

Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels

Alexander S. van Amesfoort,
Ana L. Varbanescu, Henk J. Sips
Delft University of Technology
The Netherlands

Rob V. van Nieuwpoort
ASTRON
Dwingeloo
The Netherlands

ABSTRACT

Multi-core platforms have proven themselves able to accelerate numerous HPC applications. But programming data-intensive applications on such platforms is a hard, and not yet solved, problem. Not only do modern processors favor compute-intensive code, they also have diverse architectures and incompatible programming models. And even after making a difficult platform choice, extensive programming effort must be invested with an uncertain performance outcome. By taking the plunge on an irregular, data-intensive application, we present an evaluation of three platform types, namely the generic multi-core CPU, the STI Cell/B.E., and the GPU. We evaluate these platforms in terms of application performance, programming effort and cost. Although we do not select a clear winner, we do provide a list of guidelines to assist in platform choice and development of similar data-intensive applications.

Categories and Subject Descriptors

C.1.3 [Processor Architectures]: Other Architecture Styles – Heterogeneous (hybrid) systems; C.1.4 [Processor Architectures]: Parallel Architectures; C.4 [Performance of Systems]: Design studies – Performance attributes; D.1.3 [Programming Techniques]: Concurrent Programming – Parallel Programming

General Terms

Performance, Measurement, Algorithms

Keywords

The Cell Processor, GPUs, Multi-core processors, Data-intensive kernels, Memory-bound applications

1. INTRODUCTION

High performance computing (HPC) applications used to run on supercomputers with vector or special purpose processors (for example, for digital signal processing or physics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'09, May 18–20, 2009, Ischia, Italy.

Copyright 2009 ACM 978-1-60558-413-3/09/05 ...\$5.00.

simulation). With general purpose hardware getting faster and cheaper, clusters took over the job largely.

Limitations from using more power and instruction-level parallelism (ILP) redirected processor design to increase core count. Resulting chip multi-processors (multi-cores) started to penetrate all areas of computing around 2005. Since then, multi-core platforms have proven useful for many high performance applications (see for example Sweep3D [22], RaX/ML [3], molecular dynamics [18], N-Body simulations, medical applications [23]). Consequently, new clusters employ multi-cores to solve increasingly bigger computational problems. However, not only general purpose CPUs are being designed as multi-cores. With various multi-core architectures coming from different areas to snatch a slice of the HPC market, HPC application designers have to consider different CPUs (such as Xeon, Opteron, Power, Niagara) as well as multi-core accelerators (such as Cell/B.E., GPUs, FPGAs). Generic multi-core CPUs have to run a lot of single threaded code for the foreseeable future, so newer generations duplicate cores and continue to enlarge vast cache hierarchies (Xeon, Opteron, Power). Higher cache levels are shared by more hardware threads. Specialized processors may be heterogeneous for better performance per Watt. Architectures may trade in a lot of processor state and control logic for many more, but smaller compute cores. With little on-core state per thread, memory latency must be hidden with rapid context switching. Cache and coherency logic can also be traded-in for scratch pad (distributed) memories which operate completely under software control.

In this paper we evaluate the three multi-core types in terms of performance, programming effort and cost. Together, these platform types cover a wide range of architectures targeted at HPC applications. (1) The multi-core CPU (MC-CPU), like a x86-64 dual-/quad-core, is a generic CPU with a traditional memory hierarchy. (2) The STI Cell/B.E. is a heterogeneous nine core processor, with a scratch pad memory per core and DMA transfers to access main memory. (3) The NVIDIA Geforce GPU has up to a few dozen SIMD multi-processors, and combines small caches and scratch pad memories.

To use multiple cores, algorithms must be modified to work on independent jobs of substantial size. Sequential code may not be directly reusable, as it often exhibits inefficient memory behavior and doesn't exploit performance enabling hardware features (like scratch pad memories, DMA, SIMD and hardware synchronization mechanisms). To reach peak performance, multiple layers of parallelism (in-core, multi-core and multi-processor (and for grids even multi-

cluster)) have to be dealt with. Even for experts, understanding and solving bottlenecks can be a black art. In addition, getting close to hardware peak performance is mostly reserved for applications with compute-bound kernels. Just how compute-bound an application is, can be quantified by the arithmetic intensity. This ratio is defined as the number of compute operations per transferred byte [31, 6]. With multiple cores accessing memory concurrently in interleaved patterns, we are still moving head-first into the increasing gap between compute and memory performance known as the memory wall. Typically, data-intensive applications have low arithmetic intensity, as the amount of processing per data item is low. For HPC, this means that data-intensive applications require drastic, application- and platform-specific modifications, leading to low portability. The price to pay is, that after a difficult platform choice, significant programming effort must be invested. And as bandwidth growth continues to fall behind core count increases, the situation is getting worse. This trend is exactly what makes studies into data-intensive applications on multi-cores worthwhile.

The evaluation presented in this paper provides three types of contributions, useful for other data-intensive applications that target multi-core processors: (1) guidelines to make an informed platform choice, (2) guidelines to implement the application on these platform types, and (3) insight into implementation effort and into performance behavior of optimizations and algorithmic properties.

The rest of this paper is organized as follows: In Section 2 we discuss our application kernel and its properties. The three types of platforms and the programming methods for our application are described in Section 3. Section 4 contains our experiments, followed by a discussion in Section 5. We describe relevant related work in Section 6, and provide our conclusions in Section 7.

2. DATA-INTENSIVE APPLICATIONS: AN EXAMPLE

In data-intensive computing, very large amounts of data have to be analyzed. In every stage of the analysis, data reduction takes place. There may be additional real-time requirements to keep up with the stream or to respond to events by changing analysis stages without restarting the whole process.

Some of the most challenging data-intensive processing pipelines can be found in radio astronomy. Radio astronomers want to build ever bigger telescopes, but it is impractical to construct rotatable, steel dishes with a diameter in excess of 100 meters. Already, a number of large dishes are combined using the technique of radio interferometry [10] in synthesis arrays [5], placed at distances ranging from few hundred meters to thousands of kilometers. Meanwhile, several next generation radio telescopes, like LOFAR and SKA [24, 9] are being built out of many dozens, or in a couple of years, up to a thousand small antenna stations. These “software radio telescopes” generate data streams in the order of many Tbit/s or more to be analyzed. Initial software will process the first computational phase on-line and then dump data to storage. However, it is highly desirable to push back the point where data is dumped to disk, as more on-line data reduction means less required storage. Processing must keep up with the data rate, or else data must be dropped.

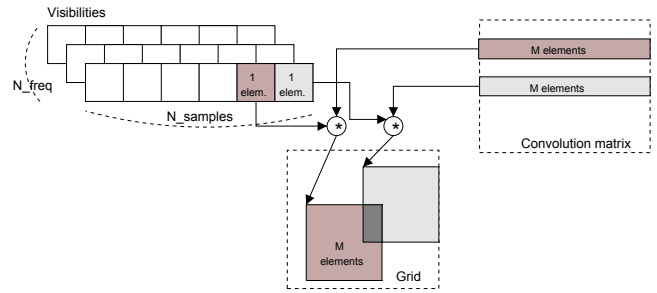


Figure 1: The gridding kernel and its access pattern. Similarly shaded regions are for a single visibility sample. Newly computed data that overlaps existing grid values is added on.

One data-intensive kernel often found in astronomy software pipelines is *convolutional resampling*. Briefly, it is used to convert an incoming data stream with telescope measurements, called visibilities, into a grid-model of the sky. Convolutional resampling can be solved using the W projection algorithm [8], a low-computation alternative to classical solutions (like, for example, the faceted approach). An extensive analysis of this application is available in [27]. The most time-consuming kernels of this application are two complementary operations, gridding and degriding. Gridding transforms visibilities sampled in the (u, v, w) space (along the antenna trajectories), into a regular grid. Degriding is the inverse operation.

The gridding kernel code is given in Listing 1. Its main data structures are four arrays: the measurement coordinates, (u, v, w) , the visibility data, V , the convolution coefficients matrix, C , and the computed grid, G . The oversampling factor is denoted by OS . P_w is the number of W planes used by the W projection. Typical values for these constants depend on various telescope and observation parameters. Without any loss of generality, we have chosen to analyze the gridding, using $OS = 8$, $P_w = 33$ and varying M , the convolution kernel size. A typical one-night astronomical measurement generates a few thousands samples ($N_{samples}$) per antenna, on thousands of frequency channels (N_{freq}).

Listing 1: The gridding computation.

```

1 // for all visibilities:
2 forall (j=0..Nfreq-1; i=0..Nsamples-1)
3 // conv kernel position in C:
4   compute cIndex=cOffset((u,v,w)[i],freq[j]);
5 // grid position to add to
6   compute gIndex=gOffset((u,v,w)[i],freq[j]);
7 // sweep the convolution kernel:
8   for (x=0; x<M; x++)
9     G[gIndex+x]+=C[cIndex+x]*V[i,j];

```

The `c_index` stores the offset inside C from where the current convolution coefficients are extracted; the `g_index` stores the offset of the G subregion where the newly computed data is added. These two indices are computed for each time sample and frequency channel, thus corresponding to each visibility. An intuitive schematic of the gridding and degriding operations is given in Figure 1.

The arithmetic intensity of gridding (and degriding) is 0.33, so for every floating point operation, three memory

bytes ($2 \times RD + 1 \times WR$) need to be displaced. Together with the irregular, unpredictable memory accesses into \mathbf{C} and \mathbf{G} , we use gridding as a typical example of a HPC data-intensive kernel. A symmetric partitioning scheme of the large data structures, \mathbf{C} and/or \mathbf{G} over available cores leads to heavy load imbalance, as shown in [26]. No single multi-core platform can possibly process all incoming data, so sub-streams have to be directed into different systems. Per system, certain (overlapping) regions of \mathbf{C} and \mathbf{G} are used much more than others. Solving these irregularities beforehand in a pre-processing step (thus by pre-sorting all data) is to be avoided as it restricts the implementation to off-line use cases.

Our study of data-intensive applications on multi-cores starts from porting and then optimizing the gridding kernel for three different multi-core platforms. To assess platform sensitivity to differences in arithmetic intensity and memory access irregularity, we varied these application properties. Arithmetic intensity was artificially changed by inserting additional computations on loaded data on line 9 of Listing 1. These operations have been carefully chosen to minimally affect register requirements, to avoid the compiler from optimizing them out, and to represent similar computational patterns (thus the same fraction of multiply-adds). For the performance studies, we have varied the convolution kernel size (M), thus assessing the effect of irregular memory accesses: larger kernel sizes generate more contiguous memory operations. Finally, for scalability studies, we have investigated the potential of such a data-intensive application to scale with the number of used cores.

3. PLATFORMS AND PROGRAMMING

In this section, we describe the three multi-core architectures and we present our parallelization efforts for the application kernel. We close this section with a comparison summary.

3.1 Multi-core CPU (MC-CPU)

Multi-core CPUs have the advantage of availability of hardware and development tools, and of cost, as they are by far the most extensively used processors in server, desktop and HPC machines. Our test platforms in this category are three systems with up to eight x86-64 cores and with SSE2 support for SIMD operations. On the quad core Xeon 5320 processor, two cores share a L2 cache. Cache coherency traffic between L2 caches passes through the front-side bus. The Core i7 has four cores, but exposes eight cores using hyperthreading. All systems use a UMA memory model with at least 4 GB RAM. The standard programming model for this platform type is based on multi-threading.

3.1.1 Gridding on the MC-CPU

Using block-wise, symmetrical visibility data distribution, we parallelized the application using `pthread`s within a few days. To solve the problem of concurrent writes to the grid, each thread works on a private grid. This approach is more efficient than a locking scheme, because the accesses into the grid and convolution matrices for each data value can start at any position and would otherwise require a shared locking structure, wasting precious memory bandwidth. Besides, conflicts are rare, except in a number of hot spots. A final reduction step centralizes the private grids, a step that can be parallelized as well without access conflicts by partitioning.

As a next step, symmetrical data distribution was replaced for a master-worker approach. Workers are still operating on a private grid, but fetch jobs from a private queue, which is filled by the master. The master improves the worker cache benefits by increasing data locality in the \mathbf{C} and/or \mathbf{G} matrices. To do so, it places jobs with (semi-)overlapping matrix areas in the same queue, without leaving any queue empty.

In this case, the computation of grid and convolution matrix offsets has to be performed by the master. We limit the filling rate with a maximum queue size to avoid effectively sorting too much data, which would be unrealistic in a streaming environment. We have also improved in-core performance of the workers with SIMD intrinsics.

3.2 STI Cell Broadband Engine (Cell/B.E.)

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor, designed by Sony, Toshiba and IBM. It was initially designed for the Playstation 3 (PS3) game console and is now also used for multimedia and HPC applications. In fact, for HPC applications, there are dedicated Cell-blades (QS20/21/22) with two interconnected Cell/B.E. processors.

The Cell/B.E. has nine cores: one Power Processing Element (PPE), acting as a main processor, and eight Synergistic Processing Elements (SPEs), acting as co-processors. A high-bandwidth Element Interconnection Bus (EIB) connects all cores, main memory, and I/O interfaces in a ring topology. The theoretical peak performance of the SPEs is 204.8 GFlops (single precision) and the combined bandwidth of the EIB is 204.8 GB/s [15]

The PPE contains the Power Processing Unit (PPU), which is a dual-threaded 64-bit Power core with VMX/Altivec unit, 2x 32 kB split instruction and data cache, and 512 kB L2 cache. The PPE's main role is to coordinate the SPEs. An SPE contains a RISC core (the SPU), 256 kB Local Storage (LS), and a Memory Flow Controller (MFC). The LS is used as local memory for instructions 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. 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. Double precision 64-bit floating point operations are issued at the lower rate of two double-precision operations every seven SPU clock cycles (14.6 GFlops).

The Cell/B.E. cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [30, 22] 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.

3.2.1 Gridding on the Cell/B.E.

To alleviate the severe SPE stalling due to the data-intensive application pattern, we have used a dynamic master-workers approach, based on the idea that each SPE works on the in-

ner loop to do the griddings, while the PPE acts as a job scheduler. The PPE sends each SPE a visibility and the associated offsets into the convolution matrix and the grid.

To increase SPE utilization, the PPE reserves a queue for each SPE, and continuously places job IDs in all these queues. Each SPEs polls its queue and solves pending work, thus reducing the communication to the PPE at a minimum. As the PPE has a good overview of all the sub-regions that are under computation at the SPEs, it can apply scheduling optimizations. If the SPEs work much faster than their queues are filled, the scheduling process on the PPE could become the application bottleneck. To alleviate this potential problem, we have implemented PPE multi-threading (with 2 and 4 threads, depending on the hardware platform). The performance gain was insignificant - below 7%, on average, showing that even with multiple threads, the speed ratio between the work producers (PPE threads) and consumers (SPEs) is too low to avoid SPE stalling.

Each SPE performs a simple loop: bring in the convolution data via DMA, perform the gridding, and do the DMA-out transfer. We further optimize this loop by skipping redundant DMA operations if the next convolution area overlaps with the one already loaded. Similarly, no DMA-out is performed if grid lines overlap. Finally, basic SPE-specific optimizations, like code vectorization, loop unrolling, and DMA double buffering have been applied for the gridding computation itself.

3.3 Graphical Processing Units (GPUs)

For this category, we use NVIDIA Geforce and Tesla GPUs with 16 or 30 SIMD multi-processors. Each multi-processor offers various local storage resources, such as 8k or 16k registers and 16 kB scratch-pad ("shared") memory, which are dynamically partitioned, and also a constant cache of 8 kB. Each cluster of 2 or 3 multi-processors share a single texture cache (8 kB per multi-processor). Constant memory (64 kB) and texture memory are stored off-chip in up to a few GB of GPU ("global") memory. A suitable way to program NVIDIA GPUs for general purpose applications is by using the API from the Compute Unified Device Architecture (CUDA) [21]. CUDA is a hardware/software architecture developed by NVIDIA to perform general purpose computations on a GPU, without the need to first translate an application into the computer graphics domain. In CUDA's Single Instruction, Multiple Threads (SIMT) model, each execution unit in a multi-processor is often counted as a core, as it runs a thread that can reference any memory address (gather/scatter) and can take any branch independently. However, each branch taken by at least one thread has to be executed by the multi-processor, masking out threads that chose a different code path. A compiler, runtime libraries and driver support are provided to develop and run CUDA applications. CUDA extends the C/C++ programming languages with vector types and a few declaration qualifiers to specify that a variable or routine resides on the GPU. It also introduces a notation to express a kernel launch with its execution environment. A kernel is launched by specifying the number of threads in a group ("block") and the number of blocks that will execute the kernel. On the GPU, a thread computes the locations of its data by accessing built-in execution environment variables, such as its thread and block indices. A global hardware scheduler assigns a block to a multi-processor when enough resources are available and a

local hardware scheduler determines which (partition of a) block runs next.

3.3.1 Gridding on GPUs

NVIDIA provides BLAS (CUBLAS) and FFT (CUFFT) libraries to accelerate the porting effort of an application based on BLAS or FFT kernels. Required functions of the CUBLAS library were taken as a starting point to port our application kernel. Since the source code of CUBLAS and CUFFT are available for download, we could specialize the required routines. We pushed looping over the visibilities into the kernel to launch a kernel only once. The visibility data is partitioned block-wise, but to run efficiently, we need at least a few hundred thread blocks. Especially on this platform, we cannot synchronize write conflicts efficiently. Atomic memory update instructions, as available on some GPUs, don't operate on floating point data. A grid locking scheme would be expensive in terms of performance. We cannot send computed grid contributions to host memory to aggregate there, because this turns a data reduction into a data expansion, worsening the I/O bottleneck on the PCIe bus. We have also considered abandoning CUDA in favor of OpenGL/GLSL, to take advantage of the graphics framebuffer blending hardware for atomically updating the grid values. This approach was also rejected, because the performance of this hardware on 32 bit floating point data is too low. Thus, to avoid write conflicts, each block of threads writes to a private grid. This solution ultimately limits the size of the grid (bad news for real astronomy usage), and apart from waiting for GPUs with more memory, only significant algorithmic changes may offer an alternative.

To improve performance, we experimented with texture fetching for the grid and the convolution matrix. We access the convolution matrix as a 1D texture. Texture caches are small and only cache reads. To access the grid, we only generate aligned and contiguous memory requests to allow the hardware to generate coalesced memory transfers on complex values (float2), substantially improving memory bandwidth. The grid updates for each data sample are computed by a block of threads working on a row at a time, which is faster than having a thread for each update, and also avoids reaching the upper limit of 512 threads per block. Other minor optimizations we tried include 2D texturing, fetching common data for all threads in a block only once and spreading it through shared memory, and software prefetching. GPUs with more memory can accomodate more private grids, so we also tuned the number of launched blocks to the size of GPU memory.

No data-dependent optimizations and/or job distribution have been applied to the GPU implementation. For example, we do not use the host CPU to fill multi-processor private queues with overlapping jobs. Although such a solution may improve data locality, its GPU performance benefits are limited by the small texture caches. Further, the coordination provided by the host CPU is too slow for the GPU's embedded hardware multi-processor scheduler, which only works efficiently on large amounts of jobs submitted at once.

3.4 Platform Comparison

We summarize the platform descriptions for parallelization in Table 1. An essential part of the performance of data-intensive applications is how each platform type deals with memory latency hiding. Multi-core CPUs hide mem-

ory latency with a cache hierarchy and prefetching. This requires temporal (and spatial) data locality. Local memory on the Cell/B.E. has to be managed by software and can hide main memory latency with prefetching (double buffering). Data for the next computation can already be fetched, while computing on the other half of local memory. On the GPU, memory latency is hidden with rapid context switching between many threads.

The Cell implementation was written first, followed by the other two. The followed order of implementation is not intentional. Although some ideas from the Cell implementation have been carried over to the MC-CPU, most code had to be changed substantially. There is very little code reuse between the platforms in data distribution and computation, because the high-level strategies to achieve high performance for our application are too different. Estimating which optimization will pay off the most beforehand is hard, and the code is easy to break. If the result grid is wrong, it can be time-consuming to locate the cause, so it is important to verify the result after each small modification.

4. EXPERIMENTS AND RESULTS

In this section we present the performance of the sequential and parallel versions of the application kernel at various optimization levels and with various parameters.

4.1 Experimental Setup

For each platform instance, we present performance graphs for convolution kernel sizes of 16x16, 32x32, 64x64, and 128x128. Apart from reporting on application performance, we also use it to demonstrate the effect of varying memory request irregularity. We show achieved performance from varying the arithmetic intensity from 1x to 2x, 3x, 6x, 12x, and 24x of our gridding performance results. The gridding algorithm has an arithmetic intensity of 0.33, but optimizations to an implementation change the actual arithmetic intensity: improving locality increases it, while compute optimizations decrease it. We also discuss the effect of the optimizations proposed for each platform on both performance and programming effort.

All tests were run under the Linux operating system, using the GNU C compiler for all multi-core CPUs and Cell/B.E. systems and CUDA 2.1 with driver version 180.22 for the GPUs. We have used several platform instances and we present them in Table 2. The Compute Rate, Memory Bandwidth, and Flop/Byte columns list theoretical maxima for single precision floating-point. For the MC-CPU instances, the highest level caches are shared by multiple cores, so they are listed per processor. For the GPUs, we count multi-processors (not cores) and list local memories per multi-processor. From the Flop/Byte ratios, we can already see that for data-intensive applications, even if we can use memory bandwidth optimally, we are going to severely under-utilize the computing capabilities of all platforms, unless we can expose significant data reuse in local memories.

4.2 Experiments with the MC-CPU

Figure 2 presents the best performance results we obtained on the MC-CPU platforms. The somewhat older Quad Opteron suffers from low bandwidth and small caches. As a dual CPU system, the Xeons beat a single Core i7, but only for the larger kernel sizes. Core i7 can only store a significant amount of data in L3 cache, which lives relatively

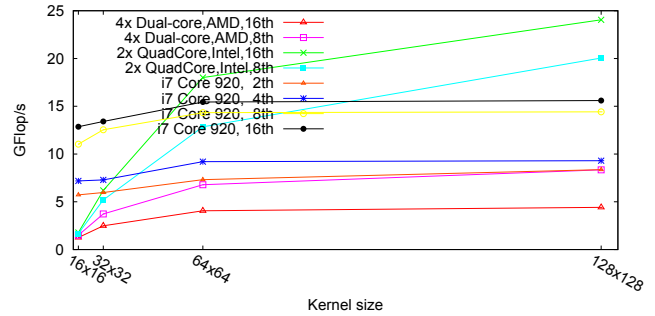


Figure 2: Application performance (GFlop/s) running on the MC-CPU platforms, using different numbers of threads

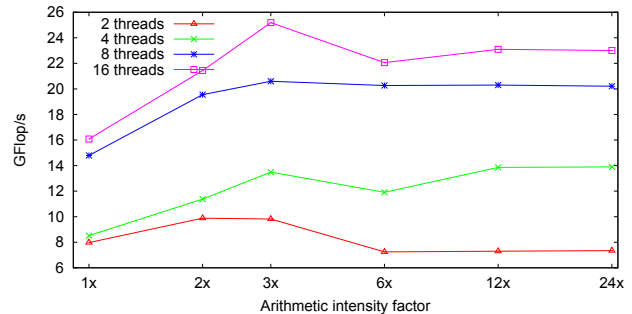


Figure 3: Performance (GFlop/s) running with various arithmetic intensity factors on the Core i7

far away in a slower clock domain (the “uncore”). Off-core is the new off-chip. On smaller kernel sizes, access irregularity is worse and the large, shared cache saves the Core i7 showing good performance over the whole range of kernel sizes. Even though none of the test platforms can run more than eight threads in parallel, some of them show a significant performance gain with more than eight threads. This behavior is caused by the scheduling of our threads and background tasks.

The performance results from various arithmetic intensities are shown in Figure 3. With one thread per virtual core, performance climbs nicely as we approach the system’s Flop/Byte ratio. Past there, the top has been reached and a tiny slowdown is observed. Compared to one thread per physical core, hyperthreading seems to work well. The reason why both 2 and 16 threads lose significantly at higher arithmetic intensity is not clear to us.

Table 3: MC-CPU Optimizations

Optimization Step	Compute rate [GFlop/s]	Gain Factor [vs. Base]	Effort [days]
Base	1.6	-	-
1	7	4.4x	4
2	18	11.3x	2
3	24.8	15.5x	4

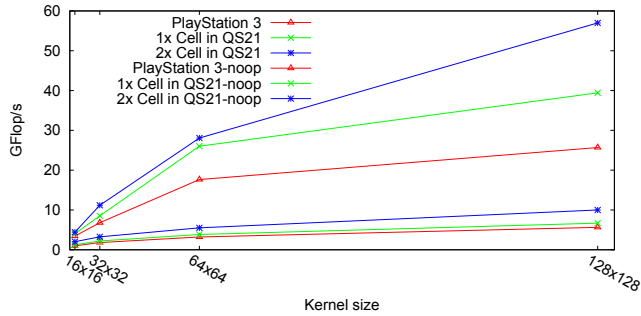
We present a brief optimization evolution in Table 3 of the effort and performance gains for a convolution kernel size of 128x128 running on the dual Xeon system. Optimization 1 is the simple parallelization with a symmetrical data distribution of the visibilities stream over the available cores, implemented using pthreads. Note that the speed-up is not as good as expected, due to the data-intensive application

Table 1: The characteristics of the platform-specific parallelizations

Platform	Memory Model	Core Type	Programming Model	In-Core Optimizations	Data Distribution	Task Scheduling
MC-CPU	Shared memory, Coherent caches	Homogeneous	Multi-Threading	SIMD	Master-Worker Queues	OS
Cell	Distributed Memory	Heterogeneous	Cell SDK	SIMD, Multi-Buffering	Master-Worker Queues	Application
GPU	Distributed Memory, Small Caches	Homogeneous	CUDA SIMT	Memory Accesses	Symmetric, Blocks	Hardware

Table 2: Platform characteristics

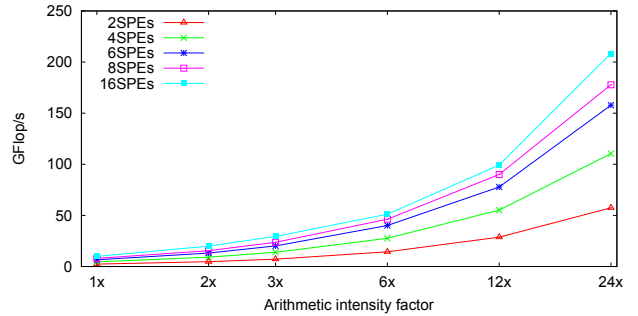
Platform Instance	Cores	Clock (GHz)	Memory per Core (kB)	Main Memory	Compute Rate (GFlop/s)	Memory Bw (GB/s)	Flop/Byte
Dual Intel Xeon 5320	2x4	1.86	32+32 L1 2x4096 L2	8 GB	59.5	21.3	2.8
Quad AMD Opteron 8212	4x2	2.00	64+64 L1 2x1024 L2	4 GB	64.0	10.7	6.0
Intel Core i7 920	4 (HT)	2.66	32+32 L1 256 L2 1x8192 L3	6 GB	85.2	32.0	2.7
STI PS3-Cell/B.E.	1+6	3.20	256	256 MB	153.6	25.6	6.0
STI Cell/B.E.	1+8	3.20	256	1 GB	204.8	25.6	8.0
STI QS21-Cell/B.E.	2+16	3.20	256	2 GB	409.6	51.2	8.0
NVIDIA Geforce 8800 GTX	16	1.35	16+8+8	768 MB	345.6	86.4	4.0
NVIDIA Tesla C1060	30	1.30	16+8+8	4 GB	936.0	102.0	9.2
NVIDIA Geforce GTX 280	30	1.30	16+8+8	1 GB	936.0	141.7	6.6

**Figure 4: Application performance (GFlop/s) running on the Cell/B.E. instances**

behavior. Next, optimization 2 is the implementation of a “queuing-system”, where each core receives a set of data to work on from the master thread. Further, the master assigns adjacent visibilities to cores such that data from the same grid/convolution region can be reused. Finally, in the third step, these “local queues” are sorted locally (i.e., by each core), further increasing temporal data locality, thus speeding up the application.

4.3 Experiments with the Cell/B.E.

As mentioned in Section 3, Cell/B.E. was the most demanding of the platforms we have ported our application to, mainly due to its distributed memory, as well as because it was the first (and stripping the application core from the astronomy details was non-trivial). Figure 4 presents the best performance results for various kernel sizes on the Cell/B.E. instances. The “noop” measurements show the performance without any data-dependent optimizations, which makes a lot of difference. Note that the application scales well with the number of utilized cores. Performance results from various arithmetic intensities on a QS21 are shown in Figure 5.

**Figure 5: Performance (GFlop/s) running with various arithmetic intensity factors on a QS21**

We have tried several optimizations, each one leading to more or less surprising results. For example, we have applied the low-level SIMD optimizations and obtained about 30% overall improvement over the non-SIMD code, indicating how low the arithmetic intensity of the application is. Further, double buffering has added some additional 20% to the original code. However, the optimizations that have paid off the most were high-level optimizations, i.e. those optimizations that tackled the data-intensive behavior of the application, not only its computational behavior. Therefore, the reference Cell/B.E. implementation (the “Base”) already includes the core-level optimizations. Table 4 presents the effects of these high-level optimizations. In the first step, we have replaced the original symmetrical data distribution with a dynamic, round-robin one, based on individual SPE queues. The PPE acts as a master and fills the queues with equal chunks of data. In a second step, the PPE acts as a smarter dispatcher, and distributes the computation such that the regions used by the SPEs from either the convolution matrix or the grid overlap. Such an approach increases data reuse, and has a significant impact on performance. Fi-

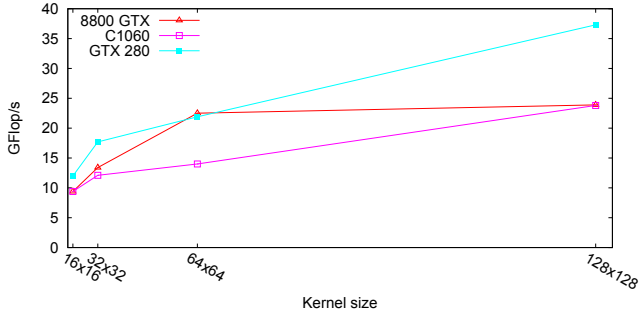


Figure 6: Application performance (GFlop/s) running on the GPUs

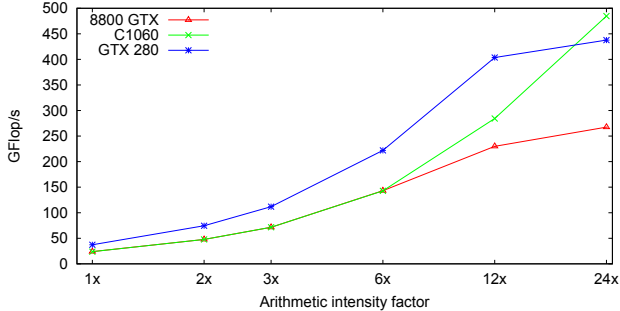


Figure 7: Performance (GFlop/s) running with various arithmetic intensity factors on the GPUs

nally, in a third step, each SPE sorts its local copy of the data queue, such that the locality is increased even more.

Note that we have applied more aggressive tuning on the Cell/B.E. by using data-dependent optimizations. However, despite the excellent performance outcome (a further improvement of a factor two in execution time), these optimizations minimize the number of performed operations by using a property of the application-specific computation. We do not report here these final results because of two reasons: (1) the same optimizations could be envisioned for all platforms, and would lead to similar gains, and (2) because they are very application specific, they may alter the generic data-intensive behavior we tackle with the other steps.

Table 4: Cell/B.E. Optimizations

Optimization Step	Compute Rate [GFlop/s]	Gain Factor (vs. Base)	Effort [days]
Base	2	-	-
1	6	3.0x	10
2	14	7.0x	25
3	57	23.5x	3

4.4 Experiments with the GPU

Figure 6 presents the best performance results for various kernel sizes on the GPUs, while Figure 7 presents the results from various arithmetic intensities.

Given the relative memory bandwidths of the 8800 GTX and GTX 280, their results make sense. We are unable to explain the performance of the C1060, which is only able to overtake the 8800 GTX on a kernel size of 64x64. An arithmetic intensity of four or more seems to wake it up. The memory bandwidth test from the NVIDIA CUDA SDK

shows that its effective bandwidth is a bit lower, but not that serious. As far as we know, it is architecturally the same as a GTX 280, but with more memory providing lower bandwidth.

Table 5 presents the effort of three optimization steps and the performance gain on the 8800 GTX GPU. Optimization

Table 5: GPU Optimizations

Optimization Step	Compute Rate [GFlop/s]	Gain Factor [vs. Base]	Effort [days]
Base	1.6	-	-
1	4.9	3.1x	10
2	22.3	13.9x	8
3	23.8	14.9x	12

step 1 involves running the application on the GPU by means of modified CUBLAS routines and moving the looping over the visibilities into the GPU. After using coalesced memory accesses in optimization step 2, the used memory bandwidth is at more than 90%. According to the NVIDIA CUDA Programming Manual [21], bandwidth of non-coalesced transfers is about four times as low as coalesced transfers on 64 bits requests, which confirms our experiences quite well. Although it would be possible to generate larger transfers by reading two complex numbers per thread (using the float4 data type), this would not improve performance significantly, as larger coalesced requests cannot be generated, and make the code more complex having to deal with more corner cases. In the final optimization step 3 we combine a few minor optimizations that together deliver under 10% extra performance, such as the best result from using the texture cache. The optimizations have a larger effect when tested against non-coalesced memory transfers, but applying them first would give a false indication of their gain. We did not include the effect of software prefetching, that is, loading convolution matrix and grid values for the next processing iteration. The performance gain is less than 3% and the resulting code quality makes sure that it is the last optimization that can be implemented.

4.5 A Brief Comparison

The Cell/B.E. is a lot faster on all kernel sizes than all of its competitors from the same generation and even slightly ahead of the GTX 280. If the queueing and sorting optimizations don't apply well to the input data, GPUs score best performance-wise, due to superior memory bandwidth. Although extensive optimizations can also be applied to multi-core CPUs with great relative effect, its performance is still below that of the other platforms, although for very small kernel sizes, the Core i7 is getting close. In data-intensive applications, the strength in performance per core of MC-CPU's cannot be leveraged.

5. DISCUSSION

Although a data-intensive application can be sped up to good application performance when spending a lot of effort, performance is still a large factor below typical performance achieved by multi-core friendlier applications. High memory bandwidth and arithmetic intensity are the main factors that influence the attained performance-level for a reasonable effort approach. Available memory bandwidth

can only be exploited if requests are regular (thus continuous, aligned, and prefetchable). Although the arithmetic intensity in data-intensive applications is often low, it must be increased by exploiting data reuse in local memory. If regular accesses do not fit the algorithm and the algorithm or platform allows little local data reuse, performance loss can be an order of magnitude. In that case the algorithm designer must go back to the drawing board.

The MC-CPU platform shows heavy under-utilization, as irregular memory accesses foil efficient caching. The Cell/B.E. performs much better, but its potential can only be uncovered after extensive data-dependent optimizations. The price to pay is a lot of programming effort into highly application- and platform-specific code. However, we have gained insight into the possible pay-off. The 8800 GTX GPU programmed using CUDA severely outperforms the other platforms only if data-dependent optimizations are ineffective, thus slowing down the other candidates. Current generation GPUs perform better, but do not overtake a single Cell/B.E. in our application.

Nevertheless, the application exposes two significant problems. First, the possibility of write conflicts in the grid requires replication for concurrent computation, thus limiting the maximum grid size. All platforms suffer from this problem, but with the need for many thread blocks and limited memory, the GPU suffers the most: only some sort of pre-processing to guarantee conflict-free writes can help, but it remains to be seen if spending a lot of cycles on the host CPU pays off in real HPC application environments. Second, data locality can be increased by optimizing data ordering. While this is possible on the CPU and Cell/B.E., using powerful general purpose cores, it is harder on the GPU, where using the “remote” host processor adds significant performance overheads, because core-local memories cannot be leveraged and all requests must pass through global memory.

In addition to insight into performance gains and implementation effort, we have drawn a number of guidelines to make an educated platform choice for implementing data-intensive applications on multi-cores:

- For data-intensive applications, choose a platform with high memory bandwidth and with a local memory size large enough to fit more than twice the size of a job.
- It is easier and more efficient to implement overlap of computation and communication on architectures with software-managed local memory, such as Cell/B.E., and a GPU. Applications that have larger job footprints, need the large cache memories of multi-core CPUs, although we expect that higher level caches will be increasingly more distant to access in the next generations.
- GPUs are especially suitable for tasks with a lot of data parallelism. A feature that can be useful for some applications is that caches can be programmed to work for 2D or 3D spatial locality. Synchronization across thread blocks (which should be very limited) is usually performed by allowing the kernel to finish and launching a new kernel. The overhead for a kernel launch is about 10-15 microseconds, depending on the number of blocks and the size of parameters.
- With the hardware platform market in major turbulence and developments following each other rapidly, the advantage of general-purpose flexibility, availability, cost, and ease of programming of MC-CPU should not be underestimated.

- With the ongoing trend that computation rate grows faster than memory and I/O bandwidth, more and more HPC applications will become data-intensive. With unfortunately designed applications, all platforms may be a bad choice, unless different algorithms are developed to solve the same problem more efficiently.

Below we give a list of guidelines to assist porting and optimization efforts for data-intensive applications, which is also useful for algorithm design.

- Optimize first for maximum effective memory bandwidth, instead of maximum compute rate. Data (structures) must often be re-outlined (for example, tiled, or arrays of structures converted to separate arrays per member (type)) to make memory accesses contiguous and aligned. Such transformations can force major code overhauls, which is often unavoidable for data-intensive applications. It is the only way to ensure that hardware can apply request optimizations, such as hardware-initiated prefetching, request reordering and coalescing, which has a major effect on performance.
- Multi-cores only perform well on applications with a high arithmetic intensity. Hence, only by effectively using local memories, data-intensive applications can perform truly well.
- Irregular access patterns must be resolved as much as possible, usually at a higher abstraction level. In our case, both the MC-CPU and Cell/B.E. profited from job queuing and smart filling. A cache can take care of misalignment.
- If an application kernel contains more than a couple of computations, overlap of memory transfers with computation must be implemented, as applying in-core optimizations on top of compiler optimizations is often more work than is justified by the pay-off.

Finally, Table 6 presents an efficiency overview of the behaviour of data-intensive applications on the three platforms. Based on all our findings, we conclude that there is no “best-platform” (already) available as “the best multi-core”, and it is still hard to choose a winner even for a specific applications class.

Table 6: An overview of the tested platforms

Platform	Perf.	Scal.	Cost	Effort	Efficiency
MC-CPU	~	+	++	+	~
Cell/B.E.	++	++	-	-	+
GPU	+		-	~	~

6. RELATED WORK

In this section we present a brief overview of previous work related to significant multi-core case-studies and the efforts we are aware of in comparing multi-core platforms.

In the past two years, all available multi-core architectures have been studied for application porting and optimizations. While MC-CPU have been mainly targeted by user-level applications, there have been some interesting case-studies that link them to HPC as well [7, 4]. The results indicate that previous expertise in SMP machines is very useful. However, developments in the direction of many dozens

of cores per processor will generate new problems with regard to main memory contention [20], cache hierarchy, and task-parallelism granularity. Therefore, a systematic way to program these platforms at a higher level of abstraction is mandatory for dealing with their next generation. Therefore, most of the software and hardware vendors already provide HPC solutions for multi-core machines; see, for example, HP’s multi-core toolkit, Microsoft’s HPC Server, or Intel’s Thread Building Blocks model. Still, most of these models are very new and unable to make, suggest or estimate the gain of major algorithmic transformations (data outline) needed for data-intensive applications. As a result, most programmers still rely on MPI and basic multi-threading for HPC, making our study a useful information source.

On the Cell/B.E. side, applications like RAXML [3] and Sweep3D [22] have revealed the challenges 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 is usually application specific. Therefore, such ad-hoc solutions can only be used as inspiration for the class of data-intensive applications. Typical ports on the Cell/B.E., like MarCell [17], or real-time ray tracing [2] are very compute-intensive, so they do not exhibit the unpredictable data access patterns and the low arithmetic intensity 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. Another option is to use one of the many programming models available for the Cell/B.E., but most of them are also focused on compute-intensive applications, and do not address the memory hierarchy properly. A notable exception is Sequoia [13], which focuses on the memory system performance, but it is hard to use for irregular parallel applications. Although RapidMind [19], a model based on data parallelism, provides back-ends for all three platform types, there are no performance evaluations and guidelines for exposing the required (low-level) data parallelism from a data-intensive application. Therefore, we consider a comparison of the results generated by these abstract models with our hand-tuned application a good direction of future work, which would help prepare these programming models to tackle data-intensive applications more efficiently. Note that both these models are fairly portable for all three platforms studied here.

In the case of GPU programming, the NVIDIA CUDA model is widely used, replacing the somewhat more cumbersome “traditional” graphics programming. As a result, there are many studies of HPC applications running on GPUs [23, 28, 16]. Most of these studies aim for a decent kernel speedup or conclude, as well as we do, that a lot of effort is required to put the application in the right form for being properly exploited with CUDA. These transformations are application specific. Furthermore, our study is the first to expose the additional difficulties that irregular data-intensive applications pose to GPUs. We also note the recent release of the OpenCL 1.0 [14] specifications, a standard for general purpose parallel programming on heterogeneous systems, with support from all major CPU, GPU, embedded system and gaming vendors.

An interesting overview that covers some of the memory-related problems we have encountered when optimizing a

data-intensive application can be found in [12]. This extensive study treats properties and trade-offs of caches, main memory, NUMA architectures, access patterns and what programmers can do to optimize for them. Another data-intensive application from radio astronomy, correlation, has very recently been compared on five multi-core architectures in [25]. It concludes that only architectures with a high bandwidth per Flop or enough local memory to reuse a lot of loaded data perform correlation computations efficiently. It also advocates for application-controlled local memory and notes that I/O must scale with the increase in cores, underlining the PCI-e bus I/O bottleneck for GPUs. Several similar application-oriented comparative studies have been developed to investigate bottlenecks and optimizations for multiple multi-cores [11, 29]. Finally, we note the introduction of the Roofline model [31] to provide visual insight into upper performance bounds of an application in order to select the most promising optimization or (multi-core) platform.

7. CONCLUSIONS

In this paper we have shown our experiences when implementing a data-intensive kernel on three multi-core platforms. When a data-intensive HPC application, possibly with irregular access patterns, has to be implemented, the biggest challenges are choosing the right platform and fitting the application to it. Each platform is not only different in its hardware architecture, but also in programming model. Although at a higher abstraction level similarities in optimization strategies can be found for all platforms, the performance gains are different, as are the techniques and effort to get there. This results in a severe lack of portability, which makes it prohibitively expensive to extensively evaluate alternatives.

We have found no “best overall platform”, as the choice depends too much on the specific application, programming experience, available design/implementation time, and funds. Therefore, we have included a list of guidelines to help making an educated choice. Similarly, we have presented guidelines for application porting and optimization on these multi-cores, focusing on memory and data optimizations. But, if algorithm parameters like memory access irregularity and low arithmetic intensity cannot be improved, the algorithm is simply multi-core unfriendly. Ultimately, hardware developments continue to increase performance conditionally, and the penalties for misbehavior are getting steeper. Only if software can be made to fit, HPC can continue to profit from next-generation multi-cores.

ACKNOWLEDGEMENTS. We would like to thank NVIDIA, and especially Dr. David Luebke, for donating some of the GPU cards used in this work. We would also like to thank IBM Research for providing us remote access to QS2x blades. Last, but not least, we would like to thank Tim Cornwell and Bruce Elmegreen for their help in the astronomy field.

8. 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):720–740, 2007.

- [2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the Cell processor. In *IEEE Symposium of Interactive Ray Tracing*, pages 15–23. IEEE Computer Society Press, September 2006.
- [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, March 2007.
- [4] A. Bode. High performance computing in the multi-core area. *International Symposium on Parallel and Distributed Computing*, pages 4–4, July 2007.
- [5] D. S. Briggs, F. R. Schwab, and R. A. Sramek. Imaging. In *Synthesis Imaging in Radio Astronomy II*, volume 180 of *Astronomical Society of the Pacific Conference Series*, pages 127–140, 1999.
- [6] 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 Lab., University of Tennessee, Knoxville, April 2007.
- [7] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Computer Architecture News*, 33(4):80–91, 2005.
- [8] 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*, volume 347, pages 86–95. ASP Press, October 2004.
- [9] T. J. Cornwell. SKA and EVLA computing costs for wide field imaging. *Experimental Astronomy*, 17:329–343, June 2004.
- [10] T. J. Cornwell and R. A. Perley. Radio-interferometric imaging of very large fields - The problem of non-coplanar arrays. *Astronomy and Astrophysics*, 261:353–364, July 1992.
- [11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, November 2008.
- [12] U. Drepper. *What Every Programmer Should Know About Memory*, November 2007.
- [13] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM Press, November 2006.
- [14] Khronos OpenCL Working Group. *OpenCL 1.0 Standard*, December 2008.
- [15] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May 2006.
- [16] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.
- [17] 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*, pages 1866–1869, July 2007.
- [18] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Molecular dynamics simulations on commodity GPUs with CUDA. In *High Performance Computing*, pages 185–196, December 2007.
- [19] M. McCool. Signal processing and general-purpose computing on GPUs. *IEEE Signal Processing Magazine*, pages 109–114, May 2007.
- [20] S. K. Moore. Multicore is bad news for supercomputers. *IEEE Spectrum*, November 2008.
- [21] NVIDIA. *CUDA Programming Guide*, December 2008.
- [22] F. Petrini, J. Fernández, 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*. IEEE Press, March 2007.
- [23] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, B. P. Sutton, and Z.-P. Liang. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.
- [24] K. van der Schaaf, C. Broekema, G. Diepen, and E. Meijeren. The LOFAR central processing facility architecture. *Experimental Astronomy*, 17(1-3):43–58, June 2004.
- [25] R. V. van Nieuwpoort and J. W. Romein. Using many-core hardware to correlate radio astronomy signals. In *23rd ACM International Conference on Supercomputing*, June 2009, to appear.
- [26] 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.
- [27] 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.
- [28] R. Wayth, K. Dale, L. Greenhill, D. Mitchell, S. Ord, and H. Pfister. Real-time calibration and imaging for the MWA (poster). In *AstroGPU*, November 2007.
- [29] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. A. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *Symposium on Parallel and Distributed Processing*, pages 1–14, April 2008.
- [30] 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*, pages 9–20. ACM Press, May 2006.
- [31] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM (CACM)*, April 2009, to appear.