Felix Söderman (felsod@kth.se)
Philip Sjösvärd (sjosv@kth.se)

Optimizing "CUDA-Raytracer" by xkevio

Formalities

Expected grade: A
Original project: `https://github.com/xkevio/CUDA-Raytracer`
Final code version: `https://github.com/felixcool200/CUDA-Raytracer-Optimized`
All separate optimization versions: `https://github.com/PSeasword/CUDAOptimizedRayTracer`

Individual contributions: Both students worked on 90% of the project together (including writing the report), while communicating verbally. The last 10% included scheduling conflicts where the students worked individually on the code. Both provided ideas for potential optimizations, and each wrote about 50% of the code and report.

Introduction

When creating realistic-looking video games and 3D renderings, one of the most challenging tasks is addressing lighting, reflections, and shadows. This complexity arises because they heavily depend on the environment and necessitate numerous calculations to determine how light interacts with its surroundings. Ray tracing is a technique used to perform these calculations. In ray tracing, a ray is cast for each pixel to determine its collisions and subsequent behavior, such as reflections and shadows. Generating realistic images requires a multitude of rays. For instance, a full HD image comprises 1920x1080 pixels, resulting in $2,073,600$ pixels, each requiring a ray (excluding those for reflections and shadows).

Due to the large number of rays, the resulting computation can, thus, be very time-consuming and energy-inefficient. Since these rays are independent of each other, they can be computed in parallel to improve performance and energy efficacy. This parallel processing is the reason why GPUs and other throughput-oriented processors are well-suited for addressing these challenges.

Methodology

The methodology of this project consists of three steps, evaluation, investigation, and implementation. These three steps are repeated after each implemented optimization. The first step is to evaluate the performance of the program to establish a baseline. Doing this requires measuring the time different tasks take during execution. Multiple techniques are used for measuring the execution times. Firstly, CPU timers are implemented to see where the majority of the execution time is. In particular, memory allocation, host-to-device memory transfer, kernel, device-to-host memory transfer, writing the image to file, and freeing the memory. Additional data is also collected using different NVIDIA profiling tools, as discussed later on. Measured time will give direct feedback about performance improvements, while other factors shown by the profiling tools may help alleviate bottlenecks and other issues hampering further performance improvements.

The second step is to investigate if there is any part of the program that underperforms. This can be more of an art than a science, as it often requires clever guesses to figure out the underlying cause of performance problems. The third and final step is to implement a patch for the given issue found in the previous step, which might require multiple attempts before realizing that there is no improvement to be had. Thereby backtracking the changes and investigating other potential underperforming sections.

Experimental Setup

To be able to test optimizations, tests will be run both in Google Colab, on an NVIDIA Tesla T4 (Turing architecture), using CUDA 11.8, and on an NVIDIA GTX 1080 Ti (Pascal architecture), using CUDA 12.2, to make sure that the results between optimizations are consistent across models and architectures.

For compilation, the `-arch=sm_61` flag will be used in combination with `nvcc` for both the T4 and the 1080 Ti. For profiling purposes, used tools include `nsight` and `nvprof`, where subsequent visualization is performed using `nvvp` to gather a greater overview of the program's execution. Due to version compatibility issues with `nsight` on the GTX 1080 Ti, it will only be used when profiling the Tesla T4. In addition to the CUDA software tools, a simple C++ implementation of a CPU timer will measure the total time for CUDA operations, as well as CPU-bound parts of the program. The CPU timer is implemented according to Figure 2.

The rendered scene used that will be used as a benchmark will consist of an 8192x8192-pixel image, totaling $67,108,864$ pixels with 19 spheres placed on three rows at different distances from the camera. The render of the benchmark scene can be seen in Figure 1.

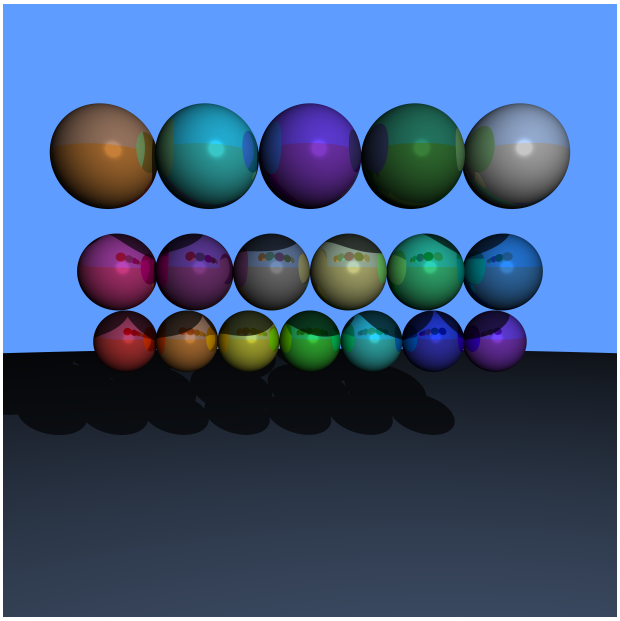

Figure 1: *Rendered image of the scene that will be used as a benchmark.*

```
auto start_CPU = std::chrono::high_resolution_clock::now();

void start_CPU_timer(){
  start_CPU = std::chrono::high_resolution_clock::now();
}

long stop_CPU_timer(const char* info){
  auto elapsed = std::chrono::high_resolution_clock::now() - start_CPU;
  long microseconds = std::chrono::duration_cast<std::chrono::microseconds>(
     elapsed).count();
  std::cout << microseconds << " microseconds\t\t" << info << std::endl;
  return microseconds;
}

//Example usage
start_CPU_timer();
start_CPU("What was timed");
```

Figure 2: *Implementation of CPU timer.*

(a) GTX 1080 Ti. **(b)** Tesla T4.

Figure 3: *Distribution of execution time for each different code version using the average of 100 consecutive runs.*

Table 1: *Execution times for GTX 1080 Ti for different code versions, using the average of 100 consecutive runs.*

	1 Base	2 Static	3 Threads	4 Float	5 Pinned	6 Transfer	7 Shared
Host mem. alloc. [µs]	426526	423572	407016	11	225815	222079	221802
Device mem. alloc. [µs]	774	783	752	753	694	670	678
HtoD mem. transfer [µs]	76352	75172	70411	191236	62092	75	78
CUDA kernel [µs]	1494161	127274	124022	124616	123722	123214	42475
DtoH mem. transfer [µs]	74155	73191	68224	401227	61316	61281	61292
Device mem. free [µs]	2817	2812	2790	2789	2796	2791	2786
Host mem. free [µs]	38709	39534	38853	38874	112489	111608	111466
Total time [µs]	2113498	742341	712070	759510	588926	521721	440579

Results

The performed optimizations resulted in several distinct versions of the code, each building on top of all previous versions. The source code for all of these versions and the `run.sh` script used to automate outputs can be found in [1]. In Figure 3, one can see the average measured execution times of 100 runs by the CPU timers in each iteration of the source code, with the exact values being shown in Table 1 and 2.

The remaining part of this section describes each iterative version of the code in detail, including all changes and the reasoning behind them. Each of these code versions has generated an identical final image to that seen in Figure 1.

Base Code

The final code used for this version can be found in `src1Base` at [1].

For this project, the used baseline (`src1Base`) differs from the code found in [2]. Firstly, since no `Makefile` was provided in the project, one needed to be created. Creating the `Makefile` was not a trivial task, since the program tried to define its own proprietary 3-dimensional vector class as an independent `.cu` file. This forced the use of the `-rdc=true` flag during compilation with `nvcc`, a flag that allows the program to call other code segments outside its compilation unit.

Table 2: *Execution times for T4 for different code versions, using the average of 100 consecutive runs.*

	1 Base	2 Static	3 Threads	4 Float	5 Pinned	6 Transfer	7 Shared
Host mem. alloc. [µs]	598827	593871	595473	11	412114	410721	414330
Device mem. alloc. [µs]	76053	69967	68688	69754	1075	1060	1065
HtoD mem. transfer [µs]	174063	174396	173066	268473	70485	75	75
CUDA kernel [µs]	2524580	44817	58656	66183	59146	59018	59190
DtoH mem. transfer [µs]	173262	173407	173323	758522	65031	65231	65176
Device mem. free [µs]	1988	1997	1973	1973	1968	1955	1962
Host mem. free [µs]	38667	38955	38722	38219	116071	115547	115952
Total time [µs]	3587442	1097413	1109903	1203139	725893	653611	657753

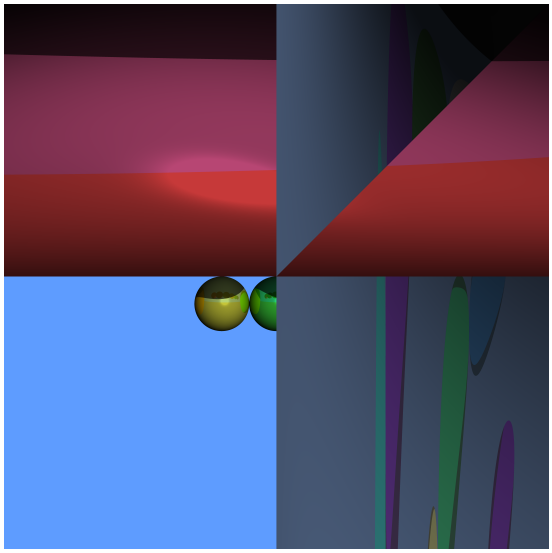

Figure 4: *Issues when using the -rdc=true flag.*

However, enabling the `-rdc=true` flag revealed a curious bug. Specifically, the rendering of the image resulted in visual artifacts, as seen in Figure 4. The issue seemed to be related to a high register usage per thread, as the issue did not seem to occur in later versions where register usage was optimized and, hence, lowered. To solve this issue, the entirety of `Vec3f.cu` was moved into `Vec3f.cuh` so that it may be compiled within the same compilation unit. Thus, the `-rdc=true` flag was no longer needed, thereby solving the issue with visual artifacts.

Other general changes that were made to the code were to adjust overall formatting to improve readability, remove unnecessary use of `auto` obscuring data types, move class and struct definitions in `main.cu` to separate files, and add a significant number of comments to describe the functionality of the code. The rendered scene was also adjusted in its complexity and resolution compared to the original code. Furthermore, the new CPU timers were added around all relevant code sections, replacing a few already existing CPU timers for greater consistency across implementations. Lastly, the reflection of the spheres was corrected to also take the background color into account (color of the sky), as that was originally not done.

Static Analysis

The final code used for this version can be found in `src2Static` at [1].

When running `nsight` on the base code (`src1Base`), it was observed that there was a poor theoretical occupancy of 25% (with an achieved occupancy of approximately 24.95%) on the T4 GPU. This resulted in a warning indicating that the theoretical occupancy was limited due to the required number of registers per thread. To be specific, the code was using a total of 130 registers per thread. Consequently, significant changes had to be made to the code as a whole.

Numerous improvements were made in terms of both performance and register usage. For instance, adding `break` to reduce unnecessary computations, removed redundant code, and optimizing mathematical computations. Additionally, and perhaps most importantly, reducing an array of `float` values containing potential intersections between each thread's ray and each sphere in the scene to a single `float`. The full list of changes can be seen in [1].

The result of these changes was 54 registers per thread instead of 130, resulting in a theoretical occupancy of 100% and an achieved occupancy of 89.34%. As seen in Figure 3, there is a noticeable difference in execution time, with the majority of the reduction occurring in the CUDA kernel. The CUDA kernel execution time decreased from $1,494,161$ µs to $127,274$ µs on the GTX 1080 Ti (a 91.5% decrease), while on the T4, the time dropped from $2,524,580$ µs down to $44,817$ µs (a 98.2% decrease).

An improvement that is not shown in Figure 3 is the reduction in time when writing the image to file. Originally, the pixels were first written to a buffer before being written to the file. Removing this intermediate step resulted in a substantial reduction of the time required, amounting to a 46.1% decrease on the T4 GPU and a 41.8% decrease on the GTX 1080 Ti. No additional changes were made to the write-to-file function, and because it accounted for a significant portion of the total execution time, it was excluded from all other optimization measurements to ensure the results remained consistent.

Threads per Block

The final code used for this version can be found in `src3Threads` at [1].

In this version of the code, only one small change was made. Specifically, the number of threads per block was changed from 16×16 (256) to 32×32 (1024) to see if any performance gains could be achieved. While no noticeable changes in execution times could be observed, `nsight` shows that the achieved occupancy increased from 89.3% to 96.0% on the T4, removing the warning `nsight` had regarding a somewhat low achieved occupancy. Additionally, no performance gains were observed when using 8×8 (64) threads per block.

Replace Vec3f With float3

The final code used for this version can be found in `src4Float` at [1].

When attempting to understand the code, it became apparent that the data structure `Vec3f` had the same features as `float3` from CUDA. Consequently, it was decided to convert the code from `Vec3f` to `float3` in accordance with the lectures, as it can increase the bandwidth [3]. Furthermore, working with `float3` appeared more straightforward than the proprietary vector class `Vec3f`. These changes involved replacing the use of `std::sqrt` and the proprietary `f_max` function (which calculates the maximum of two floats) with the CUDA functions `sqrtf` and `fmaxf`, respectively.

The execution times for this version of the code, as depicted in Figure 3, exhibited some unexpected behavior. Although the overall execution time was only marginally worse than the previous version (`src3Threads`), the host memory allocation time has become practically non-existent, at the expense of increased time for host-to-device memory transfer and freeing of device memory. Additionally, when investigating the traces using `nvvp`, it was found that the memory transfer speeds were substantially lower than expected. The transfer speeds on the T4 dropped from 10.9 GB/s to 4.3 GB/s for host-to-device and from 11.4 GB/S to 2.0 GB/s from device-to-host.

When examining the underlying causes of this phenomenon, it appeared that the memory allocation did not occur immediately, but rather when it was first used. The resulting undesired behavior seemed to be linked to the utilization of the `new` keyword in C++. When applied to a C++ class objects, the `new` keyword invokes the default constructor for each allocated element, as detailed in [4]. Consequently, when transitioning from `Vec3f` (a C++ object) to `float3` (a C struct), the default constructor is no longer defined, thereby altering the behavior of the `new` keyword.

To be able to utilize the new `float3` implementation effectively, these issues must be addressed to enhance the throughput of the data transfer between the host and device is increased, but also to get a more desired behavior regarding memory allocation. Rather than reverting to the previous version, it was decided to attempt to correct the `float3` version, as potential gains could be had if its behavior was rectified.

Pinned Memory

The final code used for this version can be found in `src5Pinned` at [1].

In an attempt to rectify the issues caused by implementing `float3`, pinned memory was used to try to force the host memory allocation. The attempt proved successful, as the time distribution between the different timed sections became closer to that of using `Vec3f`.

The use of pinned memory as opposed to pageable memory also, as expected, results in greater memory transfer speeds between host and device. More specifically, going from pageable to pinned memory in the `Vec3f` implementation (avoiding the influence of `float3`), resulted in transfer speeds improving from 10.9 GB/s to 13.0 GB/s host-to-device and 11.4 GB/s to 12.9 GB/S device-to-host on the 1080 Ti. These same improved transfer speeds are also carried over to the `float3` implementation. In the end, the decision to use `float3` instead of `Vec3f` did not notably affect performance, but simplified the code somewhat.

Frame Buffer Transfer

The final code used for this version can be found in `src6Transfer` at [1].

An observation made using CPU timers revealed that the host-to-device memory transfer was taking a considerable amount of time. Additionally, when analyzed with `nvvp`, it became evident that the durations of the host-to-device and device-to-host memory transfers were roughly equal. This observation appeared peculiar, considering that only a vector specifying the origin, a lightning source, and the spheres in the scene needed to be transferred to the device, whereas the entire frame buffer needed to be transferred to the host. Consequently, one would expect these two transfers to differentiate significantly in their execution times.

Upon inspecting the code, it became evident that the frame buffer, which was intended to store the resulting pixels of the kernel, was being sent to the device despite being empty and uninitialized. This represented a tremendous waste of resources, given that each value in the array would be overwritten within the kernel. After removing the `cudaMemcpy` operation for the frame buffer, the time required for the host-to-device memory transfer decreased from $70,485$ µs to 75 µs on the T4 and from $62,092$ µs to 75 µs on the 1080 Ti.

Shared Memory

The final code used for this version can be found in `src7Shared` at [1].

An additional effort to enhance the kernel's execution time involved the implementation of shared memory for the spheres on the device. In essence, once these spheres have been transferred to the device's global memory, one thread per block facilitates the transfer of a single sphere to the shared memory (enabling concurrent transfers to shared memory within each block). The utilization of shared memory results in quicker access times for the spheres, which are frequently accessed by the kernel during computations.

As depicted in Figure 3, the utilization of shared memory (`gpu7Shared.out`) does not yield a significant improvement in execution time on the T4. Nevertheless, it does lead to a substantial reduction in kernel execution time on the 1080 Ti, decreasing it from $123,214$ µs to $42,475$ µs, representing an improvement of 66.5%.

Final Results

After implementing all optimizations, the following results were obtained by averaging the data from 100 runs. The GTX 1080 Ti exhibited a significant reduction in execution time (excluding the time spent writing the frame buffer to disk). Specifically, the execution time decreased from $2,113,498$ µs (2.11 s) to $440,579$ µs (0.44 s), representing a 79.2% reduction in computation time, which is equivalent to 20.8% of the original time.

Similar results were observed when running the program on Google Colab using an NVIDIA T4 GPU. The execution time (excluding the time for writing the frame buffer to disk) decreased from $3,587,442$ µs (3.6 s) to $657,753$ µs (0.66 s), marking an improvement of 81.7%.

Discussion

Differences Between GTX 1080 Ti and T4

The optimizations generally affected both GPUs equally, resulting in final total execution time improvements of around 80%. However, when examining Figure 3, one can observe that the allocation of execution time across tasks differs. The 1080 Ti, percentage-wise, has significantly greater kernel execution times, while the T4 exhibits longer times for device memory allocation. Additionally, the 1080 Ti benefited noticeably from using shared memory in terms of kernel execution time, while there was no apparent change for the T4.

Some optimizations were harder to decide whether they worked or not. One such aspect was the number of threads per block. In Google Colab, increasing the number of threads per block had little effect on the last three versions (`src5Pinned`, `src6Transfer`, and `src7Shared`). However, on the 1080 Ti, overall times improved, but kernel times increased slightly (which should be the only thing affected). Due to the minor impact of these changes, it could be attributed to run-to-run variance, or they might be too insignificant to make a difference in any case.

Attempted Optimizations

Various potential optimizations were tested, with some that did not meaningfully alter the results and others that hurt them. None of these potential optimizations were included in the code versions used to collect the final results. Nevertheless, some of them may be found in alternative code versions in [1].

Reduce Control Divergence

When analyzing the `has_intersection` method within the `Ray` class of the original code, a method called by the kernel to check if a ray intersects with any of the spheres in the scene, it was found to include many if-statements. It was thought that control divergence could potentially be reduced if these five if-statements were replaced by a single large boolean expression. The change code can be seen in Figure 5, as well as in `src7Divergence` in [1] (alternative code version path to the one used in the results).

When analyzing the `has_intersection` method within the `Ray` class of the original code, which is a method invoked by the kernel to check if a ray intersects with any of the spheres in the scene, it was observed that it contained numerous if-statements. An attempt to reduce control divergence was made by replacing the five if-statements with a single extensive boolean expression. The modified code can be viewed in Figure 5 and also in the `src7Divergence` directory in the repository [1]. The `src7Divergence` version represents an alternative path of optimizations to `src7Shared` used in the final results.

```
if (d == 0) {return t0;}
if (t0 < 0 && t1 < 0) {return -1;}
if (t0 > 0 && t1 < 0) {return t0;}
if (t0 < 0 && t1 > 0) {return t1;}
return t0 < t1 ? t0 : t1;

//Replaced by
const bool t0_is_pos = t0 > 0;
const bool t1_is_pos = t1 > 0;
return ((!t0_is_pos * !t1_is_pos * -1) + (t0_is_pos * !t1_is_pos * t0) + (!
   t0_is_pos * t1_is_pos * t1) + (t0_is_pos * t1_is_pos * fminf(t0, t1)));
```

Figure 5: *Attempted control divergence optimization.*

The attempt to reduce control divergence did not yield any noteworthy changes in either execution times or in `nsight`, essentially only slightly reducing register usage. The lack of change is likely due to the rays of nearby

pixels hitting similar targets and, thus, not resulting in any major control divergence. Consequently, the optimization was removed as it added unnecessary code complexity.

Unified Memory

Another potential optimization was to see if unified memory could decrease the time needed for memory transfer or simplify code management. Nevertheless, it yielded no performance improvements, and the code's complexity increased, necessitating prefetching to maintain comparable performance. Consequently, this implementation was never fully realized.

Pass by Value

An attempt to optimize the time spent on accessing global memory involved moving the `light` and `origin` objects (consisting of 9 `float` values, equivalent to 34 bytes, and 3 `floats` values, totaling 12 bytes, respectively) from global memory to local memory by sending the data as a copy during the kernel launch. However, this adjustment did not yield any noticeable improvements. It is possible that the impact of such changes is too minimal, or any potential gains from the decreased access latency are nullified by the additional data copies required for each thread.

Multiple Rays per Thread

In previous projects involving the creation of a simple vector addition kernel, it was observed that performance improvements occurred when each thread was tasked with performing calculations for multiple indices. This performance boost appeared to result from the increased efficiency of having each thread handle a greater number of computations, as opposed to creating more threads. Consequently, a comparable approach was applied to the ray tracer, where each thread was assigned the task of iteratively tracing multiple rays (specifically, testing 4, 9, and 16 rays per thread). However, this change yielded only a performance degradation, which was most likely a consequence of the parallelization of spawning more threads outweighing the cost of creating them.

Future Work

In addition to the optimizations mentioned earlier, there are other potential methods to further improve performance. One approach involves utilizing streams to parallelize data transfer from the kernel. This would enable an overlap between kernel execution and data transfer from the device, potentially leading to noticeable performance gains. Unfortunately, given the inherent complexity of rewriting the kernel to support a subset of the 2-dimensional frame buffer, along with time constraints, there was no further investigation made.

Conclusion

In conclusion, the final version of the program was able to reduce the total execution time (excluding writing the rendered image to a file) by 79.2% on the 1080 Ti and 81.7% on the T4 in Google Colab. Furthermore, the time required to write the rendered image to a file was reduced by approximately 50% on both platforms.

References

[1] (2023, Oct.) CUDAOptimizedRayTracer. GitHub. [Online]. Available: https://github.com/PSeasword/CUDAOptimizedRayTracer

[2] (2023, Jan.) CUDA-Raytracer. GitHub. [Online]. Available: https://github.com/xkevio/CUDA-Raytracer

[3] S. Markidis. (2023, Oct.) Lecture: CUDA Essentials III - Kernel execution, indexing and vector types. DD2360 Transcription of the video lecture on Canvas. [Online]. Available: https://canvas.kth.se/courses/42842/pages/lecture-cuda-essentials-iii-kernel-execution-indexing-and-vector-types?module_item_id=735386

[4] (2023, Jun.) new expression. cppreference. [Online]. Available: https://en.cppreference.com/w/cpp/language/new