Multi-core Architectures 2019/2020

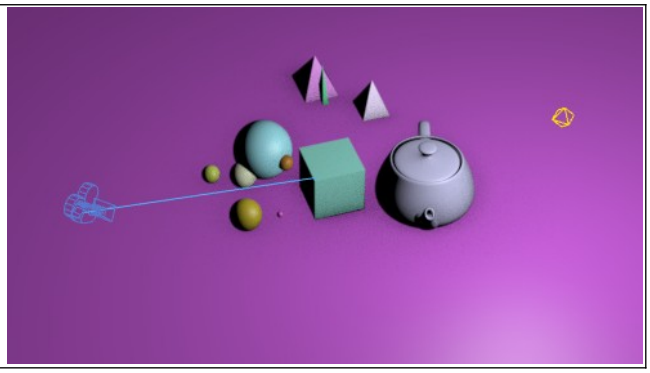
Parallel implementation evaluation report

1. Project details

Project title:	Simple ray-tracing
Student's name:	Łukasz Stalmirski
Index number:	160333
Email:	s160333@student.pg.edu.pl

2. Description of data used for experiments (including examples, when possible)

The scene was saved in FBX format. The ray-tracer generated 640x480 image without AA (Anti-Aliasing). The scene consisted of randomly distributed objects: 1 cube, 6 spheres, 1 hot teapot, 3 pyramids and 1 plane (not-aircraft), and was lighted with one white light (30 samples).



3. Environment #1 description

OS	Windows 10.0.18363.476
CPU	Intel Core i7 8700K A0 @ 3.70GHz
RAM	32GB DDR4-2400 CL15

4. Environment #2 description

OS	Windows 10.0.18363.592
CPU	Intel Core i7 8700K A0 @ 3.70GHz
GPU	NVIDIA GeForce GTX 970 A1 4GB @ 1050MHz 441.87 DCH WHQL (2020.01.06)
RAM	32GB DDR4-2400 CL15

5. Test results

Implementation	Execution time*	
	Mean [s]	**Uncertainty [s]**
Multi-core CPU + OpenMP	26.6	0.2
GPU + CUDA	55.8	0.1

* calculated over 10 executions, uncertainty calculated as: $\Delta T = \frac{T_{max} - T_{min}}{2}$

6. Implementation #1 details

The most challenging task was to fully utilize available CPU resources. First version of the application used 100% of the CPU only for the first half of the execution time, then the utilization linearly decreased.

This issue was solved by introducing tasks:
1. Ray-triangle intersection task
2. Fragment light intensity calculation task

Ray-triangle intersection task tests if the ray hits any triangle. If so, a new fragment light intensity calculation task is spawned. If the ray's depth is smaller than the maximal depth, a new ray-triangle intersection task for the reflected ray is spawned as well. When the maximal depth of a ray is reached, the last light intensity calculation task updates colors of all rays in the path back to the primary ray. The execution is stopped when there are no more tasks to process.

List of implemented optimizations:

- Bounding-boxes:	+84,54%
- Vectorization (SSE3+FMA):	+41,50%
- Tasks:	+35,42%
- Backface culling:	+13,63%

Total improvement in the optimized application:	+94,40%
Execution time in the non-optimized application:	7min 44s
Execution time in the optimized application:	0min 26s

Image generated by the implemented ray-tracer. Time: 7min 23s.
(60 objects, 390 299 triangles, 1 light source, 640x480).
The time was extended due to enabled supersampling (4 samples per pixel) which reduces aliasing.

7. Implementation #2 details

Architecture of the solution had to be rethought to fully utilize CUDA device. The concept of tasks has been removed to avoid submission of many short kernels. Instead, image generation is divided into 3 stages:

1. Intersect current ray set with all objects
2. Compute light intensities for each intersection and update primary rays
3. Spawn secondary rays for current ray set

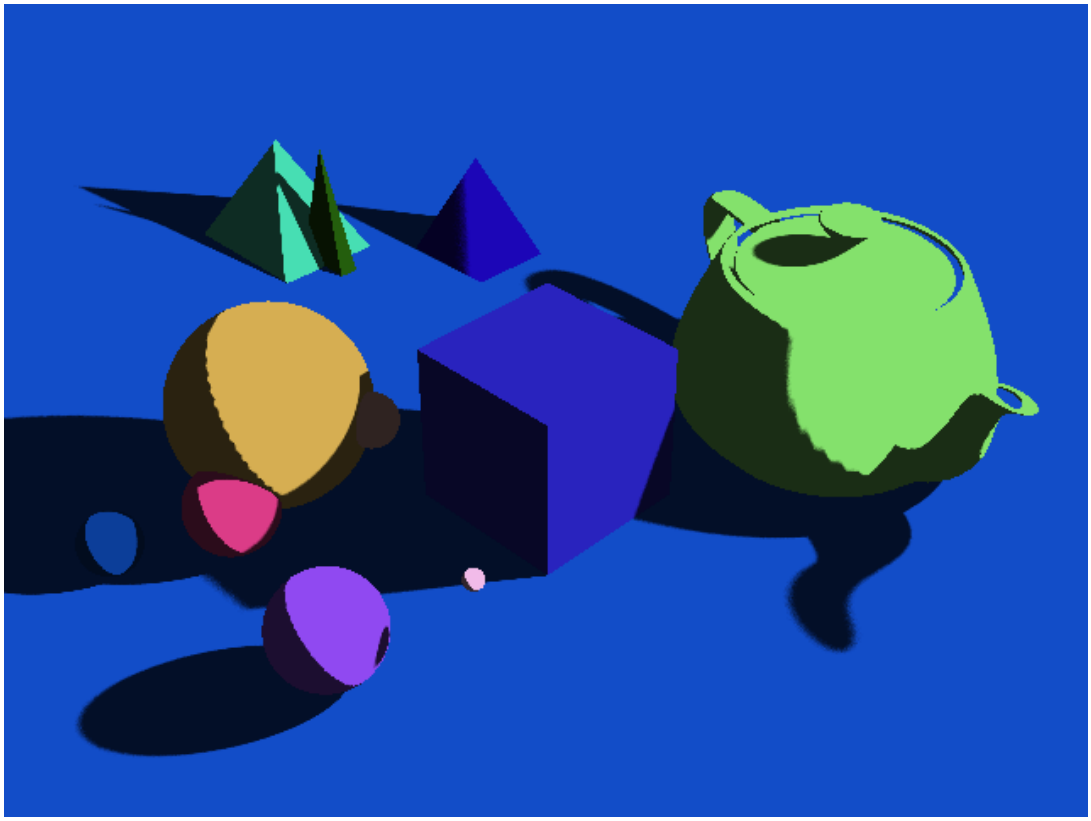
These stages are repeated until there are no more rays to process. Then final kernel is invoked, which resolves primary rays and stores data in output image memory.

This approach however results in performance much lower than CPU-based solution. Achieving better results would require deeper analysis of the kernels. Utilizing more CPU cores for CPU-GPU synchronization latency hiding also might help improve performace.

Interestingly, backface culling deteriorates performance of the application by almost 1%. This may be caused by less branching and holding threads in warps (which must execute the same instruction at a time) as well as less instruction cache misses.

List of implemented optimizations:

- Bounding-boxes:	+43,37%
- Backface culling:	-0,72%
Total improvement in the optimized application:	+42,86%
Execution time in the non-optimized application:	1min 38s
Execution time in the optimized application:	0min 55s



Test image generated with CUDA-based ray-tracer. Time: 55s 834ms.
(12 objects, 68 488 triangles, 1 light source, 640x480).

8. Survey

Fill the answers for questions related to frameworks that you used in your project:

a. How many lines of code did you write for:

   i. OpenMP implementation: 1860

   ii. CUDA implementation: 1131

   iii. MPI implementation: -

b. How would you describe programming difficulty of each framework/interface in 1-10 scale (1 – easy, 10 – difficult):

   i. OpenMP: 4

   ii. CUDA: 8

   iii. MPI: -