Jonathan Paulick, Jonathan Phung

3/19/23

GPU Acceleration of Monte Carlo Light Transport Simulation

Abstract

In collaboration with Dr. Zachary Simmons' of the Milwaukee School of Engineering's (MSOE) Physics and Chemistry Department, we attempted to accelerate the industry-standard method of light transport simulation using CUDA, NVIDIA's proprietary GPU programming language. The industry standard simulation approach is Monte Carlo, which lends itself to significant parallelization because each Monte Carlo trial can is simulated independently of the other, the fundamental underpinning of such methods. GPU acceleration of this method using CUDA has led to a significant performance speedup of approximately 16 times at best and has paved the way for future applications in the department.

Background

In biomedical optics, understanding the propagation of light in tissue could lead to major breakthroughs in both "diagnostic" and "therapeutic" applications of light (Simmons, Oct. 2022, personal communication). For diagnostics, diseased tissue may have different optical properties than normal tissue, and light-based machines could provide more efficient and non-invasive diagnoses to that end (Simmons, Oct. 2022, personal communication). Likewise, for therapy, an

understanding of the optical properties of tissue allows for the delivery of light as treatment, including “surgery-like” applications, such as “ablation” (material removal via focused light) to more “exotic treatments” such as “metabolic stimulation” (the breaking down and building up of complex molecules) and “optogenetics” (the controlling of light-sensitive cells) (Simmons, Oct. 2022, personal communication).

The industry standard method of light propagation simulation is to use Monte Carlo methods (Simmons, Oct. 2022, personal communication). Monte Carlo methods are probabilistic methods that attempt to solve a problem by running many independent and identically distributed experiments to approximate the solution through the Law of Large Numbers. One example of a problem that could be solved using Monte Carlo methods would be approximating the probability of rolling a pair of numbers on two fair, six-sided dice. While this problem’s closed-form solution is trivial to compute using the counting principle, it could also be solved using Monte Carlo methods by simulating many dice rolls, such as 100,000. By the Law of Large Numbers, the simulated probabilities of each pair of numbers appearing in a dice roll will approach the true probability.

For light propagation simulation, the trajectory of the photons of light are modeled probabilistically as it travels (Simmons, Oct. 2022, personal communication). Based on the modeled optical properties and geometry, as a photon moves through the medium, it can be scattered by, absorbed by, or escape from the medium (Simmons, Oct. 2022, personal communication). This process is then repeated for many photons, ultimately leading to a description of the overall behavior of light in the modeled medium (Simmons, Oct. 2022, personal communication). An example of such light propagation is depicted in the following figure:

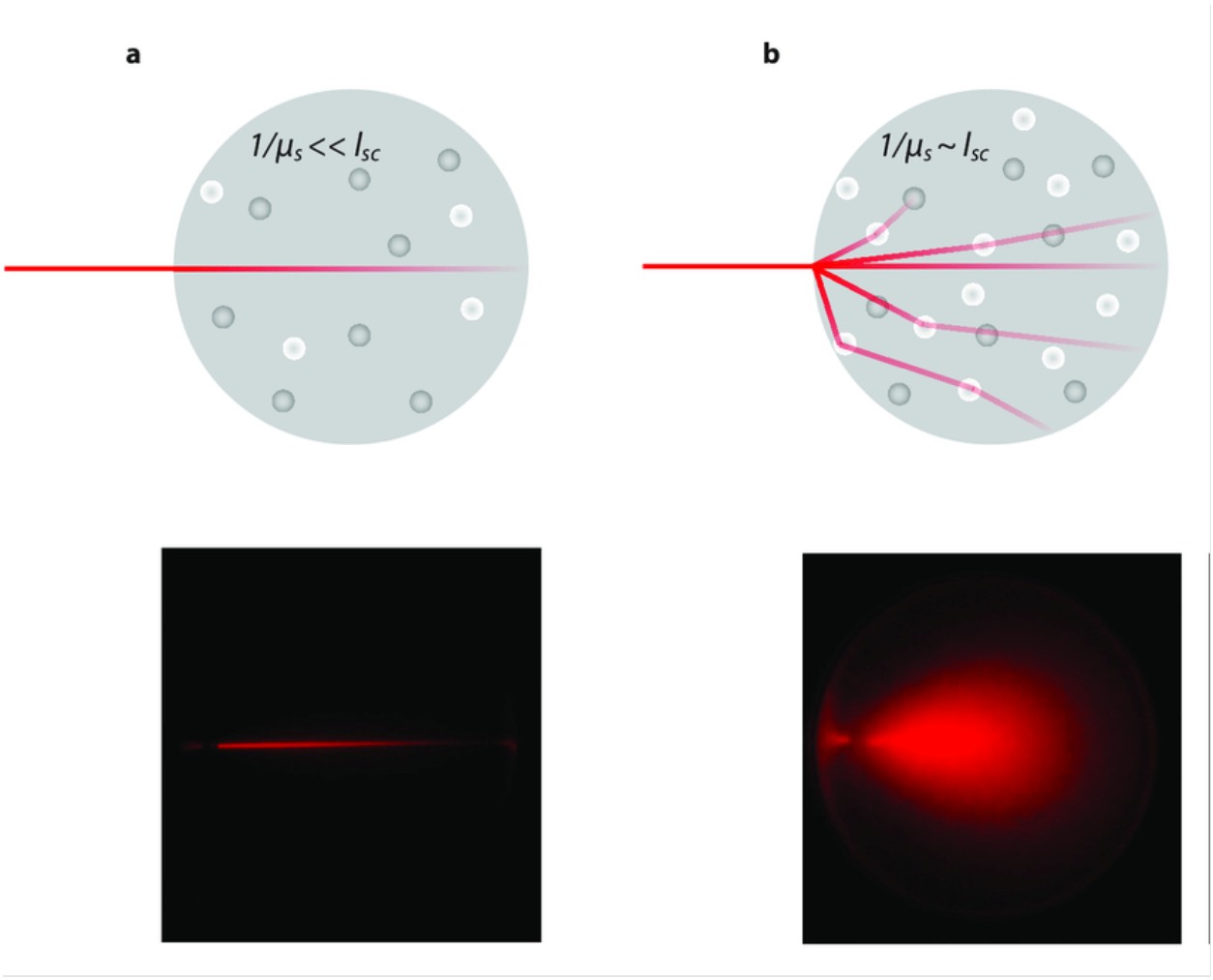

Figure 1.1: The propagation of light, in this case a laser, through two different mediums. The first is one that does not propagate light at all, which the second propagates light greatly (Ripoll, 2011).

With respect to the research the Physics and Chemistry Department is advancing, Simmons believes these simulations are a first step towards developing a full project for more dedicated research into light propagation (Simmons, Nov. 2022, personal communication). An example of their work is depicted in the following figures:

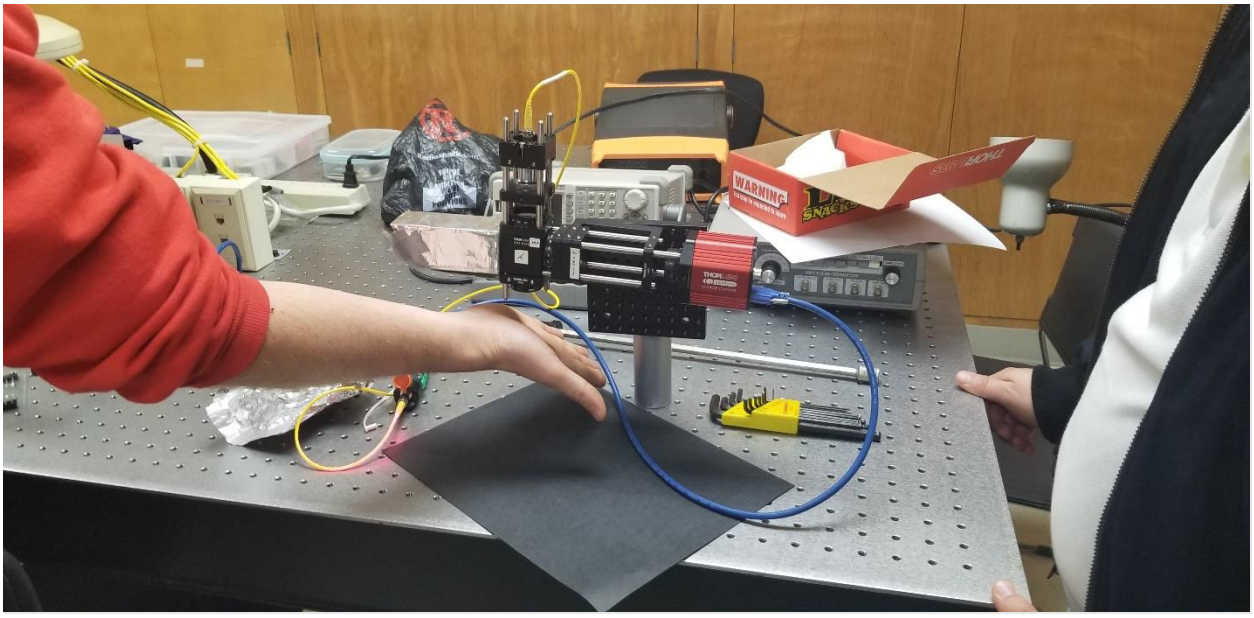

Figure 1.2: An image of the enhanced backscattering setup used in their experiments. A laser is sent through the yellow fiber optic cable and shoots down from the junction onto the black piece of paper (Simmons, Nov. 2022, personal communication). Something, in this case a hand, is placed under it and moved, reflecting the laser back into the junction and into the red camera on the right (Simmons, Nov. 2022, personal communication). Stray photons can be reflected to the left of the junction, which is also a dark black to capture them (Simmons, Nov. 2022, personal communication).

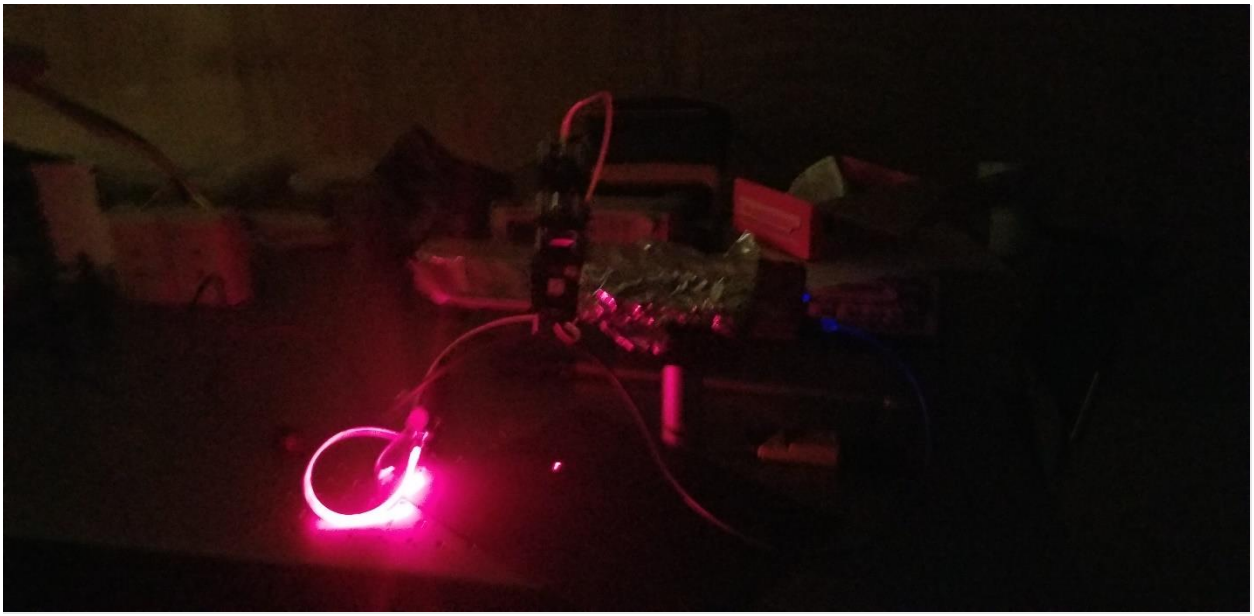

Figure 1.3: An image of the same setup in the previous figure but in the dark. Such optical experiments need to be conducted in the dark to ensure that only the light from the laser is being captured (Simmons, Nov. 2022, personal communication).

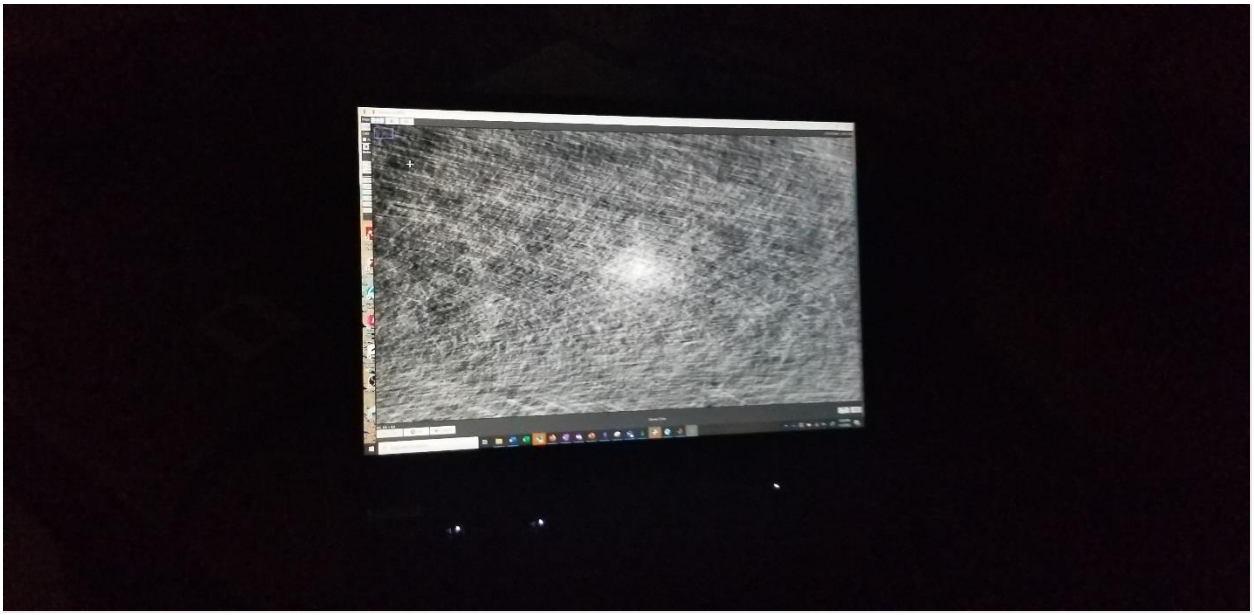

Figure 1.4: An image of the data being captured by the camera in the setup in the previous two figures. For this experiment, a hand is being held under the laser. Note that most of the data is

noisy except for a strong white dot in the center, which is the data. That dot measures the peak scattering of the laser that is returning to the camera (Simmons, Nov. 2022, personal communication). The highest peak scattering would have an entire screen of white, no peak scattering would have an entire screen of noise (Simmons, Nov. 2022, personal communication). Peak scatterings in between would be represented as a solid white circle starting from the center that would increase in size as the peak scattering increased (Simmons, Nov. 2022, personal communication).

The main difference between the work here and their work is the difficulty in measuring the movement of photons through a medium. The reason that enhanced backscattering needs to be used in their experiments is that light cannot be easily measured once it enters the medium but is easily measured once it leaves it (Simmons, Nov. 2022, personal communication). The simulation, if nothing else, provides a simple example of CUDA code to accelerate future work in the field, but is expected to provide many more benefits, which will be elaborated upon in the following section.

Algorithm

The algorithm is inspired by Dr. Scott Prahl's implementation of a simple Monte Carlo simulation of light propagation using a basic medium (Prahl, 2018, "Drop-Dead Simple Monte Carlo Codes" section). The program calculates "volumetric heating" as a function of "depth," where photons can be absorbed by the medium as they pass through it, generating heat. In general, most photons will generate lots of heat near the top of the medium, but few will make it to the very bottom (Prahl, 2018, "Drop-Dead Simple Monte Carlo Codes" section). The specific mathematics

and physics behind the algorithm are less important than the actual algorithm itself, and at a high level, it follows the same steps that Simmons describes.

To simulate one photon, the algorithm begins by initializing its three-dimensional position, three-dimensional velocity, and weight. While the photon still has a weight greater than 0, it will repeat the simulation process. The simulation process starts by having the photon move in a random direction. If it moves into the medium, some of the photon's weight will be turned into light and bounce back up the top of the surface, losing weight in the process. After it has moved, the photon will then be absorbed into the medium, turning some of its weight into heat at its current depth. Finally, after it has been absorbed, the photon will be scattered, giving it a new velocity in the medium. The process will then repeat.

As with most every Monte Carlo simulation, this algorithm was ripe for parallelization. Every photon can be simulated in parallel with no dependencies between photons. The results for every photon could then be aggregated at the end to the same effect. The only memory dependency is the final writing of the results after the conclusion of the simulation, which should stochastically consume a portion of the memory bandwidth over the entirety of the simulation. This also allows for one thread to simulate an arbitrary number of photons in parallel following the single program multiple data and task parallelism paradigms. An example execution of the application is depicted in the following figure:

```
phungj@dh-mgmt4:~/CS4981_GPU_Programming/source/Final_Project$ ./mc_cpu.out 100000
Host Photon Simulation Runtime (ms): 897.333008

Small Monte Carlo by Scott Prahl (https://omlc.org)
1 W/cm^2 Uniform Illumination of Semi-Infinite Medium

Scattering =   95.000/cm
Absorption =    5.000/cm
Anisotropy =    0.500
Refraction Index =    1.500
Number of Photons =   100000

Specular Reflection =    0.04000
Backscattered Reflection =    0.21933

Depth         Heat
[microns]     [W/cm^3]
     0        15.69142
    20        15.64131
    40        15.34257
    60        15.09813
    80        14.57534
   100        14.16942
   120        13.81268
   140        13.15010
   160        12.60067
   180        12.14764
   200        11.60336
   220        10.93949
   240        10.56965
   260        10.11071
   280         9.57895
   300         9.18011
```

Figure 2.1: A screenshot of the truncated output when having the application simulate 100,000 photons. Note the runtime metric in the first line of output, which is used in the benchmarking statistics. The truncated output is the heat generated at a depth deeper than 300 microns, including the extra heat.

As depicted in Figure 2.1 and supported by the Benchmarking section, when the CPU implementation is executed for 100,000 photons, it runs relatively quickly in under a second. If

simulating 100,000 photons provided a sufficient accuracy for the department's purposes, the value of GPU acceleration beyond a learning exercise would be questionable at best. However, Simmons makes a few arguments regarding why this project is still valuable even if some samples can be simulated via CPU: simulation efficiency depends on the modeled optical properties and GPU acceleration makes the reverse problem more feasible (Simmons, Oct. 2022, personal communication). With respect to the modeled optical properties, more complex and less responsive models can require longer simulation times per photon with more photons necessary to achieve a sufficient level of accuracy (Simmons, Oct. 2022, personal communication).

The reverse problem attempts to determine the optical properties of a medium given the experimental data gathered from it (Simmons, Oct. 2022, personal communication). In this case, the results would be the heat generated at each depth of the material and the optical properties would be the parameters, such as the anisotropy of the medium. Simmons argues that GPU acceleration makes this problem much more feasible, as many mediums could be simulated quickly, creating a library of simulations to compare unknown experimental data to (Simmons, Oct. 2022, personal communication). Much like the fundamental tradeoff between using a CPU implementation as compared to a GPU implementation, this approach would focus much more on throughput as compared to latency, as minimizing the overall time required to build such a library is more important than making any one simulation fast.

CUDA Implementation

Simulation Kernel

The simulation kernel used in the parallel GPU implementation is similar to the algorithm for sequential CPU implementation. The core functionality is the same as the kernel calls the move,

bounce, absorb and scatter methods while the photons weight is greater than zero. These methods are 90% unchanged from the CPU implementation, with the only changes resulting from the C rand method not being accessible on the GPU. The main difference between the sequential CPU implementation and the kernel for the parallel GPU implementation is the number of simulated photons. Each time we run the kernel, only one simulated photon is allowed to run. This single simulation allows us to call the kernel concurrently without a data dependency. Given the architecture of the Tesla T4, we simulated 512 photons per block. We were limited to 512 photons because there was insufficient memory to use 1024 threads per block. The rest of the differences stem from needing to move memory out of local registers into the global memory on the GPU. Looking at lines 525-531 in main.cu, we can see the writes to global memory. As these writes are relatively costly, we attempted to minimize the amount we needed to do. As part of this mitigation attempt, we utilized local variables in each kernel as they can be accessed and updated more quickly.

In translating the CPU implementation into helper functions for the GPU, we needed to learn cuRAND. To our dismay, we learned that the GPU could not call the C rand function. While this limitation makes sense, there is no drop-in replacement from NVIDIA and their CUDA libraries. To reduce the cost of calling a state stored in global memory, we instantiated a cuRAND instance in each kernel. In addition, cuRAND does not output the same values as the C rand function. To account for this, we created a helper method to scale the values to match the expected results. Lines 258-260 in main.cu are the cuRAND helper function and show how we accounted for the difference in expected output. At a high level we truncated the result of multiplying curand_uniform and the maximum random value in C into an integer.

Reduction Kernel

With the sequential CPU implementation, the use of reduction was unnecessary. With only one process writing to memory, we did not need to worry about race conditions. Race conditions occur when two or more threads modify a shared resource which can lead to unexpected values. In the parallel GPU implementation, since we are not using atomic operations to save into the same memory location, we need to combine the outputs from all the simulated photons. This kernel's goal is to minimize the cost of memory operations on the GPU.

In a time complexity analysis of the reduction code, a sequential implementation, regardless of where it runs, would be O(n). With our implementation of the reduction kernel, we implemented a work-efficient algorithm. This distinction means it prioritizes having a lower time complexity instead of reducing the allocated resources. The time complexity of our kernel is O((n-1)/log(n)), which simplifies down to O(n).

The reduction kernel we implemented utilizes n/2 threads, which means only 512 threads are needed to reduce 1024 elements into a single value. We do the first level of reduction while loading the data into shared memory to optimize the cost of reading from the GPU's global memory. One limitation of the implication is that it requires two warps worth of shared memory to ensure indexing stays within bounds. Taking the advice of Donald Knuth, we choose to set a static value for the amount of memory allocated to avoid premature optimization. This design choice allowed us to focus on large quantities of photons rather than smaller numbers, where the sequential CPU version would perform better.

We did not implement a single-threaded version of the reduction kernel for a multitude of reasons—the primary reason related to coping data from the GPU to the host. With the current reduction, only 104 copies take place from the GPU's global memory. However, if we make the reduction sequential on the host, the maximum number of copies jumps to the equation

(photons*104). An alternative would be to utilize atomic operations on the GPU; however, this would still require us to make multiple calls to global memory, which are inefficient.

This kernel did present several challenges as we are working with data that will not fit on a single GPU. We needed to get creative with performing the reduction across multiple kernel calls. The solution we created utilized the fact that the process of adding numbers is commutative. Specifically, we created a helper method which called the reduction kernel on the end of the array, tiling towards the beginning ensuring to overlap of the last element. Tiling is typical when the number of elements does not fit on the GPU. This decision reduced the number of unnecessary memory movements. Figure 3.1 demonstrates how the memory access for the reduction kernel helper works.

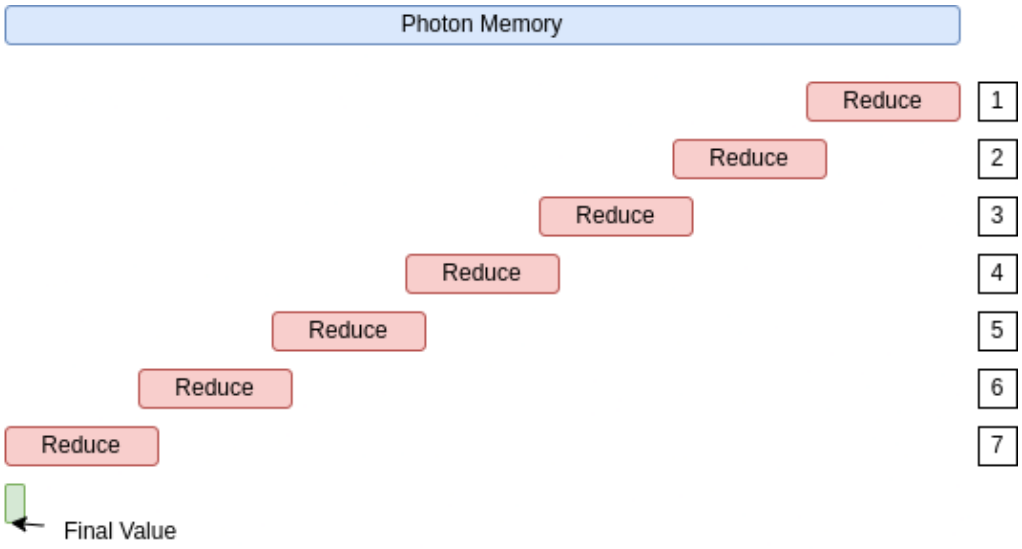

Figure 3.1: This visualization shows how we reduce the memory for a photon value to a single value. We intentionally have overlap with the reductions to reduce the number of times we move memory locations without operating on the data. A crucial part of this design is moving backward across the memory landscape, as it allows us to combine values with little cost and leaves the reduced element as the first.

A more trivial problem resulted from us utilizing shared memory in our reduction. Since shared memory is only available within the same block, we needed to call the reduction kernel multiple times to combine the values from multiple grids. The solution was even more trivial; use recursion with the helper function. This solution can reduce the three parameters within the same helper function.

Profiling

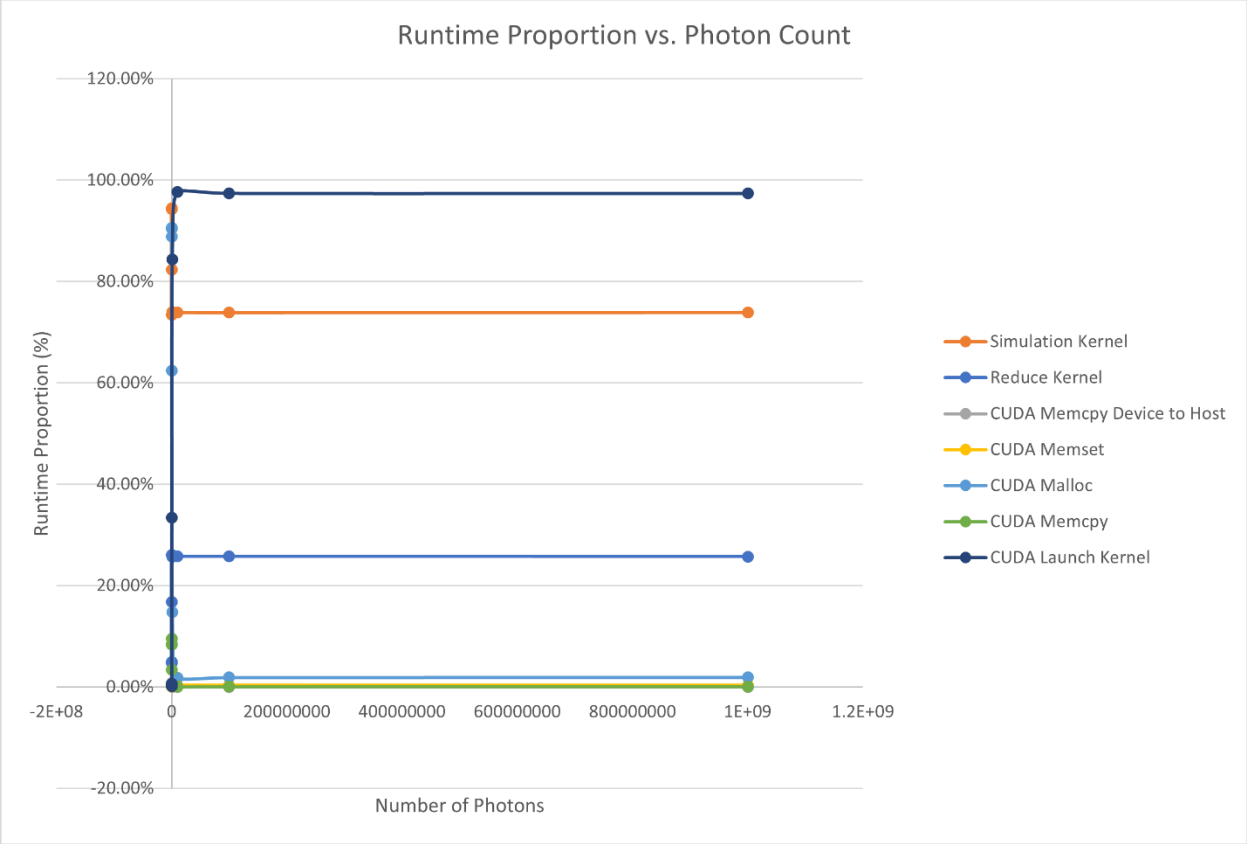

Figure 3.2: A plot of runtime proportion vs. photon count for the main functions called in our GPU implementation. Note that these proportions do not sum to 100%, as the profiling tool used disambiguates between kernel and CUDA API calls. However, a comparison of all of them

together is still worthwhile for analysis. Note that the CUDA Memcpy Device to Host profiling data is difficult to see due to the small proportion of runtime it takes.

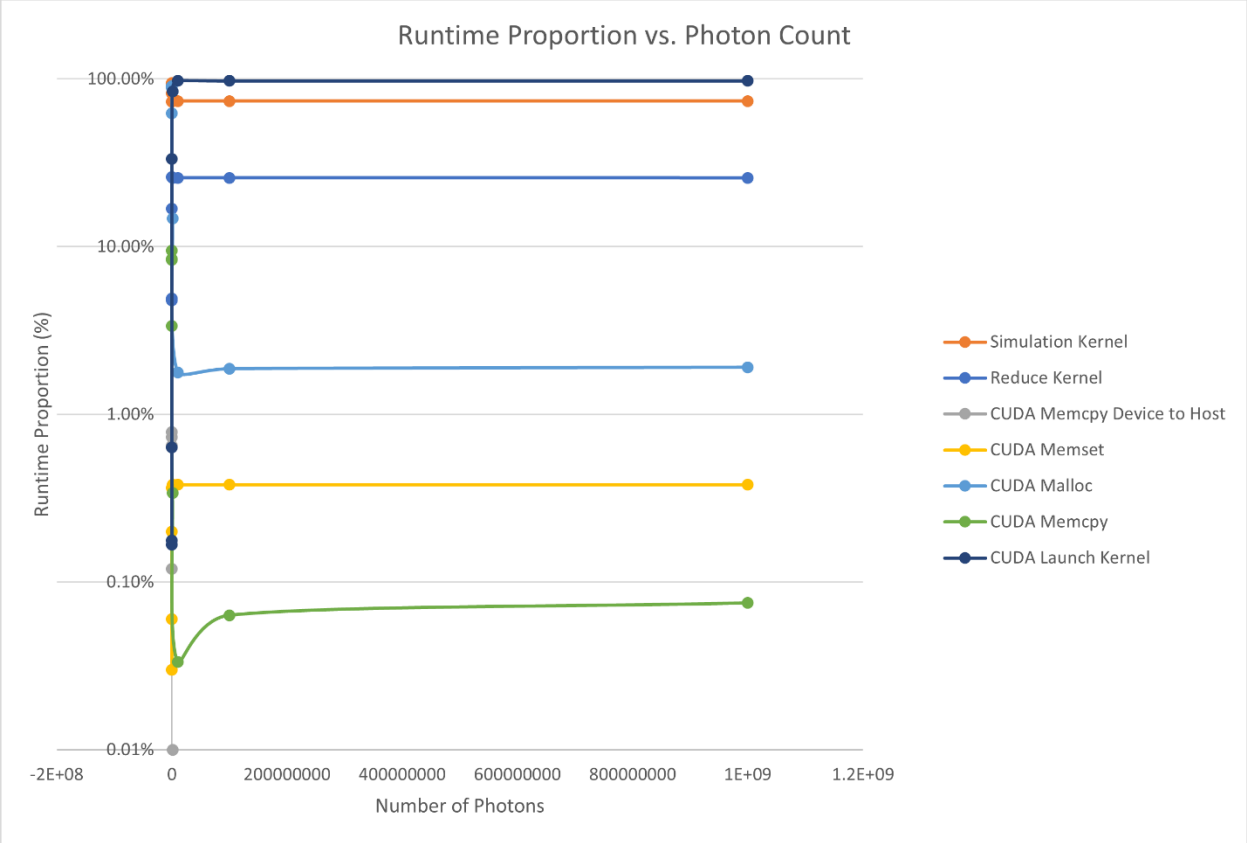

Figure 3.3: The same data in Figure 3.2 but plotted on a logarithmic scale.

In profiling the Parallel GPU implementation, we see some exciting trends. Looking at the high-level view of our code, we have two kernels. Looking at Figures 3.1 and 3.2, we can see that the percentage of time spent in the simulation kernel decreases proportionally with the amount of time spent in the reduction kernel. We expected this because we can simulate 10,000,000 in one simulation; however, we can only reduce 6,144 photons in one kernel.

Another thrilling pattern we observed is how the primarily called API transitioned from CUDA Malloc to CUDA Launch Kernel as the number of simulated photons increased. The tight coupling of this pattern with the number of reductions we perform is directly related to the number of elements we can reduce at one time. As stated above, we can reduce 6,144 elements in one kernel. This limitation means we must launch seven total kernels for 10,000 photons. Two sets of three to reduce the saved values and one for the actual simulation. The number of kernels is likely where we could better optimize our code.

Surprisingly, the amount of time spent copying data from the GPU back to the host is relatively small and only gets more insignificant with the more photons we simulate. This small amount of time is likely due to the optimizations we implemented with the reduction method. A primary goal during the design process was to reduce the number of unnecessary values we copied off the GPU.

Benchmarking

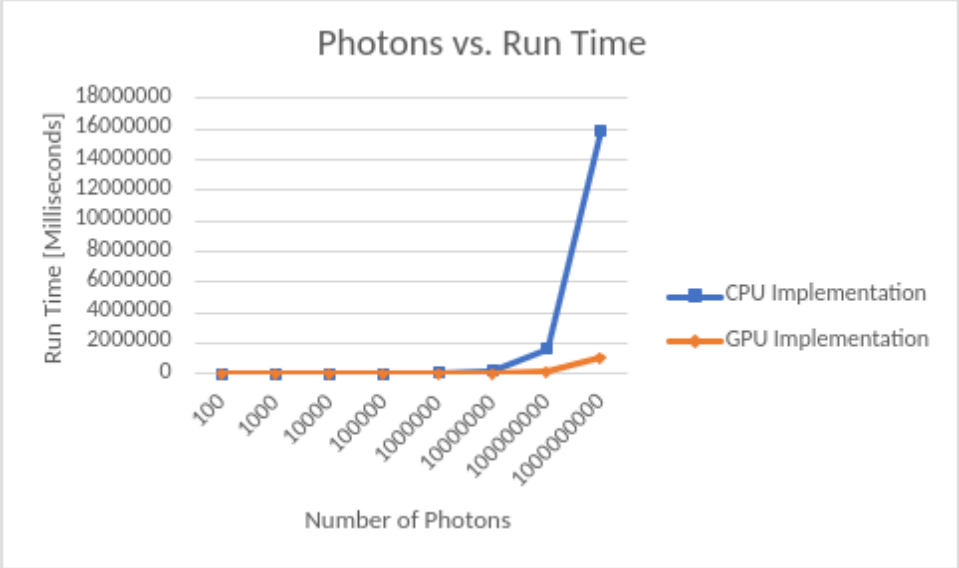

Figure 4.1: A plot of photons vs. runtime for the CPU and GPU Implementation data.

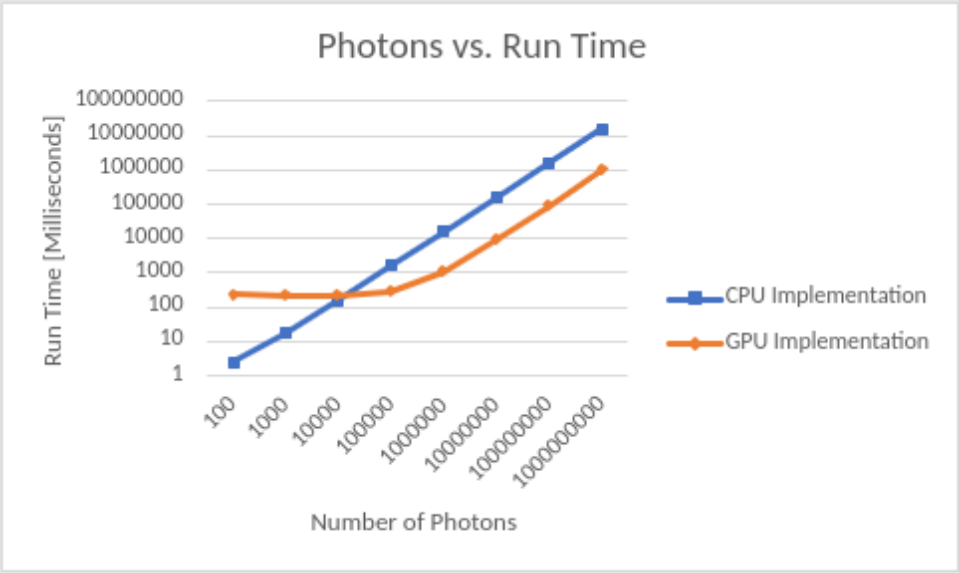

Figure 4.2: The same data plotted in Figure 4.1 but on a logarithmic scale.

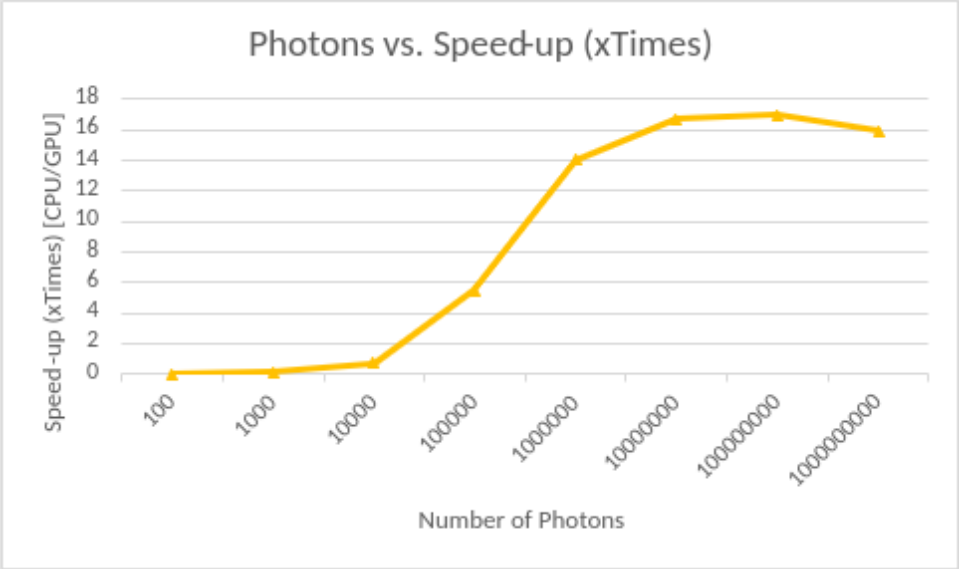

Figure 4.3: A plot of photons vs. times speedup as a ratio of CPU to GPU Implementation runtimes using the same data as Figure 4.1.

The results from our benchmarking show complete success; we achieved a maximum of a 16 times speedup when comparing our parallel GPU implementation to the sequential CPU

implementation. To truly appreciate the speedup, we can look at the generation of 1,000,000,000 photons. Converting milliseconds into hours and minutes, we can see that the average time to simulate is 4 hours and 25 minutes. This result is in stark contrast to the GPU simulation time, which includes the reduction. Converting the GPU simulation time into minutes, we get 16 minutes and 40 seconds. With this speedup, it is feasible for a physicist to run a simulation or two over lunch instead of taking over half an eight-hour workday.

Under 10,000 photons, the CPU implementation is faster. We were expecting this result due to the nature of GPU implementations. There is an overhead of starting kernels and coping memory, which is absorbed with more iterations. The strength of a GPU is the number of operations it can do simultaneously, not how fast it performs a single operation. Looking at Figure L (the logarithmic graph), we can easily follow this trend. For simulations with under 10,000 photons, the GPU implementation is around 250 milliseconds per run. These 250 milliseconds represent the cost overhead of using the GPU. During this time, the CPU runtime increases proportionately to the number of simulated photons.

We collected all results using MSOE's supercomputer: ROSIE. ROSIE is an NVIDIA-powered cluster that we utilize for our learning at MSOE. ROSIE has 1,000 CPU cores, 10 TB RAM, 100 TB high-speed NVMe SSD storage, 400 TB long-term SSD storage, and 100+ GPUs with more than 2 TB total GPU Memory (Daroach et al., 2022). In our tests, we utilized one core from an Intel(R) Xeon(R) Gold 6240 CPU running at 2.60GHz and one NVIDIA Tesla T4.

Conclusion

Overall, this project has provided us with great insight into the industry applications of GPU acceleration that we hope will support the work of our university. In conducting this project,

we learned more about biomedical optics and the experimental setup required to study it, alongside the expanding our CUDA knowledge and algorithm development. The best part of working on these projects is getting exposure to things that would never be taught in class, let alone outside of our majors entirely. While it may not be the coolest or easiest project to show off, we still feel proud of our work, and we hope that it will be continued in the future.

To that end, there is plenty of future work on our project, the most notable of which being profiling with a more powerful GPU. Unfortunately, we could not run this project on an NVIDIA DGX in ROSIE due to time constraints. This class of GPU tends to perform better than the Tesla T4 we tested on, so some additional performance could still be gained. Beyond profiling on another GPU, more in-depth profilers could dig into our kernel and tell us where further optimizations are needed.

In addition to profiling, we could further optimize our code, although we are still quite happy with the performance we observed overall. In this project, we did not focus on memory coalescing, specifically with the heat parameter we are collecting. There is most likely a more optimal way to store these data points that we did not explore at the time. In addition, we saw that the number of kernels we call increases rapidly as we increase the number of simulated photons. We could reduce the total number of kernels we create by doing more work in each kernel. This optimization is likely where we would see the most improvement, as we found in the Profiling subsection.

Works Cited

Daroach, G., Riley, D., Yoder, J., & Taylor, C. (2022, January 7). *Rosie User Guide*. Msoe.dev; Milwaukee School of Engineering. https://msoe.dev/#/

Fluorescence Molecular Tomography: Principles and Potential for Pharmaceutical Research - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Light-propagation-in-tissue-a-Highly-absorbing-medium-where-the-mean-free-path-l-sc_fig1_259206428 [accessed 2 Nov, 2022]

Prahl, S. (2018). *Monte Carlo Light Scattering Programs*. Omlc.org. https://omlc.org/software/mc/