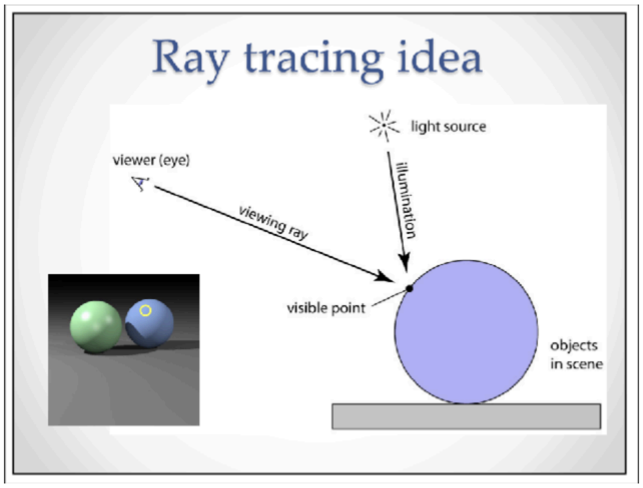
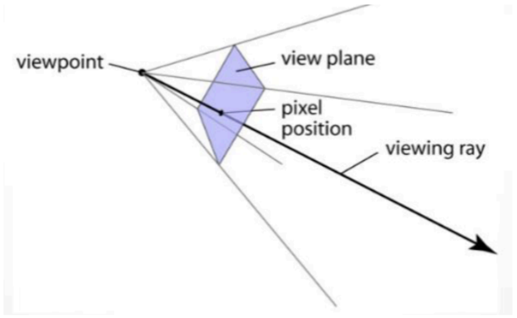
PARALLELIZATION OF RAY TRACING

Introduction :

Ray tracing is a physically inspired rendering technique which simulates the behavior of light by tracing rays from a camera through each pixel of an image plane and into a 3D scene.



The main idea of ray tracing is where rays are casted from the viewer through the image plane into the scene to determine visible surfaces, at the point of intersection, illumination from light sources is evaluated to compute the final color based on material and lighting interaction.



In the above figure we can see how viewing rays will originate from the camera viewpoint and pass through individual pixel positions on the view plane, each ray defines a unique direction into the scene, enabling independent computation of color for every pixel.

The concept is for each and every pixel, a primary ray will be generated and tested for intersection against all geometric primitives in the scene like spheres, triangles etc.. . If an intersection occurs, the renderer computes the local illumination.

Problem Description :

The problem addressed in this project is the implementation and optimization of a ray tracing issue with a strong emphasis on parallel execution. In its naive/serial version ray tracing scales poorly as image resolution increases or as scene complexity grows, since the total number of rays increases with the number of pixels, furthermore each ray may test against every object in the scene making the serial algorithm take more time for harder problems. This combination makes ray tracing an ideal case for parallelization.

From a parallel computing perspective, ray tracing is interesting because it exposes multiple levels of parallelism. At the highest level rays corresponding to different pixels are completely independent and can be processed concurrently without any synchronization, additionally recursive secondary rays generated due to reflection and refraction introduce irregular workloads across pixels wherein some arys terminates early others involve deeper recursion further motivates parallel workload distribution.

Parallelization Strategies :

In this project, a recursive ray tracer supporting sphere, triangles (normalized as well), reflection and refractions was implemented and incrementally optimized using parallelization strategies. The baseline implementation performs ray generation sequentially over the image grid, parallel execution using would be OpenMP, MPI and CUDA, along with performance oriented refinements such as loop restructuring, reducing redundant computation, improving handling of recursion in ray tracing depth, and algorithm optimization such as Bounding Volume Hierarchy (BVH). The goal is not only to demonstrate speedup but also to analyze how different design decisions affect performance, scalability and efficiency.

OpenMP Implementation :

For OpenMP implementation, we used shared memory parallelism to speedup the same pixel shading loop without changing the ray tracer's core logic. Rendering Rendering is serial at the pixel level, each pixel's ray can be generated, traced, and shaded independently, so the natural strategy is to parallelize the nested image loops and let OpenMP schedule pixels across CPU threads and OpenMP runs inside a single process and all threads share the same Scene and Image objects.

For OpenMP, we first disable dynamic thread adjustment using omp_set_dynamic(0) so the runtime doesn't change the thread count mid-run. Then we controlled threading through the environment variables rather than hard coding in the program, Before running we need to set OMP_PROC_BIND=spread and OMP_PLACES=cores to pin threads to physical cores and distribute them evenly, which reduces thread migration and improves run to run stability. We then set the OMP_NUM_THREADS (ex., 1,2,4,8) from the command line instead of setting in program to choose how many threads OpenMP should use and then the nested pixel loops are parallelized using #pragma omp parallel for collapse(2), allowing OpenMP to divide the combined (i, j) iteration space across the available threads for balanced workload and fair benchmarking.

This shows OpenMP to collapse the two nested loops into combined iteration space size (image_width * image_height) giving the scheduler a larger pool of independent iterations to distribute evenly across the available threads. Each iteration computes the pixel-center ray like the serial code, it derives (u, v) from (i, j), constructs the point on the image plane using the camera basis and then builds a Ray, and calls the existing rayTrace routine with scene.max_depth. The result is written directly into outputImg.getPixel(i, j), because each (i,j) pair maps to a unique pixel and there is no write-write conflict between threads since each thread writes to different pixels. The Scene is read-only during rendering so it is safe for us to share across the threads without any locks.

Overall, the OpenMP version preserves the original ray tracer structure and accelerates rendering by parallelizing the nested pixel loops via collapse(2) while keeping scene parsing, ray generation, shading, and image output unchanged.

MPI Implementation :

For the MPI Implementation we parallelized the ray tracer over the image rows, rendering is embarrassingly parallel at pixel level, each pixel can be shaded independently, hence we are only modifying main.cpp to distribute rows among the MPI processes and to gather the final image.

At the start of the program we are calling MPI_Init which will query the number of processes world_size and the rank world_rank and then let each rank parse the same scene file using the existing parseSceneFile function. Replicating the scene on each of the processes avoids the need for explicit communication of materials and keeps the inner ray-tracing loops identical to base implementation.

We use a block row partitioning :

- Consider H be the image height and P be the number of MPI processes.
- We can compute base_rows = H / p and remainder = H % P
- Rank r is assigned local_rows = base_rows + (r < remainder ? 1: 0)

This will ensure that all rows are covered and that the first remainder ranks receive at most one extra row so thereby the work distribution is balanced even when H is not divisible by P.

For each local row ‘j’, a rank is computed for the camera position of the respective pixel center using the same camera model as the serial code, then it traces one ray per pixel by calling existing rayTrace function/flow. To make gathering simpler, each rank stores it’s parallel image in a flat array of floats.

local_pixels[(local_row * width + i) * 3 + {0,1,2}] = (r,g,b)

Here the color struct uses r,g,b which we cast to floats before communication to reduce message size.

Once all local rows have been rendered, we measure the local elapsed time using MPI_Wtime, then compute the global wall-clock time as the maximum ranks.

MPI_Reduce(&local_ms, &global_ms, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

To reconstruct the full image, rank 0 allocates a receive buffer all_pixels and calls MPI_Gatherv with per-rank counts and displacements computed from the row partition:

- recvcounts[r] = rows_r * width * 3
- displs[r] = start_row_r * width * 3.

After the gathering, rank 0 writes the pixels back into the Image object and saves the final image file. All ranks then free their local copies of the scene and call MPI_Finalize.

Overall, MPI Implementation preserves the original ray tracer’s structure and concentrates on parallelizing the main function, per-row assignment, timing with MPI_Wtime and single gather at the end.

CUDA Implementation :

In the CUDA implementation, we took advantage of the massive parallelism of the GPU by assigning a single thread to compute the color of each pixel independently. Unlike the loop/row parallelization of CPU approaches (OpenMP and MPI), for CUDA we mapped the execution directly to the 2D image grid.

A primary challenge in this transition was the refactoring of our core data structures. Because GPU kernels cannot efficiently handle virtual functions or runtime polymorphism, we had to replace our class hierarchies with flattened, "plain old data" (POD) structures. We eliminated the use of pointers within structs to ensure that all data could be moved as contiguous blocks between the host and device. Instead of using virtual methods for different object types (like Spheres or Triangles), we utilized type-tagging and switch statements within the kernel to handle intersection logic.

We started by parsing the scene file on the host and allocating the scene data (primitives, materials, lights) directly onto the GPU device memory (DeviceScene). This enables low-latency, read-only access to the scene description during the ray tracing process, without needing to communicate back and forth with the host CPU. We also allocate a device buffer (d_output_img) using cudaMalloc to store the resulting pixel colors.

We configured the kernel execution using a 2D grid of thread blocks. Through performance tuning, we determined that blocks of 8x8 threads provided the most efficient balance between warp execution and cache locality for our ray tracing logic. The grid dimensions are calculated based upon the image height and width, making sure that every pixel is covered:

num_blocks = ((img_width + 7) / 8, (img_height + 7) / 8)

Inside the rayTracerKernel, each thread calculates its unique pixel coordinates (x,y) using the indexes of its block and thread. Threads mapping to coordinates outside of the image boundaries immediately return to handle arbitrary resolutions.

In the CPU implementations, recursion is extremely important and useful; however, recursive ray-tracing is not well-suited for GPUs due to limited stack space and the risk of stack overflow. We replaced the recursive algorithm with an iterative loop to improve performance and stability.

Instead of recursive calls returning color values up the stack, we maintain an accumulated_color and a throughput vector (initially set to 1,1,1). In each iteration of the depth loop, if a ray hits an object, the local illumination is calculated and added to the accumulated_color, weighted by the current throughput. For reflection or refraction, the throughput is modulated by the material's specular or transmission properties, and the ray is updated for the next iteration.

To accurately measure the GPU performance (excluding host overhead), we utilized CUDA events (cudaEventRecord) around the kernel launch. Once the kernel execution and synchronization are complete, the final image data is transferred back to the host using cudaMemcpy for file output.

In short, our CUDA implementation restructures the rendering loop to fit the Single Instruction, Multiple Threads (SIMT) architecture of CUDA, prioritizing memory coalescing and iterative logic as opposed to the recursive structure of CPU parallelization.

CUDA-BVH Implementation :

While the linear CUDA implementation successfully parallelized the pixel by pixel workload, its workload was confined to a linear O(N) complexity when checking every ray against every geometric primitive. For more complex scenes that contain hundreds of thousands of primitives, this became computationally expensive. To fix this, we implemented a Bounding Volume Hierarchy (BVH) acceleration structure, reducing the intersection complexity to logarithmic time O(logN).

Host-Side Construction :

Due to the complexity of dynamic memory allocation on the GPU, we perform the BVH construction entirely on the CPU before rendering begins. We utilize a top-down construction strategy:

- Primitive Sorting: The scene primitives are separated into spheres and triangles, and a separate BVH is recursively built for each to simplify memory access on the GPU. We then assign each primitive a bounding box (Axis-Aligned Bounding Box, or AABB) and group them based on their spatial centroids.
- Hierarchy Generation: a tree structure is built where every internal node contains an AABB that fully encloses the AABBS of its children. This continues until a leaf node is created, which contains a small, manageable number of actual geometric primitives (typically 1-4)

Linear BVH (LBVH) for GPU Optimization :

A standard pointer-based tree structure is inefficient for GPUs due to memory latency and the lack of a traditional heap. Therefore, the pointer-based tree is converted into a Linear Bounding Volume Hierarchy (LBVH), a flattened array-based representation optimized for the GPU's memory architecture.

- Flattening: The tree nodes are laid out in a contiguous standard std::vector (later copied to a struct array on the device) using a depth-first layout. THis ensures that child nodes are often located near their parents in memory, improving cache locality during traversal.
- Index-Based Linking: Instead of pointers, nodes reference their children and primitives using integer indexes. This structure lacks pointers, which allows the entire tree to be uploaded as a single contiguous memory block (cudaMemcpy) to the GPU global memory, enabling fast read-only access by all threads.

Device-Side Traversal Kernel :

We replaced the for-loop in our FindIntersection kernel with a stack-based traversal algorithm.

- Intersection Logic: For each ray, the kernel sequentially traverses the Sphere BVH and then the Triangle BVH. It tests the ray against the current node's AABB.
    - Miss: If the ray misses the box, the entire subtree is ignored, saving thousands of unnecessary calculations.
    - Hit: if the ray hits the box, the kernel traverses to the child nodes.
- Local Stack: Since GPU threads cannot easily recurse, we implemented an iterative traversal using a short, thread-local stack (stored in registers or local memory) to keep track of nodes to visit. When a thread processes an internal node, it pushes the children onto the stack (Right then Left) to ensure a consistent depth-first traversal of the tree structure.
- Leaf Processing: Once a leaf node is reached, the kernel performs the precise ray-triangle or ray-sphere intersection tests on the primitives contained within.

This architecture fundamentally shifts the bottleneck from arithmetic computations (solving intersection equations) to memory bandwidth (traversing the tree). However, because the BVH ignores typically 95-99% of the scene geometry for a given ray, the reduction in total operations outweighs the cost of the traversal logic. This implementation allows the renderer to scale efficiently with scene complexity, turning scenes that previously took 30+ seconds to render into tasks that complete in milliseconds.

Experimental Setup :

All CPU-based experiments were conducted on the university lab plate server (csel-plate01.cselabs.umn.edu), which is equipped with an Intel Xeon Phi 7290F processor featuring 72 physical cores and 288 hardware threads(using accordingly required threads) running at 1.50 GHz. The system provides approximately 94 GB of main memory making it suitable for shared-memory parallelism using OpenMP as well as MPI execution. The ray tracer was compiled using g++ version 13.3.0, with OpenMP 4.5 support enabled for parallel CPU execution.

All MPI experiments in this project were run on a single plate node (e.g., csel-plate01.cselabs.umn.edu). We launched up to 64 MPI ranks per run using `mpirun -np 64 ./ray_mpi <scene>` without specifying a hostfile, so all ranks were placed on the same host. Therefore, the reported MPI performance reflects single-node (intra-node) parallelism only, without inter-node communication.

GPU experiments were performed on a university lab CUDA server (csel-cuda-04.cselabs.umn.edu) equipped with a Tesla T4 GPU with 16 GB of device memory. This system also includes dual Intel Xeon Gold 6148 CPUs and approximately 93 GB of system memory. GPU implementations were compiled using the NVIDIA CUDA Toolkit version 12.9, and CUDA was used as the primary framework for GPU parallelization.

Standard C++ libraries were used across all implementations, along with OpenMP, MPI, and CUDA depending on the execution model.

These specifications are obtained using below commands in plate and cuda servers.
1. lscpu
2. free -h
3. g++ --version
4. echo | g++ -fopenmp -dM -E - | grep -i openmp
5. nvidia-smi
6. nvcc --version

Kindly refer README for how to execute the test for each implementation (Baseline, OpenMP, MPI and CUDA)

Results and Analysis :

Baseline Results and Analysis :

All experiments were run on the university plate-server compiled with g++ -O3 -std=c++17. Using the baseline serial version, the Dragon scene took several hours to render even though the output is relatively simple. Since the implementation is entirely single-threaded and does not use any acceleration structures or parallelism, ray–intersection tests are executed sequentially, resulting in poor performance for relatively complex scenes.

OpenMP Results and Analysis :

All experiments were run on the university plate-server using the OpenMP version compiled with g++ -O3 -std=c++17 -fopenmp. We used a single node and varied the number of OpenMP threads T = 1, 2, 4, 8, 16, 32, 64. Thread placement was controlled via environment variables:
1. OMP_PROC_BIND=spread
2. OMP_PLACES=cores
3. OMP_NUM_THREADS=T

Dragon Scene:
For the Dragon scene (Tests/InterestingScences/dragon.txt), the OpenMP version produced the expected image but required relatively long times. The timings are:

Thread Count (T)	Time (ms)	SpeedUp	Efficiency
1	2063492	1.00	1.00

2	1131562	1.823	0.911
4	913130	2.260	0.565
8	445789	4.629	0.578
16	254836	8.100	0.506
32	138610	14.887	0.465
64	69803	29.562	0.462

Note: SpeedUp and parallel efficiency are calculated considering T = 1 as baseline.

Observations :

- Scaling is linear with speedup and uneven with efficiency.
- Speedup continues to improve until 64 threads, but efficiency drops to ~46%, showing diminishing returns as threads increase.

Potential reasons for efficiency drops at higher thread counts :
Shared-memory contention (cache/memory bandwidth pressure), OpenMP scheduling overhead, and per-pixel workload imbalance (some pixels trace more rays / intersections than others), all of which become more visible as concurrency grows.

MPI Results and Analysis :

All experiments were run on the university plate-server using the MPI version compiled with mpicxx -O3 -std=c++17 (plus the same include paths and source files as the baseline). We used a single node and varied the number of MPI processes P=1,2,4,8,16,32,64 For each configuration, we report the maximum runtime across all ranks measured with MPI_Wtime, which approximates the wall-clock time of the slowest process.

Dragon Scene :

For the complex Dragon scene (Tests/ComplexExamples/dragon.txt), the MPI implementation produces the correct image and achieves substantial acceleration over the serial baseline. The detailed timings, speedups, and efficiencies (with P=1 as the baseline) are summarized in the table below. The runtime decreases from about 1.94×10^6 at P=1 to roughly 4.56× 10^4 at P=64, corresponding to a speedup of about 42.6 times and a parallel efficiency of approximately 66%.

MPI Processes (P)	**Time (ms)**	**SpeedUp**	**Efficiency**
1	1944829	1.000	1.000

2	1153598	1.686	0.843
4	594816	3.270	0.818
8	336976	5.771	0.721
16	174460	11.148	0.696
32	89497	21.730	0.679
64	45659	42.595	0.665

Note : SpeedUp and Parallel efficiency are calculated for considering P=1 as baseline.

As we increase the number of processes from 2 to 64, the total runtime drops by more than an order of magnitude. Doubling the number of processes from 2 to 4 and from 4 to 8 roughly halves the runtime, and increasing from 8 to 16, 32, and 64 continues to provide additional speedup, although with diminishing returns.

Observations :

- The Dragon scene exhibits good strong scaling: adding more processes consistently reduces the total runtime, especially between P = 2 and P = 16.

- At higher process counts, the speedup becomes sublinear and efficiency drops. This is expected due to MPI overheads (collective communication and synchronization) and load imbalance across rows: some rows contain more geometry and deeper ray recursion than others, so the slowest rank determines the overall runtime.

- Fixed costs such as parsing the scene file on every process and setting up MPI also become more visible as the amount of computation per process decreases.

Other scenes :

We also evaluated MPI on a variety of smaller and medium-sized scenes, including spheres1, spot_sphere, outdoor, bottle, and bottle-nolabel, as well as larger "interesting" scenes such as arm-reach, arm-top, gear, plant-h, watch-norefract, watch-bluegold, island, spaceship, lily, sword, cat, and character. For very small scenes like spheres1 and spot_sphere, MPI still reduces runtime from a few hundred milliseconds to tens of milliseconds when going up to P = 64, but the relative speedup is limited because MPI setup and communication overhead dominate. For moderately complex scenes such as bottle and bottle-nolabel, the runtime drops from tens of seconds at low process counts to a few seconds at high process counts. For the largest scenes (e.g., the watch-challenge scenes, island, and spaceship), the serial implementation either takes many hours or fails within the time limit, while the MPI implementation with P = 64 finishes within minutes to at most a few hours, making these workloads tractable on the plate-server.

Comparison with OpenMP :

By comparing the MPI timings with the OpenMP timings reported in the previous subsection, we see that both parallelization strategies achieve large speedups over the baseline serial implementation on the Dragon scene. OpenMP exploits shared-memory parallelism within a single process, while MPI distributes image rows across independent processes and then gathers the final image with MPI_Gatherv. In our experiments, the MPI configuration scales well up to P = 64 and provides competitive performance for the heaviest scenes, with its main limitations coming from communication overhead and load imbalance rather than lack of available parallel work.

CUDA Results and Analysis :

All experiments were run on the university plate-server using the CUDA implementation compiled with nvcc -O3 -std=c++17. The hardware utilized was an NVIDIA T4 GPU. For the CUDA implementation, we parallelized the rendering process by assigning each pixel to a unique GPU thread. We evaluated the performance by varying the block dimensions (B * B) to determine the optimal thread configuration for the kernel.

Dragon Scene:

For the complex Dragon scene (Tests/InterestingScenes/dragon.txt), the CUDA implementation achieved a massive performance leap over both the serial baseline and the CPU-based parallel versions (OpenMP/MPI). By offloading the intersection calculations, the most computationally expensive part of the ray tracer, to thousands of CUDA cores, the rendering time dropped from over 34 minutes (serial) to just a few seconds.

The following table summarizes the performance across different block sizes. Speedup and Efficiency are calculated using the Serial Baseline (T=1) from the original experiments (2,063,492 ms).

Block Size (BxB)	**Total Threads Per Block**	**Time (ms)**	**SpeedUp (vs Serial)**
8*8	64	390	5291x
16*16	256	513	4022x
32*16	512	554	3724x

Note: The reported time doesn't include memory allocation, Host-to-Device transfer of scene geometry, and Device-to-Host transfer of the final pixel buffer.

Observations:

- **Massive Acceleration:** The CUDA implementation provides a speedup of approximately **5291x** over the serial baseline. Even compared to the best MPI result (P=64), CUDA is roughly **117x faster**, demonstrating the efficiency of the GPU's SIMT (Single Instruction, Multiple Threads) architecture for embarrassingly parallel workloads like ray tracing.

- **Optimal Block Size:** We found that an 8 * 8 block size (64 threads) provided the best performance for this ray tracer. While larger blocks like 16 * 16 or 32 * 16 usually help hide memory latency through higher occupancy, the complex nature of our ray-intersection logic favors the 8 * 8 configuration due to increased spatial coherence and reduced warp divergence. In an 8 * 8 tile, the rays are tightly clustered; they are more likely to hit the same geometry or traverse the same paths in the scene, which keeps the threads within a warp synchronized and improves L1 cache hit rates for mesh data.Conversely, performance began to degrade at 16 * 16 and 32 * 16. In these larger configurations, the "divergence" between threads increases as rays spread across a wider area of the scene, where some rays can hit things and others don't, and the high register pressure required to store ray paths and intersection variables begins to limit the number of active blocks the SM can manage simultaneously. The 32 * 32 configuration failed to execute entirely. This is likely because it hit the hardware limit of 1024 threads per block while simultaneously exceeding the SM's physical register capacity. In a ray tracer, where each thread must track multiple vectors and material properties, the total register demand for 1024 threads often exceeds what the hardware can allocate for a single block, resulting in a black output image.

Comparison with OpenMP and MPI:

The CUDA implementation fundamentally changes the tractability of the ray tracer. While OpenMP and MPI were limited by the number of physical CPU cores and the overhead of communication/synchronization, CUDA scales across thousands of lightweight threads.

1. **Scalability:** MPI and OpenMP showed diminishing returns and falling efficiency at 64 threads/processes due to cache contention and load imbalance. CUDA manages load imbalance better through hardware-level warp scheduling, which can hide the latency of heavy rays (those requiring deeper recursion or more intersection tests) by switching to other active warps.
2. **Resource Utilization:** MPI required redundant scene parsing on every process, whereas CUDA parses the scene once on the CPU and uploads the structured data to GPU global memory, which is then accessed as a read-only resource by all threads.
3. **Productivity:** For "challenge" scenes like the *watch* or *spaceship* that took hours on MPI, the CUDA version consistently delivers results within a few seconds, making it the most viable implementation for high-resolution or high-complexity rendering tasks.

CUDA-BVH Results and Analysis:

The integration of the BVH acceleration structure delivered a significant performance leap from the linear CUDA implementation, transforming the project into a real-time renderer rather than an interactive one.

As shown in the discussion table, the speedup provided by BVH correlates directly with the geometric complexity of the scene:

- Dragon Scene: The linear CUDA implementation rendered the Dragon in 390ms. With BVH, this dropped to 12.54 ms, yielding a 31x speedup on top of the original CUDA acceleration over other implementations.
- Island Scene: This scene represents a stress test with high geometric density. The linear CUDA implementation took 31,170 ms (around 31 seconds) due to the massive number of triangle tests per ray. The BVH implementation rendered the same scene in just 16.53 ms, achieving a 1,885x speedup. This demonstrates that for complex scenes, algorithmic efficiency of O(logN) is extremely important.

Comparison to Other Methods:

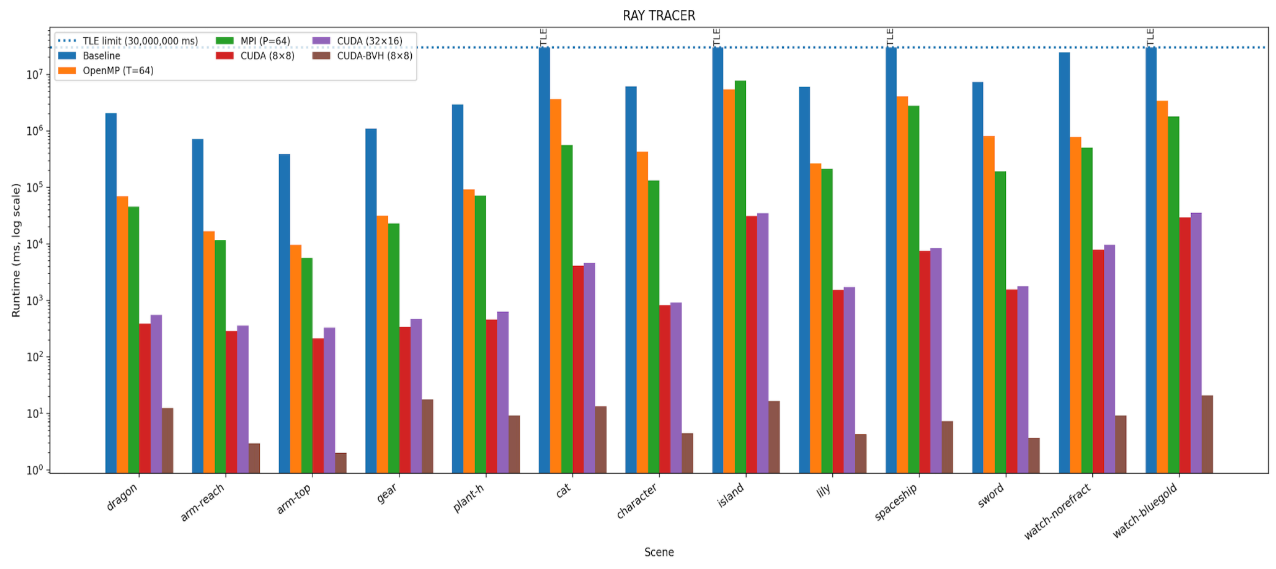
The switch from CPU (OpenMP/MPI) to GPU (linear CUDA) provided orders of magnitude improvement (reducing hours to seconds), the addition of BVH reduced runtimes from seconds to milliseconds.

- Scenes like plant-h.txt and watch-bluegold.txt, which required nearly an hour on the serial baseline and several seconds on the linear CUDA algorithm, now run at interactive frame rates (>60 FPS).
- The results confirm that CUDA provides the necessary throughput, and BVH provides the necessary algorithmic scalability to handle large-scale production scenes.

Discussion :

Scene	Baseline	OpenMP (T=64)	MPI (P=64)	CUDA (8*8)	CUDA (32*16)	CUDA-BVH (8*8)
dragon.txt	2074182	69803	45659	390	554	12.54
arm-reach.txt	716152	16784	11691	289	361	2.96
arm-top.txt	390928	9666	5689	215	330	2.00
gear.txt	1088584	31772	23030	340	467	17.65
plant-h.txt	2941415	91596	71387	462	632	9.24
cat.txt	TLE	3644770	564539	4131	4610	13.27
character.txt	6172546	428007	132941	832	911	4.52
island.txt	TLE	5418042	7787680	31170	35110	16.53
lily.txt	6029862	262834	211835	1529	1725	4.26
spaceship.txt	TLE	4113888	2791283	7497	8483	7.31
sword.txt	7324547	811034	191669	1567	1795	3.66
watch-norefract.txt	24522515 (~7hrs)	778364	505922	7968	9663	9.26
watch-bluegold.txt	TLE	3406909	1805880	29642	35634	20.84

Note:
1. All execution times are reported in milliseconds (ms).
2. TLE denotes **Time Limit Exceeded**, indicating that the execution did not complete within a practical time limit (In hours and days)



The parallel implementations produced substantial performance improvements over the baseline implementation, and parallelizing the ray tracing workload was effective because pixel and ray computations are largely independent, allowing high concurrency. As the CPU implementations were tuned to 64-way parallelism, both OpenMP (T=64) and MPI (P=64) reduced runtimes significantly compared to the baseline. MPI consistently achieved lower runtimes than OpenMP at the same parallelism, indicating better scalability for heavier scenes and reduced shared-memory contention. For scenes that were impractical sequentially (TLE in the baseline), MPI still completed successfully, demonstrating that the tuned decomposition remained robust under extreme workloads. CUDA provided the largest improvement by exploiting massive thread-level parallelism and high device bandwidth, reducing many scenes to the range of hundreds to a few thousand milliseconds. Our tuning showed that performance was highly sensitive to block configuration; surprisingly, 8x8 blocks outperformed larger 16x16 and 32x16 configurations, likely by maintaining better spatial coherence and reducing the penalty of divergent rays. Overall, the tuning progression from baseline to CPU-parallel to GPU-parallel showed increasingly large reductions in wall-clock time, with CUDA delivering the strongest performance.

Several limitations were observed as the implementation was tuned and workloads became more complex. The baseline failed to complete for multiple scenes, preventing direct speedup calculation and confirming that sequential execution is not viable at high complexity. OpenMP performance was more sensitive to load imbalance and memory bandwidth constraints, particularly for scenes with complex geometry, leading to very large runtimes in some cases even at T=64. MPI mitigated some of these issues through process-level partitioning and improved isolation of memory traffic, but incurred communication and aggregation overhead that limits perfect scaling. CUDA, although consistently fastest, did not yield uniform speedups across all scenes because ray tracing introduces control-flow divergence and irregular memory access patterns that reduce GPU utilization on the hardest cases. Throughput testing revealed that while the 8x8 size was optimal, the 32x32 block size failed to execute entirely. This failure is attributed to the high register demand per thread in our ray tracing kernel; requesting 1024 threads (32x32) per block exceeded the physical register capacity of the Streaming Multiprocessor (SM), triggering a launch failure.

These results indicate that while the tuning choices were highly effective, further improvements would require better load balancing strategies to approach ideal scaling of ray tracing.

Conclusion :

The main takeaway from this work is that ray tracing benefits enormously from parallel execution, and performance improves steadily as the implementation moves from sequential execution to CPU-based parallelism and finally to GPU acceleration. OpenMP and MPI both provided substantial speedups, with MPI scaling more effectively for complex scenes. CUDA delivered the largest performance gains by exploiting massive thread-level parallelism, reducing runtimes by orders of magnitude compared to the baseline. Then, with the implementation of BVH in CUDA, runtimes were reduced by further orders of magnitude, showcasing that, for the best results, parallelism must be matched with algorithmic efficiency. Overall, the results demonstrate that matching the computational model to the hardware architecture—such as carefully balancing thread occupancy against register resource limits—is critical for achieving peak performance.

With more time, the next steps would focus on improving load balancing and memory locality, particularly for scenes with heavy reflection and refraction that create uneven workloads. On the CPU side, dynamic scheduling strategies and finer-grained task decomposition could improve OpenMP efficiency, while overlapping communication and computation could further optimize MPI. On the GPU, the success of smaller 8x8 blocks suggests that further experimenting with ray batching to improve memory coalescing and reduce divergence would likely yield additional gains. Scaling the implementation to larger images, more complex scenes, or multi-GPU systems would also be a natural extension of this work.

Final Output Images: