15418 Final Project: Parallelizing Ray Tracing

Owen Wang
Jim Liu

Summary:
In our project we investigated numerous methods to parallelize a ray tracer in order to maximize both performance and quality of the render. Our two goals were to be able to render an acceptable quality scene at real time frame rates and also to minimize the render time of large, high-quality scenes. Starting from a CPU ray tracer, we implemented a CUDA based GPU ray tracer and augmented it to run on multiple GPUs using MPI. We also ported it onto OpenGL and added a parallelized denoiser to improve image quality at low sample counts.

Background:
The input for our project is 3D world information and the output is the ray-traced image render. We stored the world information in a BVH tree. In order to calculate ray hits, we must traverse the BVH tree. The traversal of the ray is the most costly operation in our ray tracer, with increased complexity on a GPU.

For the image, we do operations per pixel, which are the smallest grain for parallelism. The rays for ray tracing are sampled for every single pixel and the kernel that we applied to the image for the denoising is at a pixel granularity. The per pixel workload and denoise filter is highly parallelizable as the work done for each pixel is independent of work being done on any other pixel at the same time step.

Approach:
For our starter code, we used an online ray tracing project written in C++: Ray Tracing In One Weekend. The original sequential algorithm iterated through the pixels. For each pixel, it would first sample a point in the pixel then trace one ray for a depth of up to 50 bounces. Increasing the number of samples per pixel results in lower noise and a more defined image. From this algorithm, we decided the first way to parallelize would be via GPU execution by launching a thread per pixel. In addition, we added changes to the data structures for CUDA device code and also implemented the BVH tree traversal on GPU. We optimized the BVH tree traversal using a per-ray traversal method and added persistent threads.

The per-ray traversal method is for improving BVH tree traversal on GPU. It is used to decrease the block time caused by the threads in a GPU warp running different branches of code. This occurs while traversing the BVH tree when threads in the same warp decide to follow different child nodes. Suppose thread 0 follows the left child and thread 1 follows the right child, then thread 1 has to wait for thread 0 to finish the traversal so that it can start and is blocked in the meantime. This is a big waste of thread time. Therefore, the per-ray traversal method is used. It maintains a stack to store the next node to visit. In this case, the thread 1 in the above example

will just need to wait thread 0 to push the next node to the stack instead of doing the whole traversal.

The persistent thread is added for a better task scheduling. The GPU task dispatcher is not very suitable for the task where the execution time varies significantly. The dispatcher will feed a single SM with tasks unless the SM's running context or local memory is fully filled up. In this case, if one or two tasks on one SM are taking a long time, other tasks which have already been assigned must wait for them to finish before starting. Finally, after all tasks have been dispatched, there may still be several long tasks running on one SM, while another SM may be idle. As assigned tasks cannot be reassigned to another SM, starvation occurs. The persistent thread method dispatches enough tasks only to fill up all the compute units on SM. And at the same time, we maintain a global task queue for tasks to be grabbed from. In effect, we transform task assignment into finer grained dynamic assignment. We also batch tasks together so that the dynamic assignment is not too fine grained as to create too much overhead.

We also implemented an edge-aware A-trous denoising algorithm to improve image quality for real time rendering. A-trous denoising requires a normal and position buffer in addition to the color buffer for the final ray traced image. Similar to most image processing algorithms, we apply a kernel to the whole image. The A-trous blurring kernel uses the color, normal, and position buffer as the weights to avoid edges, blur out noise and while still retaining image edge sharpness.

Furthermore, we augmented the CUDA implementation with two MPI approaches. The first approach we used was to distribute an instance of the problem to every single processor. This means that each processor will render the entire image. The final result is then collected and averaged. This effectively increases the number of samples per pixel but maintains the same render time as one processor.

Another approach was to split the work of rendering one image across different processors. We currently use a static interleaved schedule where processors work on rows of blocks. This method will reduce the render time of a single scene and may be used to increase performance for the purposes of real time rendering.

Results:

We use the following performance performance measurements:

- Speedup
- Render time
- Render Quality

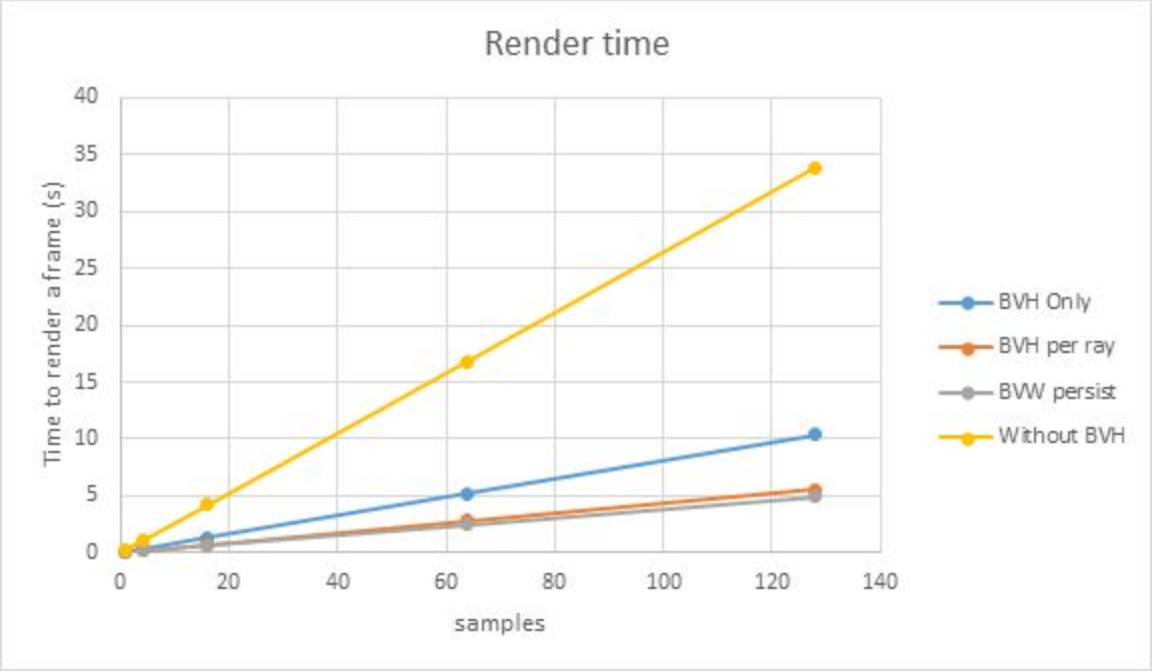
Experimental Setup:

- CPU (one thread): Intel i7-8700, GPU: GTX 1080

    - We used this machine for single GPU testing.
- MPI tests: Bridges machine, 4 x K80 GPU
    - The bridge cluster was used for multi-GPU tests.

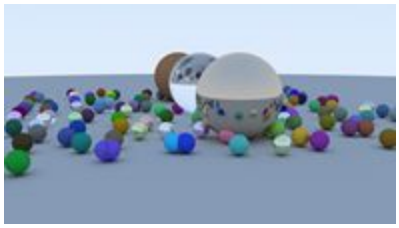
Performance Analysis:
Our method improves the performance significantly in comparison to the original CPU of ray tracer (236x speed up for one GPU).

We further recorded and compared the performance of the ray tracer using different optimization methods based on the GPU version. As the CPU ray tracer takes considerably more time (36x, graph scaling is not appropriate), we do not include the CPU ray tracer result here. (GTX 1080)

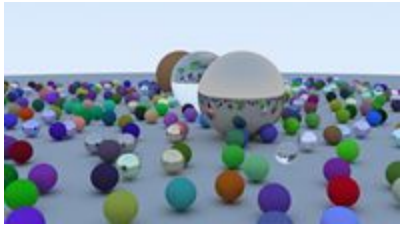


As we can conclude from the graph, the render time scales linearly with the samples. This is expected, because the average task time for every sample will not change when sample count increases. The large total sample count means that increasing the sample count further will not change the balance of work distribution.

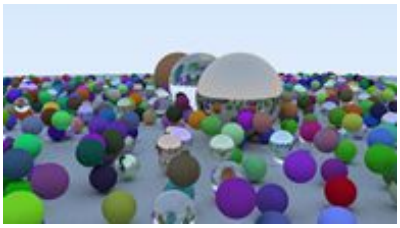
We record the speedup of each method with different scene complexity.



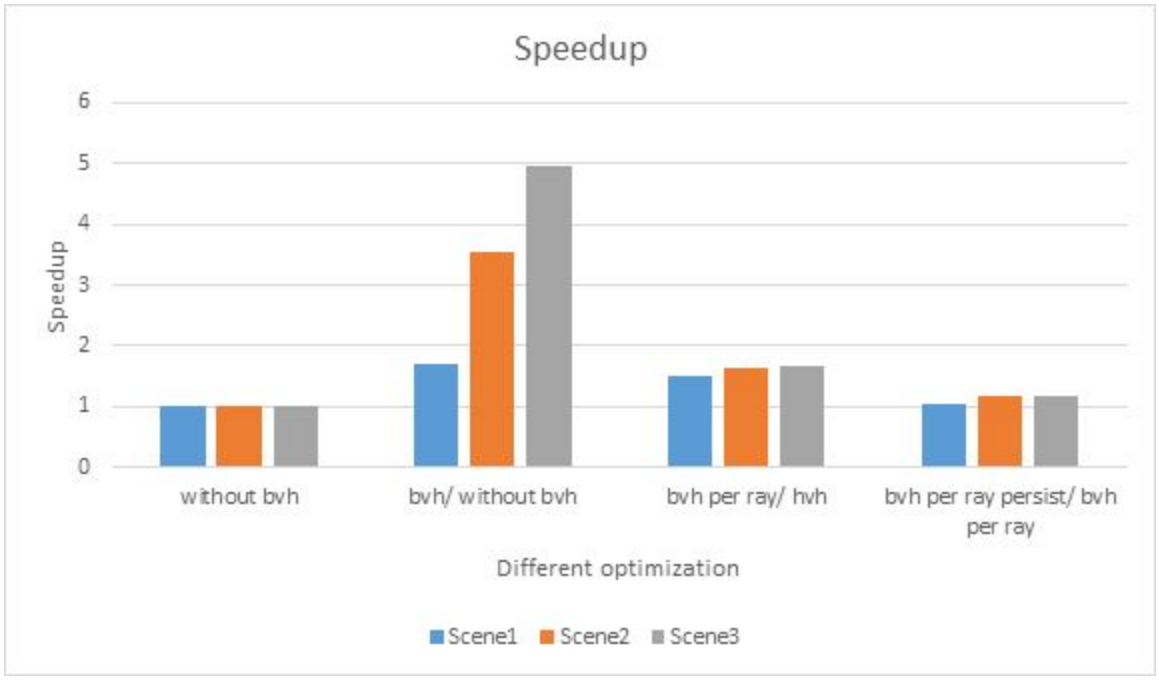
Scene1



Scene2



Scene3



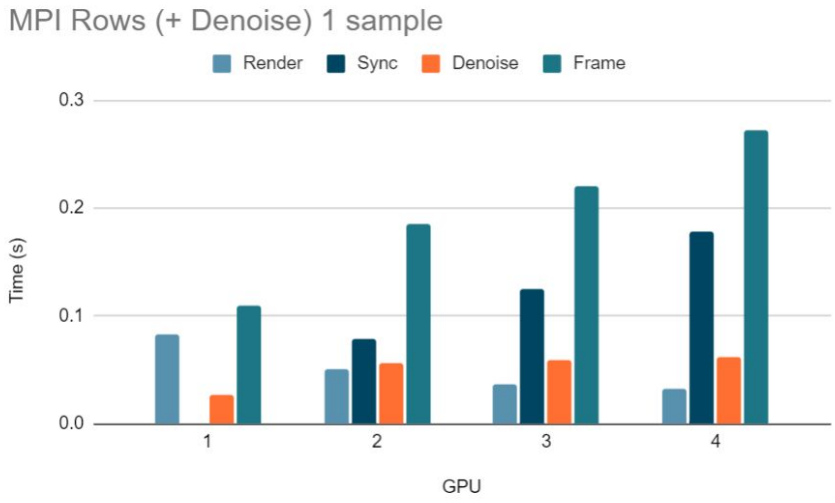
From the result above, the three optimization methods all improve the performance compared to our reference GPU implementation. More interestingly, the performance sometimes improves when the scene gets more complex. The major factor here is the BVH tree as it takes the traversal of the world from O(N) to O(log N). Furthermore, we can also observe a small performance improvement of implementing the per ray optimization as the scenes become more complex. This improvement is because the BVH is getting more complex when the scene gets complicated.

We notice an inevitable limitation for the implementation here. Although we make the BVH traversal more parallelized, the tasks between the threads may still be very unbalanced. The rays in one warp may terminate at different times because they may have a different number of bounces.

With our MPI approaches, we had two different objectives in mind. The first objective was to reduce the render time to the absolute minimum in order to maximize the frame rate. The second objective was to reduce the render time in general for large, high-quality scenes.

Our first approach was to parallelize by dividing the work of rendering one scene across multiple GPUs. In theory, we would get a linear speedup with an increasing number of processors as the work is divided up. The most appropriate method we found was using a static scheduler by interleaving rows. The interleaved approach ensures an approximately equal distribution of work given varying scene complexities. A dynamic scheduler would've provided little benefit with significant overhead both in terms of implementation and performance. Evidence of this was render time decreasing linearly. Furthermore, a static assignment of contiguous rows would also not be appropriate as there is large variance in scene complexity. For example, the top quarter of the image almost entirely consists of background.

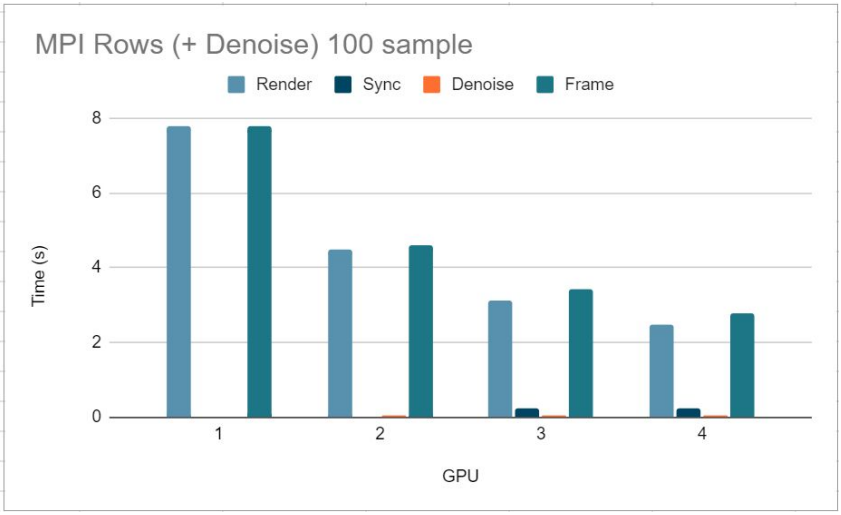
While the approach was overall good, it turned out to not be as optimal as we had hoped for our target purpose of real time rendering. For real-time rendering, we hoped to use a small sample count (~1) and make up for the quality via denoising techniques. This was to minimize the computation time required in order to achieve higher frame rates. A significant bottleneck came in the synchronization stage of the problem. With a small sample count, the total execution time was dominated by the MPI_Allreduce to combine the output buffers after rendering.



In this chart, each bar represents the time for the specific component of execution. Sync is the time for collecting the buffers from different processors from MPI_Allreduce and frame is the total time to render the frame.

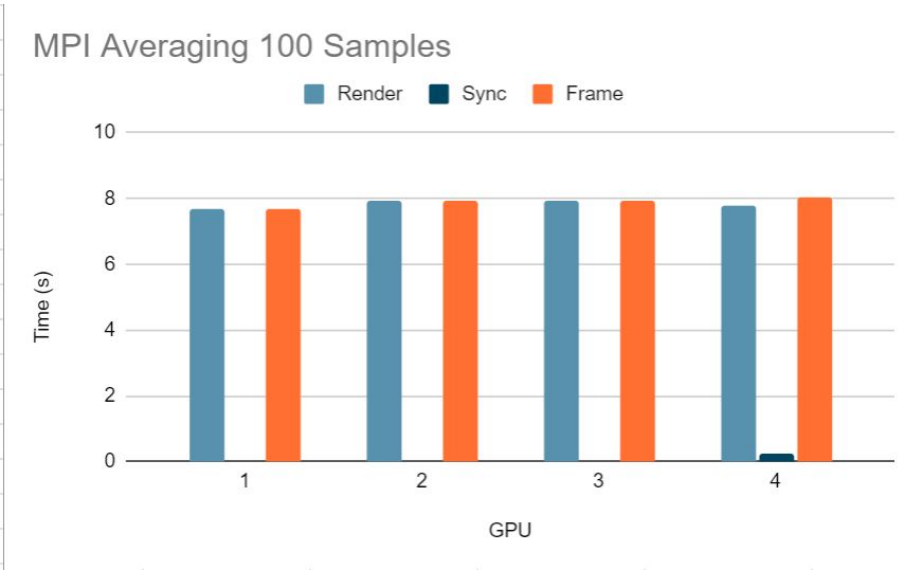
We notice that despite the rendering time decreasing as expected with increasing GPU numbers, it is offset by the increase in synchronization time. This results in increasing frame times with the number of GPUs.

With higher sample counts, we see that the contribution of synchronization time decreases significantly as rendering time increases. For 100 samples and 4 GPUs, the sync time was only around 0.28 seconds. Here we see the intended benefits of dividing the work up by rows. The scaling is approximately linear with the number of graphics cards.



An interesting observation is that the synchronization time at small samples actually increases linearly with the number of processors. With an optimal parallel binary reduction we would expect the synchronization to take O(log n). This is an area which warrants future investigation. Ultimately, the method of dividing the scene by rows only makes sense if we render greater workloads where the render time dominates.

The second approach we investigated was having each GPU render an entire image and then averaging the results. This yields a greater effective number of samples per pixel while theoretically maintaining the performance with an increasing number of processors. We can observe this in the following chart.



As we can see, the total time to render each frame remains approximately constant while the number of processors increases. The increasing processors yields a greater image quality (lower noise).

Noise reduction from 1,2,4,8 processors by averaging. Results are almost identical to increasing sample count.

In comparison to the rows method, the performance of the averaging method is better as the same amount of effective computation (total pixels rendered). This is expected as there is better workload balance as the work of each processor is exactly the same. Performance scales nearly perfectly with GPU count.

Conclusion:

Overall, we were successful in achieving our goals. The improvements made in single GPU performance was significant and allowed us to produce a real-time rendering of the scene. We demonstrated an additional vector of parallelism through MPI and multiple GPUs which allows the rapid, scalable rendering of ray-traced scenes on large, high-quality scenes.

References:

- Ray tracing In one weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html
- Dammertz, Holger, et al. "Edge-avoiding À-Trous wavelet transform for fast global illumination filtering." *Proceedings of the Conference on High Performance Graphics*. 2010.
- Aila, Timo, and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs." *Proceedings of the conference on high performance graphics 2009*. 2009.

Work Distribution:

Both: GPU ray tracing base/ denoise/ report
Jim: BVH/ per-ray traversal/ persistent threads
Owen: MPI averaging parallel/ MPI rows parallel/ MPI denoise
Credit distribution: 50%/50%