Exploiting current-generation graphics hardware for synthetic-scene generation

Michael A. Tanner[a], Wayne A. Keen[b]
[a]Air Force Research Laboratory, Munitions Directorate, Eglin AFB FL 32542-6810
[b]L3 GS&ES, 51 3rd Street, Building 9, Shalimar, FL 32579

ABSTRACT

Increasing seeker frame rate and pixel count, as well as the demand for higher levels of scene fidelity, have driven scene generation software for hardware-in-the-loop (HWIL) and software-in-the-loop (SWIL) testing to higher levels of parallelization. Because modern PC graphics cards provide multiple computational cores (240 shader cores for a current NVIDIA Corporation GeForce and Quadro cards), implementation of phenomenology codes on graphics processing units (GPUs) offers significant potential for simultaneous enhancement of simulation frame rate and fidelity. To take advantage of this potential requires algorithm implementation that is structured to minimize data transfers between the central processing unit (CPU) and the GPU. In this paper, preliminary methodologies developed at the Kinetic Hardware In-The-Loop Simulator (KHILS) will be presented. Included in this paper will be various language tradeoffs between conventional shader programming, Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), including performance trades and possible pathways for future tool development.

Keywords: scene generation, graphics processing, parallel processing

1. INTRODUCTION

The parallel nature of GPUs makes them an efficient means for solving a large set of scientific and engineering problems. Traditionally, general purpose computing on GPUs (GPGPU) has required a highly specialized skill set. Until recently, implementation of algorithms on a GPU had to be reduced into a sequence of polygon translations and rotations. Random memory access for reading or writing was not allowed. New technologies have since emerged that remove these limitations.

This paper provides an overview of the technologies that have been introduced to the marketplace over the past few years. A brief background on the predominant programming architecture (NVIDIA Corporation's CUDA technology) is provided, along with a methodology for systematically porting software from a CPU-based system to the GPU. Finally, the paper provides a brief example of how the methodology can be applied to a realistic problem, demonstrating the performance improvements realized by porting the software to the GPU.

2. GPGPU BACKGROUND

2.1 Programming Languages

Recently, two main graphics card manufactures, ATI and NVIDIA, released software development kits that allow a programmer to access the GPU's power using general purpose code requiring random memory access. This alleviates the key technical difficulties associated with solving problems in the graphics hardware. ATI called their solution Close To Metal (CTM), which has since been renamed Stream Software Development Kit (SDK) [1]. NVIDIA's solution, CUDA, currently dominates the GPGPU market[2].

Stream SDK and CUDA are vendor-specific solutions to the GPGPU problem. Apple Computer, Inc. saw the potential to bridge this gap and pushed for an open standard to be created which would allow GPGPU programming for any vendor. On December 8, 2008 the specification for OpenCL 1.0 was released by the Khronos Group, creator of the OpenGL standard. OpenCL standardizes a GPGPU language, based on the modern dialect of C, which will allow the same code to be executed on any type of GPU.

2.2 GPU versus CPU

CPUs are highly optimized for serial processing of data. In general, the CPU processes one command at a time and operates on only one unit of data. Large amounts of die area are dedicated to control logic and cache to decrease the overall clocks per instruction (CPI). The memory cache is able to prefetch data because of special and temporal locality within the code. Elaborate control logic allows instructions to be executed out-of-order and branch logic to be predicted. Very little overall die space is dedicated to arithmetic logic units (ALUs), since only one thread can execute at a time. The most modern CPUs (with up to 16 cores on a chip) simply create multiple copies of this basic architecture on one chip.

GPUs take a significantly different approach to computing. Instead of using a large area for cache and control logic, GPUs have very small control logic and cache blocks for a large number of threads. A CPU is saturated with only a few threads, whereas a GPU needs to have threads in the thousands before saturation occurs. This comes at a cost though; individual threads on a GPU will invariable perform worse than the same one on a CPU. Therefore, the performance gained from the GPU is from executing highly parallel code.

GPUs are efficient at solving data-parallel problems. Data-parallel means that a single task operates identically and independently on a set of data. In computer architecture terms, it is equivalent to Single Instruction Multiple Data (SIMD) instructions. Figure 1 provides a graphical depiction of the difference between GPUs and CPUs.

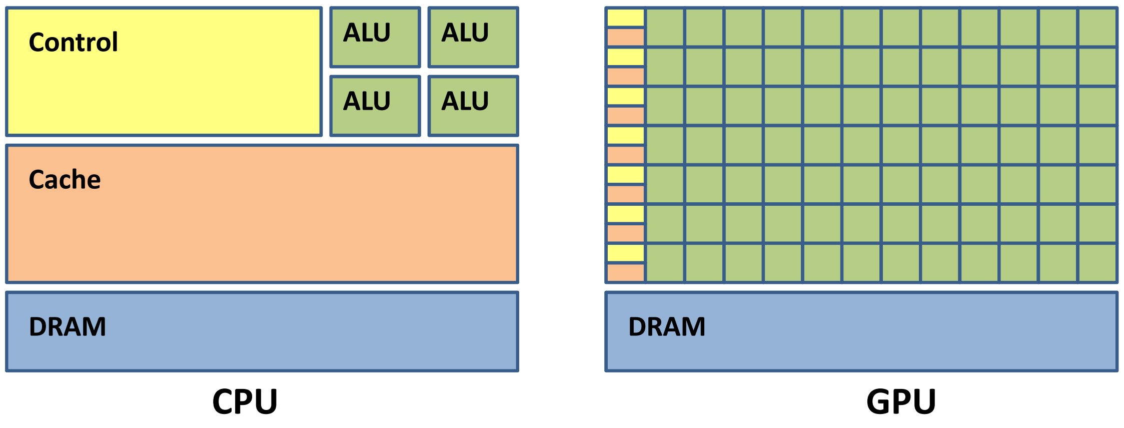

Figure 1. CPUs use large amounts of transistors for cache and control logic for a very limited number of threads. On the other hand, GPUs have very little thread overhead but have many ALUs for numerous threads. [3]

To illustrate the difference between CPUs and GPUs, consider matrix addition. For a CPU, the algorithm would look similar to Figure 2. This algorithm processes each element one at a time in series. This approach can be very slow when the matrices become large. If the matrix has n by n elements, the algorithm will take n^2 iterations of the loop to complete.

```
for i = 1 to HEIGHT do
    for j = 1 to WIDTH do
        $c_{i,j} = a_{i,j} + b_{i,j}$
    end
end
```

Figure 2. Calculate $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ on the CPU

```
i = current row
j = current column
$c_{i,j} = a_{i,j} + b_{i,j}$
```

Figure 3. Calculate $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ on the GPU

On the other hand, the GPU code in Figure 3 creates one thread for each matrix element, adds that element from **a** and **b**, and then stores that value into the proper element in **c**. A GPU can execute hundreds of threads simultaneously. For example, if a GPU can process 1,000 threads at a time, it will take $n^2/1{,}000$ iterations to complete. In practice, speed increases for GPU applications are typically between 10 to 200 times faster than their single core CPU counterpart.

The remainder of this paper will use NVIDIA CUDA terminology to explain the architecture of GPUs as described in the NVIDIA Programming Guide[3].

3. CUDA OVERVIEW

3.1 CUDA Terminology

When working with CUDA programs, it is necessary to understand some basic terminology. The *host* is the CPU, whereas the GPU is referred to as the device. A *kernel* is a small function that is executed on the device by a large number of threads.

Compute capability is a number assigned a GPU to represent its computational capabilities. A higher compute capability number means the GPU can handle more advanced mathematical and programming operations. For example, compute capability 1.0, 1.1, and 1.2 can only handle single point precision (32-bit floating point) operations, while the latest compute capability 1.3 has double precision. Also, 1.0 does not allow for any atomic memory operations, while 1.2 allows for shared and global atomic memory operations.

3.2 Thread Hierarchy

The thread hierarchy describes how the GPU executes threads as well as how threads interact with each other. When a kernel is sent to the device, it is assigned a *grid*. Grids are a logical mapping of the threads that are executed within the kernel. Each grid is made up of smaller components called *blocks*. These, in turn, are composed of individual threads. On the current generation of NVIDIA's GPUs, blocks can contain no more than 512 threads. Grid dimensions can be as large as 216 × 216. Figure 4 shows the thread hierarchy.

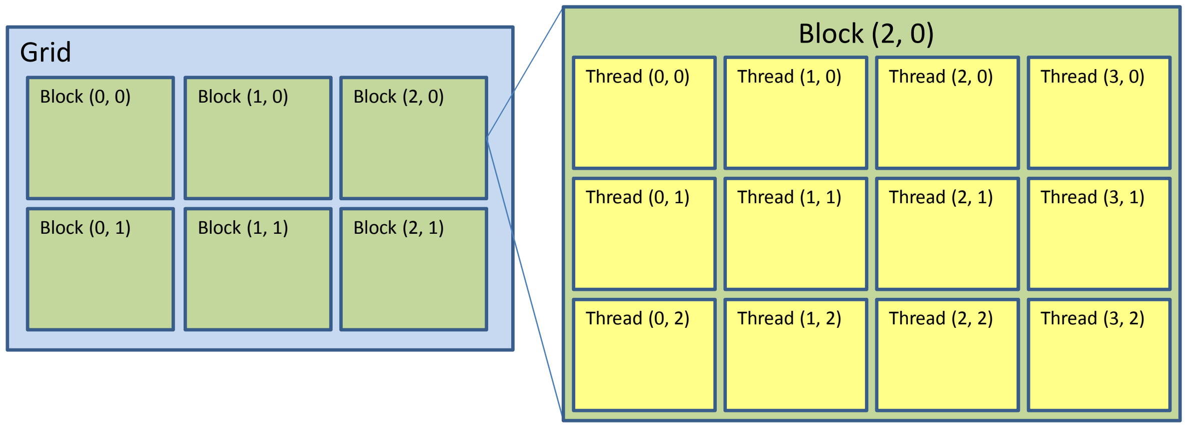

Figure 4. Every CUDA kernel is assigned a grid. Grids are subdivided into blocks. Each block contains up to 512 individual threads. Blocks and grids can be logically organized into multi-dimensional structures to simplify memory access and calculations.[3]

Grids can be logically organized into a 1D or 2D layout, whereas a block can also be organized in a 3D layout. This organization can be useful to efficiently calculate the array index for each thread. For example, if the data processed by the kernel is a 2D image, then it would make sense to use a 2D grid and block structure. The current index for each thread can be calculated using built-in CUDA variables, as shown in Equation 1. This index can then be used within each thread to perform a certain operation on the image.

$$\begin{aligned} \texttt{column} &\leftarrow \texttt{blockIdx}_x * \texttt{blockDim}_x + \texttt{threadIdx}_x \\ \texttt{row} &\leftarrow \texttt{blockIdx}_y * \texttt{blockDim}_y + \texttt{threadIdx}_y \end{aligned} \tag{1}$$

where `blockIdx` is a vector representing the current location in the grid, `blockDim` specifies the width and height of each block, and `threadIdx` is a vector representing the current location in the block.

3.3 Memory Hierarchy

Memory hierarchy and memory access methods are important to understand how to improve the speed of applications on GPUs. There are three different levels of memory: global, shared, and local. Two additional types of memory exist, texture and constant, but will not be covered due to scope limitations of this paper.

Global memory can be accessed by any thread, no matter what block it is in. This memory is the largest (most modern GPUs have up to 4 GB), but it is also the slowest. No memory reads from global memory are cached. In addition, since this memory is further away from the multiprocessor cores, there is a latency of about 400 to 600 clock cycles.

Shared memory is memory that can be accessed only by threads within the same block. Current GPUs have 16 KB of shared memory. Access to shared memory can be as fast as reading and writing to registers so long as there are no bank conflicts within a half-warp. A half-warp is defined as a group of 16 threads executed simultaneously in a single stream processor core. A bank conflict occurs if two threads within a half-warp are reading or writing to the same 32-bit block of shared memory.

Local memory can only be accessed by an individual thread. The CUDA compiler places as many local variables as it can into registers, but if there is overflow, variables are stored in a reserved section of the GPU's global memory. If variables do not fit in local registers, there will be a significant performance decrease.

The most important concept to understand about memory access is coalescing. This means that all half-warp memory accesses are within a given segment size of memory. Table 1 summarizes the rules for devices with compute capability 1.2 or higher. If these rules are followed, then only one memory access command is executed. If they are not followed, then individual memory fetch/store commands must be issued for each thread (16 individual memory fetch/store commands instead of one). For example, if the 2-byte words accessed by the threads in an individual half-warp are contained within a 64-byte contiguous spread in global memory, the read and writes to that memory will be coalesced, as shown in line two of Table 1. If just one of the words fall outside that 64-byte spread, then the processor will issue 16 individual memory fetch/store commands.

Table 1. Coalescing memory writes is very important for GPU performance. This table summarizes the rules for memory access within a half-warp (16 threads) for devices of compute capability 1.2 or higher.[3]

Byte Spread	Word Size
32 bytes	1-byte
64 bytes	2-bytes
128 bytes	4-bytes
128 bytes	8-bytes

3.4 Synchronization

Synchronization is limited within the GPU. Using the `_syncthreads()` command, threads within the same block can be synchronized. This limitation of only block-level synchronization results from all of the thread blocks not executing at the same time. The GPU has a scheduler which submits blocks for execution on an individual Single Instruction Multiple Thread (SIMT) multiprocessor core. Each SIMT contains eight scalar processor cores. These cores are capable of executing one warp (or set of 32 threads) at a time. The NVIDIA GTX 280 has 30 SIMT multiprocessor cores, which means it can execute up to 240 blocks simultaneously. Synchronizing within a core (i.e., block) requires only four clock cycles, but synchronization between multiple blocks is much more costly because of the interprocessor communication.

3.5 Performance Notes

Writing efficient code on the GPU can be difficult. Memory bandwidth and latency are two important concepts that affect overall performance. As has already been mentioned, coalescing is also very important to speed up the execution of code on the GPU.

What is not as obvious is that sending data between the host and device can cause a bottleneck in a program. On all motherboards, communication between the CPU, GPU, and RAM is mediated by the northbridge, as shown in Figure 5.

There is a limit of 12.8 GB/s between the CPU and northbridge. One-way communication between the CPU and GPU is limited to 8 GB/s with PCIe 16x Generation 2. An uncompressed, 24-bit, 1920x1080 (1080p) image uses 5.9 MB of memory and takes about one millisecond to copy from the CPU to GPU. Therefore, two milliseconds are used just to transfer data, without any useful computation.

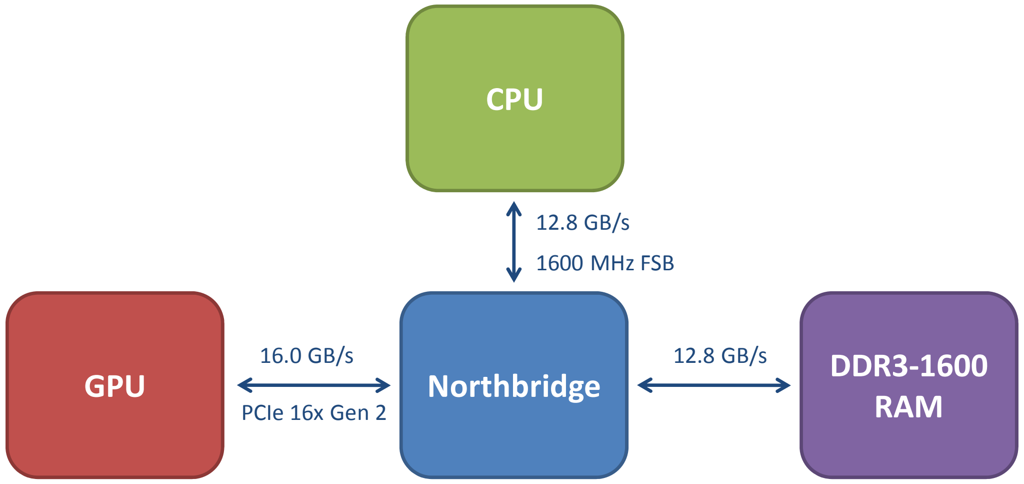

Figure 5. The CPU and GPU cannot communicate directly with each other. The northbridge mediates between the CPU, GPU, and RAM.

The data-parallel operations executed on the GPU need to be great enough that their quick execution more than makes up for the transfer time to and from the GPU. For example, a single operation of matrix addition or transpose would probably execute more quickly on the CPU, while matrix multiplication would be quicker on the GPU.

Device-to-device memory transfer refers to moving data to another portion of global memory on the same graphics card. These transfers have a bandwidth of up to 141.7 GB/s, which makes them not as susceptible to the bandwidth problems described above.

4. PROCESS FOR PORTING SOFTWARE TO THE GPU

There are a number of considerations one must take into account when porting software from running only on the CPU to running on both the CPU and GPU. First, and most importantly, it must be determined if the code or problem being ported can run efficiently on the GPU. A full understanding of the GPU and CPU architectures (Sections 2 and 3) is important for this preliminary analysis. The GPU works best for problems that can be subdivided into a large number of smaller problems that can be solved independently, i.e., very little inter-process communication or synchronization. Significant performance improvement will most likely be seen if the following conditions are met: (1) the problem requires a large number of independent and complex calculations, (2) relatively few global memory fetches/writes are required for each thread, and (3) there is a limited amount of divergent logic between threads within the same half-warp.

The last point deserves a more detailed explanation. Threads in the GPU are executed together in half-warps, or groups of 16 threads. The processor can only execute identical instructions for each thread. In other words, if there is conditional logic (e.g., `if` statements, `for` loops, etc.) then there is the potential that one conditional statement will be evaluated as true for some of the threads and false for the rest. If this is the case, while the processor is executing the "true" logic for some of the threads, the other threads in the half-warp are idle. When the commands in the "false" logic are executed, the "true" threads are idle. A significant amount of idle time for threads decreases the overall performance of the kernel.

After determining there will likely be a performance improvement from porting the software, the next step is to determine which part of the software to implement on the GPU. Benchmark the software on the CPU to identify which functions or methods are the bottlenecks for the application. These bottlenecks are typically the first part of the software you want to port over to the GPU. Again, when you identify the bottlenecks, verify that they will port well to the GPU by using the three part test identified in the first paragraph in this section. If you are using GCC (GNU Compiler

Collection) as your compiler, you can use the `-pg` flag during compilation and after you execute the program you can analyze the overall execution time for each function by running `gprof` on the executable. Matlab has a similar function called `profile`.

At this point, you can run a few quick calculations to determine how much of an *overall* speedup you may see in your software. This is accomplished by applying Amdahl's Law,

$$\text{speedup} = \frac{1}{(1 - \mathrm{P}) + \frac{\mathrm{P}}{\mathrm{S}}}, \tag{2}$$

where *P* is the percentage of the program (in execution time) that you will be porting to the GPU and *S* is the amount of speedup you expect to get on the GPU. For example, if you are porting over a function that takes 25% of the overall execution time and you expect to get a 10 times speedup, you would expect a speedup of about 1.29, a 29% increase over the baseline.

There are two major implications of Amdahl's Law. First, you should always check to see if porting the software will provide the overall speedup that you desire. In other words, even if you get a 50 times speedup for a function that only takes 5% of the execution time, your overall speedup is only 5.1%. The second implication is that there are diminishing returns from speeding up an individual function. Take the previous example, but assume that you can achieve a 200 times speedup for the function that takes 5% of the execution time. Now your overall speedup is 5.2%. Even though the function performance was improved 4 times, the overall performance was only improved by 0.1%. Your time would more likely be better spent optimizing a different function.

After identifying a function to port, the next step is to get it working on the GPU. On the first implementation, do not focus on optimizations. The primary goal is to produce functionally correct code executing on the GPU. Verify that it is producing the correct output by writing automated test plans to compare the outputs of the CPU and GPU implementations.

If all the conditions described in the first paragraph of this section apply, you can expect roughly a two to ten times speedup from this first implementation. To achieve greater speedup, you must optimize the code for the GPU. To do this, you must fully understand the material covered in Sections 2 and 3 from this paper. The NVIDIA Programming Guide and programming forum are also valuable resources. The best ways to improve performance are to limit the number of global memory reads and writes through the use of shared memory, decrease the amount of divergent logic, maximize the use of coalesced memory access (Section 3.3), and limit the amount of data transferred between the GPU and CPU (Section 3.5). If the problem fits well into the GPU architecture, based on the criteria defined at the beginning of this section, you should expect to see speedups on the order of 100 times.

5. CASE STUDY OF SOFTWARE PORT

5.1 Planck's Law

An example will be used to introduce the potential speed increases that can be achieved through the use of the GPU. In many high-fidelity scene generation scenarios, it is necessary to calculate the exact spectral radiance of an object at various wavelengths. Plank's law can be applied in this scenario to calculate the spectral radiance for a given wavelength and temperature, as shown in this equation,

$$I(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{\frac{h\nu}{kT}} - 1}. \tag{3}$$

In Equation 3, *v* is the frequency in hertz, *T* is the temperature in kelvin, *h* is Planck's constant, *c* is the speed of light, and *k* is Boltzmann's constant. A scene generation scenario using Plank's law may use a ray casting or tracing algorithm to determine all the visible points in the scene and then evaluate the spectral radiance of each facet in the scene across a spectrum of wavelengths.

5.2 Step 1 – Analyze Potential Gain from Porting Software

Before porting the software, an evaluation of whether the problem will port well into the GPU domain is necessary, referring to the three criteria listed at the beginning of Section 4. For the evaluation it is assumed that each thread calculates a single spectral radiance value for a single object. The first criterion is met since each thread performs a complex calculation that is completely independent of other threads. Each thread only has to read in two values, frequency and temperature, from global memory and write one value back to global memory, so the second criteria is satisfied. Also, it is important to note that there is relatively little communication between the GPU and CPU compared to the time required for the calculations. Finally, there is no conditional logic necessary to perform each calculation of Planck's law since it is simply evaluating an equation with known inputs. All the criteria have been met, indicating this problem will have significant performance improvement if the Plank function is ported to the GPU.

5.3 Step 2 – Benchmark the CPU Software to Identify Bottlenecks

In this example, the bottleneck of the software is the Planck's law function. The remainder of the software is overhead necessary to read in temperature data and wavelength range for the objects. Since this single function accounts for most of the execution time of the program, and the function maps well to the GPU paradigm, we can expect significant *overall* program speedups when it is ported to the GPU. In a straightforward problem like this, a detailed benchmark is not strictly needed. This program essentially consists of two steps: (1) read in the problem data from a file, (2) loop over that data and compute the Planck's law on each value. Without further analysis, it can be determined that only the Plank's calculation can be made parallel to execute on the GPU.

5.4 Step 3 – Implement on the GPU

Implementing Planck's equation on the GPU is a straightforward process. A single kernel is created to perform all the Planck equation calculations. All the memory reads and writes are coalesced by default as long as thread index is directly used to calculate the global memory location for the temperature and frequency. The amount of data being transferred over the GPU is minimal considering that most of the program time is spent calculating the result of the equation. The results from running this program on the CPU, as compared to the GPU, are shown in Table 2. The GPU used was the NVIDIA Quadro FX 5800, and two quad-core Intel Xeon processors were used for the CPU tests.

Table 2. Results from running Planck's equation on 16,777,216 (or 4,096 * 4,096) different wavelengths at a set temperature. The CPU scales linearly compared to the number of cores, whereas the GPU performs more than two times faster than would be expected with a linear performance improvement. This is due to coalesced memory access along with context switching when the processors are waiting for a memory fetch or write.

Implementation	Cores	Time (s)	Speed Increase
CPU	1	1,287.92	1.00
CPU	8	160.55	8.02
GPU	240	2.25	573.43

5.5 Interpretation of Results and Lessons Learned

This example demonstrated how a problem that would have been intractable to solve without large server farms can become much more manageable with just a single consumer-grade graphics card that cost well under $1,000. The significant speedups seen in this scenario are only seen in problems that port *extremely* well to the GPU design paradigm, it is not typical to see speedups larger than 200 times for most well-suited problems. GPU speedups are achieved in part because of memory coalesced reads and writes. In addition, compared to a CPU, the GPU processors have very low overhead to switch in new threads while old threads are waiting for a memory fetch or write to complete. The more complicated the GPU kernel and the more global memory access required, the less the overall speedup will be.

This example was presented in order to demonstrate to the reader the process of porting software from the CPU to the GPU. The example provides a straightforward case as a template for more complicated situations. A slightly more involved scenario for matrix multiplication is presented in Chapter 2 of the CUDA Programming Guide[3]. Numerous optimization techniques are provided in the CUDA Best Practices Guide. Assistance can also be obtained on the NVIDIA CUDA forums online where there is an active community of people willing to help people of all levels of experience.

6. FUTURE WORK

6.1 NVIDIA Fermi

The future of GPGPU programming holds a great amount of potential benefit for engineering simulation. In particular, NVIDIA is expected to release its next generation of GPUs in the first quarter of 2010. This architecture, codenamed Fermi, should significantly improve on NVIDIA's previous generation. [4] The number of computing cores available on each board has more than doubled from 240 to 512. From this feature alone one would expect at least double performance for any software run on Fermi compared to the GT200 architecture. However, other improvements should allow the speedup to be as much as eight times faster than the GT200 architecture. The main features include 10 times faster context switching, concurrent execution of multiple kernels, and caching of global memory. The user can specify how much of the 64 KB shared memory will be used as L1 and L2 cache. After this configuration is set, caching is transparent to the programmer.

Not all of the Fermi improvements were designed solely for performance improvement. For example, Fermi boards contain a unified address space which allows for C++ code to be executed natively. Now engineers can use object-oriented programming to rapidly port currently-existing solutions. Also, Fermi is the first graphics card architecture ever to introduce error checking and correcting (ECC) for its memory. This is especially important for scientific applications where precision and accuracy is vitally important for all calculations in a variety of different settings.

6.2 OpenCL

OpenCL was briefly mentioned in Section 2.1. This new technology allows the same piece of code to be compiled and executed on a variety of different platforms. The same code could be executed on both NVIDIA and ATI products. There are a number of reasons that the OpenCL community has not made CUDA obsolete. Probably most important is that the OpenCL drivers are in their infant stages and perform poorly in comparison to the vendor technologies. The CUDA example presented in this paper (Section 5) consistently runs about 20% - 30% slower when implemented in OpenCL. ATI's OpenCL drivers have not yet been released from beta.

As the driver technology improves, OpenCL can be expected to perform at least as well as other programming architectures. A valuable research task with mature OpenCL technology would be to compare the performance of new generations of ATI and NVIDIA GPGPUs.

7. CONCLUSION

The purpose of this paper was to provide a brief tutorial into the GPGPU world, providing the user with a toolset that will allow them to understand the GPU architecture. Specifically, the CUDA programming model was presented in detail along with a list of references for the reader to further investigate this technology. A straightforward methodology for porting software from the CPU to the GPU was presented along with a representative example. The example was an idealistic case that realized significant performance improvement on the GPU in comparison to the CPU. These demonstrated techniques can be applied to a variety of different scene generation software algorithms to improve the performance and fidelity of real-time and offline systems.

ACKNOWLEDGEMENTS

The authors would like to thank Mr. Charles Coker for providing the time and resources to conduct the research in this paper. Also, Mr. Dennis Crow, who implemented Planck's law in OpenCL, CUDA, Brook, and OpenMP. Without his efforts, this research would have taken significantly longer to conduct.

REFERENCES

[1] T.C. Jansen, "Gpu++ an embedded gpu development system for general-purpose computations," Master's thesis, University of Munich, August 2007.

[2] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *PPoPP '08: Proceedings of the 13th*

ACM SIGPLAN Symposium on Principles and practice of parallel programming. New York, NY, USA: ACM, 2008, pp. 73-82.
[3] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.3.1*, NVIDIA, August 2009.
[4] *Whitepaper – NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM},* NVIDIA, February 2010.