# Wavefront Path Tracing
## 15-618 Project Final Report

**Fulun Ma (fulunm)** **Benran Hu (benranh)**

## Summary
We implemented a CUDA path tracer using the wavefront pipelining design, and a wide BVH acceleration structure. We achieve a 1.02x to 1.79x speedup compared to naive CUDA implementation using megakernels, and 1.49x - 8.20x speedup compared to CPU implementation[1].

## Background

### GPU Ray Tracing
Path tracing, as one branch of the ray tracing methods, can potentially benefit from parallelization greatly by distributing the rays to different CPU threads using OpenMP or MPI. However, on GPUs or CPUs with SIMD support, naively assigning rays to each CUDA thread or SIMD lane can result in very poor speedup, given the highly divergent workload among rays.

A simplistic approach to parallelizing path tracing with CUDA or SIMD involves creating "megakernels" or "ray packets" that process an entire light path from beginning to end. However, this method faces challenges as rays in different threads might end at various points, encounter diverse object types, or interact with objects using different shaders. This can cause significant divergence within GPU warps since each thread might execute distinct instructions, leading to idle threads and low warp utilization. Additionally, when accessing acceleration structures like Bounding Volume Hierarchies (BVHs), the divergence in ray paths can adversely affect cache coherence.

### Wavefront Method
In contrast, wavefront path tracing divides the ray tracing process into several smaller kernels. Each kernel either advances the path to the next kernel, or terminates it. This allows opportunities to sort the rays by their origins, directions, hit surface types, or other properties, so that the acceleration structure traversal can have higher cache locality and lower divergence. If in any iteration a ray terminates earlier than its cohort in the same thread group, we can immediately fetch new rays for the next stage, instead of keeping the thread idle until reaching the end of a megakernel. This can achieve significantly better execution coherence much higher workload balance. The idea is described in [1] in detail.

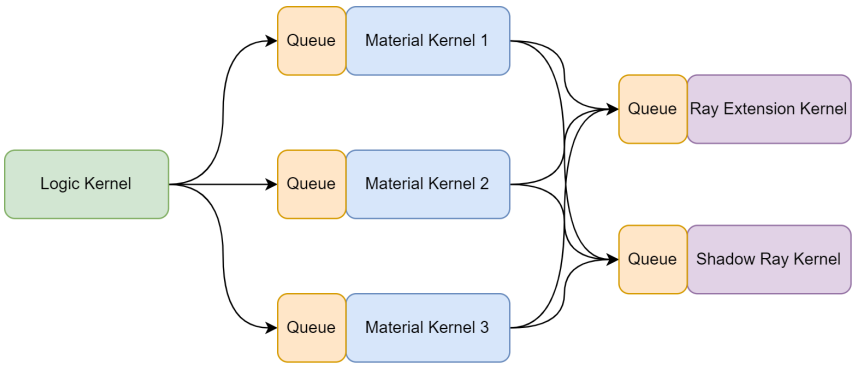



Figure 1: Diagram of a single iteration in wavefront path tracing.

[1]Our project code repository is available at https://github.com/zymk9/cmu15618_project.git, and the website is at https://zymk9.github.io/cmu15618_project/.

Figure 1 is an illustration of one possible implemention. We may divide a segment in a ray path into three stages. The logic stage is the common stage where all rays will undergo in each iteration, where we accumulate the radiance, perform light sampling, determine whether the ray should be terminated, and what is the material type at the ray hit location. Each thread in the logic kernel will enqueue material evaluation data to the corresponding queue of the material. In the material stage, each material kernel fetch from its own queue and sample the next outgoing ray direction, as will as computing the probability. The outgoing ray is then submitted to the ray casting queue, which is read in the ray cast stage where we try to find the nearest hit (or any hit) point between the ray and the scene. The intersection results are recorded and fed to the logic kernel in the next iteration.

### Acceleration Structures

In the context of GPU ray tracing, the efficient utilization of acceleration structures plays a pivotal role in expediting the process of finding the nearest primitives intersected by a ray. In our project, we employ a specialized variant known as the "wide BVH" (Bounding Volume Hierarchy) to enhance the overall performance of our ray tracing application.

The wide BVH, unlike its binary counterpart, exhibits a unique characteristic by allowing each internal node to branch into a wider range, typically between 2 and 8 child nodes. This design choice significantly reduces the height of the tree, resulting in faster traversal times. The reduced tree height, coupled with a carefully optimized traversal order, enables us to swiftly locate the nearest primitives intersected by a given ray.

The wide BVH is derived from the original binary BVH using surface area heuristics [2], with the primary objective of minimizing the Surface Area Heuristic (SAH) cost associated with the resulting structure. The SAH cost is a critical metric that influences the efficiency of traversal and intersection operations within the structure. By strategically selecting how nodes split and organize the tree, we aim to minimize the cost associated with traversing the BVH, thus accelerating the ray tracing process.

### Optimized Traversal Order

To further enhance the traversal performance, we employ an optimized traversal order when navigating the wide BVH. This order is specifically tailored to reduce the number of nodes that need to be accessed during traversal, resulting in faster ray-primitive intersection tests.

Our wide BVH implementation incorporates a heuristic method to determine the traversal order that best approximates the distance for rays with varying orientations. This adaptability ensures that the structure performs efficiently across a wide range of ray directions, ultimately leading to improved ray tracing performance.

## Approaches

Our final implementation is a wavefront path tracer with three pipeline stages and a two-level wide BVH as acceleration structures. The implementation is adapted from an existing CPU ray tracer Yocto/GL [3], and our major testing platform is a Windows laptop with a NVIDIA RTX 3060 GPU.

### Pipeline Overview

We adopt the exact three-stage wavefront pipeline mentioned in the previous section. It is composed of three main stages: `logic`, `material`, and `extend`. The logic and the extend stage contain one kernel each, whereas the material stage uses one kernel per material. The communication and synchronization between the stages are mainly done through queues and atomic operations.

## Logic Stage

The logic kernel picks up from the last intersection point from the extend kernel, samples lighting, acculumates radiance, determines the material type at the intersection, and finally decide whether the ray should be terminated or sent to the material stage. This stage is needed by all rays, and if a ray is terminated, a new sample will be fetched from the sample pool and a new ray being generated in the same kernel. This ensures that the occupancy of the logic kernel is constantly high, until the sample pool is exhausted.

The sample pool is simply a queue holding all the pixel indices in a certain order. Threads use `atomicAdd` to increment the queue front and fetch the pixel index from the queue when they complete the current ray. The buffer can be sorted based on previous computation time (number of iterations) of each pixel, so that more expensive samples are fetched first for better load balancing. But so far we only randomly shuffle it between batches.

## Material Stage

Depending on the scene complexity, the material evaluation in a megakernel design can demonstrate huge divergence issues, as different materials can have very different BRDF and PDF sampling functions, which can also be very expensive to run. It is thus reasonable to separate drastically different materials into different kernels, while merging simpler and similar ones into one.

Each material kernel has its own queue for input from the logic stage. Threads from the logic kernel use `atomicAdd` to increment the queue front of the corresponding material, and write the material information as well as the ray index into the slot corresponding to the queue front. Before dispatching material kernels, we read back the queue sizes to CPU, and launch corresponding number of threads for each material. The material kernels perform sampling and update the ray weights and the outgoing ray directions by referring to the ray index from the queue. The outgoing ray is then used in the extend stage.

There are many different choices of deciding which materials should be merged as a single kernels, and which should be separated. Currently, we decide to leave all material evaluation related to volumetric materials and materials with Dirac $\delta$ BRDF (i.e., mirrows) to the logic kernel, as they are either rare, or very easy to compute. The rest of the materials are all separated into different kernels.

## Ray Extension Stage

The extend kernel currently does not have a queue to read from. Instead, we launch the same number of threads as the logic kernels, so that extend kernel thread can directly read the corresponding ray information. This usually has a high warp utilization, as the only case where no ray casting is needed is when the material samples a ray below the surface plain.

In the extend kernel, we traverse the BVH or wide BVH using one thread per ray and find the nearest hit position. The hit information is then written to an intersection buffer, which can be used in the logic kernel. We did not implement the shadow ray casting (finding any hit) as a separate kernel as currently our ray tracer does not utilize any hit ray casting.

## Data Organization

Most of ray tracer data are stored in the GPU global memory, including the scene, camera, lighting, material, BVH structures, and the ray tracing state. The pointers to these buffers are bound to a device constant structure for fast access anywhere from the CUDA kernels.

The BVH trees are stored using a flat array, with each node specifying its children as a continuous range in the array. This facilitates uploading the BVH structure to GPU side and traversing it within kernels.

### Binary BVH to Wide BVH Conversion

Our Binary BVH to WBVH conversion process is a multi-step procedure designed to improve ray traversal efficiency within our ray tracing pipeline. The conversion begins with the computation of primitive ranges for each node in the binary BVH, specifying the range of primitives contained within each node. This information lays the foundation for subsequent stages of the conversion.

Following primitive range determination, we embark on a critical step: calculating the cost associated with various partitioning strategies for each BVH node. This comprehensive cost evaluation considers essential factors like node area, primitive count, and traversal cost. It culminates in the identification of the most optimal partitioning strategy for each node.

With cost assessments and partitioning strategies in hand, we proceed to reconstruct the wide BVH structure. This step involves the creation of wide BVH nodes, and the assignment of primitives to these nodes based on the carefully chosen optimal partitioning strategies.

To further enhance ray traversal performance, we conduct node reordering within the wide BVH structure. We create an 8x8 cost table, cost(c, s), for each node, where c represents child nodes, and s represents child slots ($s \in [0, 7]$). Each cell in the table indicates the cost associated with placing a specific child node in a particular child slot. Using this data, we efficiently determine the assignment that minimizes the total cost, employing an auction algorithm. This optimization step streamlines the traversal process, resulting in significant performance improvements.

## Iterating Process

### Initial Megakernel Version

Our first step after cloning the codebase is to adapt a naive megakernel GPU version of the ray tracer to get a sense of the performance baseline and potential improvements. We attach our preliminary comparison between the megakernel CUDA implementation and the CPU version in Table 1. We can see that even with the naive megakernel design, the CUDA implementation can still achieves 2x-5x speedup compared to the CPU implementation. The OptiX implementation uses hardware accelerated intersection tests (NVIDIA's RT cores). Note that the OptiX version uses the same megekernel design.

Comparing the performance of CUDA and of OptiX implementation, we found that the performance of ray-scene intersection test is crucial for the overall rendering performance, especially in scenes with highly complex geometries. Therefore, the first optimization we plan to implement for our wavefront version is to make the BVH traversal a separate pipeline stage. We need to make the BVH "wider" by increasing the fan-out width and decreasing the tree height, which reduces the costly pointer chasing and allows us to parallelize ray-box/ray-triangle intersection test over all threads within a group. We do not expect that this can reach the same speedup as the OptiX version, but it will be a significant optimization to consider first.

| Time (s) | Dragon | The Breakfast Room | Country Kitchen | Landscape |
|---|---|---|---|---|
| CPU | 0.0811 | 0.4537 | 0.6665 | 5.7601 |
| CUDA | 0.0180 | 0.1346 | 0.1292 | 2.7245 |
| OptiX | 0.0076 | 0.0203 | 0.0243 | 0.0888 |

Table 1: **The rendering time of 1 spp in different scenes.** CUDA refers to our initial megakernel implementation.

### Two-Stage Pipeline without Load Balancing

As we believe that the BVH traversal, i.e., the ray casting stage is the most time-consuming, and highly divergent part, we set out to first separating it as a single pass for optimization opportunity. In our

first two-stage wavefront pipeline version, we use a large kernel for all logic and material part, and another kernel for ray casting. There are one-to-one correspondence between the threads in the two kernels and the pixels, meaning that each thread will generate the new rays at the same pixel location as the old rays.

We quickly found that this pipeline suffers greatly from workload imbalance. For instance, pixels corresponding to the background region will have all most no ray intersections, hence can be traced at high speed, where as pixels corresponding to foreground geometries takes many more iterations. However, our implementation implies that threads from the low-cost region cannot help those in the expensive-to-compute areas. In many scenes with imbalanced workloads, the two-stage design falls short of the megakernel design.

### Adding Load Balancing

We realized that pipelining itself does not solve divergence or workload imbalance, but only offers a possibility to address them. We hence resort to work queues and allow threads to fetch new rays from them. It is easy to synchronize over the queue, nevertheless, multiple threads may now updating the same pixel, causing race conditions. Our first solution is to allocate a buffer large enough for all threads to write their ray tracing results to, and blending over all results after all threads have finished, instead of updating the image data inplace. However, this method cannot scale with resolution, or sample batch size. This prompts us to use atomic operations to update the image data, which does not need auxiliary buffers.

### Switching to Wide BVH

Given that using wavefront design itself does not help with the divergence internal to BVH traversal, we planned to use a wider BVH implementation to reduce the pointer-chasing latency and improve memory locality as well as execution coherence.

### Optimize Traversal Orders

Following our decision to switch to a wide BVH, we recognized the need to optimize traversal orders within this structure to maximize the benefits of its design. The effectiveness of a BVH, wide or binary, is significantly influenced by the order in which its nodes are traversed during the ray tracing process. Efficient traversal order can minimize unnecessary node visits and accelerate the process of finding ray-primitive intersections.

### Dividing Material Stage

After splitting the scene intersection part as a separate stage, a natural next move is to also separating the material sampling part, which can be highly divergent as well, especially in complex scenes with varieties of expensive-to-evaluate materials. As a result, we decided to make it an independent stage of the pipeline. We reorder the logic kernel to adapt for the change, and adding new work queues and buffers for each material kernel. This is our current version of the pipeline.

## Results

We ran different performance comparisons to see the speedup of our wavefront path tracer with respect to the megakernel version and the CPU version. We also carried out ablations on some of the design choices we made to validate if they are effective. Figure 3 shows part of the scenes we used in our testing. We measure the performance by consider the speedup in rendering 1 sample per pixel (spp) for an entire 1280x720 image area.

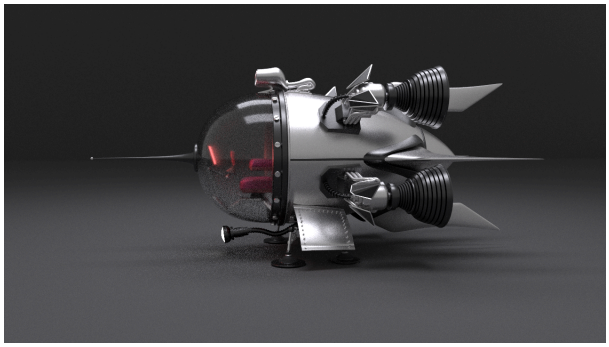
a) Salle de bain

b) Dragon

c) The Breakfast Room

d) Country Kitchen

e) Landscape



f) 4060.b Spaceship

g) Contemporary Bathroom

Figure 3: **Test Scenes Used in Results and Comparisons.** Rendered using our implementation with 512 spp. The first five scenes come from Benedikt Bitterli's rendering resources. The last one is from PBRT-v4's repo.

We use scenes of different geometry complexity and shading complexity (variety of materials and lightings). The *dragon* scene contains a single object with simple material, whereas the *landscape* contains 23,241 plant models and 3.1B triangles. The other scenes sit inbetween and feature different sets of materials and workload distribution.

For most of the testing, we time the total rendering time for rendering 1 sample-per-pixel (spp) for a 1280x720 image, and average the rendering time over 256 samples. All the comparisons are run on a laptop with an NVIDIA RTX 3060 GPU.

## Comparison between Wavefront and Megakernels

We first compare our wavefront pipeline, both the two-stage and the three-stage version, to the megekernel CUDA implementation, and the CPU implementation of the same ray tracer. Our two-stage wavefront path tracer achieves a speedup of 1.49x - 8.20x with respect to the CPU version running in parallel on 16 cores. Comparing to the megakernel version running on the same GPU, we see a speedup from 1.02x to 1.79x.

Examining the speedup and the workload, we found that the scenes where CUDA versions have lower speedup are generally those with complex material combinations and geometries (e.g., Landscape and Spaceship), which may cause severer divergence and load imbalance. On the other hand, these are also the scenes where our wavefront implementation shows a higher speedup against the megakernel version.

When comparing our two-stage and three-stage pipelines, we found that the three-stage pipeline is generally slower, except for one scene (Spaceship). We examine the actual material composition in these scenes and notice that most of them have 60-80% of the ray intersections touching the matte material, which is easy to evaluate. The rest of the rays only intersect one or two other materials, indicating that the divergence in material evaluation is very limited in most of the chosen scenes. We also examined the Spaceship scene and found that it indeed had more types of expensive materials. Another observation is that when the sample pool size is small, the ratio of expensive materials among all rays continues to increase, as cheaper samples are quickly consumed.

Overall, we believe this implies that the extra overhead of dispatching dedicated material kernels and the memory access overhead of the work queues and buffers do not offset the gains from the better coherence.

| Time (s) | CPU | Megakernel | Wavefront (2 stages) | Wavefront (3 stages) |
|---|---|---|---|---|
| Bathroom | 0.9438 | 0.2028 | 0.1438 | 0.1992 |
| Salle de bain | 0.3873 | 0.0689 | 0.0473 | 0.0765 |
| Dining Room | 0.3802 | 0.1013 | 0.0991 | 0.1298 |
| Dragon | 0.0768 | 0.0147 | 0.0101 | 0.0125 |
| Kitchen | 0.5568 | 0.1132 | 0.0679 | 0.1025 |
| Spaceship | 0.3398 | 0.4102 | 0.2287 | 0.1966 |
| Landscape | 6.0551 | 2.7203 | 1.5228 | 1.8609 |

Table 2: **Comparison between megakernel and wavefront design.** The times given are for rendering 1 sample per pixel, averaged over 256 samples.

## Comparison between Binary BVH and Wide BVH

In our next comparison, we focus on evaluating the performance differences between using a binary BVH and our optimized wide BVH in the context of our wavefront path tracing pipeline. The motivation behind this comparison is to assess the impact of the wide BVH structure on the efficiency of ray-primitive intersection tests, which are critical for overall rendering performance.

Table 3 suggests that the wide BVH implementation, contrary to expectations, resulted in longer rendering times compared to the binary BVH. This is an intriguing outcome, as the wide BVH is generally

anticipated to improve performance due to its shallower tree structure and potentially more efficient traversal. Let's analyze the possible reasons behind this unexpected outcome.

Firstly, the increased rendering times with the wide BVH may be attributed to issues with cache coherence and memory access patterns. Wide BVH nodes, containing more children than binary BVH nodes, increase the memory footprint per node. If these larger nodes do not align efficiently with GPU cache lines, this could lead to more frequent cache misses, slowing down the traversal process. Additionally, GPUs are adept at prefetching data based on predictable access patterns. The binary BVH, with its simpler and more predictable structure, might better align with the GPU's prefetching mechanisms compared to the wide BVH, which has more complex access patterns.

Another critical aspect to consider is load balancing and thread utilization. The wide BVH's structure may introduce more divergence in thread execution paths. In GPU architecture, particularly where warp coherence is vital, if threads within the same warp traverse different paths in the BVH, it can lead to underutilization of resources. Additionally, the distribution of rays among different paths in the wide BVH might be uneven, resulting in some threads being idle while others are still processing. This uneven distribution of work can affect the overall performance adversely.

Lastly, scene and data-specific factors also significantly impact the performance of the wide BVH. Different scenes vary in their complexity, in terms of both geometry and materials. In scenarios where the binary BVH already performs adequately due to simpler geometry, the wide BVH might not offer a substantial advantage. The spatial distribution and size of primitives in the scene can also affect the performance of the BVH. If the node splitting in the wide BVH does not align well with the distribution of primitives, it could lead to a less efficient traversal.

In conclusion, while the wide BVH is theoretically advantageous for ray tracing applications, its real-world performance is heavily contingent on the specifics of the implementation, the characteristics of the scenes being rendered, and the alignment with GPU architecture. Addressing these issues, such as improving memory access patterns, enhancing load balancing, and adapting the structure to specific scene characteristics, could potentially unlock the expected performance benefits of using a wide BVH。

| Time (s) | Megakernel BVH | Megakernel WBVH | Wavefront BVH | Wavefront WBVH |
|---|---|---|---|---|
| Bathroom | 0.2541 | 0.3497 | 0.1487 | 0.2060 |
| Salle de bain | 0.0929 | 0.1192 | 0.0494 | 0.0691 |
| Dining Room | 0.2030 | 0.1772 | 0.1028 | 0.0963 |
| Dragon | 0.0226 | 0.0312 | 0.0108 | 0.0148 |
| Kitchen | 0.1232 | 0.1729 | 0.0716 | 0.1066 |
| Spaceship | 0.4755 | 0.5045 | 0.2366 | 0.2619 |
| Landscape | 6.8145 | 7.2445 | 1.5949 | 3.9315 |

Table 3: **Comparison between binary BVH and wide BVH (max fan-out 8).** We compare the two BVH forms on both the megakernel and the wavefront implementation.

## Sample Batch Size Influences

As our wavefront pipeline relies on dynamicaly fetching from the sample pool to reduce workload imbalance, the sample pool size affects our performance. We ran a set of experiments to verify this result. As in Table 4, the per-sample tracing time generally increases with the sample batch size decreasing.

| Time (s) / Batch Size | 4 | 32 | 128 | 256 |
|---|---|---|---|---|
| Dining Room | 0.1136 | 0.1022 | 0.1010 | 0.1011 |
| Dragon | 0.0161 | 0.0113 | 0.0104 | 0.0102 |
| Kitchen | 0.0784 | 0.0716 | 0.0701 | 0.0689 |
| Spaceship | 0.2753 | 0.2368 | 0.2367 | 0.2298 |

Table 4: **Performance of the wavefront path tracer under different sample batch size.**

# Distribution of Work

In our project, the tasks were distributed between two team members, Fulun Ma and Benran Hu, each focusing on different aspects of the development process. Here is a breakdown of the work performed by each participant and our proposed distribution of credit for the total project.

Fulun Ma's role was specialized and crucial in certain technical aspects of the project: 1. BVH to WBVH Conversion, 2. Traversal Order Optimization.

Benran Hu's role in the project was comprehensive, covering various aspects of the path tracing system. His key contributions were: 1. Implementation of the Ray Tracing Pipeline, 2. Material Stage Development, 3. System Integration and Debugging. Notably, the initial concept and overall direction of the project were attributed to Benran Hu.

Given the extent of Benran Hu's contributions, including the original idea, project direction, and significant implementation work, alongside Fulun Ma's specialized technical input, a fair distribution of credit appears to be skewed towards Benran. A distribution of 40%-60% between Fulun Ma and Benran Hu, respectively, seems appropriate.

# References


[1] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on GPUs", *Proceedings of the 5th High-Performance Graphics Conference*, pp. 137–143, 2013.

[2] H. Ylitie, T. Karras, and S. Laine, "Efficient incoherent ray traversal on GPUs through compressed wide BVHs", *Proceedings of High Performance Graphics*. pp. 1–13, 2017.

[3] Yocto/GL, "Yocto/GL: Tiny C++ Libraries for Data-Driven Physically-based Graphics". 2022.