



RUTGERS
UNIVERSITY

ECE451/566 Final Project Report

**Electrical and Computer Engineering Department
Rutgers University, Piscataway, NJ 08854**

Gravity Simulator

CUDA-BASED SIMULATION AND RAY TRACING RENDERING

Submitted by:
Shizhe Yang , Olisadebe Ojukwu

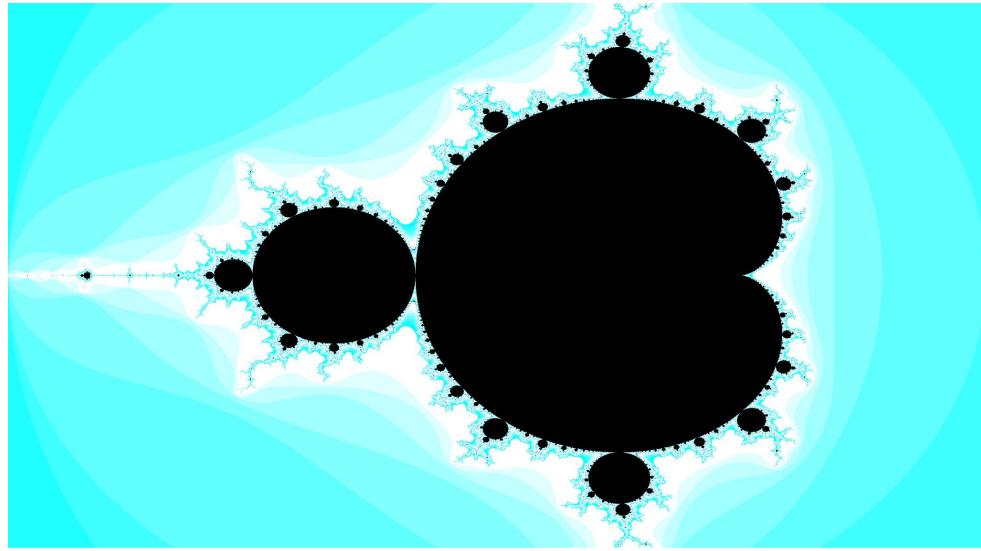
Advisor[s]:
Dov Kruger

Dec 10th, 2025

**Electrical and Computer Engineering Department
Rutgers University, Piscataway, NJ 08854**

1. Introduction

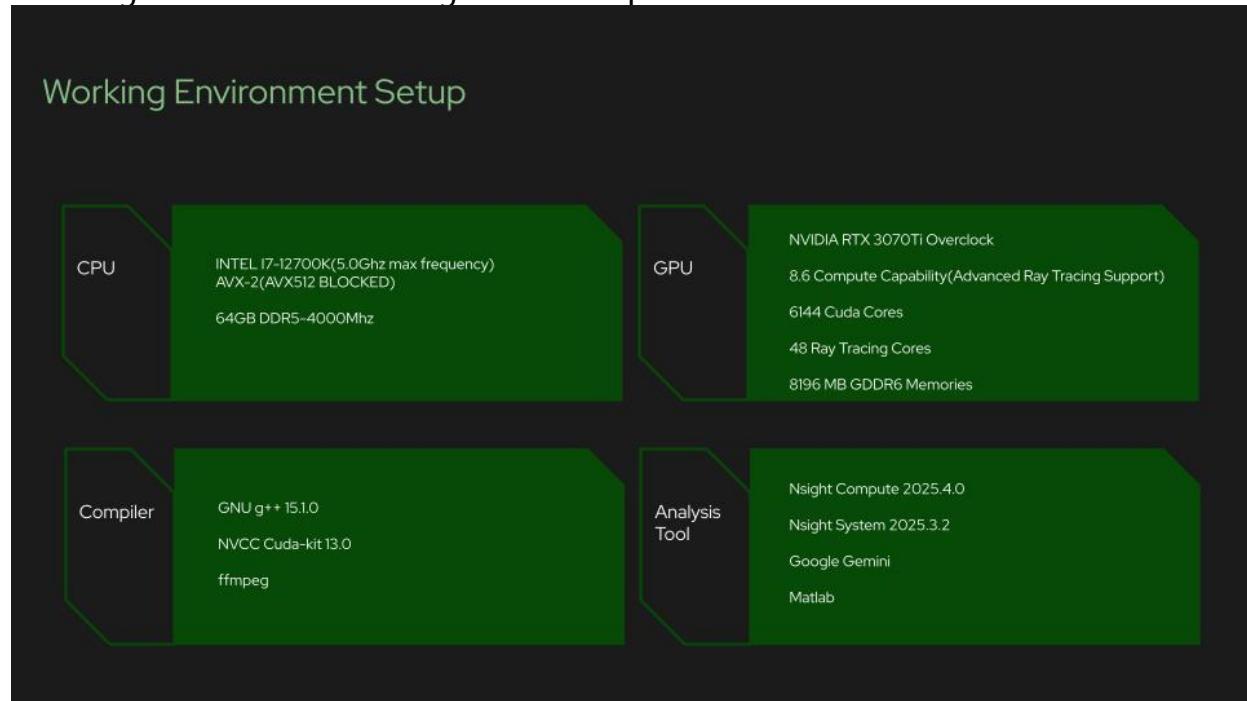
We had the idea of completing this project using CUDA during the process of finishing Mandelbrot homework. Mandelbrot is a program that immensely use the potential of GPU architecture as GPU allows massive parallel computing.



By using GPU acceleration, we are able to achieve much better performance for mandelbrot iterations than using CPU. Hence, it gives us the idea of using CUDA on gravity simulation. Our initial target is to built a naive brute-force simulator using cpp, and then convert it to CUDA codes. After the simulator compute a file containing all position data, we will use another CUDA file to generate frames using the data and render those frames using Ray Tracing techniques.

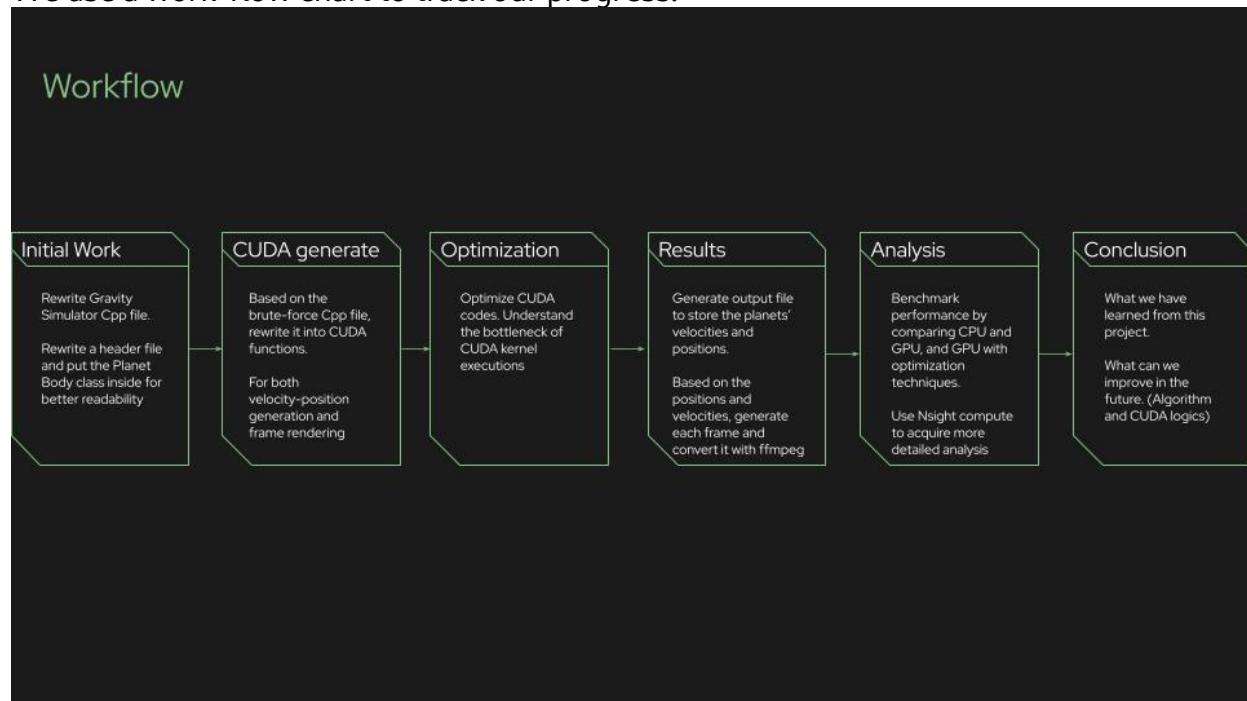
2. Methods / Optimization Process / Results

we setup our working environment with specific tools, compilers and use high end consumer grade computers for simulations.



2.1. Methods

We use a work-flow chart to track our progress.

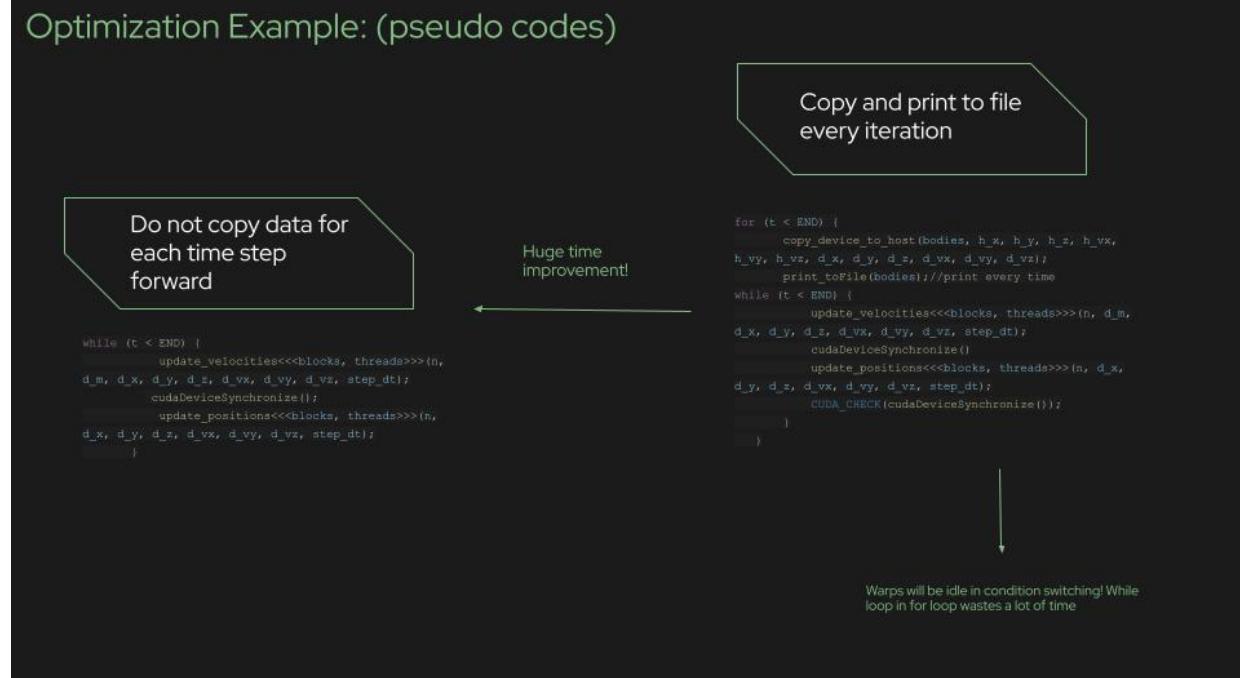


2.2. Optimization Process

While doing our project, we follow a guideline to optimize our codes.

Optimization Process				
Data Structure	Balance Workload	Grid and Block Size	Algorithms	Memory Usages
Improve Data Structure std::vector<Body> Bodies To Std::vector<std::vector<Body>> Bodies	All calculations and ray tracing to be done on GPU	For each GPU architecture, there's an optimized block size.	Minimize if-else statement in GPU kernel to prevent threads from stalling	Maximize memory usage on GPU so that memory transfer time between CPU and GPU is minimized
Store the entire loop history data inside one vector, so the memory access will be sequential	CPU is only used to write out data to txt file	For 3070Ti Ampere architecture, it is recommended to use 256 threads per block.	Use float instead of double, since Ampere architecture for 3070Ti has more FP32 cores than FP64 cores, meaning more single-precision calculation per cycle	

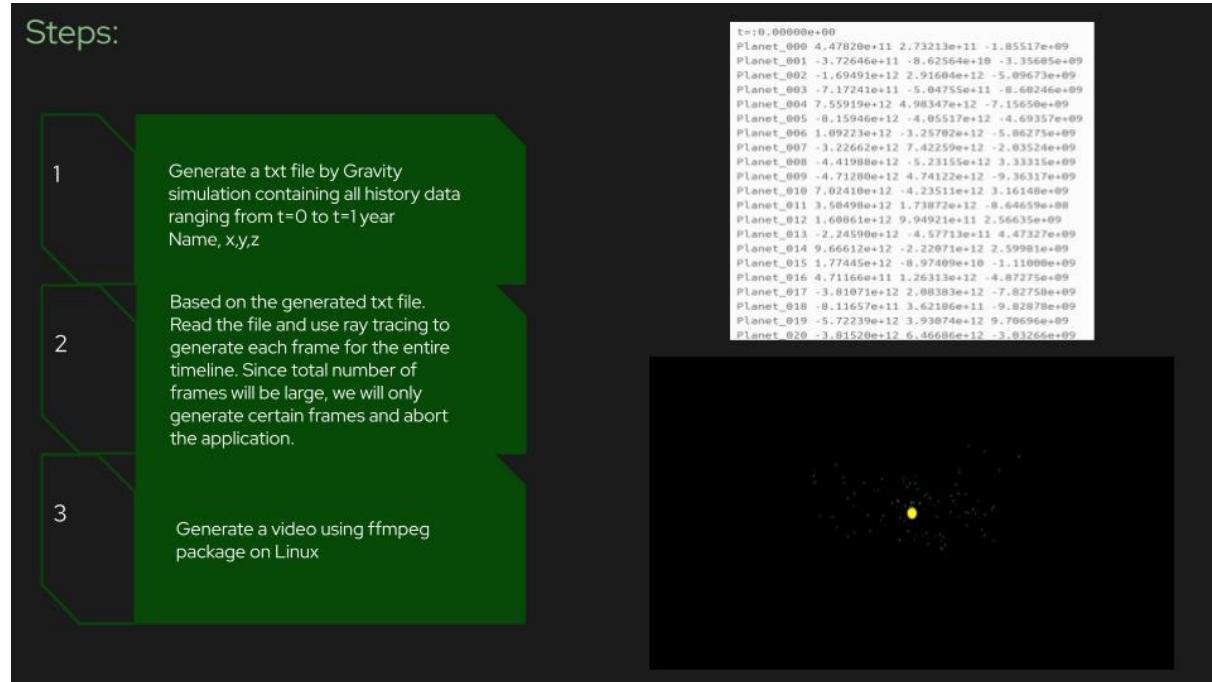
An example is provided to illustrate one of our approaches of optimization



2.3. Results

Our program separately compute a txt file containing all position data w.r.t each planet.

By using those data, the rendering program use ray tracing to render multiple frames, and we use ffmpeg package in Linux to convert all the frames into a video



A link is provided for readers to see the final video:

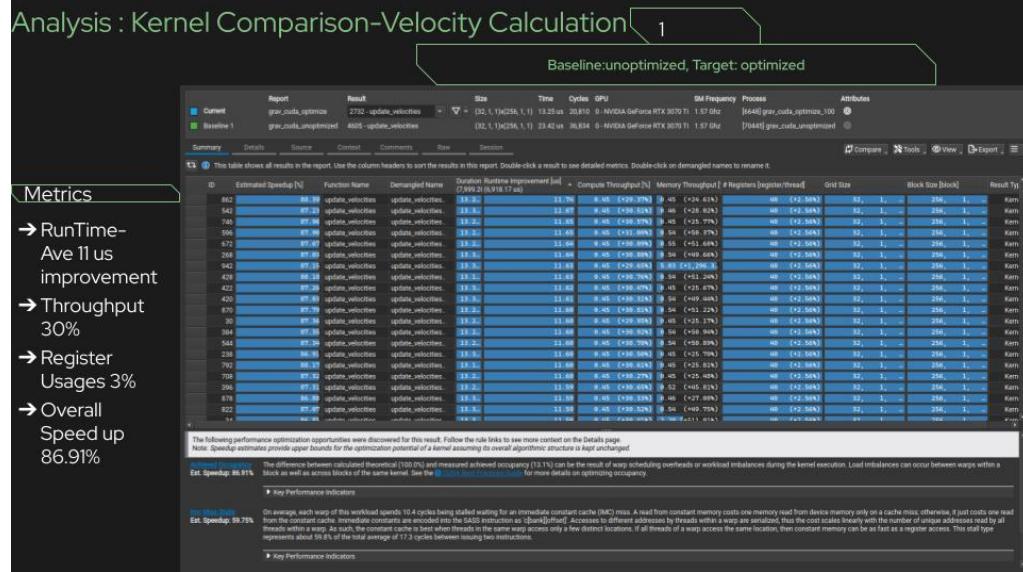
<https://drive.google.com/file/d/150FLoCXtmvFSEA8-obTG3o-hkJqxsGqG/view?usp=sharing>

3. Analysis

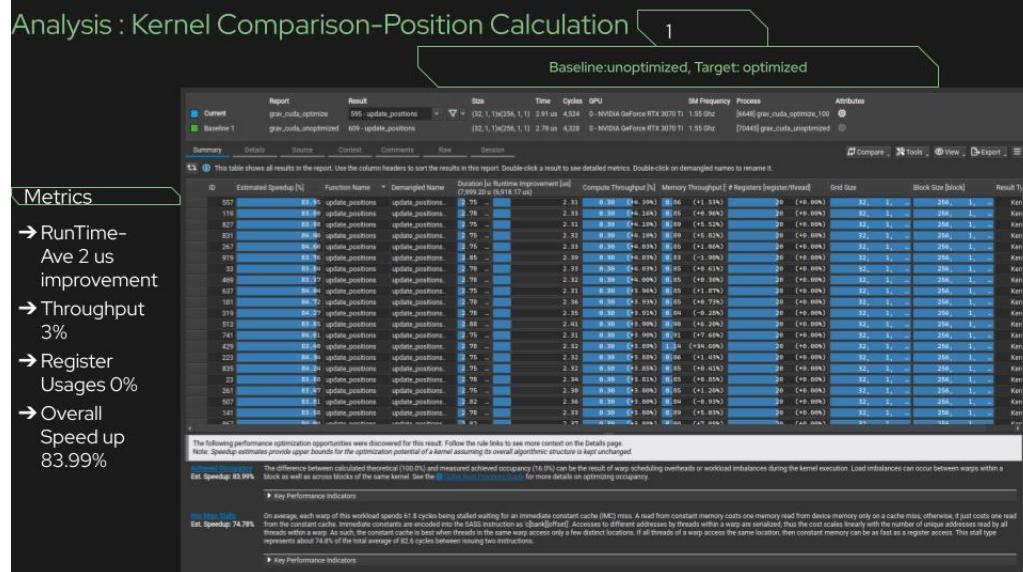
We use Nvidia Nsight Compute tool to analyze our program performance. We compare the optimized version to unoptimized one to check the program speed up.

3.1 Gravity Simulator Version Comparison

Velocity



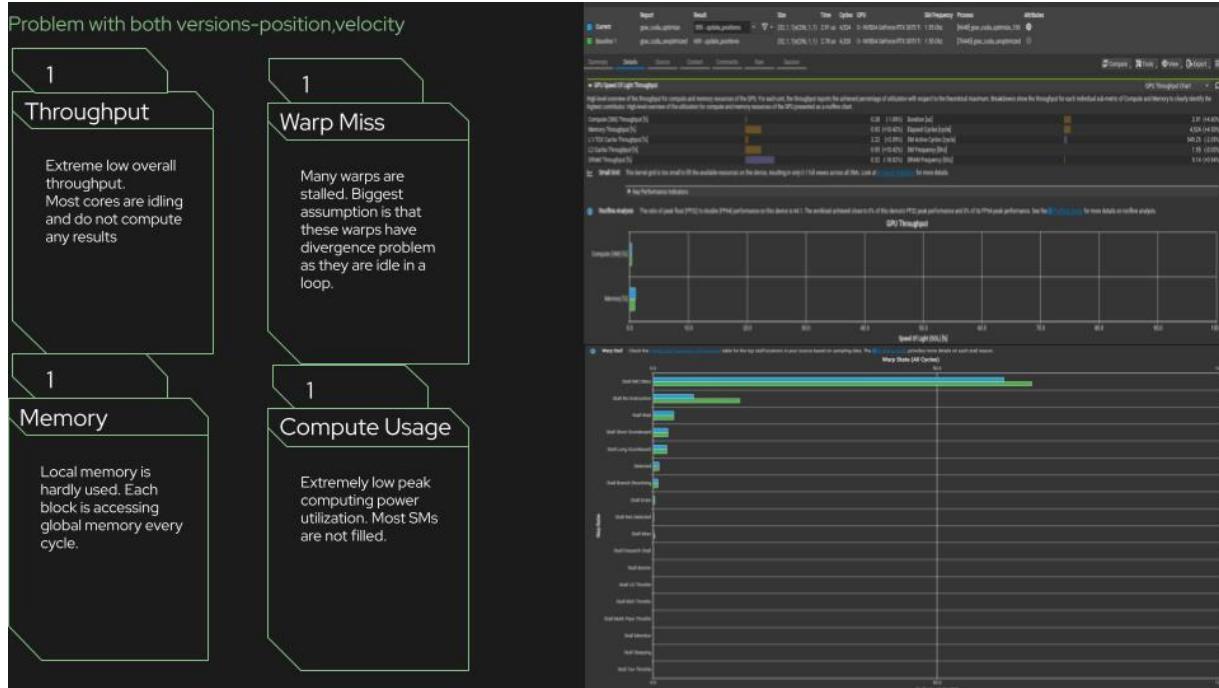
Position



We find out that by optimizing the data writing algorithm, we achieved phenomenon speed ups.

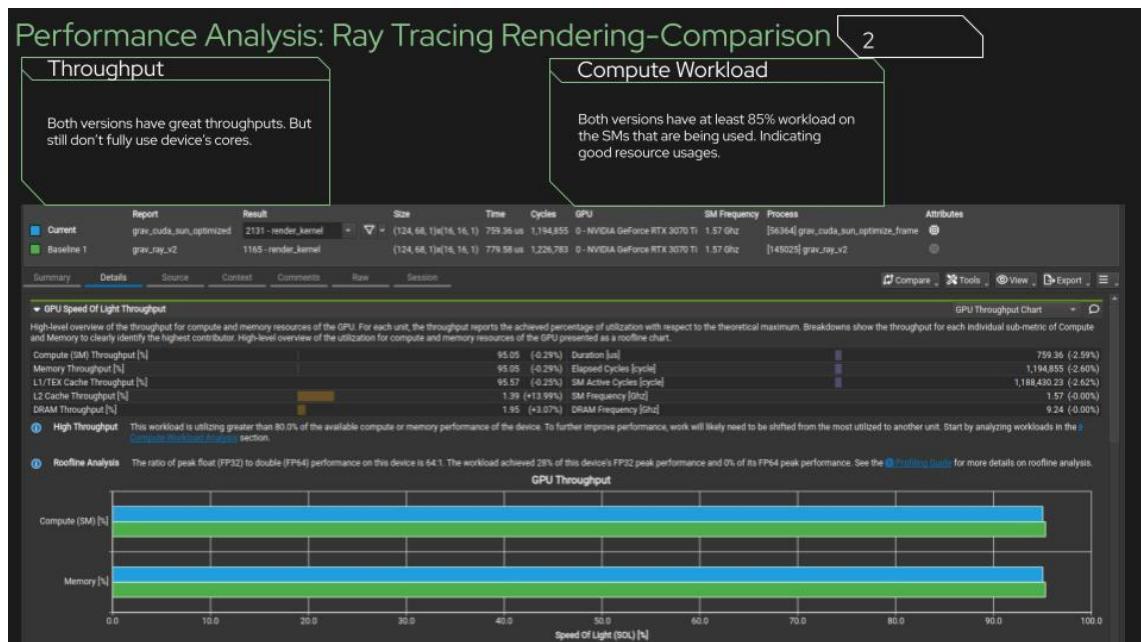
3.1.1 Problems

We discover that we are not using the full potential of SMs, and our memory latency is slowing down the program. The warp divergence problem also causes latency.



3.2 Ray Tracing Rendering

Two versions evidently perform well creating enough throughput



4. Conclusions / Summary

4.1 Conclusion

We benchmark the performance, especially between GPU and CPU

Benchmark

3

GPU vs. CPU

Iteration through 1 year, compute time
100 Planets:
CPU: 2.5 minutes
GPU: 0.97 minutes
1000 Planets:
CPU: 2.54 hours
GPU: 1.56 minutes
10000 Planets:
CPU: 31.57 hours
GPU: 1.24 hour
IM Planets:
CPU: Dead
GPU: 28.2 hours

optimized vs.
unoptimized

Iteration through 1 year, compute time
100 Planets:
OPTIMIZED: 0.6 minutes
UNOPTIMIZED: 0.97 minutes
1000 Planets:
OPTIMIZED: 1.36 min
UNOPTIMIZED: 1.56 minutes

4.2 Summary

What can we improve?

- Need to drastically increase grid size. SMs need to be saturated in order to fetch instructions more efficiently
- Still need to figure out how to use more registers, and make the program use more local memory than accessing global memory to reduce latency
- Still need to fix divergence problem

What have we learned

- Bottleneck of CUDA program
- How to use Nsight tools
- Handle divergence

In the future

- Use more complicated ray tracing platform, such as Nvidia Optix
- Replace Sun with a black hole, and render the frame with truth black hole simulation

5. Acknowledgments

We appreciate the guidance by Professor Kruger.

6. REFERENCES



References

"Unleashing the Power of NVIDIA Ampere Architecture with NVIDIA Nsight Developer Tools." *NVIDIA Technical Blog*, 14 May 2020,
<https://developer.nvidia.com/blog/unleashing-power-of-nvidia-ampere-architecture-with-nsight-developer-tools/>.

"Using Nsight Compute to Inspect Your Kernels." *NVIDIA Technical Blog*, 28 Aug. 2024,
developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels.