Accelerating Drone's Mapping System Using Parallel Tools

Peiqing Chen, Yuxiao Luo

In this report, we study accelerating a real-world drone's mapping system with parallel tools and see the practical improvement that can be made. The system we study is named as Octomap, which is the prevailing mapping system working on UAV (Unmanned Aerial Vehicles), and some robotics and some autonomous systems nowadays. We decompose the components of Octomap and realized that the ray-tracing part can be the bottleneck along the critical path. In order to relief this issue and accelerate the drone's performance, we use OpenMP which successfully raise the ray-tracing part by 51.7% and raise the speed of the whole system by 28.4%.

Hornung, Armin, et al. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees." Autonomous robots 34.3 (2013): 189-206.

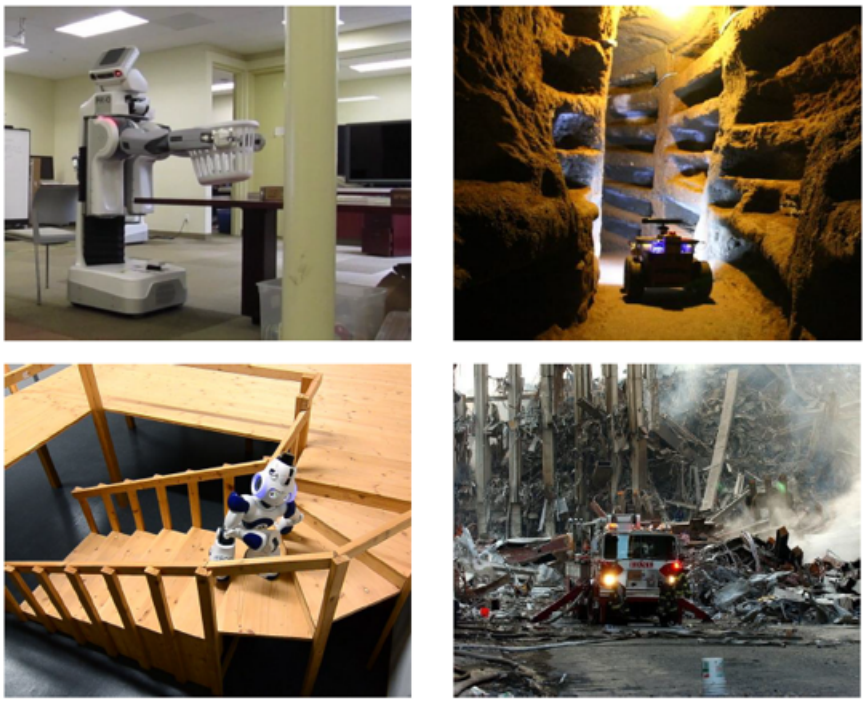

Figure 1: Real world use cases of drones

1 Background And Motivation

Drones, UAVs (Unmanned Aerial Vehicles) and robots need mapping systems to map the 3D world they perceive through their sensors (e.g. lidars and cameras) to their on-chip memories. This data structure on their memory records information of the space (e.g. space occupancy) and can guide them to perform various tasks, including path planning, obstacle avoidance, rescuing, etc (Figure 1). One of the most popular mapping system among them is Octomap.

As Figure 2 shows, Octomap maps the 3D space into 3D grids named as voxels. Each voxel is recursively divided into 8, whose side length is half of the original one. These multi-level voxels forms a tree structure, namely OcTree, where each parent node has 8 child nodes. Octomap puts an occupancy value, which is a float number, to each of the nodes in the OcTree. The parent node has occupancy value which is the maximum among all its 8 children. In this way, OcTree will be able to answer voxel occupancy queries based on different resolutions (levels in the OcTree).

2 Octomap workflow anaylsis

In Figure 3 we show the workflow of Octomap. The drone is equiped with a lidar to sense the environment. In one image shooting, the lidar can provide the drone with a perception of its surroundings, represented in the form of a point cloud. The point cloud will depict a contour of the obstacle in the front and each point in the point cloud represents a point on the obstacle surface.

The next step is ray-tracing. This step allows the drone to convert the observed points over the obstacle surface and all empty space between the drone and the obstacle into 3D grids which can be latter recorded into the back-end data structure (OcTree). More specifically, Octomap first cuts the whole space into voxels (i.e. 3D grids) with the finest resolution set in advance. Then, for each point in the point cloud, it first draws a line segment between the starting point (the 3D coordinates of the position of the drone) and the end point (the target of ray-tracing process). Afterwards,

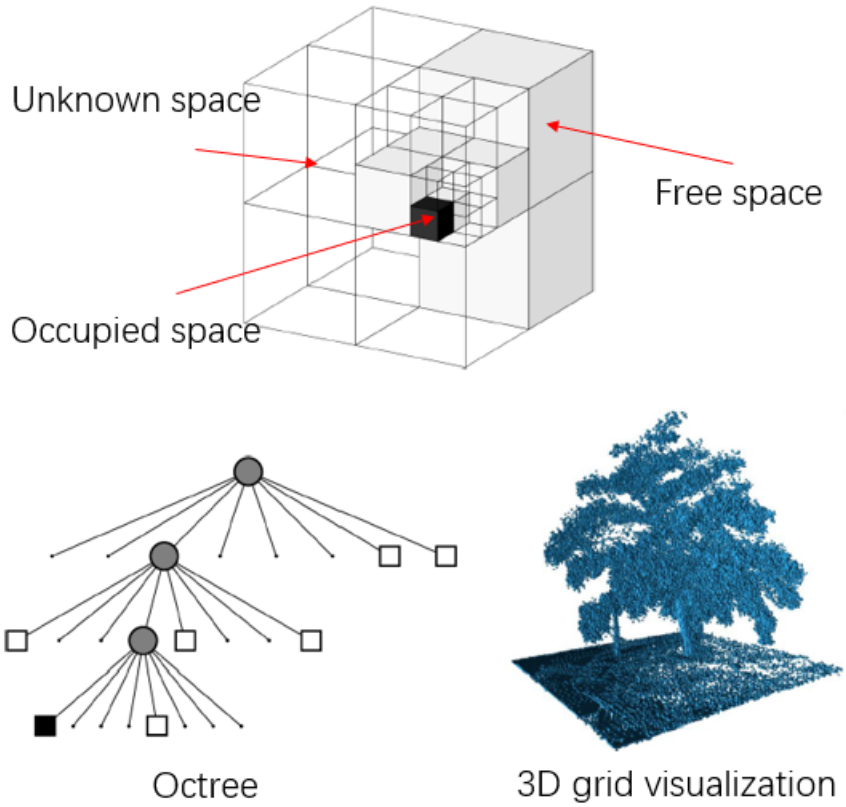

Figure 2: Data structure of OcTree and 3D visualization of using OcTree to represent the space with occupied and free voxels.

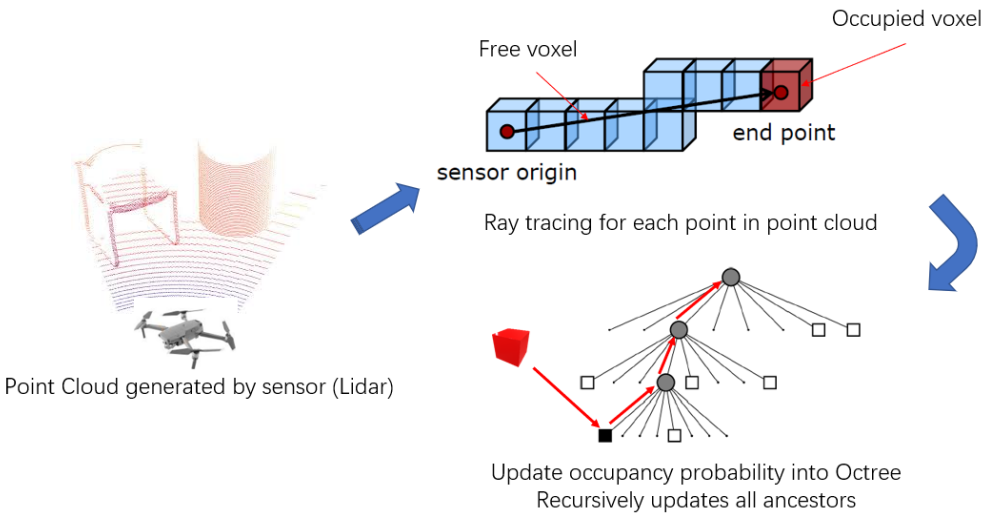

Figure 3: The workflow of Octomap can be generally divided into 3 parts: lidar sensing (point cloud generation), ray-tracing and OcTree insertion.

Octomap computes all the voxels that this line segment passes through. Finally, all the voxels except the one that contains the end point are marked as free voxels, and the last one is marked as an occupied voxel.

After performing the ray tracing process towards all the points in the point cloud, one set of free voxels and another set of occupied voxels are acquired. Notice that both sets are non-duplicated sets, which means that each voxel only has one upate to the OcTree during one point cloud processing. As is shown in Figure 3, the OcTree is a tree data structure where the last level children are the voxels of the finest resolution. Each point is first inserted into the bottom layer, and iteratively update all its ancestor nodes along the path to the root.

Figure 3 shows decomposed run time profiling of Octomap over 3 open-sourced datasets. The performance can vary from case to case in practice due to different mapping resolution and environment. However, we observe that in the worst case, the ray tracing tasks can take up to 82.7% of the total run time. Also, its intuitive that the ray-tracing process towards all the points in the point cloud can be computed in parallel. Therefore, we aim to accelerate this step using OpenMP tools.

3 Analysis Figure 5 shows how the parallel job is done on the bottom level. It is intuitive that the ray-tracing computation part (referring to Figure 4) can be paralleled. However, this will put overhead on synchronizing them in the merging 2 sets part. Also, using more threads (more cores) will accelerate the ray-tracing computation more but will stalk the merging step more. Therefore, there exists an optimal choice of number of paralleling threads.

4 Experiment We have done extensive experiment over using different number of CPU cores, different scheduling rules over three different datasets.

The default scheduling rule that we pick is ”guided”. Because this is the optimal scheduling rule to pick when the workload between each iteration is not even. Also, we try to test other scheduling rules including ”static” and 'dynamic' later. The histograms from Figure 6 to Figure 12 show the distribution of workloads between all the CPU cores when using guided scheduling and default chunk size (i.e. 1) on geb079_max50m.graph dataset.

Octomap components	Dataset 1	%	Dataset 2	%	Dataset 3	%
Data set size (total # of point clouds/points)	66 / 5903426		81 / 20073185		92631 / 14468149	
Compute ray tracing (insert to 2 hash tables)	9.878	55.169	157.43	30.2	456.765	37
Compute ray tracing (purely)	(2.600)	33.36	(70.114)	13.45	(123.079)	9.97
Hash table insertion & duplication check	(7.278)	40.65	(87.316)	16.75	(333.685)	27.03
Insert to OcTree	1.594	8.9	271.073	52	591.3255	47.9
Total run time*	17.905	100	521.294	100	1234.5	100

Figure 4: The run time decomposistion of Octomap over three open sourced datasets. These 3 datasets have different features but have the same resolution when constructing the OcTree. The first dataset scans a corridor while the second and the third datasets covers two whole campuses. We profile the run time of each component using Intel Vtune.

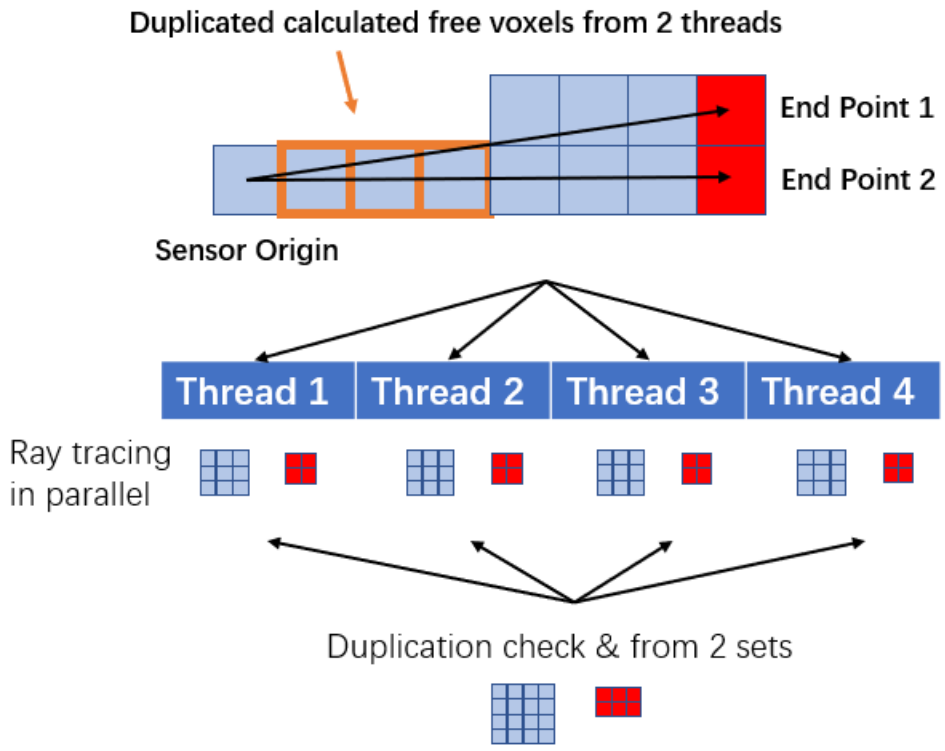

Figure 5: When using multiple threads to perform ray-tracing, the computation can be done in parallel. However, they have to merge into two sets finally and check all the duplication. In the source code, the 2 sets are realized in hash tables to minimized the duplication check cost.

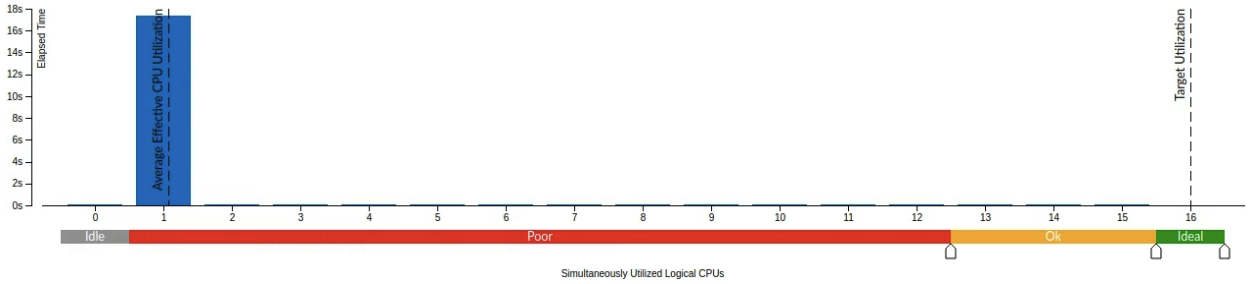

Figure 6: thread_num=1

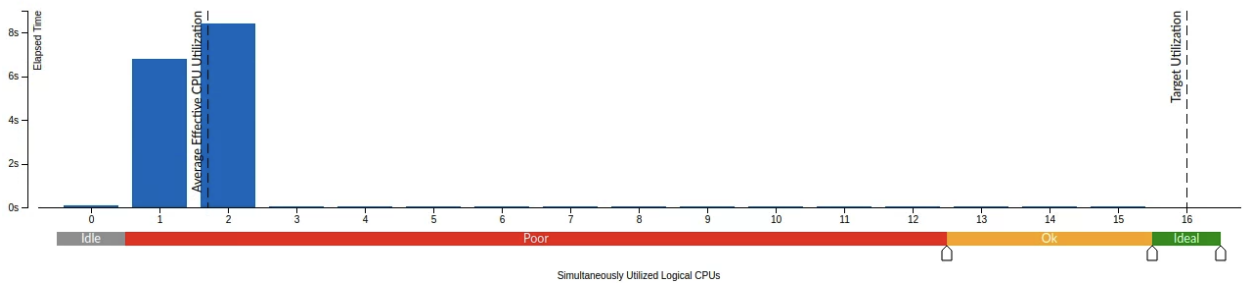

Figure 7: thread_num=2

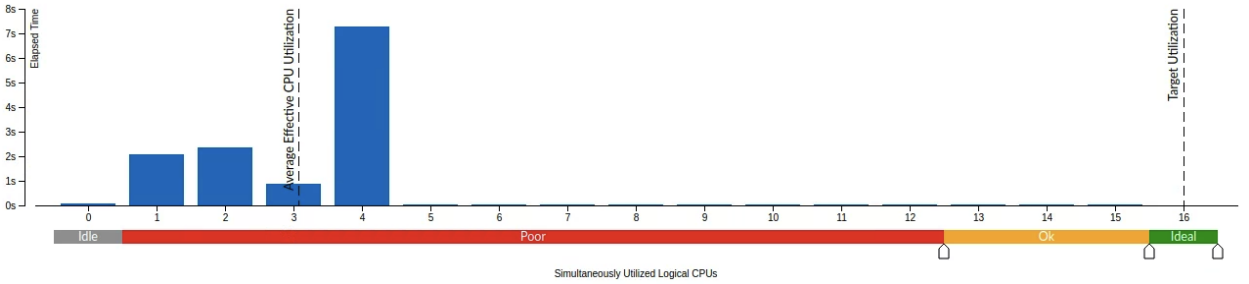

Figure 8: thread_num=4

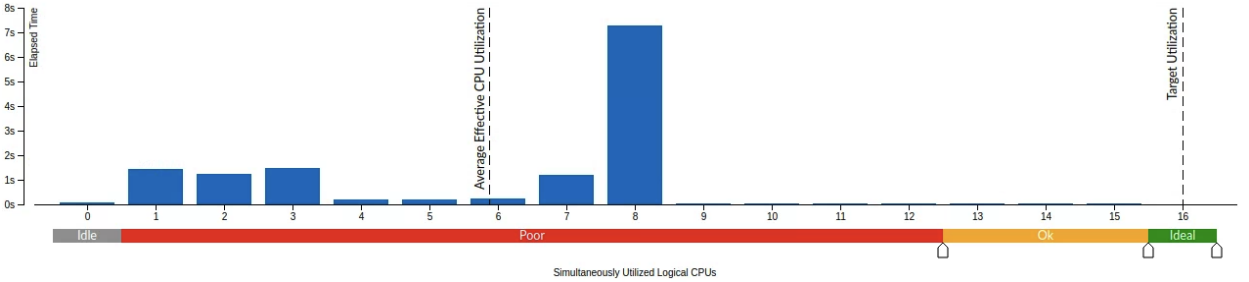

Figure 9: thread_num=8

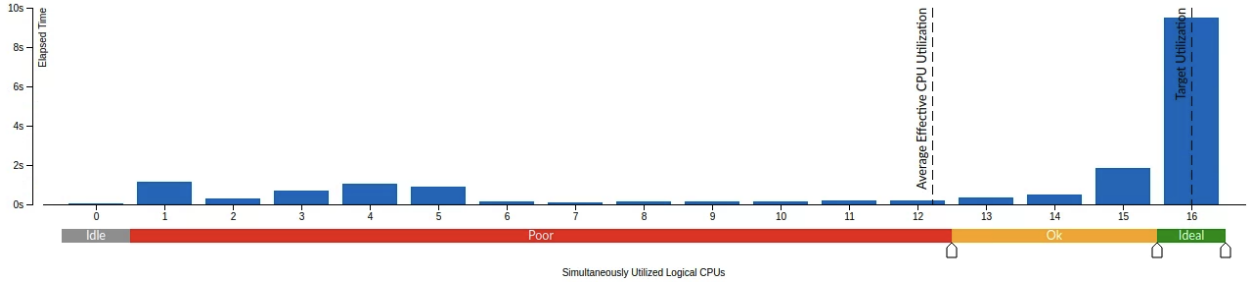

Figure 10: thread_num=16

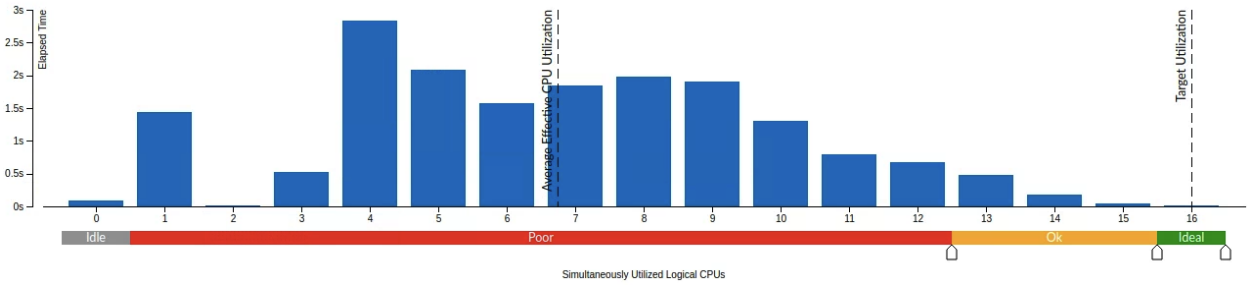

Figure 11: thread_num=32

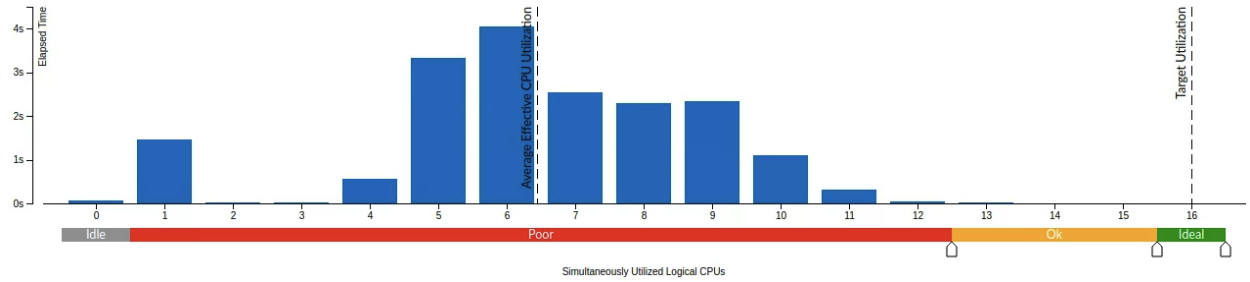

Figure 12: thread_num=64

Figure 6 to Figure 12 are the Effective CPU Utilization Histogram. These histograms display a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and overhead time adds to the ideal CPU utilization value.

Note that we run each thread on each CPU core if the number of threads is less than the number of cores. Also, we run experiments where the number of threads is bigger than the number of cores. This means that two or more cores are scheduling on the same CPU core.

Here we can see a non-even distribution of run time over each CPU core. Notice that there is one CPU core with rather high overhead. This is the core that runs all serial tasks including inserting updates into the back-end data structure Octree and reads in all the data from the file. The total run time for each settings can be seen in Figure 13. Setting $thread\_num = 4$ gives us the shortest elapsed time.

Figure 14 and Figure 15 show the run time of the ray-tracing task as well as the whole mapping system when using different number of threads with default scheduling rule that we pick is ”guided”.

We also experimented with different chunk size of 1, 2, 4, 8, 16, 32, 64 on the *fr_campus* dataset. We set *thread_num=4* to experiment, which is the ideal setting. See Figure 16 and Figure 17 for results. We can see that selecting different chunk_size does not seem to affect the ray tracing time and total runtime that much.

In the end, we experiment with *static* and *dynamic* scheduling rules over the same geb079_max50m dataset to examine the corresponding performances. Same to the setting above, we test the run time of the ray-tracing component and the whole system. See Figure 18 and Figure 19. We can see that static schedule gave worst performance on both ray-tracing and total system runtime. Dynamic schedule achieves comparable results with default guided scheduling with chunk_size=1.

Thread_num	Total Run Time (s)
1	17.3919
2	15.2238
4	12.6665
8	13.3047
16	17.5703
32	17.7016
64	18.1082

Figure 13: Total System Runtime using different thread_num on geb079_max50m dataset

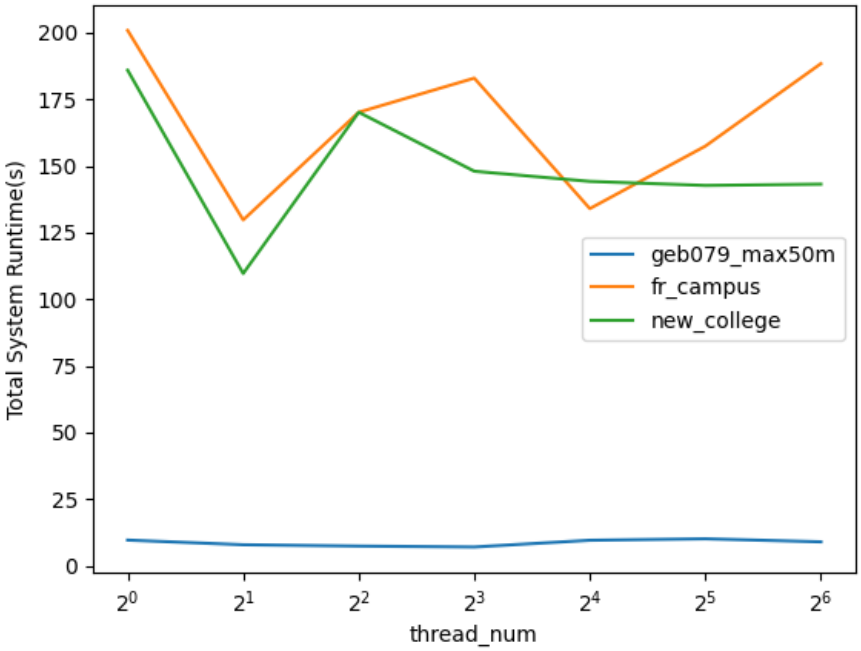

Figure 14: ray-tracing runtime (s)

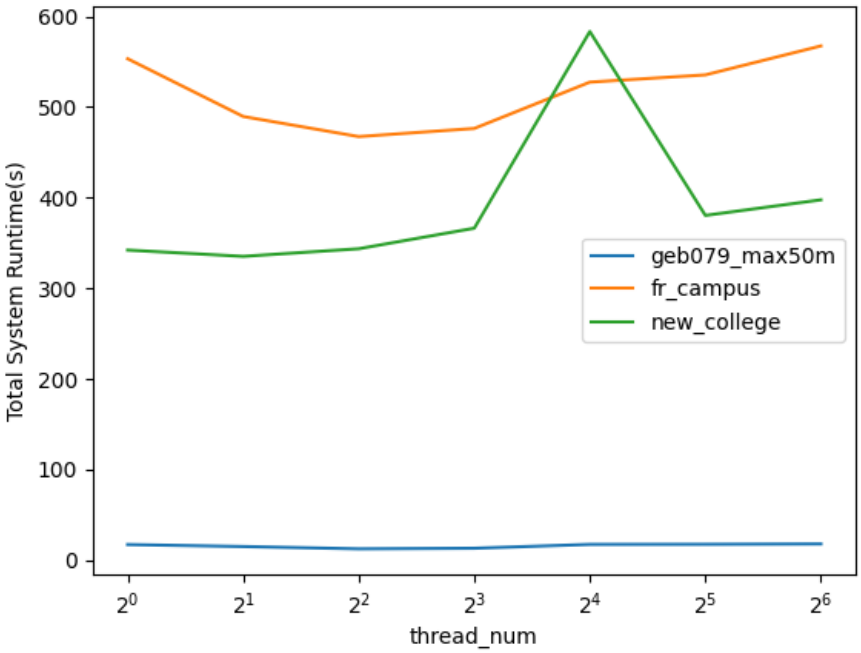

Figure 15: Total system runtime

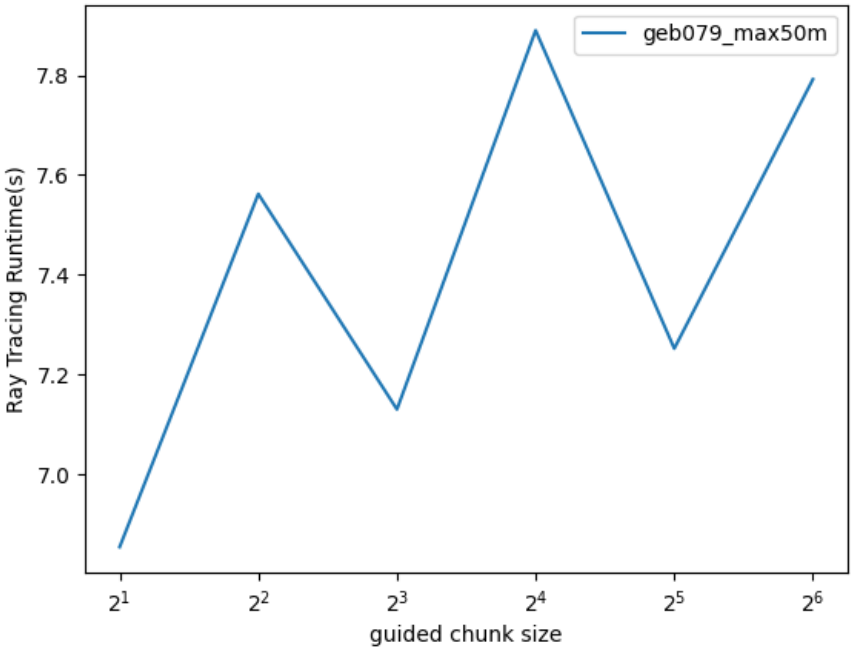

Figure 16: Ray Tracing Runtime

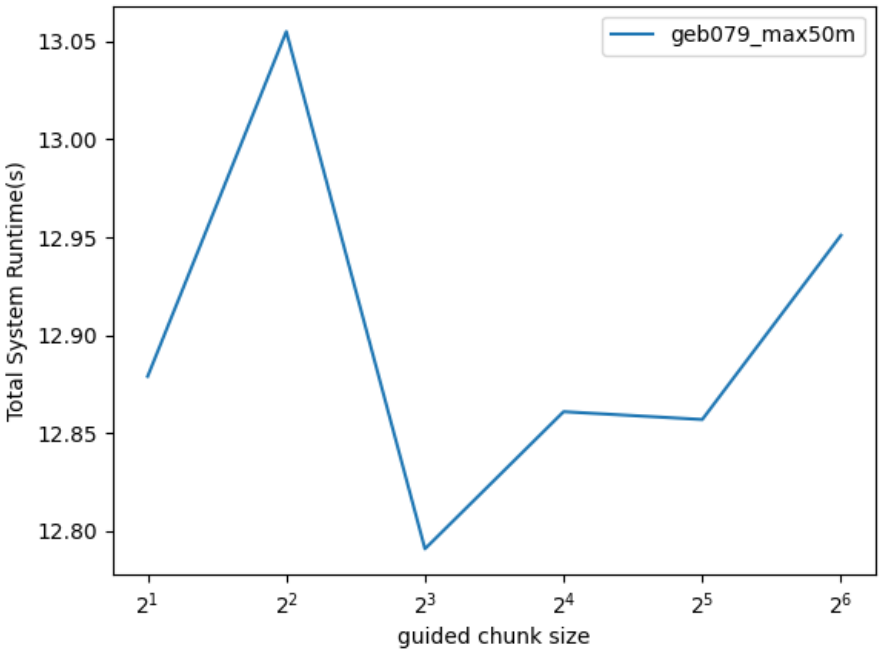

Figure 17: Total system runtime

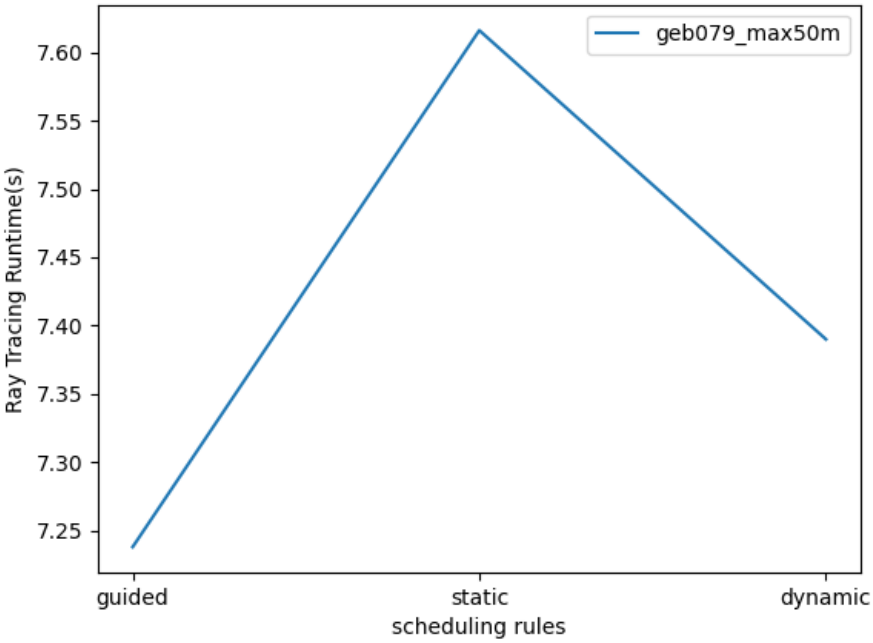

Figure 18: Ray Tracing Runtime

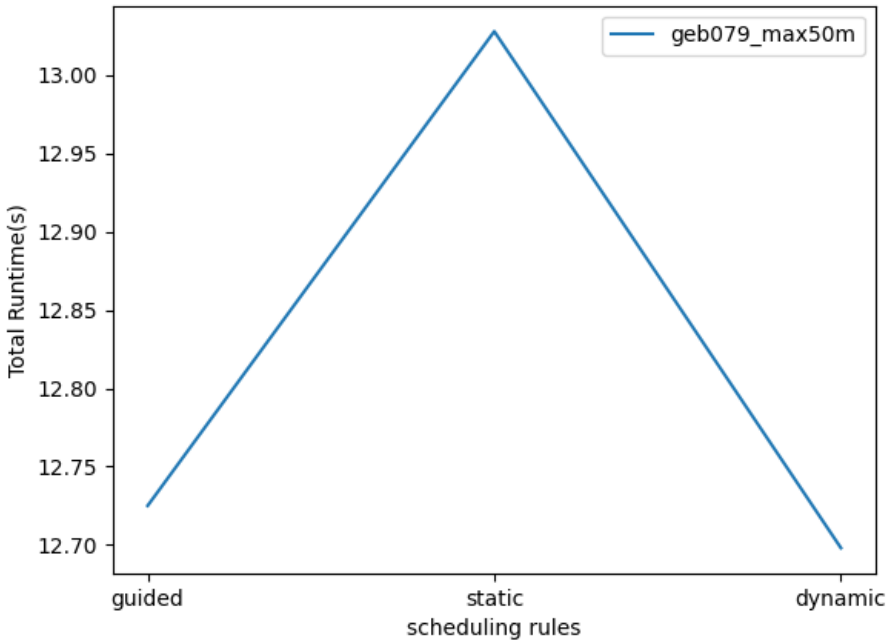

Figure 19: Total system runtime