Fall 2008

H.264 encoding on the Cell processor

ELEN E6900 – Fall 2008 – Project report

Shrivathsa Bhargav, Jonathan Chen, Larry Chen, Thomas John, Jaime Peretzman
Columbia University
Fall 2008

Table of Contents

Abstract 2
1. Introduction 2
2. H.264 2
3. x264 3
4. Design 3
4.1 Parallelization strategies considered 4
4.2 The parallel architecture for x264 5
5. Implementation 5
5.1. Hypervize framework 5
5.1.1. Is_free 6
5.1.2. Hypervize 6
5.1.3. wait_completion 6
5.2. Motion estimation code on the SPE 6
5.2.1 x264_me_search_ref 7
6. Benchmarks 7
7. Conclusion 8
8. Future work 8
References 9
Acknowledgment 9
About the authors 9

Optimizing H.264/AVC on the Cell processor

[ELEN 6900 Project Report – December 2008]

Shrivathsa Bhargav, Jonathan Chen, Larry Chen, Thomas John, Jaime Peretzman
{sb2784, jtc2119, lc2454, tj2183, jp2642}@columbia.edu

Abstract

The goal of this project was to port an existing implementation of the H.264 video encoding algorithm onto the IBM Cell processor, and optimize a selected set of its components utilizing the parallelism enabled by the eight SPEs, as well as exploiting other advantages offered by the Cell architecture. These goals have been met, and a framework was developed to address the needs presented by the extensive communication between the PPE and SPEs. Implementation details and results are presented.

1. Introduction

MPEG-4 Part 10 (also known as H.264 or Advanced Video Coding) encoding is a relative newcomer to the video codec arena, but it is quickly gaining popularity due to its high efficiency; it is capable of providing the same video quality as its predecessors but at half the bitrate (1). Its popularity has led to its adoption in Hi-Def content delivery systems, including the Blu-ray disc format. Several commercial implementations of H.264 exist, but this project uses an open-source implementation, x264.

x264 is a widely used (2), open-source (3) implementation of the H.264 algorithm. Using an open-source implementation allowed us to have unfettered access to a working implementation, while providing some comfort that, since x264 is a fairly mature project, the original code's results can be used as a golden standard with which to compare the results of the optimized code. Since an end-to-end encoder was provided, the specific components to be tweaked could be cherry-picked, and the performance impact on the overall process could then be monitored.

The Cell Blade QS21 houses a pair (4) of very-capable Cell Broadband Engine multi-core microprocessors running at 3.2 GHz, each consisting of one PPE and eight SPEs. The Blade also contains 2GB of shared memory, and runs Fedora Core 7 as the OS.

This report is structured as follows: a brief description of the relevant components of H.264 can be found in section two, the structure of x264 is discussed in section three, design constraints are mentioned in section four, while section five details the implementation choices made, as well as the framework that was constructed. Benchmarks are found in section six, while section seven concludes the report.

2. H.264

A successor to both H.263 as well as MPEG-4, the H.264 algorithm was developed jointly by the Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG) with focus on network streaming across the whole spectrum of bandwidth available today, from low-bandwidth video-conferencing to high-bandwidth high-definition movie streaming.

Since H.264 hails from a highly-engineered lineage of codecs, there is tremendous complexity involved in the encoding process. This complexity was anticipated by the technology committee, and led to the creation of "profiles". Profiles are used to categorize the features needed to decode the encoded video. There are three main profiles: Baseline, Main, and High. Baseline implements a reduced subset of H.264's features (5), and is shown in Figure 1.

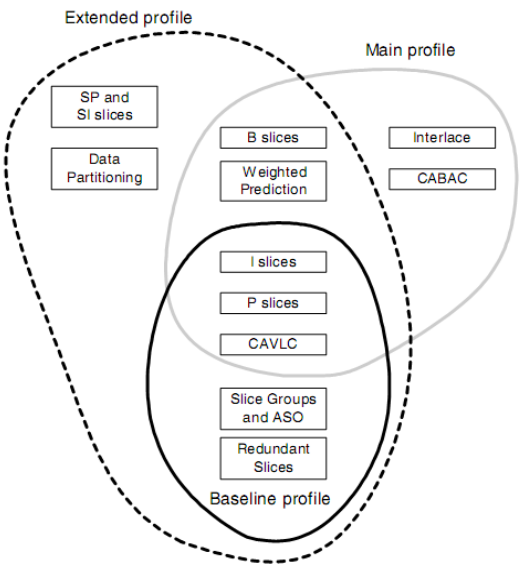

Figure 1 - Profile feature subsets

The Baseline profile was alluring because of its simplicity; Context-based Adaptive Binary Arithmetic Coding (CABAC) is foregone (as are Bi-directional slices, or B-slices), and the simpler Context Adaptive Variable Length Coding (CAVLC) is used instead.

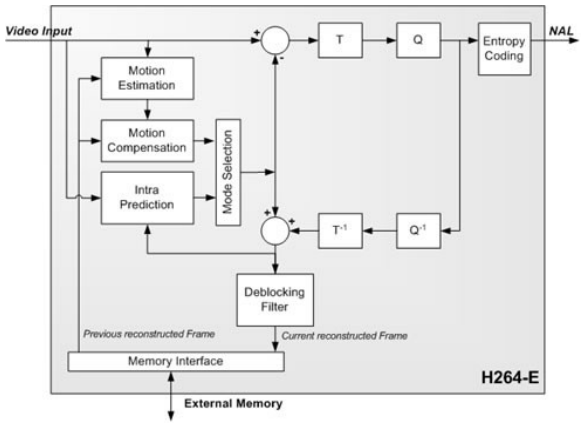

Figure 2 - Basic H.264 process flow

The basic flow of the encoding process is shown in Figure 2 - Basic H.264 process flow. A full explanation of the H.264 encoding process is beyond the scope of this report. For this discussion, it suffices to know that every modern video encoding algorithm uses motion estimation (6) and compensation (usually just called motion compensation). This space-saving technique involves predicting the directions and velocities (through vectors) of objects that move between frames. Only objects that move need to be encoded between frames. Hunting for these moving objects, however, is a process-intensive task, and motion estimation uses around 50% of all processing time during the encoding of a frame (7).

3. x264

x264, at the time of this report, is the most popular open-source H.264 encoder implementation available. It is very versatile, supporting nearly the entirety of the H.264 specification, and serves as both an encoder and a decoder. Only a small subset of what x264 has to offer, however, is being used and optimized.

Some of the parameters that can be changed include macroblock partitioning, number of passes, number of B frames, spacing between I and P frames, etc. Since there are many confusing parameters that can be specified to x264, a GUI was used to create the list of command-line arguments (8).

4. Design

Several design constraints were set up. To scale down the complexity of the project, only the Baseline profile was used. Each profile specifies a set of algorithms and features for the encoder; a matching profile must be supported by the decoder. By limiting ourselves to the Baseline profile, some quality is being sacrificed for performance and simplicity.

Since there is no way of specifying a profile to use when compiling the x264 program, each feature unrelated to our chosen profile had to be manually disabled. B slices and CABAC were disabled using command line arguments. To ensure x264's compliance, the profile selection was hardcoded in the source code to adhere to Baseline. I slices and P slices are used, and the burden of entropy coding is shifted to CAVLC, the less efficient technique compared to CABAC.

Constraining our input and output videos enabled consistent profiling; the resolution was fixed to 640 x 480 (VGA), and the bitrate was clamped to 500 kbps, which is sufficient to fit a medium-quality encoding of a three hour feature-length film onto a single, 700 megabyte CD-R. Significant time was invested in profiling the code with the constraints mentioned above. The GNU software gprof was used, which gave us an analysis of the number of processor "ticks" needed by each function. Turning on "tree-mode" gave us an even better idea of the functions called by a given node and the functions that called the given node.

Our first attempt at finding the right level of granularity to parallelize the code required us to find the leaf functions indicated by *gprof* and then offload them onto each SPE. It was soon realized this parallelization technique imposes a significant limit on the ability to scale with architecture changes in the future. Furthermore, at this level of granularity, the amount of data being transferred between the PPE and the SPE is so insignificant that it does not warrant the overhead of DMA transfers.

Sensing the drawback of this approach, a better way to explore the code tree was sought. Understand C 2.0 is a commercial (9) code analysis tool that automatically analyzes code trees and draws butterfly graphs showing child and parent functions at each function-level node. In addition, it provides an easy-to-navigate interface that allows us to jump between functions in the source code with ease.

With the aid of the code analysis tool, a significant amount of time was spent trying to decipher the x264 code. It was realized that the point of entry had to go a few levels up in the code hierarchy from the leaf nodes. The ideal candidate for parallelization would be a function where there is a meaningful amount of data to be offloaded, where interdependencies between functions are kept at a minimum, and where the data is not so large that it cannot be fit in the local store of the SPE. The parallelization strategies considered and the reasoning behind choosing our approach has been described in the next section.

4.1 Parallelization strategies considered

The general objective is to increase the granularity of parallelization: the larger the grain size, the lesser interdependencies between the different granules.

At the highest level of task division, the entire raw video stream could be broken down into multiple sections and encoded in parallel. However, this approach by design precludes the possibility of employing the encoder for encoding video in real time. Real video streams contain frames in time order. Further, such a strategy necessitates the introduction of I frames at the beginning of every new section of the original video stream. This reduces the effectiveness of motion compensation.

Another possibility considered was that of the PPU distributing entire frames to SPUs for processing. The SPUs would then co-ordinate amongst themselves to resolve dependencies between frames and their reference frames. While this is an effective strategy, it mandates porting the entire encoder code to the SPU. This was deemed beyond the scope of a single semester project. Adding to the complexity would be the need for SPU overlays, as the SPU code size would be too large to fit on the Local Store.

The above technique could be modified slightly by assigning specific functions such as Motion Estimation, Transform and Quantization, and CABAC to individual SPUs as is done in (10) and by Toshiba. Pointers to individual frames would be passed between SPUs, depending on each SPUs availability and functionality. This minimizes the code on each SPU as compared to the previous technique. Further, the entire processing is pipelined to take advantage of the parallelism between frames. Inherent in this design is the fact that each functional unit viz. Motion

Estimation, Transformation and Quantization, and CABAC operates individually on a single frame before passing it onto the next functional unit. However, the x264 architecture breaks processing down at the macroblock level and not at the frame level. For example, Motion Estimation is done on a single macroblock followed by Transform and Quantization and CABAC, after which the next macroblock is processed. This architecture does not lend itself well to pipelining between functional units as there is a strong dependency between a macroblock and previously encoded macroblocks.

An alternative approach to parallelization is to break down and port a single section of the code to the SPUs and maintain the rest of the code on the PPU as in the original implementation. This was seen as the most viable option for a single semester project. Further, to take advantage of frame level parallelism, the threading architecture in x264 was employed to encode multiple frames in parallel, with each thread spawning tasks onto SPUs.

4.2 The parallel architecture for x264

The x264 architecture was modified to make use of the multicore Cell Broadband Engine. As reasoned in the previous section, it was decided that a single unit of the encoder should be ported onto the SPUs. Further, the threading architecture in the x264 implementation was employed to exploit parallelism between threads.

The main thread in the PPU loads each frame from disk and preprocesses it. Following this, it spawns a new thread – limited by the number of threads specified by the user – to process the frame. Each thread then processes the frame macroblock by macroblock, running through the different steps of Motion Estimation and Compensation, Transformation and Quantization, Entropy Coding and Deblocking Filter. Further, if a thread requires data from another thread processing a previous frame, it uses the pthread broadcast/wait mechanism to resolve this.

Amongst the different functional blocks, Motion Estimation was identified as a potential target for porting to the SPUs. Motion Estimation takes up to 50% of the encoding time. Further, within Motion Estimation, the computation of the Sum of Absolute Differences is time intensive and amenable to parallelization.

Thus, during the processing of every macroblock, each thread identifies a free SPU and assigns the task of performing the Motion Estimation search for the macroblock to the SPU.

The following sections examine the implementation of the above described architecture. Section 5.1 discusses Hypervize, a framework for spawning tasks to free SPUs. Section 5.2 discusses the porting of the Motion Estimation code from the PPU to the SPU.

5. Implementation

5.1. Hypervize framework

The Hypervize framework (not to be confused with the hypervisor in virtualization technology) handles task distribution to the SPEs, and masks the complexities involved with handling all the SPEs. This custom framework supplies an API:

```
int hypervize(void **structure_pointer, int
num_spes, int behavior), returns int
handle.

int is_free_blocking(int handle), returns
int status.

int is_free_nonblocking(int handle),
returns int status.

void SPE_wait_completion(int handle).
```

One of the parameters into the Hypervize function is a void pointer to an array of structures. This parameter structure contains a pointer to the full frame structure (which includes frame data as well as other essential function pointers). It is the responsibility of the SPE to transfer the required data through DMA from these structures. Two variants of the Hypervize are provided: blocking and nonblocking. The API will be briefly explained below.

5.1.1. Is_free

This function is only called from within Hypervize(), and is tasked with finding the number of SPEs provided in the function call. It does this in a round-robin fashion as shown in Figure 3.

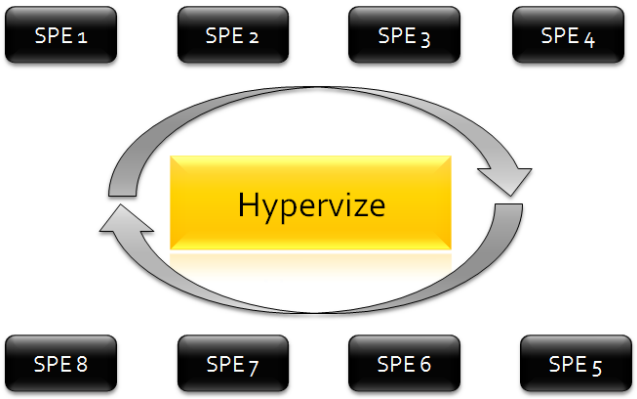

Figure 3 - Is_free() round-robin SPE status polling

An SPE is marked "idle" if it writes a known value into its interrupt outbox. Since the framework needs to operate in blocking and nonblocking modes, the SPEs could be polled in two ways. The first would continue polling the SPEs until the required number are free. This would be a blocking call. The second way would only poll the SPEs once, and if the required number of SPEs are not free, Hypervize() should return with an error status. This affords us the ability to fallback on the PPE to run the operations while waiting for the SPEs to become available.

5.1.2. Hypervize

The Hypervize() *function* (not to be confused with the framework) allows the user to specify the number of SPEs to be used and accepts as its parameters a pointer to a list of structures for the calling function.

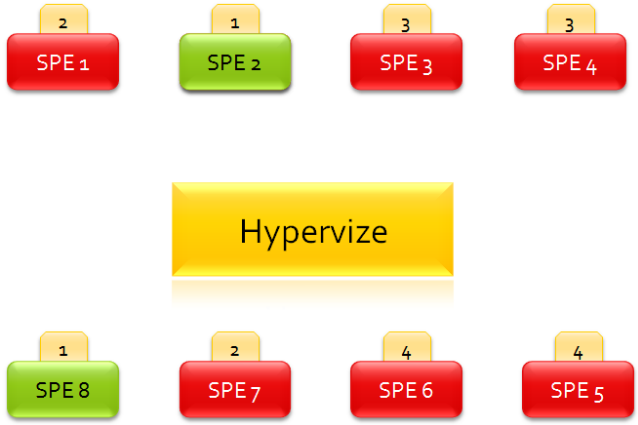

Figure 4 - Hypervize() assigning task handles

The function Hypervize() uses is_free() to poll all the SPEs in the system to detect if they are idle or busy. Once is_free() returns an array of available contexts, Hypervize() sends the address of the parameter structure to these SPEs, and marks the SPEs as busy in its internal bookkeeping table. In addition, each task is assigned a task "handle", which Hypervize() then returns to the calling function. Note that the blocking and nonblocking properties of the Hypervize() function only apply to the *assignment* of the task to the SPE, and neither version waits for the SPEs to complete the assigned task. That job is performed by wait_completion().

5.1.3. wait_completion

Once the calling function requires the data processed by a task handle (that it has received from Hypervize()), it can call wait_completion() with the handle. This function will then poll the SPEs associated with the task handle until they turn idle (having completed the task). This behavior means that wait_completion() is a blocking call.

5.2. Motion estimation code on the SPE

After studying the code tree and profiling the code, the group of functions making up motion estimation was discovered that use approximately 50%-65% of

the total processing time. On closer inspection, the x264_search_ref() function, which looks for reference macroblocks to compare to that of the currently encoded frame, uses up 50% of the motion estimation processing time, and therefore 25%-32.5% of the total execution time.

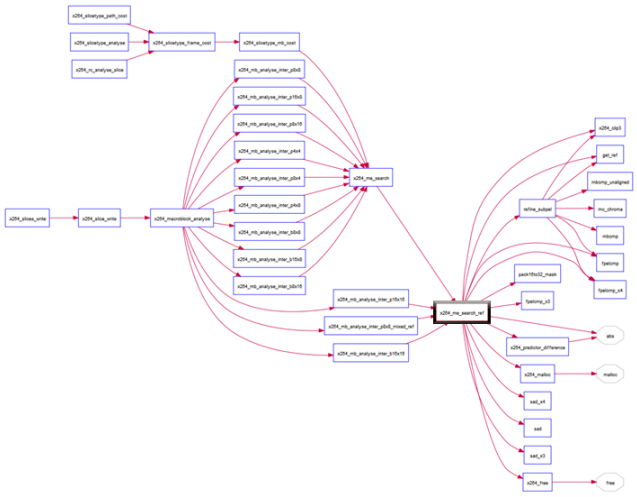

Figure 5 - Partial code tree with x264_me_search_ref() highlighted

Seeing as how this was a function with little functional dependency via Understand 2.0 (as barely seen in Figure 5), it was appropriate for us to study and evaluate x264_me_search_ref() in more depth as a decision to port this function to the Cell Broadband Processor had to be made.

5.2.1 x264_me_search_ref

The reason for x264_me_search_ref() being such a computational intensive function is the fact that it needs to find the best motion vector between the currently encoded macroblock and a series of reference macroblocks in a specified search area.

To give a better understanding of the task that me_search_ref() performs, the main operations of the function are highlighted. First, the function searches throughout the reference macroblock in the integer-pixel level; once the best integer-level macroblock is found, half pixel and quarter pixel searches are done. These search processes require the generation of the half and quarter pixels as well as the continuous computation of a difference metric between the macroblocks being analyzed, which are the sum of absolute differences (SAD) (11) and the sum of absolute transformed differences (SATD) (12).

Given these computational intensive operations and the fact that the block only depends in a specified area of the reference frames, porting x264_me_search_ref() to the SPEs would constitute a reasonable approach.

In the porting process, the x264_me_search_ref() function, its sub-functions, as well as the variables used in the functions, were replicated from the PPE's code into the SPE's code.

The data processed by the SPU was swapped in and out of the local store. This was necessary, as the local store is not big enough to accommodate all data used by x264_me_search_ref().

The actual porting process involved studying the memory footprint of this function in detail. It was necessary to mark which function x264_me_search_ref() calls and port those functions accordingly. While this sounds like a straightforward task, the complexity of this process grows very quickly when macro function declarations and variable functions pointers are considered. It was necessary for us to consult Understand 2.0 regularly to maintain a high-level understanding of the code tree to accomplish this task. Furthermore, the porting process further warrants various DMA transfers to request the data necessary for processing.

6. Benchmarks

While the initial benchmarks showed very favorable SPE performance with respect to the PPE, it was only recently discovered that PPE performance varies wildly when heavily loaded. A heavily loaded PPE performs about the same as running the code on the SPEs, but a lightly loaded PPE always significantly

outperforms the SPEs. Since this loading is unpredictable, the initial benchmarks obtained (shown in Table 1) must be taken with a grain of salt.

Table 1 - Initial performance benchmarks

FPS	Number of SPEs used					
Threads	1	2	4	8	PPE loaded	PPE no load
1	1.98	No use	No use	No use	11.73	14.87
2	2.94	2.15	No use	No use	16.83	23.75
4	3.05	3.7	4.74	No use	19.06	25.2
8	3.09	4.6	4.91	5.42	20.14	25.31
16	3.15	4.62	5.07	5.44	20.39	25.21
32	3.19	5.07	5.69	6.22	21.1	25.62
64	4.26	6.94	7.53	7.92	21.79	24.72
128	13.36	19.39	20.54	20.5	22.73	27.2

This data is shown as a surface graph in Figure 6. As evidenced, a loaded PPE performs just as good as running on all SPEs at the same time.

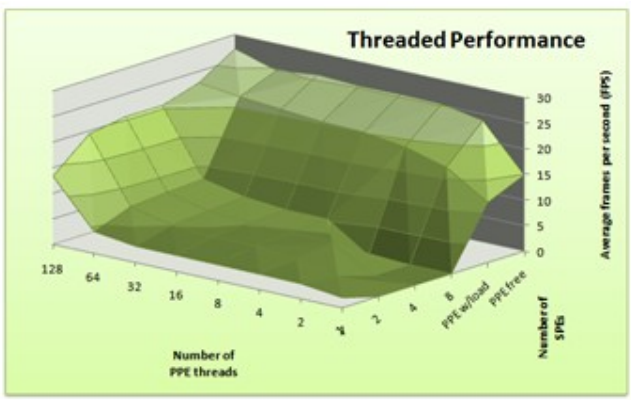

Figure 6 - Surface graph of initial benchmarks

A more reliable benchmark compares the execution time of x264_me_search_ref() on the PPE (under loaded and free conditions) and on the SPE, as shown in Table 2.

Table 2 - Execution time for x264_me_search_ref()

Execution time (ticks)		
PPE loaded	PPE no load	SPE
20451	1507	8587

This disparity in performance can be attributed to the DMA overhead overshadowing the gains obtained in computation speed by porting the code to the SPU. However, this is to be expected given that the size of the code section ported to the SPU was kept low, given the time constraints. Porting more tasks to the SPU would compensate for the DMA overhead, resulting in a consistent speedup.

7. Conclusion

x264, the open-source implementation of the H.264 encoding algorithm, has been ported onto the Cell BE. Each member of the team contributed significantly to the porting process and it was a valuable learning experience.

The GNU profiler, *gprof*, was used to profile the code and to determine the processing time required and the frequency with which each function is called by each parent function. Understand C 2.0 (a commercial code analysis tool) helped analyze the code hierarchy and resolve the interdependencies between them.

When the project first began the plan was to only optimize the leaf nodes, but this contradicted with scalability. A good balance between a meaningful amount of data to work with and a granularity that would allow data to fit in the local store of an SPE was needed. For reasons explained in the Design section, the main component that was optimized was motion estimation. The result, due to DMA overhead, was a marginal speedup when all 8 SPEs were running in a multithreaded architecture. This speedup is relative to the PPE, and is heavily depended on current PPE load, which is nondeterministic. While the current version of the code has minimal speedup, it is a good platform to base further porting of code from the PPU to the SPU. Porting more functions to the SPU would compensate for the DMA overhead, resulting in a more consistent speedup.

8. Future work

Some future directions that warrant exploration include:

- Porting over more functions to the SPEs with the use of SPE overlays/contexts. If contexts are used, Hypervize can be upgraded to keep track of existing contexts on SPEs to prevent redundancy in context switches.
- Creating a pseudo-shared memory model among the SPEs. If one SPE needs a certain piece of data, it can first check with the other SPEs to see if the data already exists in the local store of a sibling SPE (from where it can be transferred using DMA) before it attempts to retrieve the data from the main memory through the PPE (a costly DMA transfer).
- Use the OpenMP framework to automatically parallelize less computationally-intensive tasks to the SPEs. While this will not be as efficient as manually porting critical sections of code over, an overall speedup may still be observed because the SPEs can be faster than the PPE, depending on PPE loading.

References

1. **Apple Inc.** Apple's FAQ on H.264. *Apple Web Site.* [Online] http://www.apple.com/quicktime/technologies/h264/faq.html.
2. **Videolan project.** List of projects that use x264. *Videolan project web site.* [Online] http://www.videolan.org/developers/x264.html.
3. **x264 project web site.** x264 home page. *x264 project web site.* [Online] http://x264.nl/.
4. **IBM.** QS 21 Cell Blade. *IBM web site.* [Online] http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs21/index.html.
5. **Richardson, Iain E. G.** *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia.* Chichester : John Wiley & Sons Ltd, 2003.
6. *Implementation of a basic H.264/AVC Decoder.* **Fiedler, Martin.** Chemnitz : s.n., 2004. http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/sa-mfie.pdf.
7. *Parallelization of high-performance video encoding on a single-chip multiprocessor.* **Wu, Di, et al.** Dubai : s.n., 2007. International Conference on Signal Processing and Communications. http://www.da.isy.liu.se/pubs/diwu/diwu-icspc2007.pdf.
8. **Leiming.** Leiming's x264 GUI. *Doom9 forum.* [Online] October 4, 2008. http://forum.doom9.org/showthread.php?t=112770.
9. **Scientific Toolworks.** Understand 2.0. *SciTools* . [Online] http://scitools.com/products/understand/.
10. **Jagmohan, Ashish, et al.** *H.264 Video Encoding Algorithm on Cell Broadband Engine.* s.l. : Department of Multimedia Technologies, IBM TJ Watson Research.
11. **Wikipedia.** Sum of absolute differences. *Wikipedia.* [Online] [Cited: December 19, 2008.] http://en.wikipedia.org/wiki/Sum_of_absolute_differences.
12. **Wikipedia**. Sum of absolute transformed differences. *Wikipedia.* [Online] [Cited: December 19, 2008.] http://en.wikipedia.org/wiki/Sum_of_absolute_transformed_differences.

Acknowledgment

The authors would like to thank adjunct faculty Prof. Lurng-Kuo Liu for his suggestion of this topic, and his support during the project, and for ingraining the ideal of "calculated risk-taking" while handling projects of this complexity. In addition, the authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of their QS20 Cell Blade that have contributed to this project, as well as the Cell Virtual Loaner program for access to the QS21 Cell Blade.

About the authors

Shrivathsa Bhargav graduated from the SUNY, Stony Brook with a Bachelor's Degree in Electrical Engineering (Cum Laude). He is interested in digital VLSI circuit design and design automation.

Jonathan Chen graduated from the University of British Columbia, Canada with a Bachelor's of Applied Science in Computer Engineering. His professional interests are digital circuits and embedded systems.

Larry Chen graduated from the University of Waterloo, Canada with a Bachelor's Degree in Computer Engineering. His interests are embedded systems and circuits.

Thomas John graduated from the College of Engineering Guindy, Chennai in India with a Bachelor's Degree in Electrical and Electronics Engineering. His interests are embedded control systems and computer architecture.

Jaime Peretzman graduated from the Monterrey Institute of Technology with a Bachelor's Degree in Mechatronics Engineering. His interests are embedded systems and robotic design.