

High Performance MPEG-2 Software Decoder on the Cell Broadband Engine

David A. Bader
College of Computing
Georgia Institute of Technology
bader@cc.gatech.edu

Sulabh Patel
Electronic Arts, Inc.
sulabh@gmail.com

Abstract

The Sony-Toshiba-IBM Cell Broadband Engine is a heterogeneous multicore architecture that consists of a traditional microprocessor (PPE) with eight SIMD co-processing units (SPEs) integrated on-chip. While the Cell/B.E. processor is designed with multimedia applications in mind, there are currently no open-source, optimized implementations of such applications available. In this paper, we present the design and implementation behind the creation of an optimized MPEG-2 software decoder for this unique parallel architecture, and demonstrate its performance through an experimental study.

This is the first parallelization of an MPEG-2 decoder for a commodity heterogeneous multicore processor such as the IBM Cell/B.E. While Drake et al. have recently parallelized MPEG-2 using StreamIt for a streaming architecture, our algorithm is quite different and is the first to address the new challenges related to the optimization and tuning of a multicore algorithm with DMA transfers and local store memory. Our design and efficient implementation target the architectural features provided by the heterogeneous multicore processor. We give an experimental study on Sony PlayStation 3 and IBM QS20 dual-Cell Blade platforms. For instance, using 16 SPEs on the IBM QS20, our decoder runs 3.088 times faster than a 3.2 GHz Intel Xeon and achieves a speedup of over 10.545 compared with a PPE-only implementation. Our source code is freely available through SourceForge under the CellBuzz project.

1. Introduction

The Cell Broadband Engine (or the Cell/B.E.) [6] is a novel architectural design by Sony, Toshiba, and IBM (STI), primarily targeting high performance multimedia and gaming applications. It is a heterogeneous multicore chip that is significantly different from conventional multi-processor or multicore architectures. It consists of a traditional mi-

croprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. There are several unique architectural features in the Cell/B.E. that clearly distinguish it from current microprocessors: the Cell chip is a computational workhorse, and it offers a theoretical peak single-precision floating point performance of 204.8 GF/s. We can exploit parallelism at multiple levels on the Cell: each chip has eight SPEs with two-way instruction-level parallelism on each SPE. Further, the SPE supports both scalar as well as single-instruction, multiple data (SIMD) computations [5]. The on-chip coherent bus and interconnection network elements have been specially designed to cater for high performance on bandwidth-intensive applications (such as those in gaming and multimedia).

Despite being architected for multimedia applications, at present there are few applications available to the open-source community that target the Cell/B.E. Therefore, to showcase the potential of the Cell/B.E. with regards to multimedia applications, we chose to design and implement the MPEG-2 video decoder. The MPEG-2 standard for motion video is already widely used around the world for the transmission of digital television and for video distributed on DVD (or similar formats). Also, the compression method behind MPEG-2 is similar in structure to other codecs, making it a good choice to implement first.

The main contribution of our paper is the first parallelization of an MPEG-2 decoder for a commodity heterogeneous multicore processor such as the IBM Cell/B.E. In 1996, Bilas *et al.* [1] described a real-time software-based MPEG-2 decoder. This decoder does not match to the Cell/B.E. processor because it generates substantial data that can not fit in the 256 KB of Local Store space available for both code and data on each SPE. While Drake *et al.* [2] recently parallelized MPEG-2 using StreamIt for a streaming architecture, their approach gives little insight on the parallelization for a heterogeneous multicore processor, such as the Cell/B.E. Our multicore design and efficient imple-

mentation target the architectural features provided by the heterogeneous multicore processor where one must address the new challenges related to the optimization and tuning with several levels of parallelism, the use of DMA transfers and local storage, as well as SIMD-ization on each core.

We give an experimental study on two platform that use the 3.2GHz Cell Broadband Engine processor: the Sony PlayStation 3 and IBM QS20 dual-Cell Blade. For comparison, we compare the performance using a commodity microprocessor with the same 3.2GHz clock frequency, an Intel Xeon processor. For instance, we run over three times faster using the IBM QS20 over the Intel Xeon platform. Also, ours is the first MPEG-2 decoder for the Cell/B.E. and is freely-available as source code and pre-compiled libraries from the CellBuzz SourceForge repository (<http://sourceforge.net/projects/cellbuzz>).

We first present an overview of the Cell/B.E. architecture in Section 2, along with an overview of the MPEG-2 decoding algorithm in Section 3. The overview of MPEG-2 also includes a brief analysis of the availability of concurrency within the algorithm. We then present details of our implementation in Section 4 and present how we increased performance in motion reconstruction in Section 4.1. Finally, we present our performance results in Section 5.

2. Cell/B.E. Architecture

The Cell Broadband Engine is a heterogeneous multicore chip that is significantly different from conventional multiprocessor or multi-core architectures. It consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Fig. 1 gives an architectural overview of the Cell. We refer the reader to [9, 3, 7] for additional details.

The PPE is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KB L1 instruction and data caches, and a 512 KB L2 cache. It is a dual issue, in-order execution design, with two way simultaneous multithreading. Ideally, all the computation should be partitioned among the SPEs, and the PPE only handles the control flow.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The SPU is a microarchitecture designed for high performance data streaming and data intensive computation. The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPEs and the PPE. The SPE includes a 256 KB *local store* (LS) memory to hold SPE program's instructions and data. The SPE cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into the

Local Store or write computation results back to the main memory. DMA is non-blocking so that the SPE can continue program execution while DMA transactions are performed.

The SPE is an in-order dual-issue statically scheduled architecture. Two SIMD [5] instructions can be issued per cycle: one compute instruction and one memory operation. The SPU branch architecture does not include dynamic branch prediction, but instead relies on compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as far as possible.

The MFC supports naturally aligned transfers of 1,2,4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 KB. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. Peak performance is achievable when both the effective address in main memory and the Local Store address are 128 bytes aligned and the transfer is an even multiple of 128 bytes. In the Cell/B.E. processor, each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in on-chip communication. Kistler *et al.* [7] analyze the communication network of the Cell/B.E. processor and state that applications that rely heavily on random scatter and gather accesses to main memory can take advantage of the high communication bandwidth and low latency.

With a clock speed of 3.2 GHz, the Cell/B.E. processor has a theoretical peak performance of 204.8 GF/s (single precision). The EIB supports a peak bandwidth of 204.8 GB/s for intrachip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

3. MPEG-2 Decoding Algorithm

The MPEG-2 video coding standard defines a lossy compression algorithm that takes advantage of spatial and temporal correlation in order to achieve high compression ratios. Spatial correlation finds similarities within each picture to eliminate redundancy, while temporal correlation finds similarities between successive pictures. The complete MPEG-2 standard is very detailed, as it is very flexible to meet a wide array of demands and is available as an ISO document [8].

The most important aspect of the MPEG-2 standard is its layered hierarchy. Fig. 2 provides an overview of the hierarchy. Vital to parallelization is that different parts of the stream are marked with byte-aligned codes (startcodes). This allows for fast, random access to the various structures within the datastream without performing extensive header

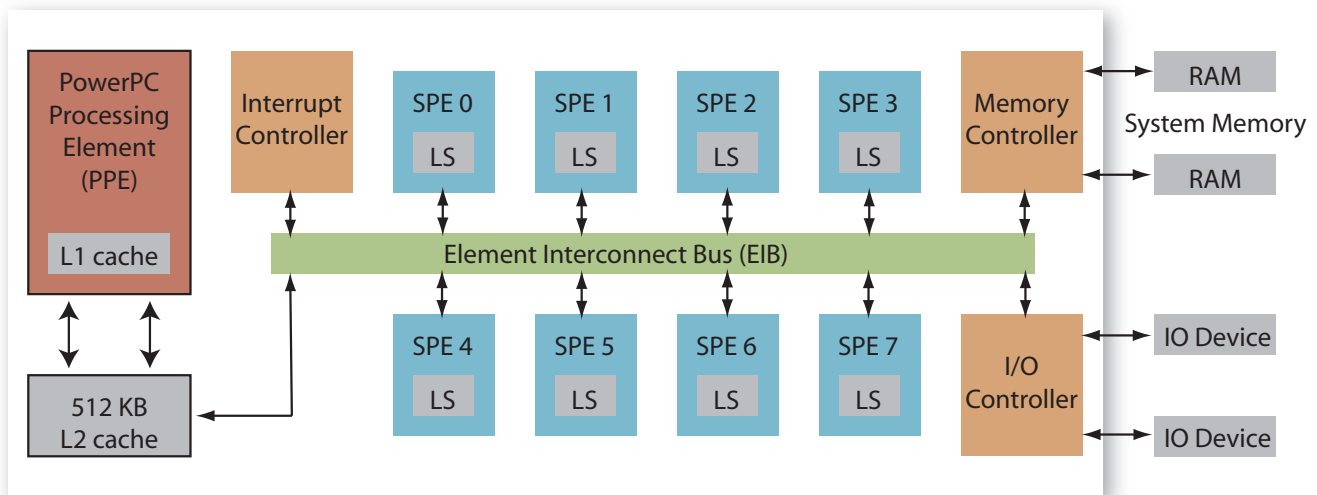


Figure 1: Cell Broadband Engine Architecture

decoding.

The highest level of the hierarchy is the *sequence* level. Each sequence is made up of *groups of pictures*, or *GOPs*. Each GOP is a grouping of a number of I-, P-, and B-pictures. I-pictures, or *intra coded pictures*, use only spatial compression. This means that I- pictures can be decoded independently of the other pictures, providing starting points for decoding. While P- and B-pictures use spatial and temporal compression, they vary on the reference pictures used by their temporal compression. P- pictures, or *predictive coded pictures*, use temporal compression based on past I- or P- pictures (reference pictures). B- pictures, or *bidirectionally-predictive coded pictures*, use temporal compression based on past and future reference pictures. Here, past and future refer to the presented order of pictures, not the decoded order of pictures.

Pictures corresponding to a frame (progressive) or a field (interlaced) contain *slices*. A slice cannot span multiple rows of the picture, but there can be multiple slices per row. Each slice consists of *macroblocks*, which are in turn divided into *blocks*. Macroblocks and blocks do not have startcodes associated with them.

Decoding an MPEG-2 stream is described by Fig. 3. Huffman and run-length decoding are performed to decode headers (sequence, GOP, and slice headers) and to generate the quantized block (quantization introduces the lossy portion of the standard). Performing inverse quantization on each block gives the coefficients for the next step, the inverse discrete cosine transform (IDCT). The IDCT is applied to obtain the block's spatial data (I-Picture) or prediction error (P- and B-pictures). If necessary, motion com-

ensation is performed to generate the final data from the prediction error data and reference pictures.

Each level in the MPEG-2 hierarchy provides a possible point of parallelization. Bilas *et al.* [1] examine each point and discover that the only acceptable parallelism points are at the GOP level and at the slice level. The authors test on a multiprocessor system and consider several important factors in coming to this conclusion. The points of comparison include memory footprint, load balancing between processors, and scalability when adding additional processors. Other levels of MPEG-2 are rejected as points of parallelism because doing so would create too many serial portions of execution, due to dependencies, or tasks too variable in size to properly load balance. Drake *et al.* [2], however, use macroblock decoding as their point of parallelism. Within macroblocks, the authors parallelize motion vector decoding and applying the IDCT. Since the focus of their work is primarily message passing and using a streaming data model, we thought a similar approach might work well on the Cell/B.E.

We confirmed these findings using test implementations on the Cell/B.E. Slice level parallelism works very well. Through profiling we find that the majority of the execution time is spent in the IDCT portion of decoding, confirming macroblock decoding as a candidate for parallelism. However, when we examine macroblock level parallelism closely, we observe that the next largest portion of execution time is spent in header decoding on the PPE. Also, since the SPEs are issuing DMA commands for small amounts of memory, the EIB is not being utilized fully. Therefore we use parallelism at the slice level. Doing so achieves a

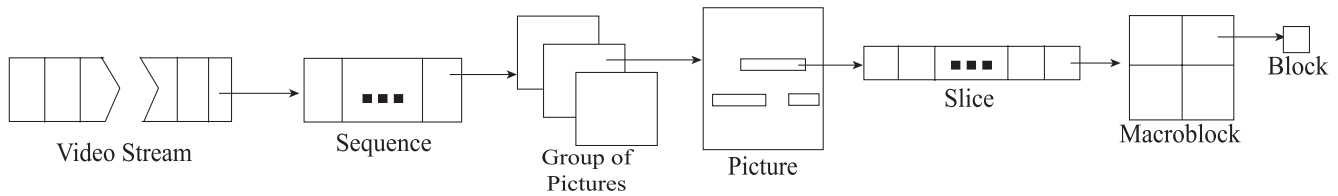


Figure 2: MPEG-2 Structural Hierarchy

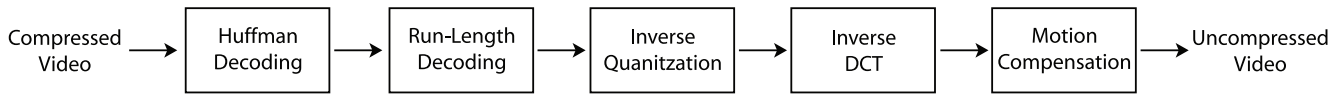


Figure 3: Sequential decode sequence for a MPEG-2 block

balance between utilizing the EIB, the Local Store, and parallelizing a larger portion of header decoding.

4. Cell/B.E. MPEG-2 Decoding

The mantra of achieving high performance on the Cell/B.E. should be iterative performance optimization of the code. Given the complexity involved in extracting maximum performance from the architecture, particularly the SPEs, a single pass approach would be too overwhelming for a programmer, particularly a novice. To emphasize this point, we outline each optimization step of our process. The source file names from our final implementation are also provided. Except for the following changes, most of the reference implementation was not modified.

Our software is based on the reference implementation of MPEG-2 provided by MPEG (Moving Pictures Expert Group) [4]. In this implementation, most of the data is in global space, which makes the initial analysis step relatively simple. The first step consists of simply splitting the code base in two, one part to run on the PPE and the other part to run on the SPEs. The PPE code primarily handles the I/O and global data structures, while the SPE part contains the computational tasks involved with decoding MPEG-2 streams. We analyze the slice decoding function (in *getpic.c*) to determine the inputs and outputs, including the dependent global data. Since each functional unit is compiled separately, we rely mostly on the compiler generated errors in this step after splitting the code base into separate PPE and SPE bases.

Once we created two separate code bases, we started an incremental procedure of parallelizing the program on the Cell/B.E. First, we achieve program correctness. To simplify debugging, we limited the program to synchronously decoding a single slice from the first frame on one SPE and then exiting. The first slice of the first picture should always be from an I-picture, so this eliminates the motion compensation code and allows for faster iteration. This step creates

a control block structure containing data that the SPE needs from the PPE's global space, and the DMA commands on the SPE side for bringing the control block in and writing the decoded slice back out. We create the control block and launch the decoding job in a blocking fashion inside the function for decoding a picture (*getpic.c:picture_data()*). After successfully completing this task, we move next to decoding one I-picture completely on one SPE.

The next step requires decoding a P- or B- picture, and would prove very challenging. Located in *recon.c* is the code which uses the decoded motion vectors and the reference pictures to reconstruct a macroblock in the current picture. This should not be confused with *motion.c*, which is the code dealing with decoding the motion vectors. This task requires the ability to perform potentially random access in the reference pictures from the SPE side. This would clearly require a software cache system on the SPE side, because of the limited amount of Local Store space. We examine the methodology used in optimizing this software cache system in Section 4.1. For now, we simply issue a blocking DMA command to bring in the necessary data from the reference pictures for each motion vector.

The final task we face is the creation of a work-queue system so that the PPE can execute in parallel with the SPEs. This is also the step where we introduce the ability to run on multiple SPEs at once. We design a standard work-queue model, with providers and consumers, and use the Cell/B.E. mailbox messaging system. The latency of the mailbox system is negligible compared with the overall time needed to decode a slice. This code is in *CellInterface.c* on the PPE side. After completing this task, we begin the next phase of the process – improving the performance of the application.

One possible optimization we found during profiling relates to decoding variable length headers on the SPEs. Each level of the MPEG-2 hierarchy has an associated startcode and header. To conserve space, the headers are stored using a variable length scheme, which means that decoding

these headers requires a significant amount of branches. Since the branch miss penalty on the SPE is higher than on the PPE, there are two options. The first option is that all header decoding can be performed on the PPE and all relevant information can be passed using the control block. The other option is to decode as much of the headers as possible on the SPEs. The first option emphasizes minimizing branch penalties while the second emphasizes increased parallelization.

Because the structure of the reference implementation made it easy, we decided to simply implement two methods and compare directly. Through performance analyses, we discover that performing all header decoding on the PPE does not scale at all. Using more than 3 SPEs results in SPE starvation while the PPE decodes the next work unit. Therefore, empirically, we find that the increase in parallelization and load-balancing far outweighs the branch miss penalties that the SPEs incur.

After solidifying the algorithm, and performing the motion vector optimization described in Section 4.1, we apply common low-level techniques for improving performance on the Cell/B.E. Primarily, we double buffer DMA transfers and, where possible, apply SIMD instructions in computationally expensive portions of the code.

A potential problem we notice during profiling is the amount of code needed in order to decode a slice. The complete executable on the SPE side is approximately 142 KB. This does not leave much room for software cache systems or double buffering, especially considering the amount of data associated with each slice and picture. Overlays could potentially be used, but almost all of the code is used between DMA commands. Therefore, currently most of the DMA commands issued are blocking.

Using this iterative performance optimization process is indicative of how development for the Cell/B.E. should be approached. Attempting to tackle all of the nuances of the Cell/B.E. at once can be a daunting task, especially for a novice Cell/B.E. programmer. Fig. 4 shows the overall sequence of events in decoding a picture.

4.1. Optimizing Motion Reconstruction

During the performance optimization process, we find that the majority of the running time is spent waiting on the DMA commands in *recon.c* to complete. Our procedure for optimization uses profiling combined with simply removing the DMA commands while maintaining the same amount of SPU computation, not worrying about the program's correctness, and then measuring the effect on running time.

The technique of removing the DMA commands while maintaining the computational nature of the application helps determine whether the dual-issue capability of the SPEs is being utilized completely. If the running time im-

proves dramatically, then the DMA commands are a bottleneck and double-buffering or a software cache should be considered. If the running time is not improved, then the running time is bound to the amount of computation and SIMD-ization or other techniques should be utilized to improve running time.

In our case, the running time of the application improves dramatically without the DMA commands used by the motion reconstruction code. Through further analysis of access patterns, we find that the majority of our test files required alternating access to slices during motion reconstruction. Although this is not indicative of the general case, we introduce a software cache system which maintains the last two accessed slices. The software cache also utilizes the least recently used (LRU) eviction policy since, once a slice is used completely (after several alternating accesses) in motion reconstruction, it is not likely to be accessed again until the next picture. Each cache line refers to an entire slice from a reference picture, in order to exploit locality in the references along the x -axis.

The process we use for finding the performance bottleneck in motion reconstruction highlights the effectiveness of two simple techniques when dealing with the Cell/B.E. The first technique is to simply eliminate all DMA commands that could potentially be double-buffered or cached and see how running time is affected. The second technique is to simply profile access patterns in order to determine the best method of hiding DMA latency, with either a double-buffering scheme or the use of a software cache.

5. Performance Results

In order to analyze the performance and scalability of our MPEG-2 library for the Cell Broadband Engine processor, we use a benchmark video stream for our experimental studies. Our test runs take as input a 704×480 MPEG-2 stream with 450 frames. We remove all display duties in order to remove variability between graphical display operations between architectures.

We use two types of Cell/B.E. platforms to compare the performance of our MPEG-2 implementation: an IBM QS20 dual-Cell/B.E. blade (IBM QS20) and a Sony PlayStation 3 (Sony PS3). The IBM QS20 uses two 3.2 GHz Cell/B.E. processors, 1 GB of RAM, and 16 available SPEs. We compile our application using the IBM XLC compiler with the `-O3` flag and we compile a PPE-only reference implementation with both XLC and `gcc`. The stock Sony PS3 runs YellowDog Linux, and has a single 3.2 GHz Cell/B.E. processor, 256 MB of RAM, and 6 available SPEs. On the Sony PS3, we compile our application using the IBM XLC compiler, and as a comparison we compile one version of the PPE-only version using `gcc`. Both compilers are called with the `-O3` flag.

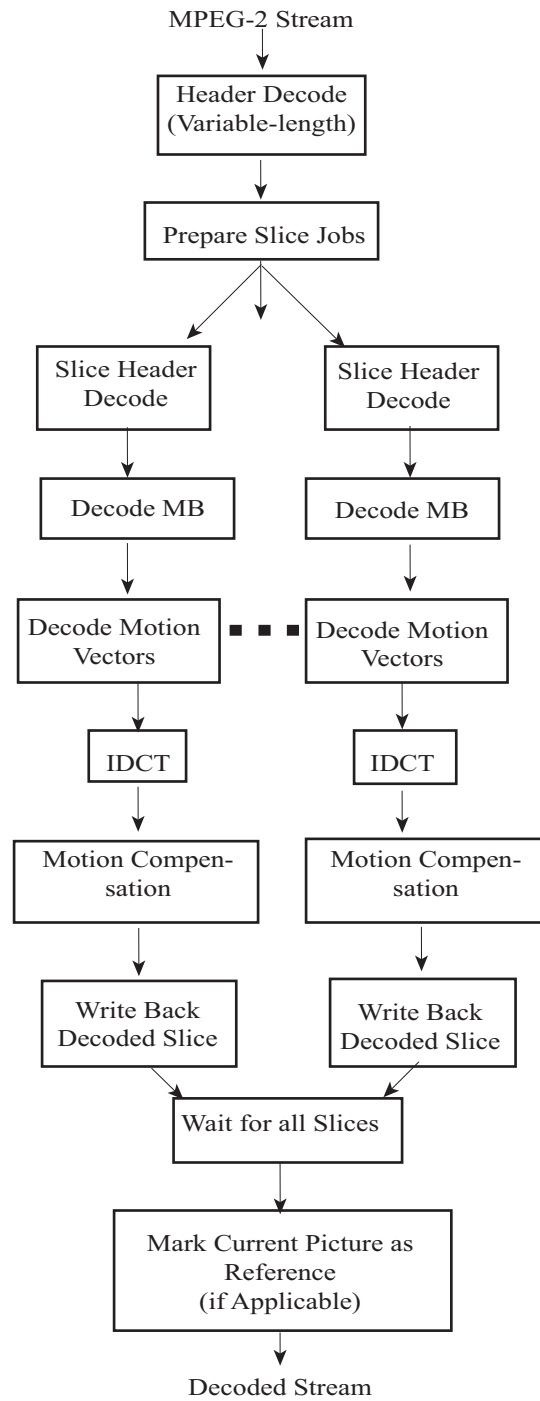


Figure 4: Parallel decode sequence. The parallel portion is run on the SPEs and the serial portions are run on the PPE.

Comparison of MPEG-2 Decoding Performance

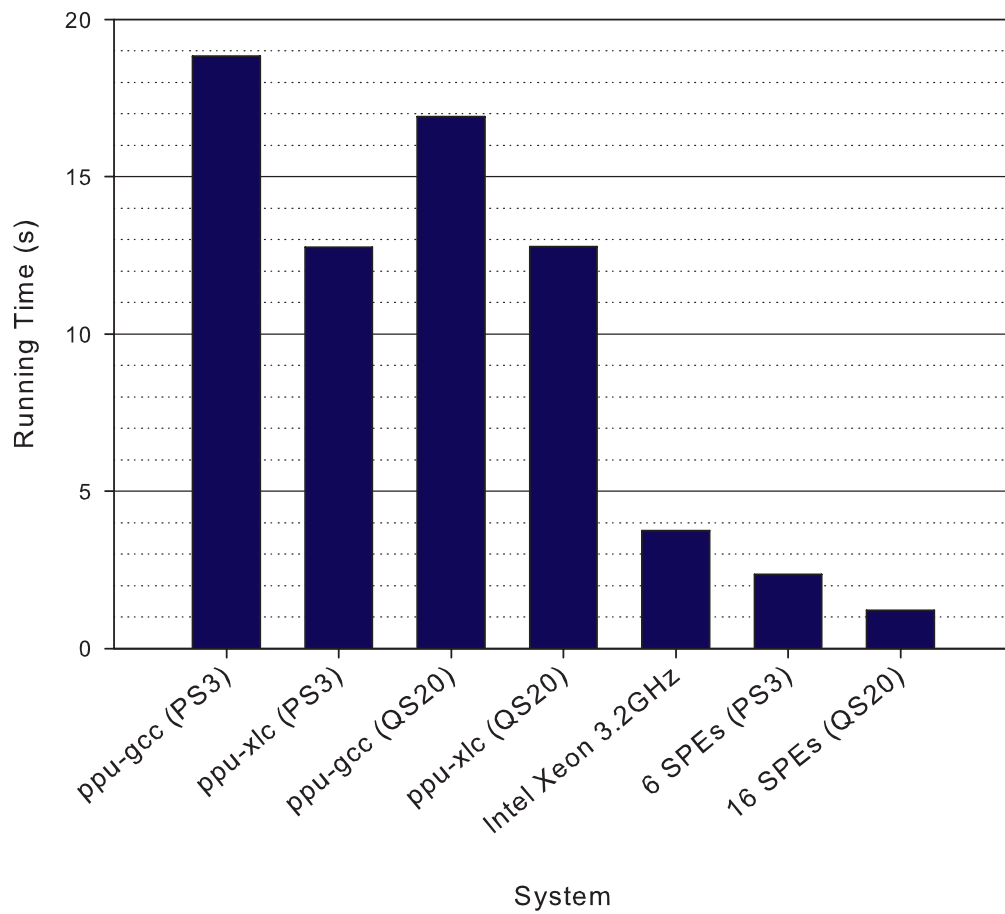


Figure 5: Comparing the fastest Cell/B.E. implementation of MPEG-2 against other architectures and PPE-only references.

As a comparison, we also run the reference implementation on a 3.2 GHz Intel Xeon Processor with 4 GB of RAM. We compile the program with `gcc` using the `-O3` flag. The difference in RAM is not significant since the application and benchmark fits within the memory of all three platforms.

Platform	Sony PS3	IBM QS20
1 PPE + 0 SPEs (<code>gcc</code>)	18.836	16.909
1 PPE + 0 SPEs (<code>xlc</code>)	12.759	12.777
1 PPE + 1 SPEs (<code>xlc</code>)	13.413	13.805
1 PPE + 2 SPEs (<code>xlc</code>)	6.766	6.983
1 PPE + 3 SPEs (<code>xlc</code>)	4.549	4.663
1 PPE + 4 SPEs (<code>xlc</code>)	3.636	3.726
1 PPE + 5 SPEs (<code>xlc</code>)	2.788	2.863
1 PPE + 6 SPEs (<code>xlc</code>)	2.354	2.399
1 PPE + 7 SPEs (<code>xlc</code>)		2.330
1 PPE + 8 SPEs (<code>xlc</code>)		1.952
1 PPE + 9 SPEs (<code>xlc</code>)		1.905
1 PPE + 10 SPEs (<code>xlc</code>)		1.575
1 PPE + 11 SPEs (<code>xlc</code>)		1.513
1 PPE + 12 SPEs (<code>xlc</code>)		1.482
1 PPE + 13 SPEs (<code>xlc</code>)		1.469
1 PPE + 14 SPEs (<code>xlc</code>)		1.434
1 PPE + 15 SPEs (<code>xlc</code>)		1.212
1 PPE + 16 SPEs (<code>xlc</code>)		1.210

Table 1: MPEG-2 Decoder Execution Times (in seconds) on the Sony PlayStation 3 and the IBM QS20 Dual-Cell Blade. As a reference, the benchmark takes 3.736 seconds on a 3.2GHz Intel Xeon processor.

Table 1 shows the raw running-time numbers for varying numbers of SPEs on the PS3. Both tables also include the running-times for the reference implementation compiled with `XLC` and `gcc` on each platform. Running-time here means the amount of time each program needs to completely decode every picture in the datastream.

In Fig. 6 we plot the results from using varying numbers of SPEs (in order to show scalability) for the IBM QS20 and Sony PS3. Finally, we compare the 6 SPE run-time on the PS3, the 16 SPE run-time on the QS20, the PPE-only reference compiled using `gcc` on both PS3 and QS20, a PPE-only reference compiled with `XLC` on both PS3 and QS20, and a reference on a 3.2 GHz Intel Xeon compiled using `gcc` in Fig. 5.

We first observe that using 4 SPEs on either the QS20 or PS3 beats the running-time of the reference implementation running on the Intel Xeon processor. After this, the noticeable result is that, when using SPEs, the QS20 performs slightly slower than the PS3 when both use the same number of SPEs. Also, using profiling we observe that memory allocation seems to run slightly slower on the QS20 than the

PS3. Both of these results are due to the two platforms running different operating systems and the fact that the PS3 will always have the same amount of RAM. We believe that this limitation allows the operating system to optimize memory allocation.

Another result between the QS20 and the PS3 is the difference in running-times of the reference implementations compiled with `XLC` and `gcc` when run only on the PPE. Both runs of the `XLC` compiled application are similar in running-time, but the running-time of the two `gcc` compiled applications differ greatly. The difference between the `XLC` versions can be explained by the operating system differences, while the differences between the `gcc` versions can be attributed to the use of slightly different versions of `gcc`.

6. Conclusions

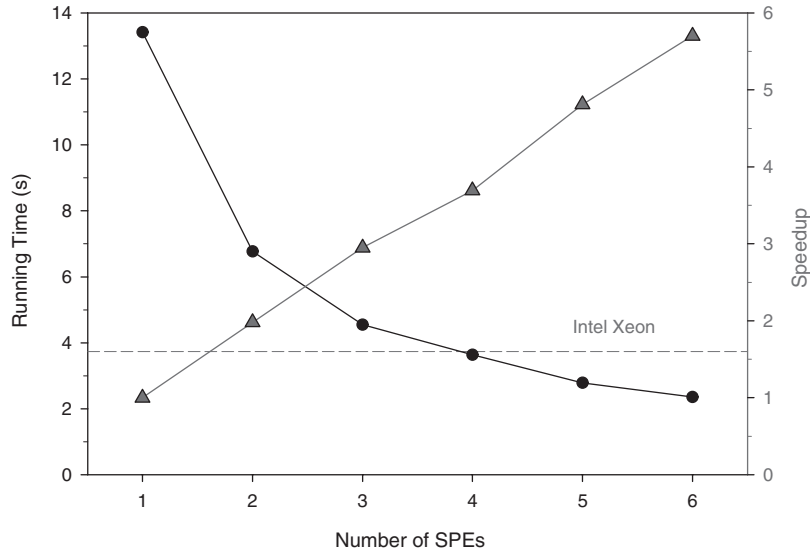
By using various techniques for programming with the Cell/B.E., we have developed what we believe is the first open-source implementation of a multimedia application optimized for this architecture. Our experience highlights some of the general techniques that may be employed for optimizing other algorithms and applications for this highly-capable architecture. These techniques should be similarly useful when implementing other decoders or image processing applications. Our choice of implementing MPEG-2 should be useful to others porting other decoders to the Cell/B.E. since MPEG-2 utilizes many similar techniques to achieve compression.

Our application achieves significant performance gains over the Intel Xeon when using 4 or more SPEs. Using 6 SPEs, the maximum available on the PS3, our application achieves a speedup of 1.587 over the Xeon on the PS3 and a speedup of 1.557 over the Xeon on the QS20. Using 16 SPEs, the maximum available on the QS20 without code modification, our application achieves a speedup of 3.088 over the Xeon.

7. Acknowledgments

This work was supported in part by an IBM Shared University Research (SUR) award and NSF Grants CNS-0614915, CAREER CCF-0611589, and DBI-0420513. We acknowledge our Sony-Toshiba-IBM Center of Competence for the use of Cell Broadband Engine resources that have contributed to this research.

Performance of MPEG-2 Decoder On Sony PlayStation 3



Performance of MPEG-2 Decoder On IBM QS20

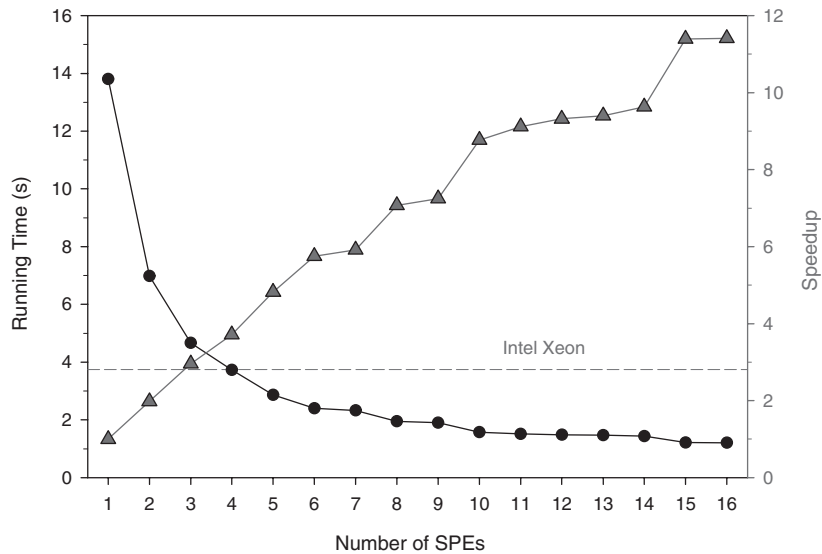


Figure 6: A comparison of the MPEG-2 decoder performance on the Sony PS3 / 3.2GHz Cell/B.E. processor (top) and IBM QS20 / two 3.2GHz Cell/B.E. processors (bottom). These plots gives the running time (in seconds) and speedup (with respect to the running time on the same platform using a single SPE) as the number of SPEs is increased to the maximum number of SPEs on each platform. Note that horizontal dashed line is the running time for the same benchmark problem using a 3.2GHz Intel Xeon processor.

References

- [1] A. Bilas, J. Fritts, and J. Singh. Real-time parallel MPEG-2 decoding in software. Technical Report TR-516-96, Department of Computer Science, Princeton University, 1996.
- [2] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe. MPEG-2 decoding in a stream programming language. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2006)*, Rhodes, Greece, Apr. 2006.
- [3] B. Flachs, S. Asano, S. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processor unit for a Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 134–135, San Francisco, CA, USA, February 2005.
- [4] M. S. S. Group. MPEG-2 encoder/decoder, version 1.2, July 1996.
- [5] C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano. The vector floating-point unit in a synergistic processor element of a Cell processor. In *Proc. 17th IEEE Symposium on Computer Arithmetic*, pages 59–67, Washington, DC, USA, 2005. IEEE (ARITH '05) Computer Society.
- [6] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
- [7] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.
- [8] M. P. E. G. (MPEG). ISO/IEC 13818-2: 1995 (e) recommendation ITU-T H.262 (1995 e).
- [9] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 184–185, San Francisco, CA, USA, February 2005.