Data Handling Inefficiencies between CUDA, 3D Rendering, and System Memory

Abstract – While GPGPU programming offers faster computation of highly parallelized code, the memory bandwidth between the system and the GPU can create a bottleneck that reduces the potential gains. While CUDA, a prominent GPGPU API, can transfer data to and from system code, it can also access data used by 3D rendering APIs. In an application that relies on both GPU programming APIs to accelerate 3D modeling and an easily parallelized algorithm, the hidden inefficiencies of nVidia’s data handling with CUDA become apparent. First, CUDA uses the CPU’s store units to copy data between the graphics card and system memory instead of using a more efficient method like DMA. Second, data exchanged between the two GPU-based APIs travels through the main processor instead of staying on the GPU. These two problems result in a non-GPGPU implementation of a program that runs faster than the same program using GPGPU.

I. INTRODUCTION

GPGPU programming (General-Purpose computation on Graphics Processing Units) is becoming a popular method of accelerating parallel workloads and achieving substantial speedup. As GPGPU gains popularity, new programming APIs are written to provide a layer of abstraction to programmers to allow their code to run on any graphics card. Emerging APIs like OpenCL [1] and DirectCompute [2] allow programmers to write code for any graphics card, while existing APIs like nVidia’s CUDA [3] allow code to execute only on nVidia’s GPUs. There have already been several successful applications of GPGPU in the areas of image processing, encryption, and database applications (see, e.g., [4,5],[6]).

Interestingly, there was similar development on the first GPU acceleration APIs, 3D rendering. Early 3D rendering was handled by the CPU, but the calculations moved to highly parallelized hardware, and APIs like Direct3D and OpenGL abstract the underlying hardware. Modern nVidia consumer graphic cards support both the graphics acceleration and general-purpose computing within the same application with the possibility of sharing data between the two APIs. The benefits of this are twofold: (1) Either method can develop data for the other to use, utilizing the specialization of each API. (2) Applications that depend on both types of data processing can benefit from the specialized hardware.

In this paper, we describe an application that utilizes both, the 3D rendering specialization of graphics cards, and the massively parallel computational power of the GPU, to accelerate an alternative video compression method called Model-Based Coding (MBC) [7]. The MBC encoding process (explained in detail in Section III) requires image processing of the 2D image of a 3D model. These requirements make MBC a natural candidate for the graphics processor, since both the 3D rendering and GPGPU capabilities of the graphics card can be utilized. This paper analyzes how the application and system performance is affected by utilizing two GPGPU programming methods in the MBC encoder, and it compares and contrasts the advantages and disadvantages of each method.

II. RELATED WORK

Numerous papers discuss the advantages of GPGPU programming. 3D rendering APIs have been used for arbitrary calculations using the programmable stages of the 3D rendering pipeline. Most commonly, image processing [8-10] can benefit greatly using pixel shaders due to the specialized architecture of a GPU. However, utilizing a 3D rendering API for general-purpose computations requires adapting a specialized solution for something it was not designed to do. When nVidia added the ability to run general-purpose code on their hardware, many researchers adapted highly parallelizable algorithms to run on CUDA [4-6, 11]. However, these papers discuss only their implementation on GPU hardware and many only touch on algorithmic optimizations to maximize performance for the architecture.

Only recently have researchers begun to explore the limitations of GPGPU programming. Amorim et al. [12] look at a single problem, Jacobian iterations, and test its performance using both OpenGL pixel shaders and CUDA. They also test the performance of utilizing the different internal formats and specialized memories of each API. Kothapalli et al. [13] look at modeling CUDA kernel performance based on its instruction types, memory accesses, and execution parameters. Using their models, the authors were able to estimate kernel runtime in the programs they profiled across a range of input data sizes. Coutinho et al. [14] are developing methods of profiling CUDA programs to gauge performance bottlenecks. Using their profiler, the researchers were able to identify bottlenecks in two CUDA programs by analyzing the time spent performing different tasks using the native CUDA PTX code. Researchers are now
beginning to explore GPGPU implementations to identify the limitations and the types of applications that can truly benefit from GPGPU programming.

III. MODEL BASED VIDEO CODING

Traditional video compression algorithms such as MPEG-2/4 and H.264/AVC [15], [16] treat video frames as signals described in a statistical framework. Compression is achieved by applying a discrete cosine transform (DCT) to blocks within each frame (or differential frame), then quantizing the DCT coefficients, and then entropy encoding the quantized data. To fit within strict bandwidth constraints, such algorithms increase the amount of irreversible (lossy) compression, but this process can result in severely degraded video quality (e.g., severe blocking artifacts).

In Model Based Coding (MBC), rather than compressing frames of video, the system analyzes each frame to compute a small set of 3D model parameters. These parameters are sufficient to describe the objects-of-interest to a given level of detail. Only these parameters and not DCT coefficients are stored and/or transmitted. During decoding, a computer-rendering system uses these model parameters to create a rendition of the original scene.

MBC has the potential to radically improve video-conferencing and face-to-face video applications such as distance education. This is not only due to its extremely low bandwidth requirement, but also due to its valuable features not found in traditional video compression such as the ability to use interchangeable models, unlimited scalability with screen size (irrespective of the available communication bandwidth), and the ability to adapt to different external and user-chosen inputs such as ambient lighting and model detail. Although these benefits are known to the video-coding community, MBC is not currently popular due to the computational complexity required to fit a 3D model to a 2D video.

Specifically, transforming a captured video sequence into a model-based representation requires analysis of both spatial and temporal properties of the source video. This model-based analysis attempts to fit a 3D model to the source video by using a combination of computer vision, geometric modeling, and optimizations.

The primary bottleneck in model-based analysis stems from the optimization procedure employed during model-fitting, making MBC at least an order of magnitude slower than the DCT/motion-compensation-based analysis used in MPEG 2/4 and H.264/AVC [17]. Figure 1 depicts the steps employed in model-based analysis of facial video. The algorithm depicted uses a technique called analysis-by-synthesis, which consists of the following steps:

1. Estimate facial parameters (e.g., expression, pose) using feature analysis and previous frames
2. Synthesize the face using estimated parameters;
3. Calculate the error between the synthesized face and actual face, e.g., using peak signal-to-noise ratio (PSNR).
4. Estimate the new pose and facial expression based on the error
5. Iterate until the algorithm converges on a minimum error (best estimate).

IV. GPGPU IMPLEMENTATIONS

A. Initial Analysis

This algorithm was first implemented using the MPEG-4 Facial Animation specification for the model’s facial animation parameters (FAPs) [18]. The encoder used the GPU and Direct3D to render the model and the CPU to perform all the other steps. Initial analysis of this program showed that it could not encode 30 frames per second video in real-time. Real-time encoding is an important requirement if MBC is intended for video conferencing. Analysis of the time spent within the functional blocks of the program showed that the vast majority of the time was spent in the error calculation. Upon closer inspection, the bottleneck was identified as the transfer of pixel data from the graphics card. Figure 2 shows the initial results.

To mitigate this bottleneck, we decided to move this calculation to the graphics card, utilizing its general-purpose GPU capabilities, as shown in Figure 3. With the
CPU based method, the reference image represented by the R head resides in CPU accessible memory. The guessed image, the G head, must be copied from GPU accessible memory to CPU memory to calculate the error, e. With the GPU based MSE, the reference head is copied to GPU memory beforehand and the MSE is calculated on the GPU. The error value is then copied back to CPU memory to continue the encoding processes. This approach would solve two problems with a CPU-based approach to the error calculation. First, the calculation of the mean-square error (MSE), an intermediate value used to calculate PSNR, is a highly data-parallel calculation. The subtraction and squaring parts of the MSE equation in Figure 4 work on independent data, so this part of the calculation will benefit greatly from parallelization utilizing the high number of processors on a GPU to reduce the calculation time. Secondly, performing the MSE calculation on the GPU will keep the large amount of pixel data on the graphics card, removing the need to transfer all that data to system memory. Instead, only the result will need to be returned from the GPU.

\[
error = \sum_{i=1}^{N} (\hat{x}_i - x_i)^2
\]

Figure 4. Mean Square Error equation

B. Direct3D Pixel Shaders

At the time of this writing, there are two mature methods of performing the MSE calculation on a modern graphics card. The first approach is to utilize the programmable shader engines in 3D rendering APIs. In order to increase the realism of 3D rendering, previously fixed functions of the rendering pipeline were made programmable. Graphics card manufactures opened up the vertex and, more importantly, pixel processing engines to support user-programmable code for new rendering techniques. User-programmable code allows developers to create more realistic mathematic models for calculating the final colors of the pixels. Pixel-shader programs are designed to calculate a final value for a single pixel [19]. Early attempts at GPGPU programming exploited this feature to accelerate easily parallelized code. Input data was given to the graphics card as matrices masquerading as textures, shader programs were written to perform the necessary calculations for a single output value, and the resulting data was read back as the rendered image. This approach can be difficult to program since it requires understanding of the 3D rendering APIs to ‘trick’ the graphics card into properly processing the data.

Since our code already uses a 3D rendering API to generate one part of the input data for the error calculation, a second rendering pass can perform the error calculation. To turn the rendered image of the head into
an input for the next rendering, the output storage location of the first pass is changed to an off-screen rendering surface. This off-screen rendering surface behaves like the normal back-buffer that copies the pixel data to the monitor, but the off-screen surface can use another object, e.g., a texture object, to actually store the pixel data. This essentially creates the data in that object automatically as a result of completing that rendering pass.

A texture object is attached to the off-screen rendering surface to store the resulting pixel data, which can then be used as one of the inputs to the second rendering pass. The reference image is also brought in as the other input for the second pass using multitexturing, which allows multiple texture objects to determine the final color of a 3D object. The squaring and summation for each of the color channels of one pixel are programmed into the Direct3D pixel shaders, loading one pixel’s color from each texture object and storing the resulting MSE in the pixel’s red color channel. The pixel shader engine will automatically execute the pixel shader program for every pixel that is generated in the output of the second rendering pass. However, the summation inside each pixel shader program can only add the three color channels of that pixel. To sum the values of every pixel, the pixel shader result for each pixel is mapped onto its own unique 3D surface with each surface placed one behind another from the perspective of the screen. Alpha blending, which performs a weighted summation of the 3D surface visible in one pixel and the 3D surfaces hidden behind the front-most surface to calculate the pixel’s final color, takes the values from each surface and performs the final summation. Since the program only requires one value back from this calculation, the render target is only one pixel, which is read back as a 32-bit float and used to finish the PSNR computation.

C. CUDA Implementation

The CUDA implementation of the MSE calculation is performed in the same way as the shader implementation, but the final summation is implemented using three separate programs.

The first program loads one pixel’s data and performs the subtraction and squaring for each of the color channels. This result is stored in the smaller, but faster shared memory that is accessible to all the threads in a block of threads that are executed together. Since CUDA can execute only up to 512 threads together in one thread block, the results from that thread group are added together and written out as a single value to the graphic card’s global memory, which other thread groups can access.

The second program takes the intermediate sum from each thread group, sums those values, and writes that value back to global memory. This program executes only if the number of intermediate results is larger than the maximum number of threads per block.

After the number of intermediate results that can fit within one thread block are computed, the final program adds all the intermediate sums and stores the result in a variable that is read back to system memory. The encoder then finishes the PSNR calculation on the CPU and continues execution.

The reference image is transferred to the graphic memory via CUDA’s standard memory copy functions. CUDA has two options to acquire the guessed image from OpenGL. The first method is to copy the image to system memory followed by a copy to CUDA’s memory on the graphics card. This requires that the image data detour through system memory to allow the main program to copy the data into CUDA controlled memory. The second option is to use a pixel buffer object (PBO) to store the image data. PBOs can keep the data on the graphics card’s memory and allows CUDA to access the data by mapping the buffer onto its memory space. Both methods are implemented to show the effect of data handling within CUDA’s API.

V. EXPERIMENTAL SETUP

Since there is no existing simulator for graphics cards or CUDA-capable devices, all testing took place on a Core i7 920 at 2.67 GHz and an nVidia GeForce 9800 GTX+. The rest of the system specs can be found in Table 1. All testing took place in Windows XP using nVidias 196.21 drivers. All of the programs were compiled using Visual Studio 2005. We used the August 2008 version of the DirectX SDK, the OpenGL Utility Toolkit (GLUT) 3.7.6 [20], and the OpenGL Utility Toolkit version 1.5.1 [21] to handle the 3D rendering. CUDA programs are compiled using the CUDA Toolkit version 2.3. The authors used Intel’s VTune [22] to monitor CPU events during execution to show how the processor is utilized during the execution of each program.

Table 1. System Specifications.

<table>
<thead>
<tr>
<th>L1 Instruction Cache</th>
<th>32 KB per core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4-way associative</td>
</tr>
<tr>
<td></td>
<td>64 B lines</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32 KB per core</td>
</tr>
<tr>
<td></td>
<td>8-way associative</td>
</tr>
<tr>
<td></td>
<td>64 B lines</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB per core</td>
</tr>
<tr>
<td></td>
<td>8-way associative</td>
</tr>
<tr>
<td></td>
<td>64 B lines</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8 MB shared</td>
</tr>
<tr>
<td></td>
<td>8-way associative</td>
</tr>
<tr>
<td></td>
<td>64 B lines</td>
</tr>
<tr>
<td>System Memory</td>
<td>3 GB DDR3 @ 1066 MHz</td>
</tr>
<tr>
<td>Chipset</td>
<td>Intel X58</td>
</tr>
</tbody>
</table>
The encoder uses a 300-frame clip of a commonly used video in model based video research titled wow [18, 23-24]. The clip is generated by decoding the (FAPs) from the existing wow sequence and saving the frames as bitmap files. These bitmaps are then stored in a ramdisk to remove the overhead of reading files from the hard drive. The size of the rendered OpenGL image is set to 352x288 (CIF) resolution [25]. The encoder will encode the 300 reference frames using the same model in the image and will exit upon completion of the last frame.

VI. RESULTS AND DISCUSSION

A. Total Runtime

Table 2 shows the total runtime of each program. The DirectX-based CPU program took 437 seconds to process the 10 seconds of video. When the error calculation is moved to the GPU using the pixel shader, the completion time drops to 83 seconds, which is a 5.25x improvement. However, the OpenGL and CUDA solution that used the OpenGL interoperability (as used in nVidia’s CUDA-based OpenGL post-processing example [26]) was slower at 98 seconds, suggesting that it is more efficient to keep the data within the API rather than move it. However, when the image is copied to system memory before moving to CUDA’s memory space, a supposedly less efficient method, the total runtime is reduced to 74 seconds. Most surprising is the OpenGL-based program that performed the calculations on the CPU finished in 54 seconds. Thus, implementations that should be less efficient complete the same computations in less time.

To find out where the time is being spent, the CPU time from all the CPU cores is broken down by process in Figure 5. In the DirectX- and CPU-based code, the majority of the time is spent in the main program. When the error calculation is moved to the pixel shaders, the time spent in the main program drops while the supporting libraries gain a few seconds. On the other hand, the OpenGL programs only see a few seconds spent in the main program. The CUDA-based programs see substantially more time spent in the OpenGL support libraries, CUDA libraries, and the kernel, eclipsing the time reduced from the main program. Based on this, CUDA requires a significant amount of time to service its data transfer and computation execution.

Overall, the programs that move the error calculation to the GPU see a drop in the time spent in the main program code, but the required support libraries see additional time to handle the GPU-based computations.

B. Cache Statistics

The cache statistics in Figures 6-8 show some interesting trends with the amount of data handled with each implementation.

The Direct3D pixel shader implementation demonstrates the type of changes that are expected when a data-heavy computation is moved to the GPU. The total number of cache accesses drop drastically, especially in the higher level caches, since the CPU is no longer handling the large image data. There is also a slight decrease in the cache hit rates, but the high spatial locality of the image data probably helped boost hit rates.

However, the OpenGL implementations see either little change in the number of cache accesses or a large increase in the number of cache accesses. While this behavior is expected for the implementation that uses system memory to move the data between OpenGL and CUDA, the CUDA implementation that uses the PBO sees more memory accesses. In addition, cache hit rates

![Figure 5. CPU time (based on unhalted cycles)](image-url)
Figure 6. Total cache accesses by program

Figure 7. Normalized Cache Accesses (Normalized to OpenGL-CPU)

Figure 8. Cache Hit Rate
suffer when CUDA is used, suggesting that utilizing CUDA adds memory overhead on top of the computational overhead seen in the CPU time breakdown.

C. Processor Stalls

Finally, an analysis of the processor stalls shown in Figure 9 narrows down the source of CUDA’s unexplained behavior.

Between the two Direct3D programs, there is a marked decrease in the two most prevalent stalls, the store buffer stalls and the reservation station stalls. The store unit stall decrease is attributed to the decrease in the amount of data transferred by keeping the hundreds of kilobytes of image data on the graphics card and transferring only the 4-byte result. The reservation station stalls decrease shows the superscalar hardware attempting to parallelize the error calculation due to large number of pixels and the limited number of processing units on the CPU.

Looking to the OpenGL-based programs, the CPU-based implementation sees only a small number of store stalls, despite needing the image data to move to system memory. OpenGL natively handles reading pixel information from a graphics card in the API and transfers the data via direct memory access. Direct3D, on the other hand, requires that data to be copied using the C memcpy function, using the processor to load and store the data. The CUDA implementations also have a large number of store unit stalls, suggesting the CUDA specific CudaMemcpy function also utilized the main processor to move the data to CUDA. This, combined with the cache analysis, suggests that the CUDA libraries are handling the data transfer between the two GPU computation libraries in an inefficient manner.

To verify that the data copy is the utilizing the CPU, the CUDA-based encoders encode the same video at three different input resolutions. The increase in the number of bytes transferred and the number of store unit stalls is
shown in Figure 10. When the number of store unit stalls is compared to the amount of data copied, there is a close correlation between the increase of both the input size and the store unit stalls.

In addition, to verify that it is the data copy and not the kernel execution that is generating the store unit stalls, a simple CUDA program was written that executed its kernel 100 times with and without a data copy before the kernel executes. The program that copied the data before the kernel executed saw a 74x increase in the number of store unit stalls. The additional store unit stalls seen in the CUDA program were the result of the image data moving to CUDA controlled memory.

D. Discussion

Based on these findings, it appears that CUDA's libraries must use code running on the main processor to move data between the two graphics card APIs. If there is no option to perform the copy directly on the graphics card and system-level code is required, there appears to be a more efficient method of handling the transfer. The OpenGL implementation that used the CPU to calculate the MSE showed that DMA can handle the data transfer more efficiently because of the shorter total execution time and better resource utilization.

The CUDA libraries handle data copying between system memory and graphics memory using the CPU's load/store unit, as shown by the results in this paper. This has the distinct disadvantage of requiring CPU time to handle the transfer of several kilobytes of data for every MSE calculation that the encoder encounters. In addition, this behavior seems to be tied to the presence of the C memcpy function in the code. The Direct3D programs require an explicit memcpy statement to copy data from the graphics card to system memory. In addition, the CUDA APIs have a CUDA-specific memcpy statement that is used to move data to CUDA controlled memory. OpenGL, however, uses DMA to transfer the pixel data to system memory. Since the OpenGL implementation that performed the MSE calculation on the CPU was the fastest and did not use any memcpy functions in the author's code, DMA is a much faster and efficient method for moving data.

In spite of this discovery, we wanted to know if a GPU implementation of the MSE calculation was faster on the GPU than on the main processor. To test this, the two main components of the MSE calculations were timed individually and the entire calculation was timed to show if and where the calculation performed better. Unsurprisingly, the highly parallel subtraction and squaring was 12.9x faster using CUDA, while the highly serial summation was about 5% slower on the GPU. The entire MSE calculation was 70% faster on the GPU than on the main processor, showing that there should have been an improvement if the data-handling problem did not exist.

VII. CONCLUSIONS

While GPGPU computations can provide significant computational speedup over the main system processor, they still suffer from one of the main bottlenecks present in any computer system: moving data to where it is needed. Here, we performed an analysis of a standard model-based coding system, an application in which data generated on the graphics card using one GPGPU solution is needed as input for another GPGPU API.

While the initial location and destination reside on the graphics card, nVidia's data-handling within CUDA shows some inefficient trends. Both CUDA implementations showed an increase in total CPU work and an increase in the amount of data handled by the CPU. Worse, the integrated OpenGL interoperability provides worse performance and significantly more overhead than if it is not used.

These problems stem from CUDA's use of the main processor to move data to the graphics card's memory instead of handling the data transfer internally on the graphics card or using the more efficient DMA architecture. Until this is rectified, programs that are more data intensive than computationally intensive will not find much, if any, improvement in computation time by using CUDA.

VIII. REFERENCES


Symposium on Parallel & Distributed Processing, 2009.


