CUDA:
GRAVITY SIMULATION w Ray Tracing

Shizhe Yang

OLISADEBE OJUKWU

Introduction

CUDA can accelerate calculation in parallel drastically.

We acquired the idea of completing a CUDA project through Mandelbrot homework.

During Mandelbrot homework, we explored ways to optimize and analyze our codes.

Mandelbrot Idea:

Huge Speed Up

Compared to CPU, GPU acceleration is enormous.

Most ideal case: for 100 million iterations, speed up is 10x.

Limitations

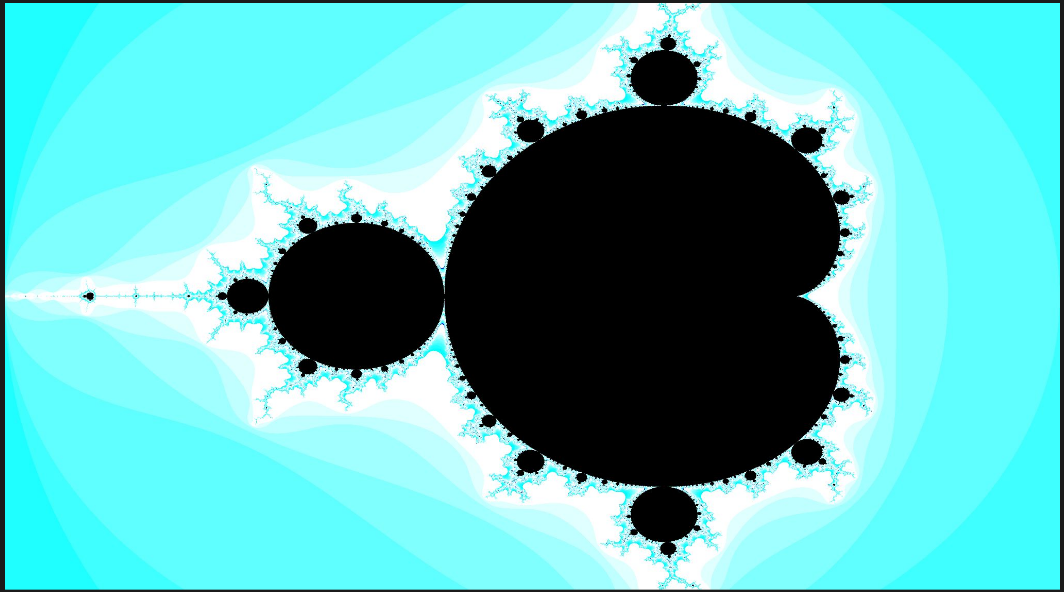
For GPU, it costs a lot of cycles if some of the threads are idle.

If-else statements will cause some threads to execute while other threads are waiting.

CPU is extremely slow compared to GPU

Inspiration

GPU has huge advantages over CPU on repeated, single-precision, parallel processing of images. So we want to do a project on CUDA parallel processing on generating gravitational simulation images.



Methodology

We create the gravity simulation based on the naive

Brute-force simulation file provided in ECE451 repo.

Starting from the CPU version, then we convert the CPU

Version to GPU version for acceleration.

Workflow

Initial Work

Rewrite Gravity Simulator Cpp file.

Rewrite a header file and put the Planet Body class inside for better readability

CUDA generate

Based on the brute-force Cpp file, rewrite it into CUDA functions.

For both velocity-position generation and frame rendering

Optimization

Optimize CUDA codes. Understand the bottleneck of CUDA kernel executions

Results

Generate output file to store the planets' velocities and positions.

Based on the positions and velocities, generate each frame and convert it with ffmpeg

Analysis

Benchmark performance by comparing CPU and GPU, and GPU with optimization techniques.

Use Nsight compute to acquire more detailed analysis

Conclusion

What we have learned from this project.

What can we improve in the future. (Algorithm and CUDA logics)

Working Environment Setup

CPU

INTEL I7-12700K(5.0Ghz max frequency)
AVX-2(AVX512 BLOCKED)

64GB DDR5-4000Mhz

GPU

NVIDIA RTX 3070Ti Overclock

8.6 Compute Capability(Advanced Ray Tracing Support)

6144 Cuda Cores

48 Ray Tracing Cores

8196 MB GDDR6 Memories

Compiler

GNU g++ 15.1.0

NVCC Cuda-kit 13.0

ffmpeg

Analysis Tool

Nsight Compute 2025.4.0

Nsight System 2025.3.2

Google Gemini

Matlab

Optimization

We tried to identify the bottleneck and what is causing the Latency in our codes. We focus on memories usages and Divergence problems. Partition workload between CPU and GPU is crucial

For example:

Velocity and Position are Co-dependent! Cannot compute velocity without computing position first.After stepping forward in time, velocity needs to be calculated using the new position!

Memory Usages

Threads are always accessing the global memories. Cache hit rate is high but still not solving the locality problem. Need to use registers more to speed up the program

Host-Device

Constantly copying memories from host-device, back and forth. It wastes huge amounts of time, as threads will be idle waiting for the data. Need to balance the workload

Algorithms

Threads throttling issue occurs frequently in the program execution process. If-else statement costs a lot of time for threads to wait while other threads are executing.Sometimes loop is inevitable

Optimization Process

Data Structure

Improve Data Structure
std::vector<Body>
Bodies
To
Std::vector<std::vector
<Body> Bodies> Bodies

Store the entire loop history data inside one vector, so the memory access will be sequential

Balance Workload

All calculations and ray tracing to be done on GPU

CPU is only used to write out data to txt file

Grid and Block Size

For each GPU architecture, there's an optimized block size.

For 3070Ti Ampere architecture, it is recommended to use 256 threads per block.

Algorithms

Minimize if-else statement in GPU kernel to prevent threads from stalling

Use float instead of double, since Ampere architecture for 3070Ti has more FP32 cores than FP64 cores, meaning more single-precision calculation per cycle

Memory Usages

Maximize memory usage on GPU so that memory transfer time between CPU and GPU is minimized

Optimization Example: (pseudo codes)

Copy and print to file every iteration

```
for (t < END) {
        copy_device_to_host (bodies, h_x, h_y, h_z, h_vx,
h_vy, h_vz, d_x, d_y, d_z, d_vx, d_vy, d_vz);
        print_toFile(bodies);//print every time
while (t < END) {
            update_velocities <<<blocks, threads>>>(n, d_m,
d_x, d_y, d_z, d_vx, d_vy, d_vz, step_dt);
            cudaDeviceSynchronize ()
            update_positions <<<blocks, threads>>>(n, d_x,
d_y, d_z, d_vx, d_vy, d_vz, step_dt);
            CUDA_CHECK (cudaDeviceSynchronize ());
        }
    }
```

Warps will be idle in condition switching! While loop in for loop wastes a lot of time

Huge time improvement!

Do not copy data for each time step forward

```
while (t < END) {
            update_velocities <<<blocks, threads>>>(n,
d_m, d_x, d_y, d_z, d_vx, d_vy, d_vz, step_dt);
          cudaDeviceSynchronize ();
            update_positions <<<blocks, threads>>>(n,
d_x, d_y, d_z, d_vx, d_vy, d_vz, step_dt);
        }
```

Results

A short video is generated as the final result.

We can do a live demo (if time permitting)

Steps:

1

Generate a txt file by Gravity simulation containing all history data ranging from t=0 to t=1 year
Name, x,y,z

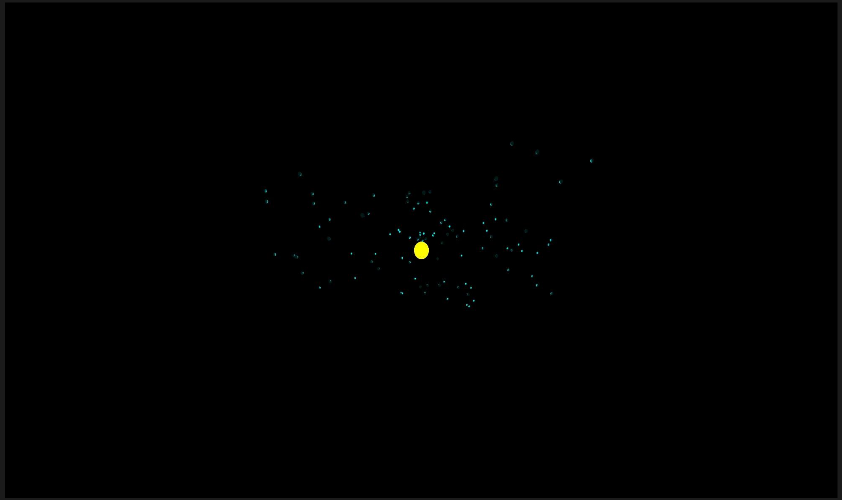
2

Based on the generated txt file. Read the file and use ray tracing to generate each frame for the entire timeline. Since total number of frames will be large, we will only generate certain frames and abort the application.

3

Generate a video using ffmpeg package on Linux

```
t=:0.00000e+00
Planet_000 4.47820e+11 2.73213e+11 -1.85517e+09
Planet_001 -3.72646e+11 -8.62564e+10 -3.35605e+09
Planet_002 -1.69491e+12 2.91604e+12 -5.09673e+09
Planet_003 -7.17241e+11 -5.04755e+11 -8.60246e+09
Planet_004 7.55919e+12 4.98347e+12 -7.15650e+09
Planet_005 -8.15946e+12 -4.05517e+12 -4.69357e+09
Planet_006 1.09223e+12 -3.25702e+12 -5.86275e+09
Planet_007 -3.22662e+12 7.42259e+12 -2.03524e+09
Planet_008 -4.41988e+12 -5.23155e+12 3.33315e+09
Planet_009 -4.71280e+12 4.74122e+12 -9.36317e+09
Planet_010 7.02410e+12 -4.23511e+12 3.16148e+09
Planet_011 3.50498e+12 1.73872e+12 -8.64659e+08
Planet_012 1.60861e+12 9.94921e+11 2.56635e+09
Planet_013 -2.24590e+12 -4.57713e+11 4.47327e+09
Planet_014 9.66612e+12 -2.22071e+12 2.59981e+09
Planet_015 1.77445e+12 -8.97409e+10 -1.11000e+09
Planet_016 4.71166e+11 1.26313e+12 -4.87275e+09
Planet_017 -3.81071e+12 2.08383e+12 -7.82758e+09
Planet_018 -8.11657e+11 3.62186e+11 -9.82878e+09
Planet_019 -5.72239e+12 3.93074e+12 9.70696e+09
Planet_020 -3.81520e+12 6.46686e+12 -3.03266e+09
```



Final Result

Analysis

Nvidia provides system and kernel wide tools for

analyzing memory, core usages, and many other metrics

1

Compare kernels in Gravity Simulation

2

Analyze Ray Tracing Kernel Performance

3

Benchmark

Analysis : Kernel Comparison-Velocity Calculation

1

Baseline:unoptimized, Target: optimized

Metrics

- → RunTime- Ave 11 us improvement
- → Throughput 30%
- → Register Usages 3%
- → Overall Speed up 86.91%

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	grav_cuda_optimize	2732 - update_velocities	(32, 1, 1)x(256, 1, 1)	13.25 us	20,810	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[6648] grav_cuda_optimize_100	
Baseline 1	grav_cuda_unoptimized	4605 - update_velocities	(32, 1, 1)x(256, 1, 1)	23.42 us	36,834	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[70445] grav_cuda_unoptimized	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration (7,999.2(	Runtime Improvement [us] (6,918.17 us)	Compute Throughput [%]	Memory Throughput ['	# Registers [register/thread]	Grid Size	Block Size [block]	Result Typ
862	88.39	update_velocities	update_velocities..	13.2...	11.74	0.45 (+29.37%)	0.45 (+24.61%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
542	87.23	update_velocities	update_velocities..	13.3...	11.67	0.45 (+30.51%)	0.46 (+28.02%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
746	87.96	update_velocities	update_velocities..	13.2...	11.65	0.45 (+30.57%)	0.45 (+25.77%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
596	87.90	update_velocities	update_velocities..	13.2...	11.65	0.45 (+31.00%)	0.54 (+50.37%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
672	87.67	update_velocities	update_velocities..	13.2...	11.64	0.45 (+30.09%)	0.55 (+51.68%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
268	87.03	update_velocities	update_velocities..	13.3...	11.64	0.45 (+30.80%)	0.54 (+49.66%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
942	87.15	update_velocities	update_velocities..	13.3...	11.63	0.45 (+29.65%)	5.03 (+1,296.3...	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
428	88.18	update_velocities	update_velocities..	13.1...	11.63	0.45 (+30.76%)	0.54 (+51.24%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
422	87.26	update_velocities	update_velocities..	13.3...	11.62	0.45 (+30.47%)	0.45 (+25.67%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
420	87.03	update_velocities	update_velocities..	13.3...	11.61	0.45 (+30.31%)	0.54 (+49.44%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
870	87.79	update_velocities	update_velocities..	13.2...	11.60	0.45 (+30.81%)	0.54 (+51.22%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
30	87.36	update_velocities	update_velocities..	13.2...	11.60	0.45 (+29.95%)	0.45 (+25.17%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
384	87.35	update_velocities	update_velocities..	13.2...	11.60	0.45 (+30.92%)	0.54 (+50.94%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
544	87.34	update_velocities	update_velocities..	13.2...	11.60	0.45 (+30.70%)	0.54 (+50.89%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
238	86.91	update_velocities	update_velocities..	13.3...	11.60	0.45 (+30.50%)	0.45 (+25.70%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
792	88.17	update_velocities	update_velocities..	13.1...	11.60	0.45 (+30.61%)	0.45 (+25.81%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
708	87.32	update_velocities	update_velocities..	13.2...	11.60	0.45 (+30.27%)	0.45 (+25.48%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
396	87.31	update_velocities	update_velocities..	13.2...	11.59	0.45 (+30.65%)	0.52 (+45.81%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
878	86.88	update_velocities	update_velocities..	13.3...	11.59	0.45 (+30.33%)	0.46 (+27.08%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern
822	87.07	update_velocities	update_velocities..	13.3...	11.59	0.45 (+30.52%)	0.54 (+49.75%)	40 (+2.56%)	32, 1, ...	256, 1, ...	Kern

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note: *Speedup estimates provide upper bounds for the optimization potential of a kernel assuming its overall algorithmic structure is kept unchanged.*

Achieved Occupancy
Est. Speedup: 86.91%

The difference between calculated theoretical (100.0%) and measured achieved occupancy (13.1%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide for more details on optimizing occupancy.

▸ Key Performance Indicators

Imc Miss Stalls
Est. Speedup: 59.75%

On average, each warp of this workload spends 10.4 cycles being stalled waiting for an immediate constant cache (IMC) miss. A read from constant memory costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache. Immediate constants are encoded into the SASS instruction as 'c[bank][offset]'. Accesses to different addresses by threads within a warp are serialized, thus the cost scales linearly with the number of unique addresses read by all threads within a warp. As such, the constant cache is best when threads in the same warp access only a few distinct locations. If all threads of a warp access the same location, then constant memory can be as fast as a register access. This stall type represents about 59.8% of the total average of 17.3 cycles between issuing two instructions.

▸ Key Performance Indicators

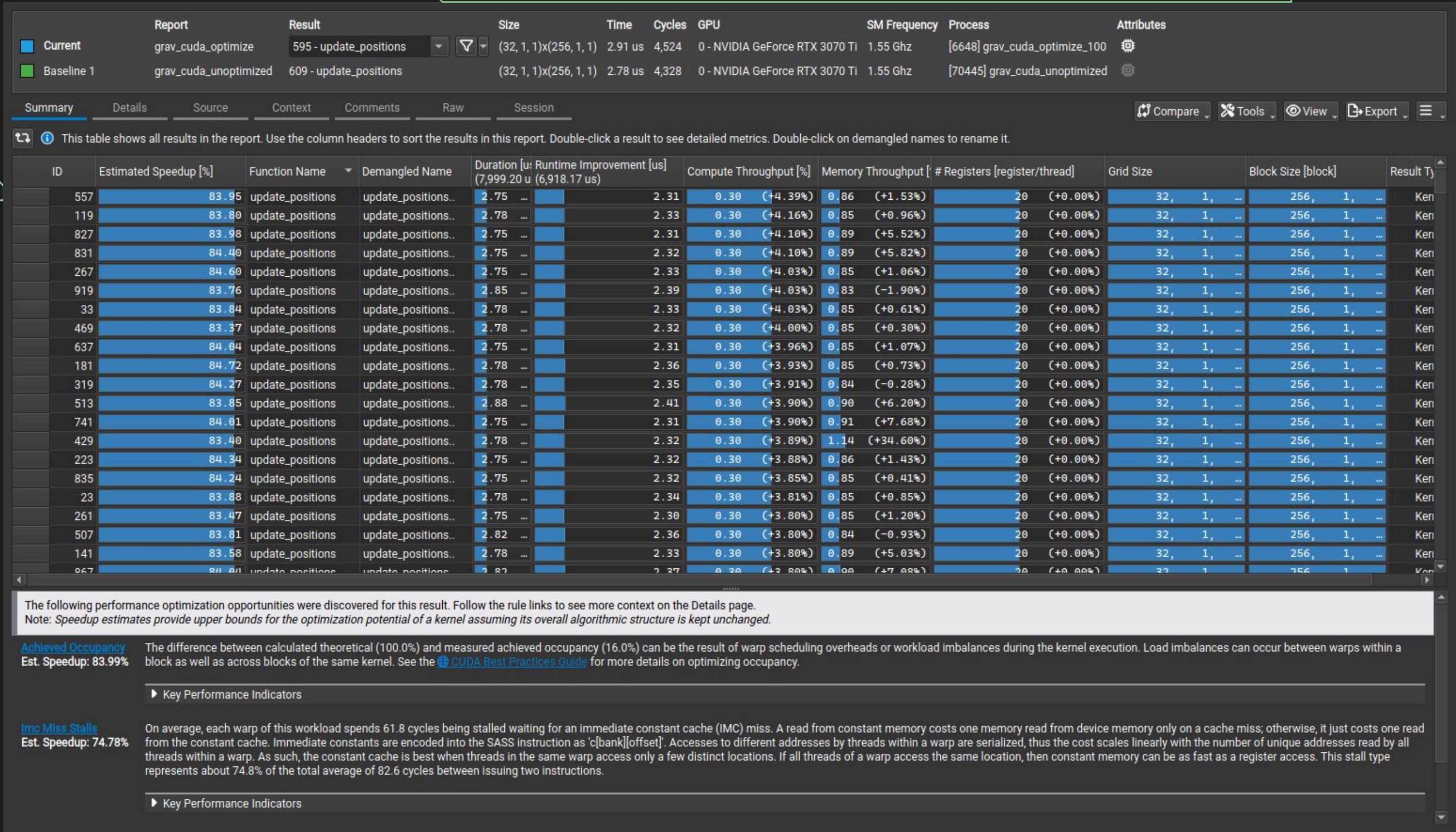
Analysis : Kernel Comparison-Position Calculation

1

Baseline:unoptimized, Target: optimized

Metrics

- ➔ RunTime-Ave 2 us improvement
- ➔ Throughput 3%
- ➔ Register Usages 0%
- ➔ Overall Speed up 83.99%



	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	grav_cuda_optimize	595 - update_positions	(32, 1, 1)x(256, 1, 1)	2.91 us	4,524	0 - NVIDIA GeForce RTX 3070 Ti	1.55 Ghz	[6648] grav_cuda_optimize_100	
Baseline 1	grav_cuda_unoptimized	609 - update_positions	(32, 1, 1)x(256, 1, 1)	2.78 us	4,328	0 - NVIDIA GeForce RTX 3070 Ti	1.55 Ghz	[70445] grav_cuda_unoptimized	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [u: (7,999.20 u	Runtime Improvement [us] (6,918.17 us)	Compute Throughput [%]	Memory Throughput ['	# Registers [register/thread]	Grid Size	Block Size [block]	Result Ty
557	83.95	update_positions	update_positions..	2.75 ...	2.31	0.30 (+4.39%)	0.86 (+1.53%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
119	83.80	update_positions	update_positions..	2.78 ...	2.33	0.30 (+4.16%)	0.85 (+0.96%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
827	83.98	update_positions	update_positions..	2.75 ...	2.31	0.30 (+4.10%)	0.89 (+5.52%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
831	84.40	update_positions	update_positions..	2.75 ...	2.32	0.30 (+4.10%)	0.89 (+5.82%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
267	84.60	update_positions	update_positions..	2.75 ...	2.33	0.30 (+4.03%)	0.85 (+1.06%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
919	83.76	update_positions	update_positions..	2.85 ...	2.39	0.30 (+4.03%)	0.83 (-1.90%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
33	83.84	update_positions	update_positions..	2.78 ...	2.33	0.30 (+4.03%)	0.85 (+0.61%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
469	83.37	update_positions	update_positions..	2.78 ...	2.32	0.30 (+4.00%)	0.85 (+0.30%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
637	84.04	update_positions	update_positions..	2.75 ...	2.31	0.30 (+3.96%)	0.85 (+1.07%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
181	84.72	update_positions	update_positions..	2.78 ...	2.36	0.30 (+3.93%)	0.85 (+0.73%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
319	84.27	update_positions	update_positions..	2.78 ...	2.35	0.30 (+3.91%)	0.84 (-0.28%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
513	83.85	update_positions	update_positions..	2.88 ...	2.41	0.30 (+3.90%)	0.90 (+6.20%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
741	84.01	update_positions	update_positions..	2.75 ...	2.31	0.30 (+3.90%)	0.91 (+7.68%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
429	83.40	update_positions	update_positions..	2.78 ...	2.32	0.30 (+3.89%)	1.14 (+34.60%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
223	84.34	update_positions	update_positions..	2.75 ...	2.32	0.30 (+3.88%)	0.86 (+1.43%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
835	84.24	update_positions	update_positions..	2.75 ...	2.32	0.30 (+3.85%)	0.85 (+0.41%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
23	83.88	update_positions	update_positions..	2.78 ...	2.34	0.30 (+3.81%)	0.85 (+0.85%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
261	83.47	update_positions	update_positions..	2.75 ...	2.30	0.30 (+3.80%)	0.85 (+1.20%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
507	83.81	update_positions	update_positions..	2.82 ...	2.36	0.30 (+3.80%)	0.84 (-0.93%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı
141	83.58	update_positions	update_positions..	2.78 ...	2.33	0.30 (+3.80%)	0.89 (+5.03%)	20 (+0.00%)	32, 1, ...	256, 1, ...	Kerı

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note: *Speedup estimates provide upper bounds for the optimization potential of a kernel assuming its overall algorithmic structure is kept unchanged.*

Achieved Occupancy
Est. Speedup: 83.99%

The difference between calculated theoretical (100.0%) and measured achieved occupancy (16.0%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide for more details on optimizing occupancy.

▸ Key Performance Indicators

Imc Miss Stalls
Est. Speedup: 74.78%

On average, each warp of this workload spends 61.8 cycles being stalled waiting for an immediate constant cache (IMC) miss. A read from constant memory costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache. Immediate constants are encoded into the SASS instruction as 'c[bank][offset]'. Accesses to different addresses by threads within a warp are serialized, thus the cost scales linearly with the number of unique addresses read by all threads within a warp. As such, the constant cache is best when threads in the same warp access only a few distinct locations. If all threads of a warp access the same location, then constant memory can be as fast as a register access. This stall type represents about 74.8% of the total average of 82.6 cycles between issuing two instructions.

▸ Key Performance Indicators

Problem with both versions-position,velocity

1
Throughput

Extreme low overall throughput.
Most cores are idling and do not compute any results

1
Warp Miss

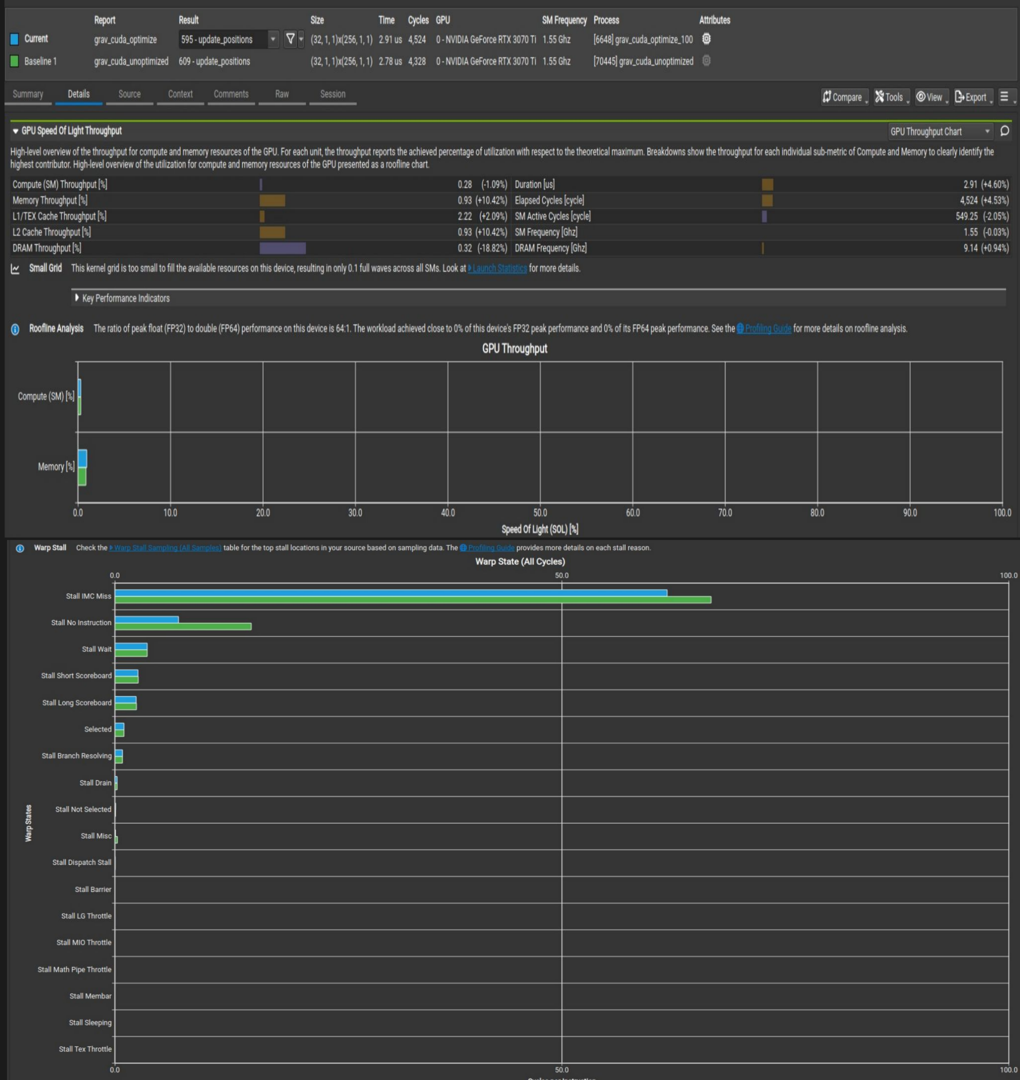
Many warps are stalled. Biggest assumption is that these warps have divergence problem as they are idle in a loop.

1
Memory

Local memory is hardly used. Each block is accessing global memory every cycle.

1
Compute Usage

Extremely low peak computing power utilization. Most SMs are not filled.

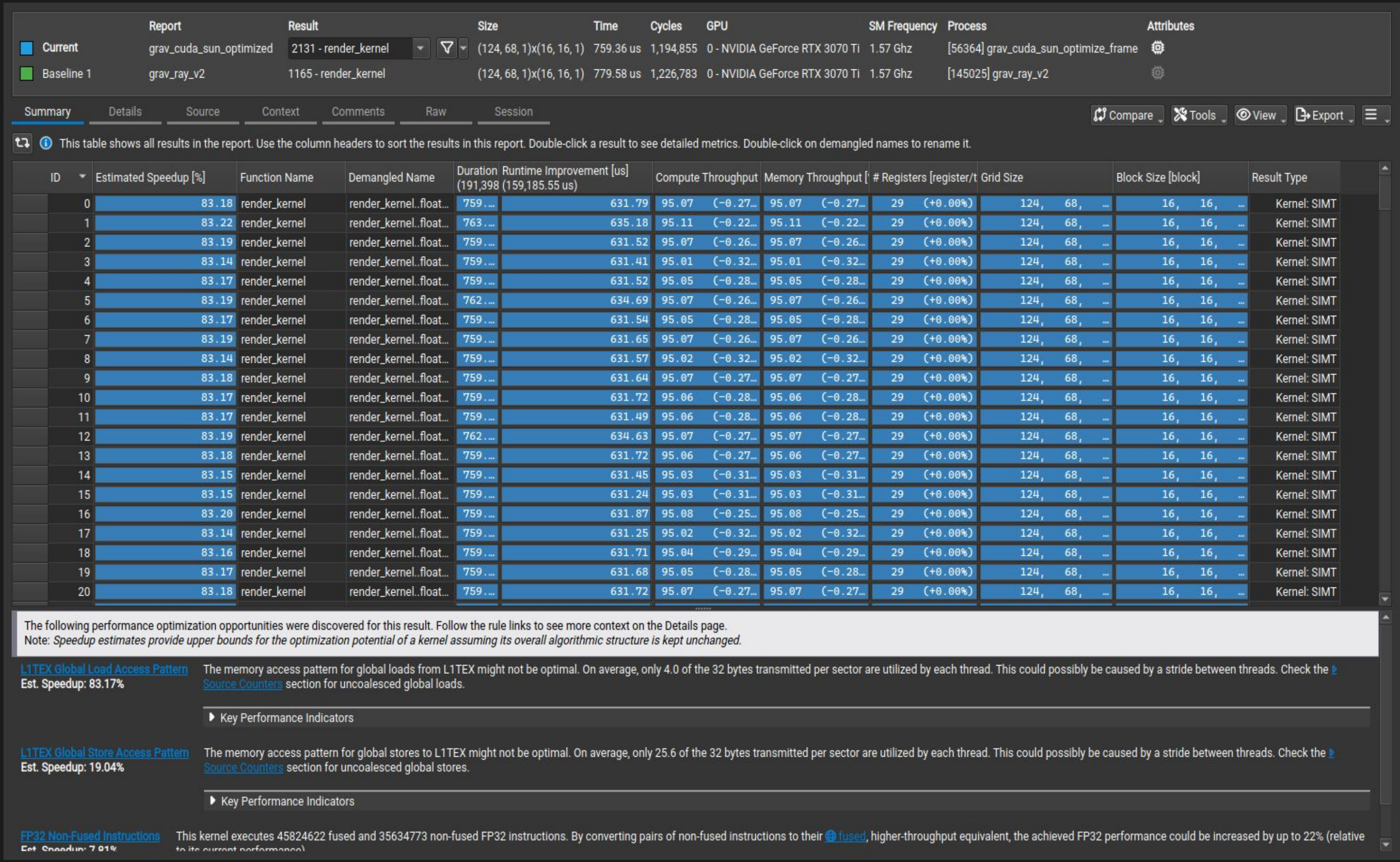


Performance Analysis: Ray Tracing Rendering-Comparison

2

Key Findings:

Analysis shows that two version have basically no difference.
The reason should be that Sun is always at the origin,as a fixed point. So the rendering kernel always need to render the frame,no matter if the program needs to read the position from file or not.



	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process
Current	grav_cuda_sun_optimized	2131 - render_kernel	(124, 68, 1)x(16, 16, 1)	759.36 us	1,194,855	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[56364] grav_cuda_sun_optimize_frame
Baseline 1	grav_ray_v2	1165 - render_kernel	(124, 68, 1)x(16, 16, 1)	779.58 us	1,226,783	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[145025] grav_ray_v2

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration (191,398	Runtime Improvement [us] (159,185.55 us)	Compute Throughput		Memory Throughput [		# Registers [register/t		Grid Size	Block Size [block]	Result Type
0	83.18	render_kernel	render_kernel..float...	759....	631.79	95.07	(-0.27...	95.07	(-0.27...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
1	83.22	render_kernel	render_kernel..float...	763....	635.18	95.11	(-0.22...	95.11	(-0.22...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
2	83.19	render_kernel	render_kernel..float...	759....	631.52	95.07	(-0.26...	95.07	(-0.26...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
3	83.14	render_kernel	render_kernel..float...	759....	631.41	95.01	(-0.32...	95.01	(-0.32...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
4	83.17	render_kernel	render_kernel..float...	759....	631.52	95.05	(-0.28...	95.05	(-0.28...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
5	83.19	render_kernel	render_kernel..float...	762....	634.69	95.07	(-0.26...	95.07	(-0.26...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
6	83.17	render_kernel	render_kernel..float...	759....	631.54	95.05	(-0.28...	95.05	(-0.28...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
7	83.19	render_kernel	render_kernel..float...	759....	631.65	95.07	(-0.26...	95.07	(-0.26...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
8	83.14	render_kernel	render_kernel..float...	759....	631.57	95.02	(-0.32...	95.02	(-0.32...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
9	83.18	render_kernel	render_kernel..float...	759....	631.64	95.07	(-0.27...	95.07	(-0.27...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
10	83.17	render_kernel	render_kernel..float...	759....	631.72	95.06	(-0.28...	95.06	(-0.28...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
11	83.17	render_kernel	render_kernel..float...	759....	631.49	95.06	(-0.28...	95.06	(-0.28...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
12	83.19	render_kernel	render_kernel..float...	762....	634.63	95.07	(-0.27...	95.07	(-0.27...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
13	83.18	render_kernel	render_kernel..float...	759....	631.72	95.06	(-0.27...	95.06	(-0.27...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
14	83.15	render_kernel	render_kernel..float...	759....	631.45	95.03	(-0.31...	95.03	(-0.31...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
15	83.15	render_kernel	render_kernel..float...	759....	631.24	95.03	(-0.31...	95.03	(-0.31...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
16	83.20	render_kernel	render_kernel..float...	759....	631.87	95.08	(-0.25...	95.08	(-0.25...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
17	83.14	render_kernel	render_kernel..float...	759....	631.25	95.02	(-0.32...	95.02	(-0.32...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
18	83.16	render_kernel	render_kernel..float...	759....	631.71	95.04	(-0.29...	95.04	(-0.29...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
19	83.17	render_kernel	render_kernel..float...	759....	631.68	95.05	(-0.28...	95.05	(-0.28...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT
20	83.18	render_kernel	render_kernel..float...	759....	631.72	95.07	(-0.27...	95.07	(-0.27...	29	(+0.00%)	124, 68, ...	16, 16, ...	Kernel: SIMT

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note: *Speedup estimates provide upper bounds for the optimization potential of a kernel assuming its overall algorithmic structure is kept unchanged.*

L1TEX Global Load Access Pattern
Est. Speedup: 83.17%
The memory access pattern for global loads from L1TEX might not be optimal. On average, only 4.0 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the Source Counters section for uncoalesced global loads.

Key Performance Indicators

L1TEX Global Store Access Pattern
Est. Speedup: 19.04%
The memory access pattern for global stores to L1TEX might not be optimal. On average, only 25.6 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the Source Counters section for uncoalesced global stores.

Key Performance Indicators

FP32 Non-Fused Instructions
Est. Speedup: 7.81%
This kernel executes 45824622 fused and 35634773 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP32 performance could be increased by up to 22% (relative to its current performance).

Performance Analysis: Ray Tracing Rendering-Comparison

2

Throughput

Both versions have great throughputs. But still don't fully use device's cores.

Compute Workload

Both versions have at least 85% workload on the SMs that are being used. Indicating good resource usages.

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	grav_cuda_sun_optimized	2131 - render_kernel	(124, 68, 1)x(16, 16, 1)	759.36 us	1,194,855	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[56364] grav_cuda_sun_optimize_frame	
Baseline 1	grav_ray_v2	1165 - render_kernel	(124, 68, 1)x(16, 16, 1)	779.58 us	1,226,783	0 - NVIDIA GeForce RTX 3070 Ti	1.57 Ghz	[145025] grav_ray_v2	

Summary Details Source Context Comments Raw Session Compare Tools View Export

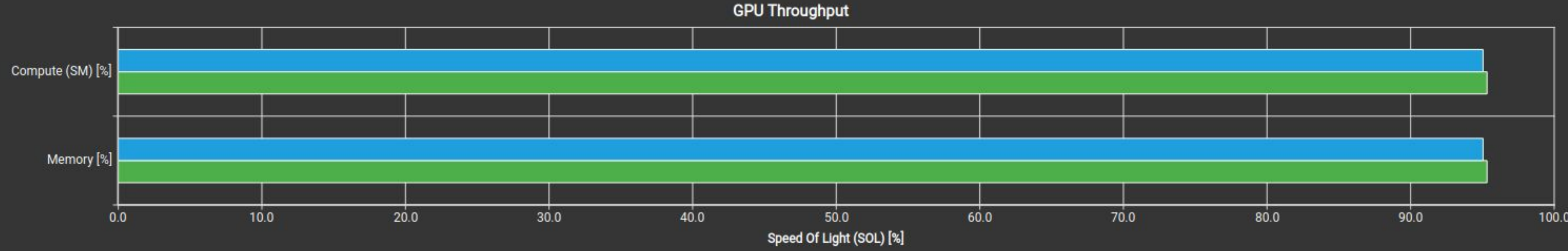
GPU Speed Of Light Throughput GPU Throughput Chart

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	95.05 (-0.29%)	Duration [us]	759.36 (-2.59%)
Memory Throughput [%]	95.05 (-0.29%)	Elapsed Cycles [cycle]	1,194,855 (-2.60%)
L1/TEX Cache Throughput [%]	95.57 (-0.25%)	SM Active Cycles [cycle]	1,188,430.23 (-2.62%)
L2 Cache Throughput [%]	1.39 (+13.99%)	SM Frequency [Ghz]	1.57 (-0.00%)
DRAM Throughput [%]	1.95 (+3.07%)	DRAM Frequency [Ghz]	9.24 (-0.00%)

High Throughput This workload is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (FP32) to double (FP64) performance on this device is 64:1. The workload achieved 28% of this device's FP32 peak performance and 0% of its FP64 peak performance. See the Profiling Guide for more details on roofline analysis.



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:`1.24Hour`
1M 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:`1.36min`
UNOPTIMIZED:`1.56 minutes`

Conclusion

What we've learned from this project

What we hope to do to keep improving this project

What we didn't do well, and what we've done well

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

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.

Thank you