ECE408 Final Project Milestone 3

Group 08

Native Performance Analysis

The native CUDA version code is run under spp = 1000, and NVIDIA Nsight Compute is used for profiling.

GPU Speed of Light

Time (s)	3.03
SoL SM (%)	38.48
SoL Memory (%)	67.18

After analyzing the utilization for compute and memory resources of the GPU, we can clearly see that memory is more heavily utilized than Compute. Therefore, a good optimization approach is to improve warp utilization or to reduce memory throughput.

Memory Workload

Memory Throughput (GB/s)	437.63
Global Load Cached	11.57G
Local Load Cached	4.89G
Shared Load	0
L1 Hit Rate (%)	64.13

Memory throughput is heavy here. Although most of the global memory access has been cached, there is still a huge amount of uncached local memory access. We should be able to further improve L1 Hit Rate by optimizing local load and store.

Scheduler Statistics

Theoretical Warps / Scheduler	16
Active Warps / Scheduler	3.91

Eligible Warps / Scheduler	0.29
Issued Warps / Scheduler	0.22

The size of warps should be 16 theoretically, but only 3.91 of them are active. Work imbalance of warps could be the main reason. There is still room for further optimization.

Warp State Statistics

Warp Cycles per Issued Instruction	17.42
Stall No Instructions	10.95
Stall Wait	3.14

On average each warp of this kernel spends 10.9 cycles being stalled due to not having the next instruction fetched yet. This represents about 62.8% of the total average of 17.4 cycles between issuing two instructions. A high number of warps not having an instruction fetched is typical for very short kernels with less than one full wave of work in the grid. Further improvement of warps utilization is needed.

Launch Statistics & Occupancy

Theoretical Occupancy (%)	25
Th. Active Warps per SM	16
Achieved Occupancy (%)	24.58
Waves per SM	19.20
Registers Per Thread	110

Theoretical occupancy is only a quarter due to the large amount of registers used by each thread. Resources can be better utilized if we can reduce register consumption.

Optimization 1 Optimizing Iteration Code

In our previous attempt in Milestone 2, we converted the recursion in *Radiance()* function to iteration by using a stack to keep track of the variables and set a hard iteration limit. In the revised code, we removed the limit on iteration and removed the stack. Instead, we update two variables: accumulated color and accumulated reflectance.

	Native	With Optimization 1	Improvement (%)
Time (s)	3.03	2.74	-9.57%
SoL SM (%)	38.48	69.24	+79.93%
SoL Memory (%)	67.18	15.11	-77.51%

Memory Workload

	Native	With Optimization 1	Improvement (%)
Memory Throughput (GB/s)	437.63	0.015	-99.99%
L1 Hit Rate (%)	64.13	99.99	+55.91%

This approach hugely decreases the memory throughput. We eliminate all local memory loads and stores, thus nearly all memory access has been cached.

Optimization 2 Double to Float

Double precision operations can be several times slower than single precision operations on some GPUs. We converted the type of some variables from double to float and marked constants in the code as 'f' to make the auto-conversion to double as late as possible.

	With Optimization 1	With Optimization 1&2	Improvement (%)
Time (s)	2.74	2.58	-5.84%
SoL SM (%)	69.24	65.79	-4.98%
SoL Memory (%)	15.11	15.03	-0.53%

Since the only change for Optimization 2 is lowering accuracy for couple operations, the final result meets our anticipation that the running time goes down a little bit while others stay the same.

Optimization 3 Optimizing Block Size

We changed the block size from 16*16 to 32*16. Larger block size ensures more warps available in each SM, therefore achieving better warp scheduling. Since the number of registers per SM could not exceed 65536 in current GPU, a larger block size won't execute. Moreover, we make the block length able to be divisible by the image size in order to eliminate branch divergence.

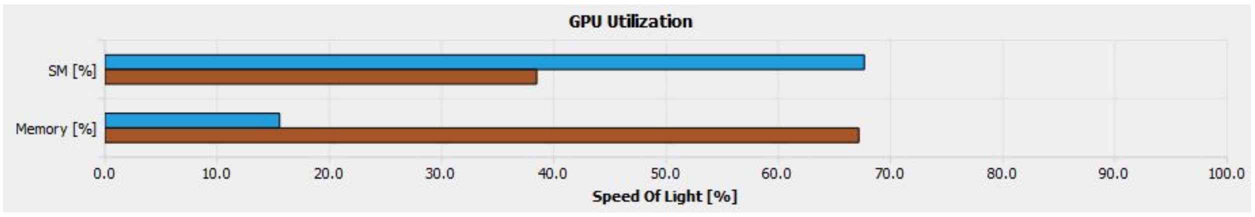
	With Optimization 1&2	With Optimization 1,2&3	Improvement (%)
Time (s)	2.58	2.48	-3.88%
SoL SM (%)	65.79	67.66	+2.84%
SoL Memory (%)	15.03	15.57	+3.59%

This approach further shortens the time by 3.88%.

Overall Results

GPU Speed of Light

	Native	With Optimization 1,2&3	Improvement (%)
Time (s)	3.03	2.48	-18.15%
SoL SM (%)	38.48	67.66	+75.84%
SoL Memory (%)	67.18	15.57	-76.82%

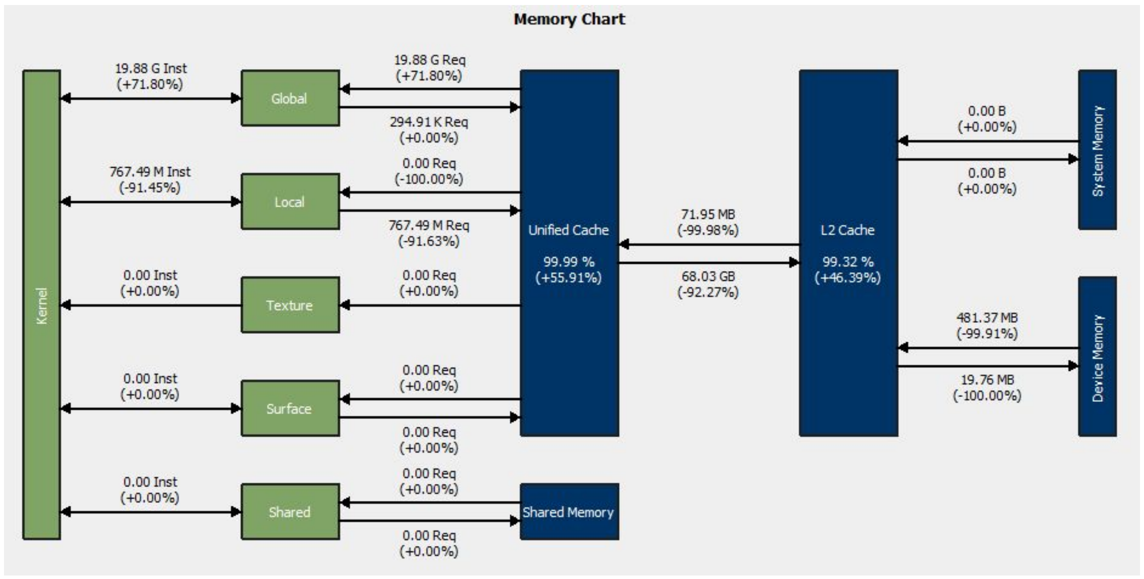


In all, we successfully optimized the usage of SM by 75.84%, and reduced the memory throughput by 76.82%. The improvement of time was not very significant compared with other factors, but it achieved a lower baseline.

Memory Workload

	Native	With Optimization 1,2&3	Improvement (%)
Memory Throughput (GB/s)	437.63	0.211	-99.95%
Global Load Cached	11.57G	19.88G	+71.80%
Local Load Cached	4.89G	0	-100%

Shared Load	0	0	0
Global Store	294K	294K	0
Local Store	4.09G	0.77G	-81.23%
L1 Hit Rate (%)	64.13	99.99	+55.91%



By converting 4.98G local load into global load, we could successfully obtain a L1 cache hit of 99.99% , thus decreasing memory throughput by 99.95% without using shared memory.

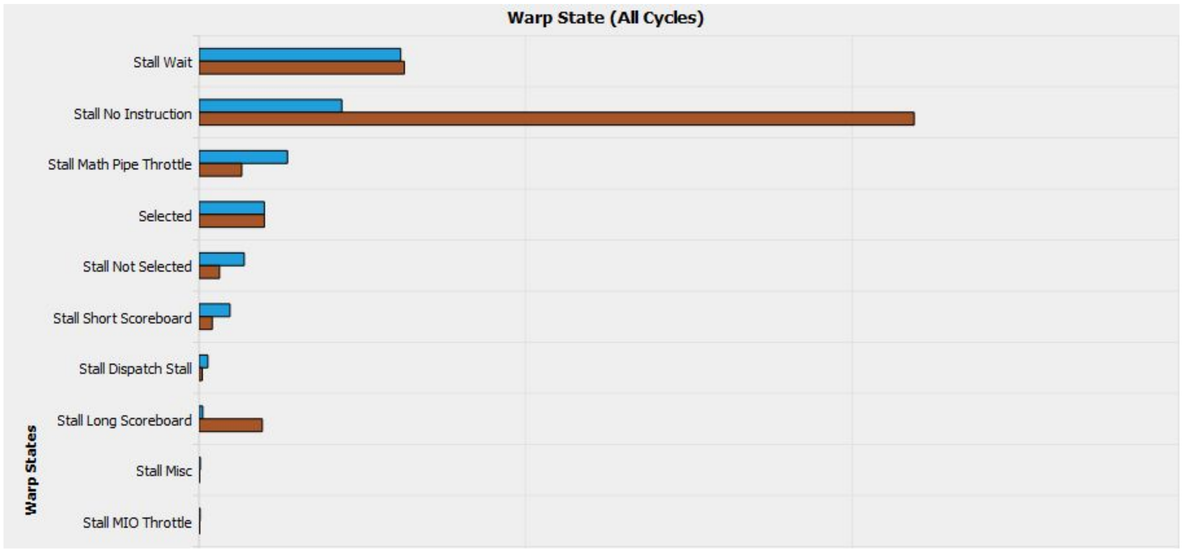
Scheduler Statistics

	Native	With Optimization 1,2&3	Improvement (%)
Theoretical Warps / Scheduler	16	16	0
Active Warps / Scheduler	3.91	3.92	+0.18%
Eligible Warps / Scheduler	0.29	0.73	+149.88%
Issued Warps / Scheduler	0.22	0.43	+93.31%

The improvement on eligible warps was significant by around 150%. There’s no obvious shift in active warps, and load balancing was still not optimal. It could be a limitation of the current ray-tracing algorithm, which would be investigated in the future.

Warp State Statistics

	Native	With Optimization 1,2&3	Improvement (%)
Warp Cycles per Issued Instruction	17.42	9.03	-48.17%
Stall No Instructions	10.95	2.19	-80.04%
Stall Wait	3.14	3.09	-1.80%



From the above graph, we can see that the warp stall was greatly eliminated, especially warps stall due to no instructions, which indicates a better utilization of warps.

Launch Statistics & Occupancy

	Native	With Optimization 1,2&3	Improvement (%)
Theoretical Occupancy (%)	25	25	0
Th. Active Warps per SM	16	16	0
Achieved Occupancy (%)	24.58	24.11	-1.89%
Waves per SM	19.20	19.20	0
Registers Per Thread	110	112	1.82%

After our optimization approaches, two extra registers are used per thread, which will slightly worsen the occupancy but has little effect on overall performance.