Multicore Platforms for Scientific Computing: Cell BE and NVIDIA Tesla

J. Fernández, M.E. Acacio, G. Bernabé, J.L. Abellán, J. Franco
Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia
Facultad de Informática, Campus de Espinardo s/n, 30100 Murcia (Spain)

Abstract - *There are two multicore platforms that are currently concentrating an enormous attention due to their tremendous potential in terms of sustained performance: the Cell Broadband Engine (Cell BE from now on) and the NVIDIA Tesla computing solutions. The former is a recent heterogeneous chip-multiprocessor (CMP) architecture jointly developed by IBM, Sony and Toshiba to offer very high performance, especially on game and multimedia applications. In fact, it is the heart of the PlayStation 3. The latter are general-purpose GPUs (GPGPU) used as data-parallel computing devices based on the Computed Unified Device Architecture (CUDA) common to the latest NVIDIA GPUs. The common denominator is a multicore platform which provides an enormous potential performance benefit driven by a non-traditional programming model. In this paper we try to provide some insight into the peculiarities of both, as regards their cost, performance, programmability and limitations, in order to target scientific computing.*

Keywords: parallel programming, multicore, Cell BE, NVIDIA Tesla, CUDA

1 Introduction

Nowadays, multicore architectures are omnipresent and can be found in all market segments. In particular, they constitute the CPU of many embedded systems (for example, video game consoles, network processors or GPUs), personal computers (for example, the latest developments from Intel and AMD), servers (the IBM Power6 or Sun UltraSPARC T2 among others) and even supercomputers (for example, the CPU chips used as building blocks in the IBM Blue-Gene/L and Blue-Gene/P systems). This market trend towards CMP (or chip-multiprocessor) architectures has given rise to platforms with a great potential for scientific computing: the Cell BE [28] and the GPGPUs [16, 22] whose best representative is the NVIDIA Tesla GPGPU series [20].

From the architectural point of view, the Cell BE can be classified as a heterogeneous CMP. In particular, the first generation of the chip integrates up to nine cores of two distinct types [13]. One of the cores, known as the *Power Processor Element* or PPE, is a 64-bit multithreaded PowerArchitecture-compliant processor with two levels of on-chip cache that includes the vector multimedia extension (VMX) instructions. The main role of the PPE is to coordinate and supervise the tasks performed by the rest of cores. The remaining cores (a maximum of eight) are called *Synergistic Processing Elements* and provide the main computing power of the Cell BE. Each SPE includes a local memory for keeping instructions and data that is not coherent with the PPE main memory. In this way, SPEs can simultaneously execute up to eight independent threads that need to be explicitly synchronized through a number of hardware-supported mechanisms. In addition, data transfers to and from the SPE local memories must be explicitly managed by programmers using a DMA engine.

CUDA [17] is a new hardware and software architecture for issuing and managing computations on the GPU, without the need of mapping them to a graphics APIs [25], common to the latest NVIDIA developments. Each CUDA-enabled device behaves as a massively-threaded computing device with a significant amount of on-board memory. Thus, it is composed of a variable number of *thread processors* and a *thread execution manager* that handles threading automatically. Both the amount of on-board memory and the number of thread processors depend on the specific GPU model. Data set is divided into smaller chunks stored in the on-board memory in order to feed the thread processors. Thereafter threads are intended to run in lockstep in a SIMD fashion acting as a data-parallel computing device. Unlike Cell BE developers, CUDA programmers don't have to write explicitly threaded code. Instead, the design of a correct data layout becomes the crucial task to obtain good performance and requires writing some explicit code.

The rest of the paper is organized as follows. In Section 2 we provide a short revision of the architecture of the Cell BE, a description of its main communication and synchronization primitives, and a review of the software infrastructure available to programmers. Next, in Section 3 we provide similar information for CUDA and the the NVIDIA Tesla C870 architecture. Then, a comprehensive comparison of both platforms is performed in Section 4. Finally, Section 5 gives the main conclusions of the paper.

2 Cell BE

2.1 Architecture

The Cell Broadband Engine (Cell BE) [13] is an heterogeneous multicore chip composed of one general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized co-processors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and an I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB) (see Figures 1(a) and 1(b)) [3].

The latest version of the Cell BE processor, running at 3.2 GHz, has a theoretical peak performance of 204.8 Gflops (single precision) and 14.63 Gflops (double precision). The EIB supports a peak bandwidth of 204.8 GB/s for intra-chip data transfers among the PPE, the SPEs, and the memory and the I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

The PPE is the main processor of the Cell BE, and is 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*), a 32 KB Ll instruction cache, a 32 KB L1 data cache, and a 512 KB L2 cache. The PPE is a dual issue, in-order execution, 2-way SMT processor. The PPE comprehends two different units, namely *PowerPC Processor Unit* (PPU) and *PowerPC Processor Storage Subsystem* (PPSS) (see Figure 1(c)).

On the other hand, each SPE is a 128-bit RISC processor specifically designed for high-performance on streaming and data-intensive applications [6]. Each SPE consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC) (see Figure 1(d)). SPUs are a in-order processors with two pipelines and 128 128-bit registers. 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 floating-point numbers. As opposed to the PPE, SPEs do not have a private cache memory. In contrast, SPUs include a 256 KB *Local Store* (LS) memory to hold both instructions and data of SPU programs, that is, SPUs cannot access main memory directly. The MFC contains a *DMA Controller* and a set of memory-mapped registers or *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.

2.2 Programming

The Cell BE has been specifically designed to exploit multiple levels of parallelism at the same time: (a) each SPE executes a different thread, (b) an SPE can overlap computation and communication by using non-blocking DMA operations, (c) SIMD instructions perform the very same operation on multiple data simultaneously, and (d) SPEs have two pipelines that can execute two instructions concurrently. Nevertheless, the main advantage also becomes the major drawback: Cell BE programming is as flexible as complex. Flexibility stems from the possibility to use a number of programming models depending on the application domain [10]. Complexity is due to the fact that threads must communicate and synchronize across program execution. To do that, the PPE and the SPEs can use a variety of mechanisms provided by the Cell BE architecture: *DMA transfers*, *mailboxes*, *signals* and *atomic operations* [1, 2].

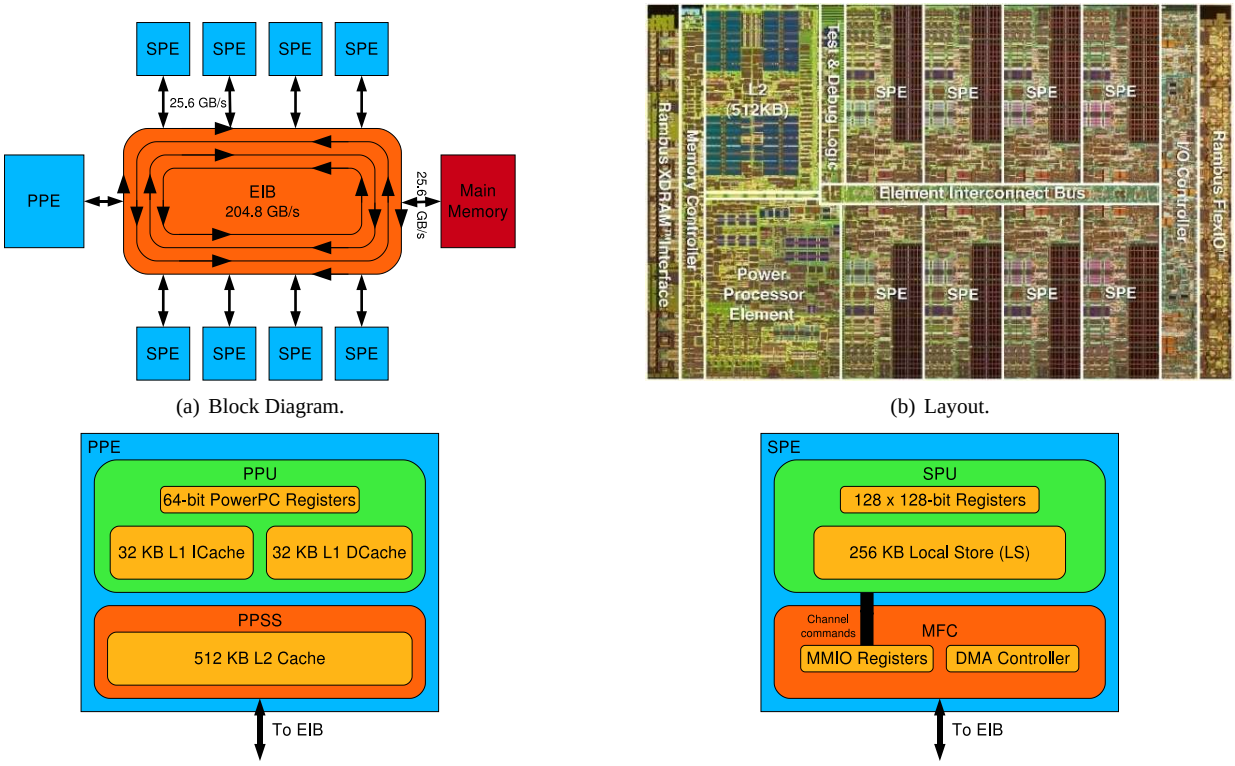
SPEs use DMA transfers to read from (GET) or write to (PUT) main memory. DMA transfer size must be 1, 2, 4, 8 or a multiple of 16 Bytes up to a maximum of 16 KB. DMA transfers can be either blocking or non-blocking. The latter allow to overlap computation and communication: there might be up to 128 simultaneous transfers between the eight SPE LSs and main memory. In addition, an SPE can issue a single command to perform a list of up to 2048 DMA transfers, each one up to 16 KB in size. In all cases, peak performance can be achieved when both the source and destination addresses are 128-Byte aligned and the size of the transfer is an even multiple of 128 Bytes [14].

Mailboxes are FIFO queues that support exchange of 32-bit messages among the SPEs and the PPE. Each SPE includes two outbound mailboxes, called *SPU Write Outbound Mailbox* and *SPU Write Outbound Interrupt Mailbox*, to send messages from the SPE; and a 4-entry inbound mailbox, called *SPU Read Inbound Mailbox*, to receive messages. Every mailbox is assigned a channel command and a MMIO register. The former allows the owner SPE to access the outbound mailboxes. The latter enables remote SPEs and the PPE to access the inbound mailbox.

In contrast, signals were designed with the only purpose of sending notifications to the SPEs. Each SPE has two 32-bit signal registers to collect incoming notifications. A signal register is assigned a MMIO register to enable remote SPEs and the PPE to send individual signals (*overwrite mode*) or combined signals (*OR mode*) to the owner SPE.

Read-modify-write atomic operations enable simple transactions on single words residing in main memory. For example, the *atomic_add_return* atomic operation adds a 32-bit integer to a word in main memory and returns its value before the addition.

Cell BE programming requires separate programs, written in C/C++, for the PPE and the SPEs, respectively. The PPE program can include C intrinsics (e.g., `vec_add`), to use its VMX unit; and library function calls [12], to manage threads and perform communication and synchronization



(a) Block Diagram.

(b) Layout.

(c) PowerPC Processor Element.

(d) Synergistic Processing Unit.

Figure 1. *Cell BE Architecture.*

operations (e.g., `spe_context_run`, `spe_mfcio_put` and `spe_in_mbox_write`). The SPE program follows an SPMD model (*Single Program Multiple Data*). It includes C intrinsics [9], to execute SIMD instructions, and communication and synchronization operations (e.g., `spu_add`, `mfc_get` and `spu_read_in_mbox`); and function calls to the SDK library [11], to carry out complex tasks of different nature (`transpose_matrix`, `fft_2d`, etc.).

3 CUDA

3.1 Architecture

All the latest NVIDIA developments such as GeForce 8 series, Quadro FX 5600/4600 and Tesla solutions are compliant with the *Compute Unified Device Architecture* (CUDA). Nevertheless, the NVIDIA Tesla GPGPUs, namely C870, D870 and S870, are the only ones that have been specifically designed for general-purpose computing given the fact that they have no graphics output. In particular, the NVIDIA Tesla C870 is a homogeneous CMP, with 128 cores and 1.5 GB of on-board memory, attached to the main CPU through a PCIe x16 interface. Under this configuration, the NVIDIA Tesla features a theoretical peak performance of 518 Gflops (single precision), a peak on-board memory bandwidth of 76.8 GB/s and a peak main memory bandwidth of 4 GB/s. In turn, the NVIDIA Tesla D870 and S870 computing solutions comprise two and four C870 units in a desktop and a 1U rack-mount chassis, respectively. Therefore, they come with 256 and 512 cores, and 3 GB and 6 GB of on-board memory, to provide a theoretical peak performance of 1036.8 Gflops and 2073.6 Gflops, respectively.

Each CUDA-compliant device is a set of multiprocessor cores (see Figure 2(a)), capable of executing a very high number of threads concurrently, that operates as a coprocessor to the main CPU or host. In turn, each multiprocessor has a SIMD architecture, that is, each processor of the multiprocessor executes a different thread but all the threads run the same instruction, operating on different data based on its `threadId`, at any given clock cycle. The NVIDIA Tesla C870 has sixteen multiprocessors with eight processors each.

Both the host and the device maintain their own DRAM, referred to as *host memory* and *device memory* (on-board memory). Device memory can be of three different types (see Figure 2(b)): *global memory*, *constant memory* and *texture memory*. They all can be read from or written to by the host and are persistent through the life of the application. Nevertheless, global, constant and texture memory spaces are optimized for different memory usages. The NVIDIA Tesla C870 has 1.5 GB of global memory and 64 KB of constant memory.

Multiprocessors have on-chip memory that can be of the four following types: *registers*, *shared memory*, *constant cache* and *texture cache* (see Figure 2(a)). Each processor in a multiprocessor has one set of local 32-bit read-write *registers* per processor. A parallel data cache of *shared memory* is shared by all the processors. A read-only *constant cache* is shared by all the processors and speeds up reads from the constant memory. A read-only *texture cache* is shared by all the processors and speeds up reads from the texture memory. The local and global memory spaces are implemented as read-write regions of device memory and are not cached. The NVIDIA Tesla C870 has 8192 registers and 16 KB of shared memory per multiprocessor.

A portion of a parallel application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device by many threads running on different processors of a multiprocessor. Such a function, called a *kernel*, is compiled to the instruction set of the device and downloaded into it.

A kernel is organized as a set of thread blocks as shown in Figure 2(c). A *thread block* is a batch of threads that can cooperate together by efficiently sharing data through the shared memory and synchronizing their execution to coordinate memory accesses using the primitive `__syncthreads()`. Each thread block executes on one multiprocessor. Each thread has its own *thread ID*, which is the number of the thread within a one-, two- or three-dimensional array of arbitrary size. The use of multidimensional identifiers helps to simplify memory addressing when processing multidimensional data.

The number of blocks in a thread block is limited (512 threads per block in the NVIDIA Tesla C870). Therefore, blocks of equal dimension and size that execute the same kernel can be batched together into a *grid of thread blocks*. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. In contrast, this mode allows thread blocks of the same kernel grid to run on any multiprocessor, even from different devices, at any time. Again, each block is identified by its *block ID*, which is the number of the block within a one- or two-dimensional array of arbitrary size for the same reasons as above. It is worth noting that kernel threads are extremely lightweight, i.e. creation overhead is negligible and context switching is essentially free.

In this scenario, a thread can access device memory through the following memory spaces: read-write per-thread registers, read-write per-thread local memory, read-write per-block shared memory, read-write per-grid global memory, read-only per-grid constant memory and read-only per-grid texture memory. Note that the global, constant, and texture memory spaces are persistent across kernel launches by the same application.

3.2 Programming

CUDA tries to simplify the programming model by hiding thread handling from programmers, i.e. there is no need to write explicit threaded code in the conventional sense. Instead, CUDA includes C/C++ software-development tools that allow programmers to mix host code with device code [7]. To do so, CUDA programming requires a single program written in C/C++ with some extensions to the C language [17]:

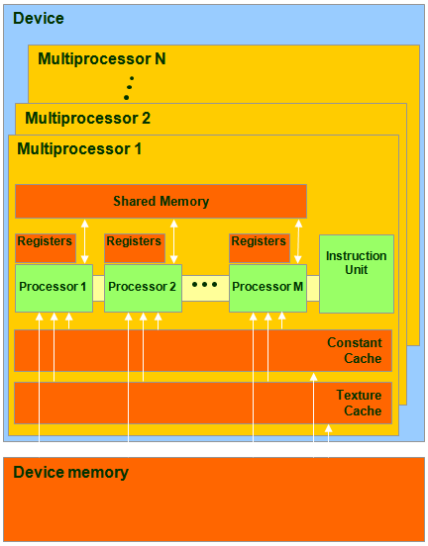
- Function type qualifiers for functions that execute on the device (`__global__` and `__device__`).
- Variable type qualifiers for variables that reside on device memory (`__device__`, `__shared__` and `__constant__`).
- Four built-in variables that specify the grid and block dimensions, the block index within the grid and thread index within the block (`gridDim`, `blockDim`, `blockIdx` and `threadIdx`), accessible in `__global__` and `__device__` functions.
- An *execution configuration construct* to specify the dimension of the grid and blocks when launching kernels, declared with the `__global__` directive, from host code (for example, `function<<<gridDim, blockDim, shm_size>>>(parameter_list)`).

Besides, CUDA comes with a runtime library, split into a host component, a device component and a common component, that supports built-in vector data types and texture types to access texture memory, and provides a number of mathematical functions, type conversion and casting functions, thread synchronization functions, and device and memory management functions. Finally the CUDA environment also includes two higher-level mathematical libraries of common usage, namely CUBLAS [18] and CUFFT [19].

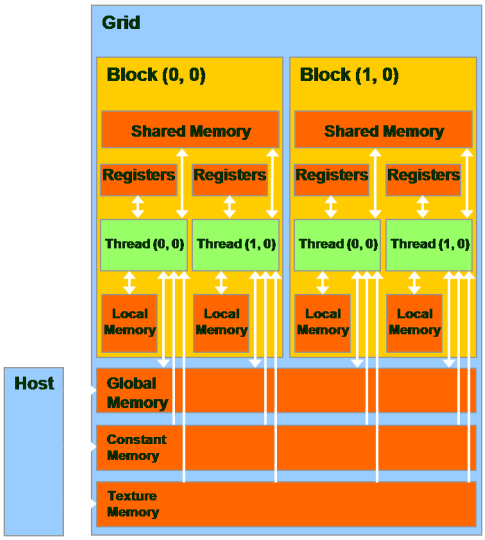
Kernel launches using the above mentioned execution configuration construct are asynchronous, that is, control returns to host immediately. Then, the main CPU is free to do whatever is required until `cudaThreadSynchronize()` is invoked so that the host blocks until all previous CUDA calls complete. It is worth noting that memory allocation and movement of data between host memory and device memory is left to programmers. They must allocate the required buffers in either host or device memory, and also copy data back and forth between host memory and device memory, using the functions provided by the runtime library (`cudaMalloc()`, `cudaFree()` and `cudaMemcpy()`).

4 Comparison

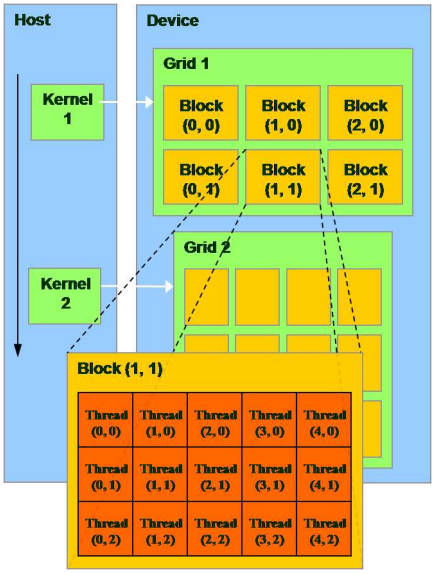
In this section we compare a single Cell BE with the NVIDIA Tesla C870 considering different theoretical and practical aspects and based on our own experience with these platforms:



(a) CUDA Hardware Model.



(b) CUDA Memory Model.



(c) CUDA Programming Model.

(d) NVIDIA Tesla C870.

Figure 2. *CUDA Architecture.*

- **Cost**. CUDA-enabled devices range from less than one hundred dollars for the lowest GeForce 8 series models up to $1300 dollars for the NVIDIA Tesla C870. The PlayStation 3 includes a fully-programmable Cell BE by installing a Linux distribution such as YDL 6.0 [27] and can be found at around $500 (60 GB version). However, the PlayStation 3 is not suitable for scientific computing in many cases due to its memory shortage (less than 200 MB left for applications) as well as the unavailability of two out of eight SPEs. IBM dual Cell-based blades are priced at around $10.000. The cost measured in $/Gflop in function of these numbers is ten times higher for the Cell BE solution when compared to the NVIDIA Tesla C870.

- **Main memory bandwidth**. Programming the Cell BE involves explicitly moving data back and forth between main memory and SPEs' LSs. On the other hand, programming the NVIDIA Tesla C870 also involves moving data back and forth between main memory and device memory. Therefore main memory bandwidth is a key parameter to get good performance. In this sense, SPEs can get very close to the theoretical peak memory bandwidth of 25.6 GB/s [1]. Meanwhile, most CUDA-enabled devices have a PCIe x16 interface which means a theoretical peak device-to-host bandwidth of 4 GB/s. Nevertheless, this limit is not reached by the NVIDIA Tesla C870 that obtains 1.5 GB/s for regular memory and 3.1 GB/s for page-locked memory.

- **Theoretical peak performance**. A single Cell BE has a theoretical peak performance of 204.8 Gflops for single precision and 14.63 Gflops for double precision. In the meantime, the NVIDIA Tesla C870 has a peak performance of 518 Gflops for single precision and has no support for double precision. These numbers show the notable potential of these platforms when compared with state-of-the-art conventional microprocessors [21]. In spite of that, the actual performance achieved by applications on these platforms greatly depends on the characteristics of the target problem and the ability of programmers to optimize their codes [23, 24, 26].

- **IEEE-compliant floating-point support**. Neither the Cell BE nor CUDA-enabled devices have fully IEEE-compliant floating-point support [10, 21]. Consequently, results generated by applications using these platforms may slightly differ from their serial counterparts under certain circumstances. Finally, both platforms are expected to provide double-precision floating-point support in their next generations.

- **Target applications**. The Cell BE is able to exploit not only data-level parallelism but also task-level parallelism and provides support for both single- and double-precision floating-point operations. Moreover, it enables a number of different programming models such as the function-offload model, the pipeline model or the shared-memory multiprocessor mode among others [10]. In contrast, CUDA-enabled devices, including the NVIDIA Tesla C870, aim at data-intensive applications with a very high arithmetic intensity that only need single-precision floating-point math. In this sense, the Cell BE provides a more versatile solution at the expense of a more modest performance.

- **Programming learning curve**. Even though the learning curve of Cell BE programming is much harder than that of the CUDA environment, both platforms require a significant training effort. Cell BE developers have to deal with thread management, data movement through DMA transfer operations, thread coordination using mailboxes, signals and atomic operations, and SPE code SIMDization. All these tasks require a good understanding of the Cell BE microarchitecture along with the use of non-standard C intrinsics to perform such operations. In contrast, CUDA hides most microarchitectural details of NVIDIA Tesla C870 from programmers, thread management and scheduling are handled by the thread execution manager, code doesn't need to be SIMDized, and data movement is considerably simpler. However, it should be noted that the C extensions and the runtime library functions impose many restrictions [17] that make programming more difficult than it could appear at first glance.

- **Code optimization**. For now, developing efficient code for the Cell BE requires a number of manual optimizations mostly on the SPE code [5]. Such optimizations include double-buffering DMA data transfers, reordering instruction scheduling to maximize dual-issue cycles, branch hinting to reduce performance impact of branches, generation of SIMDized code to fully exploit the SPE architecture, and a number of memory-alignment issues. Life is not much easier for CUDA programmers. In order to maximize performance, they must expose as much parallelism as possible (structuring the algorithm to maximize independent parallelism, creating as many threads as possible, and taking advantage of asynchronous kernel launches and asynchronous data transfers by means of CUDA streams); optimize memory usage for maximum bandwidth (minimizing data transfers across the PCIe link, and optimizing memory access patterns to get coalesced global memory accesses and shared memory accesses with no or few bank conflicts); maximize occupancy to hide latency; and optimize instruction usage for maximum throughput (minimizing use of low-throughput instructions and divergent warps). For more details see [21].

- **Debuggability**. The Cell BE SDK includes the `spu-timing` tool and the Mambo simulator [4] that allow to perform static and dynamic timing analysis of SPE code, and also a performance debugging tool and a visual performance analyzer. The CUDA compiler helps debugging by enabling a device emulation mode that allows to use native debug support. Also, CUDA incorporates a visual profiler that relies on hardware performance counters to help identifying potential performance problems. Our experience indicates that debugging code is still an arduous task in both platforms.

- **Portability**. Source code specifically written and optimized for the Cell BE is highly architecture-dependent. However, if future generations of the Cell BE are fully-compliant with the Cell BE Architecture specification [8], programmers should be able to port code to the new Cell BE versions with no much effort. CUDA source code is intended to be operational with no changes in all CUDA-enabled devices [17]. But to some extent CUDA source code is bound to the specific device microarchitecture it was primarily developed for. In this sense, when programmers determine the best layout of grids and blocks, they must take into consideration factors as the number of thread processors or the device memory configuration.

- **Integration**. Fine-grained parallelism exploited by both platforms nicely complements the comparatively coarse-grained parallelism available in other parallel programming APIs such as MPI. In this way, these platforms can easily integrate into higher-level hierarchical parallel systems [15].

5 Conclusions

The Cell Broadband Engine and the NVIDIA Tesla C870 are currently concentrating an enormous attention due to their tremendous potential in terms of sustained performance. However, they pose a number of architectural and programming peculiarities that are worth analyzing and comparing. In this paper we have provided not only a complete description of both platforms but also a comprehensive comparison based on our own experience with these platforms. We have explored different aspects such as their cost, main memory bandwidth, performance, floating-point support, target applications, programmability, debuggability, portability and integration. This analysis exposes the pros and cons of both platforms and can help application programmers to pick the most appropriate platform for parallelizing scientific codes.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03".

References

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. CellStats: a Tool to Evaluate the Basics Synchronization and Communication Operations of the Cell BE. In *16^{th} Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, February 2008.

[2] J. L. Abellán, J. Fernández, and M. E. Acacio. Characterizing the Basic Synchronization and Communication Operations in Dual Cell-Based Blades. In *International Conference on Computational Science*, Krákow, Poland, Junio 2008.

[3] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB On-chip Network. *IEEE Micro*, 27(5):6–14, September 2007.

[4] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a Full System Simulator for the PowerPC Architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2005.

[5] A. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. S. an d Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[7] T. R. Halffill. Parallel Processing with CUDA. *MicroProcessor Report Online*, January 2008.

[8] IBM Systems and Technology Group. *Cell Broadband Engine Architecture V1.01*, October 2006.

[9] IBM Systems and Technology Group. *C/C++ Language Extensions for Cell BroadBand Engine Architecture V2.4*, March 2007.

[10] IBM Systems and Technology Group. *Cell Broadband Engine Programming Tutorial Version 2.1*, March 2007.

[11] IBM Systems and Technology Group. *Cell Broadband Engine SDK Libraries Version 2.1*, March 2007.

[12] IBM Systems and Technology Group. *SPE Runtime Management Library Version 2.1*, March 2007.

[13] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[14] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3):2–15, May/June 2006.

[15] Los Alamos National Laboratory. *Los Alamos RoadRunner*, 2008. http://www.lanl.gov/roadrunner/.

[16] D. Manocha. General-Purpose Computation Using Graphic Processors. *IEEE Computer*, 38(8):85–88, August 2005.

[17] NVIDIA Corporation. *NVIDIA Compute Unified Device Architecture (CUDA) Programming Guide Version 1.1*, November 2007.

[18] NVIDIA Corporation. *NVIDIA CUDA CUBLAS Library Version 1.0*, June 2007.

[19] NVIDIA Corporation. *NVIDIA CUDA CUFFT Library Version 1.1*, October 2007.

[20] NVIDIA Corporation. *NVIDIA Tesla Computing Solutions for HPC*, 2008. http://www.nvidia.com/page/hpc.html.

[21] NVIDIA Tutorial at PDP'08. *CUDA: A New Architecture for Computing on the GPU*, February 2008.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[23] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proceedings of International Conference on Parallel and Distributed Systems*, Long Beach, CA, April 2007.

[24] F. Petrini, D. Scarpazza, O. Villa, and J. Fernández. Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors. In *Proceedings of International Conference on Parallel and Distributed Systems*, Long Beach, CA, April 2007.

[25] M. Pharr, editor. *GPU Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Randima Fernando, Series Editor. Addison-Wesley, 2005.

[26] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, and W. mei W. Hwu. Program Optimization Study on a 128-Core GPU. In *Proceedings of First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[27] Terra Soft Solutions. *Yellow Dog Linux v6.0*, 2008. http://www.terrasoftsolutions.com/products/ydl/.

[28] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *Proceedings of the 3^{rd} ACM Conference on Computing Frontiers*, Ischia, Italy, May 2006.