Accelerating H.264 Advanced Video Coding with GPU/CUDA Technology

Christopher Rivere Escobedo
Donald Bren School of ICS
Honors Thesis
6/3/2011

Abstract

With the rise of streaming media on the Internet and the YouTube revolution, the demand for online videos is costing companies a significant amount of bandwidth. To alleviate the resources needed for streaming media, video compression removes redundant information and minimizes the loss in quality experienced by a human audience. In response to the need of better compression for high definition video across many environments like the Internet, the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) worked together to create a new modern standard called H.264.

H.264 Advanced Video Coding is the current standard in video compression. H.264 is a powerful yet flexible approach to video encoding across many platforms and applications. H.264 encoders take the digital frames of a video and reduce the file size by applying predictions, transforms, quantization, and bitstream encoding. Inter and intra-prediction are the two prediction modes that allow H.264 to achieve high performance in compression by reducing redundancies across temporal and spatial domains. The advanced techniques used in H.264 will provide a strong foundation for future video encoder standards.

The author investigated speeding up motion estimation in a H.264 software implementation using multicore graphics processor (GPU) and optimizing the GPU code for different types of memory. The overhead of memory transfers to the GPU proved to be slower than the original CPU-based version.

Table of Contents

1. Introduction

With the rise of streaming media on the Internet and the YouTube revolution, many users are watching videos every day. This new demand for videos is costing companies a significant amount of bandwidth and energy to transfer HD 1080P quality video from servers to viewers. To compensate for the large demand of streaming media, companies like Google and Netflix are investing large amounts of resources into server farms. In terms of energy, the amount of electricity used by data centers in the world in 2005 was 1% of the world's electricity consumption, with the U.S. accounting for 37% of this percentage [1]. Video compression is used to reduce file sizes with variable loss in clarity and video quality to help reduce the energy requirements for streaming media. High performance compression algorithms reduce the energy impact that each individual user makes when they view videos online. However, advanced compression algorithms can require significantly more computations, especially for large HD movies. After an introduction to video compression and H.264, this paper presents research done by the author to port and optimize some of the most time consuming portions of the video encoding algorithm on an NVIDIA GPU.

1.1 Purpose Of Compression

The goal of compression algorithms is to remove redundant information and minimize the loss in quality experienced by a human audience. Compression algorithms can be applied to audio, video, and image files. There are two versions of compression, lossy and lossless. Lossy compression permanently loses some information during the encoding process at the benefit of achieving high compression performance compared to lossless [2]. Losing information is not detrimental since lossy compression can focus on only losing details that the humans cannot detect. Thus both lossy and lossless compression types can have the same amount of visual quality but they will differ greatly in file sizes and computational complexity.

2 What is H.264

H.264 Advanced Video Coding is the current standard in video compression [3]. H.264 codecs consist of an H.264 encoder and decoder pair. An encoder takes a video as input and compresses it using the steps outlined in a specific video encoding algorithm. The decoder then performs the opposite operations of the encoder to uncompress the video for viewing on an output device such as a monitor or television [3]. H.264 encoders take the digital frames of a video and reduce the file size by applying predictions, transforms, quantization, and bitstream encoding [3].

2.1 The Creation of H.264

In response to the need of better compression for high definition video across many environments like the Internet , the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) worked together to create a new modern standard. These two groups together formed the Joint Video Team (JVT) in 2001 in order to work together to develop H.264. The International Telecommunication Union (ITU) published the first final draft of the paper, "Recommendation for H.264," which was approved in May of 2003 [4]. These two groups have previously developed video coding standards. The Recommendation paper only defines the decoder portion of H.264. More specifically, the JVT define the syntax of an encoded H.264 bitstream and how the decoder should read this bitstream [4]. The JVT do not define an encoder and therefore provide flexibility to implementers of a H.264-compliant device. Although not including an encoder definition may seem detrimental for establishing consistent H.264 encoder implementations, the JVT gives H.264 implementers flexibility by providing many profiles and levels that allow H.264 devices to vary in H.264 compliance. [4]. For example, there is no point in integrating the advanced video compression features of H.264 on a basic cell phone device. In a device with limited computational power and memory space, these advanced features would not make sense. This is one of the many features that make H.264 the current standard in video encoding today.

3. Overview of the H.264 Video Encoding Process

[2] provides a summary of the H.264 video encoding process. First, a video source is provided as input to the H.264 encoder. H.264 processes video frames by first dividing them into units called macroblocks. A macroblock is a 16x16 array of pixels. The encoder does not encode the pixels of macroblocks themselves, but instead forms predictions of macroblocks and encodes residual macroblocks. Next, the samples of the residual macroblocks are transformed using an integer transform, which approximates the discrete cosine transform [2]. The transform outputs coefficients, which are then quantized. Quantization is the process of dividing the transformed coefficients by an integer [2]. Note however that H.264 is a lossy compression algorithm since after quantization some of the lower order bits of the residual samples are removed [2]. Lastly, the coefficients are passed to an entropy encoder that reduces redundancies of repeated coefficients by applying statistical methods [2]. The entropy encoder outputs a bitstream that is ready for decoding by an H.264 decoder, which will apply approximately the same process as the encoder but in reverse.

3.1 How H.264 Macroblock Prediction works

H.264 is a block-based motion-compensated video encoding standard that achieves high compression results by forming accurate predictions of macroblocks from either previously coded frames or previously coded samples in the current frame [2]. To determine the best prediction of the current macroblock, H.264 uses two different methods each with their own benefits and costs. An accurate prediction will yield a small residual and thus will require fewer bits to transmit [2].

3.1.1 Inter-prediction in H.264

The first method is called inter-prediction or motion-compensated prediction, and it achieves the highest compression performance of the two methods by reducing the temporal redundancies across different video frames [2]. Inter-prediction uses motion estimation and motion compensation to create accurate predictions of macroblocks. Motion estimation is the process of searching for the best match of the current macroblock in a previously coded frame [2]. After finding the best match, motion compensation is the process of subtracting the best match from the current macroblock to form a residual [2]. Rather than directly encode the actual pixels of a frame in a video, video compression achieves high compression ratios by only encoding the differences between the current frame and a previously encoded one. The reasoning behind this approach is that if you can accurately predict the current frame from a previous frame, then the amount of energy that remains when you calculate their difference will be quite low which results in fewer bits being required to predict the current macroblock [2]. H.264 improves on previous standards by allowing more options for effective motion estimation, which further reduces the entropy of residuals and lowers the overall bit rate of a video.

As an example of inter-prediction, imagine a video of a solid green ball moving across a black background. The region that is changing is the location of the ball. Assuming it is moving slowly, motion estimation can accurately predict the location of the ball in the next frame by providing a vector that points to the location of this ball from the previous frame [2]. In both frames, the tennis ball contains the exact same pixel values. The only difference is that the pixels in the later frame are shifted in the direction of the ball. Motion compensation is accomplished when the motion vector is provided as part of the prediction. In this example, the residual would be zero since the tennis ball is the same in both frames. The H.264 encoder would transmit both the residual block and motion vector to be transformed, quantized, and entropy encoded [2].

3.1.2 Intra-prediction in H.264

Intra-prediction focuses on reducing spatial redundancies within a single frame. Whereas inter-prediction makes use of motion estimation to search for prediction candidates, intra-prediction uses spatial extrapolation to form predictions of a macroblock. Intra prediction uses previously coded neighboring pixels surrounding the current macroblock to extrapolate a prediction [2]. H.264 allows many different extrapolation modes to help minimize the cost of this prediction. However, intra-prediction makes less accurate predictions when compared to inter-prediction and therefore is usually only used to establish the first reference frame in a set of frames to be encoded [2].

An example of intra-prediction is a blank wall in a room with gradually changing lighting. The lighting will gradually change the color of the wall from dark to light. This forms gradients that are ideal for spatial extrapolation since the pixels will be highly correlated with each other [2]. As the light changes, the pixels along the wall will change in a specific order that can be accurately predicted by spatial prediction.

3.2 Why H.264?

H.264 is the current standard in video encoding because of its ability to reduce file sizes from up to 50% compared to the previous standards [2]. The JVT intended H.264 to be a powerful yet flexible approach to video encoding across many platforms. H.264 contains a set of profiles that allow the H.264 implementers to take into account how much compression and computational power the videos will require. There are many profiles that fall within the full range of devices [2]. This is why many applications such as video conferencing, streaming HD broadcasts, Blu Ray, and cell phones can all use H.264 for video compression.

4. Personal Undergraduate Research on H.264.

The purpose of this undergraduate research project is to speed up the H.264 video encoding process by using a graphics processing unit (GPU) to find the best prediction in motion estimation for inter-prediction.

4.1 Research Tools and Devices

The H.264 software modified for this project was 464.h264ref from SPEC CPU2006, which is based on version 9.3 of the H.264 AVC reference implementation created by the JVT [5]. Gprof was used to profile the H.264 video encoding of "foreman_qcif.yuv" using the main profile parameter set [5]. The H.264 binary was executed using the command, "./h264refO3 -d foreman_ref_encoder_main.cfg ." The tests were run on an Ubuntu Linux 11.04 machine with a 2.13GHz Intel® Core™2 Duo processor, 1 GB of memory, and a NVIDIA GT 220 graphics card with 48 CUDA cores. For GPU programming NVIDIA provides the CUDA Toolkit 3.2, which contains the CUDA compiler and the Visual Profiler for analyzing the performance of kernel functions that execute on the GPU.

4.2 Related Research on Optimizing H.264

Although H.264 can achieve impressive compression performance, the advanced algorithms require significant computations for large video files, which can take much longer to encode than previous video encoding standards. Research on H.264 focuses on reducing computational requirements and video encoding time by speeding up either intra-prediction or inter-prediction.

4.2.1 Intra-Prediction Optimizations

The authors of [6] discuss how H.264 uses a rate-distortion optimization (RDO) technique to determine the best encoding mode for a specific macroblock. H.264 uses a Lagrangian RDO method for intra-prediction to sequentially compare up to 592 RD computations before choosing the intra-mode that minimizes the bits required to predict the current macroblock [6]. The authors of [6] present an algorithmic solution using fast mode selection and early RDO calculation termination to reduce the number of RD computations necessary for finding the best intra-prediction mode. In addition, for encoding Predictive-frames the best prediction mode was chosen after considering both inter and intra-prediction modes [6]. Since both prediction modes are considered, any speed up achieved in the intra-prediction portion can benefit inter-prediction as well.

4.2.2 Inter-Prediction Optimizations

There is a significant amount of research on motion estimation because it accounts for most of the encoding time in H.264 [7,8]. The authors of [7] state that when running the H.264 baseline profile at level 3.1 in their experiments, motion estimation accounts for 92% of the encoding time. To speed up motion estimation, the authors of [7] implement changes to the algorithm to remove some dependencies in order to pipeline the motion estimation that occurs between successive macroblocks. In contrast, [8] performs motion estimation on a down-sampled image and use the motion vectors found in the smaller pictures to estimate the motion vector predictions on larger version of the images. This improves the cost estimates over implementations that ignore the prediction component completely [8]. Interestingly the down-sampling was performed on a GPU using NVIDIA's CUDA architecture.

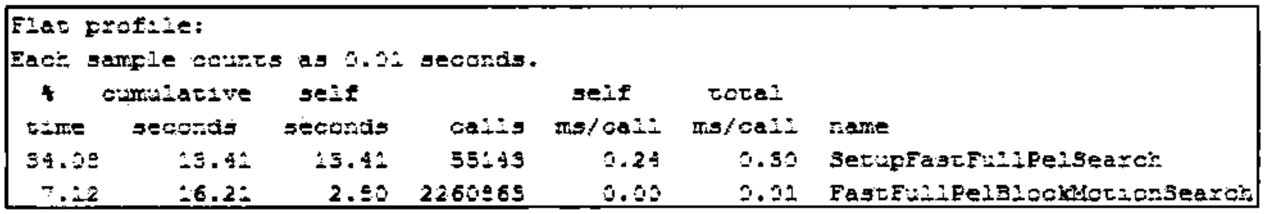
4.3 Brief Introduction to NVIDIA's CUDA Architecture

Traditionally programming on GPU's required learning a graphics programming language such as OpenGL. The languages were made for performing graphics, and were not well-suited for non-graphics purposes [9]. To make it easier to program GPUs, NVIDIA launched in 2006 a new software and hardware architecture for their graphics cards called Compute Unified Device Architecture (CUDA) [10].

CUDA has enabled a large increase in the popularity of General Purpose Computation on GPUs (GPGPU). CUDA provides a set of abstractions that allow programmers to avoid representing scientific applications as graphic applications [9]. There is a low learning curve for CUDA because the programming language is an extended version of C [9]. CUDA gives programmers access to the resources on the GPU such as shared and global memory, as well as easy to use macros for tasks such as CPU to GPU memory transfers and starting functions on the GPU, which are called kernels [9,10].

4.4 Research Methodology

To find the most called functions in the H.264 program, the parameter set called main profile was analyzed using gprof. The two most called functions were both related to motion estimation.



```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 54.08     13.41    13.41    55143     0.24     0.50  SetupFastFullPelSearch
  7.12     16.21     2.80  2260865     0.00     0.01  FastFullPelBlockMotionSearch
```

Figure 1. Flat Profile Log of "gprof h264refO3 gmonO3.out –p"

Each function contained a parallel loop that was identified as a possible candidate for speed up on the GPU. The author opted to work on the smaller of the two loops inside the function, FastFullPelBlockMotionSearch, since the function was called over two million times and appeared to have lower data transfer requirements.

4.4.1 Data Dependency Considerations

One problem with coprocessing between a GPU and a CPU is data dependencies. The GPU and CPU do not share memory spaces [10]. For a function to execute on a GPU, all of the necessary data must be transferred from the CPU to the DRAM of the GPU. Although a GPU is specialized for highly parallel computations, the data transfer overhead can cause a GPU solution to become much slower than a CPU solution [10]. Therefore it is important to consider how much data transfer is required for each kernel call. The author determined that out of four arrays of data referenced in the loop, only one array changes prior to each kernel call. The other three arrays remain constant throughout the encoding and therefore only need to be transferred to the GPU memory once. The array that changes is an array of residual costs in the form of sums of absolute differences (SAD). Before each kernel call, the array must be transferred from the CPU to the GPU.

In addition to the SAD array, five integers need to be sent to the GPU prior to each kernel call. The author had two options for transferring the five integers since the memory required for five integers is not large. The first option was to pack the five integers into an array and transfer the array in the same manner as the SAD array. The second option was to pass the integers as parameters to the kernel function. Any parameters passed as arguments to the kernel function are stored in shared memory in each thread block on the GPU. Although shared memory is much faster than global memory because it is on-chip, the overhead of copying the arguments to each thread block and the possibility of bank conflicts can reduce the performance of shared memory [10]. There will be two versions of each kernel function to compare the two options in terms of same results, the next step was to optimize the code on the GPU to perform faster than the CPU version.

4.4.3 Timing Methods

The author received timing results using the techniques in [11]. The parallel loop on the CPU was timed using the "clock_gettime" function with parameter "CLOCK_REALTIME." The CPU timer had a resolution of ten milliseconds whereas the timer on the GPU has a resolution of half a microsecond. The kernel function was timed using both the GPU and CPU timers. The CUDA Visual Profiler also provides timing of the kernel, but the results differ from the methods in [11]. It is possible that the CUDA profiler is more accurate than [11] and therefore provides a better picture in terms of kernel performance.

4.4.4 Kernel Functions with Different Memory Types

The author created three different GPU implementations of the parallel loop. The difference between the kernels is where the three constant arrays reside in memory. The first version of the GPU kernel only used global memory, which is the largest and slowest memory on the GPU [11]. The second kernel used texture memory that has the same latency as global memory but is cached and optimized for 2D spatial locality [10]. The last kernel function uses constant memory that has less latency then global memory and is cached. Reading from the constant cache can be as fast as reading from a register if groups of threads read from the same memory location. If threads read from different addresses than memory access is serialized [11]. Shared memory was only used for passing arguments to the kernel. Shared memory is used for threads to cooperate at the block level [10]. For my kernels, each thread calculates one value independently from other threads so there is no thread cooperation.

```
  Kernel for global memory
__global__ void
testKernelGlobal(int* d_block_sad, int* g_spiralX, int* g_spiralY, int* d_mvbits, int max_mvd,
       int* g_results, int lambda_factor, short pred_mv_x, short pred_mv_y, int offset_x, int offset_y )
{
   int mcost =  ;
   mcost = d_block_sad[thread] + MV_COST2 (lambda_factor,  , offset_x + g_spiralX[thread],
           offset_y + g_spiralY[thread], pred_mv_x, pred_mv_y);
       write data to global memory
    g_results[thread] = mcost;
}
```

Figure 3. Kernel using global memory only and passing integers as arguments.

4.5 Experiment Results

Overall the times showed that the GPU version of the parallel loop were about three times slower than the CPU version. It is possible that the graphics card used in these experiment did not have enough cores and processing power to match the CPU. Also the amount of floating point computation in the loop was not enough to justify the costs in memory transfers. CPU times appear to be larger because they account for additional overhead on the CPU side related to the GPU kernel call. Table 1 shows that the fastest kernel was the constant memory version that passes the integer arguments as an array. Texture was second and global memory was last. The cache in the constant and texture memory definitely helped to speed up the function compared to pure global memory.

Kernel Version	Time (µs) using GPU timer	Time (µs) using CPU timer	CPU version time (µs)
Global Memory	18.800	29.241	6.441
Global Memory2	22.302	29.249	5.701
Texture Memory	17.339	27.816	6.430
Texture Memory2	22.038	29.476	5.752
Constant Memory	16.566	27.867	6.416
Constant Memory2	22.082	29.476	5.730

Table 1. Timing results for kernels using timer functions inside the source code. Results denoted with a "2" indicate that the integer arguments were passed as arguments to the kernel, not as an array. Times are averaged over 4000 function calls.

For additional analysis of the kernel functions, GPU profiling was performed by using a program called the CUDA Visual Profiler. The CUDA profiler provides data on all kernels, as well as counters that describe specific aspects about the program execution results such as the number of global memory load requests and kernel runtime in GPU and CPU terms. Interestingly the CUDA profiler showed that in terms of GPU time, Texture Memory2 was actually the fastest kernel, followed by Constant Memory2 and Global Memory2.

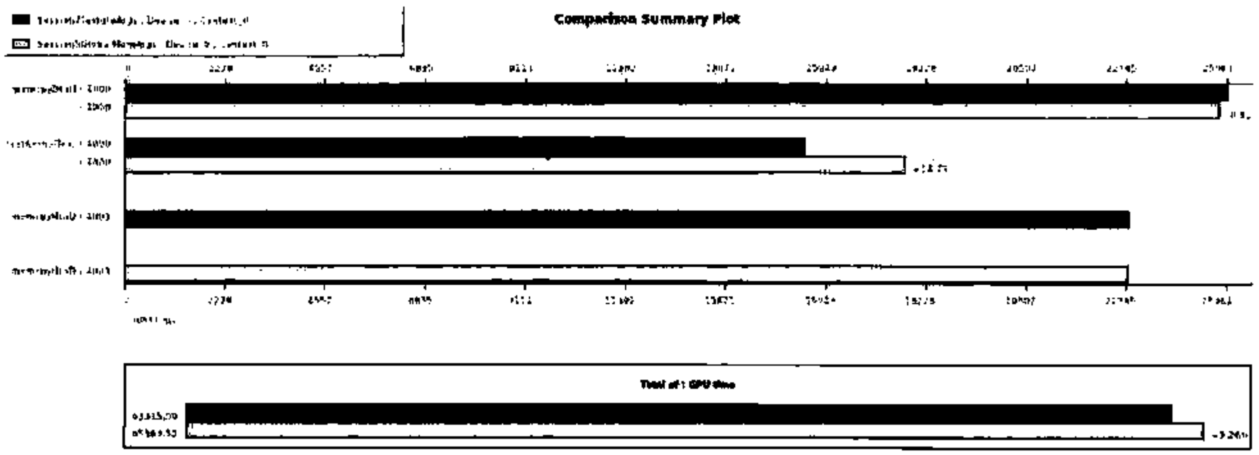


Figure 4. GPU time comparing the texture memory kernel with global memory kernel.

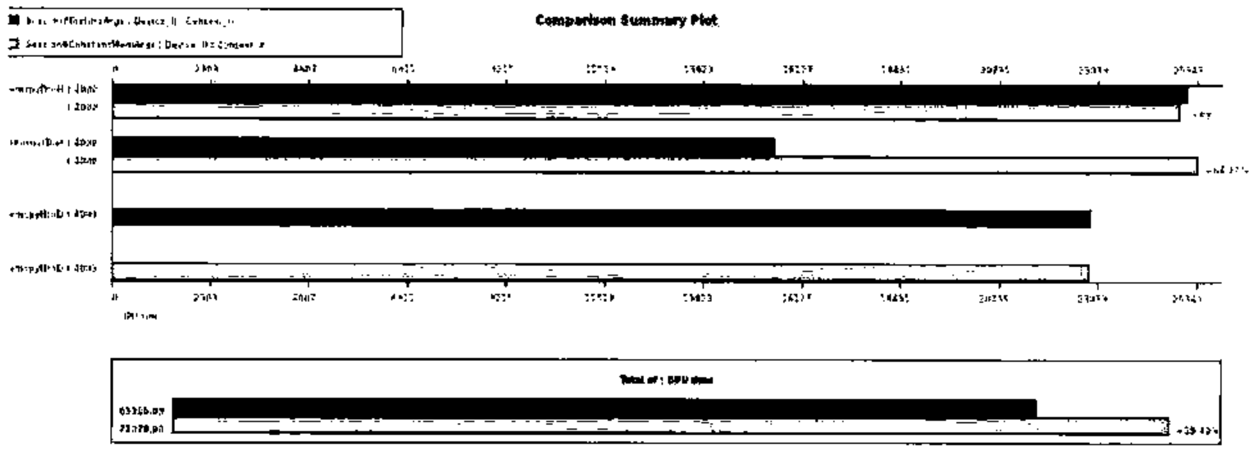


Figure 5. GPU time comparing the texture memory kernel with constant memory kernel.

In Figures 4 and 5, the second row shows texture memory in green, and either constant or global memory shown in brown. The bar length represents GPU time in microseconds. The CUDA Visual Profiler shows in the figure below that the overall global memory throughput of the constant memory is 39.07% less than the texture memory kernel. Global memory efficiency is important for GPU performance and this shows one reason why the texture memory is performing better than the constant memory.

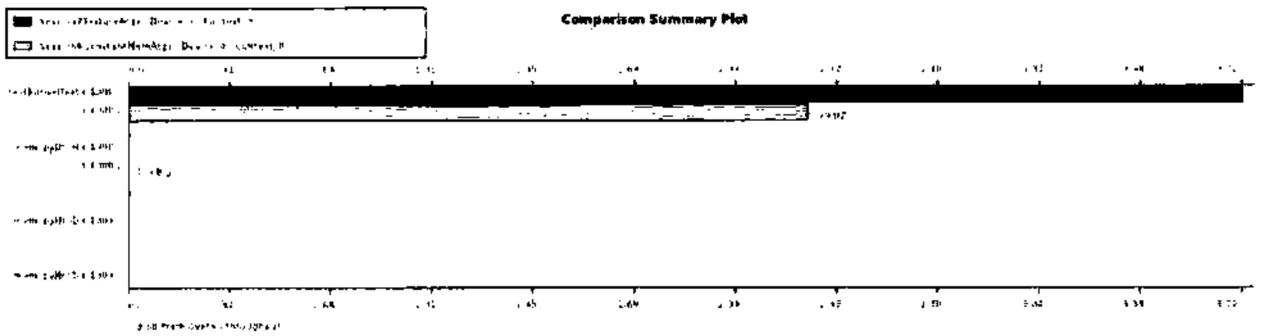
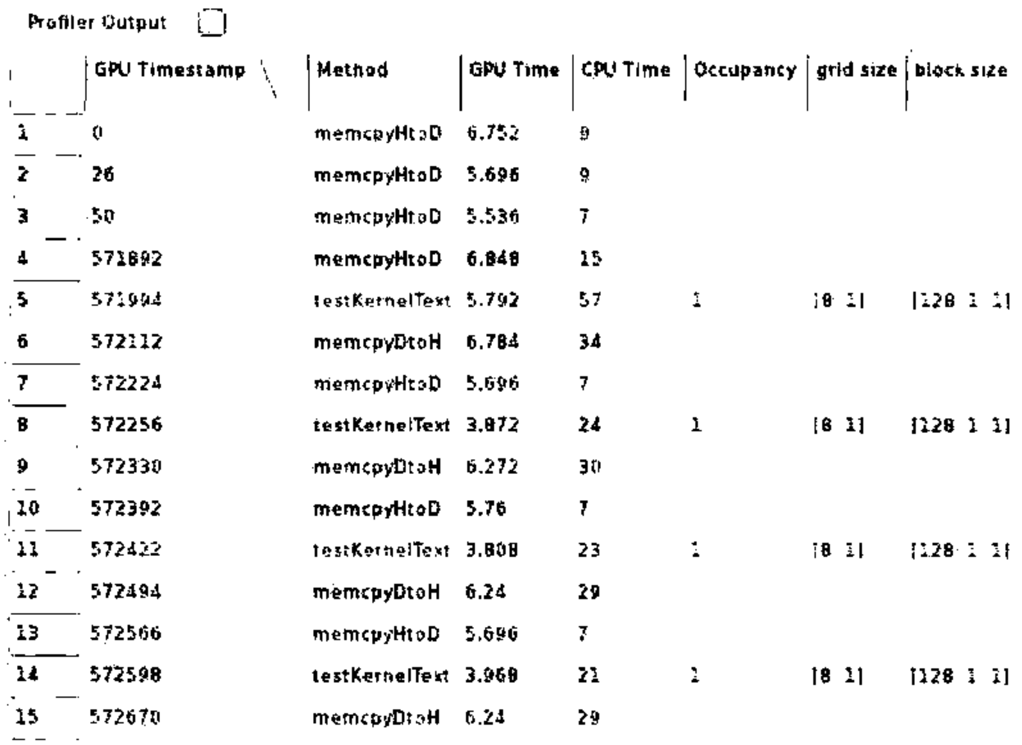


Figure 6. Overall global memory throughput of the texture and constant memory kernels.

One difference between the times reported by the methods in [11] and the CUDA profiler is the overhead of kernel calls. [11] shows more overhead for the kernel execution time when compared with the CUDA profiler. For example, in row eight of Figure 7 the time in microseconds recorded for the kernel function is 3.872. Although this time appears to be faster than any time shown in Table 1, the time required to transfer the array of SADs prior to each kernel call (i.e. row seven of Figure 7) is equal to the fastest time of the CPU version of the parallel loop. Thus any speedup from the kernel function is negated due to the memory transfer of the SAD array. Note that the first call to the kernel function is longer than subsequent kernel calls due to initialization that takes place in the GPU.



Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	Occupancy	grid size	block size
1	0	memcpyHtoD	6.752	9			
2	26	memcpyHtoD	5.696	9			
3	50	memcpyHtoD	5.536	7			
4	571892	memcpyHtoD	6.848	15			
5	571994	testKernelText	5.792	57	1	[8 1]	[128 1 1]
6	572112	memcpyDtoH	6.784	34			
7	572224	memcpyHtoD	5.696	7			
8	572256	testKernelText	3.872	24	1	[8 1]	[128 1 1]
9	572330	memcpyDtoH	6.272	30			
10	572392	memcpyHtoD	5.76	7			
11	572422	testKernelText	3.808	23	1	[8 1]	[128 1 1]
12	572494	memcpyDtoH	6.24	29			
13	572566	memcpyHtoD	5.696	7			
14	572598	testKernelText	3.968	21	1	[8 1]	[128 1 1]
15	572670	memcpyDtoH	6.24	29			

Figure 7. Profile output summary for Texture kernel with integer argument passing.

5. Conclusion

When technologies like Blu Ray become obsolete, the next standard in video encoding will improve upon the techniques established by the JVT. Although hard drives continue to grow cheaper and larger, energy usage is becoming an increasingly expensive problem that technology like video compression can help to solve. The author found that although the fastest GPU kernel was 1.48 times faster than the CPU version, the overhead of transferring one array of SADs was equal to the execution time of the CPU version of the parallel loop. A future implementation would aim to eliminate this memory transfer in order for the GPU to have a chance at becoming faster than the CPU version of the parallel loop in motion estimation.

6. Acknowledgements

The author thanks Alexander V. Veidenbaum for being his faculty advisor and for finding a project and GPU for him to perform research. The author thanks Taesu Kim for helping with

CUDA debugging and research advice through email. The author also thanks Rosario Cammarota for pointing out the locations of parallel loops and for helping the author install Ubuntu 11.04, CUDA Toolkit 3.2, and the CUDA Visual Profiler.

7. References

[1] J.G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters, vol. 3, pp.* 34008, Sept. 2008.

[2] I.E. Richardson, *The H.264 Advanced Video Compression Standard, Second Edition. Wiley*, vol..
[http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470516925.html]

[3] Iain Richardson. "White Paper: An Overview of H.264 Advanced Video Coding." Internet: http://www.vcodex.com/files/H.264_overview.pdf, 2007 [4/9/2011].

[4] (2007, Nov.) ITU-t Recommendation H.264 : Advanced video coding for generic audiovisual services. International Telecommunications Union. [Online]. Available:

[5] K. Sühring. "464.h264ref SPEC CPU2006 Benchmark Description." Internet: http://www.spec.org/cpu2006/Docs/464.h264ref.html, 2006 [May 11, 2011].

[6] M. Jafari and S. Kasaei, "Fast intra- and inter-prediction mode decision in H.264 Advanced Video Coding", Int. Journal of Computer Science and Network Security, 8 (No. 5), May 2008, pp. 1 – 6.

[7] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture, pages 37–47, New York, NY, USA, 2010. ACM.

[8] L. Chan, J.W. Lee, A. Rothberg, and P. Weaver. (2009). "Parallelizing H.264 motion estimation algorithm using CUDA." Massachusetts Institute of Technology. [Online]. Available: https://sites.google.com/site/x264cuda/Home/x264-cuda.pdf?attredirects=0&d=1 [May 16, 2011].

[9] "What is GPU Computing?" Internet: http://www.nvidia.com/object/GPU_Computing.html, 2011 [May 20, 2011].

[10] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture-Programming Guide Version 3.0, 2010.

[11] NVIDIA Corporation. NVIDIA CUDA Best Practices Guide, Version 3.0, 2010 NVIDIA Fermi Architecture. http://www.nvidia.com/object/fermi_architecture.html.