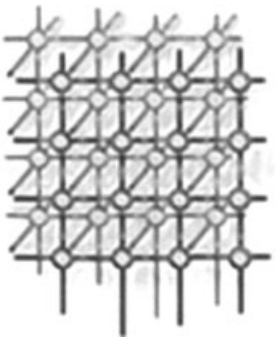
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:85–100
Published online 10 July 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1335



Evaluating application mapping scenarios on the Cell/B.E.‡

Ana Lucia Varbanescu[1,*,†], Henk Sips[1], Kenneth A. Ross[2,3], Qiang Liu[4], Apostol (Paul) Natsev[2], John R. Smith[2] and Lurng-Kuo Liu[2]

[1]*Delft University of Technology, The Netherlands*
[2]*IBM T.J. Watson Research Ceter, NY, U.S.A.*
[3]*Columbia University, NY, U.S.A.*
[4]*IBM China Research Lab, Beijing, China*

SUMMARY

Applications running on multicore platforms are difficult to program, and even more difficult to optimize, mainly due to (1) the several layers where the optimizations occur and (2) the multitude of available resources to be exploited in parallel. Although low-level optimizations only target code running on individual cores, high-level optimizations (e.g. data- and task-parallelism) target the overall application performance. In this paper, we focus on the latter, by evaluating possible mapping scenarios of a real application on a heterogeneous multicore processor. Specifically, we focus on analyzing the impact of combining data- and task-parallelism for a multimedia analysis application running on the Cell Broadband Engine (Cell/B.E.). We find that both low-level and high-level optimizations are important for the overall application speed-up. However, we show that a speed-up factor of over 20 for the application running on Cell/B.E. can only be obtained if core utilization is increased by combining data- and task-parallelism. Thus, we consider this case study essential for building expertise in both application optimization and performance analysis for multicore platforms. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: multicore processors; multiprocessor system-on-chip; performance evaluation; Cell/B.E.; mapping and scheduling parallel applications

*Correspondence to: Ana Lucia Varbanescu, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands.
†E-mail: A.L.Varbanescu@tudelft.nl
‡The work presented here has been mostly done at IBM TJ Watson Research Center, U.S.A., and it is partly supported by the Scalp project `http://scalp.ewi.tudelft.nl` funded by STW-Progress, The Netherlands.

1. INTRODUCTION

Evaluating and optimizing the performance of multicore systems are major concerns for both hardware and software designers. The complexity of multicore architectures and the variety of their potential applications have slowed down the development of extensive performance case studies for these platforms. Furthermore, the lack of standard benchmarking suites and the lack of meaningful metrics for comparison [1] have led to a rather sparse set of experiments, aimed at porting and optimizing individual applications [2,3].

In general, porting an application on a multicore platform has followed a simple three-step path: (1) split the applications into parallel tasks suitable—in terms of size and computation—to the available cores, (2) port the tasks on the specific cores, and (3) iteratively optimize each of these tasks for the specific core it is running on. These low-level, core-specific optimizations, although leading to high task performance gains, may have a limited impact in the overall application performance [4]. To alleviate this problem, an additional layer of application-level optimizations must focus on the optimal mapping of the application tasks on the platform's available cores.

In this paper, we present a performance case study for the Cell Broadband Engine (Cell/B.E.) processor. The Cell has one power-processing element (PPE) and eight synergistic-processing elements (SPEs), and it is considered revolutionary due to its architecture and its peak performance. Although well known as the core of the Playstation 3 (PS3) game console, Cell/B.E. is now extensively used for high-performance computing (HPC) applications. For example, previous experiments [2,5,6] have shown that applications can be optimized to impressive levels of performance by using mainly core-level optimizations. However, a thorough evaluation of the overall application performance is still missing.

This high-level parallelization forms the target of this paper. For our case study, we use MarCell, a multimedia analysis application, already ported on the Cell/B.E. [7]. We show that by combining data- and task-parallelism, we can further increase the overall application performance because we are able to increase core utilization. One of our results is a short list of generic guidelines for efficient

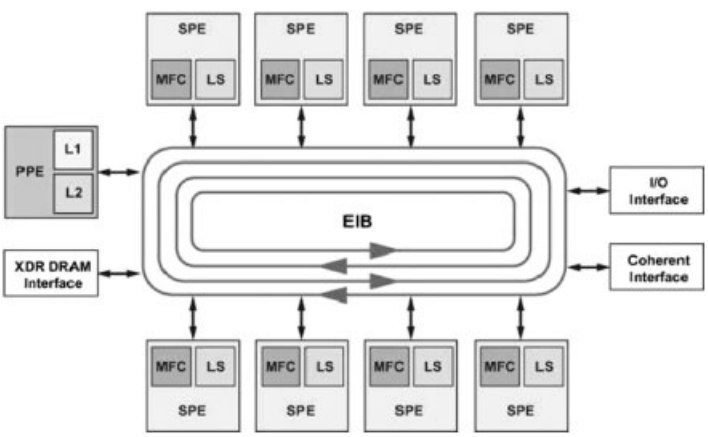


Figure 1. The Cell Broadband Engine.

DOI: 10.1002/cpe

application parallelization on this particular multicore processor. We claim that such guidelines, extracted from case-study applications, provide fundamental knowledge for any attempts towards an automated parallelization compiler or a performance predictor/estimator for multicore processors.

In the remainder of the paper, we first introduce the Cell/B.E. processor in Section 2 and our MarCell application in Section 3. In our previous work, we showed how successful the port of the application was on Cell/B.E. and the level of achieved performance [4,7]. Still, due to low overall core utilization, there is still room for significant performance improvement. Thus, we present several scenarios for combining data- and task-parallelism for MarCell, and we show the impact of these scenarios on the overall application performance in Section 4. In Section 5, we discuss how a Cell/B.E. application has to be built to enable mapping scenarios construction and evaluation. We provide relevant pointers to related work in Section 6 and we summarize our findings in Section 7.

2. CELL BROADBAND ENGINE

The Cell/B.E. is a heterogeneous multicore processor, featuring the main core of the PS3 game console. Nevertheless, its complex architecture and its impressive announced peak performance recommend it as a good target platform for multicore programming experiments.

A block diagram of the Cell processor is presented in Figure 1. Cell has nine cores: the PPE, acting as a main processor, and eight SPEs, acting as co-processors. These cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [5] to real-time ray tracing [6] or bioinformatics [3].

The PPE contains the power-processing unit (PPU), a 64-bit PowerPC core with a VMX unit, separated L1 caches (32 kB for data and 32 kB for instructions), and 512 kB of L2 Cache. Its main role is to run the operating system and coordinate the SPEs. An SPE contains a RISC-core (the SPU), a 256 kB local storage (LS), used as local memory for both code and data and managed entirely by the application, and a memory flow controller. All SPU instructions are 128-bit SIMD instructions, and all the 128 SPU registers are 128-bit wide. All processing elements are connected by a high-speed high-bandwidth element interconnection bus (EIB), together with the main system memory, and the external I/O. The maximum data bandwidth of the EIB has a theoretical peak at 204.8 GB s^{-1} [8].

Cell programming is based on a simple multi-threading model: the PPE spawns threads that execute asynchronously on SPEs, until interaction and/or synchronization is required. The SPEs can communicate with the PPE with simple mechanisms such as signals and mailboxes for small amounts of data, or DMA transfers via the main memory for larger data.

For the work presented in this paper, we have run our experiments on two platforms: the Playstation3, a game console, and the Q20 Cell-blade, a platform targeted towards HPC and provided for remote access by IBM. PS3 has one Cell/B.E. processor, running at 3.2 GHz, with six out of the eight SPEs fully available for programming; the Q20 blade has two Cell/B.E. processors, allowing uniform access to 16 SPEs, allowing for better scalability experiments. Both machines had Fedora Core 6 Linux and IBM's SDK2.1 installed for development and testing purposes.

DOI: 10.1002/cpe

3. MARCELL

In this section, we briefly present MarCell, the application we have used for our performance case study.

3.1. From MARVEL to MarCell

By multimedia content analysis, one aims to detect the semantic meanings of a multimedia document [9], be it a picture, a video, and/or audio sequence. Given the fast-paced increase in available multimedia content—from personal photo collections to news archives—automated mechanisms for content analysis and retrieval must be developed; for such applications, execution speed and accuracy are the most important performance metrics.

An example of an automated system for multimedia analysis and retrieval is MARVEL, developed by IBM Research. MARVEL uses multi-modal machine learning techniques in an initial training phase, generating a specific set of models. The models are further used to automatically annotate multimedia content, thus allowing for later searching and retrieval of the content of interest. The main goal of MARVEL is to make organizing large multimedia collections much more efficient, thus requiring very fast and accurate processing.

MarCell is the Cell/B.E. version of MARVEL [7]. In the porting process, the focus was placed on the computation-intensive part of the original application, namely the image classification portion. The processing flow of this engine, presented in Figure 2, consists of a series of feature extraction filters performed on each image in the available collection, followed by an evaluation of these features against the precomputed models. In MarCell, each of the filters, as well as the concept detection, runs on the SPEs. To allow this migration, the C++ code from MARVEL had to be ported to plain C code, and then aggressively optimized for the SPE cores. However, due to our stub-based strategy, the overall C++ application structure, functionality, and data flow are completely

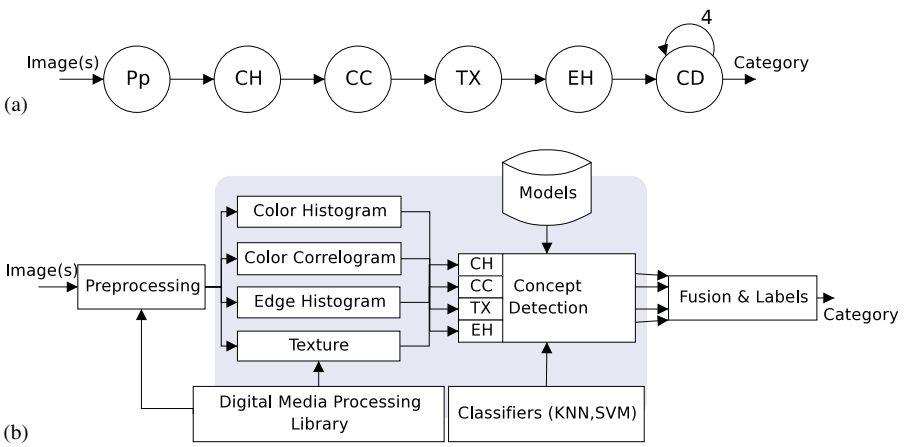


Figure 2. The processing flow of MarCell: (a) the sequential flow of the original application and (b) the data flow diagram. The shaded part is executed on the SPEs.

DOI: 10.1002/cpe

Table I. The speed-up factors of the SPE-enabled version of MarCell over the original MARVEL version.

Kernel	SPE (ms)	Speed-up vs PPE	Speed-up vs desktop	Speed-up vs laptop	Overall contribution (%)
App start	7.17	0.95	0.67	0.83	8
CH Extract	0.82	52.22	21.00	30.17	8
CC Extract	5.87	55.44	21.26	22.45	54
TX Extract	2.01	15.56	7.08	8.04	6
EH Extract	2.48	91.05	18.79	30.85	28
ConceptDet	0.41	7.15	3.75	4.88	2

preserved [4]. The preprocessing step (`AppStart`) includes the image reading, decompressing, and storing in the main memory (as an RGB image). The four feature extraction techniques that have been entirely ported to run on the SPEs are color histogram (`CHExtract`) [10] and color correlogram (`CCExtract`) [11] for color characteristics, edge histogram (`EHExtract`) [12] for contour information, and quadrature mirror filters for texture information (`TXExtract`) [13]. The concept detection is based on an SVM algorithm, which computes a final classification coefficient using a series of weighted dot-products [14]. From the resulting coefficients, the image can be properly categorized according to the given models.

3.2. MarCell: first performance measurements

After the application was divided into potential parallel tasks, all five kernels are implemented in C. Furthermore, we have performed manual core-level optimizations (e.g. code SIMDization, branch elimination, loop unrolling and/or interchanging, etc.), DMA multi-buffering, and even slight algorithm tuning has been performed on each of these kernels. To have a first idea of the optimized kernels performance, we have compared their execution times on Cell/B.E. with those obtained by the original kernels running on the PPE, as well as on two reference commodity systems[§]: a desktop machine with a Pentium D processor (dual-core, running at 3.4 GHz), from now on called 'Desktop', and a laptop with a Pentium Centrino (running at 1.8 GHz), from now on called 'Laptop.' The performance numbers are presented in Table I.

Further core-level optimization for any of these kernels is beyond the purpose of this paper. Thus, in the remaining sections, we considered these kernel performance numbers as the reference ones, and use them just to compose and evaluate the scheduling scenarios for the complete application.

4. PARALLELIZATION SCENARIOS

In this section, we explore three types of scenarios for optimizing a parallel version of MarCell, by combining data- and task-parallelism at different granularities. We present our experiments and we comment on their results.

[§]Note that the reference machines have only been chosen as such for convenience reasons; there are *no* optimizations performed for any of the Pentium processors.

DOI: 10.1002/cpe

4.1. Task-parallelism

For the first version of MarCell, we have preserved the sequential nature of the original MARVEL. In this first scenario (called `SPU_seq` from now on), we have statically assigned each one of the five kernels to a dedicated SPE. The main application, running on the PPE, executes them *sequentially*, in the same order as the original, general-purpose MARVEL, schematically presented in Figure 3(a). By *sequential*, we mean there is no task-parallelism between the SPEs, as the PPE is firing only one SPE kernel at a time, waiting for its completion before firing the next (see Figure 3(b)).

Once the code is assigned to the SPEs, the PPE can enable task-parallelism by firing all four feature extraction kernels in parallel, waiting for them to complete, and then run the concept detection. Note that the concept detection for each feature cannot start before the feature extraction is completed. To preserve this data dependency, we consider three mapping options for the concept detection kernel (also seen in Figure 4): `SPU_par_5` runs the kernel sequentially, using only one additional SPE (five in total), or runs the CD kernel in parallel, either by using four separate SPEs (eight in total) in `SPU_par_8`, or combining each feature extraction kernel with its own concept detection, using no additional SPEs (four in total) in `SPU_par_4`. In general, to implement the task-parallel

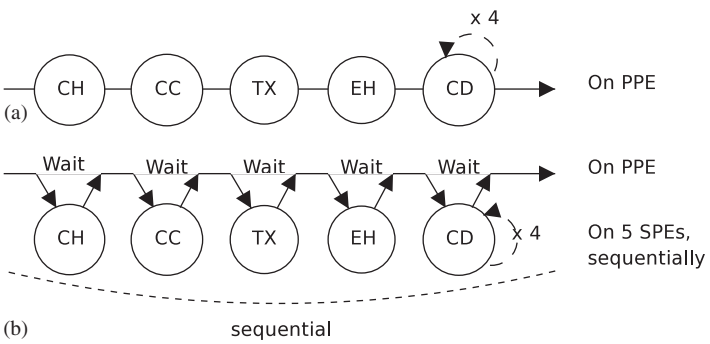


Figure 3. Task-parallelism: (a) the original application, with all kernels running on the PPE and (b) all kernels are running on SPEs, but they are fired sequentially.

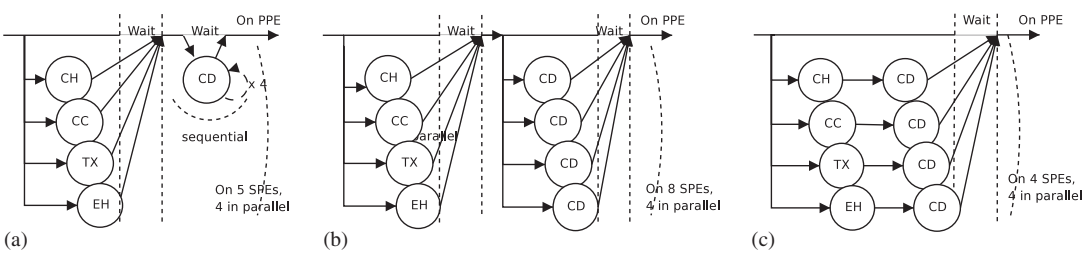


Figure 4. Task-parallelism scenarios: (a) SPU_par_5: CD running on one SPE; (b) SPU_par_8: CD running on four separate SPEs; and (c) SPU_par_4: CD is combined with feature extraction kernels, running on the same four SPEs.

DOI: 10.1002/cpe

behavior, the PPE code is modified such that (1) the calls to the SPE kernels become non-blocking, (2) the busy-waiting on an SPE's mailbox is replaced by an event-handler, and (3) eventual data consistency issues are solved by synchronization. Overall, these are minor modifications.

We have implemented all three task-parallel versions of MarCell and measured their execution times on both the PS3 and the Cell-blade. Because the PS3 has only six SPEs available, the implementation of the `SPU_par_8` requires thread reconfiguration for at least two of the four SPEs involved. Due to the thread-switching overhead, the application execution time increases by 20%. Using the Cell-blade, the same scenario runs using dedicated SPEs for each kernel, thus the execution time is significantly lower. Also note that combining the concept detection with each of the feature extraction kernels caused a slight decrease in the performance of the extraction kernels[¶].

To evaluate the performance of the task-parallel implementations, we have compared the `SPU_par` scenarios with the `SPU_seq` scenario and with the original sequential application running on the PPE and on both reference systems. We have performed experiments on a set of 100 images, 352×240 pixels each, and we report the results as the average over the entire set. Figure 5 presents the speed-up results of all these scenarios, whereas Figure 6 shows the corresponding platform core utilization. For reference purposes, we have normalized the speed-up with the execution time of the Desktop machine (the fastest execution of the original MARVEL code). The speed-up of the parallel version is smaller than expected because the PPE waits for all feature extraction kernels to finish, thus being limited by the speed-up of the slowest one (in this case, the color correlogram kernel, which represents more than 50% of the execution time). The performance differences between the three `SPU_par` scenarios are small due to the very low contribution of the concept detection kernel in the complete application.

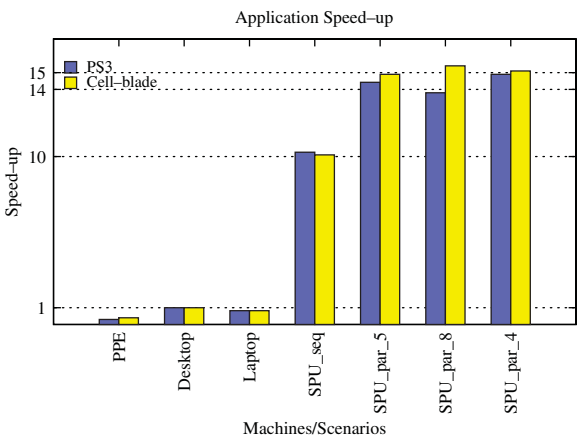


Figure 5. Speed-up results of the task-level scenarios of MarCell.

[¶]To maximize the performance for a given kernel, the programmer uses most of the non-code LS space for DMA operations; once a new piece of code is added, together with its own memory requirements, this DMA slack space is reduced, thus more DMA operations are required; this can significantly impact the kernel performance.

DOI: 10.1002/cpe

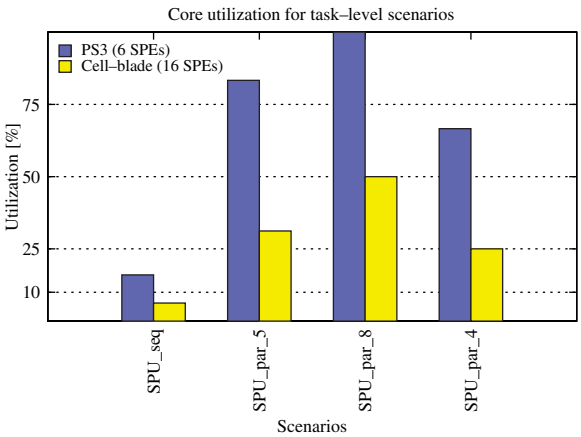


Figure 6. Overall platform utilization of the task-level scenarios of MarCell.

4.2. Data-parallelism

As seen before, task-parallelism is often too coarse grained to achieve good resource utilization, thus leading to poor overall application performance. Attempting to increase the core utilization on the available Cell/B.E. platforms, we have implemented the data-parallel version of the application. In this case, each one of the kernels was parallelized to run over a given number of SPEs, performing computations on slices of the input image. The merging of the results is done by the PPE, such that the communication overhead induced by the SPE–PPE communication (i.e. the number of DMA operations, in the case of Cell/B.E.) is not increased.

We have measured the data-parallel versions of the kernels running individually on the Cell/B.E. The results show almost linear speed-ups for both `TXExtract` and `CCEXtract` for up to eight kernels. This limitation is due to the input picture size—a data set with larger pictures would move this threshold higher. The `CHExtract` kernel fails to provide better performance with the number of SPEs it uses due to its small computation-to-data transfer ratio. Running on image slices, one kernel may execute faster due to less computation *and* smaller communication, a combination that can lead to slightly superlinear speed-up, as in the case of the `EHExtract`; however, even in this case, `EH_Extract` only obtains speed-up on up to seven SPEs.

When composing the entire application as a sequence of data-parallel kernels, as in Figure 7, the overall performance is highly influenced by kernel reloading/reconfiguration, which must happen after each SPE kernel execution, on all available SPEs. There are two options for implementing this switching:

- Thread re-creation, which can accommodate kernels with bigger memory footprints, at the price of a higher execution time; the solution is not suitable for applications with low computation-to-communication ratios. In the case of MarCell, for more than two SPEs, the thread switching becomes dominant in the application performance.
- Transparent overlays, which combine several SPE programs in an overlayed scheme; the reconfiguration is done transparently to the user, who invokes the kernel like it would already

DOI: 10.1002/cpe

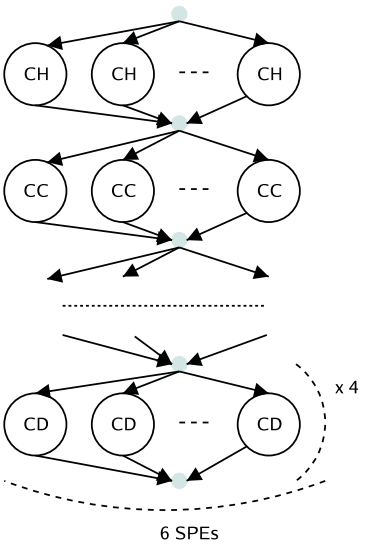


Figure 7. Data-parallel MarCell: each kernel is running over multiple SPEs, but no different kernels run in parallel.

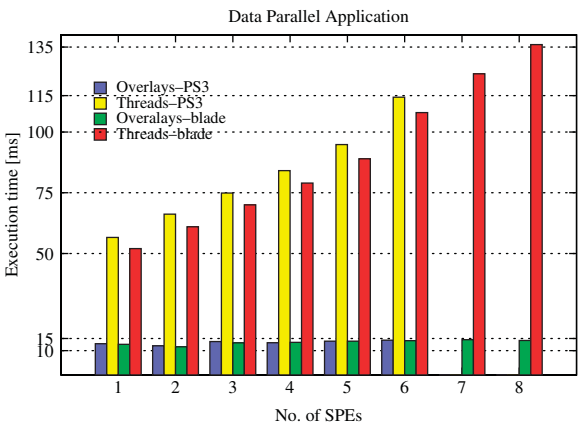


Figure 8. The overall performance of the data-parallel application.

be loaded in the SPE memory. When the kernel is not available, it is automatically loaded in the SPE local store via DMA, replacing the previously available kernel(s). The use of overlays leads to a more efficient context switching, but the combination of the SPEs codes in the overlayed scheme requires additional size tuning for data structures and/or code.

Figure 8 presents the application speed-up with the number of SPEs for the two proposed context switching mechanisms, measured on both the PS3 and the Cell-blade. We have only showed the

DOI: 10.1002/cpe

measurements up to eight SPEs because no significant improvement of the speed-up is achieved on more than seven SPEs. Furthermore, although the data-parallelized kernels have linear performance with the number of used SPEs, the relationship does not hold for the overall application due to the very expensive context switching. Note that the performance of the thread re-starting implementation worsens much faster because the number of reconfigurations is linearly growing with the number of used SPEs.

From the experimental results, we conclude that data parallelization on Cell/B.E. is not efficient on its own for applications that require the kernels code to be switched too often. Furthermore, any efficient data-parallel implementation for an SPE kernel should allow dynamic adjustment of both its local data structures (i.e. LS memory footprint) and its input data size (i.e. the image slice size): when less LS memory may be needed for a slice, more space for DMA-transferred data can be available, thus lowering the transfer overheads.

4.3. Combining data- and task-parallelism

Next, we have evaluated the solutions to combine data-parallel kernels (i.e. running over multiple SPEs) and task-parallelism (i.e. multiple feature extractions running in parallel). As mentioned earlier, the concept detection can be run only *after* the completion of the corresponding feature extraction, as it requires the complete feature vector (i.e. all slices) for its computation. For a complete set of experiments, we have measured the performance of all the possible static mappings. A static mapping includes no kernel reconfiguration; thus, one SPE is running only one feature extraction kernel, followed by a concept detection operation for the same kernel (i.e. the concept detection is parallelized on the same number of SPEs as the feature extraction kernel).

The results of these measurements on the PS3 platform are presented in Figure 9[‖]. For the PS3, the best mapping, with $T_{ex} = 9.58$, is `1-3-1-1`, which is an intuitive result given the significant percentage of the overall application time spent on the `CCExtract` kernel. We have repeated the

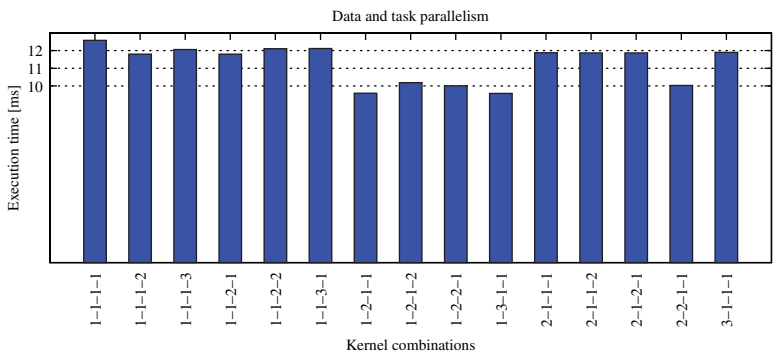


Figure 9. Using combined parallelism for mapping the application on PS3.

[‖]The notations of the scenarios has the form `x-y-z-w`, coding the number of SPEs used for each feature extraction, in the order CH,CC,TX,EH.

DOI: 10.1002/cpe

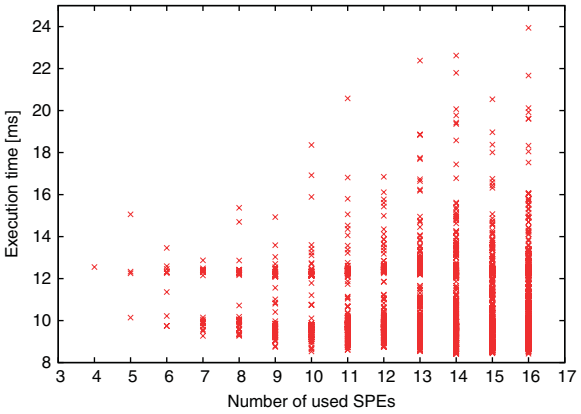


Figure 10. Using combined parallelism for mapping the application on the Cell-blade.

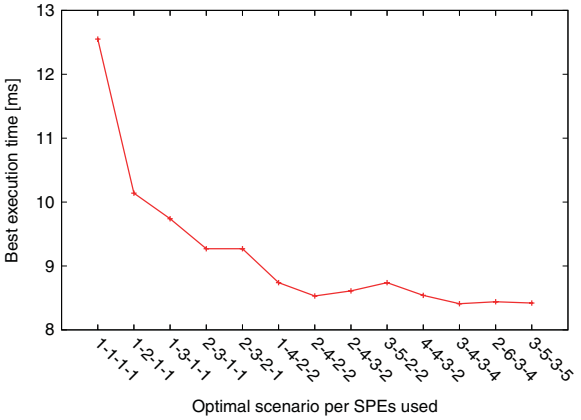


Figure 11. Best scenario for a given number of utilized SPEs.

same type of experiments on the QS20 blade. Figure 10 presents the clustering of the execution times for a specific number or used SPEs (from 4 to 16); Figure 11 shows the best scenario when considering the number of utilized SPEs varying between 4 and 16. Note that the best mapping, with $T_{ex} = 8.41$ ms, is `3-4-3-4`. Also note that this mapping is *not* utilizing all available SPEs, which shows that for the given input set, the application runs most efficiently on 14 SPEs, whereas a very good ratio performance/utilized cores is already obtained by `2-4-2-2`, using only 10 SPEs.

Given the overall application performance, we conclude that combining data- and task-parallelism on Cell is the best way to gain maximum performance for applications with multiple kernels, as seen in Figure 12. This significant performance improvement is mainly due to the increase in overall core utilization for the entire platform.

DOI: 10.1002/cpe

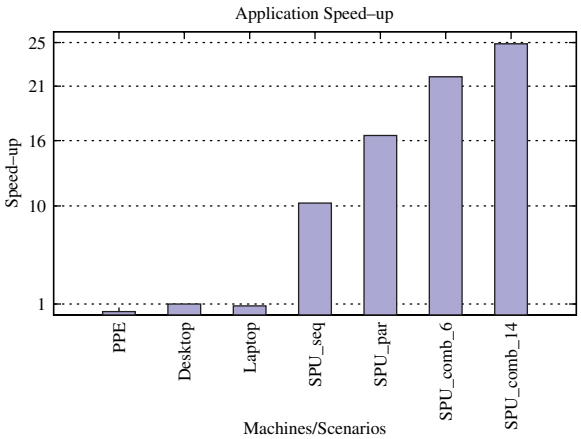


Figure 12. Maximized application speed-up for each of the presented scenarios.

5. DISCUSSION

In this section, we discuss how generic our parallelization approach is. We present both MarCell's and Cell's relevance in the current multicore era and we list our generic guidelines for enabling high-level performance tuning on the Cell.

5.1. MarCell relevance

The code base for Cell/B.E. performance analysis is currently a sum of various ported applications, such as RAxML [3], Sweep3D [2], real-time raytracing [6], CellSort [15], digital signal processing [16], MarCell, etc. Although each one of them has been analyzed individually, they can be thought of as representatives for larger classes of potential Cell applications. Because there are no benchmark suites for heterogeneous multicores (yet), the performance analysis and results obtained by these applications provide a better understanding of the Cell's performance characteristics.

MarCell is the first multimedia analysis application running on the Cell/B.E.. It is a valid use-case for multicores, because its processing performance influences directly its accuracy. Further, MarCell provides a series of practical advantages for a mapping study on such a platform. First, the task-parallel application has a low number of *non-symmetrical* tasks. Each of these tasks is highly optimized, and the difference between the high individual speed-ups and the relatively low overall application speed-up makes the mapping investigation worthwhile. The low data dependency between the tasks allows the evaluation of several scheduling scenarios.

MarCell is an example of a series–parallel application [17]. On a platform such as Cell/B.E., for *any* SP-compliant application, the simplest way to increase core utilization (and, as a result, to gain performance) is to combine data- and task-parallelism. For more data-dependent applications, there are still two options: to parallelize them in an SP form or to use more dynamic parallelization strategies, such as job-scheduling.

DOI: 10.1002/cpe

5.2. Cell/B.E. relevance

Combining data- and task-parallelism is not a new idea in parallel computing—it has been investigated before for large distributed systems (e.g. [18]). However, the scale differences between multi-core processors and such large multi-processors are too big with respect to both the application components and the hardware resources. Thus, previous results cannot be directly applied. A new, multi-core specific investigation is required to establish (1) if such a combination is possible on these platforms, (2) how large the programming effort is, and (3) if it has any significant performance impact.

We have chosen the Cell/B.E. as a target platform able to answer all these questions. None of the immediately available alternatives is suitable for such an investigation: GPU architectures are massively data-parallel, embedded multi-cores are very limited in resources, and generic-purpose multi-cores are (so far) only homogeneous. Even further, being a high-performance processor, the Cell/B.E. will continue to be the target for many computation-intensive applications [19]. Based on our results, it is evident that all these applications must take into account the mapping factor as a potential performance booster.

5.3. Cell/B.E. parallelization guidelines

What makes programming on the Cell/B.E. difficult is the complexity of the architecture combined with the five parallelization layers the programmer needs to exploit. In many cases, the choice is guided only by trial and error or by experience. We are able to synthesize our experience in four empirical guidelines. Following them should enable any application parallelization for the Cell/B.E. to efficiently use, combine, and optimize data- and task-parallelism.

1. Focus on the problem parallelism, not on the architecture resources: the application has to be explicitly decomposed down to the level of its *non-parallel units*. Such a unit, characterized by *both* its computation and its data, can be scheduled in parallel or in sequence with other units, but never further decomposed. The algorithm is then built as a scheduling graph of these units. For MarCell, an example is `CHExtract(slice)`.
2. Optimize all SPE code with low-level techniques. Ideally, this step is done by the compiler, but currently there are still many optimizations (e.g. SIMD-ization, DMA multi-buffering) to be performed by hand. This step typically leads to a significant performance gain, but it is also the most time-consuming optimization step. For MarCell, it took 2.5 person-months for the five kernels.
3. Take the scheduling decisions only on the PPE, distributing work and data either to maximize a static metric (for MarCell, we have chosen core utilization) or following a dynamic strategy (FCFS, round-robin, etc.). The PPE should map data-dependent kernels on the same core, to minimize data copying.
4. Build and evaluate all possible mapping scenarios, eventually adding particular constraints (such as, for example, use the minimum number of SPE cores). To evaluate performance, one can easily instantiate and execute all these scenarios. A better alternative is to (easily) build an automated tool to generate all possible mappings and calculate their execution times using a generalized form of Amdhal's law [4].

DOI: 10.1002/cpe

6. RELATED WORK

Most of the performance evaluation for applications running on Cell aims to show the very high level of performance that the architecture can actually achieve, by combining mostly optimizations on the SPE side [20]. For example, the first application that emphasized the performance potential of Cell/B.E. has been presented by IBM in [21], where a complete implementation of a terrain rendering engine is shown to perform 50 times faster on a 3.2 GHz Cell machine than on a 2.0 GHz PowerPC G5 VMX. Besides the impressive performance numbers, the authors also present a very interesting overview of the implementation process and task scheduling, but no alternatives are given. Another interesting performance-tuning experiment is presented in [6], where the authors discuss a Cell-based ray tracing algorithm. In particular, this work looks at more structural changes of the application and evolves towards finding the most suitable ray tracing algorithm on Cell. Finally, the lessons learned from porting Sweep3D (a high-performance scientific kernel) on the Cell processor, presented in [2], were very useful for developing our strategy, as they pointed out the key optimization points that an application has to exploit for significant performance gains. Again, none of these two performance-oriented paper look at the coarser grain details of resource usage.

Although core-level optimizations lead to very good results [7], the overall application performance is many times hindered by the mapping and/or scheduling of these kernels on the Cell/B.E. platform resources [4]. A similar intuition with the one we had when optimizing MarCell forms the main claim of the work presented in [3,22]. Here, the authors take a task-parallel application and try to change the granularity of the parallelization on-the-fly, based either on previous runs or on runtime predictions. The results of this multi-grain parallelization are similar to ours, showing how combining two different approaches significantly increases application performance. Still, due to the specific applications used as case studies, the granularity at which the scheduling decisions are taken is significantly different.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a performance case study of running multi-kernel applications on Cell/B.E. For our experiments, we have used the MarCell, a multimedia analysis and retrieval application, which has five computational kernels, each one of them ported and optimized for an SPE. Our measurements show that combining these core-level optimizations leads to an overall application speed-up of more than 10, even when using the SPEs sequentially. Furthermore, we have tested task and data parallelization of the application, by either firing different tasks in parallel (MPMD) or by using all available SPEs to compute one kernel faster (SPMD). The results showed how task parallelization leads to a speed-up of over 15, whereas the data parallelization worsens performance due to a high SPE reconfiguration rate. Last but not least, we have shown how various scheduling scenarios for these kernels can significantly increase the overall application speed-up. In order to detect the best scenario, we have performed a large number of experiments, evaluating all possible schedulings without SPE reprogramming. For our two Cell-based platforms, we have found the best speed-up factors to be over 21 for the PS3 (6 SPEs available) and over 24 for the Cell-blade (16 SPEs available).

DOI: 10.1002/cpe

We have drawn three important conclusions. First, we have not only proved how task-parallelism can be a good source of speed-up for independent kernels, but we have also shown how a significant imbalance in the execution times of some of these kernels can alter overall performance. Second, we have shown how data-parallelism is, in most cases, inefficient due to the very high overhead of SPE reprogramming. Finally, we have shown how an application combining the two approaches—data- and task-parallelism—can gain the advantages of the two; however, this reconfigurability potential does come at the price of programming effort, because all kernels need to be programmed to accept variable I/O data sizes and local structures, and the main thread needs to be able to run various SP combinations of these kernels.

ACKNOWLEDGEMENTS

We would like to thank Michael Perrone, Gordon Braudaway, Karen Magerlein, and Bruce D'Amora for their valuable support and ideas during the development and experiments of this application.

REFERENCES

1. Intel. Measuring application performance on multi-core hardware. *Technical Report*, Intel, September 2007.
2. Petrini F, Fernàndez J, Kistler M, Fossum G, Varbanescu AL, Perrone M. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. *IPDPS 2007*, Long Beach, CA. IEEE/ACM: New York, March 2007.
3. Blagojevic F, Stamatakis A, Antonopoulos C, Nikolopoulos DS. RAxML-CELL: Parallel phylogenetic tree construction on the Cell Broadband Engine. *IPDPS 2007*, Long Beach, CA. IEEE/ACM: New York, March 2007.
4. Varbanescu AL, Sips H, Ross KA, Liu Q, Liu L-K, (Paul) Natsev A, Smith JR. An effective strategy for porting C++ applications on Cell. *ICPP'07*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2007; 59–68.
5. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K. The potential of the Cell processor for scientific computing. *ACM Computing Frontiers '06*. ACM Press: New York, 2006; 9–20.
6. Benthin C, Wald I, Scherbaum M, Friedrich H. Ray tracing on the Cell processor. *IEEE Symposium on Interactive Ray Tracing 2006*, Salt Lake City, Utah, U.S.A., September 2006; 15–23.
7. Liu L-K, Liu Q, (Paul) Natsev A, Ross KA, Smith JR, Varbanescu AL. Digital media indexing on the Cell processor. *ICME 2007*, Beijing, China, July 2007.
8. Kistler M, Perrone M, Petrini F. Cell multiprocessor communication network: Built for speed. *IEEE Micro* 2006; **26**(3):10–23.
9. Wang Y, Liu Z, Huang J-C. Multimedia content analysis using both audio and visual clues. *IEEE Signal Processing Magazine* 2000; **17**(6):12–36.
10. Smith JR, Chang S-F. Tools and techniques for color image retrieval. *SPIE '96*, San Jose, CA, U.S.A., vol. 2670, 1996.
11. Huang J, Ravi Kumar S, Mitra M, Zhu W-J, Zabih R. Image indexing using color correlograms. *CVPR '97*. IEEE Computer Society: Washington, DC, U.S.A., 1997; 762.
12. Fisher B, Perkins S, Walker A, Wolfart E. Hypermedia image processing reference. http://www.cee.hw.ac.uk/hipr/html/hipr_top.html [1996].
13. Naphade MR, Lin C-Y, Smith JR. Video texture indexing using spatio-temporal wavelets. *ICIP*, Rochester, NY, U.S.A., vol. 2, 2002; 437–440.
14. Naphade MR, Smith JR. A hybrid framework for detecting the semantics of concepts and context. *CIVR*, Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL, U.S.A., 2003; 196–205.
15. Gedik B, Bordawekar RR, Yu PS. CellSort: High performance sorting on the Cell processor. *VLDB '07*, VLDB Endowment, Vienna, Austria, 2007; 1286–1297.
16. Bader DA, Agarwal V. FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. *HiPC*, Goa, India, 2007; 172–184.
17. Gonzalez-Escribano A. Synchronization architecture in parallel programming models. *PhD Thesis*, Department of Informática, University of Valladolid, 2003.
18. Uk B, Taufer M, Stricker T, Settanni G, Cavalli A, Caflisch A. Combining task- and data parallelism to speed up protein folding on a desktop grid platform. *CCGRID*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2003; 240.

19. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K. The potential of the cell processor for scientific computing. *ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2006.
20. Gschwind M. Chip multiprocessing and the Cell Broadband Engine. *CF '06*: *Proceedings of the 3rd Conference on Computing Frontiers*. ACM Press: New York, NY, U.S.A., 2006; 1–8.
21. Minor B, Fossum G, To V. Terrain rendering engine (white paper). http://www.research.ibm.com/cell/whitepapers/TRE.pdf [May 2005].
22. Blagojevic F, Feng X, Cameron K, Nikolopoulos DS. Modeling multi-grain parallelism on heterogeneous multicore processors: A case study of the Cell BE. *HiPEAC'08*, Göteborg, Sweden, 2008.

DOI: 10.1002/cpe