An Effective Strategy for Porting C++ Applications on Cell

Ana Lucia Varbanescu, Henk Sips
Delft University of Technology
{A.L.Varbanescu,H.J.Sips}@tudelft.nl

Kenneth A. Ross
Columbia University, NY
kar@cs.columbia.edu

Qiang Liu
IBM CRL, Beijing
qiangliu@cn.ibm.com

Lurng-Kuo Liu, Apostol (Paul) Natsev, John R. Smith
IBM T.J. Watson Research Center, NY
{lkliu,jsmith,natsev}@us.ibm.com

Abstract

In this paper we present a solution for efficient porting of sequential C++ applications on the Cell B.E. processor. We present our step-by-step approach, focusing on its generality, we provide a set of code templates and optimization guidelines to support the porting, and we include a set of equations to estimate the performance gain of the new application. As a case-study, we show the use of our solution on a multimedia content analysis application, named MARVEL. The results of our experiments with MARVEL prove the significant performance increase in favor of the application running on Cell when compared with the reference implementation.

Keywords: Cell BE processor, multi-core, MPSoC, parallelization, porting technique, C++ applications

1. Introduction

Within the last five years, the notion of multiprocessor-system-on-chip (MPSoC) has transformed from paper-based block diagrams to prototypes, and, for very few cases, even to real hardware. This evolution is spectacular not only in its very fast pace, but also in the mindset changes that it involves. Processor architects have shifted their focus from increasing processor frequency to increasing processor functionality. As a result, processors are no longer based on a single very fast, very complex core, but rather built-up from multiple, simpler cores, that can achieve higher processing power by working in parallel. While single-core processors aim to exploit instruction level parallelism in an user-transparent fashion, multi-core processors allow (also) user-assisted parallelization at higher levels.

While the hardware community grasped the MPSoC ideas quite quickly, the software community is somewhat more skeptical. The complexity of these platforms, together with the increased responsibility that has moved from hardware into software still raises eyebrows and questions about the efficient use of such architectures. Simply put, it is not yet clear how can these machines be programmed so that they gain performance without wasting (too much) effort, and it is also not yet clear what applications can really benefit from the increased on-chip functionality. It is fair to say that the software community is still going through the learning phase, exploring applications and architectures combinations in order to gather enough expertise to infer disciplined programming models and automated programming tools.

One of the most complex existing multi-core processors is the Cell Broadband Engine (Cell B.E.), a heterogeneous architecture developed by a consortium of three industrial partners: Sony, Toshiba, and IBM. The architecture features nine cores: one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs), connected by a fast high-bandwidth bus. Typically, the PPE runs as a main processor, while the SPEs act as application-specific accelerators. All cores share the same main memory, but the SPEs have direct access only to their Local Storage (LS) and can access the main memory only via DMA. Currently, programming the Cell B.E. is considered difficult, but worth the effort, as various applications have seen spectacular performance gains when ported on this machine.

In this context, this paper presents an application independent strategy to port large, cycle-hungry applications on the Cell B.E. Using this strategy, one can quickly enable an application to run on Cell, following a simple partitioning scheme to enable task-level parallelism. The tasks are to be removed from the main application, ported for the SPEs, and plugged-in while preserving the overall flow of the application. The code templates to enable the plug-in mechanism are presented in the paper. Due to this modular solution, SPE tasks can be independently enhanced towards near-optimal performance. While the strategy itself cannot

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

guarantee performance gain, we include a simple sanity-check equation that, given the tasks performance gains and scheduling, allows a quick estimate of the overall application speed-up. Because the effective level of achieved performance is highly application dependent, we acknowledge that our generic scheme can limit the potential of specific applications, but we argue that, due to its simplicity and effectiveness, it provides a good starting point for more sophisticated techniques.

As a proof-of-concept, we have chosen to apply our strategy on a multimedia content analysis and retrieval application, named MARVEL. In a nutshell, the application analyzes a series of images and classifies them by comparison against precomputed models. MARVEL is a good candidate for a case-study, because it is a large sequential C++ application (more than 50 classes, over 20.000 lines of code) and it performs computation intensive operations, like image features extraction and image classification. The experiments we have conducted by running MARVEL on Cell show that our strategy works for large applications, and it leads to good overall speed-up, gained with a moderate programming effort.

The remainder of the paper is organized as follows. Section 2 introduces the Cell B.E. processor, explaining its main features. Section 3 presents our porting approach, and Section 4 looks at overall application performance. Section 5 describes MARVEL, our porting experiments and their results. In Section 6 we have included several references to related work, while Section 7 concludes the paper and presents our future work directions.

2. Cell B.E.

The Cell Broadband Engine is a heterogeneous multicore processor, mainly known as featuring the Playstation 3 game console. Nevertheless, its complex structure and impressive estimated performance make it an interesting target platform for multicore programming experiments.

A block diagram of the Cell processor is presented in Figure 1. Cell has nine cores: the Power Processing Element (PPE), acting as a main processor, and eight Synergistic Processing Elements (SPEs), acting as co-processors. These cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [20] to image processing applications [3] and games [8].

The PPE contains the Power Processing Unit (PPU), a 64-bit PowerPC core with a VMX unit, separated L1 caches (32KB for data and 32KB for instructions), and 512KB 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 256KB Local Storage (LS), used as local memory for both code and data and managed entirely by the application/user, and a Memory Flow Controller (MFC). The

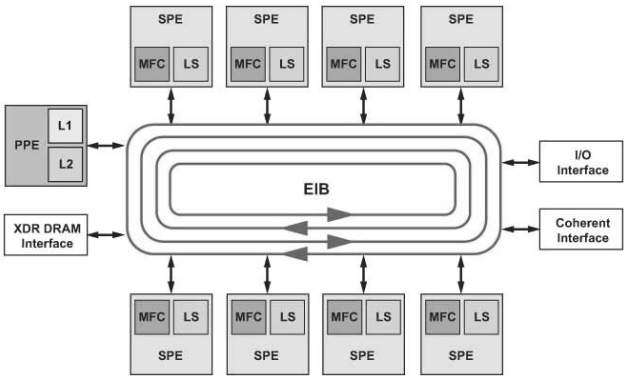

Figure 1. The Cell Broadband Engine

MFC contains separate modules for DMA, memory management, bus interfacing, and synchronization with other cores. All SPU instructions are 128-bit SIMD instructions, and all the 128 SPU registers are 128-bit wide. The single precision operations (integers and single floating point) are issued at a rate of 8, 16, or 32 operations per cycle (dual-pipeline, multi-way SIMD) for 8-bit, 16-bit and 32-bit numbers respectively. The double precision operations (64-bit floating point) are issued at the lower rate of two double-precision operations every seven SPU clock cycles. 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 [12].

Cell programming is based on a simple multi-threading model [6, 11]: 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 like signals and mailboxes for small amounts of data, or DMA transfers via the main memory for larger data. The major source of performance for Cell resides in exploiting its various parallelism layers: from task and data parallelism across multiple SPEs, to SPE code SIMDization, and multi-buffering for DMA transfers [9]. While applications written from scratch can natively exploit all these features, ported (sequential) applications have to systematically enable them.

3. The Porting Strategy

In this section we present in more detail our technique for allowing C++ applications to be ported on Cell. To keep the process efficient, we need to balance the porting effort and the performance gain. Thus, while we aim for the application to be able to use all the available parallelism layers provided by Cell, we make use of as much code as possible from the initial application. Furthermore, to evaluate the

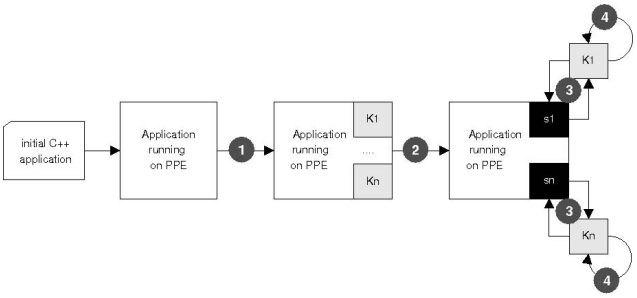

Figure 2. The porting strategy of a sequential C++ application to Cell BE

gradual improvements of the application, we need a mechanism that allows application partitioning and code replacement while preserving the application functionality all the way through the porting process (i.e., the porting does not destroy the overall data and control flow of the entire application).

The general strategy is presented in Figure 2. The application is first ported on the PPE. Next, its most compute-intensive kernels have to be identified, and further detached from the main application as potential candidates for execution on the SPEs. Each kernel will become a new SPE thread. In the main application, each kernel is replaced by a stub that will invoke the SPE-code and gather its results (step 2 in Figure 2). As a result of the separation, all former shared data between the application and kernel has to be replaced by DMA transfers (step 3 in Figure 2). Finally, the kernel code itself has to ported to C (to execute on the SPE) and further optimized (step 4 in Figure 2).

3.1. The PPE Version

A sequential C++ application running on Linux is, in most cases, directly compilable for execution on the PPE processor. The problems that appear in this phase are rather related to the application compile and build infrastructure than to the code itself; notable exceptions are the architecture specific optimizations (like, for example, i386 assembly instructions) which may need to be ported to the PowerPC architecture.

Once running on the PPE, the application performance is not impressive, being two to three times slower than the same application running on a current generation commodity machine. This result is not a surprise, as the PPE processing power is quite modest[12].

3.2. Kernels Identification

First level of parallelization requires an application partitioning such that the most time-consuming kernels are migrated for execution on the SPEs (step 1 in Figure 2). To make efficient use of the SPEs processing power, the kernels have to be small enough to fit in the local store, but large enough to provide some meaningful computation. Kernels are not necessarily one single method in the initial C++ application, but rather a cluster of methods that perform together one or more processing operations on the same data. However, this grouping should not cross class boundaries, due to potential data accessibility complications. Furthermore, kernels should avoid computation patterns that do not balance computation and communication, thus accessing the main memory (via the DMA engines) too often. Typically, SPEs use small compute kernels on large amounts of data, allowing the compute code to be "fixed" in the SPE memory, while the data is DMA-ed in and out.

To identify the kernels, the PPE application running is profiled (using standard tools like `gprof` or more advanced solutions like Xprofiler[1]), and the most "expensive" methods are extracted as candidate kernels. Based on the execution coverage numbers, we find the computation core of each kernel. Based on the the application call graph, each kernel may be enriched with additional methods that should be clustered around its computation core for reasons of convenient data communication and easy porting. Performance-wise, we state that the bigger the kernel is, the more significant the performance gain should be, but also the more difficult the porting becomes.

3.3. The `SPEInterface` stub

In order to preserve application functionality at all times, we have opted to implement the stub interface as an new class, named `SPEInterface`. The class manages the interface between the code running on the SPE and the main application running on the PPE, as presented in Figure 3. Basically, for each kernel, we instantiate an object of type `SPEInterface` and configure its parameters according to the kernel-specific requirements. Every such object will manage the communication between the application and one SPE kernel. Thus, the changes required on the main application side are:

- Instantiate the `SPEInterface` class to a new object, say `Kernel1Interface`.
- Wrap *all* the required member data of the original class into a common data structure, and preserve/enforce data alignment for future DMA operations[7].
- Allocate the output buffers for kernel results; typically, for simplicity, these buffers are also included in the data wrapper structure
- Communicate the address of this data structure to the kernel, which will fetch its required data via DMA.

[1] `http://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.xprofiler.html`

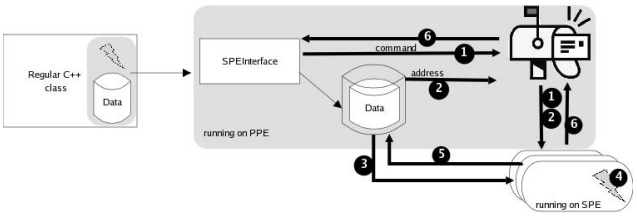

Figure 3. The SPEInterface. From the regular C++ code, the shaded data and methods are replaced by calls to an SPEInterface object

- Replace the call to the code implemented by the kernel with a call to either `Kernel1Interface->Send` or `Kernel1Interface->SendAndWait` methods, thus invoking the SPE code and waiting for the results.
- Use the results put by the kernel in the output buffers to fill in the corresponding class member structure

In order to avoid the high penalty of thread creation and destruction at every kernel invocation, our approach statically schedules the kernels to SPEs. Thus, once the PPE application instantiates the kernel interfaces, the SPEs are activated and kept in an idle state. When calling `Kernel1Interface->Send` or `Kernel1Interface->SendAndWait`, the application sends a command to the kernel, which starts execution. When execution is completed, the kernel resumes its waiting state for a new command.

3.4. Kernel migration

The algorithm to migrate the kernels on the SPEs is:

- Define data interfaces - the data required by the kernel from the main application must be identified and prepared for the DMA transfer, which may require (minor) data allocation changes in the main allocation, to enforce alignment.
- Establish a protocol and medium for short message communication between the kernel and the application - typically, this channel is based on the use of mailboxes or signals.
- Program the DMA transfers - the SPE local data structures must be "filled" with the data from the main application. Data is fetched by DMA transfers, programmed inside the kernel. For data structures larger than the SPE available memory (256KB for both data and code), iterative DMA transfers have to be interleaved with processing
- Port the code - the C++ code has to be transformed in C code; all the class member data references have to be replaced by local data structures.
- For large data structures, the kernel code must be adapted to run correctly in an incremental manner, i.e., being provided with on slices of data instead of the complete structures.
- Implement the function dispatcher (i.e., the kernel idle mode) in the `main` function of each kernel, allowing the kernel component functions to be executed independently, as dictated by the main application commands

For example (more to come in Section 5), consider an image filter running on an 1600x1200 RGB image, which does not fit in the SPE memory, so the DMA transfer must be done in slices. This means that the filter itself only has access to a slice of data, not to the entire image. For a color conversion filter, when the new pixel is a function of the old pixel only, the processing requires no changes. However, for a convolution filter[2], the data slices or the processing must take care of the new border conditions at the data slice edges.

3.5. The PPE-SPE Communication

After performing the correct changes in the main application and in the SPEkernel, the communication protocol between the two has to be implemented. The steps to be performed, indicated by the numbers in Figure 3, are:

1. The `Kernel1Interface` writes a command to the mailbox of the target SPE. The `main()` function of the SPE kernel processes the command by choosing the kernel function to execute
2. The `Kernel1Interface` writes the address of the data structure required by the kernel to the mailbox. The kernel reads this address from the mailbox, and it uses it to transfer the wrapper structure via DMA.
3. The kernel code uses DMA transfers to get the "real" data in its LS.
4. Once the kernel gets the data via DMA, it performs the operation.
5. The kernel programs the DMA transfer to put results in the designated output buffer
6. The kernel signals its termination, putting a message in its output mailbox. The `Kernel1Interface` object gets the signal, either by polling or by an interrupt, and copies the results from the buffer back to the class data.

4. Overall Application Performance

In this Section we briefly discuss the SPE-specific optimizations and their potential influence in the overall application performance gain.

[2] `http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Convolut-2.html`

4.1. The Kernel Optimizations

After the initial porting, based as much as possible on reusing the C++ code, the kernels have to be further optimized. Among the optimizations that can be performed [5] we quote:

- optimize the data transfer - either by DMA multi-buffering, or by using DMA lists
- vectorize the code - to make use of the SIMD abilities of the SPEs; use the data type that minimally suffices the required precision, to increase the SIMDization ways.
- use local SPE data structures that are contiguous (i.e., use arrays rather then linked lists).
- remove/replace branches or, if not possible, use explicit branch hints (using the `__builtin_expect` function) to provide the compiler with branch prediction information
- change the algorithm for better vectorization - interchange loops, replace multiplications and divisions by shift operations, etc.

All these optimizations are highly dependent on the kernel processing itself, so there are no precise rules or guidelines to be followed for guaranteed success. Even though a large part of the overall application performance gain is obtained by these optimizations, they are not the subject of this paper, so we do not explain them further, but refer the reader to [4, 17]. Still, we note that due to the inherent modularity of our strategy, which allows subsequent optimizations to be performed iteratively, as different kernel versions that adhere to the same interface can be easily plugged in via the `SPEInterface` stub.

4.2. Overall application performance

In the case of large C++ applications, the overall performance gain is highly dependent on the application structure, in terms of (1) how representative the kernels are in the overall computation, and (2) how can they be scheduled to compute in parallel. Thus, although one can assume that the speed-up obtained from an optimized kernel running on one SPE can be between one and two orders of magnitude (of course, depending on the kernel) compared to its PPE version, the overall application speed-up may be much lower if all these kernels are only covering a small part of the entire application. In terms of scheduling, if the application preserves the initial execution style, the PPE stalls during the SPE execution, which actually translates to a sequential execution on multiple cores, as see Figure 4(b). If the structure of the application allows it, the execution model should increase concurrency by using several SPEs and the PPE in parallel, as seen in Figure 4(c). We shall analyze both

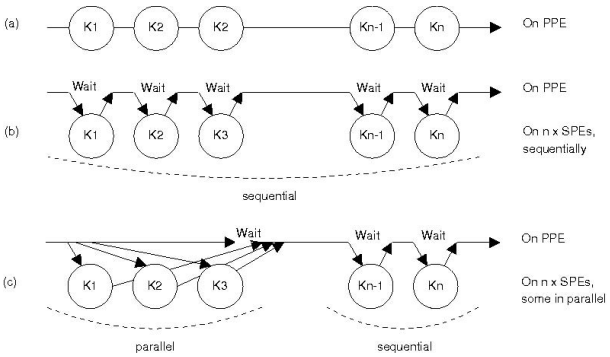

Figure 4. Application scheduling on the Cell cores

these scheduling solutions in the following paragraphs, and present results for both scenarios.

To estimate the performance gain of kernel optimizations, we use Amdhal's law as a first order approximation. Given K_{fr} as the fraction of the execution time represented by a kernel, and $K_{speed-up}$ as the speed-up of the kernel over the PPE, the resulting application speed-up, S_{app} is:

$$S_{app} = \frac{1}{(1 - K_{fr}) + \frac{K_{fr}}{K_{speed-up}}} \quad (1)$$

For example, for a kernel with K_{fr}=10% of an application, a speed-up $K_{speed-up} = 10$ gives an overall speed-up $S_{app} = 1.0989$, while the same kernel optimized to $K_{speed-up} = 100$ gives an overall speed-up $S_{app} = 1.1098$. Thus, the effort of kernel optimization from $K_{speed-up} = 10$ to $K_{speed-up} = 100$ is not worth in this case.

Considering that n SPE kernels ($K^i, i = 1..n$) run by the main application sequentially, like in Figure 4 (b), the overall performance can be estimated by:

$$S_{app} = \frac{1}{(1 - \sum_{i=1}^{n} K_{fr}^i) + \sum_{i=1}^{n} \frac{K_{fr}^i}{K_{speed-up}^i}} \quad (2)$$

Application performance can be improved in case the PPE application can schedule groups of SPE kernels in parallel, like in Figure 4 (c). Consider the n kernels split in G groups, each group having its kernels running in parallel. Note that the groups (due to data dependencies, for example) are still executed sequentially. If each group has g_j items, such that $n = \sum_{j=1}^{G} g_j$, the speed-up can be estimated by:

$$S_{app} = \frac{1}{(1 - \sum_{i=1}^{n} K_{fr}^i) + \sum_{j=1}^{g} \max_{k=1}^{p_j} (\frac{K_{fr}^l}{K_{speed-up}^l})} \quad (3)$$

By evaluating equations 2 and/or 3, one can determine (1) what is the weight of kernel optimizations in the overall application speed-up, and (2) what can be the gain of running the SPEs in parallel. Comparing these numbers (examples will follow in Section 5) with the estimated effort for the required changes for these different scenarios, we can evaluate the efficiency of the porting process.

5. The Case-Study - experiments and results

In this section we present a case study for our strategy. Besides briefly presenting the application itself, we focus on the porting roadmap and the experiments we have performed on the Cell. The experimental results show very promising results: the Cell application has one order of magnitude speed-up over the application running on Pentium machines.

5.1. Marvel

Because multimedia applications are considered very suitable case studies for the performance potential of multicore processors, we have chosen a multimedia content analysis and retrieval system as a case study for our application. Briefly, multimedia content analysis [19] refers to the ability of a system to detect the semantic meanings of a multimedia document, 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 multimedia content analysis and retrieval must be developed. For such systems, execution speed and accuracy are the most important performance metrics. MARVEL[3], developed by IBM Research, uses multi-modal machine learning techniques for bridging the semantic gap for multimedia content analysis and retrieval. After a short training phase, MARVEL is able to automatically annotate multimedia content, thus allowing further searching for and retrieval of content of interest. The main goal of MARVEL is to help organizing large and growing amounts of multimedia content much more efficiently. MARVEL consists of two engines: (1) the multimedia analysis engine, which applies machine learning techniques to model semantic concepts in video from automatically extracted audio, speech, and visual content [1], and (2) the multimedia retrieval engine, which integrates multimedia semantics-based searching with other search techniques for image and/or video searching [15].

In the remainder of this paper, we focus on MARVEL's multimedia analysis engine as the computation intensive part of the application. In particular, our goal is to increase

[3]MARVEL stands for Multimedia Analysis and Retrieval

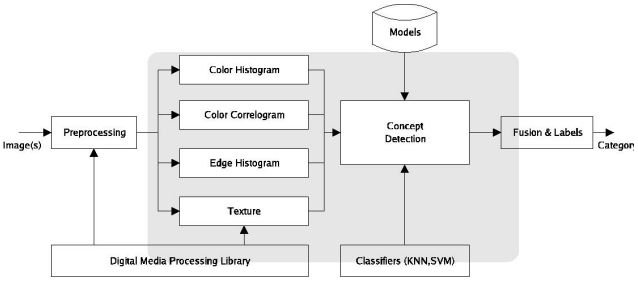

Figure 5. The processing flow of Marvel. The shaded part is ported to the SPEs

the execution speed of semantic concept detection in images. The processing flow of this simplified MARVEL version is presented in Figure 5. The preprocessing step includes (1) image reading, decompressing and storing it in the main memory as an RGB image, and (2) models reading and storing in memory. From the RGB representation, we extract some visual features of the image as vectors. The extracted features go through the concept detection phase, based on a collection of precomputed models and using one of the several available statistical classification methods like Support Vector Machines (SVMs), k-nearest neighbor search (kNN), etc. For our case-study, we have chosen four feature extraction techniques, for color (two), edge, and texture information, and we have opted for an SVM-based classification.

5.2. Porting MARVEL on Cell

It took roughly one day to compile and execute the Linux version of Marvel on the PPE. Next, we have profiled its execution for one 352x240 pixels color image, and for a set of fifty such images. For one image, the feature extraction and concept detection represented 87% of the execution time (the rest being application preprocessing); for the 50-images set, feature extraction and concept detection increased to 96% of the execution time (a large part of the preprocessing is a one-time overhead). Further, based on the profiling information, we have identified the core methods (in terms of execution time) for each of the feature extraction algorithms, as well as for the concept detections. Around these core methods we have clustered the additional methods required for easy data wrapping, processing and transfer, and we defined five major kernels, briefly listed below together with their coverage from the per-image application execution time (i.e., *without* the one-time overhead):

1. Color histogram extraction (CHExtract) - 8% from the application execution time.
   The color histogram of an image is computed by discretizing the colors within an image and counting the

number of colors that fall into each bin [18]. In MARVEL, the color histogram is computed on the HSV image representation, and quantized in 166 bins.

2. Color correlogram extraction (CCExtract) - 54% from the application execution time.
   The color correlogram feature in MARVEL quantifies, over the whole image, the degree of clustering among pixels with the same quantized color value. For each pixel P, it counts how many pixels there are within a square window of size 17x17 around P belonging to the same histogram bin as P [10].
3. Texture extraction (TXExtract) - 6% from the application execution time.
   Texture refers to a visual pattern or spatial arrangement of the pixels in an image. In MARVEL, texture features are derived from the pattern of spatial-frequency energy across image subbands [14].
4. Edge histogram extraction (EHExtract) - 28% from the application execution time.
   The edge histogram extraction is a sequence of filters applied in succession on the image: color conversion RGB to Gray, image edge detection with the Sobel operators, edge angle and magnitude computation per pixel, plus the quantization and normalization operations specific to histogram-like functions.
5. Concept Detection - 2% from the application execution time for the classification of all four features: CHDetect, CCDetect, TXDetect, EHDetect.

The remainder of the application execution time for one image (2%) is the image reading, decoding and rescaling. In our tests, we used all images of the same size, such that rescaling (otherwise a costly operation) is not required.

To have a first idea of the application and kernels performance, we have compared their execution times on the PPE at 3.2GHz with the execution of the original Linux application on two reference systems[4]: a desktop machine with a Pentium D processor (dual-core, running at 3.4GHz), from now on called "Desktop", and a laptop with a Pentium Centrino (running at 1.8GHz), from now on called "Laptop". The results show an average kernel slow-down of 2.5 from the Laptop to the PPE, and about 3.2 from the Desktop to the PPE. The exception is the image preprocessing step (which is mainly I/O access), which was only slowed down by a factor of 1.2 from the Laptop, and 1.4 from the Desktop; finally, the one-time overhead (i.e., the application-wise I/O operations) are also about the same on these three reference measurements, representing, for one image, 60% of the total execution time on the PPE (indeed, larger than the image processing), and about 80% from the total execution time on the other two reference machines.

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

5.3. Migrating the Kernels on SPEs

Once the five kernels have been identified, we have to port them on the SPEs. Every kernel becomes an SPE thread, structured as presented in the self-explaining Listing 1. Note that for each function, the `main` function enables both blocking (i.e., PPE polling for a new message) and non-blocking (i.e., PPE is interrupted by the new message) behavior of the PPE-SPE messaging protocol.

Listing 1. An example of a `main` function of an SPE kernel

```
// Kernel code :
int Function1(unsigned int address) { ... }
int Function2(unsigned int address) { ... }
//more functions ...
int main(unsigned long long spu_id, unsigned long long
     argv) {
 unsigned int addr_in, opcode;
 int result;
while(1) {
 opcode = (unsigned int) spu_read_in_mbox(); // read
      operation code from the mailbox
 switch (opcode) {
  case SPU_EXIT: return 0;
  case SPU_Run_1: {
    addr_in = (unsigned int)spu_read_in_mbox();// read
         the address from the mailbox
    result = Function1(addr_in); // run function
    if (POLLING)
      spu_write_out_mbox(result); //write result in
           the mailbox
    if (INTERRUPT)
      spu_write_out_intr_mbox(result); //write result
           in the "interrupt" mailbox
    break;
  }
  case SPU_Run_2: {...}
...
}
```

For three of the five chosen kernels, we have first measured the behavior before SPE-specific optimizations. Compared with the PPE version, the speed-ups of CHExtract, CCExtract and EHExtract were 26.41, 0.43 and 3.85, respectively. The significant difference in these results are mainly due to the specific computation structure of each kernel. Further on, we have proceeded with the SPE code optimizations, tuning the "reference" algorithm to allow better opportunities for a broad range of manual optimizations. like double and triple buffering of DMA transfers, loop unrolling, 16-ways and/or 4 ways SIMD-ization, branch removal etc. Table 1 shows the final speed-up of each SPE kernel over its initial PPE version.

5.4 Building the `SPEInterface`

Once the kernels have been completely implemented, they have to be reintegrated in the application itself. The

Table 1. SPE vs. PPE kernel speed-ups

Kernel	Speed-up	Coverage[%]
CH Extract	53.67	8%
CC Extract	52.23	54%
TX Extract	15.99	6%
EH Extract	65.94	28%
ConceptDet	10.80	2%

SPEInterface class is presented in code Listing 2. To implement the SendAndWait method, we have used a simple 2-way communication protocol based on mailboxes, as seen in Listing 3.

Listing 2. The SPEInterface class snippet

```
class SPEInterface {
private:
// more fields
  spe_program_handle_t spe_module;
  speid_t spuid;
public:
  SPEInterface(spe_program_handle_t module);
  ~SPEInterface();
  int thread_open(spe_program_handle_t module);
  int thread_close(int cmnd);
  int SendAndWait(int functionCall, unsigned int value);
  int Send(int functionCall, unsigned int value);
  int Wait(int timeout);
// more code
};
```

Listing 3. The SendAndWait protocol

```
int SPEInterface::SendAndWait(int functionCall,unsigned
     int value) {
  int retVal;
// send command and address
   spe_write_in_mbox(spuid,functionCall);
   spe_write_in_mbox(spuid,value);
// wait for result (i.e., completion signal)
   while(spe_stat_out_mbox(spuid)==0);
// read result to clean-up the mailbox
   retVal=spe_read_out_mbox(spuid);
   return retVal; }
```

The changes required in the main application to accommodate the dual execution - on SPE or PPE - are presented in Listing 4.

5.5. Experiments and results

With the optimized version of the application, we have performed experiments on sets of one, ten and fifty images. For each image, we have performed the four feature extractions and the corresponding concept detection. The concept detection was performed with a number of models summing up 186 vectors for color histogram, 225 for color correlogram, 210 for edge detection and 255 for texture. Figure 6 presents the speed-ups of the kernels execution times on the two reference machines, on the PPE and on the SPE.

Next, we have calculated the application estimated performance, using equations 1 and 2 in Section 4, and using

Listing 4. The main application modifications

```
// interface initialization
#ifdef RUN_ON_SPU
  extern spe_program_handle_t SPE_Kernel1;
  SPEInterface *Kernel1Interface = new CellInterface(
       SPE_Kernel1);
#endif
// more code
#ifdef RUN_ON_SPU
Histogram& ColorHistogram::extract(const ColorImage& A)
     {
// some code - preprocessing the image
   FILL_MSG_FROM_COLORIMAGE(msg_color, A); // data
        wrapping
   int val=Kernel1Interface->SendAndWait(SPU_Run,(
        unsigned int)&msg_color); // call SPE kernel
   memset(histogram, (float *)msg_color.histogram,
        HISTOGRAM_SIZE); // put data back
   free_align((void *)msg_color.histogram); // free
        buffer from wrapped data
   return *this;
}
#else
Histogram& ColorHistogram::extract(const ColorImage& A)
     {
// original code }
#endif
```

the measured speed-up numbers for each kernel. In all scenarios, we map at most one kernel per one SPE. We have evaluated three scenarios:

1. All kernels execute sequentially (i.e., no task parallelism between SPEs). This scenario is equivalent with the use of a single SPE, but it avoids the dynamic code switching: $S_{app}^{SingleSPE}$(SPE:Desktop)=10.90
2. All feature extractions run in parallel; the concept detection is still running on a single SPE, thus all concept detection operations run sequentially: $S_{app}^{Multi-SPE}$(SPE:Desktop)=15.28
3. All feature extractions run in parallel; the concept detection code is replicated on the other 4 SPEs, to allow parallel execution, and each extraction is immediately followed by its own detection: $S_{app}^{Multi-SPE2}$(SPE:Desktop)=15.64. The very small difference between the two parallelization solutions can be explained by (1) the high impact of the color correlogram extraction, which dominates the feature extraction operations, (2) the small impact of the concept detection (only 0.5%, on average), and (3) the image preprocessing part, which runs on the PPE.

We have performed experiments for scenarios 1 (Single SPE) and 2 (Parallel SPEs). We have compared the time measurements with the application running on PPE, Desktop, and Laptop. The results, matching the estimates with an error of less than 2%, are presented in the speed-up graph from Figure 7.

Note that to obtain this fast execution of Marvel on the Cell, we have performed aggressive optimization for each of

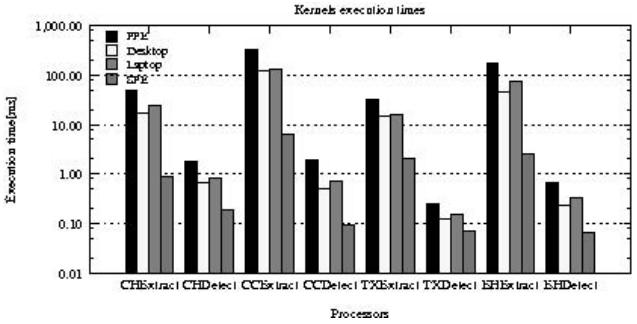

Figure 6. SPE kernel performance (execution time, logarithmic scale)

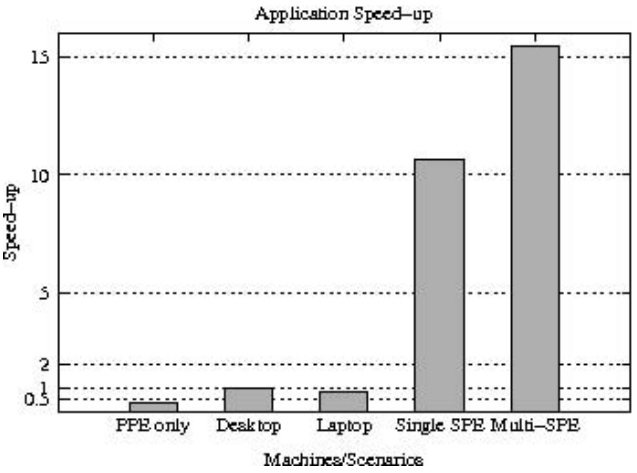

Figure 7. Speed-up results for all the experiments

the kernels and we have used several parallelism layers of the processor. Thus, the comparison with the results on the two reference processors, the Pentium D and the Pentium Centrino, is only informative, as no architecture-specific optimizations were performed on the generic Linux code running on these machines.

6. Related Work

Most of the work done so far on the Cell BE processor has focused either on (1) application development and performance evaluation, aiming to prove the level of performance that the architecture can actually achieve, or on (2) high level programming models, proving that the programmer can be abstracted away (to some degree) from the low-level architectural details. In this respect, our approach falls somewhat in between, as we present a generic strategy for fast porting of sequential C++ applications on the Cell, and we prove its applicability by developing and evaluating a case-study application.

For example, the MPI microtask model, proposed in [16], aims to infer the LS administration and DMA transfers from the message passing communication between microtasks that are defined by the user. Once this infrastructure is defined, a preprocessor splits the microtasks in basic tasks, free of internal communication, and schedules them dynamically on the existing SPEs. While the notions of kernels and microtasks/basic tasks are quite similar, we note that the MPI microtasks work on C only. For using this model on C++ applications, the programmer will have to first apply our strategy: kernel identification, isolation, data wrapping. While the clustering and scheduling of the microtask model are quite efficient, it will also require more transformations on the kernel side, thus decreasing the efficiency of a proof-of-concept porting. CellSs, a programming model proposed in [2], provides an user annotation scheme to allow the programmer to specify what parts of the code shall be migrated on the SPE. A source-to-source compiler separates the annotated sequential code in two: a main program (that will be running on the PPE) and a pool of functions to be run on the SPEs. For any potential call to an SPE-function, a runtime library generates a potential SPE task which can be executed when all its data dependencies are satisfied, if there is an SPE available. However, CellSs cannot directly deal with C++ applications. On the applications side, among the first applications that emphasized the performance potential of Cell BE have been presented by IBM in [13, 8]. In [13], a complete implementation of a terrain rendering engine is shown to perform 50 times faster on a 3.2GHz Cell machine than on a 2.0GHz PowerPC G5 VMX. Besides the impressive performance numbers, the authors present a very interesting overview of the implementation process and task scheduling. However, the entire implementation is completely Cell oriented and tuned, allowing optimal task separation and mapping, but too low a flexibility to be used as a generic methodology for porting existing applications. An interesting application porting experiment is presented in [3], 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 (i.e., different than, say, a reference C++ implementation) algorithm to implement ray tracing on Cell. Finally, the lessons learned from porting Sweep3D (a high-performance scientific application) on the Cell processor, presented in [17], 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.

7. Conclusions

To the best of our knowledge, this work is the first to tackle the systematic porting of large C++ application on

Cell BE. Our approach does not require an extensive port of the C++ code on the Cell, but it is rather based on detecting compute intensive kernels of the initial application, isolate them from the C++ class structure and migrate them for execution on the SPEs, while the rest of the C++ application runs on the PPE. The method is generic in its approach, being applicable for any C++ application. The porting process allows the application to be functional at all times, thus permitting easy performance checkpoints of intermediate versions. The optimization of the SPE kernels code is still the responsibility of the programmer, but given that the resulting application is modular, it allows for iterative improvements. Finally, to support effective optimization of the SPEs, we provide a simple method to evaluate the performance gain of a potential optimization in the overall performance of the application.

We have validated our methodology by presenting our experience with porting a multimedia retrieval application that comprises of more than 20000 lines of code and over 50 classes. The performance of the ported application - namely, the overall application speed-up of the Cell-based application - is one order of magnitude above the one of a commodity machine, running the reference sequential code.

We conclude that our strategy is well-suited for a first attempt of enabling an existing C++ code-base to run on Cell. It is recommended for large applications, when rewriting from scratch in a Cell-aware manner is not really an option. Also, the results of our strategy can be used for proof-of-concept approaches evaluating the performance potential of Cell for given applications.

ACKNOWLEDGEMENTS. Most of this work has been done at IBM TJ Watson Research Center, where all the authors had the opportunity to meet and cooperate closely. Furthermore, we would like to thank Michael Perrone, Gordon Braudaway, Karen Magerlein, and Bruce D'Amora, from IBM TJ Watson, for their valuable support and ideas provided during the development of this application.

References

[1] A. Amir, W. Hsu, G. Iyengar, C.-Y. Lin, M. Naphade, A. Natsev, C. Neti, H. J. Nock, J. R. Smith, B. L. Tseng, Y. Wu, and D. Zhang. IBM Research TRECVID-2003 system. In *NIST Text Retrieval (TREC)*, Gaithersburg, MD.

[2] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the cell be architecture. volume SC'06. IEEE Computer Society Press, November 2006.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the Cell processor. volume IEEE Symposium on Interactive Ray Tracing 2006, pages 15–23, September 2006.

[4] N. Blachford. Cell architecture explained version 2 part 3: Programming the Cell. `http://www.blachford.info/computer/Cell/Cell3v2.html`, 2006.

[5] N. Blachford. Programming the Cell processor-part 2: Programming models. `http://www.blachford.info/computer/articles/CellProgramming2.html`, 2006.

[6] N. Blachford. Programming the Cell processor-part 3: Programming models. `http://www.blachford.info/computer/articles/CellProgramming3.html`, 2006.

[7] D. A. Brokenshire. Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance. `http://www-128.ibm.com/developerworks/power/library/pa-celltips1/`, 2006.

[8] B. D'Amora. Online Game Prototype (white paper). `http://www.research.ibm.com/cell/whitepapers/cell_online_game.pdf`, May 2005.

[9] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM Press.

[10] J. Huang, S. R. Kumar, M. Mitra, W.-J. Zhu, and R. Zabih. Image indexing using color correlograms. In *CVPR '97*, page 762, Washington, DC, USA, 1997. IEEE Computer Society.

[11] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[12] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[13] B. Minor, G. Fossum, and V. To. Terrain rendering engine (white paper). `http://www.research.ibm.com/cell/whitepapers/TRE.pdf`, May 2005.

[14] M. R. Naphade, C.-Y. Lin, and J. R. Smith. Video texture indexing using spatio-temporal wavelets. In *ICIP (2)*, pages 437–440, 2002.

[15] A. Natsev, M. Naphade, and J. R. Smith. Semantic space processing of multimedia content. In *ACM SIGKDD 2004*, Seattle, WA, August 2004.

[16] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1), March 2006.

[17] F. Petrini, J. Fernàndez, M. Kistler, G. Fossum, A. L. Varbanescu, and M. Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *IPDPS 2007, Long Beach, CA*. IEEE/ACM, March.

[18] J. R. Smith and S.-F. Chang. Tools and techniques for color image retrieval. In *SPIE '96*, volume 2670, 1996.

[19] Y. Wang, Z. Liu, and J.-C. Huang. Multimedia content analysis using both audio and visual clues. *IEEE Signal Processing Magazine*, 17(6):12–36, November 2000.

[20] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientic computing. In *ACM Computing Frontiers 06*, pages 9–20. ACM Press, 2006.