Exploiting the Cell/BE architecture with the StarPU unified runtime system

Cédric Augonnet, Samuel Thibault, Raymond Namyst, Maik Nijhuis

Exploiting the Cell/BE architecture with the StarPU unified runtime system

Cédric Augonnet[1], Samuel Thibault[1], Raymond Namyst[1], and Maik Nijhuis[2]

[1]INRIA Bordeaux Sud-Ouest – LaBRI – University of Bordeaux
[2]Vrije Universiteit Amsterdam
cedric.augonnet@inria.fr   {samuel.thibault,raymond.namyst}@labri.fr   maik@cs.vu.nl

Abstract. Core specialization is currently one of the most promising ways for designing power-efficient multicore chips. However, approaching the theoretical peak performance of such heterogeneous multicore architectures with specialized accelerators, is a complex issue. While substantial effort has been devoted to efficiently offloading parts of the computation, designing an execution model that unifies all computing units is the main challenge.
We therefore designed the STARPU runtime system for providing portable support for heterogeneous multicore processors to high performance applications and compiler environments. STARPU provides a high-level, unified execution model which is tightly coupled to an expressive data management library. In addition to our previous results on using multicore processors alongside with graphic processors, we show that STARPU is flexible enough to efficiently exploit the heterogeneous resources in the CELL processor. We present a scalable design supporting multiple different accelerators while minimizing the overhead on the overall system. Using experiments with classical linear algebra algorithms, we show that STARPU improves programmability and provides performance portability.

1   Introduction

Multicore architectures are now widely adopted. Desktop personal computers and even laptops typically contain a multicore CPU and a powerful graphic card (GPU). The multicore Cell processor is used both in the PlayStation 3 gaming console as well as the Roadrunner supercomputer.

Similarly to the use of accelerating devices such as GPUs, core specialization addresses HPC's demand for more computational power, alongside with a better power efficiency. Future processors will therefore not only get more cores, but some of them will be tailored for specific workloads. While such designs intend to address architectural limits, exploiting them efficiently introduces numerous challenging issues at all levels, ranging from programming models and compilers to the design of scalable hardware solutions. The design of efficient runtime systems for these architectures is therefore a critical issue.

In a previous study, we have shown that the STARPU runtime system efficiently supports platforms with multicore CPUs alongside GPUs [1]. By delegating the management of low-level resources to STARPU, compilation environments or high performance libraries can concentrate on their primary algorithmic concerns in a portable fashion.
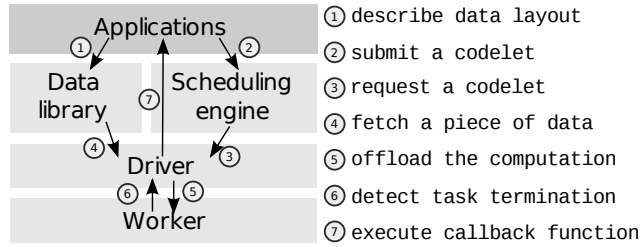
Applications	① describe data layout
① ②	② submit a codelet
Data library	③ request a codelet
⑦	
Scheduling engine	④ fetch a piece of data
④	⑤ offload the computation
Driver ③	⑥ detect task termination
⑥ ↑ ↓ ⑤	⑦ execute callback function
Worker	

Fig. 1. The path of a *codelet* through STARPU.

In this paper, we demonstrate that the design of STARPU is flexible enough for efficiently supporting the CELL architecture while taking its specificities into account. Using an asynchronous approach, we show that STARPU handles multiple accelerators with little overhead. The efforts required to port an application to the CELL architecture are very limited when using STARPU. Furthermore, STARPU's high-level scheduling optimizations are directly applicable on a variety of heterogeneous platforms. Therefore STARPU makes a big step towards **performance portability**.

The remaining of this paper is organized as follows. First, Section 2 introduces STARPU. Section 3 presents the architecture of the CELL processor and the CELL Run Time Library (CELL-RTL), which is used by STARPU. The design of a driver for CELL-RTL is discussed in Section 3.2, and we evaluate its efficiency in Section 4. After comparing our results with existing work in Section 5, we conclude this paper and give future work directions in Section 6.

2 STARPU, a unified runtime system

The STARPU runtime system offers support for heterogeneous multicore processors by offering a high level abstraction of tasks, named *codelet*, which can be executed on different architectures such as homogeneous multicore processors, GPUs and CELL processors. STARPU uses all computing resources at the same time by transparently mapping *codelets* as efficiently as possible on all available resources while hiding low-level technical mechanisms.

From a programming point of view, STARPU is not a new language but a library that execute tasks explicitly submitted by the application. STARPU also takes particular care of scheduling those tasks efficiently and allows scheduling experts to implement custom scheduling policies in a portable fashion.

The design of STARPU is organized around two main components: a data management library that offers a high level interface for manipulating data distributed across a heterogeneous machine; and a unified execution model which executes the tasks encapsulated into the *codelet* structure.

A data management library. Accessing main memory from an accelerator core, such as a GPU, is usually either impossible or extremely costly. STARPU therefore provides a data management library which offers a high level interface for manipulating the pieces of data which are used by *codelets* [1]. This library offers a *distributed*
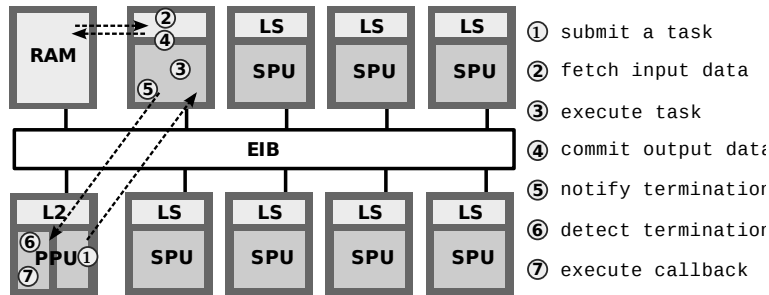
Fig. 2. The CELL-RTL and the CELL architecture.

shared memory by enforcing data coherency along the machine while protecting it from concurrent modifications. Its transparent data migration or replication mechanisms also perform extra transformations that are useful in a hybrid environment, such as automatically converting endianness or remapping data into a layout that is more suitable for the target architecture. As shown in Figure 1, applications describe the data layout so that the drivers can perform data transfers at a high abstraction level.

A unified execution model. Applications *asynchronously* submit tasks to STARPU in the form of *codelets*. A ***codelet*** encapsulates a task that can be executed on one or more of the compute resources controlled by STARPU. We will denote the compute resources as *workers*. A *codelet* contains a high-level description of the data it accesses, its implementations on the various *workers*, and a callback function which STARPU calls after executing the *codelet*. Programmers can also include extra hints (*e.g.,* priorities) to guide the scheduling engine.

Figure 1 shows that supporting a new architecture in STARPU requires limited efforts. On the one hand, such a *driver* must support launching the computation of the *codelet* it got from the scheduler. On the other hand, it needs to implement the methods for transferring buffers to or from the corresponding architecture. There is exactly one driver per *worker*, therefore there can be multiple instances of a driver. The drivers currently available for multicore CPUs and NVIDIA GPUs are *synchronous*. When a *codelet* is executing, the corresponding driver waits for the completion of the *codelet* and it can not execute other *codelets*.

3   Extending STARPU for the CELL processor

The CELL processor is a heterogeneous multicore chip composed of a main hyper-threaded core, which is the Power Processing Unit or PPU, and 8 coprocessors named Synergistic Processing Units or SPUs. The SPUs can only directly access 256 KB local memory. All cores are interconnected by the Element Interconnect Bus (EIB). Data transfers between the main memory and SPUs' local memory require explicit DMA mechanisms, which use the EIB. The EIB also supports hardware *mailbox* mechanisms for synchronization purposes.

3.1 The CELL Run Time Library

The CELL Run Time Library (CELL-RTL), co-developed in Amsterdam Vrije Universiteit and Université de Bordeaux, was initially designed as a back-end for the CELL-SPACE framework for building streaming applications on CELL processors [9]. CELL-RTL executes tasks on the SPUs with little overhead while applying some optimizations such as *multibuffering* (overlapping multiple memory transfers and computation) or task reordering. As shown on Figure 2, CELL-RTL implements efficient data transfers, and since it offers an interface to asynchronously submit tasks to the SPUs, adding a basic CELL-RTL driver requires limited efforts.

All these characteristics make CELL-RTL an excellent target for our STARPU runtime system, which can somehow be seen as an extension to CELL-RTL that features a data management library which protects data from concurrent accesses, and a scheduling engine which distributes the *codelets* over the accelerators.

However, a few considerations have to be studied before designing a CELL-RTL driver for STARPU. In particular, since SPUs typically handle fine grain tasks, it is crucial to hide most overhead by overlapping DMA transfers and computation. CELL-RTL therefore supports multiple pending tasks on a single SPU, and automatically applies *multibuffering* techniques. Unlike the current implementation of the STARPU drivers for multicore and NVIDIA GPUs, the CELL-RTL driver for the SPUs therefore has to be **asynchronous**, which means that task submission is non-blocking so that it is possible to submit multiple jobs simultaneously.

3.2 Designing a STARPU driver for CELL-RTL

In this section, we study the requirements to implement the CELL-RTL driver. Firstly, the fine granularity of jobs that typically run on SPUs implies that it is crucial that STARPU asynchronously submits multiple jobs to CELL-RTL. In order to scale with the number of SPUs, we must use a single thread to control all SPUs at the same time to avoid the important cost of context switches on the PPU (we measured a $1.6\mu s$ overhead per context switch). Having a SPU to send an interrupt to the PPU to notify the termination of a task is extremely costly. In CELL-RTL, SPUs send notification messages by the means of DMA transfers. As a result, we need a mechanism to poll CELL-RTL regularly in the background which we call **progression**. Thirdly, synchronizing cores within the CELL architecture is expensive. In order to limit the amount of interactions between the PPU and the different SPUs, we use a CELL-RTL mechanism called *job chaining* [9].

An asynchronous driver controlling multiple workers. Instead of synchronously submitting a single task at a time and waiting for its termination, the driver for the CELL-RTL requests multiple *codelets* from the STARPU *scheduling engine* and submits them all at once to CELL-RTL. Since both STARPU and CELL-RTL use similar callback mechanisms, the callback of a CELL-RTL job is responsible for the termination of the corresponding STARPU *codelet*.

Similarly to the driver for the NVIDIA GPUs [1], the CELL-RTL driver requires one CPU thread, which in turn needs to run on a CPU core. In contrast with usual multicore machines equipped with GPUs, the CELL has more coprocessors than CPU

cores (up to 8 SPUs vs 2 PPU contexts if we consider hyperthreading). Having a separate driver for each coprocessor would therefore overload the PPU due to the numerous context switches between the driver threads, impacting the overall performance. An asynchronous driver which controls multiple workers does not have this problem and can scale with the number of accelerators. This approach can also be applied to multi-GPU setups, which are now becoming standard.

Progression within STARPU. Efficient synchronization between the different cores is a challenging issue which CELL-RTL addresses by using DMA for notifying task terminations instead of using the inefficient *hardware mailbox* mechanisms [9]. Although this DMA approach is very efficient, it requires that the application polls regularly to detect the completion of SPU tasks. Thus, STARPU needs a progression mechanism to poll the CELL-RTL driver.

Polling only before or after the submission of a new task is not sufficient: STARPU must not be blocked waiting for a resource that can only be released by the CELL-RTL driver. Let us for instance consider two tasks A and B which both modify a data item D. First, the driver gets A from the scheduler, takes a lock that protects D, asynchronously submits task A to CELL-RTL, and then gets B from the scheduler. Before executing B, it needs to take the lock that protects D. Waiting for this lock would block the driver and thus deadlock: the driver does not poll CELL-RTL, since it waits for the lock. Therefore it does not notice the completion of A, and A does not release its lock on D. Not only the CELL-RTL driver, but the whole of STARPU therefore needs a progression mechanism which avoids such deadlocks.

The simplest solution to ensure progression is launching a separate progression thread. Since the hyperthreaded PPU supports two threads, adding a second thread to the thread already devoted to the driver may seem reasonable. This however yields too much overhead as shown in Section 4. Moreover, in most cases polling before the submission of a new task should be sufficient.

Considering that deadlocks only occur when the driver for CELL-RTL is blocked, another solution consists in adding a progression mechanism within every blocking procedure which could wait for the termination of a task. As illustrated by Figure 1, the only resources manipulated by a STARPU driver are *codelets* (step 3) and data (step 4). Instead of getting blocked waiting for a lock that protects a piece of data which is used by another task, the data management library will make STARPU progress until the resource is available again. Likewise, if a driver requests a *codelet* from the scheduling engine and none is schedulable, the scheduler will make STARPU progress. In both cases, consuming CPU cycles to poll on behalf of the blocked driver is not an issue since the corresponding worker would have been stalled anyway. Moreover, such a polling is likely to improve reactivity, thereby reducing the time while the worker would have been waiting otherwise.

Transparent job chaining. Instead of submitting a single job to CELL-RTL at a time, programmers can directly inject a list of jobs that the SPUs fetch autonomously: instead of handling the submission and termination one job at a time, the PPU only submits a single chained job, and gets notified upon the completion of the entire chain.

STARPU exploits this job chaining mechanism transparently for the programmer. Since the design of the scheduling engine of STARPU is based on a set of *codelet*
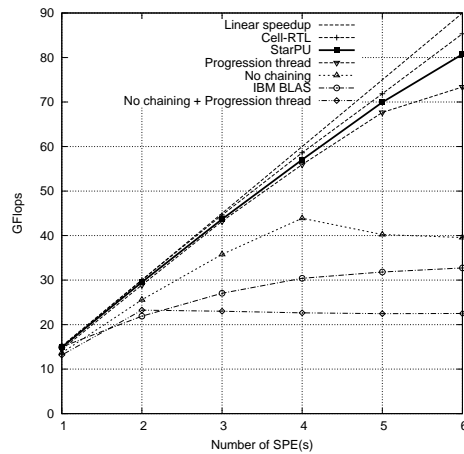
Fig. 3. Matrix multiplication.

queues, the CELL-RTL driver can directly request a list of *codelets* from the scheduler. It is worth noting that the scheduling engine may have applied scheduling optimizations beforehand, *e.g.,* reordering the *codelet* list. In other words, the CELL-RTL drivers gets all the benefits of the scheduling policies which are enforced by STARPU, regardless of the underlying hardware.

When using CELL-RTL without the support from STARPU, programmers explicitly have to construct those chains. Manually selecting the optimal number of chains and their respective size is not necessarily simple. For example, programmers have to supply at least two chains per SPU to ensure that *multibuffering* is possible. Requiring application programmers to have such a specific knowledge of the CELL architecture and the design of the CELL-RTL itself is not compatible with our portability concerns. In contrast, getting STARPU to construct those chains automatically relieves programmers from a serious burden. When the CELL-RTL driver gets a list of *codelets* from the scheduler, the list it split into multiple chunks. To ensure the use of multibuffering, there are up to two chunks per SPU. However, we create less chunks if their length is below some minimal size, so that we avoid injecting inefficient small chains.

4 Evaluation

In order to validate our approach, we present how well STARPU performs with a few applications which we run on the CELL processor. We then discuss the effects of the various design choices we have made.

Experimentation platform. All the experiments presented in this paper were executed on a SONY PLAYSTATION 3 running a FEDORA CORE 8 LINUX 2.6.26 kernel and using IBM SDK 3.1. Applications can use 1 PPU and 6 SPUs, which are clocked at 3.2 GHz. It is important to note that with this configuration, the memory available is below 200 MB. We therefore only present benchmarks with a very small size.
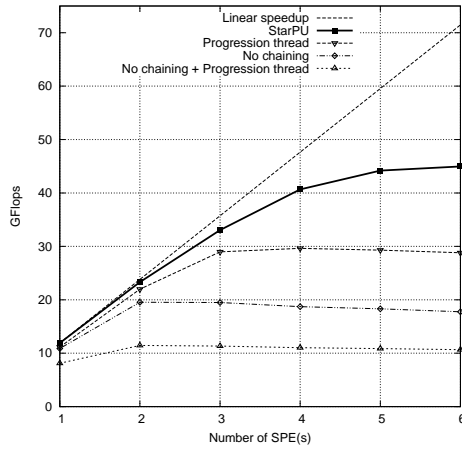
Fig. 4. CHOLESKY decomposition.

Table 1. Impact of job chaining on scalability.

	Speed on 1 SPU (in GFLOPS)		Speedup on 6 SPUs (against 1 SPU)	
	without chaining	with chaining	without chaining	with chaining
Matrix multiplication	13.67	14.90	2.89	5.42
CHOLESKY decomposition	10.89	11.91	1.63	3.77

Benchmarks. We first analyze the behaviour of two single precision dense linear algebra applications written on top of STARPU. Since there is no comprehensive implementation of the BLAS kernels currently available for the SPUs, we use the SGEMM kernel from IBM SDK, and we use the SPOTRF and STRSM kernels written by KURZAK *et al.* [7]. Figures 3 and 4 illustrates the benefits of our different optimizations. A direct implementation with CELL-RTL gives the overhead caused by STARPU itself. Matrix multiplication is a set of independent identical tasks, so that we obtain near linear speedups on Figure 3. The scalability of the CHOLESKY decomposition algorithm is however limited by a lack of parallelism, which our optimization techniques help to reduce.

The impact of job chaining. By reducing the need for synchronization, job chaining reduces the amount of interaction between STARPU and the coprocessors. As a result, Table 1 illustrates that job chaining gives better task throughput so that each SPU gets better performance. Chaining also improves scalability on both those benchmarks which hardly get any speedup otherwise.

The choice of the progression mechanism. Avoiding the use of a progression thread brings even more benefits on scalability and on the overall performance. As mentioned in Section 3.2, the use of a separate progression thread creates a sensible burden for the operating system which is measured in Table 2. This table gives the output of the UNIX `time` command, which not only gives the total execution time, but also the amount of time spent in the LINUX kernel. We also use the content of the

Table 2. OS overhead during CHOLESKY decomposition.

	With progression thread	No progression thread
Total execution time	3.2s	2.1s
Time spent in the OS	1.5s (47.9%)	0.48s (22.9%)
Number of calls to schedule() per second	CPU0 : 142 CPU1 : 4242	CPU0 : 47 CPU1 : 51

Table 3. The impact of prioritized tasks on CHOLESKY decomposition.

	Without priority	With priority
1 SPU	11.57 GFLOPS	11.91 GFLOPS
6 SPUs	43.42 GFLOPS	44.96 GFLOPS

`/proc/schedstat` file to determine the number of calls to the `schedule()` function of the LINUX kernel. When using a separate thread, this number explodes from one every $20\,ms$ to one every $250\,\mu s$, which shows the pressure on the operating system. Avoiding overloading the system not only saves some computational power for the applications that run on the PPU, but it also improves overall system reactivity and thus reduces synchronization overhead.

A low overhead. It is interesting to compare our results with other runtime systems which run the same computation kernels. The simplicity of the matrix multiplication algorithm makes it easy to re-implement it using different tools. Figure 3 gives a comparison of the performance of different implementations of a matrix multiplication based on the same computation kernel on SPUs.

We have first built a reference implementation which directly uses the PPU interface of the SPU-accelerated BLAS library shipped with IBM SDK 3.1 for the Cell processor. We have then realized a STARPU implementation and an implementation directly on top of CELL-RTL which does not use STARPU.

All implementations use the same computation kernel, so they obtain similar results on a single SPU; however, the performance of the reference implementation does not scale. Although STARPU adds an additional software layer, and protects its data from concurrent accesses, Figure 3 shows that STARPU has a low overhead: the performance of the CELL-RTL implementation is only slightly better than that of the STARPU implementation. Moreover, STARPU automatically exploits job chaining while we explicitly had to construct chains of proper size in the CELL-RTL implementation.

The benefits of STARPU scheduling policies. The scheduling engine, which is independent from the drivers, can reorganize the lists of *codelets* according to the scheduling policy before the CELL-RTL driver requests from those lists.

Table 3 illustrates the benefits of a scheduling policy with support for priorities. Previously we obtained similar improvements using this scheduling policy with the same application using a multicore machine equipped with a GPU [1]. This experiment confirms that STARPU is generic enough to efficiently exploit architectures as different as GPUs and CELL processors.

A portable approach. Provided the proper computer kernels, we only had to ensure proper data alignment, and add them to the SPU function library in CELL-RTL.

Porting our benchmarks therefore only required adding a couple lines to the code that already runs on multicore processors alongside with a GPU. The benefits resulting from scheduling policies show that STARPU offers portable performance.

5 Related work

Regardless of portability issues, accelerators are typically programmed using constructors' low level API nowadays. In addition to standard graphic APIs, AMD's FIRE-STREAMand especially NVIDIA's CUDA are currently the most common way to program GPUs. CELL processors are usually programmed directly with the LIBSPE.

While most efforts aim at optimizing computation kernels, the demand for a unified approach is illustrated by the OPENCL standard which not only attempts to unify programming paradigms, but also proposes a low-level device interface. Likewise, substantial attention is given to the use of well-established standards such as MPI [10] or OPENMP [2]. STARPU could provide a common runtime system for the numerous programming languages that were designed (or extended) to exploit accelerators [8, 4].

Various runtime systems were designed to offer support for the CELL architecture [3, 9, 12]. While most approaches adopt similar tasking APIs, few of them target heterogeneous platforms with various accelerator technologies at the same time, and even fewer consider the use of both accelerators and multicore CPUs. IBM ALF [3] and SEQUOIA [5] for instance target both CELL and multicore processors, but not GPUs. CHARM++ also offers support for both CELL [6] and GPUs [12], however there is no performance evaluation available yet for GPUs to the best of our knowledge. Contrary to STARPU which offers a flexible interface to design portable scheduling policies, SEQUOIA and CELLGEN [11] do not focus on the scheduling problematics and only use basic load balancing mechanisms which may not be sufficient when dealing with irregular applications and heterogeneous platforms.

6 Conclusions and future work

In this paper, we described how our STARPU runtime system was extended to efficiently taking advantage of the CELL architecture. We discussed how to support multiple accelerators while keeping a low overhead on the operating system, thereby saving computational power on the PPU. We have shown that STARPU and CELL-RTL integrate well. CELL-RTL efficiently executes the tasks that were injected by the STARPU driver while applying low-level CELL specific optimizations on the SPUs. STARPU and its higher-level abstraction enforces data consistency and maps *codelets* onto the different SPUs.

Surprisingly, when porting our existing linear algebra benchmarks, the most serious concern was finding the proper compute kernels given the very limited number of BLAS kernels currently implemented. Once these kernels were available, porting existing linear algebra benchmarks to the CELL only required adding a few lines of code, which demonstrates the contribution of STARPU in terms of *programmability*.

STARPU is not only *portable* in the sense that it allows a single application to run on multiple architectures. It also allows *performance portability*, since the application benefits from STARPU optimizations regardless of the underlying hardware.

In the future, we plan to extend our data management library to be fully asynchronous. This enhancement allows maintaining consistency directly at the SPUs. It also allows transparent support of SPU pipelining, which is particularly useful for streaming applications.

We will also port our model to other architectures, for instance by adding a driver for the OPENCL unified device interface. We will also implement dynamic code loading mechanisms in CELL-RTL to avoid wasting the limited local memory on SPUs. We are currently porting the PASTIX and the MUMPS sparse matrix solvers. We also envision using STARPU as a backend for high-level languages (*e.g.,* CellSs or HMPP) that could automatically generate STARPU *codelets*.

Selecting the optimal *codelet* granularity is a difficult algorithmic issue, especially in a heterogeneous context. STARPU could thus give feedback to the applications or the compiler environment so that it can adapt its behaviour dynamically or even offline.

References

1. C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Highly Parallel Processing on a Chip*, 2008.
2. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. In *ACM/IEEE conference on SuperComputing*, 2006.
3. C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the Cell Broadband Engine processor. In *Conference On Computing Frontiers*, 2008.
4. R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. Technical report, CAPS entreprise, 2007.
5. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Young Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *ACM/IEEE Conference on Supercomputing*, 2006.
6. D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006.
7. J. Kurzak, A. Buttari, and J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9), 2008.
8. M. D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the Rapid-Mind Development Platform. In *GSPx Multicore Applications Conference*, 2006.
9. M. Nijhuis, H. Bos, H. Bal, and C. Augonnet. Mapping and synchronizing streaming applications on Cell processors. In *International Conference on High Performance Embedded Architectures & Compilers*, 2009.
10. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the Cell Broadband Engine processor. *IBM Syst. J.*, 45(1), 2006.
11. S. Schneider, J.S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09 Proceedings*, New York, 2008. ACM.
12. L. Wesolowski. An Application Programming Interface for General Purpose Graphics Processing Units in an Asynchronous Runtime System. Master's thesis, 2008.