J Supercomput (2012) 62:787–803
DOI 10.1007/s11227-012-0749-y

Stencil computations on heterogeneous platforms for the Jacobi method: GPUs versus Cell BE

José M. Cecilia · José L. Abellán · Juan Fernández · Manuel E. Acacio · José M. García · Manuel Ujaldón

Published online: 15 February 2012

Abstract We are witnessing the consolidation of the heterogeneous computing in parallel computing with architectures such as Cell Broadband Engine (Cell BE) or Graphics Processing Units (GPUs) which are present in a myriad of developments for high performance computing. These platforms provide a Software Development Kit (SDK) to maximize performance at the expense of dealing with complex and low-level architectural details which makes the software development a daunting task. This paper explores stencil computations in several heterogeneous programming models like Cell SDK, CellSs, ALF and CUDA to optimize the Jacobi method for solving Laplace's differential equation. We describe the programming techniques to extract the maximum performance on the Cell BE and the GPU, and compare their computing paradigms. Experimental results are shown on two Nvidia Teslas and one

J.M. Cecilia (✉)
Dept. of Computer Science, Catholic University of Murcia, Murcia, Spain
e-mail: jmcecilia@ucam.edu

J.L. Abellán · M.E. Acacio · J.M. García
Dept. of Computer Engineering, University of Murcia, Murcia, Spain

J.L. Abellán
e-mail: jlabellan@ditec.um.es

M.E. Acacio
e-mail: meacacio@ditec.um.es

J.M. García
e-mail: jmgarcia@ditec.um.es

J. Fernández
Intel Barcelona Research Center, Intel Labs, Universitat Politècnica de Catalunya, Barcelona, Spain
e-mail: juan.fernandez@intel.com

M. Ujaldón
Computer Architecture Department, University of Malaga, Malaga, Spain
e-mail: ujaldon@uma.es

IBM BladeCenter QS20 blade which incorporates two 3.2 GHz Cell BEs v 5.1. The speed-up factor for our set of GPU optimizations reaches 3–4×, and the execution times defeat those of the Cell BE by an order of magnitude, also showing great scalability when moving towards newer GPU generations and/or more demanding problem sizes.

Keywords Hardware accelerators · GPGPU · CELL · Stencil computations

1 Introduction

Heterogeneous computing (or hybrid computing) is being consolidated in the landscape of high performance computing. Heterogeneous systems refer to hardware platforms using a variety of different types of computational units. A computational unit can be a general-purpose processor (CPU), a special-purpose processor (i.e. SPEs, GPUs or even DSPs)), or custom acceleration logic (i.e. field-programmable gate arrays or FPGAs). The demand for increased heterogeneity in computing systems is mainly due to physical reasons (memory-wall and power-wall [3]) and the need for high-performance, highly-reactive systems that interact with other environments like audio/video, control or network applications.

Heterogeneous architectures, including Cell BE and GPUs, have been successfully used in high performance computing. The Cell BE architecture [13] is an heterogeneous multi-core chip composed of a general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized co-processors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and the I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB). Besides, the newest versions of programmable GPUs deliver extremely high floating point performance for scientific applications which fit their architectural idiosyncrasies [20]. They contain up to 512 streaming processors (SPs) which are organized into groups of 8 SPs, namely Streaming Multiprocessors (SMs). Each GPU contains its own high bandwidth off-chip memory (video memory of several Gigabytes of GDDR5 DRAM).

Major vendors and research groups have released software components which provide a simpler way to handle parallelism in heterogeneous computing. CUDA (Compute Unified Device Architecture) [15] is Nvidia's solution as a simple block-based API for GPU programming. Examples on the Cell side are ALF, CellSs, Sequoia and MicroMPI [17]. All of these models are expected to converge in OpenCL [22] as a higher level standard shared by a wide set of heterogeneous systems.

Stencil computations are those in which each computing node in a multi-dimensional grid is updated with weighted values contributed by neighboring nodes. These neighbors comprise the stencil, and multiple iterations across the array are usually required to achieve convergence or to simulate time steps. Among those stencil codes, our work focuses on the Jacobi method to solve Laplace's differential equation. We address the design space on Cell BE and GPUs. We analyze the main features of stencil computations and also their corresponding architectural idiosyncrasies, by using the Jacobi method as a case study. A set of optimization paths are

also explored, trying to illustrate the strength of CUDA for accelerating those stencil computations.

The rest of the paper is organized as follows. Section 2 briefly introduces the Cell BE and the CUDA architectures together with their programming models. Section 3 describes the Jacobi method representing stencil computations and compares its parallelization on the Cell BE and the GPU. Section 4 outlines optimizations, Sect. 5 evaluates performance, and finally, Sects. 6 and 7 describe some related work and conclude, also pointing some directions for future work.

2 Hardware platforms and programming models

2.1 The cell broadband engine (Cell BE)

The PPE is the main processor of the Cell BE, responsible for running the operating system and coordinating the SPEs. It is a traditional 64-bit PowerPC (PPC) processor core with a VMX unit (*Vector/SIMD Multimedia Extension*), 32 KByte L1 instructions and data caches, and a 512 KByte L2 unified cache. The PPE is a dual issue, in-order execution, 2-way SMT processor, containing two different units called *PowerPC Processor Unit* (PPU) and *PowerPC Processor Storage Subsystem* (PPSS).

Each SPE is a 128-bit RISC processor which consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC). The SPUs are in-order processors with two pipelines and 128 registers of 128 bits. All SPU instructions are inherently SIMD operations that the proper pipeline can run at four different granularities: 16-way 8-bit integers, 8-way 16-bit integers, 4-way 32-bit integers or single-precision floating-point numbers, or 2-way 64-bit double-precision. SPEs do not have a private cache memory, but a 256 KByte LS memory to hold instructions and data, and SPU programs cannot access main memory directly. The MFC contains a *DMA Controller* and a set of memory-mapped registers called *MMIO Registers*. Each SPU can write its MMIO registers though several *Channel Commands*. The DMA controller supports DMA transfers among the LSs and main memory. These operations can be issued by the owner SPE, which accesses the MFC through the channel commands, or the other SPEs (or even the PPE), which access the MFC through the MMIO registers. In addition, the PPE and the SPEs can use a variety of hardware-supported mechanisms like *Mailboxes*.

Cell BE programming requires separate programs, written in C/C++, for the PPE and the SPEs, respectively [10]. The PPE program can include extensions (e.g., `vec_add`), to use its VMX unit; and library function calls [11], to manage threads and perform communication and synchronization operations. (e.g., `spe_context_run`, `spe_in_mbox_write` and `spe_signal_write`).

2.2 Compute unified device architecture (CUDA)

CUDA comprises a programming model and a hardware architecture for the GPU, making it operate as a highly parallel computing device. Each GPU consists of a set of SIMT (Single Instruction Multiple Threads) multiprocessors (SM), each of

them containing Stream Processors (SPs) [15]. Different memory spaces are available on each GPU: The global (or device) memory is the only space accessible by all multiprocessors, and each multiprocessor has its own private memory space called shared memory.

A CUDA parallel program has two parts: a sequential code executed by the CPU (host), and a parallel code (kernel) executed by the GPU. The host code transfers data between main memory and global memory via PCI-express, and also sets up the kernel parameters for each GPU (like the number of blocks per grid and the number of threads per block).

The device code is grouped into one or more program routines called kernels, and they are called from the host code as if they were procedures or objects in C/C++. A kernel is a piece of code programmed in a SPMD (Single Program, Multiple Data) style, that is, the same code is executed over different data by distinct threads on different SP cores. The kernel computation is performed by all these threads running in parallel.

The CUDA execution model [19] is based on a hierarchy of abstraction layers: grids, blocks, warps and threads. The thread is the basic execution unit that is mapped to a single SP. A block is a batch of threads which can cooperate together because they are assigned to the same multiprocessor, and therefore they share all the resources included in this multiprocessor, such as register file and shared memory. A grid is composed of several blocks which are equally distributed and scheduled among all multiprocessors. Finally, threads included in a block are divided into batches of 32 threads called warps.

The programmer declares the number of blocks and threads per block. Each block and thread has its own and unique identifier (thread id and block id), which allow the programmer to select different data and code depending on them.

Memory accesses and synchronization schemes also play important roles in the CUDA programming model. Memory latency can be greatly reduced if the memory access follows the correct pattern [18]. Global synchronization is not provided at the device side: only threads within a block are synchronized, and therefore block synchronization mechanisms must be explicitly implemented by the host through consecutive kernel invocations or via atomic instructions.

3 Stencil computations for the Jacobi method

Jacobi [14] is a popular algorithm for solving Laplace's differential equation on a square domain, regularly discretized [6]. The kernel (see Fig. 1) is based on the following idea: Let us consider a body represented by a 2D array of particles, each with an initial value of temperature. This body is in contact with a fixed value of temperature on the four boundaries, and Laplace's equation is solved for all internal points to determine their temperature as the average of the four neighboring particles. Taking this task as the computational core, a number of iterations are performed over the data to recompute average temperatures repeatedly, and the values gradually converge to a finer solution until the desired accuracy is reached.

```
for (k=0; k<4096; k++) {
  for (i=0; i<N; i++)
    for (j=0; j<N j++)
      M[i][j] = 0.2*(M'[i][j]+M'[i-1][j]+
             M'[i+1][j]+M'[i][j-1]+M'[i][j+1]);
  for (i=0; i<N; i++)
    for (j=0; j<N j++)
      M'[i][j] = M[i][j]; }
```

Fig. 1 Jacobi's solver pseudocode

Note that iterations have to be serialized due to carried-loop dependencies, but parallelism is enabled within iterations for the computation as each particle is independent. Thus, the workload depends more on the number of iterations, whereas the amount of parallelism that can be extracted from the code relies more on the size of the 2D input matrix. At the end, our Jacobi kernel consists of three nested loops, with the two innermost being of length N (which is the matrix dimension), and the outermost being of length k (the number of iterations) (see Fig. 1). The algorithm complexity can be expressed as $O(k \cdot N^2)$.

For experimental purposes, we consider a constant number $T = 4096$ iterations to compute an input square 2D-matrix of single-precision floating-point elements, although performance is largely independent of the number of iterations T. The Cell BE and GPU architectures present distinct properties to face an optimal parallelization of stencil computations based on the Jacobi method:

- *Intensive memory access.* To update each computing node in a multi-dimensional grid, weighted values contributed by neighboring nodes are employed and have to be read from memory. When comparing the amount of data read from and written to memory with the time spent on calculation, memory access becomes a potential bottleneck. The Cell BE and GPUs behave differently here due to limited explicitly-managed on-chip memories. Those memories are handled using tiling, coalescing and double-buffering techniques in order to alleviate memory pressure on each architecture.
- *Massive parallelism.* Stencil computations are allowed to use a large number of threads during parallelization. Depending on the architecture features, the pool of threads can be increased or decreased to obtain the maximum performance, by setting a thread per element in the multi-dimensional grid or a thread per data set. A priori, a large number of independent threads has more impact on the GPU, where computing model is highly multi-threaded and the number of processors is much higher.
- *Data locality.* Adjacent nodes are needed for computing each node. This allows for threads executed by the same SIMD unit to address contiguous memory blocks throughout the node computation. This influences equally Cell and GPUs, but the key issue now is to take advantage of the fastest levels in the memory hierarchy.
- *Synchronization requirements.* Multiple iterations across the array are usually required to achieve convergence or to simulate time steps on stencil computations. Therefore, a synchronization among all processing elements is required to update frontier values. The Cell BE is better suited to improve this issue through a barrier

synchronization; GPUs, on the other hand, have to finalize the kernel to get global synchronization among all streaming processors.
– *Low floating-point arithmetic intensity.* For the particular case of the Jacobi method, only five operations are needed at each computing node (see the Jacobi's solver pseudocode of Fig. 1). Even though the instruction set implemented by GPUs is more restricted than that of the Cell BE, the simplicity of the kernel allows a draw in this concern. Anyway, the Jacobi method does not constitute an ideal partner for any of these platforms, where a high arithmetic intensity becomes an essential issue for attaining high performance computing.

4 Application design

In this section, we discuss several CPU and Cell BE implementations of the Jacobi solver. We propose different computational patterns, introduce two different programming models for the Cell BE, and describe several GPU techniques that are potentially useful to increase the data bandwidth during the Jacobi computation.

4.1 Jacobi solver implementation on Cell BE

Figure 2 shows the parallelization of the Jacobi solver by using the Cell SDK. First, the PPE allocates matrices A and B in main memory (see number 1). Then, the PPE statically assigns a set of columns to each available SPE by dividing the number of columns by the number of available SPEs. This exploits SIMD parallelism, and then, columns per SPE are logically divided into chunks. For instance, Fig. 2 shows a column set equivalent to three chunks: SET1, SET2 and SET3 (see number 2). These SETs determine the number of chunks to be computed.

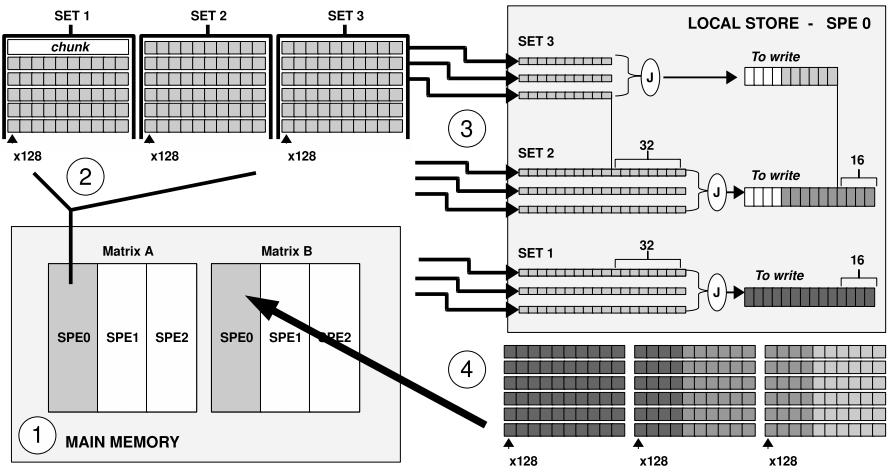

Fig. 2 Jacobi implementation using the SDK. Arrows indicate data transfers, and the computational order is determined by numbers

The PPE synchronizes with all SPEs through mailbox operations in order to initialize the computation. Then, each SPE transfers to its *local store* its corresponding chunks from matrix A (see number 3), with the order of the transfers per SET going from the top to the bottom chunk. Figure 2 shows groups of three chunks in *local store*, with the top-adjacent and the bottom-adjacent chunks being required to process each chunk.

Eventually, the left-adjacent and/or right-adjacent chunks may be transferred as 128-bytes aligned subsets of these chunks to enhance performance (see [10]). For example, in Fig. 2 they are transferred for chunks belonging to SET2, which introduces an extra overhead particularly in large matrices.

Finally, all SPEs synchronize in a barrier which takes just 650 ns for a 16-SPE synchronization, similarly as the overhead in [8], and then matrix pointers are swapped and the *stop-condition* is verified. In our particular case, the SPE0 checks the *stop condition* and sends an acknowledgment or exit message through mailboxes, accordingly.

4.1.1 Increasing the memory bandwidth

The Jacobi method on Cell BE is designed to reduce memory transfers between main memory and local storage of SPEs, and thus to harness the Cell BE computational resources to the maximum. In this sense, it avoids transferring the left elements when processing the first element of a particular chunk, leaving untouched the first four elements (see the first white-colored elements in chunks *To write*, except those chunks belonging to SET1). Instead, it computes the next four elements from the right-adjacent chunk (except for chunks belonging to SET3).

As a result of the computation (see ellipse *J*), the chunks *To write* contain the additional 4 elements (16 bytes) from the right-adjacent chunk. These chunks are written to main memory by computing SET3, SET2 and SET1 in reverse order, to avoid overwriting the first four elements of those chunks. As a result of the computation, number 4 shows the final SETs which are written to matrix B.

All transfers between main memory and local storage of SPEs have been accomplished by using a double-buffering technique in order to overlap computation with communications. In particular, we use double-buffering for (1) the transfers per SET going from the top chunk to the bottom chunk, and (2) the final SETs which are written to matrix B. 1024-bytes chunks were selected as optimal size for transfers according to statistics reported in [1] in order to efficiently exploit latencies and bandwidths on Cell.

4.1.2 Alternative Cell BE high-level programming models

A number of programming models have arisen as an attempt to reduce the high complexity of programming provided by the flexible SDK. In this way, CellSs or ALF programming models make transparent some regular operations such as task scheduling or data-transfers management to programmers, at the cost of degrading performance in most cases.

In CellSs, programmers declare parallel tasks (or functions) which the runtime will attempt to execute in parallel on available SPEs. To do so, we identify two main

```
#pragma css start                                          1
                                                           2
for (k=0;k<4096;k++){ {                                    3
                                                           4
// Read Mold matrix and write Mnew matrix                  5
for (i=0;i<ROWS;i++){                                      6
                                                           7
for (j=0;j<COLUMNS;j+=sizeof_chunk)                        8
{                                                          9
Jacobi(&Mold[i][j],                                        10
&Mold[i+1][j],                                             11
&Mold[i+2][j],                                             12
&Mnew[i+1][j]);                                            13
}                                                          14
}                                                          15
                                                           16
// Barrier                                                 17
#pragma css barrier                                        18
// Interchange pointers to matrices                        19
tmp=Mold;                                                  20
Mold=Mnew;                                                 21
Mnew=tmp;                                                  22
}                                                          23
                                                           24
#pragma css finish                                         25
```

Fig. 3 Jacobi solver on CellSs

tasks to solve the Jacobi method: *Jacobi* and *Stop_Condition*, respectively. The former receives as input three specific chunks (or sets of consecutive matrix elements) from the matrix A (top, center and bottom chunks), processes them according to the Jacobi algorithm shown in Fig. 1, and, finally, stores the output-parameter chunk into matrix T. The latter checks whether *stop condition* has been satisfied or not.

Figure 3 outlines our CellSs algorithm. The implementation for the function tasks and the special cases for computing frontier elements have been omitted for simplicity. On each inner iteration, the *Mold* matrix is read in three chunks (top, center and bottom) from three consecutive rows (lines 10–12). The center chunk is computed (line 10) and written to the corresponding row of matrix *Mnew* (line 13). This process is repeated until all the elements in the matrix *Mnew* have been computed. Upon completion, a barrier is performed (line 18) to check the *stop condition* (line 21). Finally, pointers to matrices are swapped prior to starting the next iteration (lines 24–26).

Figure 4 shows the ALF version of the Jacobi method. This version is based on the host-partition scheme presented in [9]. We assume the reader is familiar with ALF particularities, thus the code has been highly simplified. ALF programming model specifies tasks, work blocks or DTLs (see Sect. 2.1). Similarly to the CellSs version, the ALF version performs two task based on the Jacobi algorithm. The SPE code has been omitted because it is equivalent to the *Jacobi* function in CellSs. A task context buffer is used by applications that require common persistent data that can be referenced and updated by all work blocks. In this way, upon completion the execution of each work block, a new partial summation updates a local copy of the task context buffer. At the end, the ALF runtime merges all of those local copies and assesses the final result.

```
for (k=0;k<4096;k++){ {
// Read Mold matrix and write Mnew matrix
for (i=0;i<ROWS;i++){
for (j=0;j<COLUMNS;j+=sizeof_chunk) {
// Create a new work block
alf_wb_create(task_handle,ALF_WB_SINGLE,&wb_handle);
// DTL elements for input chunks from Mold matrix
alf_wb_dtl_begin(wb_handle, ALF_BUF_IN, 0);
alf_wb_dtl_entry_add(wb_handle, &Mold[i*n + j]);
alf_wb_dtl_entry_add(wb_handle, &Mold[(i+1)*n + j]);
alf_wb_dtl_entry_add(wb_handle, &Mold[(i+2)*n + j]);
alf_wb_dtl_end(wb_handle);
// DTL elements for output chunk into Mnew matrix
alf_wb_dtl_begin(wb_handle, ALF_BUF_OUT, 0);
alf_wb_dtl_entry_add(wb_handle, &Mnew[(i+1)*n + j]);
alf_wb_dtl_end(wb_handle);
// Enqueue a new work block
alf_wb_enqueue(wb_handle);
}
}
alf_task_finalize(task_handle);
// Barrier
alf_task_wait(task_handle, -1);

// Swap pointers to matrices
tmp=Mold;
Mold=Mnew;
Mnew=tmp;
}
```

Fig. 4 Jacobi solver on ALF

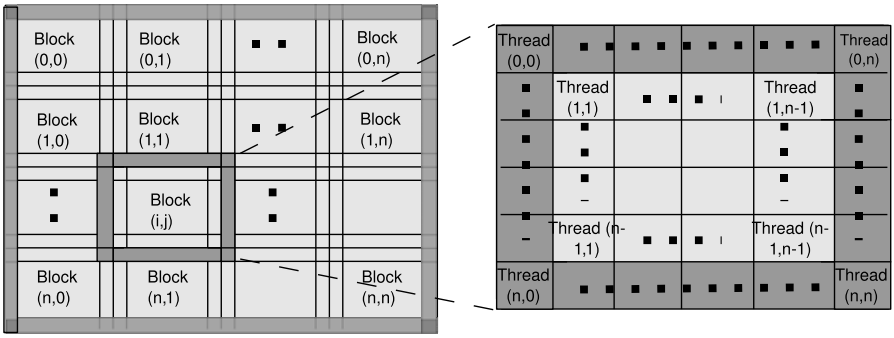

Fig. 5 Threads deployment for the CUDA parallelization strategy

4.2 Jacobi solver implementation on CUDA

Figure 5 shows the threads deployment for the parallelization of the Jacobi method using CUDA. Blocks and threads are deployed following a 2D layout to balance the decomposition of the computational domain on each matrix dimension. Adjacent blocks share data placed on boundaries, and each thread within a block is responsible for updating a single element on each iteration.

Among all possibilities concerning an input matrix of size $N \times N$ and a squared block of $B \times B$ threads, we have selected $N = 1024, 2048, 4096, 8192$ and $B = 14, 16, 18, 20$ for representing good choices after a preliminary survey.

4.2.1 Shared memory optimizations

Our CUDA baseline implementation does not use shared memory. All threads access the device memory to read an element together with its four matrix neighbors and later update its value with the average. From this departure point, three optimizations were incrementally developed:

1. Each input element read from device memory is stored into shared memory by the owner thread prior to the actual computation, and the output result is written back into device memory. The kernel length increases from 34 to 78 instructions, but this variant notably reduces the pressure on device memory, just requiring 18 GB/s of memory bandwidth compared to 122 GB/s in our baseline version.

   On the Tesla C870, 99.68% of the memory accesses to device memory are non-coalesced when running the code using CUDA Compute Capabilities 1.0 (CCC 1.0). On the Tesla C1060, things are very different, for this device uses coalescing rules based on CCC 1.3, leading to a 100% of coalesced accesses. Benefits are therefore larger on the Tesla C1060 GPU.
2. Our second optimization uses an internal register as substitute of the shared memory cell on each thread, eliminates unnecessary synchronization barriers between threads at block level, and enables data prefetching. These enhancements behave similarly on CCC 1.0 and 1.3, and are translated into minor improvements in the overall execution time.
3. The third optimization reduces the tile size to decrease the use of shared memory. In CCC 1.0, the maximum number of threads assigned to a multiprocessor is 768, whereas in CCC 1.3 this number reaches 1024. In the first case, the tile size is decreased to reduce the amount of shared memory used (4120 bytes) so that we can assign three blocks of 256 threads to each multiprocessor. In the second case, the tile size is reduced even more until we can assign four blocks of 256 threads, which increases parallelism leading to slightly better results. Insights of different tiling strategies for stencil codes can be found in [21].

4.2.2 The effect of larger 2D stencils

Our next alternative kernel tries to evaluate the effect of changing the 2D stencil size, which imposes a coarser granularity on SIMD parallelism. Instead of a single element, a 2×2 matrix of elements was assigned to every thread. Using this new stencil, partial sums on diagonal elements of the matrix can be reused for computing the output elements on the other diagonal (see Fig. 6), saving two arithmetic operations and four memory accesses on each thread at the expense of using two registers for storing auxiliary values. However, other optimization parameters may be affected, such as the memory granularity; the coalesced access to device memory could be compromised by not using double buffering between shared and device memory. Therefore, there is a trade-off between the number of operations performed and having a homogeneous memory access.

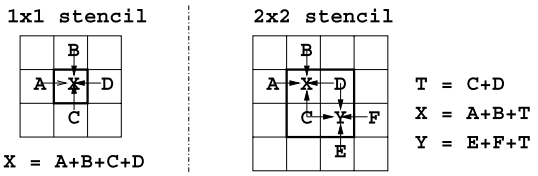

Fig. 6 Benefits of increasing the stencil size: some redundant operations may be saved

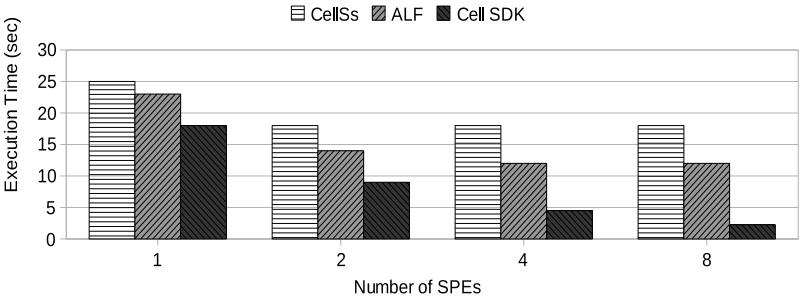

Fig. 7 Execution time (secs.) on the Cell BE for a 2048 × 2048 input matrix depending on the programming model: Cell SDK, CellSs and ALF

5 Performance evaluation

The Cell BE architecture was managed using the IBM SDK v3.0 and Fedora Core 7 installed on a regular PC acting as host. This development kit includes a simulator, named Mambo, allowing programmers to execute binary files compiled for the Cell BE architecture. Experimental results were measured on a Cell-based IBM BladeCenter QS20 blade which incorporates two 3.2 GHz Cell BEs v 5.1, namely Cell0 and Cell1, endowed with 1 GByte of main memory.

Our first GPU-based platform is dual-socket, Intel Core 2 duo E6850 3 GHz, which acts as a host machine for our Tesla C870. Our second GPU-based platform incorporates a four-socket, quad-core Intel Xeon E5530, acting as host for our four Nvidia Tesla C1060 GPUs. Host PCs run under Ubuntu 10.04, and the NVIDIA CUDA SDK and compilation toolkit, release 2.3.

5.1 Cell BE platform

Figure 7 shows the elapsed time (in seconds) achieved by varying the programming models on the Cell BE (up to 8 SPEs are used to illustrate performance trends). They emphasize the performance degradation by using high-level programming languages like CellSs or ALF, compared to a low-level programming with the Cell SDK, which improves elapsed times of ALF and CellSs versions with linear scalability. High-level alternatives need to reduce the overhead when running fine-grain applications like our Jacobi solver. In those cases, an additional effort for programming the hardware at SDK level becomes quite rewarding.

Table 1 Execution times (in seconds) for our Jacobi baseline implementation when varying the CUDA block size

Matrix size	(Threads deployment per CUDA block)			
	(14 × 14)	(16 × 16)	(18 × 18)	(20 × 20)
(a) Tesla C870 GPU				
1024^2	13.50	13.16	13.70	14.13
2048^2	52.73	50.74	52.57	52.43
4096^2	206.99	203.35	207.06	211.28
8192^2	843.55	850.18	899.46	852.26
(b) Tesla C1060 GPU				
1024^2	3.27	2.34	3.24	3.071
2048^2	12.73	8.72	11.88	11.594
4096^2	50.36	34.60	46.28	44.402
8192^2	211.03	144.02	211.16	177.795

5.2 GPU platform

We start with Table 1, where the optimal threads deployment for the Jacobi baseline implementation is analyzed on either Tesla C870 (left) or Tesla C1060 (right). We see that 16 × 16 constitutes the optimal layout and number of threads per block, with a penalty around 5–10% for the top 3 remaining cases (those sizes not shown where tested, but led to worse results). In addition, a comparison between GPU platforms allows us to establish an improvement factor roughly between 4× and 5× on the Tesla C1060.

Our CUDA baseline implementation does not use shared memory. From this departure point, we gradually develop the three optimizations involving shared memory as described in Sect. 4.2.1. Table 2(a) shows the execution times for all these versions on a Tesla C870 and Table 2(b) does the same for the Tesla C1060 GPU, where an average speed-up factor of 3.5× is roughly attained. Between code versions, a remarkable improvement arises when enabling shared memory, and marginal gains are reported for the two remaining optimizations: coalescing and memory bank conflicts.

Execution times on larger tiles are shown in Table 3 on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and our kernel uses shared memory without further optimizations. Times slowdown 30–40% on average with respect to the case in which each thread computes a single element, proving that context switch is free in CUDA and the block startup (CUDA runtime overhead) is not: Using a 1 × 1 stencil we require 341 × 341 block calls, whereas using a 2 × 2 stencil, we just need 157 × 157 block calls. The lesson here is that coarser grain parallelism hurts performance on the GPU, and consequently, results on 3 × 3 or 4 × 4 tiles worsen and were not even tried.

5.3 Overall comparison

Figure 8 summarizes the performance of the best implementation on either the Cell SDK or CUDA for different matrix sizes: 1024 × 1024, 2048 × 2048, 4096 × 4096

Table 2 Execution times (in seconds) for our Jacobi implementation when using different optimizations. Between parenthesis, we show the speed-up factor versus the baseline implementation on the same platform. Threads deployment is 16×16 in all cases

Input matrix size	Baseline: no shared memory used	Optimization 1: using shared memory	Optimizations $1+2$: shared memory + coalescing	Optimizations $1+2+3$: also solving banks conflicts
(a) Tesla C870				
1024^2	13.16	3.77 (3.49×)	3.76 (3.50×)	3.88 (3.39×)
2048^2	50.74	14.49 (3.50×)	14.45 (3.51×)	14.71 (3.45×)
4096^2	203.35	55.60 (3.65×)	55.59 (3.65×)	57.45 (3.54×)
8192^2	850.18	243.00 (3.50×)	241.81 (3.51×)	241.81 (3.51×)
(b) Tesla C1060				
1024^2	2.34	0.73 (3.20×)	0.65 (3.60×)	0.63 (3.71×)
2048^2	8.72	2.79 (3.12×)	2.47 (3.53×)	2.42 (3.60×)
4096^2	34.60	11.45 (3.02×)	9.93 (3.48×)	9.66 (3.58×)
8192^2	144.02	45.70 (3.15×)	40.35 (3.57×)	40.29 (3.57×)

Table 3 Execution times (in seconds) on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and the code version uses shared memory without further optimizations. The stencil size is the number of elements computed by each thread

Threads deployment	Stencil size		Slowdown factor
	One	2×2	
14×14	13.91	19.31	38%
16×16	11.45	15.43	34%
18×18	13.27	18.16	36%
20×20	13.83	18.40	33%

and 8192×8192. These times only concern the parallel stage of the execution time (initial startup is discarded).

The Tesla C1060 outperforms Cell BE in all cases. In particular, the maximum speed-up is reached for $2K \times 2K$ matrices: 3.6× gain factor versus Tesla C870 and 10× versus Cell BE platform. Larger problem sizes do not increment this gain, mainly because of the global synchronization penalty that GPUs incur (see Fig. 9), which is closely associated with the number of threads involved in the execution. On a GPU, this number increases with the problem size, whereas it remains constant on the Cell BE leaving the synchronization overhead unchanged.

6 Related work

Along the years, researchers have traditionally investigated tiling/blocking as a consolidated way of improving cache locality and parallelism, with stencil computations

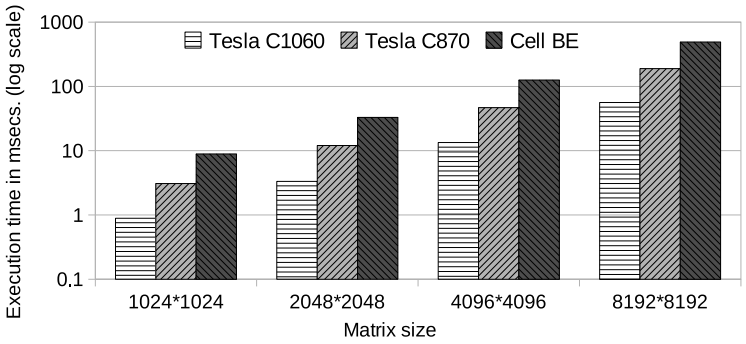

Fig. 8 Execution time for the two different programming models and architectures: Cell SDK on the Cell BE and CUDA on GPUs

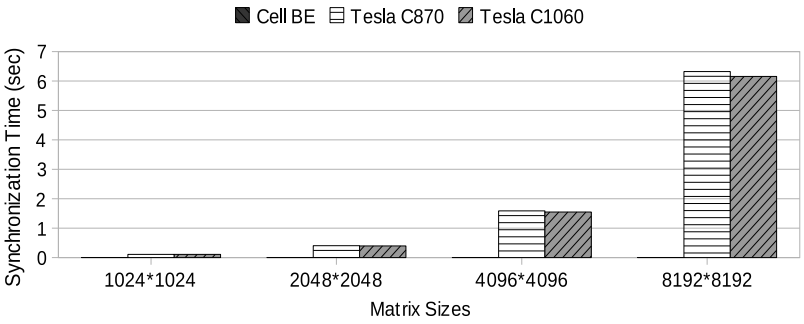

Fig. 9 Synchronization times for the Cell BE platform, Tesla C870 and Tesla C1060

considered as one of the most rewarding applications in this respect. Recent contributions are more ambitious and propose automatic tuning for tiling stencil computations. This trend includes developments on multicores [5, 21], the Cell BE [4], and ultimately the GPU [24].

As time goes by, stencil computations are being offloaded from the Cell BE and more GPU deployments gain attention in the heart of the scientist community. Listed in order of affinity with our work, we may select the following four contributions: Datta et al. [5] tune a benchmark of 3D stencil kernels on GPUs and multicores, Christen et al. [4] consider a 7-point stencil kernel to be implemented on GPUs and the Cell BE, Amorim et al. [2] perform a comparison of the Jacobi method between a GPU parallelization using OpenGL and CUDA, and finally, Venkatasubramanian et at. [24] implement the Jacobi method on GPUs and hybrid CPU/GPU systems. Other efforts on implementing stencil computations on GPUs we may highlight are provided by Fang et al. [7], who benchmark ATI/AMD GPUs using stencil codes and provide different techniques such as those proposed in this paper. Maruyama et al. [16] provide a compiler-based programming framework that automatically translates user-written structured grid code into scalable parallel implementation code for GPU-equipped clusters. Finally Unat et al. [23] give importance to stencil codes by providing a programming model that targets stencil methods.

Focusing on the work performed specifically on Jacobi method, Amorim et al. [2] use diagonal matrices and a different access pattern than ours to compare results against a CPU implementation on a quad-core AMD Phenom processor, obtaining a 78× speed-up factor. On the other hand, the work in [24] was developed in parallel to ours with a similar methodology. Our implementation sacrifices two idle threads on each half-warp to be rewarded on coalesced and conflicts-free accesses to memory banks, since memory bandwidth is more a bottleneck than the availability of computing cores within the GPU. Also, padding is more profitable in our coalesced case because it allows us to take advantage of remarkable improvements introduced in CUDA Compute Capabilities 1.3 when using the Tesla C1060 platform.

7 Conclusions and future work

We have explored the parallelization of stencil computations depending on different programming models and hardware platforms under the general framework of high performance computing. More in particular, critical aspects like arithmetic intensity or memory access patterns can be modeled under the umbrella of stencil computations, and we use the Jacobi method for solving Laplace's differential equation as a case study algorithm. Programming models include CellSs, ALF and Cell SDK on the IBM Cell BE architecture, and CUDA on Nvidia Tesla GPUs.

By studying our initial implementations, their inefficiencies, and the behavior of our optimizations, we have learnt several key insights about the underlying architectures and their programming models, namely:

- The main bottleneck on both architectures is the memory bandwidth. On the Cell SDK, we use 1024-byte chunks and double buffering technique to overlap communication and computation. On CUDA, data parallelism, threads deployment and memory hierarchy are primary issues we focus our analysis on, with great success for our optimization techniques scoring a consistent 3–4× factor improvement with respect to the baseline Jacobi code.
- The trade-off between programming effort, performance and scalability is analyzed on the Cell BE architecture. Programming with CellSs is the simplest way of developing parallel applications on the Cell BE, but it shows low performance and poor scalability with respect to our SDK implementation, particularly in scenarios with more SPEs and not enough coarse grain parallelism. Similar conclusions can be drawn from the ALF implementation, with slightly better results though. Our third candidate, the SDK version, reports good scalability at the expense of development complexity.
- An outstanding scalability is also reported when moving towards newer GPU generations: Gain factors are 4–5× when migrating the code from a Tesla C870 model (as of 2007, with 128 streaming processors) to the C1060 counterpart (the upgraded model as of 2009, extended to 240 streaming processors).

On a side-by-side comparison, GPUs show better performance than the Cell BE platform on a similar hardware complexity: The Nvidia Tesla C1060 defeats the IBM BladeCenter QS20 by an order of magnitude, also showing better behavior on more

demanding problem sizes and a promising future on incoming GPU generations. In general, streaming and arithmetic intensive kernels achieve higher performance on GPUs than Cell, but our kernel for Jacobi is bandwidth limited, preventing us from further optimizations. Synchronization costs were also seen higher on the GPU as a function of the problem size, whereas they remained constant on Cell.

Programming heterogeneous architectures represents a challenge for programmers, no matter the platform we choose to work with. We pretend to reduce our implementation effort in the future by adopting converging models foreseen in the horizon. Rapidmind was a corporate initiative pioneering this movement. The company was acquired by Intel in 2009 to end up with the Array Building Blocks (ArBB) [12] parallel programming model for multicore and many-core architectures. This effort was followed by OpenCL [22] as a joint industry-academia consortium seen as a flagship for hybrid computing. Our goal from now on goes to analyze the possibilities of OpenCL on a wide variety of high performance platforms, analyzing different computational patterns derived from the general framework of stencil computations. In this respect, the Jacobi kernel we present here is just a first step on a long road ahead us.

Acknowledgements This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under projects 00001/CS/2007, 15290/PI/2010 and under the fellowship 12461/FPI/09, by the Spanish MICINN and European Commission FEDER funds under projects Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04. We also thank NVIDIA for hardware donation under Professor Partnership 2008–2010 and CUDA Teaching Center Award 2011–2012.

References

1. Abellán JL, Fernández J, Acacio ME (2008) Characterizing the basic synchronization and communication operations in dual cell-based blades. In: International conference on computational science, Krakow, Poland.
2. Amorim R, Haase G, Liebmann M, Weber dos Santos R (2009) Comparing CUDA and OpenGL implementations for a Jacobi iteration. In: Smari WW (ed) Proceedings of the 2009 high performance computing & simulation conference (HPCS'09), IEEE, New Jersey. Logos Verlag, Berlin, pp 22–32
3. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. Tech rep UCB/EECS-2006-183, EECS Department, University of California, Berkeley
4. Christen M, Schenk O, Neufeld E, Messmer P, Burkhart H (2009) Parallel data-locality aware stencil computations on modern micro-architectures. In: Proceedings of the 2009 IEEE international symposium on parallel & distributed processing (IPDPS '09). IEEE Computer Society, Washington, pp 1–10
5. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing (SC '08). IEEE Press, Piscataway, pp 1–12
6. Demmel JW (1997) Applied numerical linear algebra. In: Society for industrial and applied mathematics. SIAM, Philadelphia
7. Fang X, Tang Y, Wang G, Tang T, Zhang Y (2010) Optimizing stencil application on multi-thread GPU architecture using stream programming model. In: Proceedings of 23rd international conference (ARCS), Hannover, Germany, pp 234–245
8. Gaona E, Fernández J, Acacio ME (2009) Fast and efficient synchronization and communication collective primitives for dual cell-based blades. In: Euro-Par, pp 900–911
9. Hill J (2007) Scientific programming on the cell using ALF. Tech rep, HPCx consortium
10. Systems IBM Technology Group (2007) Cell broadband engine programming tutorial version 2.1

11. IBM Systems and Technology Group (2007) SPE runtime management library version 2.1
12. Intel: Array building blocks (2012). http://software.intel.com/en-us/articles/intel-array-building-blocks/
13. Kahle J, Day M, Hofstee H, Johns C, Maeurer T, Shippy D (2005) Introduction to the cell multiprocessor. IBM J Res Dev 49(4/5):589–604
14. Lester BP (1993) The art of parallel programming. Prentice-Hall, Upper Saddle River
15. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) Nvidia tesla: a unified graphics and computing architecture. IEEE MICRO 28(2):39–55. http://doi.ieeecomputersociety.org/10.1109/MM.2008.31
16. Maruyama N, Nomura T, Sato K, Matsuoka S (2011) Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis (SC '11), New York, USA, pp 11:1–11:12
17. McCool MD (2008) Scalable programming models for massively multicore processors. IEEE MICRO 96(5):816–831
18. NVIDIA: (2008) NVIDIA CUDA programming guide 2.0
19. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) Gpu computing. Proc IEEE 96(5):879–899
20. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell T (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26(1):80–113
21. Renganarayana L, Harthikote-matha M, Dewri R, Rajopadhye S (2007) Towards optimal multi-level tiling for stencil computations. In Proceedings of 21st IEEE international parallel and distributed processing symposium (IPDPS), Long Beach, CA, USA
22. Stone JE, Gohara D, Shi G (2010) Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des Test Comput 12(3):66–73. http://dx.doi.org/10.1109/MCSE.2010.69
23. Unat D, Cai X, Baden SB (2011) Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the international conference on supercomputing (ICS '11). ACM, New York, pp 214–224
24. Venkatasubramanian S, Vuduc RW, None N (2009) Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In: Proceedings of the 23rd international conference on supercomputing (ICS '09). ACM, New York, pp 244–255