A COMPARISON OF THREE COMMODITY-LEVEL PARALLEL ARCHITECTURES: MULTI-CORE CPU, CELL BE AND GPU

ANDRÉ RIGLAND BRODTKORB[1] AND TROND RUNAR HAGEN[1,2]

This is a draft. Final version will appear in the proceedings of the Seventh International Conference on Mathematical Methods for Curves and Surfaces

[1] SINTEF, Dept. Appl. Math., P.O. Box 124, Blindern, N-0314 Oslo, Norway
[2] Centre of Mathematics for Applications (CMA),University of Oslo, Norway
Email: {Andre.Brodtkorb,Trond.R.Hagen}@sintef.no

ABSTRACT. We explore three commodity parallel architectures: multi-core CPUs, the Cell BE processor, and graphics processing units. We have implemented four algorithms on these three architectures: solving the heat equation, inpainting using the heat equation, computing the Mandelbrot set, and MJPEG movie compression. We use these four algorithms to exemplify the benefits and drawbacks of each parallel architecture. Parallel architectures, Multi-core, Cell BE, GPU, SIMD.

1. INTRODUCTION

The gain in performance of computer programs has typically come from increased processor clock frequency and increased size of system memory. New computers have been able to handle larger problems in the same timeslot, or the same problem at greater speed. Recently, however, this trend has seemed to stop, and in the most recent years we have actually seen a *decrease* in clock frequency. The new trend is instead to increase the *number* of processor cores. There are several different multi-core designs in commodity hardware: your typical multi-core CPU consists of a few *fat* cores with a lot of complex logic; graphics processing units (GPUs) consist of several hundred *thin* processors with reduced functionality; and the Cell BE [1] consists of a mixture of fat and thin cores.

A key point to designing algorithms for these architectures is to understand their hardware characteristics. The aim of this work is to give a good understanding of how the hardware performs, with respect to a set of four example algorithms: solving the heat equation, inpainting missing pixels using the heat equation, computing the Mandelbrot set, and MJPEG movie compression. These four algorithms are chosen because they display different characteristics found in many real-world applications.

2. RELATED WORK

There has been a lot of research into high-level programming abstractions to parallel architectures. All abstractions attempt to create an intuitive and simple API with as low as possible performance penalty. APIs such as OpenMP [2] and MPI [3] show good performance and have become de facto standards in shared memory and cluster systems, respectively. These two APIs have also been exploited to create abstractions to programming the Cell BE [4, 5]. CellSs [6] is another abstraction for the Cell BE, consisting of a source to source compiler and runtime system. There have also been several high-level abstractions to programming the GPU for general purpose computation (GPGPU [7]). The most active languages today are CUDA [8] and Brook [9], where CUDA is a vendor specific language from NVIDIA, and Brook is an API with an extended version for AMD GPUs called Brook+ [10]. RapidMind [11] is a high-level C++ abstraction offering a common platform for different back-ends: multi-core CPUs, the Cell BE, and GPUs. This enables the programmer to easily run the same code on different hardware setups by simply changing the back-end. The back-ends themselves are responsible for low-level optimization and tuning, letting the programmer focus on high-level optimization of algorithms. OpenCL [12] is an API ratified by the Khronos group, in the same family of standards as OpenGL [13]. OpenCL offers a common low-level interface to program architectures such as multi-core CPUs, Cell BE and GPUs. Approved in December 2008, the first compilers have now appeared for GPUs, and the number of supported architectures is expected to rise. Such an open standard with support for multiple platforms is a great step in unifying heterogeneous programming efforts. In this work, we do not use or discuss the

aforementioned higher-level abstractions, but instead present implementations created using lower-level tools for each architecture.

The algorithms we examine have been implemented earlier on all three architectures. The heat equation and other partial differential equations (PDEs) have been implemented on the Cell BE [14] and on the GPU [15]. Inpainting using PDE-based techniques has also been implemented on the GPU [16], but not, to our knowledge, on the Cell BE. A Mandelbrot generator is part of the NVIDIA CUDA SDK code samples, and writing an optimized Mandelbrot set generator for the Cell BE was the topic for the linux.conf.au 2008 hackfest. The main building block of MJPEG movie compression is also part of the NVIDIA CUDA SDK, and MJPEG 2000 has been implemented on the Cell BE [17]. We emphasize that our aim is not to compete with these implementations, but rather to show how implementations with a similar amount of optimization can uncover differences.

3. Architectures

Multi-core CPUs, the Cell BE, and GPUs all consist of several processors, with varying type and composition. This section briefly describes these architectures and the low-level languages and tools we have used to program them. We have chosen to compare models that were released in the same time period, and argue that the intrinsic differences we identify do not change dramatically with newer generations.

Modern CPUs incorporate a multi-core design that consists of multiple fat cores on a single chip, where each core has a large cache and a lot of logic for instruction level parallelism (ILP). More than half of the chip area is typically used for cache and logic, leaving a relatively small number of transistors for pure arithmetic operations. To utilize all cores in the multi-core processor we use OpenMP, a C, C++ and Fortran API for multi-threading in a shared memory system. In C++, the API consists of a set of compiler pragmas that for example can execute loops in parallel.

The Cell BE is a heterogeneous processing unit, consisting of one power processor element (PPE) and eight synergistic processing elements (SPEs). The PPE is a regular fat CPU core, while the SPEs are thin cores capable of SIMD execution on 128 bits of properly aligned memory. The next version of the Cell BE is planned to have two PPEs and 32 SPEs. In the current version, each SPE has a small *local store* to hold their program and data, and is connected to the PPE through a fast on-chip interconnect. As opposed to the PPE, the SPEs have little or no ILP or cache, dedicating almost all transistors to floating point calculations. The API that we have used to access and program the SPEs is the SPE Runtime Management Library version 2 (libspe 2) [18]. In our use of libspe 2, the PPE program typically creates a context for each of the SPEs, and loads an SPE program (written in a subset of C++) into each context. The PPE then sets the SPEs to execute the programs. The SPEs are responsible for explicitly transferring data between local store and main memory using direct memory access (DMA). DMA executes asynchronously, meaning we can overlap memory transfer with computation. This effectively hides memory latency and transfer time. While the SPEs are computing, the PPE mainly acts as a supervisor handling support functions for the SPEs. This use is often referred to as the SPE-centric programming model, where the SPEs run the main part of the program.

The gaming industry is the main driving force behind the development of GPUs. Traditionally, the GPU only had a fixed set of graphics operations implemented in hardware, but recent generations have become programmable. Current high-end graphics cards have several hundred *stream processors* where almost all transistors are dedicated to floating point arithmetics. The NVIDIA GeForce 9800 GX2 for example, has 256 stream processors. The last generations of GPUs from NVIDIA can be programmed using CUDA, an extension to C that views the GPU as a general stream processor. CUDA typically uses the GPU to execute the same program, referred to as the *kernel*, over a large set (or stream) of data. To execute the kernel, the input data is first transferred to the GPU, and then the computational domain is divided into equally sized blocks. Each block has one virtual thread per element, and the blocks are computed independently. When the kernel has completed execution on all blocks, the data is typically read back to the CPU.

We have benchmarked the four algorithms on two different commodity level hardware setups. The CPU and GPU implementations were run on the same system, consisting of an Intel Core 2 Duo 2.4 GHz processor with 4 MiB cache, 4 GiB system memory, and an NVIDIA GeForce 8800 GTX card with 768 MiB graphics memory. The Cell BE implementations were run on a PlayStation 3, containing the 3.2GHz Cell BE processor with 6 available SPEs and 256 MiB system memory. The CPU has a theoretical performance of about 20 GFLOPS per core, and each SPE on the Cell BE has a theoretical performance of about 25 GFLOPS. The GPU has a theoretical performance of about 520 GFLOPS. All the implementations have been compiled using level three optimization and the *fast-math* option enabled. They also contain approximately the same amount of platform specific optimizations. The time and effort needed to program the different architectures, however, differs somewhat. Implementing the CPU version required less

LISTING 1. CellHeat.cpp

```
for (i=1; i<=height; ++i) {
  //Increment front, back, up, center, down, next
  DMA_REQUEST_ROW_FROM_MEMORY(input[next]);
  DMA_WAIT(input[down]);
  DMA_WAIT(output[front]);
  for(j=1; j<width-1; ++j)
    output[front][j] = 0.125f * (input[up][j]
                               + input[center][j-1]
                               + 4.0f*input[center][j]
                               + input[center][j+1]
                               + input[down][j]);
  DMA_REQUEST_ROW_TO_MEMORY(output[front]);
}
```

time and debugging efforts than the GPU version and the Cell BE version. The difference in effort comes from two main contributors: prior experience with the architecture and quality of programming tools, debuggers and profilers. If we attempt to disregard the impact of prior experience, we can give a few general remarks on programming effort. Of the three architectures, we find that the CPU implementation uses the highest level API, and requires the least programming effort. Both the GPU and Cell BE implementations on the other hand, use lower level APIs that require detailed knowledge of the hardware to avoid performance pitfalls. Thus, they require more programming effort to reach the same level of optimization as the CPU.

4. ALGORITHMS AND RESULTS

To compare the architectures, we have implemented four different algorithms: solving the heat equation with a finite difference method, inpainting missing pixels using the heat equation solver, computing the Mandelbrot set, and MJPEG movie compression. We have selected these algorithms because they exhibit different computational and memory access properties that are representative for a large range of real-world problems: the first two are memory streaming problems with regular and irregular memory access patterns, respectively, and the last two are number crunching problems with uniformly and nonuniformly distributed computation. All four algorithms can easily be parallelized and should thus fit the architectures well.

The following sections describe the algorithms and some details for the Cell BE and GPU implementations. Our CPU implementations use OpenMP to parallelize the execution, using a static scheduler for the two algorithms with a predetermined work-load and a dynamic scheduler for the two with a data-dependent work-load. We employ the `omp parallel for` pragma on the outmost loops, thus minimizing OpenMP overheads.

4.1. **The Heat Equation.** The heat equation describes how heat dissipates in a medium. In the case of a 2D homogeneous and isotropic medium, it can be written as $u_t = a(u_{xx} + u_{yy})$, where a is a material specific constant. Using an explicit finite difference scheme, the unknown solution $u_{i,j}^{n+1}$ in grid point (ih, jh) at time $(n+1)k$ is given by

$$u_{i,j}^{n+1} = a\frac{k}{h^2}\left(u_{i-1,j}^{n} + u_{i+1,j}^{n} + 4u_{i,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}\right), \quad (1)$$

where h and k determine the spatial and temporal resolution, respectively.

Characteristics -. The heat equation benchmark shows how efficient each architecture is at streaming data. Solving the heat equation using this explicit finite difference scheme requires approximately the same number of memory accesses as arithmetic instructions. As the processor clock speeds are much higher than the memory clock speeds on our architectures, the performance should be limited by memory bandwidth. The memory access pattern, however, is regular and thus enables efficient hardware pre-fetching techniques on the multi-core CPU. Thus, the multi-core CPU should get full use of its large cache as long as the two cores keep from fighting over the same set of cache lines. On the Cell BE we can try to hide memory access by overlapping this with computation, and on the GPU explicitly gather the data into fast on-chip memory called *shared memory*.

This algorithm displays properties found when solving other PDEs using explicit finite difference/element/volume methods. Image filtering, such as convolution, is another class of algorithms with the same properties.

Implementation -. By examining the computational molecule for $u_{i,j}^{n+1}$, we see that it depends on the value at grid-point (ih, jh) and the four closest neighbours in the previous time-step. Because of this dependency, we cannot simply update the value of $u_{i,j}^{n}$ in-place. By having two buffers, we store the result of odd time-steps in the "odd"

buffer, reading from the "even" buffer, and conversely for even time-steps. This way we can update all elements in parallel at each time-step.

On the Cell BE, we divide the domain into blocks with an equal number of rows, and let each SPE calculate its sub-domain. Listing 1 shows the Cell BE algorithm in pseudo code, and Figure 1 is a graphical representation. For each row we want to compute in the output, we need to explicitly gather the needed data from main memory to local store and write the result back. Using asynchronous memory transfer, we can overlap with computation, and only keep four rows of input in local store at any given time. We have used intrinsic SIMD instructions in the actual implementation, as the SPEs are SIMD processors.

On the GPU, we divide the computational domain into blocks with one virtual thread per element. For each block, each thread reads one element from the GPUs main memory into shared memory. The shared memory is fast on-chip memory accessible to all virtual threads in the same block. We also have to read the *apron* (also called ghost cells) into shared memory. The apron consists of the data-elements outside the block that we need for our computational molecule. Figure 1 shows the block and its apron. When all data has been read into shared memory, each thread computes one output element and writes the result to GPU main memory.

Results -. Figure 2 shows the runtime of computing one step of the heat equation on the different architectures. For the GPU, this includes the time it takes to read the result back from graphics memory. The other two architectures place the result in main memory without the explicit read-back. Table 1 breaks the GPU and Cell BE run-times up into time spent computing and time spent waiting for memory transfers to complete. The table shows that the GPU suffers from an expensive read-back of data from the GPU to the CPU, whilst the Cell BE is able to hide much of the memory reads and writes by overlapping with computation. This explains why the GPU barely beats the multi-core CPU implementation. The GPU, however, will outperform the other architectures if the data is allowed to remain on the GPU throughout multiple time-steps.

4.2. **Inpainting.** Noisy images (e.g., from poor television reception) can be repaired by *inpainting*. Here we use the heat equation on masked areas as a naïve example of inpainting. Using this approach, information from the area surrounding a block of noisy pixels will be diffused into the block and fill in the missing values. Technically, each masked element is updated using Equation (1), whereas we set $u_{i,j}^{n+1} = u_{i,j}^{n}$ for unmasked elements.

Characteristics -. The inpainting benchmark shows how efficient each architecture is at streaming data, executing conditionals, and computing on a subset of the data. Inpainting using the heat equation requires even fewer arithmetic instructions per memory access than solving the heat equation. This is because we have to read the mask for all elements, but only run computations on a few of them. This memory access pattern will further make matters worse, as the CPU will underutilize its cache. The GPU will have problems with the added conditional as all processors in one *warp* (currently 32 stream processors) are forced to run commands synchronously. The effect is that even if only one processor in the warp branches differently from the others, the whole warp must evaluate both sides of the branch.

This algorithm has properties that are also found in other algorithms with a lot of memory access and little computation, e.g., image processing algorithms working on masked areas.

Implementation -. Both the Cell BE and GPU implementations of this algorithm are very similar to the heat equation solver. The major difference is that the Cell BE and GPU must explicitly gather the mask into local store and shared memory, respectively.

Results -. For each image resolution, we used a noise mask to inpaint approximately 5% of the image using ten time-steps of the heat equation. Figure 2 shows that the GPU performs relatively better compared to the heat equation solver. This is because the GPU is able to keep the data in graphics memory throughout the ten passes. This lessens the effect of the single read-back, as shown in Table 1. Because the GPU executes one warp synchronously, it uses about the

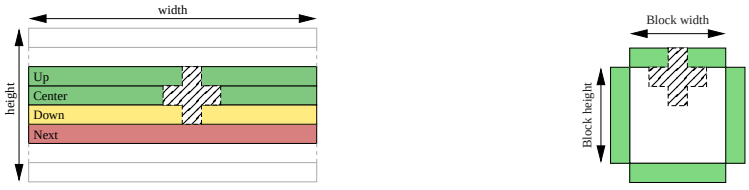

FIGURE 1. The heat equation computed on the Cell BE (left) and on the GPU (right).

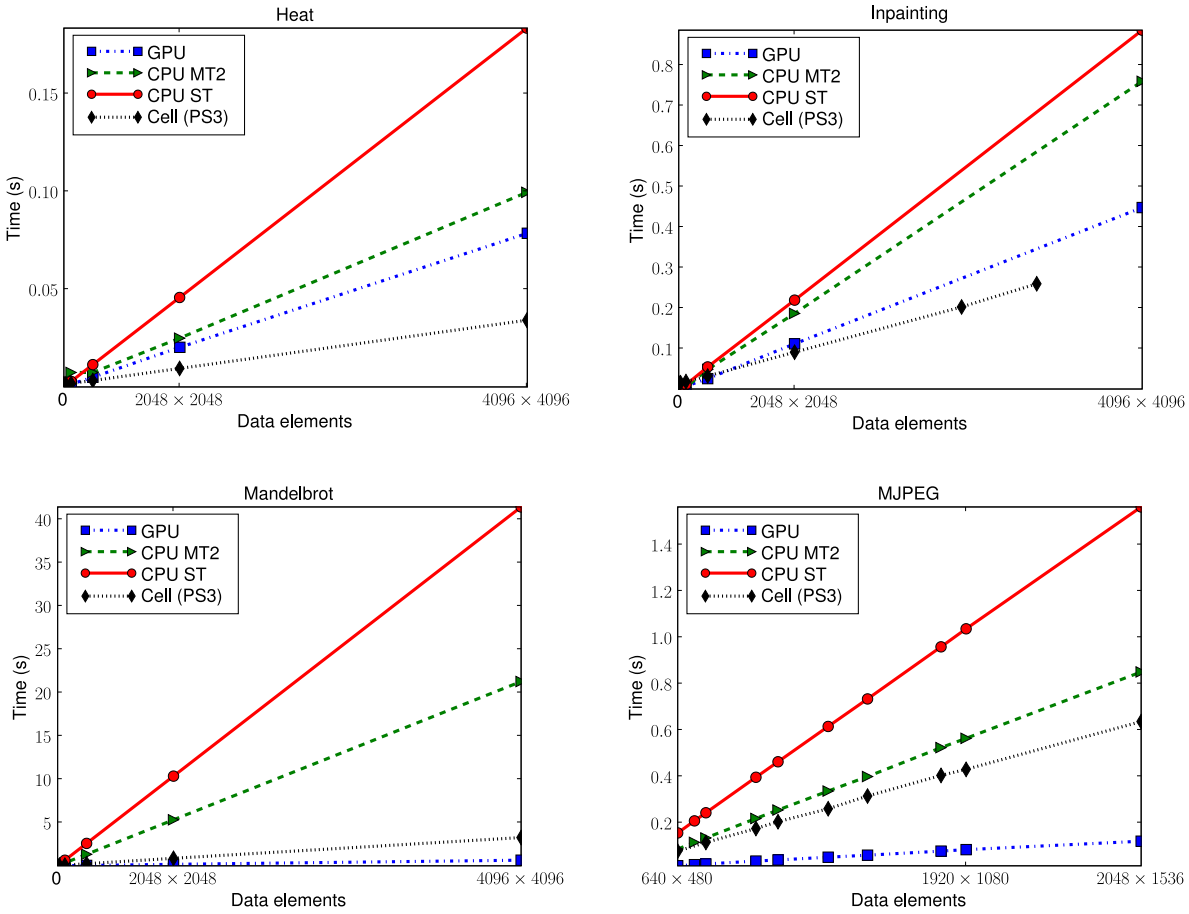

FIGURE 2. Runtime of the four different algorithms: the heat equation (top left), inpainting (top right), the Mandelbrot set (bottom left), MJPEG (bottom right). CPU ST represents the serial CPU version, and CPU MT2 represents the multi-threaded CPU version running on two CPU cores.

TABLE 1. Breakdown of running times. For the GPU, the percentages represent kernel execution time and GPU-CPU transfer time. For the Cell BE, the time represents time spent computing, and stall time while waiting for DMA requests to complete.

		Heat	Inpainting	Mandelbrot	MJPEG
GPU	Memory	75%	55%	0%	80%
	Computation	25%	45%	100%	20%
Cell BE	Memory	10%	10%	0%	5%
	Computation	90%	90%	100%	95%

same time to complete one pass of inpainting, as it does to complete one pass of the whole heat equation. The Cell BE computes four elements synchronously using intrinsic SIMD instructions. However, it only runs slightly faster per pass than the heat equation. This can be explained by the fact that the SPEs do not contain a conditional branch predictor, making the branch almost as expensive as computing the heat equation itself. Using a computationally more demanding algorithm would diminish the effect of the branch on the Cell BE. There was also not enough physical system memory to benchmark domains larger than approximately 3600×3600 for the Cell BE. The single threaded CPU version runs much faster per pass than the heat equation, whilst the multi-threaded version only has a marginal speedup. This can be explained by the increased load on the memory bus, as the additional mask has to be loaded

into cache, compared to the heat equation solver alone. Using multiple cores does not increase performance when the bottleneck is the memory bandwidth.

4.3. **The Mandelbrot Set.** The Mandelbrot set is a fractal that has a very simple recursive definition:

$$M = \left\{ c \in \mathbb{C} : z_0 = c, \quad z_{n+1} = z_n^2 + c, \quad \sup_{n \in \mathbb{N}} |z_n| < \infty \right\}. \tag{2}$$

Informally, it is the set of all complex numbers that do not tend towards infinity when computing z_{n+1}. When computing the Mandelbrot set, one uses the fact that c belongs to M if and only if $|z_n| < 2$ for all $n \geq 0$. Typically, one picks a set of discrete complex points C and a fixed m, and then the point $c \in C$ is assumed to be in the set if $|z_n| < 2$ for all $n \leq m$.

Characteristics -. This benchmark shows how well each architecture performs floating point operations, and how it copes with dynamic workloads. Computing whether a complex coordinate belongs to the Mandelbrot set requires a lot of computation, and only a single memory write. For coordinates that are in the set the program has to compute all m iterations while coordinates outside often compute only a few. This means that neighbouring pixels often have very different workloads, as the boundary has a highly complex geometry.

Computing the Mandelbrot set exhibits properties also found in algorithms such as ray-tracing and ray-casting, as well as many other iterative algorithms. Algorithms with a lot of computation per memory access, such as protein folding, also show these properties.

Implementation -. Using the abscissa as the real part and the ordinate as the imaginary part we create a set of complex numbers C. For each point $c \in C$, a while-loop computes z_n until $|z_n| > 2$ or $n > m$. The pixel is colored using the value of n when the loop terminates, yielding a gray-scale image where all white pixels are assumed to be part of the Mandelbrot set.

In our Cell BE implementation, we partition the domain into lines, where only the real part of c varies between pixels. We know that the computational time for each line can differ drastically, so we use a dynamic load distribution. The PPE simply fills a fixed length queue with line-numbers for each SPE, and the SPEs then start processing their queue. The PPE continues to enqueue line-numbers in non-full queues until the whole domain has been enqueued. The actual computation is done using SIMD instructions to utilize the hardware.

Results -. Table 1 shows that computing whether a complex coordinate belongs to the Mandelbrot set or not is computationally intensive, and Figure 2 shows that both the GPU and Cell BE perform very well. Using multiple CPU cores scales perfectly. On the GPU, it does not drastically affect the results that two pixels close to each other can have very different workloads. Even though each warp is executed synchronously, most warps simply contain only pixels within the set, or outside it. Thus, the number of warps with a mixture of pixels within and outside the set is often far less than the number of warps with a relatively homogeneous workload.

4.4. **MJPEG.** MJPEG is an "industry standard" for compression of a movie stream. The main part of the algorithm consists of dividing each frame into 8×8 blocks, computing the discrete cosine transform (DCT), and then quantizing each block:

$$\begin{aligned} p_{u,v} &= \alpha(u)\alpha(v) \sum_{i=0}^{7} \sum_{j=0}^{7} g_{i,j} \cos\left[\frac{\pi}{8}(i+0.5)\,u\right] \cos\left[\frac{\pi}{8}(j+0.5)\,v\right], \\ & \alpha(n) = \begin{cases} \sqrt{1/8} & , n = 0 \\ \sqrt{2/8} & , n \neq 0 \end{cases}, \\ r_{u,v} &= \text{round}\left(p_{u,v}/q_{u,v}\right). \end{aligned}$$

Here, $g_{i,j}$ is the element (i, j) from the 8×8 block, $p_{u,v}$ is the amplitude of frequency (u, v), $q_{u,v}$ is element (u, v) of the quantization matrix, and $r_{u,v}$ is the result.

Characteristics -. The MJPEG results show how efficient each architecture is with a typical data flow, where the computationally intensive part of the code is accelerated, and the rest of the code runs on a single CPU core. Computing the DCT and then quantizing is an algorithm that is both cache friendly and requires a lot of computation per memory access. The performance should therefore be limited by processing power rather than memory access. Computing the cosines is typically also very expensive.

The MJPEG algorithm is representative for many image and movie compression algorithms, as they all do a lot of computation per memory access. These properties are also displayed in computationally heavy image processing algorithms, such as computing the FFT and image registration.

Implementation -. To optimize for DMA transfer, our Cell BE version computes a row of blocks. Since the total amount of work is constant in each block-line, we use a static load distribution, dividing the domain into an equal number of block-lines per SPE. The SPE then computes each block in each block-line until there are no more block-lines to compute.

MJPEG compression fits the GPU perfectly, as CUDA already assumes that computation should be split up into blocks. We simply use a block-size of 8×8 and let CUDA automatically schedule the blocks in our GPU implementation.

Results -. Figure 2 shows the time it takes to compute the DCT, quantize, and Huffman code one image consisting of an intensity channel and two chrominance channels with half the horizontal and vertical resolution of the intensity image. Table 1 breaks down the time spent on DCT and quantization, as these are the parts that have been accelerated. The GPU is heavily penalized for overhead connected with starting each memory transfer. It has to upload the three images and download them again after the computations. The overhead with starting these six memory transfers is substantial. The Cell BE does not suffer from such overheads, but is able to overlap almost all memory access by computation. However, Huffman coding using the PPE is very slow. The PPE takes 50% more time to complete compared to the CPU, even though it is the exact same code.

5. Summary

We have examined how a set of four algorithms perform on three sets of commodity level parallel hardware. All the algorithms we displayed, except inpainting, scaled well on the CPU. The inpainting algorithm saturates the memory bus, which again limits performance. The Cell BE performs well on algorithms where memory access can be hidden by computations, and when it comes to raw floating point performance. However, it has a relatively slow PPE core that can limit performance. The limited system memory on the PlayStation 3 can certainly also be a problem. The GPU is, by far, the best performing architecture for computationally intensive operations, but transferring memory to and from the graphics card can be very expensive.

This work has focused on four applications that show different properties of the three architectures. This is a small comparison, and it would be of great use to broaden the number of algorithms, architectures, and programming languages to give a broader understanding of commodity level parallel architectures.

The authors would like to thank the anonymous reviewers for their thorough comments and feedback.

References

[1] IBM, Sony, Toshiba: Cell Broadband Engine programming handbook version 1.1 (2007)
[2] OpenMP Architecture Review Board: OpenMP application program interface version 2.5 (2005)
[3] Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (2003)
[4] Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the Cell Broadband Engine processor. IBM Systems Journal **45** (2006) 85–102
[5] Eichenberger, A., O'Brien, J., O'Brien, K., Wu, P., Chen, T., Oden, P., Prener, D., Shepherd, J., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the Cell processor. In: Intl. Conf. on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2005) 161–172
[6] Bellens, P., Perez, J., Badia, R., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SuperComputing'06. (2006)
[7] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26** (2007) 80–113
[8] NVIDIA corporation: NVIDIA CUDA programming guide version 1.1 (2007)
[9] Buck, I., Foley, T., Horn, D., Sugerman, J., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware (2004)
[10] AMD Corporation: AMD stream computing revision 1.3.0 (2008)
[11] McCool, M.: Data-parallel programming on the Cell BE and the GPU using the rapidmind development platform (2006) GSPx Multicore Applications Conference.
[12] Khronos OpenCL Working Group: The OpenCL specification 1.0 (2008)
[13] OpenGL Architecture Review Board, Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL. 6th edn. Addison-Wesley (2007)
[14] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: Computing Frontiers '06. (2006)

[15] Hagen, T., Henriksen, M., Hjelmervik, J., Lie, K.A.: How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine. Geometric Modelling, Numerical Simulation and Optimization: Industrial Mathematics at SINTEF, Springer verlag (2007)
[16] Hagen, T., Rahman, T.: GPU-based image inpainting using a TV-Stokes equation. Preprint (2008)
[17] Muta, H., Doi, M., Nakano, H., Mori, Y.: Multilevel parallelization on the Cell / B.E. for a motion JPEG 2000 encoding server. In: MULTIMEDIA '07. (2007)
[18] IBM: Software development kit for multicore acceleration version 3.0 (2007)