Tips, Tricks and Troubles: Optimizing for Cell and GPU

Håkon Kvale Stensland, Håvard Espeland, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{haakonks, haavares, griff, paalh}@simula.no

ABSTRACT

When used efficiently, modern multicore architectures, such as Cell and GPUs, provide the processing power required by resource demanding multimedia workloads. However, the diversity of resources exposed to the programmers, intrinsically requires specific mindsets for efficiently utilizing these resources - not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion-JPEG implementations indicates that there exists a large potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices, inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks and troubles with respect to efficient utilization of the available resources.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUE**]: Concurrent Programming—*Parallel programming*

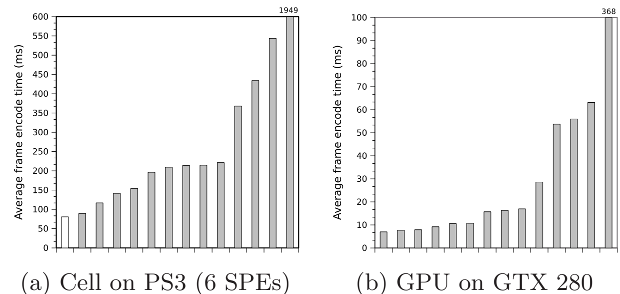
General Terms

Measurement, Performance

1. INTRODUCTION

Heterogeneous systems like the STI Cell Broadband Engine (Cell) and PCs with Nvidia graphical processing units (GPUs) have recently received a lot of attention. They provide more computing power than traditional single-core systems, but it is a challenge to use the available resources efficiently. Processing cores have different strengths and weaknesses than desktop processors, the use of several different types and sizes of memory is exposed to the developer, and limited architectual resources require considerations concerning data and code granularity.

We want to learn how to *think* when the multicore system at our disposal is a Cell or a GPU. We aim to understand how to use the resources efficiently, and point out tips, tricks and troubles, as a small step towards a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we have looked at effective programming for the workload-intensive yet relatively straight-forward Motion-JPEG (MJPEG) video encoding task. Here, a lot of CPU cycles are consumed in the sequential discrete cosine transformation (DCT), quantization and compression stages. On single core systems, it is almost impossible to process a 1080p high definition video in real-time, so it is reasonable to apply multicore computing in this scenario.



(a) Cell on PS3 (6 SPEs) (b) GPU on GTX 280

Figure 1: Runtime for MJPEG implementations.

Our comparison of 14 different implementations on both Cell and GPU gives a good indication that the two considered architectures are complex to use, and that achieving high performance is not trivial. Derived from a sequential codebase, these multicore implementations differ in terms of algorithms used, resource utilization and coding efficiency. Figure 1 shows performance results for encoding the "tractor" video clip[1] in 4:2:0 HD. The differences between the fastest and slowest solution are 1869 ms and 362 ms per frame on Cell and GPU, respectively, and it is worth noting that the fastest solutions were disk I/O-bound. To gain experience of what works and what does not, we have examined these solutions. We have not considered coding style, but revisited algorithmic choices, inter-core data communication (memory transfers) and use of architecture-specific capabilities.

In general, we found that these architectures have large potentials, but also many possible pitfalls, both when choosing specific algorithms and for implementation-specific decisions. The way of thinking cross-platform is substantially different, making it an art to use them efficiently.

[1]Available at ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/test_sequences/1080p/tractor.yuv

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NOSSDAV'10, June 2–4, 2010, Amsterdam, The Netherlands.
Copyright 2010 ACM 978-1-4503-0043-8/10/06 ...$10.00.

2. BACKGROUND

2.1 SIMD and SIMT

Multimedia applications frequently perform identical operations on large data sets. This has been exploited by bringing the concept of SIMD (single instruction, multiple data) to desktop CPUs, as well as the Cell, where a SIMD instruction operates on a short vector of data, e.g., 128-bits for the Cell SPE. Although SIMD instructions have become mainstream with the earliest Pentium processors and the adoption of PowerPC for MacOS, it has remained an art to use them. On the Cell, SIMD instructions are used explicitly through the vector extensions to C/C++, which allow basic arithmetic operations on vector data types of intrinsic values. It means that the programmer can apply a sequential programming model, but needs to adapt memory layout and algorithms to the use of SIMD vectors and operations.

Nvidia uses an abstraction called SIMT (single-instruction, multiple thread). SIMT enables code that uses only well-known intrinsic types but that can be massively threaded. The runtime system of the GPU schedules these threads in groups (called warps) whose optimal size is hardware-specific. The control flow of such threads can diverge like in an arbitrary program, but this will essentially serialize all threads of the block. If it does not diverge and all threads in a group execute the same operation or no operation at all in a step, then this operation is performed as a vector operation containing the data of all threads in the block.

The functionality that is provided by SIMD and SIMT is very similar. In SIMD programming, vectors are used explicitly by the programmer, who may think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types.

2.2 STI Cell Broadband Engine

The Cell Broadband Engine is developed by Sony Computer Entertainment, Toshiba and IBM. As shown in Figure 2, the central components are a Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPE) connected by the Element Interconnect Bus (EIB). The PPE contains a general purpose 64-bit PowerPC RISC core, capable of executing two simultaneous hardware threads. The main purpose of the PPE is to control the SPEs, run an operating system and manage system resources. It also includes a standard Altivec-compatible SIMD unit. An SPE contains a Synergistic Processing Unit and a Memory Flow controller. It works on a small (256KB) very fast memory, known as the local storage, which is used both for code and data without any segmentation. The Memory Flow Controller is used to transfer data between the system memory and local storage using explicit DMA transfers, which can be issued both from the SPE and PPE.

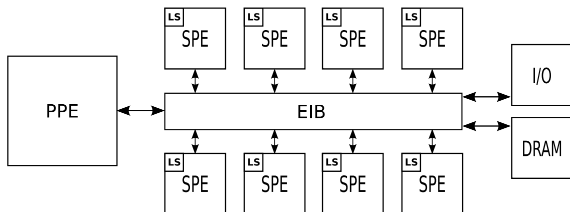


Figure 2: Cell Broadband Engine Architecture

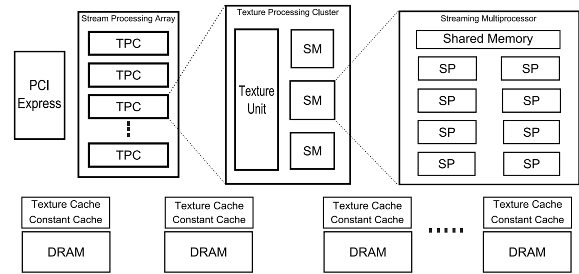


Figure 3: Nvidia GT200 Architecture

2.3 Nvidia Graphics Processing Units

A GPU is a dedicated graphics rendering device, and modern GPUs have a parallel structure, making them effective for doing general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia has released the CUDA framework with a programming language similar to ANSI C. In CUDA, the SIMT abstraction is used for handling thousands of threads.

The latest generation available from Nvidia (GT200) is shown in Figure 3. The GT200 chip is presented to the programmer as a highly parallel, multi-threaded, multi-core processor - connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters (TPC) with 3 streaming multiprocessors (SM). A single SM contains 8 stream processors (SP) which are the basic ALUs for doing calculations. GPUs have other memory hierarchies than an x86 processor. Several types of memory with different properties are available. An application (*kernel*) has exclusive control over the memory. Each thread has a private local memory, and the threads running on the same stream multiprocessor (SM) have access to a shared memory. Two additional read-only memory spaces called constant and texture are available to all threads. Finally, there is the global memory that can be accessed by all threads. Global memory is not cached, and it is important that the programmer ensures that running threads perform *coalesced* memory accesses. Such a coalesced memory access requires that the threads' accesses occur in a regular pattern and creates one large access from several small ones. Memory accesses that cannot be combined are called uncoalesced.

3. EXPERIMENTS

By learning from the design choices of the implementations in Figure 1, we designed experiments to investigate how performance improvements were achieved on both Cell and GPU. We wanted to quantify the impact of design decisions on these architectures.

All experiments encode HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* test sequence. However, we used only the first frame of the sequence and encode it 1000 times in each experiment to overcome the disk I/O bottleneck limit. This becomes apparent at the highest level of encoding performance since we did not have a high bandwidth video source available. All programs have been compiled with the highest level of compiler optimizations using gcc and nvcc, respectively, for Cell and GPU. The Cell experiments have been tested on a QS22 bladeserver (8 SPEs, the results from Figure 1 were on a PS3 with 6 SPEs) and the GPU experiments on a GeForce GTX 280 card.

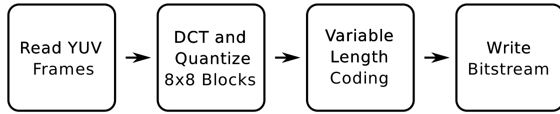


Figure 4: Overview of the MJPEG encoding process

3.1 Motion JPEG Encoding

The MJPEG format is widely used by webcams and other embedded systems. It is similar to videocodecs such as Apple ProRes and VC-3, used for video editing and postprocessing due to their flexibilty and speed, hence the lack of inter-prediction between frames. As shown in Figure 4, the encoding process of MJPEG comprises splitting the video frames in 8x8 macroblocks, each of which must be individually transformed to the frequency domain by forward discrete cosine transform (DCT) and quantized before the output is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC, our encoder uses predefined Huffman tables for compression of the DCT coefficients of each macroblock. The VLC step is not context adaptative, and macroblocks can thus be compressed independently. The length of the resulting bitstream, however, is probably not a multiple of eight, and most such blocks must be bit-shifted completely when the final bitstream is created.

The MJPEG format provides many layers of parallelism; Starting with the many independent operations of calulating DCT, the macroblocks can be transformed and quantized in arbitrary order, also frames and color components can be encoded separately. In addition, every frame is entropy-coded separately. Thus, many frames can be encoded in parallel before merging the resulting frame output bitstreams. This gives a very fine-level granularity of parallel tasks, providing great flexibility in how to implement the encoder. It is worth noting that many problems have much tighter data dependencies than we observe in the MJPEG case, but the general ideas for optimizing individual parts pointed out in this paper stand regardless of whether the problem is limited by dependencies or not.

The forward 2D DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D-plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label 1D-plain uses two consecutive 1D transformations with a transpose operation in between and after. This avoids symmetries, and the 1D transformation can be optimized further. One optimization uses the AAN algorithm, originally proposed by Arai et al. [1] and further refined by Kovac and Ranganathan [5]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the postscale operation, and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [2].

More algorithms for calculating DCT exist, but they are not covered here. We have implemented the different DCT algorithms as scalar single-threaded versions on x86 (Intel Core i5 750). The performance details for encoding HD video were captured using oprofile and can be seen in Figure 5. The plot shows that the 1D-AAN algorithm using two transpose operations was the fastest in this scenario, with the 2D-matrix version as number two. The average encoding time for a single frame using 2D-plain is more than 9 times that of a frame encoded using 1D-AAN. For all algorithms, the DCT step consumed most CPU cycles.

3.2 Cell Broadband Engine Experiments

Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts is available for mapping the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but also the entropy coding step can run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores, and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We solved this by adding a frame queue between the frame reader and the DCT step, and another queue between the DCT and VLC steps. Since a frame is processed in full by a single processor, the AAN algorithm is well suited for the Cell. It can be implemented in a straight-forward manner for running on SPEs, with VLC coding placed on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD) resulting in double encoding throughput, which can be seen in Figure 6 (Scalar- and Vector/PPE).

Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk in addition to dispatching jobs to SPEs. To be able to do this, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, i.e., two rows of luma and a single row of both chroma channels. The results show that

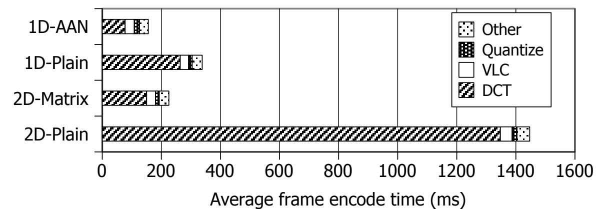


Figure 5: MJPEG encode time on single thread x86

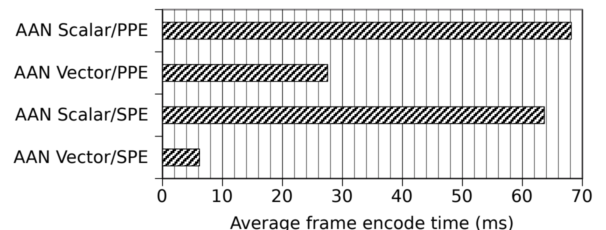


Figure 6: Encoding performance on Cell with different implementations of AAN and VLC placement

the previous encoding speed was limited by the VLC as can be seen in Figure 6 (Scalar- and Vector/SPE).

To get some insight into SPE utilization, we collected a trace (using pdtr, part of IBM SDK for Cell) showing how much time is spent on the encoding parts. Figure 7 shows the SPE utilization when encoding HD frames for the Scalar- and Vector/SPE from Figure 6. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand-code SIMD intrinsics to obtain high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version, and additionally 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

With the vector version of DCT and quantization, the VLC coding uses about 80 % of each SPE. This can possibly be optimized further, but we did not find time to pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to get high performance on this architecture. In particular, correctly implementing an algorithm using vector intrinsics is imperative. Of the 14 implementations for Cell in Figure 1, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization, and the Vector/SPE implementation in Figure 6 is a combination of these two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit-shifting of large fields on Cell as well as GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles, as seen in Figure 5, and the awkward process of gathering profiling data on multicore systems later on. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work, and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are coupled loosely. In that way, they can both be replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 1 to find out what differentiates the fastest from the medium speed implementations, we found some distinguising features: The most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfers and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed, DMA operations that left SPEs in a stalled state waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture, and we have only touched upon a few of them. Some of these are

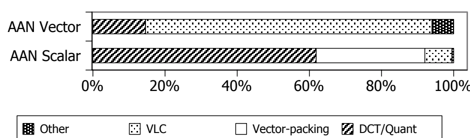


Figure 7: SPE utilization using scalar or vector DCT

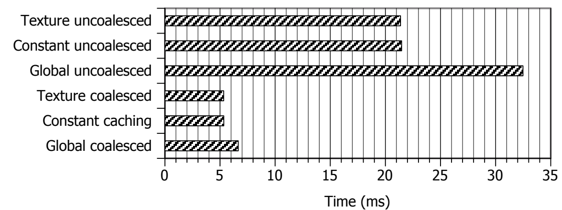


Figure 8: Optimization of GPU memory accesses

obvious, but not all, and to get acceptable performance out of a program running on the Cell architecture may require multiple iterations, restructuring and even rewrites.

3.3 GPU Experiments

As for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Coalesced memory accesses are known to have large performance impacts. However, few quantified results exist, and efficient usage of memory types, alignment and access patterns remains an art. Weimer et al. [11] experimented with bank conflicts in shared memory, but to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and write data to and from memory with both uncoalesced and coalesced access patterns [7], and used the Nvidia CUDA Visual Profiler to isolate the GPU-time for the different kernels.

Figure 8 shows that an uncoalesced access pattern decreases throughput in the order of four times due to the increased number of memory transactions. Constant and texture memory are cached, and the performance for uncoalesced accesses to them is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative even when cached memory types are used. It is also important to make sure the memory accesses are correct according to the specifications of particular GPUs because the optimal access patterns vary between GPU generations.

To find out how memory accesses and other optimizations affect programs like a MJPEG encoder, we experimented with different DCT implementations. Our baseline DCT algorithm is the 2D-plain algorithm. The only optimizations in this implementation are that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. As we observed in Figure 8, cached memory spaces improve performance compared to global memory, especially when memory accesses are uncoalesced. The second implementation, referred to as 2D-plain optimized, is tuned to run efficiently using principles from the CUDA Best Practices Guide [6]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using boolean arithmetics and manual loop unrolling. Our third implementation, the 1D-AAN algorithm, is based upon the scalar implementation

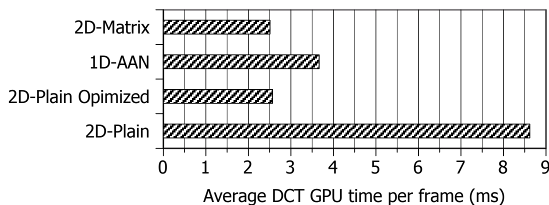


Figure 9: DCT performance on GPU

used on the Cell. Every macroblock is processed with eight threads, one thread per row of eight pixels. The input image is stored in cached texture memory, shared memory is used for temporarily storing data during processing. Finally, the 2D-matrix DCT using matrix multiplications where each matrix element is computed by a thread. The input image is stored in cached texture memory, and shared memory is used for storing data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU were used. Transferring data over the PCI Express bus is expensive, and asynchronous transfers help us reuse the kernels and hides some of the PCI Express latency by transferring data in the background.

To isolate the DCT performance, we used the CUDA Visual Profiler. The profiling results of the different implementations can be seen in Figure 9, and we can observe that the 2D-plain optimized algorithm is faster than AAN. The 2D-plain algorithm requires significantly more computations than the others, but by correctly implementing it, we get almost as good performance as with the 2D-matrix. The AAN algorithm, which does the least amount of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

This experiment shows that for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the theoretical complexity of an algorithm.

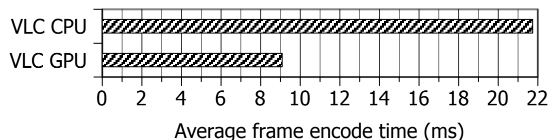


Figure 10: Effect of offloading VLC to the GPU

The last GPU experiment considers entropy coding on the GPU. As for the Cell, VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host, or by using atomic OR on the GPU. For the experiments here, we chose the former since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 10 shows the results of an experiment that compares MJPEG with AAN DCT with VLC performed on the host and on the GPU, respectively. We achieved a doubling of the encoding performance when running VLC on the GPU. In this particular case offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or faster GPU, the encoder will still scale.

4. DISCUSSION

Heterogeneous architectures like Cell and GPU provide large amounts of processing power, and achieving encoding throughputs of 480 MB/s and 465 MB/s, respectively, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions, and that there may be large differences in the way a programmer must think.

The architectures of GPU and Cell are very different, and in this respect, some algorithms may be more suited than others. This can be seen in the experiments, where the AAN algorithm for DCT calculation performed best on both x86 and Cell, but did not achieve the highest throughput on GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

The programming models used on Cell and GPU mandate two different ways of thinking parallel. The approach of Cell is very similar to multi-threaded programming on x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches, and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads, and only synchronization through barriers. This hides the architecture complexity, and is therefore a simpler concept to grasp for some programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figure 1. However, to get the highest possible performance, the programmer must also understand the nitty details of the architecture to avoid pitfalls like warp divergence and uncoalesced memory accesses.

Deciding at which granularity the data should be partitioned is very hard to do correct *a priori*. The best granularity for a given problem differs with the architecture and even different models of the same architecture. One approach towards accomplishing this is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data partitioning, code partitioning is hard to do correctly in advance. In general, a rule of thumb is to write modular code to allow moving the parts to other cores. Also, a fine granularity is beneficial, since small modules can be merged again, and also be executed repeatedly with small overhead. Offloading is by itself advantageous as resources on the main processor become available for other tasks. It also improves scalability of the program with new

generations of hardware. In our MJPEG implementations, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance on both Cell and GPU, but it may not always be the case that offloading provides higher throughput.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort for accomplishing this throughput was much higher on the Cell. This was mainly caused by the tedious process of writing a SIMD version of the encoder. Porting the encoder to the GPU in a straight-forward manner without significant optimizations for the architecture yielded a very good offloading performance compared to native x86. This indicates that the GPU is easier to use, but to reap the full potential of the architecture, one must have the same deep level of understanding as with the Cell architecture.

5. RELATED WORK

Heterogeneous multi-core platforms like the Cell and GPUs have attracted a considerable amount of research that aims at optimizing specific applications for the different architectures such as [9] and [4]. However, little work has been done to compare general optimization details of different heterogeneous architectures. Amesfoort et al. [10] have evaluated different multicore platforms for data-intensive kernels. The platforms are evaluated in terms of application performance, programming effort and cost. Colic et al. [3] look at the application of optimizing motion estimation on GPUs and quantify impact of design choices. The workload investigated in this paper is different from the workload we benchmark in our experiments, but they show a similar trend as our GPU experiments. They also conclude that elegant solutions are not easily achievable, and that it takes time, practice and experience to reap the full potential of the architecture. Petrini et al. [8] implement a communication-heavy radiation transport problem on Cell. They conclude that it is a good approach to think about problems in terms of five dimensions and partitioning them into: process parallelism at a very large scale, thread-level parallelism that handles inner loops, data-streaming parallelism that double-buffers data for each loop, vector parallelism that uses SIMD functions within a loop, and pipeline parallelism that overlaps data access with computations by threading. From our MJPEG implementations we observed that programmers had difficulties thinking parallel in two dimensions. This level of multi-dimensional considerations strengthens our statement that intrinsic knowledge of the system is essential to reap full performance of heterogeneous architectures.

6. CONCLUSION

Heterogeneous, multicore architectures like Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial, and in order to learn how to think and use the resources efficiently, we have experimentally evaluated several issues to find the tricks and troubles.

In general, there are some similarities, but the way of thinking must be substantially different - not only compared to an x86 architecture, but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account both when choosing a specific algorithm and making implementation-specific decisions. A lot of trust is put on the compilers of development frameworks and new languages like Open CL, which are supposed to be a "recompile-only" solution. However, to tune performance, the application must still be hand-optimized for different versions of the GPUs and Cells available.

Acknowledgements

The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and National Science Foundation, for the use of Cell Broadband Engine resources. We also acknowledge Alexander Ottesen, Ståle Kristoffersen, Øystein Gyland, Kristoffer Egil Bonarjee, Kjetil Endal and Kristian Evensen for their contributions.

7. REFERENCES

[1] Arai, Y., Agui, T., and Nakajima, M. A fast dct-sq scheme for images. *Transactions of IEICE E71*, 11 (1988).

[2] Cabeen, K., and Gent, P. Image compression and the discrete cosine transform. In *Math 45*, College of the Redwoods.

[3] Colic, A., Kalva, H., and Furht, B. Exploring nvidia-cuda for video coding. In *ACM SIGMM conference on Multimedia systems (MMSys)* (2010), ACM, pp. 13–22.

[4] Curry, M., Skjellum, A., Ward, H., and Brightwell, R. Accelerating reed-solomon coding in raid systems with gpus. In *International Parallel and Distributed Processing Symposium (IPDPS)* (April 2008), IEEE, pp. 1–6.

[5] Kovac, M., and Ranganathan, N. JAGUAR: A fully pipelined VLSI architecture for JPEG image compression standard. *Proceedings of the IEEE 83*, 2 (1995).

[6] Nvidia. Nvidia cuda c programming best practices guide 2.3, 2009.

[7] Ottesen, A. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2009.

[8] Petrini, F., Fossuma, G., Fernandez, J., Varbanescu, A. L., Kistler, M., and Perrone, M. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS)* (March 2007), IEEE, pp. 1–10.

[9] Sachdeva, V., Kistler, M., Speight, E., and Tzeng, T.-H. K. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. *Parallel Computing 34*, 11, 616–626.

[10] van Amsesfoort, A., Varbanescu, A., Sips, H. J., and van Nieuwpoort, R. Evaluating multi-core platforms for hpc data-intensive kernels. In *ACM Conference on Computing Frontiers (ICCF)* (2009).

[11] Weimer, W., Boyer, M., and Skadron, K. Automated dynamic analysis of cuda programs. In *Third Workshop on software Tools for MultiCore Systems (STMCS)* (2008).