Scientific Programming 17 (2009) 113–134
DOI 10.3233/SPR-2009-0276
IOS Press

Building high-resolution sky images using the Cell/B.E.

Ana Lucia Varbanescu [a,*], Alexander S. van Amesfoort [a], Tim Cornwell [b], Ger van Diepen [d], Rob van Nieuwpoort [d], Bruce G. Elmegreen [c] and Henk Sips [a]

[a] *Department of Computer Science, Delft University of Technology, Delft, The Netherlands*
[b] *National Telescope Facility, Australia*
[c] *IBM T.J. Watson Research Center, NY, USA*
[d] *ASTRON, The Netherlands*

Abstract. The performance potential of the Cell/B.E., as well as its availability, have attracted a lot of attention from various high-performance computing (HPC) fields. While computation intensive kernels proved to be exceptionally well suited for running on the Cell, irregular data-intensive applications are usually considered as poor matches. In this paper, we present our complete solution for enabling such a data-intensive application to run efficiently on the Cell/B.E. processor. Specifically, we target radioastronomy data gridding and degridding, two resembling imaging filters based on convolutional resampling. Our solution is based on building a high-level application model, used to evaluate parallelization alternatives. Next, we choose the one with the best performance potential, and we gradually exploit this potential by applying platform-specific and application-specific optimizations. After several iterations, our target application shows a speed-up factor between 10 and 20 on a dual-Cell blade when compared with the original application running on a commodity machine. Given these results, and based on our empirical observations, we are able to pinpoint a set of ten guidelines for parallelizing similar applications on the Cell/B.E. Finally, we conclude the Cell/B.E. can provide high performance for data-intensive applications at the price of increased programming efforts and with a significant aid from aggressive application-specific optimizations.

Keywords: Multi-core processors, radioastronomy, data-intensive memory-bound applications, Cell/B.E.

1. Introduction

A large part of current radioastronomy research focuses on building larger radio telescopes with better resolutions. Projects like LOFAR [26], ASKAP [6] or SKA [22] aim to provide highly accurate astronomical measurements by collecting huge streams of radio synthesis data, which are further processed in several stages and transformed into high-resolution sky images. When designing and deploying this data processing chain, radio astronomers have quickly reached the point where computational power and its efficient use is critical. For example, it is estimated that LOFAR can produce over 100 TB/day [26]; for SKA, which has about 1500 times more antennas, the number will increase with at least 5 orders of magnitude. For such huge collections of radioastronomy data, whatever cannot be processed in time has to be stored, and whatever cannot be stored is lost. Thus, a small increase in processing performance may translate into a significant advantage in terms of storage space and cost.

This is a typical high performance computing (HPC) problem, typically solved by a supercomputer or a dedicated cluster. However, in the past few years, more typical HPC applications have been successfully implemented on architectures based on multi-core processors – e.g. Sweep3D or RaX/ML on the Cell/B.E. [3,18], or molecular dynamics applications on GPUs [13]. Based on their promising results, more effort is put nowadays into similar experiments for other application fields.

The major pitfall of this trend is that (legacy) sequential code for HPC applications, even if directly usable on the multi-core processors, is not exploiting any of the performance enhancing features of these platforms. Efficient application implementation for these architectures requires multiple layers of parallelism and iterative, machine-specific optimizations in order to come close to peak performance [31]. Even

*Corresponding author. E-mail: A.L.Varbanescu@tudelft.nl.

so, this peak performance, may not be accessible to all applications. While, computation-intensive kernels can be easily tuned to expose more parallelism, data-intensive applications usually require significant algorithmic changes to avoid core underutilization and, as a result, large gaps between observed and peak performance. Because these changes are mostly expertise-based, as well as application and architecture specific, any successful case-study becomes a potential set of guidelines.

In this paper we present such a case-study: the implementation and optimization of the most time-consuming radioastronomy imaging kernels (i.e., data gridding and degridding) on the Cell/B.E. processor. Both kernels are based on convolutional resampling, a basic processing block that dominates the workload of image synthesis. Optimizing this kernel has a direct impact on the performance and hardware requirements of any of modern large radiotelescope [6]. Based on a brief analysis, we model the parallel application using the master-workers paradigm. Although the implementation and subsequent optimizations posed some challenges, especially related to proper load balancing and data distribution, the obtained performance proves this paradigm to be a good alternative for tackling data-intensive applications on the Cell/B.E. processor.

Thus, our contributions through this paper are threefold: (1) we present a model-driven approach on how a memory-bound data-intensive application can be parallelized on the Cell/B.E., (2) we examine the efficiency, performance, and scalability of the new application, as well as its overall suitability for a real-life radiotelescope, and (3) we propose a short list of guidelines for identifying and approaching similar problems. Compared with our previous work, presented in [29], our parallelization is based on a model-driven, systematic technique, we include a thorough performance analysis of each optimization we have performed, and we are able to propose a clear, 10-points guidelines list to approach similar applications.

The remainder of this paper is organized as follows. Section 2 presents a radioastronomy primer, to familiarize the reader with the problem space and terminology. Application analysis and modeling are presented in Section 3. We discuss the parallelization, implementation, and optimizations of the gridding/degridding kernels on the Cell/B.E. in Section 4. Our experiments and their results are discussed in Section 5. We briefly survey significant related work in Section 6 and we conclude our findings with a list of empirical guidelines for similar applications in Section 7.

2. Radioastronomy imaging

One of the radioastronomy goals is to obtain accurate images of the sky. In this section, we introduce the fundamental concepts and terminology required for the reader to understand the background of our application as well as the execution context of our target application.

2.1. Radio interferometry

Radio interferometers are a solution for obtaining high resolutions for the sky images that would otherwise require reflectors of impractically large sizes. Being built as arrays of connected radiotelescopes (with various antennas types and placement geometries), and using the aperture synthesis technique [25], they are able to work as a single large "combine" telescope. However, this solution comes at the price of additional computation, as radio interferometers do not measure the brightness of the sky directly; instead, each pair of antennas measures a (sort of Fourier) component of it. Combining these components into a single image requires significant post-processing.

Each pair of antennas defines *a baseline*. Increasing the number of different baselines in the array (i.e., varying the antenna numbers and/or placement) increases the quality of the generated sky image. The total number of baselines, B, in an array of A antennas is $B = A(A-1)/2$, and it is a significant performance parameter of the radiotelescope. For example, the LOFAR radiotelescope [26] has a total of $B = 1830$ baselines, while SKA [22] should have approximately 1500 times as many.

The simplified path of the signal from each baseline to a sky image is presented in Fig. 1. The signals coming from any two different antennas have to be correlated before they are combined. A correlator reads these (sampled) signals and generates a corresponding set of *complex visibilities*, $V_{b,f,t}$, one for each baseline b, frequency channel f and moment in time t. In other words, taking measurements for one baseline with a sampling rate of one sample/s for 8 straight hours will generate a set of $8 \cdot 3600$ visibilities for each frequency channel (tens to hundreds of them) and each polarization (typically, 2 or 4 in total). For example, using 256 frequency channels (a small number) for LOFAR and its 1830 baselines may lead to about 13.5 GB of data.

Finally, imaging is the process that transforms these complex visibilities into accurate sky images. In prac-

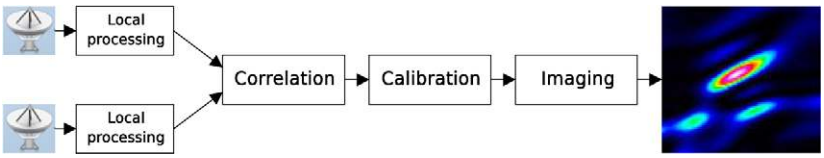


Fig. 1. The software pipeline from antennas to sky images.

tice, building a sky image translates into reconstructing the sky brightness, I (at a given frequency), as a function of some angular coordinates, (l, m), on the sky. The relationship between the sky brightness, $I(l, m)$ and the visibility $V(u, v, w)$ is given by Eq. (1). Note that the values (u, v, w) are the components of the baseline vector between the two interferometer elements, and are expressed in units of wavelength of the radiation [20]. For the case when $w = 0$ (e.g., for a *narrow field of view*, i.e., all baselines are in the same (u, v) plane), we obtain the two-dimensional form of the visibility–brightness relationship, as given in Eq. (2); this equation shows that $V(u, v)$ and $I(l, m)$ are a Fourier pair. Thus, we can compute $I(l, m)$ as the Fourier inverse transform of $V(u, v)$, as shown in Eq. (3); for the image reconstruction, we use the discrete transform (shown in Eq. (4)) on the observed (sampled) visibilities. For the more general case of non-coplanar baselines (i.e., $w \neq 0$, also known as *wide field of view*), there are two main approaches: (1) use the *faceted approach*, where we approximate the wide field as a sum of a lot of narrow fields for which the simplification $w = 0$ holds, compute the FFT for each one of these *facets*, and combine the results [7], or (2) use the W-projection algorithm [5], which computes the FFT for projections of the baseline vector and the observed source vector on a (fairly small) number of planes (P_w) parallel to the (u, v) plane and sums the results. For similar accuracy, P_w is much smaller than the potential number of facets; therefore, the W-projection method is considered more computationally efficient, and it is the one we use in this paper.

$$V(u,v,w) = \int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} \times \mathrm{e}^{-2\pi\mathrm{i}(u\cdot l+v\cdot m+w\cdot\sqrt{1-l^2-m^2})}\,\mathrm{d}l\,\mathrm{d}m, \quad (1)$$

$$V(u,v) = \int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} \times \mathrm{e}^{-2\pi\mathrm{i}(u\cdot l+v\cdot m)}\,\mathrm{d}l\,\mathrm{d}m, \quad (2)$$

$$I(l,m) = \sqrt{1-l^2-m^2} \times \int V(u,v)\mathrm{e}^{+2\pi\mathrm{i}(u\cdot l+v\cdot m)}\,\mathrm{d}u\,\mathrm{d}v, \quad (3)$$

$$I_d(l,m) = \frac{1}{N}\sum_{t=1}^{N} V_d(u_t,v_t) \times \mathrm{e}^{+2\pi\mathrm{i}(u_t\cdot l+v_t\cdot m)}. \quad (4)$$

2.2. Building the images

The computational process of building a sky image has two phases: imaging and deconvolution. The imaging phase generates a *dirty image* directly from the measured visibilities, using FFT. The deconvolution "cleans" the dirty image into a *sky model*. A snapshot of this iterative process is presented in Fig. 2. Note that the gridding and degridding operations are the main targets for our parallelization, due to their time-expensive execution (it is estimated that 50% of the time spent in the deconvolution loop is spent on gridding and degridding).

Note that the measured visibilities are sampled along the uv-tracks (i.e., the trajectory that a baseline defined by (u, v) generates due to the Earth's rotation). Before any FFT operations, data has to be placed in a regularly spaced grid. The operation used to interpolate the original visibility data to a regular grid is called *gridding*, and it is performed by using a support kernel that minimizes the aliasing effect in the image plane [24].[1] *Degridding* is the "reverse" operation used to extract information from the grid and project it "back" to the uv-tracks. Degridding is required when new visibility data is used to refine the current model. A simplified diagram of the gridding/degridding is presented in Fig. 3. For the remainder of this paper, we focus on gridding and degridding, as they represent the most time-consuming parts of the sky image building software chain.

[1] Explaining how these coefficients are computed is far beyond the purpose of this paper. Therefore, we redirect the interested readers to the work of Schwab [24].

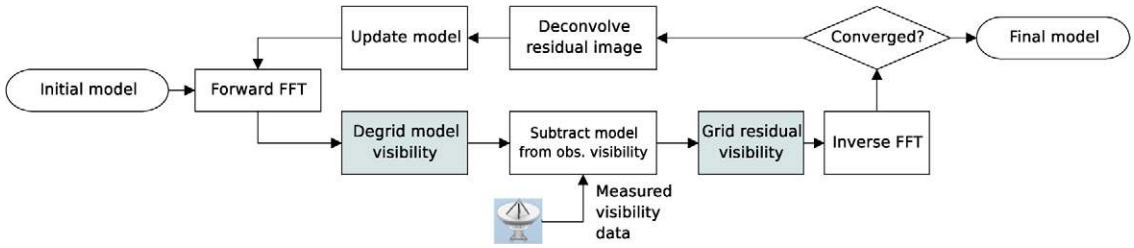


Fig. 2. A diagram of the typical deconvolution process in which a model is iteratively refined by multiple passes. The shaded blocks (gridding and degridding) are both performed by convolutional resampling.

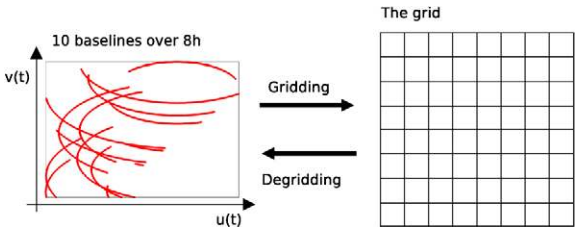


Fig. 3. A simplified diagram of the gridding/degridding operations. The data sampled along the (u, v) tracks in the left-hand side is gridded on the regular grid. The data in the grid is degridded along the same (u, v) tracks.

3. Gridding and degridding

In this section we analyze the characteristics of gridding and degridding, focusing on their computation and data access patterns. Further, we build simple models for each of the two applications, models that shall be further used as starting points for developing the Cell/B.E. parallel application.

3.1. Application analysis

For one high-resolution sky image, data is collected from B baselines, each one characterized by $(u, v, w)_{b,t}$, i.e., its spatial coordinates at moment t. During one measurement session, every baseline b collects N_{samples} for each one of the observed N_{freq} frequency channels. These samples, $V[(u, v, w)_{b,t}, f]$, are called "visibilities".

3.1.1. Data and computation

Gridding is the transformation that interpolates the irregular visibilities $V[(u, v, w)_{b,t}, f]$ on a $2^g \times 2^g$ regular grid, **G**. Each visibility sample is projected over a subregion *SG* of $m \times m$ points from this grid. To compute the *SG*′ projection, the visibility value is weighted by a set of $m \times m$ coefficients, called a support kernel *SK*, which is extracted from a matrix that stores the convolution function, **C**. Note that this convolution function is *oversampled*, i.e., its resolution is finer than that of the original sampling by a factor of $os \times os$. Equation (5) synthesizes this computation. The *SG*′ projection is finally added to the *SG* region of the grid, updating the image data. All data from all baselines is gridded on the same image grid, **G**. The size of the projection, $m \times m$, is a parameter calculated based on the radiointerferometer parameters, and it is anywhere between 15×15 and 129×129 elements. Degridding is the operation that reconstructs the $V'(u, v, w)_{b,t}$ samples from the image grid by convolving the corresponding support kernel, *SK*, and grid subregion, *SG*. Equation (6) synthesizes this computation.

Figure 4 illustrates the way both gridding and degridding work, while Listing 1.1 presents a simplified pseudo-code implementation:

$$\begin{aligned}
&\forall 0 \leqslant x \leqslant (m \cdot m),\\
&\quad SG'(g_{\text{offset}}(u_k, v_k) + x)\\
&\qquad = \mathbf{C}(c_{\text{offset}}(u_k, v_k) + x) \cdot V(u_k, v_k), \qquad (5)\\
&V'(u_k, v_k)\\
&\quad = \sum_{x=1}^{m \cdot m} \mathbf{C}(c_{\text{offset}}(u_k, v_k) + x)\\
&\qquad\qquad \times G(g_{\text{offset}}(u_k, v_k) + x). \qquad (6)
\end{aligned}$$

The `c_index` stores the offset inside **C** from where the current convolution kernel, $SK_M(b, t, f)$ is extracted; the `g_index` stores the offset of the corresponding grid subregion, $SG_M(b, t, f)$, in **G**. These two indices are computed for each time sample, baseline and frequency channel, thus corresponding to each visibility $V[(u, v, w)_{b,t}, f]$.

The essential application data structures are summarized in Table 1. To give an idea of the application memory footprint, consider the case of $B = 1000$ baselines, $SK_M = 45 \times 45$ elements (equivalent of a 2000 m maximum baseline), $os = 8$ (an usual

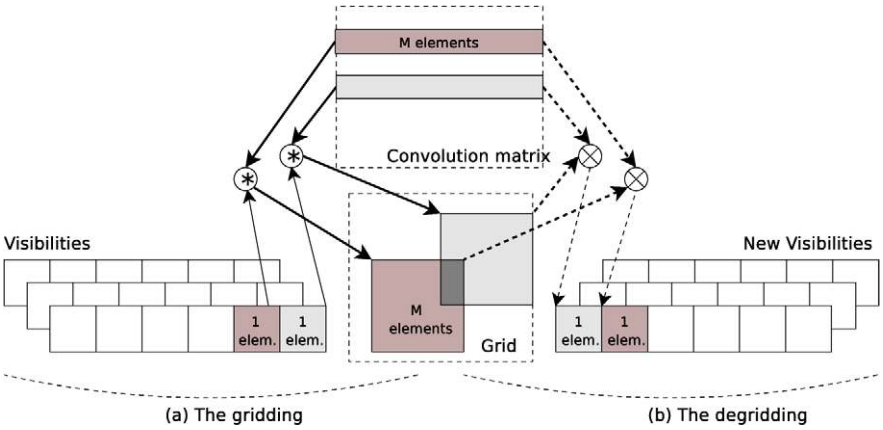


Fig. 4. The two kernels and their data access pattern: (a) gridding (continuous lines) and (b) degridding (dotted lines). Similarly shaded regions are for a single visibility sample. For gridding, newly computed data that overlaps existing grid values is added on. For degridding, newly computed visibilities cannot overlap.

Listing 1.1
The computation for the gridding/degridding

```
1 forall (b=1..N_baselines,t=1..N_samples,f=1..N_freq) // for all frequency channels
2     compute c_index=C_Offset((u,v,w)[b,t],freq[f]); // the SK offset in C
3     SC(b,t,f) = extract_support_kernel(C, c_index, m*m); // get the mxm coefficients
4     compute g_index=G_Offset((u,v,w)[b,t],freq[f]); //the SG offset in G
5     SG(b,t,f) = extract_subregion_grid(G,g_index,m*m); // get the mxm grid data
6     for (x=0; x<(m*m); x++)   //sweep the convolution kernel
7         if (gridding)    {SG'[x]=C[x]*V[(u,v,w)[b,t],f];SG[x]+=SG'[x];}
8         if (degridding) {V'[b,t,f]+=SG[x]*C[x];}
```

Table 1
The main data structures of the convolutional resampling and their characteristics. Note that B is the number of baselines

Name	Symbol	Cardinality	Type	Access pattern
Coordinates/baseline	(u, v, w)	$B \times N_{\text{samples}}$	`Real`	Sequential
Visibility data	V	$B \times N_{\text{samples}} \times N_{\text{freq}}$	`Complex`	Sequential
Convolution matrix	**C**	$m^2 \times os^2 \times P_w$	`Complex`	Irregular
Grid	**G**	512×512	`Complex`	Irregular
Support kernel	SK_M	$M = m \times m$	`Complex`	Sequential
Subregion grid	SG_M	$M = m \times m$	`Complex`	Sequential

oversampling rate) and $P_w = 33$ (the number of planes in the W-projection algorithm). Then, **C** has over 4M elements of data, taking about 34 MB. Further, for a sampling rate of 1 sample/s, an 8 hour measurement session generates 28800 samples on each frequency channel for each baseline; for only 16 frequency channels and 1000 baselines, we have over 450M visibilities, which take about 3.5 GB of memory/storage.

3.1.2. Data locality

Low arithmetic intensity and irregular memory accesses are non-desirable features for parallel applications. They are even worse for multi-core processors, where their negative impact on application performance is increased [4].

As both gridding and degridding are memory intensive applications (the arithmetic intensity is slightly less than $1/3$), locality plays a significant role in

their performance. To analyze data locality and its reuse potential, we trace the memory accesses in both **C** and **G**. The access patterns are generated by the `c_index` and `g_index` offsets. Based on their computation as non-linear functions of the $(u, v, w)_{b,t}$ and f – see Listing 1.1, lines 2 and 4, we can conclude the following: consecutive time samples on the same baseline may not generate consecutive **C** and **G** offsets: `c_index(b,t,f)` $\neq$ `c_index(b,t+1,f)` and `g_index(b,t,f)` $\neq$ `g_index(b,t+1,f)`, which means that the corresponding support kernels and/or grid subregions may not be adjacent. The same holds for two measurements taken at the same time sample t on different consecutive channels. As a result, we expect a scattered sequence of accesses in both **C** and **G**.

We have plotted these access patterns for three different data sets, and the results can be seen in Fig. 5. The left-hand side graphs show the accesses in the execution order (i.e., ordered by their time sample). The right-hand side graphs show the results for the same data, only sorted in increasing order of the indices, therefore indicating the *ideal* data locality that the application can exploit. For these "sorted" graphs, note that the flatter the trend is, the better locality is. For random data, i.e., randomly generated (u, v, w) coordinates, the memory accesses are uniformly distributed in both **C** and **G**, as seen in Fig. 5(a, c). Note that the more scattered the points are, the worse the data locality is. Next, we have used a set of (u, v, w) coordinates from a real measurement taken from a single baseline. Note, from Fig. 5(e, g), the improved locality, which is due to the non-random relationship between $(u, v, w)_{b,t}$ and $(u, v, w)_{b,t+1}$.[2] Also, accesses from measurements taken from ten baselines are plotted in Fig. 5(i, k). The potential for data-reuse is similar for **C**, and somewhat increased in **G**. Finally, Fig. 5(b, d, f, h, l) (the sorted graphs) indicate that *spatial* data locality does exist in the application, but exploiting it requires significant data pre-processing for *all* (u, v, w) samples.

3.2. Application modeling

The task graphs for both kernels are presented in Fig. 6. Note the similar structure: a main loop of $N = N_{\text{baselines}} \times N_{\text{samples}} \times N_{\text{freq}}$ iterations, whose core is a sequence of inner tasks. So, we can model the two kernels as follows (we use a simplified version of the PAMELA notation [27]):

```
Gridding = G1;for(i=1..N){G2;G3;G4;G5}
Degridding = D1;for(i=1..N){D2;D3;D4;D5}
```

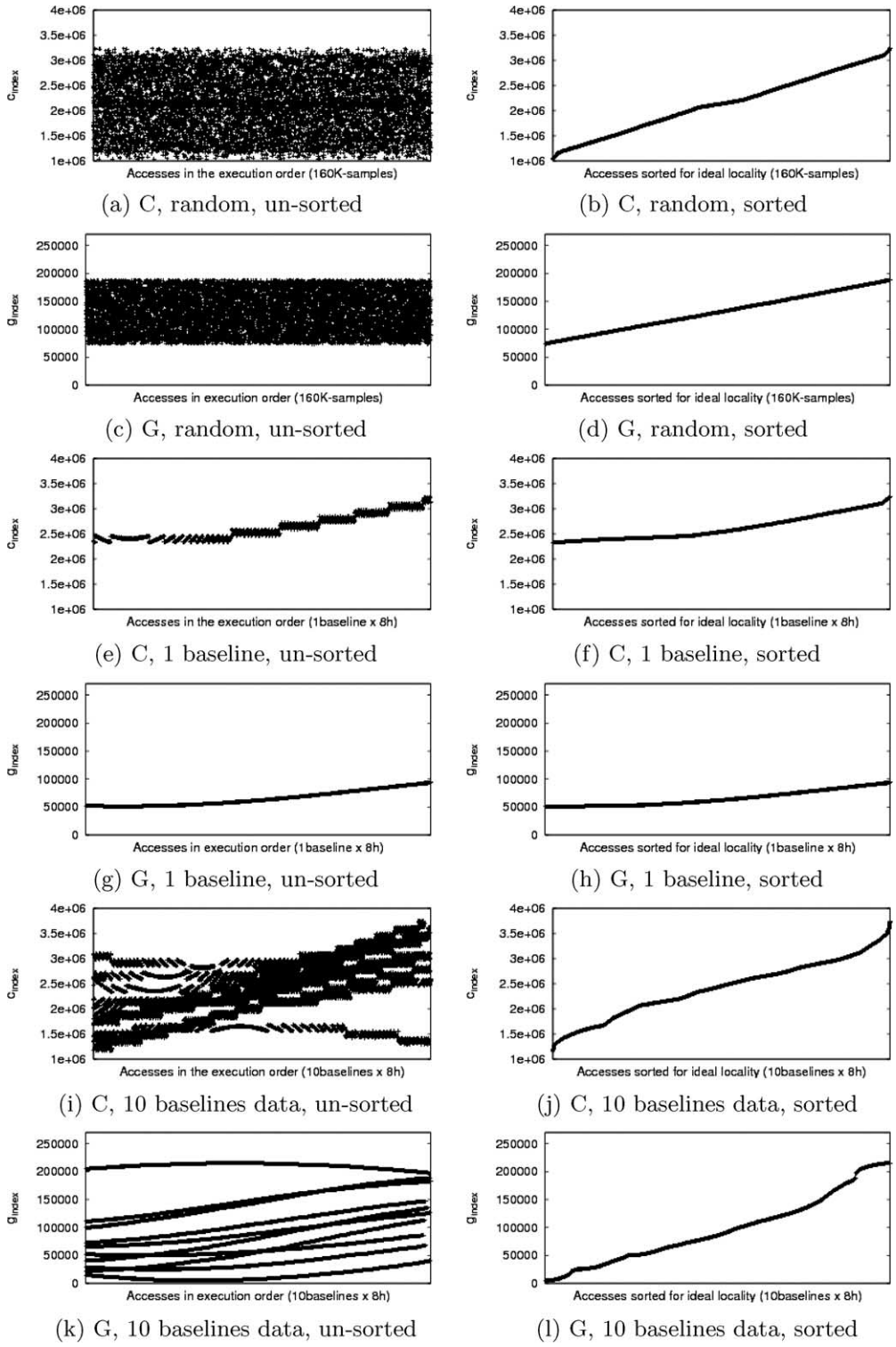
We characterize each of these tasks by a set of parameters that will be used to guide our parallelization decisions. For each task, we focus on the following properties:

- Input data: the type and number of the elements required for one execution.
- Output data: the type and number of the elements produced by one execution.
- Computation : communication: a qualitative estimation (i.e., from very low to very high) of the ratio between the data communication needs of the kernel (e.g., the time it takes to get the data in/out) and the time spent computing the data. Note that data size can have a significant impact on this parameter.
- I : O ratio: the ratio between the size of the input and output data.
- Memory footprint represents the size of the data required for a single task execution; as all the significant data structures have either `Real` or `Complex` values, we can measure the memory footprint in number of `Real` elements.
- Parallelism represents the parallelization potential of a task, which quantifies how easy a task can be parallelized. For example, an embarrassingly parallel task is highly parallelizable, while a file read is hardly parallelizable.
- Impact is an estimation of the time taken by a single task execution (i.e., within one iteration of the main loop). Note that this estimation may be sensitive to application-specific parameters. Also note that some tasks are outside the loop and, as such, will not impact the main core computation.

Based on these parameters, the task characterizations for both gridding and degridding are summarized in Table 2.

A quick look at Table 2 proves that both gridding and degridding are memory bound (see the I : O ratio, memory footprint), data-intensive (see the computation : communication ratio) kernels. As such applications are not a good match for Cell/B.E.-like machines, where the core-level memory is not shared, a lot of effort will have to be put into investigat-

[2] The $(u, v, w)_{b,t+1}$ can be computed from the $(u, v, w)_{b,t}$ coordinates by taking into account the Earth rotation in the interval between the two samples.



(a) C, random, un-sorted (b) C, random, sorted

(c) G, random, un-sorted (d) G, random, sorted

(e) C, 1 baseline, un-sorted (f) C, 1 baseline, sorted

(g) G, 1 baseline, un-sorted (h) G, 1 baseline, sorted

(i) C, 10 baselines data, un-sorted (j) C, 10 baselines data, sorted

(k) G, 10 baselines data, un-sorted (l) G, 10 baselines data, sorted

Fig. 5. Access patterns in **C** and **G**, with various data and data-sizes, original and sorted. For the original data, darker regions (i.e., more points) signal good temporal and spatial data locality. For the sorted data, the flatter the curve, the better the spatial locality over the entire data set is.

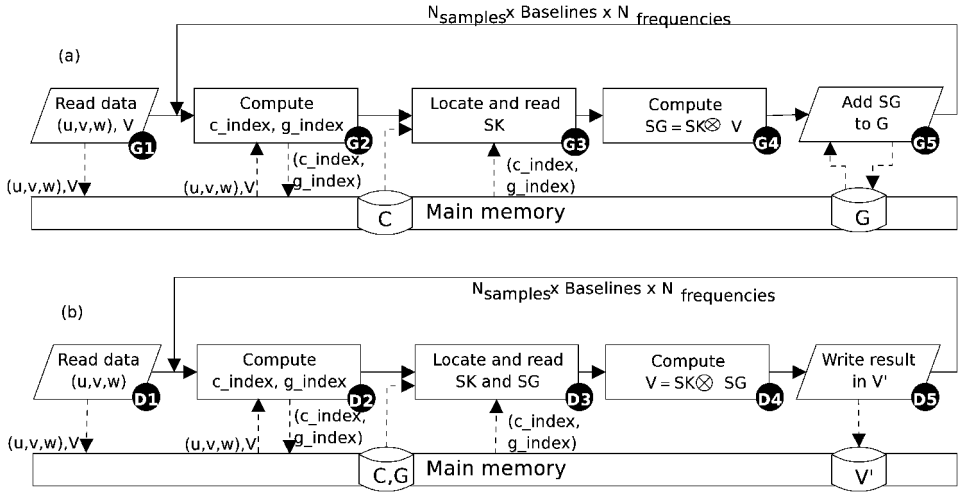


Fig. 6. The execution phases of both kernels: (a) gridding, (b) degridding.

Table 2
The characteristics for the kernels' tasks

Task	Input data	Comp : Comm	Output data	I : O ratio	Memory (Real el.)	Parallelism	Impact (%)
G1	$(u,v,w), V$	$--$	$(u,v,w), V$	1 : 1	$2 \times N$	$--/-$	$-$
G2	$(u,v,w), f$	$+$	c_index g_index	1 : 1	c_1	$\sim$	5
G3	c_index, **C** g_index, **G**	$--$	SK, SG	$1 : m^2$	$c_2 \times m^2 + c_3$	$+$	20
G4	SK, SG	$+$	SG'	$2m^2 : m^2$	$c_4 \times m^2 + c_5$	$++$	65
G5	SG'	$--$	G	$m^2 : 1$	$c_6 \times m^2 + c_7$	$--$	10
D1	(u,v,w), **G**	$--$	(u,v,w), **G**	1 : 1	$N + c_8$	$--/-$	$-$
D2	$(u,v,w), f$	$+$	c_index g_index	1 : 1	c_9	$\sim$	5
D3	c_index g_index	$--$	SK, SG	$1 : m^2$	$c_{10} \times m^2 + c_3$	$+$	20
D4	SK, SG	$+$	V	$m^2 : 1$	$c_{11} \times m^2 + c_{12}$	$++$	75
D5	V	$--$	V'	1 : 1	$c_{13} \times m^2 + c_{14}$	$--$	$-$

Notes: $N = N_{\text{baselines}} \times N_{\text{samples}} \times N_{\text{freq}}$; the "Comp : Comm" ratio and the "Parallelism" are qualified from very low to very high, with $--$, $-$, $\sim$, $+$, $++$; c_i are constant whose value are irrelevant; a task with no impact on the main computation core is denoted with $-$.

ing two major parallelization aspects: task distribution (i.e., change the sequential patterns of the original application model into parallel ones) and data management (i.e., use data parallelism and pipelining among the tasks to avoid excessive memory traffic).

4. Parallelization on the Cell/B.E.

In this section, we describe our model-guided parallelization for the gridding/degridding kernels. Further, we present one implementation solution and we describe the additional cell-specific and application-specific optimizations that had a significant impact on the kernels performance. As our approach is similar for both kernels, we only use the gridding parallelization for the detailed explanations. Finally, we present the outline of our method as a list of guidelines for enabling other similar applications to run on the Cell/B.E.

4.1. The Cell/B.E.

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 Cell/B.E.'s peak performance of 204 single precision GFlops [11], it was quickly considered a good target platform for HPC applications. 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 PPE contains the Power Processing Unit (PPU), a 64-bit PowerPC core with a VMX/AltiVec unit, separated L1 caches (32 kB for data and 32 kB for instructions) and 512 kB 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 256 kB 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. All SPU instructions are 128-bit SIMD instructions, and all 128 SPU registers are 128-bit wide.

The Cell/B.E. cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [18,31] to image processing applications [2]. 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.

The impressive performance of the Cell/B.E. comes mainly from exploiting its various parallelism layers, from task and data parallelism across multiple SPEs to SPE code SIMDization and multi-buffering for DMA transfers [9]. While applications written from scratch can natively exploit all these features, legacy applications have to systematically enable them, by iteratively modifying the code. A rule of thumb for the parallelization effort is: the lower the parallelization level is, the more modifications to the original code it requires. For example, application partitioning in tasks requires little intrusion in the task code, while code SIMDization may require changes at almost every instruction.

4.2. A model-driven application parallelization

To have efficient implementations of gridding and degridding on the Cell/B.E., we aim to delegate and optimize most of their computationally intensive parts to the SPEs, thus increasing the overall application performance. To decide what parts of the whole gridding computation chain can be efficiently moved to the SPEs, we start from a simple gridding model, and we investigate the performance effects each SPE-PPE configuration might have. The model of the sequential gridding chain, depicted in Fig. 6 is:

```
Gridding=G1;for(i=1..N){G2;G3;G4;G5;}
```

4.2.1. The main loop

To enable several parallelization options, we need to transform the sequential loop into a parallel one; in turn, this transformation allows the computation inner core, `{G2;G3;G4;G5}`, to be executed in an SPMD (single process, multiple data) pattern over `P` processors. Such a transformation can only be performed if the iteration order is not essential and if the data dependencies between sequential iterations can be removed. To remove the data dependency between G3 and G5 (via **G**, as seen in Table 2), we assign each processor its own copy the grid matrix, **G**.local. Further, because the grid is computed by addition, which is both commutative and associative, the iteration order is not important. Thus, after all computation is ready, an additional task (G5.final) is needed to sum all **G**.local's into a final **G** matrix.[3] The new model becomes:

```
Gridding=G1;par(p=1..N)
         {G2;G3;G4;G5};G5.final
```

Note that the same transformation is simpler for the degridding kernel, as there is no data dependency between D2 and D5.

4.2.2. Data read (G1/D1)

The input task, `G1`, has to perform the I/O operations and will run on the PPE. Its output buffer, a collection of $(V, (u, v, w))$ tuples, is located in the main memory. For the off-line execution (input data is stored in files), the parallelization among several PPEs is possible, but will only gain marginal application performance by reducing the start-up time. In the case of online (streaming) execution, parallelization is only useful in the highly unlikely case of a streaming rate (i.e., G1's input) much larger than the memory rate (i.e., G1's output). Given these considerations, we did not parallelize `G1` (and, similarly, `D1`); further, these tasks are not taken into account when measuring application performance.

[3] This simple solution is possible for any operation that is both commutative and associative, like addition in this case.

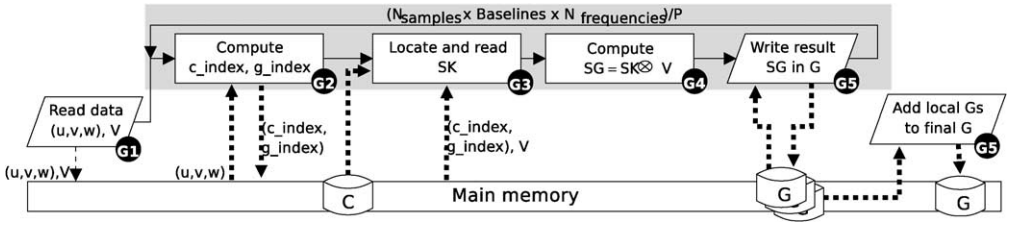


Fig. 7. The execution phases of the parallel gridding kernel. The shaded part is a candidate for execution on each SPE. Bold dotted lines represent off-core memory transfers.

4.2.3. The computation core (G2–G5/D2–D5)

Tasks `G2`, `G3`, `G4`, `G5` represent the main computation core of the gridding kernel. Once the parallel loop is in place, we have to map this sequence of tasks and their data on the available SPEs, enabling several instances to run on different data (SPMD). Figure 7 shows the task graph of application running on the PPE and one SPE (the shaded part). The bold dotted lines represent off-core memory transfers: because the tasks running on the SPE can only process data in their own local memory, they have to use DMA to fetch data from the main memory, compute, and then use DMA again to store the data in the main memory. Compared with Fig. 6, these DMA transfers represent additional overhead.

The new model is:

```
Gridding=G1@PPE;par(p=1..P)
         {for(i=1..N/P){G2@SPE;
         G3@SPE;G4@SPE}};G5@PPE
```

We further decompose the model by exposing the DMA operations (note that `G3@SPE}` is replaced by its DMA operations), and we obtain:

```
Gridding=G1@PPE;
         par(p=1..P){
             for(i=1..N/P){
                 DMA_RD(u,v,w);G2@SPE;
                    DMA_WR(g_index,c_index);
                 DMA_RD(g_index,c_index,
                    SK,V);G4@SPE;DMA_WR(SG);
                 DMA_RD(SG,G.local);G5@SPE;
                    DMA_WR(G.local)}
}; G5.final@PPE
```

4.2.4. The final addition (G5.final/D5.final)

The final computation stage, `G5.final`, computes **G** by summing the SPE copies, **G**.local, using, for example, a binary tree model [10]. However, as this computation is not part of the main core, it represents only a small fixed overhead (depending only on the number of SPEs and the grid size) to the overall application time; thus, it is not parallelized and not included when measuring application performance.

4.3. Mapping on the Cell/B.E.

Further, we investigate the mapping this model on a Cell/B.E. processor for the streaming data scenario (i.e., the most generic case for data input) and several platform-specific optimizations.

4.3.1. Task mapping

To avoid expensive context switching, the mapping of the G2, G3, G4 and G5 tasks on the SPEs can be done in two ways:

- Assign one task per SPE, in a task-parallel model. In this case, data is streaming between SPEs, without necessarily going to the main memory. One iteration takes T^T_{unit}, and $N/(N_{\text{SPEs}}/4)$ iterations can run simultaneously; t_{stream} is the data transfer time between two SPEs. Equation (7) gives a rough estimation of the execution time for this solution.

$$\begin{aligned} T^T_{\text{unit}} &= t_{G2} + t_{\text{stream}} + t_{G3} + t_{\text{stream}} \\ &\quad + t_{G4} t_{\text{stream}} + t_{G5}, \\ T^T_{\text{total}} &= \frac{N}{N_{\text{SPEs}}/4} \times T^T_{\text{unit}} = \frac{4 \times N}{N_{\text{SPEs}}} \times T^T_{\text{unit}}. \end{aligned} \tag{7}$$

  This solution is appealing due to its simplicity, but we have dismissed it as the computation imbalance between G2, G3, G4 and G5 leads to severe core underutilization.
- Assign the entire chain, G2–G3–G4–G5, to one SPE, in a pure SPMD model. The data is shared in temporary buffers in the SPE local store and, when running out of storage space, in the main memory. Note that the number of DMA RD/WR operations is reduced in the new parallel appli-

cation model, due to data sharing via the local store:

```
Gridding=G1@PPE;
          par(p=1..P){
             for(i=1..N/P){
                DMA_RD(u,v,w);G2@SPE;
                   DMA_RD(SK,V);
                G3@SPE;G4@SPE;
                   DMA_RD(G.local);
                G5@SPE;DMA_WR(G.local)}
}; G5.final@PPE
```

For this new solution, one iteration takes T^D_{unit}, and N iterations can run simultaneously; Eq. (8) presents a rough estimation of the execution time:

$$\begin{aligned} T^D_{\text{unit}} &= t_{G2} + X * t_{mm} + t_{G3} \\ &\quad + Y * t_{mm} + t_{G4} \\ &\quad + Z * t_{mm} + t_{G5}, \\ T^D_{\text{total}} &= \frac{N}{N_{\text{SPEs}}} \times T^D_{\text{unit}}. \end{aligned} \tag{8}$$

Note that T^D_{unit} includes a potential overhead for accessing the main memory, via the X, Y and Z coefficients. Even so, this solution has a higher performance potential and enables higher core utilization, so we proceed to its implementation and optimizations.

4.3.2. Data distribution

There are three main data structures used by the SPEs computation: **C**, **G** and the $(V, (u, v, w))$ tuples. A symmetrical data distribution is not possible for all three of them, as consecutive visibility measurements will generate non-adjacent regions in **C** and **G**. In other words, a block of data from the visibilities array will not map on a contiguous block in neither **C** nor **G**. Thus, only one array shall be symmetrically distributed to the SPEs. There are two good candidates for this initial distribution: the grid and the visibilities.

G is the smallest array and the one with the highest re-use rate (i.e., more additions are made for each grid point). However, a symmetrical distribution of grid regions to all SPEs will generate a severe load imbalance between SPEs because the grid is not uniformly covered by real data processing – see also Fig. 5.

Due to its sequential access pattern, the $(V, (u, v, w))$ tuples array is also suitable for symmetrical distribution. For offline processing, a simple *block distribution* can be used to spread the N visibilities on the N_{SPEs} cores. For online processing, some type of *cyclic distribution* is required to keep the load and utilization uniform among the SPEs.

Next, we need to verify that all data structures used by G2–G3–G4–G5 fit in the memory of a single SPE. The required data size for all three tasks, as computed from Table 2, is:

$$\begin{aligned} \text{Size} &= \text{sizeof}(SK) + \text{sizeof}(SG) + \text{sizeof}(SG') \\ &\quad + \text{sizeof}((u, v, w)_{b,t}, g_{\text{index}}, c_{\text{index}}, V_{b,t}) \\ &= 3 \times m^2 \times \text{sizeof(complex)} \\ &\quad + (4 \times \text{sizeof(complex)}) \\ &\quad \sim 3 \times m^2 \times \text{sizeof(complex)}. \end{aligned} \tag{9}$$

Given the local store size of 256 kB (= 32K complex numbers), we can estimate the maximum size of the convolution kernel to be $SK_M = m \times m = 100 \times 100$, which is smaller than the required 129×129 maximum. Thus, the G2–G3–G4–G5 chain cannot run without communicating with the main memory via DMA (i.e., in Eq. (8), $X, Y, Z \neq 0$). To minimize the DMA overhead influence on the T^D_{unit}, we resort to double buffering and pipelining.

Double buffering and pipelining. Because the Cell/B.E. DMA operations can be asynchronous, we overlap computation and communication to increase the application parallelism. The new application model (note that `G3@SPE` has been replaced by its implementation, the two DMA_RD operations for *SK* and *SG*) is:

```
Gridding=G1@PPE;
          par(p=1..P){
             for(i=1..N/P){
               par(DMA_RD(u,v,w),
                   G2@SPE(u,v,w),DMA_RD(V),
                   DMA_RD(SK),
                   G4@SPE(V,SK,SG'.local),
                   DMA_RD(SG.local),
                   G5@SPE(SG'.local,G.local),
                   DMA_WR(SG.local));
}; G5.final@PPE
```

For a simpler implementation, the tasks chain is "broken" into: the indices computation (`G2@SPE`), which only depends on scalar data, like $((u, v, w), V)$ and the grid computation, (`G4@SPE-G5@SPE`), which re-

quires the array data sets. For the G2@SPE, the computation itself is overlapped with the DMA transfer of the visibility data, V. For the rest of the chain, we need to solve two more DMA-related constraints: (1) a single DMA operation is limited to 16 kB of data, and (2) one transfer works only on contiguous data regions. For SK_M, for which the convolution matrix construction guarantees that all $m \times m$ elements are in a contiguous memory region, we need $\text{dma}_{SK_M} = \text{sizeof}(SK_M)/16\text{ kB} = m^2 \times 8$ consecutive transfers. For SG_M, which is a rectangular subregion in a matrix, we need to read/write each line separately, thus implementing $\text{dma}_{SG_M} = m$ consecutive DMA transfers. As the grid is read one line at a time, the computation of the current line can be overlapped with the reading/writing of the next line. This pipeline effect is obtained by double buffering. The detailed model is:

```
Gridding=G1@PPE;
         par(p=1..P){
           for(i=1..N/P){
             DMA_RD(u,v,w);
              G2@SPE(u,v,w) || DMA_RD(V));
             pipeline (k=1..m-1) {
              DMA_RD(SK[k,*]);
              DMA_RD(SK[k+1,*]) ||
                G4@SPE(V,SK[k,*],
                SG'.local[k,*]);
              DMA_RD(SG.local[k,*]);
              par{DMA_RD(SG.local[k+1,*]),
                 G5@SPE(SG'.local[k,*],
                 SG.local),
                 DMA_WR(SG'.local[k-1,*]));}
             }
}; G5.final@PPE
```

We estimate the new T^D_{unit} in Eq. (10):

$$\begin{aligned} T^D_{\text{unit}} &= t_{\text{DMA\_RD}} + \max(t_{\text{DMA\_RD}}, t_{G2}) \\ &\quad + m \times \big(t_{\text{DMA\_RD}} + \max(t_{\text{DMA\_RD}}, t_{G4}) \\ &\quad + t_{\text{DMA\_RD}} \\ &\quad + \max(t_{\text{DMA\_RD}}, t_{G5}, t_{\text{DMA\_WR}})\big). \quad (10) \end{aligned}$$

SIMD-ization. Finally, we need to optimize the computation code itself, by applying core specific optimizations: loop unrolling, SIMD-ization, branch removal, etc. However, due to the small amount of computation, as well as its overlap with communication, these optimizations may not have a significant impact in the overall execution time. Nevertheless, we have implemented the core computation of both G4 and G5 using the Cell/B.E. SIMD intrinsics. Although a theoretical speed-up factor of 4 can be obtained for perfect code SIMD-ization, the observed improvement has been not larger than 40%.

4.4. Application-specific optimizations

Note that all the optimizations presented so far are data independent, addressing only the structure of the application and not its runtime behavior. In this final parallelization step, we discuss a few additional, data-dependent optimizations, which aim to increase data locality and re-use between consecutive data samples. The experimental results presented in Section 5 show how important these data optimizations can be for the overall application performance.

4.4.1. SPE-pull vs. PPE-push

Online data distribution from the PPE to the SPEs can be performed either by the PPE pushing the data (i.e., sending the address of the next elements) or by the SPE pulling the data (i.e., the SPE knows how to compute the address of its next assigned element). Although the SPE-pull model performs better, it is less flexible as it requires the data distribution to be known. The PPE-push model allows the implementation of a dynamic master-worker model, but the PPE can easily become the performance bottleneck. The choice or combination between these two models is highly application dependent; for example, in the case of gridding/degridding, all data-dependent optimizations are based on a PPE-push implementation.

4.4.2. Input data compression

A first observation: if a set of n visibility samples generate the same (c_index,g_index) pair, thus using the same *SK* for updating the same **G** region, the projection can be executed only once for the entire set, using the sum of all n data samples:

$$SG' = \sum_i^n SK \times V[i] = SK \times \sum_i^n V[i].$$

We have only applied this optimization for consecutive samples. We have noticed that this data compression gives excellent results for (1) a single baseline, when the frequency channels are very narrow, or (2) for a large number of baselines, where equal (c_index,g_index) pairs appear by coincidence on different baselines. Further, there is a higher prob-

ability of such sequences to form when the visibility data is block-distributed rather than cyclic-distributed.

However, compressing these additions is limited by an upper bound, due to concerns related to accuracy. This upper limit, i.e., a maximum numbers of additions that preserve accuracy, is computed such that the overall error is not larger than the accepted level of noise. Lower noise levels will decrease the impact of this optimization.

4.4.3. PPE dynamic scheduling

To increase the probability of sequences of visibilities with overlapping *SK* and/or *SG* regions, the PPE can act as a dynamic data scheduler: instead of assigning the visibility samples in the order of their arrival or in consecutive blocks, the PPE can compute the `(c_index,g_index)` pair for each $V(u,v,w)_{b,t}$ and distribute the corresponding data to an SPE that already has more data in that region. To balance the PPE scheduling with the SPE utilization, a hybrid push-pull model is implemented using a system of shared queues. Each queue is filled by the PPE, by clustering visibility data with adjacent *SK*'s and *SG*'s *while* preserving load balancing and consumed by one SPE. As the PPE requires no intermediate feedback from the SPEs, there is virtually no synchronization overhead. The PPE may become a bottleneck only if the visibilities clustering criteria are too restrictive and/or too computationally-intensive; implementing multiple threads on the PPE side may partially alleviate this problem.

4.4.4. Grid lines clustering

Each SPE has in its own queue a list of N_q visibilities that need to be gridded, so it computes a $m \times m$ matrix for each one of them. However, as the computation granularity is lower (i.e., we compute one line at a time, not the entire subregion), and because a DMA operation can fetch one complete grid line, the SPE can compute the influence of several visibilities on the complete grid line. Thus, the SPE iterates over the **G** lines, searches its local queue for visibilities that need to be gridded on the current **G** line, and computes all of them before moving to the next grid line. This approach increases the grid re-use, but it also increases the computation for each visibility by additional branches.

4.4.5. Online and offline sorting

Finally, the best way to increase data re-use is to be able to sort the data samples by their `(g_index)` coordinate. In the offline computation mode, sorting all visibilities may enable a simple and balanced data block distribution among the SPEs. However, as the data volume is very large, such an overall sort may be too expensive. The other option (also valid in the case of online data processing) is for the SPEs to sort their own queues before processing them. Such an operation is very fast (at most a few thousands of elements need to be sorted), and it increases a lot the efficiency of the grid-lines clustering trick.

4.5. Parallelization summary and guidelines

The outline of our parallelization approach is:

1. Identify the main computation core of the application, and recognize a data-intensive behavior by evaluating its computation : communication ratio.
2. Split the application into tasks and draw the sequential task graph. Evaluate the data I/O and memory footprint of each task.
3. Depending on the number of tasks and their load, evaluate the potential task mappings on the available SPEs; tasks on separated SPEs communicate by data streaming, while tasks running on the same SPE share data in the local store and/or main memory. Choose the solutions that offer the highest number of parallel instances.
4. Evaluate data distribution options. For static, symmetrical distributions, use either data that is sequentially accessed or data that is frequently reused. For irregularly accessed data, introduce a PPE-based scheduler to dynamically distribute data among the SPEs.
5. When using a PPE-scheduler, implement a queue-based model for the SPEs, to increase SPE utilization and reduce unnecessary synchronization overhead. Further, implement a multi-threaded version of the PPE computation and reserve at least one thread for the queue filling, such that this operation is only marginally affected by other PPE tasks.
6. Analyze and implement any potential data-dependent optimization on the PPE scheduler; note that such a transformation may require modifications to the SPE code as well.
7. Expand the model to include the data transfers, including the DMA operations. Remove the I/O dependencies between tasks on the same SPE and evaluate the data streaming performance for tasks running on different SPEs.
8. Verify the memory footprint for each task or combination of tasks. If too large, reduce task granularity to fit, and recompose up to the original computation size using a pipeline model.

9. Analyze communication and computation overlapping by either double-buffering of instruction reshuffling.
10. Implement core-specific optimizations and eventual application-specific optimizations.

Using these guidelines, any other data-intensive application can be parallelized on the Cell/B.E. Although the speed-up factor obtained using this approach is highly application dependent, the queue-based scheduling on the PPE side offers a good solution for increased core utilization.

5. Experiments and results

This section presents our experimental study on the performance and scalability of the gridding/degridding implementations on the Cell/B.E. We include a detailed performance analysis which, together with the novel experimental results for processing multiple baselines on a single Cell/B.E. processor complement our previous work [29].

Our performance study focuses on high-level application optimizations. Therefore, although we have implemented the typical SPE-level optimizations (including DMA double buffering, SIMD-ization and loop unrolling), we do not present in detail the improvements they provide over non-optimized code (details can be found in [28]). Rather, we discuss the custom optimizations at the job scheduling level, i.e., at the administration of the PPE–SPE job queues. Further, we analyze the effect of data-dependent optimizations. Finally, we present the performance and scalability analysis of the fully optimized application, and discuss the impact of results on the on-line application scenario.

5.1. The experimental set-up

All the Cell/B.E. results presented in this paper have been obtained on a QS21 blade, provided for remote access by IBM. We have validated the blade results by running the same set of experiments (up to 6 SPEs) on a local PlayStation 3 (PS3). The results are consistent, excepting the cases when the overall PS3 memory is too small for the application footprint, leading to a rapid decrease in performance. For example, gridding with a support kernel larger than 100×100 elements is twice as slow as the sequential application, while several attempts to test the application on more than 500 baselines have crashed the PS3 multiple times.

We compare these results with a reference time, measured for the execution of the original, sequential application on an Intel Core2-Duo 6600 processor, running at 2.4 GHz.[4]

The application is implemented using SDK3.0. We have used three different test data sets, $(V, (u, v, w))_{b,t}$: real data, i.e., a collection of real data gathered by 45 antennas in an 8-hours observation session (990 baselines, with 2880 data samples on 16 frequency channels each); randomly generated data and single-value data, i.e., all $(u, v, w)_{b,t}$ coordinates are equal.

The main performance metric for the gridding and degridding is the execution time: ultimately, the astronomers are interested in finding out how fast the problem can be solved. Therefore, we present our performance results using *time per gridding*, a normalized metric defined as the execution time of any of the two kernels divided by the total number of operations that were performed. The smaller the time per gridding is, the better the performance is. For the purpose of estimating how good the performance we achieve is in absolute terms, we also report the application throughput, which is computed by estimating the useful computation FLOPs for each gridding operation, and dividing by the measured time per gridding.

Finally, as both gridding and degridding involve a similar computation pattern and the measured results follow the same trends, we only plot and discuss the gridding results (for specific results on degridding, we refer the reader to [28]).

5.2. A first Cell/B.E. implementation

The first Cell/B.E. implementation is based on a PPE scheduler that uses separate job queues for each SPE. The PPE fills each queue with pairs of the `(c_index,g_index)` indices – i.e., the coordinates from where the SPE has to read the support kernel, *SK*, and the grid subregion, *SG*. The PPE fills the queues in chunks, with a fixed size of (at most) 1024 pairs per queue.

On the SPE side we have implemented a "standard set" of optimizations. Because the convolution matrix was reorganized such that any usable support kernel is contiguous in memory, *SK* is fetched (after proper padding for alignment and/or size) either using a sin-

[4]The choice for this particular machine was only due to availability, and we use the execution time on Core2-Duo only as a measure of the performance of the reference sequential code, not as a measure of the potential performance of this particular processor.

gle DMA operation or a DMA-list, depending on its size. As the grid subregion *SG* is a rectangular section from the **G** matrix, it requires reading/updating line by line. Therefore, for *SG*, we have implemented multi-buffering: while the current grid line is being processed, the next one is being read and the previous one is being written. As each SPE works on a "proprietary" grid, no inter-SPE synchronization for reads and/or writes is necessary. For each (*SK*, *SG*) pair, the visibility computation, i.e., the loop that computes the $m \times m$ complex multiplications, is SIMD-ized. Because complex numbers are represented as `(real,imaginary)` pairs, the loop is unrolled by a factor of 2, and each iteration computes two new grid points in parallel. Finally, the queue is polled regularly: the SPE performs one DMA read to fetch the current region of the queue it is working on. All new `(c_index,g_index)` pairs are processed before a next poll is performed.

The performance of one visibility computation (the compute intensive part of the gridding) reaches 9 GFlops for the largest kernel, and 2.5 GFlops for the smallest one. Therefore, more aggressive optimizations can still be performed to increase the SPE performance. However, this path requires significant SPE code changes (which are interesting for future work), without solving the overall memory bottleneck of the application. In the remainder of this section, we focus on high-level optimizations and their ability to efficiently tackle the memory-intensive behavior of the application.

We have tested our first Cell/B.E. implementation on real data collected from 1, 10 and 100 baselines, using seven different kernel sizes. For each kernel size, we have measured the time per gridding using the platform at 100%, 50%, 37.5%, 25%, 12.5% and 6.25% of its capacity (16, 8, 6, 4, 2 and 1 SPEs, respectively). Figure 8 presents the performance results for gridding real data collected from 100 baselines, using four representative support kernel *SK* sizes (between 17×17 and 129×129). The results show that the performance increases with the size of the support kernel. This behavior is due to the better computation-to-communication ratio that larger kernels provide: a longer kernel line requires more computation, which hides more of the communication overhead induced by its DMA-in operation. We also point out that for this initial implementation, the performance gain is not significant when using more than 4 SPEs, a clear sign that the platform is underutilized.

When running the gridding for 10 baselines, a small performance loss – less than 5% is observable versus

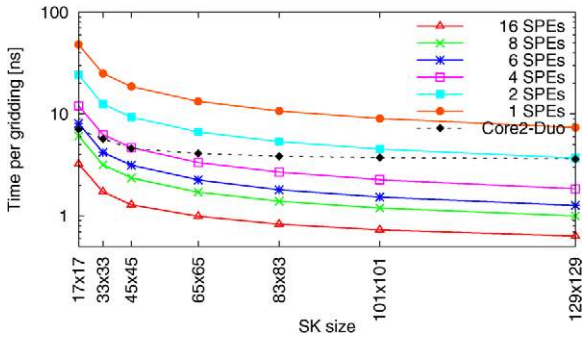


Fig. 8. Time per gridding results for seven different kernel sizes. The best performance is achieved for the largest kernel size, which allows for the best computation-to-communication ratio. Note the logarithmic Y scale.

the 100-baselines data set. Further, the performance obtained for the 10-baselines data set is up to 20% better than the performance obtained for a single baseline. This improvement is due to the larger amount of work, which in turn leads to better utilization and load balancing among SPEs.

Figure 9(a) presents the application throughput for the same reference Cell/B.E. implementation, using the same 100-baselines data set. We notice the throughput increases for larger support kernels, but it still remains below 10% of the theoretical peak performance (25.6 GFlops per SPE). Figure 9(b) shows the application throughput per core, where the original implementation, running on the Core2-Duo, outperforms the Cell/B.E. version. Finally, Fig. 10 shows the application throughput for the largest support kernel (129×129) on all three data sets: 1, 10 and 100 baselines. The results show that the througput increases with the number of baselines.

5.3. Improving the data locality

We tackle data locality problem at two levels: at the PPE level, by controlling the way the queues are filled, and on the SPE level, by detecting overlapping **C** and/or **G** regions to be grouped, thus skipping some DMA operations. Remember that the PPE distributes data to the SPE queues in "chunks", such that each queue receives a sequence of `(c_index,g_index)` pairs (not a single one) before the next queue is filled. To optimize this distribution, the PPE checks on-the-fly if multiple consecutive `(c_index,g_index)` pairs overlap, case in which they are all added into the same queue. When the SPEs read their queues, they can exploit the data locality these sequences of pairs provide. Additionally, we have added a local SPE sorting

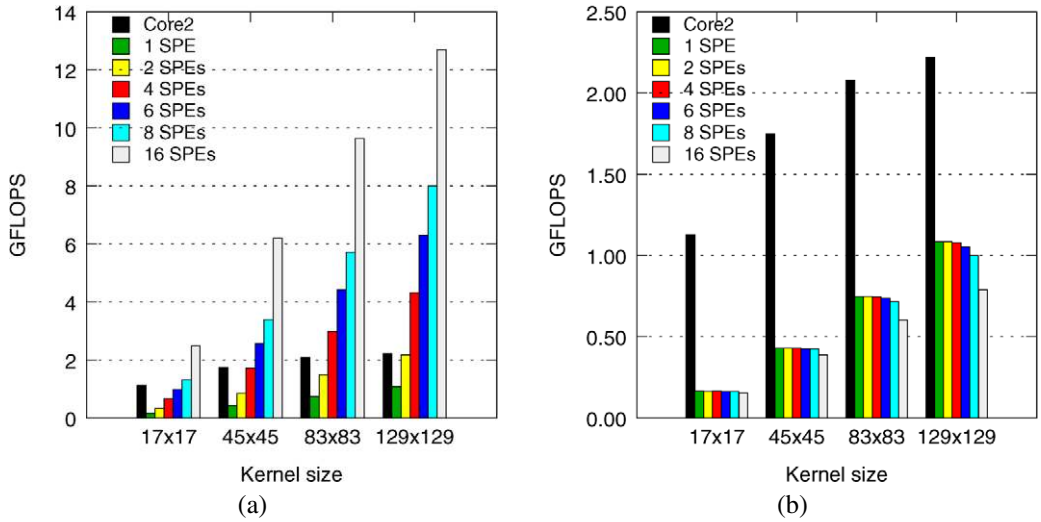


Fig. 9. Application throughput for the first Cell/B.E. implementation (a) per platform and (b) per core for 4 different *SK* sizes. Note that the best throughput is obtained for the largest kernel, 129 × 129.

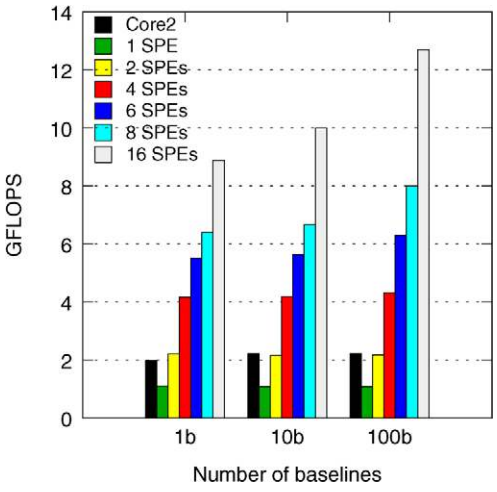


Fig. 10. Application throughput for $SK = 129 \times 129$ for the first Cell/B.E. implementation running on data from 1, 10 and 100 baselines.

algorithm for the queues. As a result, multi-baseline data locality can be exploited to further increase performance.

Figure 11 shows the performance of the application with the same fixed 1024 queue chunks on the PPE side, but including the data sorting on the SPE side.

We have also measured the performance effect of the chunk size variations for queue filling. Figure 12 shows the performance effect of six such sizes (64, 128, 256, 512, 1024 and 2048) for the same 100 baselines. We have only plotted the time per gridding of the 8 and 16 SPEs configurations with a support kernel of 129 × 129. The difference in performance can reach even 25% between the best and the worst tested cases. Furthermore, the best chunk sizes are not the same for both configurations, which indicates that a further study needs to address a way to compute this parameter as a function of the kernel size and the number of SPEs.

5.4. PPE multithreading

An important performance question is whether the PPE acting as a dynamic scheduler for the visibility data in the SPE queues can lead to a performance bottleneck. In other words, is there any danger of the PPE queues filling rate to be too small for the SPE processing rate? To answer this question, we have implemented a multi-threaded PPE version. Given that the QS21 blade we have used for experiments has 2 Cell/B.E. cores, we can run up to 4 PPE threads in parallel. We have measured the application performance when using 1, 2 and 4 PPE threads that fill the queues in parallel. The performance variations – within 5% of the PPE multi-threaded version when compared with the single-threaded version, are not a conclusive indication for the initial question. Therefore, we do not use the additional PPE threads, and we intend to conduct further investigation to determine if there is any other way to increase performance by PPE multi-threading.

5.5. Data-dependent optimizations

Finally, we have also applied the "data set compression" optimization, by avoiding replacing complex

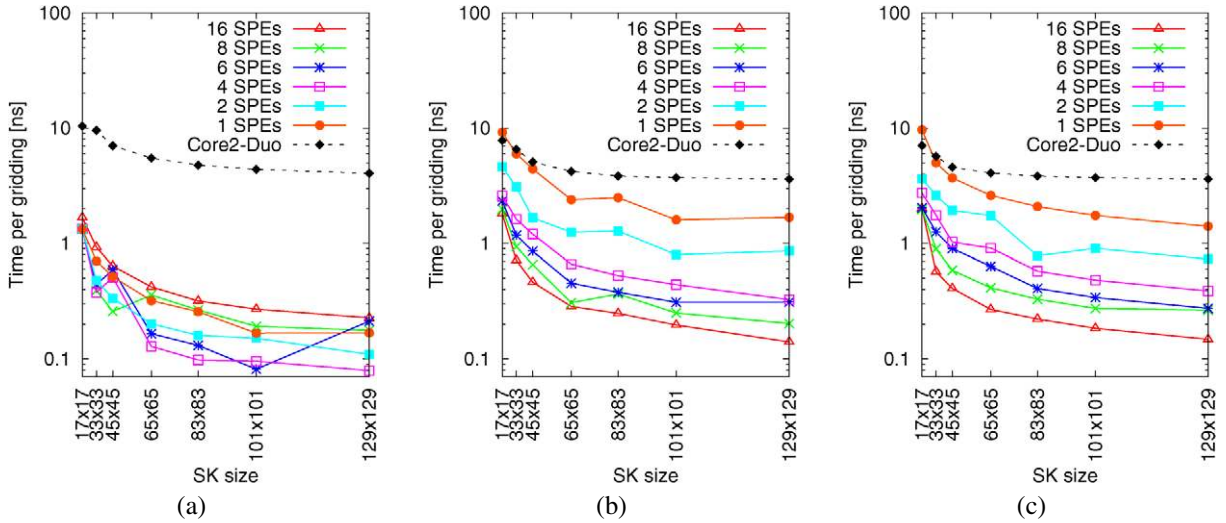


Fig. 11. The performance of the Cell/B.E. gridding with sorted SPE queues, running on data from (a) 1, (b) 10 and (c) 100 baselines. Due to the increased data locality, the application performance is significantly better.

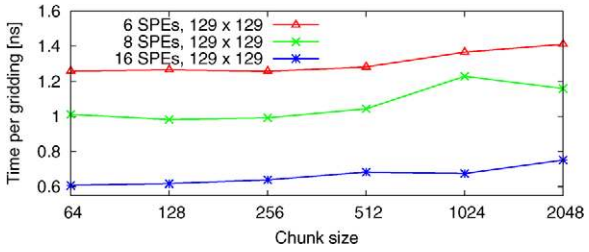


Fig. 12. The influence of the chunk sizes to the overall performance of the application. For example, the difference in performance reaches 23.5% for a 16 SPEs configuration. Note the logarithmic scale for the x axis.

multiplication with a complex addition any time a sequence of identical pairs, `c_index,g_index`, have to be scheduled. This optimization increases the execution time on the PPE side (and its utilization as well), but can lead to good performance by minimizing the DMA transfers on the SPE side. Despite the increase in performance, the SPE utilization for this solution is not significantly increased – the performance comes from reducing the number of computations the SPE has to solve and slightly increasing the computation on the PPE side. Note that this optimization is more efficient on the PPE side, as branches are more expensive on the SPEs.

Figure 13 shows the performance of the application with these data optimizations enabled.

Because a computation of the application throughput is no longer useful here, we give an estimate on our improvement by comparing the gained performance with the real measurements against two particular data sets: a random one, and one with all (u, v, w) values identical. Figure 14 shows these results. Note that, as expected, the data-compression on real measurements "fits" the real curve between the best – the one-value set – and the worst – the random set.

5.6. Summary

5.6.1. Best overall performance

To conclude our performance quest, we present a comparison between the performance obtained by the reference code and our Cell/B.E. implementation when running on data from the complete set of 990 baselines. Figure 15 present these results.

The speed-up factor on the 16 SPE of the QS21 blade versus the original implementation is between 3 and 8, while the data-dependent optimizations can increase this factor to more than 20.

Further, we present a comparison of the performance evolution for four different kernel sizes. The graph presents the performance of two sequential implementations, with and without data-dependent optimizations, as well as the performance of the Cell/B.E. implementation, using 1 or 2 processors (8 and 16 SPEs, respectively) from the QS21 blade, without any optimizations, with the queue sorting, and with the same data dependent optimizations. Figure 16 presents all these results. Note that for small kernel sizes, the performance difference between the Cell/B.E. and the original implementation running on an Intel Core2-Duo is not very large. However, as soon as the support kernel size increases, the gap becomes signifi-

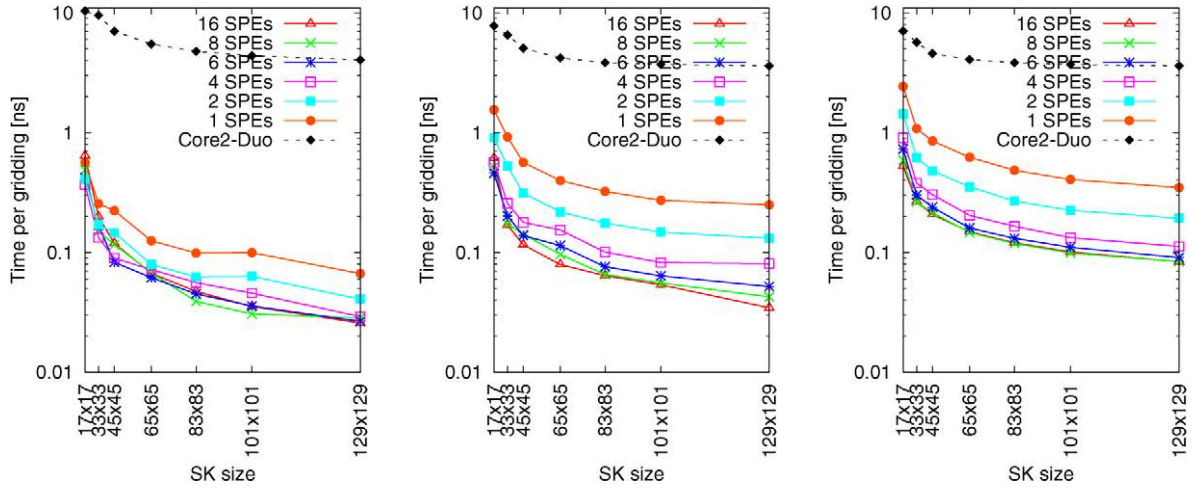


Fig. 13. The performance of application with the aggressive data-dependent optimizations, running on data from 1, 10 and 100 baselines. Note that the Core2-Duo application has been also optimized using similar data-dependent optimizations.

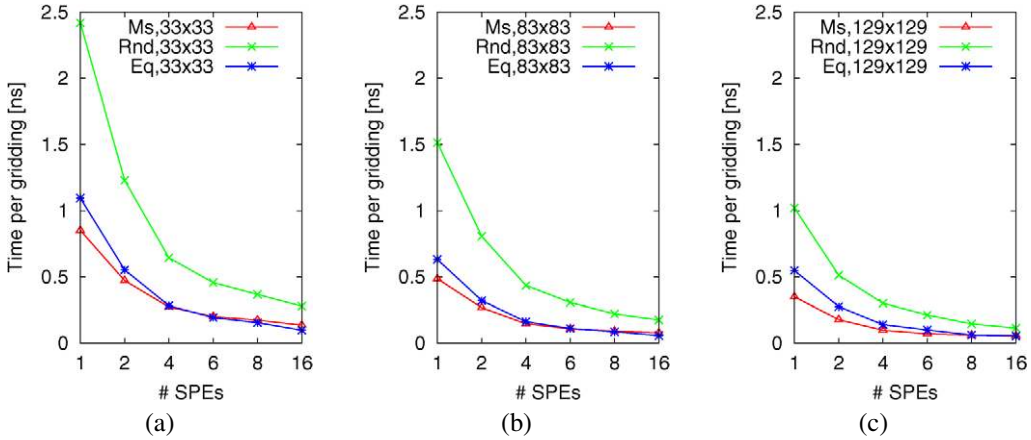


Fig. 14. The overall application performance for different kernel sizes. The number of baselines is 100. 'Ms' stands for the real measurements data set, while 'Rnd' and 'Eq' stand for the random and one-value sets, respectively.

cantly larger. Finally, note that the impact of the data-dependent optimizations is also significant for the sequential application.

5.6.2. Scalability

We estimate the scalability of this implementation by studying its behavior for various numbers of baselines. Figure 17(a–c) shows how the fully optimized application behaves for 1, 10, 100, 500 and 990 baselines. Note that for one baseline, the execution time is small, mainly because the work volume is small. For 10+ baselines, we notice a significant execution time increase (20%) – this happens because of a larger amount of data, exhibiting low data re-use. For 100 baselines or more, data re-use is increasing, thus the aggressive data-dependent optimizations become more important in the overall execution time. Overall, increasing the baseline numbers between 100 and 990 does not affect performance significantly, which in turn signals good application scalability with the number of baselines (i.e., the size of the data set). Note the difference in scale (more than a factor of 5) between the performance for small kernels and large kernels.

5.6.3. Streaming

Finally, based on the overall performance, we evaluate if the application can be used for online processing. In this scenario, data is streaming in at a given rate, and the current samples have to processed *before* the new set of samples is read. We have simulated data samples from a large number of baselines by ran-

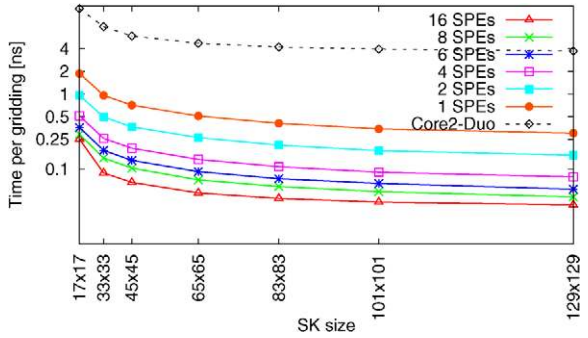


Fig. 15. The overall performance of the best implementation versus the original, reference code. The number of baselines is 990. Note the logarithmic scale.

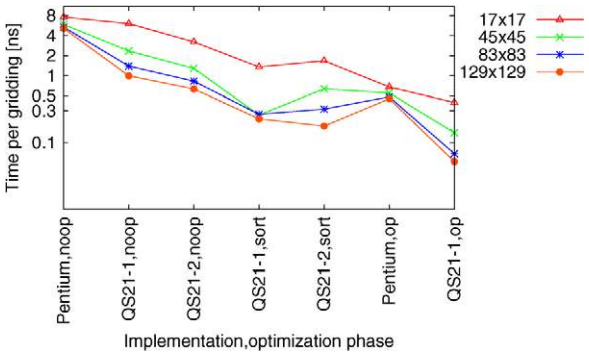


Fig. 16. The performance evolution for both the original implementation ("Pentium") and the Cell/B.E. implementations (1 and 2 processors). "noop" stands for no optimizations, "op" stands for data-dependent optimizations, "sort" refers to sorted SPE queues, and "all" is the Cell/B.E. version with all optimizations. Note the logarithmic scale.

dom numbers, and we have measured the time per gridding for $SK_M = 129 \times 129$ to be about 0.22 ns, or a time per spectral sample of 3.66 μs. For a streaming rate of 1 measurement/second, we have to process $B \times N_{\text{freq}}$ spectral samples every second. The current performance level allows about 260 baselines with 1024 frequency channels, or more than 530 baselines with 512 frequency channels. Above these limits, i.e., if the application generates more samples per second, the streaming rate is too high for the application performance, and the online processing will either require temporary buffering space, or multiple processors to split the load.

Given all these results, we conclude that the application can reach about 20–30% of the Cell/B.E. peak performance, depending on the particular problem size. Furthermore, our approach is scalable and optimized such that it manages to increase the radioastronomers' productivity by a factor of 20. Even further, these results indicate that using this implementation, on-line processing is possible using a single Cell/B.E. for systems of up to 500 baselines and 500 frequency channels. Future work will focus on increasing the SPE code performance, as well as on even finer tuning of the queuing system (automatic adjustments of the queue sizes, comparisons with other dynamic scheduling options, etc.). Finally, we note that similar applications are to be found also in other novel HPC fields, like medical imaging [21,23] or oil reservoir characterization [19].

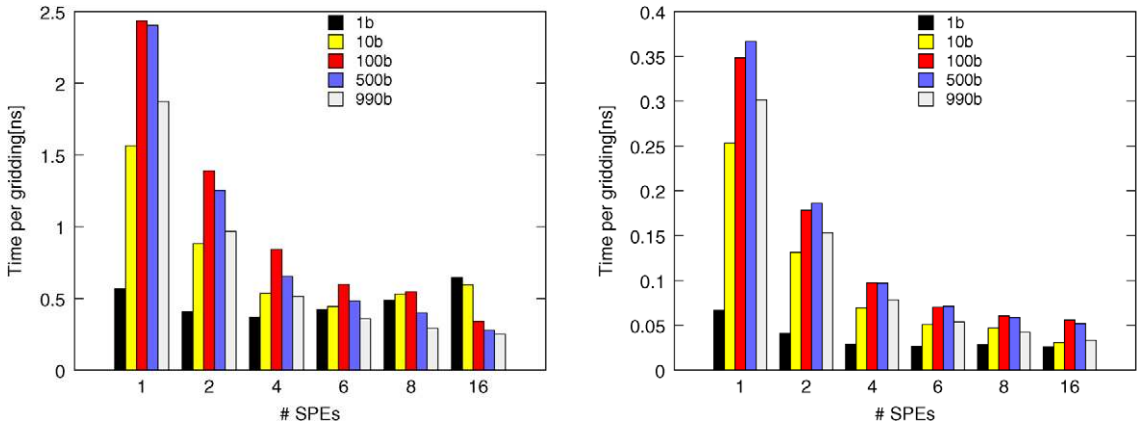


Fig. 17. Application scalability for 1, 10, 100, 500 and 990 baselines, and two different kernel sizes: (a) the smallest, 17×17, and (b) the largest, 129×129. The performance decreases between 1 and 10 baselines (due to the data-dependent optimizations, which reduce the input data set), but increases from 10 to 100+ baselines, a positive sign for the real application scale.

6. Related work

In this section we present a brief overview of previous work related to both high-performance radioastronomy applications and significant Cell/B.E. case-studies. In this context, our work proves to be relevant for both fields, quite disjoint until very recent times.

On the Cell/B.E. side, applications like RAxML [3] and Sweep3D [18], have also shown the challenges and results of efficient parallelization of HPC applications on the Cell/B.E. Although we used some of their techniques to optimize the SPE code performance, the higher-level parallelization was too application-specific to be reused in our case.

Typical ports on the Cell, like MarCell [12], or real-time ray tracing [2] are very computation intensive, so they do not exhibit the unpredictable data access patterns and the low number of compute operations per data byte we have seen in this application. Irregular memory access applications like list-ranking have been studied in [1]. Although based on similar tasks decompositions, the scheme proposed in our work is simpler, being more suitable for data-intensive applications.

Despite all these excellent case-studies for applications and application types, Cell/B.E. has only been used recently for radioastronomy applications, mainly due to I/O and streaming concerns. In fact, astronomers mainly use shared memory and MPI to parallelize their applications on large supercomputers. To use MPI on Cell/B.E., a very lightweight implementation is needed and tasks must be stripped down to fit in the remaining space of the local store. The only MPI-based programming model for Cell is the MPI microtask model [17], but the prototype is not available for evaluation. OpenMP [16] is not an option as it relies on shared-memory.

Currently, efforts are underway to implement other parts of the radio-astronomy software pipeline, such as the correlation and calibration algorithms on Cell and GPUs. Correlation may be very compute-intensive, but it is much easier to parallelize – it has already been implemented very efficiently on Cell and FPGAs [8]. Apart from the general-purpose GPU frameworks like CUDA [15] and RapidMind [14], we are following with interest the work in progress on real-time imaging and calibration [30], which deals with similar applications.

7. 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 exploited. Even worse, some applications are not naturally suitable for these processors. In these cases, additional effort needs to be put in parallelization, implementation, and optimizations to be able to achieve (very) good performance.

The gridding/degridding kernels are good examples of such applications. Used extensively in radioastronomy imaging, these data intensive kernels require a specific parallelization approach to obtain an efficient and scalable implementation on the Cell/B.E. In this paper, we have shown how to build a parallelization model for these two kernels, as well as how to implement and optimize it. In the modeling phase, we have used a top-down approach, starting from the high-level application parallelization (i.e., the task and data distribution between cores), and decomposing the model until (some of) the low-level parallelization layers could be exposed (i.e., double buffering, SIMD-ization). We have shown that the top-down decomposition of the model can capture enough information to allow for a quick, yet correct implementation. Both the modeling and the implementation techniques can be easily adapted and reused to other similar applications.

Further, we have evaluated a series data-dependent optimizations. Although such optimizations are hard to re-use directly, the performance improvements they have produced show that data-dependent application behavior has to be studied and, if possible, exploited.

Our performance results are very good: after standard optimizations, the speed-up factor of the parallel application running on a Cell blade (16 SPEs, from 2 Cell/B.E. processors) over the original, sequential application running on a commodity machine is between 5 and 10; after applying the application-specific optimizations, the speed-up factor has grown to more than 20. The overall programming effort was about 2.5 men-months. These results, combined with the proven application scalability, prove that a single Cell/B.E. can handle on-line processing for over 500 baselines and 500 frequency channels, when data is streamed at a rate of 1 sample/second.

For the future, we plan to evaluate the performance of the same application on other multi-core proces-

sors and, based on a productivity study, to classify the most efficient multi-core processor for radioastronomy imaging.

Acknowledgments

We would like to thank Michael Perrone, Fabrizio Petrini, and Daniele Scarpazza from IBM T.J. Watson, as well as Andrew Mattingly from IBM Australia, for their help.

References

[1] D. Bader, V. Agarwal, K. Madduri and S. Kang, High performance combinatorial algorithm design on the Cell Broadband Engine Processor, *Parallel Computing* **33**(10/11) (2007), 720–740.

[2] C. Benthin, I. Wald, M. Scherbaum and H. Friedrich, Ray tracing on the Cell processor, in: *IEEE Symposium of Interactive Ray Tracing*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2006, pp. 15–23.

[3] F. Blagojevic, A. Stamatakis, C. Antonopoulos and D.S. Nikolopoulos, RAxML-CELL: Parallel phylogenetic tree construction on the Cell Broadband Engine, in: *IEEE International Parallel and Distributed Processing Symposium*, IEEE Press, Los Alamitos, CA, USA, 2007.

[4] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra and G. Bosilca, SCOP3: A rough guide to scientific computing on the PlayStation 3, Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[5] T. 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*, Pasadena, CA, USA, October 2004, pp. 86–95.

[6] T.J. Cornwell, SKA and EVLA computing costs for wide field imaging, *Experimental Astronomy* **17** (2004), 329–343.

[7] T.J. Cornwell and R.A. Perley, Radio-interferometric imaging of very large fields – The problem of non-coplanar arrays, *Astronomy and Astrophysics* **261** (1992), 353–364.

[8] L. de Souza, J.D. Bunton, D. Campbell-Wilson, R.J. Cappallo and B. Kincaid, A radio astronomy correlator optimized for the Xilinx Virtex-4 SX FPGA, in: *International Conference on Field Programmable Logic and Applications*, IEEE Press, Los Alamitos, CA, USA, 2007, pp. 62–67.

[9] M. Gschwind, Chip multiprocessing and the Cell Broadband Engine, in: *ACM Frontiers*, ACM Press, New York, NY, USA, 2006, pp. 1–8.

[10] J. JaJa, *Introduction to Parallel Algorithms*, Addison-Wesley Professional, Reading, MA, USA, 1992.

[11] M. Kistler, M. Perrone and F. Petrini, Cell multiprocessor communication network: Built for speed, *IEEE Micro* **26**(3) (2006), 10–23.

[12] L.-K. Liu, Q. Liu, A.P. Natsev, K.A. Ross, J.R. Smith and A.L. Varbanescu, Digital media indexing on the Cell processor, in: *IEEE International Conference on Multimedia and Expo*, Beijing, China, July 2007, pp. 1866–1869.

[13] W. Liu, B. Schmidt, G. Voss and W. Müller-Wittig, Molecular dynamics simulations on commodity GPUs with CUDA, in: *High Performance Computing*, Springer, Berlin/Heidelberg, Germany, 2007, pp. 185–196.

[14] M. McCool, Signal processing and general-purpose computing on GPUs, *IEEE Signal Processing Magazine* **24**(3) (2007), 109–114.

[15] nVidia, CUDA – Compute Unified Device Architecture Programming Guide, 2007.

[16] K. O'Brien, K. O'Brien, Z. Sura, T. Chen and T. Zhang, Supporting OpenMP on the Cell, in: *International Workshop on OpenMP*, Springer, Dordrecht, The Netherlands, 2007, pp. 65–76.

[17] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu and T. Nakatani, MPI microtask for programming th Cell Broadband Engine processor, *IBM Systems Journal* **45**(1) (2006), 85–102.

[18] F. Petrini, J. Fernàndez, M. Kistler, G. Fossum, A.L. Varbanescu and M. Perrone, Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine, in: *IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, March 2007.

[19] M. Prange, W. Bailey, H. Djikpesse, B. Couet, A. Mamonov and V. Druskin, Optimal gridding: A fast proxy for large reservoir simulations, in: *SPE/EAGE International Conference on Reservoir Characterization*, Abu Dhabi, UAE, 2007, pp. 172–184.

[20] R. Rengelink, Y. Tang, A.D. Bruyn, G. Miley, M. Bremer, H. Rttgering and M. Bremer, The Westerbork Northern Sky Survey (WENSS), a 570 square degree mini-survey around the North ecliptic pole, *Astronomy and Astrophysics Supplement Series* **124** (1997), 259–280.

[21] D. Rosenfeld, An optimal and efficient new gridding algorithm using singular value decomposition, *Magnetic Resonance in Medicine* **40**(1) (1998), 14–23.

[22] R.T. Schilizzi, P. Alexander, J.M. Cordes, P.E. Dewdney, R.D. Ekers, A.J. Faulkner, B.M. Gaensler, P.J. Hall, J.L. Jonas and K.I. Kellermann, Preliminary specifications for the square kilometre array, Technical Report v2.4, www.skatelescope.org, November 2007.

[23] H. Schomberg and J. Timmer, The gridding method for image reconstruction by Fourier transformation, *IEEE Transactions on Medical Imaging* **14**(3) (1995), 596–607.

[24] F. Schwab, Optimal gridding of visibility data in radio interferometry, in: *Indirect Imaging*, Cambridge University Press, Cambridge, UK, 1984, pp. 333–340.

[25] A. Thompson, J. Moran and G. Swenson, *Interferometry and Synthesis in Radio Astronomy*, Wiley-VCH, Germany, 2001.

[26] K. van der Schaaf, C. Broekema, G. Diepen and E. Meijeren, The LOFAR central processing facility architecture, *Experimental Astronomy* **17**(1–3) (2004), 43–58.

[27] A. van Gemund, Performance prediction of parallel processing systems: The PAMELA methodology, in: *International Conference on Supercomputing*, ACM Press, New York, NY, USA, 1993, pp. 318–327.

[28] A.L. Varbanescu, A. van Amesfoort, T. Cornwell, B.G. Elmegreen, R. van Nieuwpoort, G. van Diepen and H. Sips, The performance of gridding/degridding on the Cell/B.E., Technical report, Delft University of Technology, January 2008.

[29] A.L. Varbanescu, A. van Amesfoort, T. Cornwell, B.G. Elmegreen, R. van Nieuwpoort, G. van Diepen and H. Sips, Radioastronomy image synthesis on the Cell/B.E. Technical report, Delft University of Technology, August 2008.

[30] R. Wayth, K. Dale, L. Greenhill, D. Mitchell, S. Ord and H. Pfister, Real-time calibration and imaging for the MWA (poster), in: *AstroGPU*, Princeton, NJ, USA, November 2007.

[31] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, The potential of the Cell processor for scientific computing, in: *ACM International Conference on Computing Frontiers*, ACM Press, New York, NY, USA, 2006, pp. 9–20.