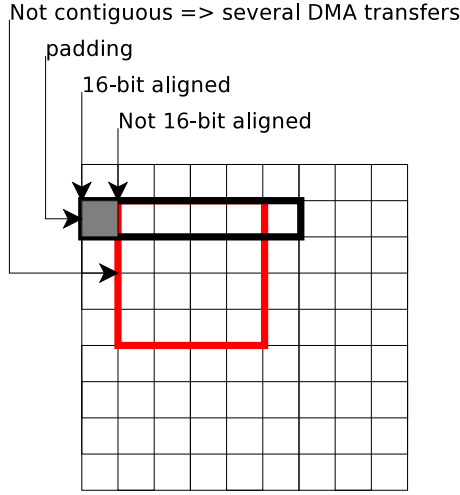


Figure 8: DMA transfer of a large matrix subregion

4.3 Cell issues and solutions

The ideal scenario for running this algorithm on Cell would be to move all computation on the SPEs. Note that the amount of computation is small when compared to the amount of data required for it. For the SPEs, this translates to a very low computation-to-communication ratio, because data has to be brought in the LS before usage. This communication is done by DMA. Note that there are a few important restrictions related to the DMA transfers (see Figure 8):

- The maximum size of one DMA transfer is 16KB. This means only a subregion of 32×32 DP complex numbers or 32×64 SP complex numbers can be transfered at once. Any larger region requires several successive DMA transfers.
- The DMA transfers work with *16-bit aligned addresses*. This constraint may result in data padding.
- The DMA transfers only work on contiguous memory areas. Thus, a rectangular subregion from a bigger matrix cannot be transfered in a single transfer; most likely, transferring such a subregion will require a number of transfers at least equal with the number of subregion lines.

In order to increase the efficiency of the application, one must find a solution to increase SPE data re-use. The main candidates for data re-use are the convolution and the grid subregions.

The grid

Each data sample adds-up to a subregion of the grid. The size of such subregion, as well as the size of the convolution support kernel, is $SK_M = m \times m$ elements. Considering the storage space for *both* subregions, the SPE LS may accommodate a grid subregion of 129×129 elements for the SP solution (133KB), or 100×100 elements for the DP solution (160KB).

For the gridding operation, the final grid is composed by addition. Thus, each SPE can store its own Grid subregions and then the PPE or another (available) SPE can recompose the grid. In this context, note the following:

- It is not efficient for the targetted grid subregion to be brought in the LS, as this will require one or several DMA operations. It is easier for the SPE to compute a “clean” subregion which can be stored in a contiguous memory region and can be transfered back to main memory by DMA once full or no longer useful;
- It is not necessarily efficient to store in the SPE LS a grid subregion that is larger than the region computed using by one data sample, as long as future access to the same region cannot be predicted;
- Double-buffering can be implemented for two distinct grid subregions: while one of the regions is computed, the other one, already computed by the SPE, can be DMA-ed to the main memory. However, this solution is very expensive in terms of memory consumption, and we did not explore its performance.

Thus, the space reserved for the grid inside the LS is $m \times m \times \text{sizeof}(\text{Complex})$.

For the degriding operation, each data point is “refined” using its corresponding grid *and* convolution subregions. In this case, the grid subregion must be brought in the SPE LS from the main memory, increasing the DMA-in time of the application. On the other hand, the data that needs to be returned to the main memory is much smaller, thus the DMA-out operation is practically skipped.

The convolution matrix

The computation for one data sample requires a subregion of $m \times m$ elements from the convolution matrix. This subregion has to be brought in the SPE LS line by line (because the subregion is not contiguous in the main memory); once each line is used, it can be discarded and the next line brought in. Thus, the space reserved for the convolution subregion inside the LS will be at least m Real elements. In this case, the use of a multi-buffering scheme is mandatory for hiding (some of the) communication latency with computation. Thus, we should consider at least $m \times \text{sizeof}(\text{Real}) \times \text{BUFFERS}$ as space reserved for the convolution matrix. The number of buffers to be used varies between 2 and 4. Note that the non-contiguous storage of the convolution matrix in the main memory requires a number of m DMA transfers, having a significant impact on the overall gridding performance.

4.4 Application parallelization

We have started by analyzing two parallelization options: full-parallelization and per-point parallelization. We present our analysis on these solutions, as well as their experimental results, in the following paragraphs. Note that the Cell/B.E. experiments are run on two platforms: (1) a PlayStation3 gaming console, and (2) a Q20 Cell blade, a platform for high-performance computing. PS3 has one Cell/B.E. processor, running at 3.2GHz, with six out of the eight SPEs fully available for programming; the Q20 blade has two Cell/B.E. processors, providing uniform access to 16 SPEs. Both machines have Fedora Core 6 Linux and IBM’s SDK2.1 installed for development and testing purposes.

Table 2: The computation and communication operations for each kernel

Kernel	Communication	Computation
Gridding	DMA-in/Mailbox: $1 \times V((u, v, w)_i)$	$m \times m$ MADD
	DMA-in: $m \times m \times Real$, non-contiguous	
	DMA-out : $m \times m \times Complex$, contiguous	
Degridding	DMA-in: $m \times m \times Real$, non-contiguous	$m \times m$ MADD
	DMA-in: $m \times m \times Complex$, non-contiguous	
	DMA-out/Mailbox: $1 \times V'((u, v, w)_i)$	

4.4.1 Full Parallelization

The key point when parallelizing this application is to manage its data distribution. The simplest solution is to choose a symmetrical distribution, i.e., data is split evenly among the SPEs. Each core receives a piece of V , $\mathbf{C}$, and $\mathbf{G}$, and computes its share. This solution should provide linear speed-up when increasing the number of participating cores. However, this is not a suitable option for convolutional resampling, because of the irregular access patterns to the main data structures: one block from V is not corresponding to a computable, compact block in either $\mathbf{C}$ or $\mathbf{G}$. Furthermore, distributing either $\mathbf{C}$ or $\mathbf{G}$ evenly among the computing cores leads to overall load-imbalance, as the accesses in these matrices are clustered around particular sub-regions (see 4); these hot-spots become known only at runtime. Finally, in a symmetrical data distribution of the visibility data V_i , each SPE requests the $(u, v, w)_i$ from the PPE, computes the indexes c_{index_i} and g_{index_i} locally, and then use DMA to bring the respective blocks from $\mathbf{C}$ and $\mathbf{G}$ in LS. After M multiply-add operations, the new pair of indexes, $c_{index_{i+1}}$ and $g_{index_{i+1}}$ needs to be computed and a new DMA transfer takes place. For this scenario, the computation-to-communication ratio is very small due to excessive DMA transfers, thus the SPEs are severely underutilized. The overall speed-up (on 16SPEs) is only about a factor of 3.

We conclude that an arbitrary, symmetrical data distribution cannot work efficiently for the convolutional resampling; due to its dynamic behaviour, the applications requires an adaptive parallel data layout.

4.4.2 Per-point parallelization

The idea behind this solution is to allow SPEs to work on the computation itself, and let the PPE act as a job scheduler. The PPE sends each SPE a data point V_i and the associated c_{index_i} and g_{index_i} values. Thus, the loops from lines 1,2,3 in Listing 1 are done by the PPE, while the SPEs are executing the inner loop (lines 7-9 in the same Listing 1). The PPE uses a round-robin policy, giving each SPE one data sample in each iteration. For this case, the performance bottleneck is the communication time, i.e., the time spent on DMA operations between the SPEs and the PPE. Table 2 presents a summary of the computation and communication requirements for each kernel.

Non-contiguous convolution region To analyze the influence of the DMA transfers on the gridding/degridding application performance, we have measured the SPE performance for different SK_M sizes, separating the DMA and computation times for both griddign and degridding. Figure 9 shows the time taken by the SPE to compute one grid line (i.e., using a single support kernel line), considering code SIMD-ization enabled/disabled, and showing the DMA and computation

separately. Note that the DMA contribution is roughly the same in all cases, namely $T_{DMA} = 400ns$ for the gridding and $T_{DMA} = 800ns$ for the degriding, and it represents the DMA overhead that was still not hidden by the double-buffering. Figure 10 shows the time spent by one SPE to compute one operation (gridding or degriding addition). These times are calculated by measuring the time spent for the entire gridding and degriding, respectively, and dividing it by the total number of operations involved; as a result, it includes the DMA overhead. The smaller the time is, the more efficient the parallelization is.

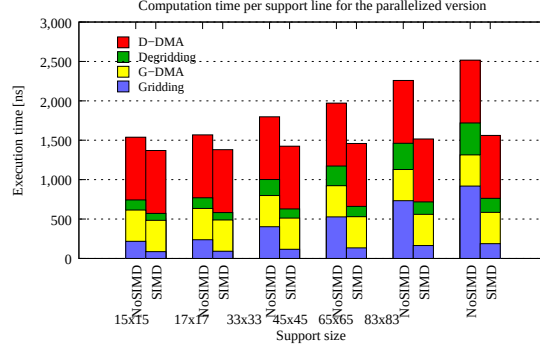


Figure 9: The SPE execution time for one convolution/grid line from the support kernel

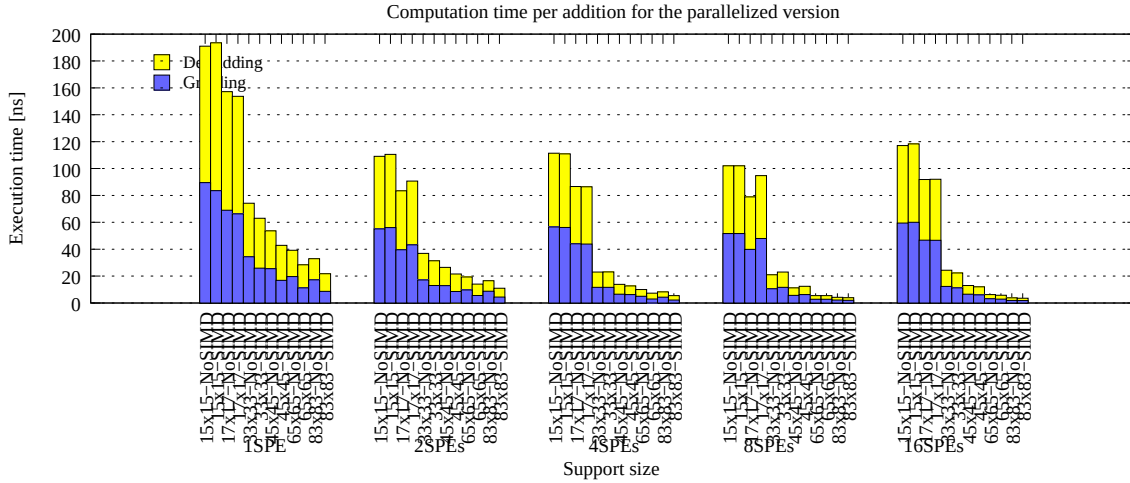


Figure 10: The SPE execution time for one addition

On the PPE side, we have measured the overall execution time of each operation. The results are shown in Figure 11 for one baseline. We have taken measurements also for ten baselines (both real and random data), and the results scale linearly (i.e., the times are 10 times bigger), thus we do not include the graph here.

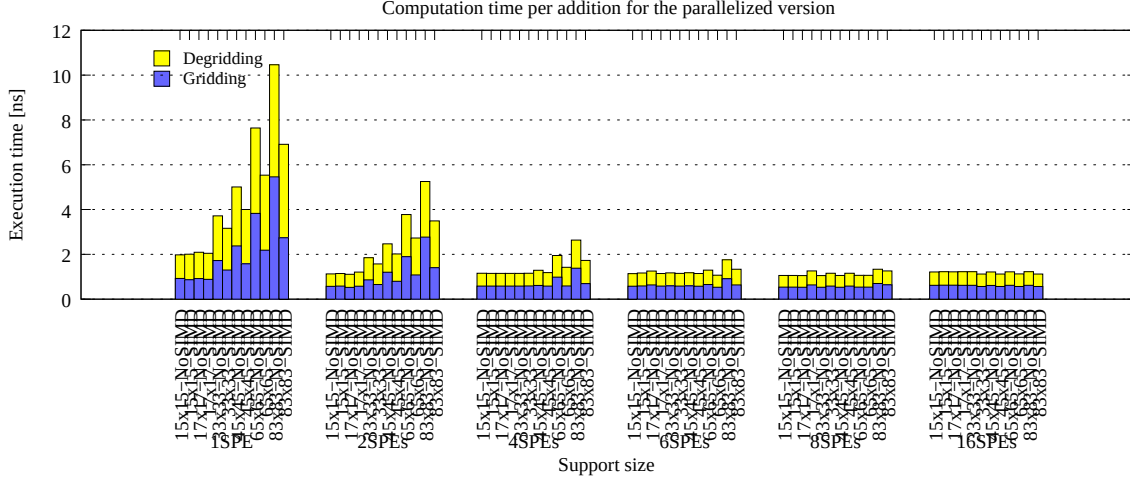


Figure 11: The overall execution time for 2880 data samples (one baseline, entire night)

Contiguous convolution region As a consequence of the non-contiguous region storing the support kernel, the application communication overhead increases linearly with the size of this kernel. To measure the performance penalty of this solution, we have repeated the experiments, only considering the convolution matrix reorganized such that the convolution subregions are contiguous (i.e., the $\mathbf{C}$ is a 3D matrix, for example). In this case, an SPE loads the convolution subregion as a contiguous chunk of size $m \times m$ elements. Figure 12 presents the execution time for one convolution support kernel, separating the DMA transfers from the computation. To emphasize the performance gain, we have normalized the execution per-line, and we have also included the data series from the non-contiguous version, as seen in Figure 9. Note that the performance gain is more than a factor of 2 for large kernel sizes. Further, figure 13 presents the execution time for one add operation; to quickly grasp the performance gain, compare with Figure 10. Note the almost constant value for the computation time, and a visible decrease in performance for small sizes of the subregion matrix, due to the constant DMA overhead. Finally, Figure 14 shows how this performance improvement reflects in the overall application time. Comparing with Figure 11, note that the performance gain (i.e., lower execution time) is between 40% and 60%.

5 The tuned application and its performance

In this section we present the new data arrangement used by the tuned algorithm. Further, we discuss its performance (1) as a sequential application, (2) using the same per-point parallelization strategy as used for the original application, and (3) using an optimized version of this per-point strategy.

5.1 The new application

Based on our measurements showing a significant improvement of the application performance when the convolution support kernel can be stored in a contiguous memory region, a new version

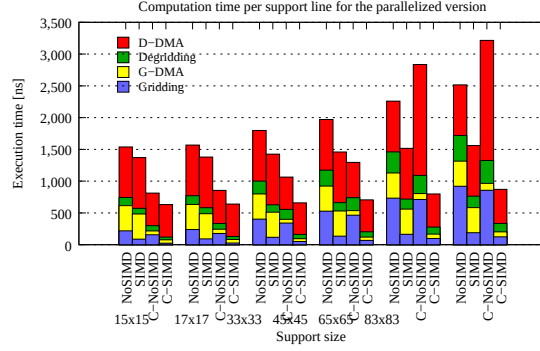


Figure 12: The SPE execution time for one addition (grid/degrid) for a contiguous convolution subregion

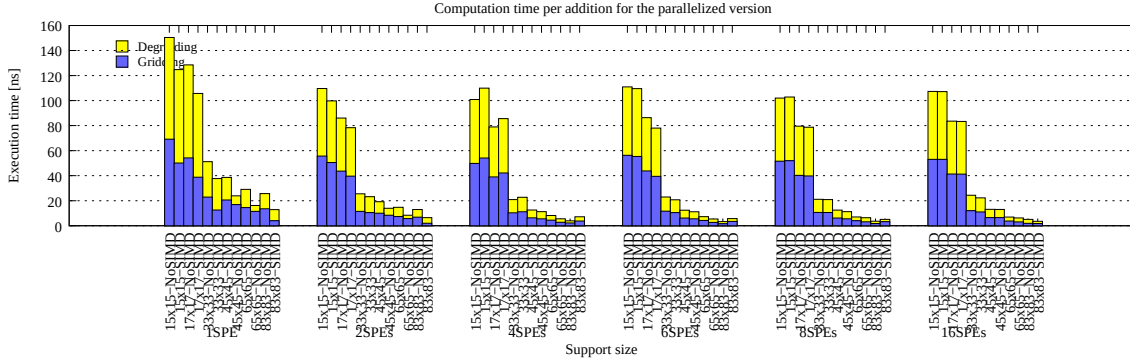


Figure 13: The SPE execution time for one addition, with contiguous access to **C**

of the convolutional resampling algorithm has been implemented. For this version, the data stored in the convolution matrix has been reorganized as a large array of SK_M matrices, each having $M = m \times m$ elements. The drawback for this reorganization is the size increase for the new **C** matrix.

We have tested the **tConvolveNEW** sequential application on the same series of machines as the original application. We present the comparative results in Figure 15, which shows the overall execution time for a single baseline, using support kernels of various sizes, and in Figure 16, which shows the time per addition for the same set of experiments. Note that performance of the new application is better than the performance of the original application by a factor ranging between 27% and 34%.

To analyze the data access patterns distribution of the new application, we have plotted the positions in **C** and **G** from where consecutive accesses for an 8h measurement are requested, in a similar series of experiments as with the original application. The plots are presented in Figure 17. On the left-hand side graph, one can see the accesses in the execution order; the right-hand side graph shows the results for the same data, only sorted. Note that the original access patterns are quite sparse, providing low temporal and spatial locality, while the sorted graphs show better data

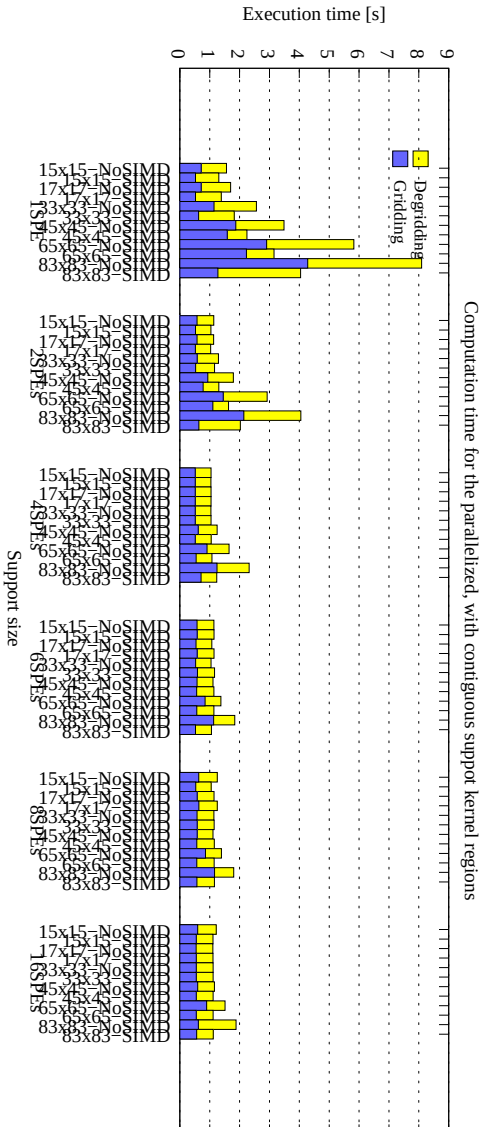


Figure 14: The overall execution time for 2880 data samples (one baseline, entire night)

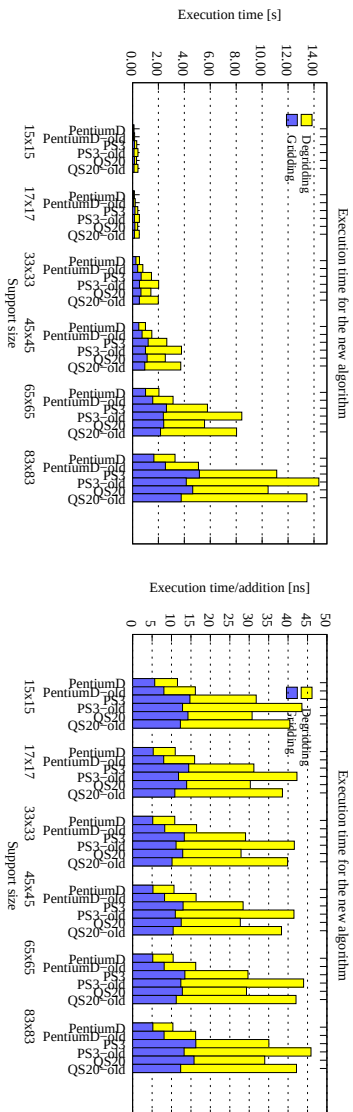


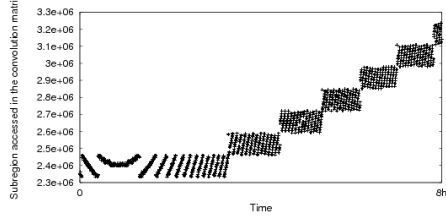
Figure 15: Overall execution time of the grid- Figure 16: Execution time per operation for the grid- degrading/degrading algorithms

locality. We have run the same experiments for random generated data, as well as for 10 baseline files, shuffled. As expected, data locality is severely disturbed in both these other cases².

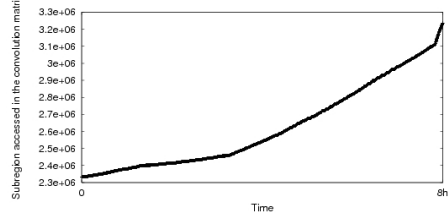
5.2 The improved parallel version

As seen in Section 4, a full-parallelization strategy (i.e., a symmetrical data parallel approach) does not work on the Cell because of the low computation-to-communication ratio of the convolution operation. Thus, we focus on the per-point parallelization strategy. First, we have adapted the same parallelization scheme applied for the original version of the application, and measured its performance. The results for are presented in Figure 5.2(a1), (b1) and (c1). Note that even for the best execution time, which is observed when using 8SPEs on the Cell blade and 6SPEs on the PS3, the speed-up compared to the execution on the PentiumD is not larger than 3, a very poor

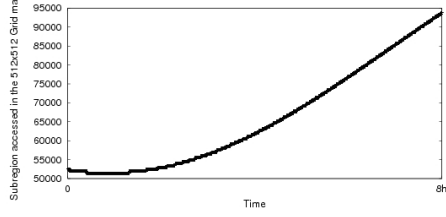
²The plots are not included because the file size and viewing time increase too much.



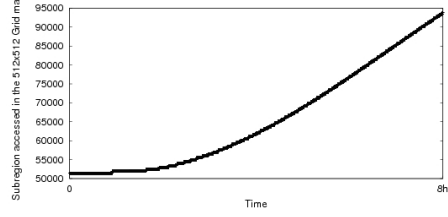
(a) C, 1 baseline, un-sorted



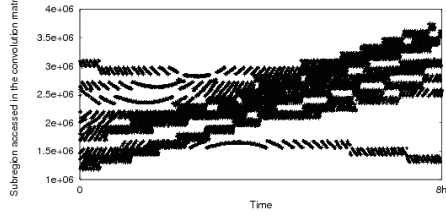
(b) C, 1 baseline, sorted



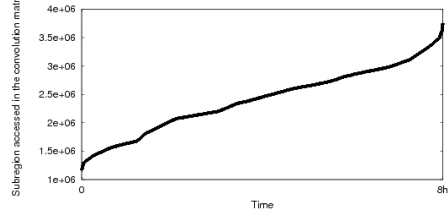
(c) G, 1 baseline, un-sorted



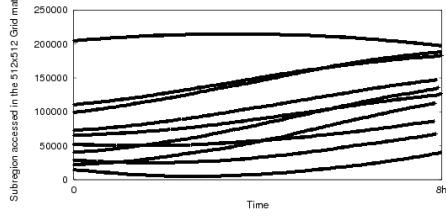
(d) G, 1 baseline, sorted



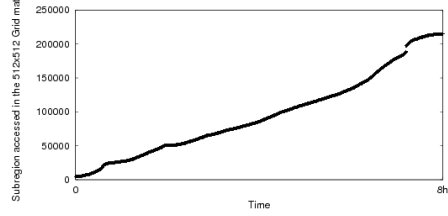
(e) C, 10 baselines data, un-sorted



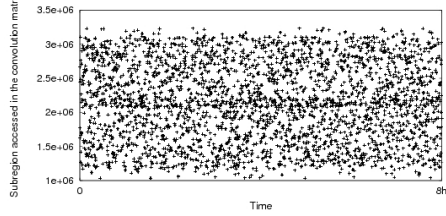
(f) C, 10 baselines data, sorted



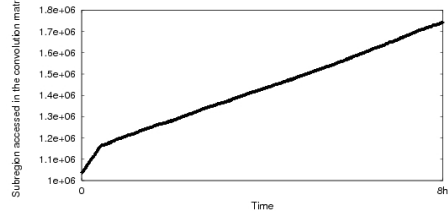
(g) G, 10 baselines data, un-sorted



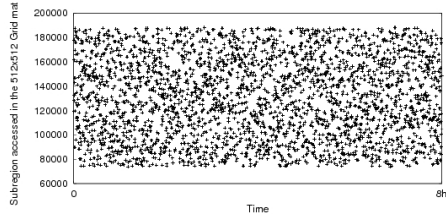
(h) G, 10 baselines data, sorted



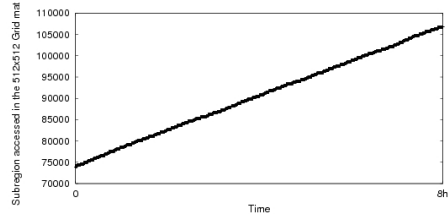
(i) C, random, un-sorted



(j) C, random, sorted



(k) G, random, un-sorted



(l) G, random, sorted

Figure 17: Access patterns in **C** and **G**, with various data and data-sizes, original and sorted.

result for a Cell-like platform.

This result is justified by the severe underutilization of the SPEs, due to the low communication-to-computation ratio of the application. To confirm this hypothesis, we have measured the SPE utilization for this parallel application, for both the SIMD and non-SIMD option. We have run experiments for the meaningful SPE combinations - 1, 2, 4, 6, 8, and 16 active SPEs - as well as support kernel sizes - $m = 15, 17, 33, 45, 65, 83$. The results are presented in Figure 5.2(a), and they show that for the best performance, using 6SPEs or 8SPEs, the SPE utilization does not go beyond 30%.

To make the parallel version more efficient, we have to increase SPEs utilization. To do so, we have improved the parallelization by using one job-queue for each SPE. Thus, the PPE reserves a queue for each SPE, and continuously places jobs in all these queues. The SPEs poll the queues and solve pending work *in chunks, not piece by piece*, thus reducing the communication to the PPE at a minimum. As the PPE has a good overview of all the subregions that are under computation over the SPEs, it also applies scheduling optimizations. For example, it can compare a new request with the ones being processed and decide to send it to an SPE that may already have the right support kernel loaded.

The SPE performs a simple loop: bring in the $\mathbf{C}_{local}$ data via DMA, compute values for $\mathbf{G}_{local}$, and do the DMA-out transfer. We further optimize this loop by checking if the new $\mathbf{C}_{local}$ overlaps with the already available one and, if so, no DMA-in is performed. Similarly, no DMA-out is performed if the $\mathbf{G}_{local}$ versions overlap.

The results of the improved version running on a varying set of SPEs are presented in Figure 5.2(a2),(b2),c(2). Please note the scale difference between the two columns of graphs, a clear indication of the significant performance improvement (by approximately a factor of 10). Also, we have measured the SPE utilization, and we present our results in Figure 5.2(b); they confirm that the SPE utilization has been improved toward a maximum of 95-99% for the best execution times.

Next, we have measured the overall performance improvement of the new job-queue parallelization. The support kernel size was fixed at $M = 83 \times 83$ elements. We have evaluated the execution time of the reference code as well as of the parallelized version running in different configurations on the two available Cell/B.E. platforms. Figure 20 presents our results (with logarithmic scale). The execution time for our Cell/B.E. versions is compared against the execution time of the same reference platforms introduced in Section 4 (a PentiumD machine, and the PPE processors from a Cell-blade and a PS3).

The best execution time is obtained when using 2 Cell/B.E. processors on the QS20 blade (i.e., using 16 SPEs). Note that the performance penalty of the execution on the PS3-Cell (i.e., on 6 SPEs) is less than 15%, which makes the PS3 an efficient alternative to the more expensive QS20. The speed-up factor of Cell/B.E. over the reference execution time is over 20.

Finally, we have measured the execution time for different problem sizes (i.e., support varies between 15×15 and 83×83) running on different numbers of SPEs (1, 2, 4, 6, 8, and 16). We present the results of these experiments in Figure 21. Note that several variants are close to an overall optimal, making it hard to guess which configuration will be the most efficient. Ideally, this optimal configuration should be computed by an automated performance predictor, but such tool is not yet available for the Cell/B.E.

5.3 Multi-baseline parallelization

The questions guiding our future work are: “does the job-queue solution scale up? how?”. We enumerate here three solutions, currently labeled as work-in-progress, for approaching the up-

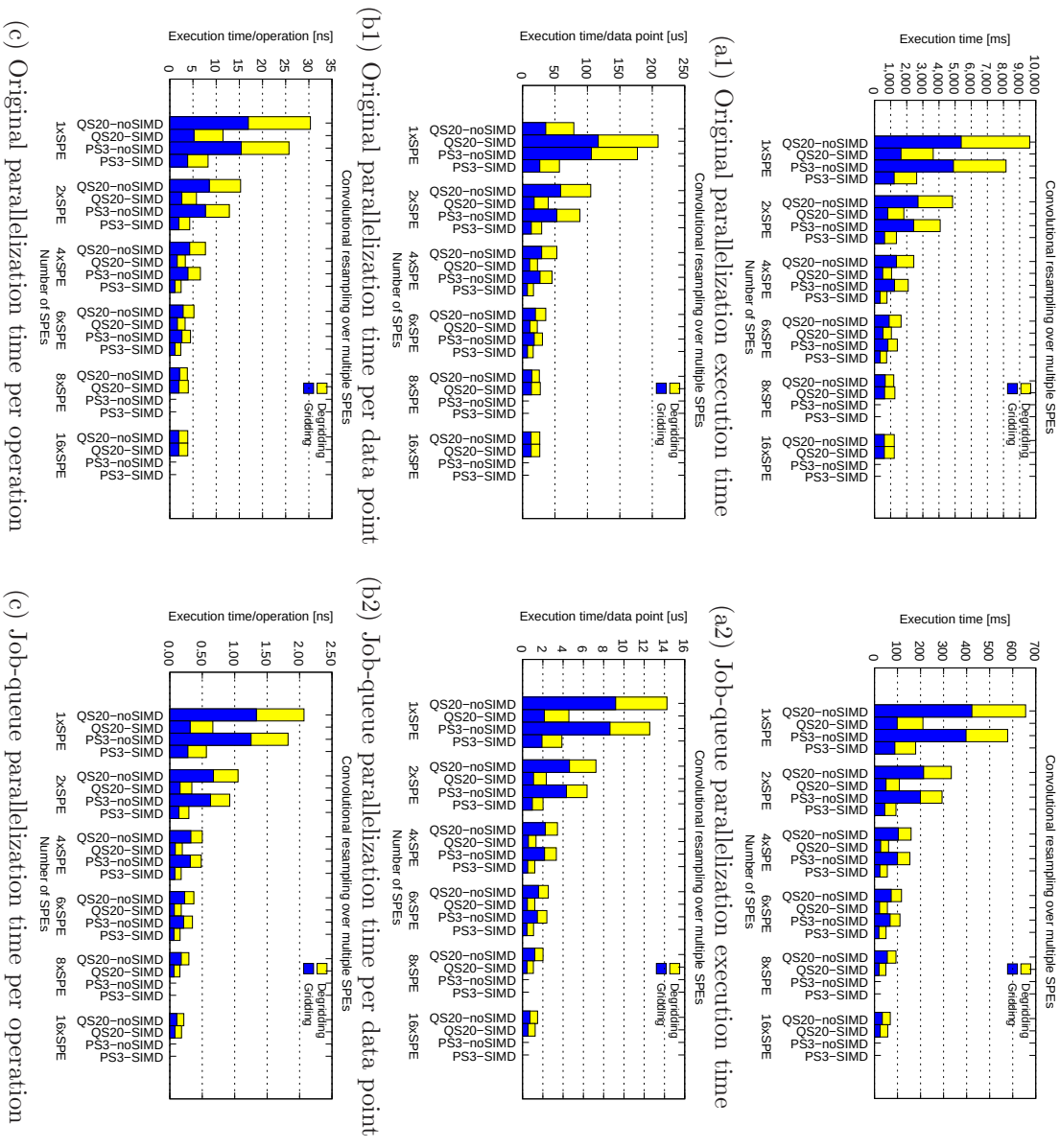


Figure 18: The results using the job queues strategy on the tuned application

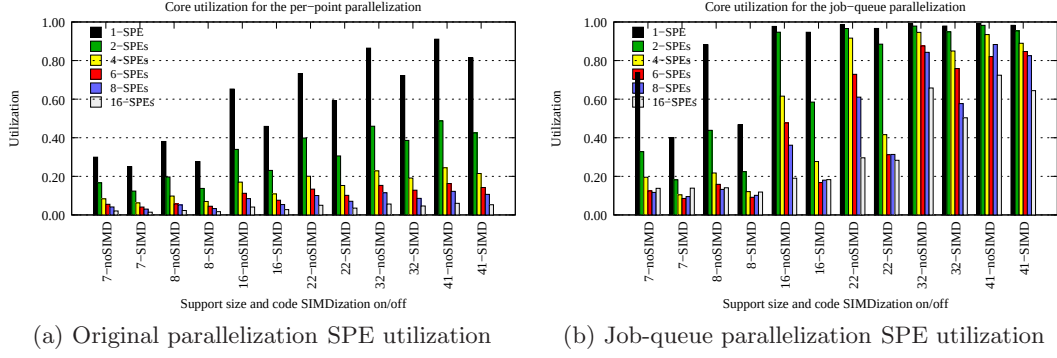


Figure 19: The results using the job queues strategy on the tuned application

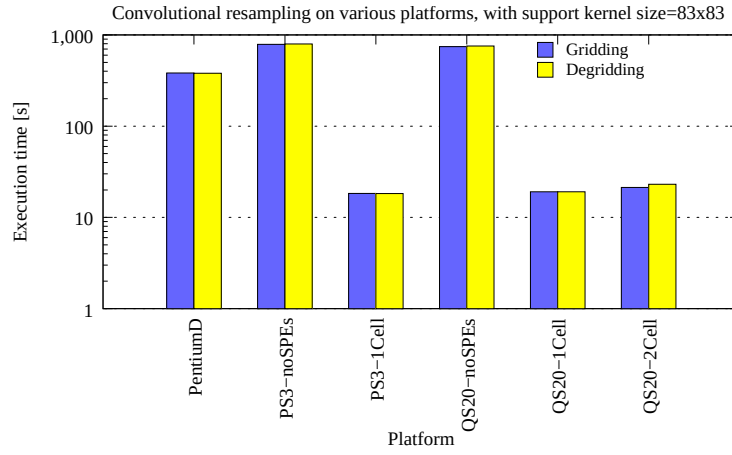


Figure 20: The overall application performance of various instances of the convolutional resampling. The reference support kernel size is 83×83

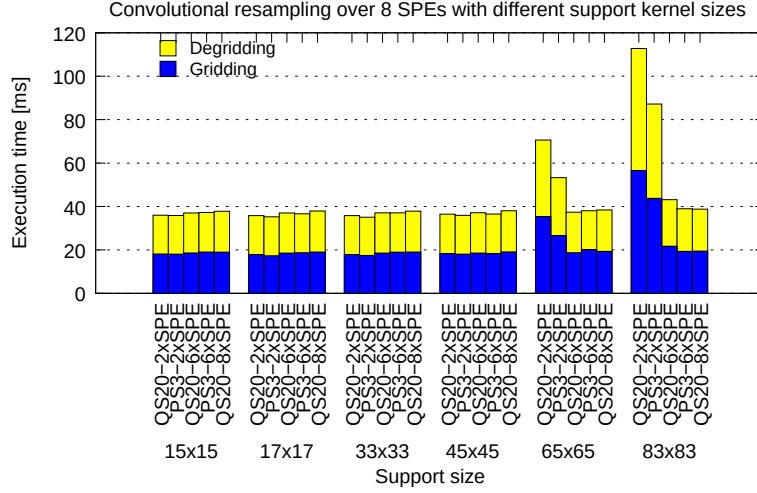


Figure 21: The application performance for different sizes of the support kernel

scaling of the gridding/degridding kernels and we briefly comment on each of them.

One Cell/B.E. per baseline. The simplest solution for the application scaling is to consider solving only one baseline per Cell/B.E. The solution is easy to build, and the only slight decrease from the linear speed-up is due to the final summation of each Cell’s grids. This distributed summation can be done in parallel with any classical algorithm of complexity $O(\log_2 X)$, where $X=B$ is the number of used Cell/B.E. Of course, such a solution will underutilize each of the Cell nodes.

Combining baselines. To implement this solution, we replicate the master-workers approach for each participating Cell/B.E. and we “connect” them all by using a higher-level scheduler to distribute the work to the right Cell/B.E. processor. Again, the scheduler should take decisions that will increase temporal data locality at the Cell/B.E. level and, consequently, at the SPEs level.

Streaming and sorting. The timings measured in our experiments are valid only when the application has a critical mass of data to force the SPEs to be highly utilized. If this is not the case, the best solution is to (partially) buffer the streaming data and pre-sort it such that groups of baselines that have more chances to reuse the same **C** and **G** subregions are allocated to the same Cell/B.E. Such a preparation of data may increase the pre-processing time, but it will definitely decrease the $T_{sample}^{multi-Cell}$ due to proper clustering; in most cases when sorting is feasible (no more than 1000 baselines, buffered for no more than one hour), this solution increases application performance.

To conclude, the application can scale up with the numbers of baselines and the number of allocated Cell/B.E. processors, and the modular approach we have taken simplifies the “upgrade”. Our preliminary empirical results show that up to 4 baselines can be processed by a single Cell/B.E.

without significant performance penalty versus the original, single-baseline model. Further results from our on-going evaluation of the multi-Cell alternatives (which will be available by the time the review is finished) will indicate clearly which one of the three solutions is suitable for various problem sizes.

6 Conclusions and Future Work

HPC applications are hard to port efficiently on multi-core processors, due to the multiple levels of parallelism that need to be properly addressed. In this report, we have presented such an application, i.e., a radioastronomy kernel, and we have shown our experience in trying to efficiently parallelize it on the Cell/B.E. processor. Our approach is based on a locality-enhancing parallelization, which isolates and assigns the lower-level compute intensive blocks to the memory-poor worker cores, and dedicates master cores to execute the irregular control- and data-flow. Due to its high-level view of the problem, our approach can be easily extended to parallelize dynamic data-intensive applications on heterogeneous multi-cores. We have applied this strategy to the convolutional resampling kernel, one of the very first successful attempts to port radioastronomy kernels on a heterogeneous multi-core processor. The experimental results included in the paper show a 20 times performance improvement of the Cell/B.E. solution over the original sequential code running on a commodity machine. Once the application scales-up, even if we are able to process four baselines per processor, we still need several hundreds of Cell/B.E. to process all data streams on-line.

In the near future, we aim to provide a scalability analysis and validate it by a thorough experimental study. We plan to use a performance prediction tool to estimate the expected overall performance and compute the hardware requirements to run our solution in a large radiotelescope environment; further, we will implement and optimize our scale-up scenarios for on-line processing with streaming data and test the performance in a multi-Cell environment.

References

- [1] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the Cell processor. volume IEEE Symposium on Interactive Ray Tracing 2006, pages 15–23, September 2006.
- [2] T.J. Cornwell, K. Golap, and S. Bhatnagar. *EVLA memo 67*. National Radio Astronomy Observatory, <http://www.nrao.edu>, 2003.
- [3] T.J. Cornwell, K. Golap, and S. Bhatnagar. W projection: A new algorithm for wide field imaging with radio synthesis arrays. In *Astronomical Data Analysis Software and Systems XIV ASP Conference Series*, volume 347, page 86, 2004.
- [4] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.
- [5] A.R. Thompson, J.M. Moran, and G.W. Swenson. *Interferometry and synthesis in radio astronomy*. Wiley, New York, 2001.
- [6] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the Cell processor for scientific computing. In *ACM Computing Frontiers 06*, pages 9–20. ACM Press, 2006.