Final Project Report

Jiyu Hu (jiyuh)

Zhaohong Lyu (zhaohonl)

1 Summary

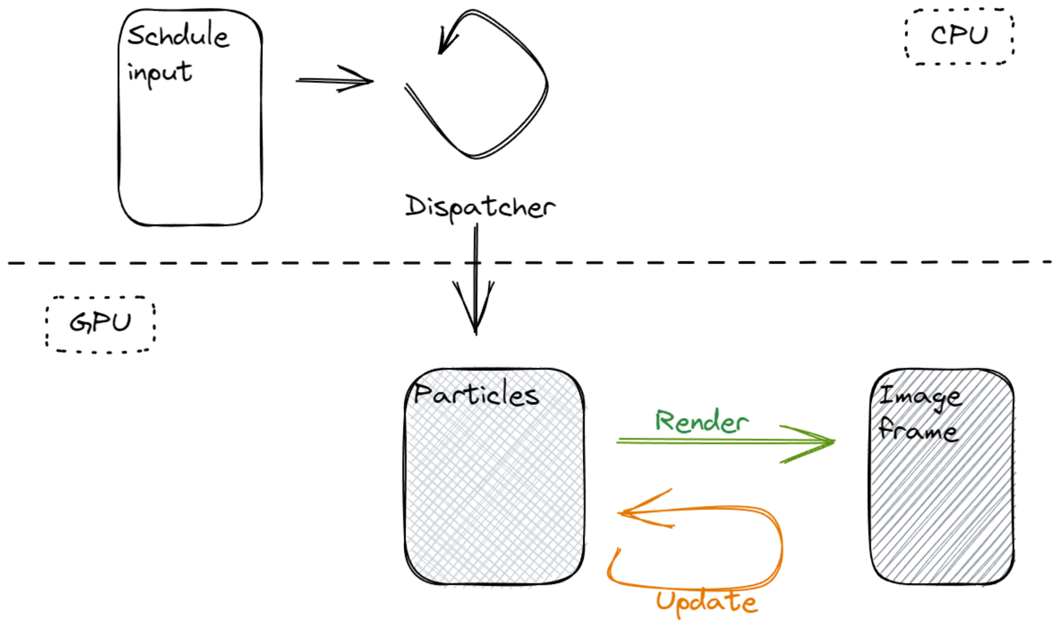
We developed a CUDA renderer for fireworks that runs on the GPU in real-time, supporting various patterns and colors. We conducted a thorough analysis to identify performance bottlenecks and optimized parallel efficiency and memory usage to improve overall performance.

2 Background

2.1 Repository Structure

- “input/”: pre-set schedules of fireworks
- “main.cpp”: entry point of renderer, responsible for parsing the input, initializing OpenGL and the benchmark tool, and starting rendering
- “kernel.*”: CUDA logic of rendering and updating particles
- “helper*” common utility functions in CUDA
- “pattern.cu_inl”: template of patterns for fireworks
- “color.cu_inl”: template of colors for fireworks
- “benchmark.cpp”: entry point of benchmark tool without X11 forwarding
- “seq.*”: sequential implementation

2.2 Workflow

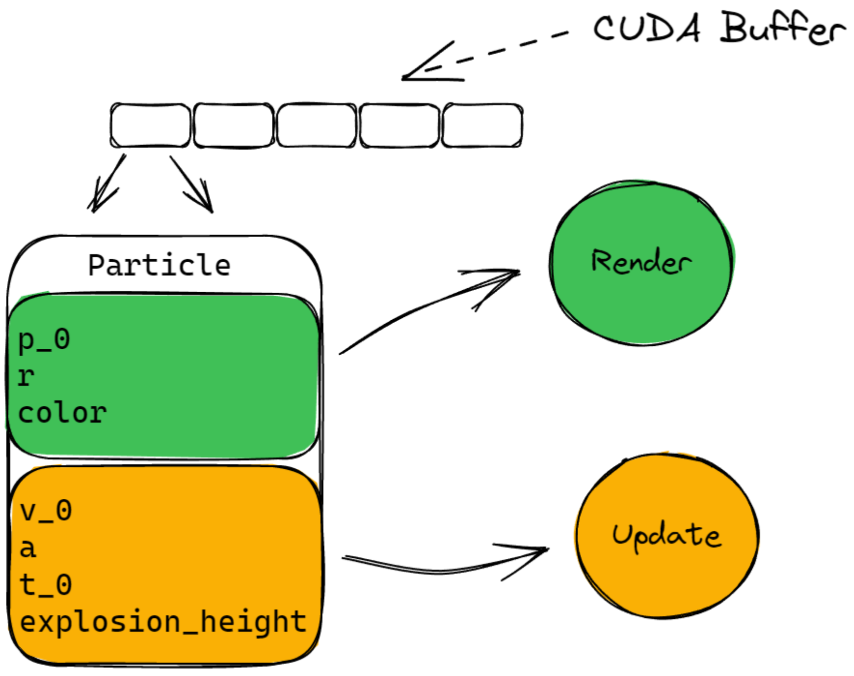


This Figure demonstrates a high-level workflow of our implementation. Let's dive into some key components that follow this design.

2.3 Key Data Structure

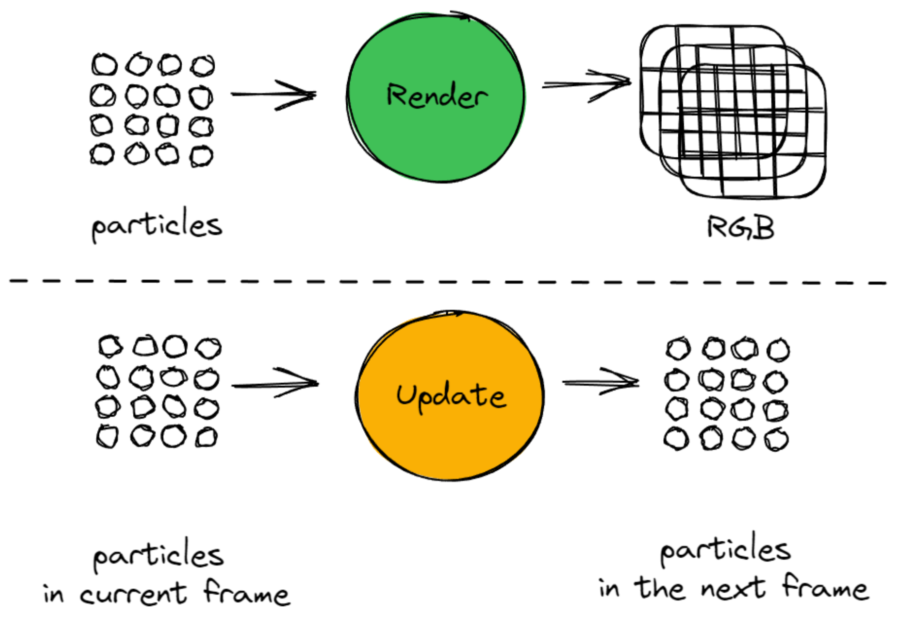
C/C++
```
// A basic unit of a firework
struct particle {
  // current coordinates of the particle in the x and y
axis
  float2 p_0;
  // current velocity on the x and y axis of the particle
  float2 v_0;
  // gravity vector for physical trace simulation
  float2 a;
  // radius of the particle
  float r;
  // timestamp of when the particle is scheduled to fire
  float t_0;
  // height where the particle bursts into an explotion
  float explosion_height;
  // id of firework explosion patterns
  unsigned char color;
};
```

2.4 Key Operations



This Figure shows how our main algorithm interacts with particles.

2.5 Algorithm



The Figure above describes two algorithms, render and update, which function as black boxes that compute outputs based on corresponding inputs.

2.6 Computation and Parallelization

Rendering is computationally expensive because there are many in-process particles and the size of the frame is fairly large. We can either process particles in parallel, or pixels in parallel. Updating is also computationally expensive. This can be beneficial from parallelization as well. Because when updating particles, each particle is independent of each other. Updates can be performed in parallel, leading to significant performance improvements.

2.7 Workload

The workload can be mainly divided into two parts, both of them are data-parallel, but in a different way. The rendering is parallel in that each pixel needs all the particles to render, but the overhead of iterating through too many particles can be burdensome. When updating the particles, each particle is mostly independent of other particles. The only dependency is when moving the particles from schedule to the actual display buffer, when the particle explodes and when the firework finishes display and should be evicted. In the particle update scenario, the workload can be parallelized over the particles with the help of barriers and reduction algorithms.

3 Approach

3.1 Technologies

Our experience with building a simple CUDA renderer in assignment 2 showed us the power of parallel CUDA rendering, and we wanted to explore its full potential. Following a similar approach, we decided to build and run our fireworks CUDA renderer on GHC machines equipped with NVIDIA GeForce RTX 2080 GPUs. We used CUDA C++ for parallelization and OpenGL for real-time display of the rendered output. To visualize the fireworks effect, we relied on X11 forwarding to interact with our graphical application.

3.2 Mapping

As mentioned above, the kernel is mainly divided into two parts: rendering and particle updates. This divide of workload makes the mapping of CUDA threads to workloads extremely easy because the interaction between particles and the rendering process is entirely divided into two parts. For the render part, threads are mapped to actual pixels, but we also used batched process and divided the entire canvas into subsections for faster rendering. For the particle update part, the threads are mapped to particles. Think about the case that a single upshooting particle exploding into multiple particles, there seems heavy interaction between particles, but in reality they can be coordinated on time without any actual synchronization happening.

3.3 Serial Version

We have also implemented a sequential version of the same program using C++ on a single CPU core. The effect is far from usable as we can imagine the serial version needs to iterate through all the pixels with all the in display particles. We then switched to use CUDA to parallelize our code to be practically usable for real-time rendering.

3.4 Optimization

To optimize OpenGL rendering, we bound the CUDA buffer as a texture, which eliminated the need for unnecessary data movement between the GPU and CPU. This meant that we no longer had to copy frame data around, resulting in improved performance.

We applied various optimizations to achieve our goal of building a realistic and efficient fireworks renderer. The first optimization is to transfer from sequential implementation to parallel rendering in CUDA.

At this point, the particles are already able to display, but at a relatively low FPS. Then we looked into the bottlenecks in the program and pinpointed that the particle updates takes a significant amount of time to execute. The reason is because of a large number of random number generation in CUDA kernel. We switched to an optimized version of generating pseudorandom numbers in CUDA by not passing through random number states by incrementing the seed per generation of random number. This optimization already makes our display at around 30 FPS when the particles are not dense at the same time.

Then we applied similar tiling approach as assignment2 to decrease the number of particles each thread needs to iterate over for dense display, but the speed up was not as good as that of assignment2. During the implementation of tiling, we also optimized the memory usage of the program by allowing memory reuse with a circular allocation of memories. By circular allocation, we use two pointer to indicate the occupied space in the buffer. This is especially efficient in our scenario because the fireworks are put into the buffer in an sequential order, and will be evicted in a roughly sequential order, so using only two pointers to keep track of buffer usage can save the space and time overhead of a asynchronous lookup table while not giving too much false positives/negatives. We implemented two eviction policies: conservative and aggressive. The conservative policy will stop eviction once the buffer tail hits an in use slot, while the aggressive policy evicts all slots before the last empty slot. While the aggressive policy can evict fireworks that are still in display, in reality the effect is acceptable because the fireworks anyways have an fade-out effect, so the ones being evicted are usually barely seeable. With tiling, we need to in addition implement a reduction algorithm to determine the head and tail of the buffer since threads in each tile now do not have the information of all particles.

We have later realized that the X11 forwarding has become the bottleneck in rendering the fireworks at a higher FPS. We first had this suspicion when we observed that the framerate of the application differed a lot on different client hardwares (nearly 100 FPS on desktop against barely 30 FPS on laptop). After break-down timing, we verified that the openGL buffer swap is slowing the entire process down, when combined with X11 forwarding. The frame is transferred through network and moved to the client machines' display memory. We learned about the direct/indirect rendering of OpenGL on X11 forwarding, but the GHC machine already uses the optimized direct forwarding (the OpenGL objects are generated on the server machine and the image is directly transferred to clients instead of the objects). We thought of some ways to speed up this process such as compressing, transferring and decompressing but abandoned this line of exploration as it deviates from the goal of parallel optimization.

To alleviate the problem of fluctuation in framerates. We hooked the virtual time directly to machine clock time. This means that the virtual time increment between each frame is no longer a constant but changes as the frame rate changes. Instead of updating the position/velocity of the particles at each frame, only the initial values are stored and the timely information are calculated on-the-fly given a function over time. This change of timing framework also benefits a lot in later design of extensible firework pattern and color framework. In our implementation, user can easily design new type of fireworks on their own because the pattern and colors are all functions to time. Therefore, it is extremely easy to implement fireworks of changing colors and light effect as well as different explosion shapes.

3.5 Existing Codes

We referenced the code base from assignment 2 [1] and also this blog to learn about how to bind OpenGL buffer directly to CUDA device memory: [2]

4 Results

4.1 Experimental Setup

We tracked the frame rate (FPS) as a measure of performance during the execution of the program.

To help create various benchmark inputs, we implemented a random fireworks schedule generator. One common setup we used was to orchestrate 256 fireworks within 150 seconds with 256 particles in each firework (firework time density = 437 particles/sec).

During the experiment setup, we observed a limitation with remote interaction, specifically X11 forwarding. This method relied heavily on network bandwidth as it involved significant data transfer. For the following experiments, we had some tests with X11 forwarding enabled, and some without X11 forwarding. Since remote graphical application interaction was not our focus of our project, we didn't put much effort into resolving the degraded performance when using X11 forwarding. There are potential solutions to this issue, such as utilizing other software like X2Go with a modified NX 3 protocol. We leave further investigation into these solutions for future research.

4.2 Benchmark Results

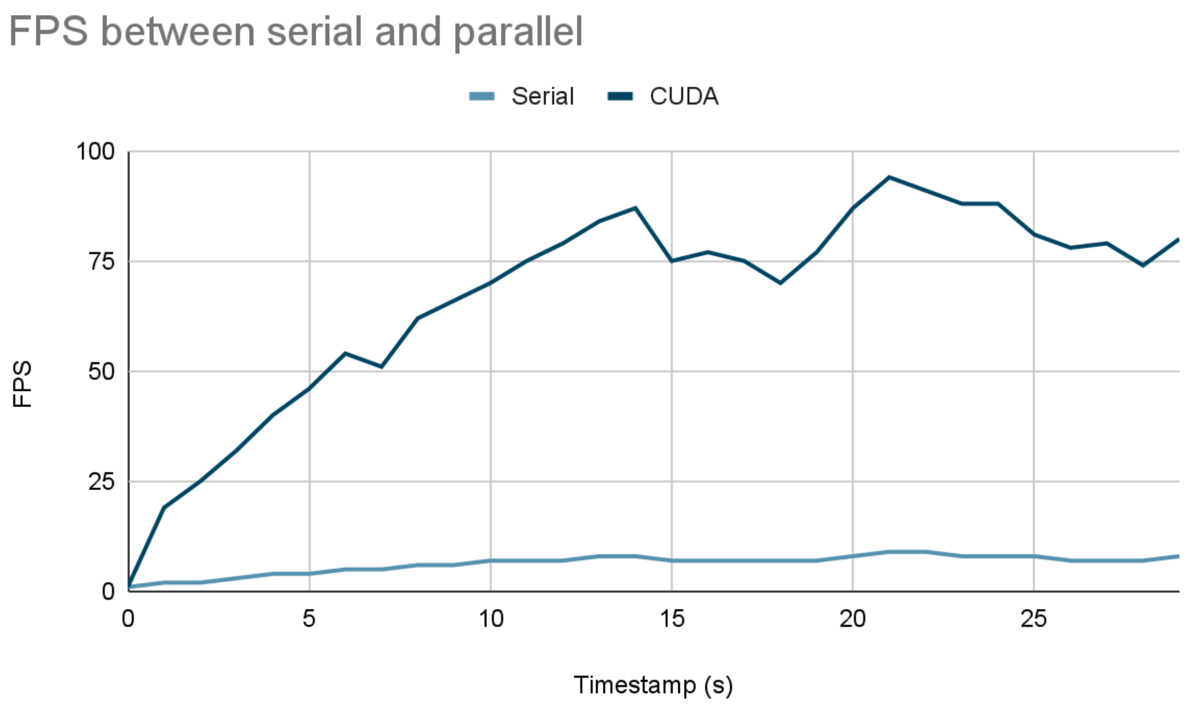
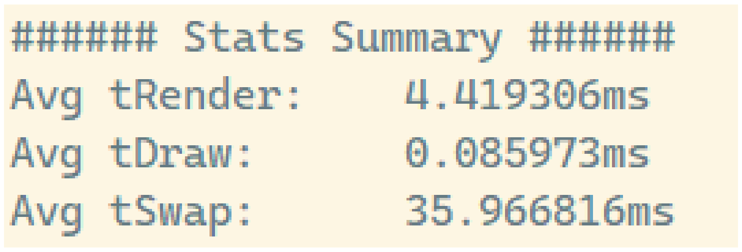


Figure 1: Real-time rendering serial vs parallel (window size 800*400)

From Figure 1, we can observe that, after the framerate stabilizes, the parallel version achieves a framerate as high as 90 FPS, while the sequential version peaks at around 10 FPS, about 9~10x speedup compared to the sequential version. Also note that the bottleneck of the parallel program is the X11 forwarding while the bottleneck of the sequential program is the rendering itself.



```
###### Stats Summary ######
Avg tRender:     4.419306ms
Avg tDraw:       0.085973ms
Avg tSwap:       35.966816ms
```

Figure 2: Real-time rendering timing break down (window size 800*400)

Figure 2 shows the time break down of the display function in OpenGL, we can observe that the actual rendering time by the CUDA kernel is insignificant compared to the buffer swapping time, which consists of time to X11 forward the data. Therefore, in the following benchmarks, we are evaluating the kernel as a standalone function to eliminate the influence of X11 forwarding overhead – afterall our rendering program does not have to run through X11 forwarding.

4.3 Experiment with Workloads

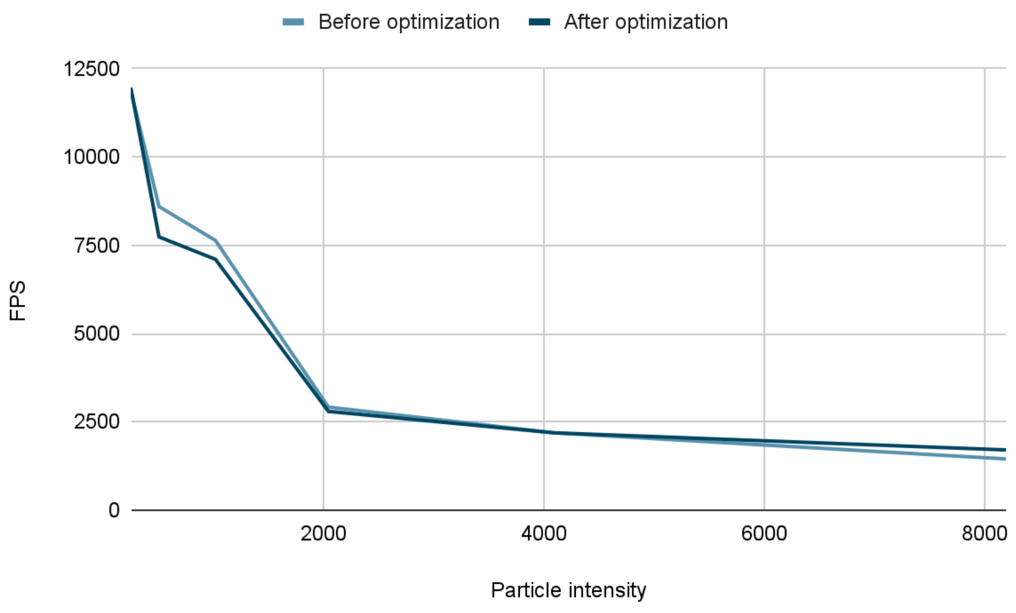


Figure 3: Benchmark of standalone kernel

We benchmark the normal implementation and the optimized one with tiling against different particle intensities (particles/sec). The result is not satisfactory as we see negligible speedup due to the optimization. We still see some potential in the optimization as we can see from Figure 3 that the optimized kernel starts to outperform the unoptimized one on larger particle densities.

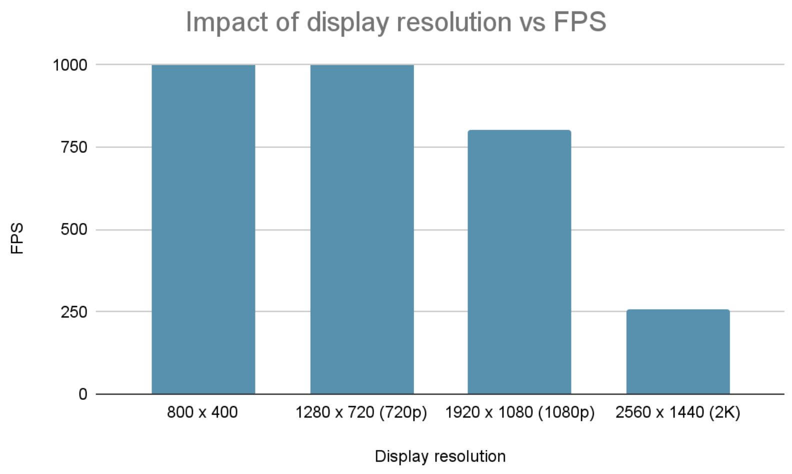


Figure 4: Benchmark of standalone kernel

We run our implementation in different display resolutions as shown in Figure 4. Note that the FPS values of resolutions of 800x400 and 1280x720 were much higher than 1000. For the purpose of better plot visualization, we truncated the higher end. In rendering, a difference between 1000 FPS and 2000 FPS is indistinguishable for the human eye.

Increasing the resolution leads to a decrease in FPS, as there are more pixels to render. Overall, we are satisfied with the performance of our implementation, even in 2K resolution.

4.4 Speedup Limitation

Aside from the limitation of X11 forwarding. Our different strategies hit different limitations. The unoptimized kernel is compute intensive because each thread needs to iterate over all the particles. We can observe a clear FPS drop when increasing the particle density. Besides, the only memory access to retrieve the particles is sequential in the kernel.

We attribute failure to speedup of the optimized kernel to two aspects: enormous random memory access and extra reduction for buffer space synchronization. The tiling strategy we adopt is similar to assignment 2 in that the threads in each tile are mapped to particles first to determine existence and then perform rendering on the reduced set of particles that fall into the tile. Our suspicion on random memory access is speculative because we did not have time to perform break-down timing and running Nvidia Visual Profiler for deeper analysis. The basis of the speculation comes from the fact that we failed to fit the particle data into shared memory space because we had more spec to record for each particle so when later rendering the particles, the threads need to randomly access the existing particles from global memory. However, we were able to verify that the extra synchronization of buffer pointers did add around 1.2x runtime overhead to the kernel.

4.5 Conclusion

In conclusion, our choice of CUDA is a reasonable choice for building a fireworks renderer. We exploit the high parallel computing capabilities in GPUs. The benchmark results corroborate our choice. We think overall GPU will perform far more better than CPU on highly parallelizable workloads, especially image rendering, as the synchronization overhead is extremely low. Therefore, CUDA is definitely more suitable than OpenMP/MPI for our project.

5 References

[1] Assignment 2: A Simple CUDA Renderer (15-418/618 Spring 2023)

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OPENGL.html https://github.com/NVIDIA/cuda-samples/tree/26665bf33b2df3f03f52467d954090f8a19d0e71/Samples/0_Introduction/simpleCUDA2GL

[2] CUDA for Engineers: 2D Grids and Interactive Graphics:

https://www.informit.com/articles/article.aspx?p=2455391&seqNum=2

6 Credit

Input generator	Zhaohong
CUDA implementation	Jiyu
CUDA optimization	Jiyu
Benchmark tool	Jiyu
Report draft	Zhaohong
Poster design	Zhaohong

Distribution: 50%-50%

7 Repository

Link to website: https://v1siuol.github.io/fireworks

Link to code: https://github.com/JiyuuuHuuu/15618-project