CS267 Final Project
Parallelized BVH Construction for Path Tracer

Haohua Lyu	Siyu Zhang	Bingxin Zhang
haohua@berkeley.edu	c_u@berkeley.edu	winizhbx@berkeley.edu
3037443695	3037441927	3037444306

1 Introduction & Contributions

Bounding Volume Hierarchy (BVH) is a widely used acceleration structure for detecting potential object collisions (e.g. ray-triangle intersections). This project is aiming to accelerate the BVH construction by using GPU to optimize the performance. Our work is based on the CS284A path tracer [1], so it also involves updating the existing CPU codes to support CUDA parallelism. Our final solution uses CUDA to implement the BVH construction and improves the performance for max **40** times faster than the CPU version when running with 600k primitives. We select 15 assets/meshes with different number of primitives and conduct run-time experiments on them. Fig. 1 shows the GPU version run-time performance versus the CPU version. The Config 4 is the version we have moved all parts to GPU, while Config 2 only have half of the process on GPU. The full GPU version has the best run-time performance. Moreover, with more primitives in the environment, more time will be saved by using GPU.

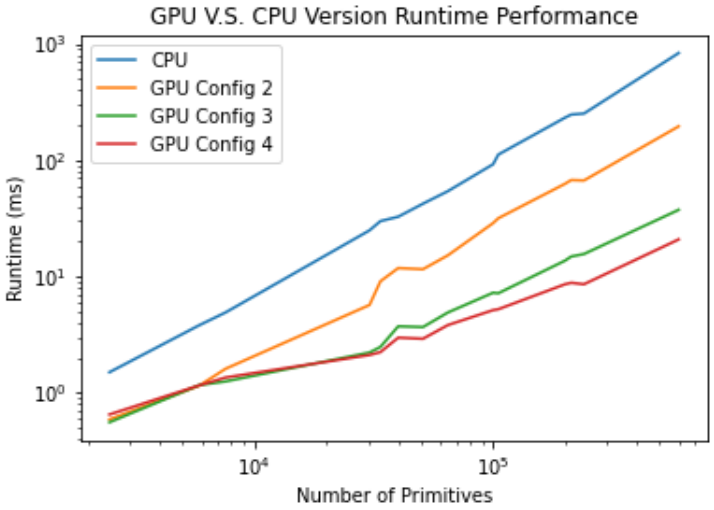

Figure 1: CPU v.s. GPU with 3 different configurations run time performance. GPU Config 4 is the version which we have moved all parts to GPU. Details see in Table 2.

This report will discuss the algorithm, data structures, CUDA implementations, experiments, and future work in the following sections.

For this project, all members contributed to different parts of the code and the write-up. Specifically, our focuses are:

- Haohua Lyu: implemented GPU multiple versions, ran experiments, and worked on write-up;
- Siyu Zhang: implemented GPU multiple versions, ran experiments, and worked on write-up;
- Bingxin Zhang: ran experiments, and worked on poster and write-up.

2 Algorithm Details

Bounding Volume Hierarchy (BVH) is a structure which is widely used to quickly identity pairs of potentially colliding 3D objects. While common strategies include top-down and bottom-up construction, our approach uses a hybrid strategy, building the nodes in a top-down fashion with Morton code and adding the bounding boxes from leaves to root. This technique, involving the use of Morton codes, is often called Linear BVH (LBVH). Fig. 2 shows an example of a LBVH tree. The time complexity of this approach is $O(n)$ [2] [3].

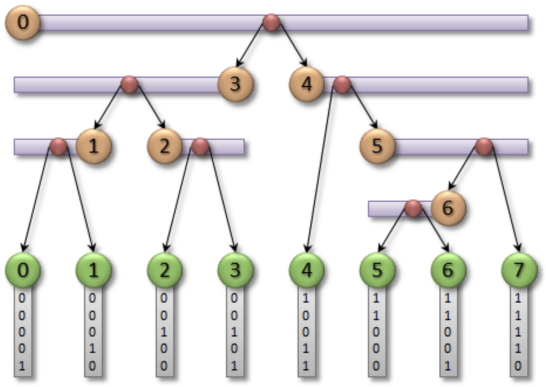

Figure 2: BVH Sturcture [4].

Generally, we build the BVH tree in 4 steps:

1. Calculate Morton code for all primitives, based on their spatial locations;
2. Sort Morton code as binary numbers;
3. Build internal nodes based on Morton similarity in a top-down fashion;
4. Parallel reduction by summing up bounding boxes.

The first step is to calculate the Morton codes, which can be calculated by the 3D point's binary fixed-point representation of its coordinates. We take the fractional part of the coordinates and expand it by inserting two "gaps" after each bit. Then we interleave the bits of the three coordinates together to form a single binary number [4]. After calculating Morton codes, we sort the codes at the second step. The detailed kernel implementation is further discussed in Sec. 3.2.

Once the codes are sorted, we build the internal nodes in a top-down fashion. The nodes will be generated using BFS method, and we build the hierarchy as shown in Fig. 2. Then the final step is to summing up the bounding boxes after the hierarchy has been built. We need to assign the bounding box for each of the nodes and sum up them to perform the parallel reduction.

3 Data Structure and Implementations

In this section, we discuss the implementation details of the aforementioned algorithm. We first walk through the data structures used in the project, specifically the two types of nodes used across CPU and GPU. We then discuss the use of CUDA kernels and features. We also discuss certain roadblocks we encountered when implementing the algorithm.

We build our project using the CS284A Path Tracer project [1], which is capable of physics-based rendering. Once the project is executed, it would read models as a list of primitives (i.e., basic shapes like triangles) and pass it to the BVH constructor. In the original CPU implementation, the BVH constructor would build a BVH tree with the struct `BVHNode` and pass the tree to the path-tracing renderer for quick ray-object intersection detection.

In our implementation, we follow the four steps discussed in Sec. 2 when entering the BVH construction stage. We use the class `LBVHTreeBuilder` with the struct `LBVHTreeNode` to build a LBVH tree, with simplified data structures for reduced memory transfer and easy access on GPU. The construction can be performed either on CPU or GPU, and difference is further experimented and discussed in Sec. 4.2.1. Finally, we convert the LBVH tree back to a BVH tree to match the original API.

3.1 Data Structure

We discuss specific data structures used in our project, as well as challenges encountered when using these data structures with separate compilation.

3.1.1 BVH node and LBVH node overview

In an abstract level, nodes in a BVH tree should include the following pointers:

1. A list of all primitives contained in the node;
2. an axis-aligned bounding box surrounding all primitives in the node;
3. the parent of the node;
4. and the left and right children of the node.

In the original CPU implementation, the BVH tree is built with the struct `BVHNode`. Note that many passes of the primitives are required, and the list of primitives do not change. Hence, instead of each node storing the primitives' data, `BVHNode` only stores two iterators `start` (inclusive) and `end` (exclusive) to iterate through a shared main list of primitives in a given range. The bounding box is stored as a `bbox` struct, which contains the two `Vector3D` points defining the bounding box and the extent (difference) between them. Finally, three `BVHNode` pointers provide references to root, left child, and right child respectively.

To minimize the memory transfer between CPU and GPU and facilitate access on GPU, we design the `LBVHTreeNode` struct with different data structures. First, since host memory addresses are not available on GPU, we change all pointers to indices of specific arrays, and pass those arrays to the GPU. That is, we have arrays of primitives and nodes and store integer indices of primitives and nodes that pointing to array positions. This also reduce the total memory size being passed between host and device. Once the LBVH tree is built on the GPU, they are passed back to host and reconstructed to a BVH tree by reassigning the memory pointers.

3.1.2 Struct in CPU and GPU

When implementing the data structures, we find that the reconstructed BVH tree has invalid memory locations loaded in the nodes. Specifically, we observe that the children node addresses are off by 24 bytes when the node is being accessed inside a `.cu` file and inside a `.cpp` file. By further looking into the `BVHNode` struct, we find that the struct starts with a `bbox` and then with the memory addresses followed. That is, we find the size of the bounding box is differed on `.cpp` and `.cu` files by 24 bytes.

Looking deeper inside the bounding box, it contains three `Vector3D` points, which are the two 3D points in the opposite corners of the bounding box and the extent (difference) between the two points. Each of the `Vector3D` point contains three `double` representing the x, y, z coordinates. By comparing its size inside the `.cpp` and `.cu` files, we find that a `Vector3D` struct is 24 bytes in a `.cu` file, but 32 bytes in a `.cpp` file. Since there are three `Vector3D` inside the bounding box, it creates the 24-byte offset.

We eventually realize that the difference is caused by the separate compilation of C++ and CUDA code. The `.cpp` files are compiled using MSVC on a Windows platform, while `.cu` files are compiled by NVCC. On MSVC, the `Vector3D` struct is aligned to 16 bytes by MSVC, potentially because of compiler optimization for AVX (YMM) registers on modern CPUs. Thus, although the three doubles only take 24 bytes, the struct takes 32 bytes when compiled by MSVC. NVCC does not have such features, so the struct size is still 24 bytes. This difference results in the offset on the memory addresses.

This issue is resolved by adding macros to force alignment to 16 bytes on both compilers. For NVCC, this is done by calling `__align__(16)`; for MSVC, we add the `alignas(16)` call to enforce the alignment as well.

3.2 CUDA Implementations

Most calculation on GPU is done through CUDA kernels. To add CUDA support, we include two additional files: `lbvh.h` and `lbvh.cu`, as well as code across different files supporting conversion between LBVH and BVH trees. New CUDA functions and kernels are mostly implemented inside `lbvh.cu`; they are briefly introduced below, in the sequence of their orders in the pipeline.

- The `getRootBBox` function iterates through the list of primitives and combine their bounding boxes to obtain the root bounding box, which is the total space occupied by the asset. This space is later further split into smaller boxes for Morton code calculation. In this step, we utilize `thrust::reduce` from the Thrust library and use a lambda expression to pass in the basic operation (i.e., combining two bounding boxes to get a larger one). The GPU cores perform the reduction on all primitives simultaneously.

- The `calculateMortonCode` function calls the `calcMortonCodeAndInitPrimIdx` kernel, which takes in a primitive and calculate Morton code based on its relative position in the total space calculated earlier. Again, the GPU cores run the kernel on all primitives simultaneously and put the Morton code into an array.

- The `sortMortonCode` function sort the calculated Morton code to put spatially near primitives in close index positions. This function utilizes `thrust::sort_by_key` to perform radix sort on the primitive indices, the morton code, and the bounding box indices simultaneously.

- The `processInternalNode` kernel splits a given range of primitives, prepares child nodes, and records the parent-child relationships. The GPU cores run the kernel on the root node and

corresponding new nodes in a top-down fashion.

- The `calculateBoundingBox` kernel takes the list of nodes as inputs and calculate the bounding boxes of leaf and internal nodes. The GPU cores run the kernel on the leaf nodes and then their parent internal nodes in a bottom-up fashion.

For most of these functions, we also implement a CPU version to allow better debugging and comparisons. This allows us to have different configurations, with the pipeline partially or fully running on GPU. The configurations are further discussed in Sec. 4.2.1.

4 Experiments

In this section, we discuss our experiments and analyze our CUDA implementation of BVH construction in terms of both run-time/speed and quality of the resulting BVH trees (i.e. how much does it improve the rendering time for the path tracer). We firstly reason our choices of assets/meshes for the following experiments, and then compare the CPU version with different configurations of GPU implementations throughout the project (See Sec. 4.2.1). In Sec. 4.2.2, the run-time breakdown of the fastest implementation would be analyzed, whereas the quality of implementation would be evaluated in Sec. 4.2.3.

4.1 Assets/meshes Choices

There are 15 different assets/meshes (See their mesh structures in the Appendix B) chosen as the rendering target in our following experiments. The target object is loaded through a DAE file (stands for Digital Asset Exchange file format) and processed by the path tracer. The meshes all consist of triangles as primitives and the number of primitives ranges from ~2500 to 600K (See Table 1). Meanwhile, each mesh is simply one object in the scene (limited by the load ability of the exiting CPU path tracer) and the camera is positioned right in front of each (See Appendix B). Nevertheless, this wide covering range of the number of primitives is good enough to analyze the performance of our implementation. Besides the variety in the number of primitives which can be understood of the problem size, this set of assets also covers the variety in spatial distribution. In other words, some of the meshes spatially cluster together, while some of them consist of triangles that are located spread out a certain space. This property of assets would, to some extent, affect the speed of BVH construction and the quality of the final BVH tree, which would be discussed in Sec. 4.2.1.

4.2 Performance Analysis

In this section, we present several experiments and plot to analyze the performance of our CUDA implementation of BVH construction. The run-time comparison and breakdown experiments are run on a Windows 11 system with an Intel Core i7-11800H CPU and an NVIDIA GeForce RTX 3080 Laptop GPU (8G VRAM and 6144 CUDA Cores). In an overview, both the run-time of BVH construction and the quality of the final BVH tree show a good performance. The GPU BVH construction is at most **40** times faster than the CPU version if we do not take into account the overhead of CPU-GPU memory exchange in such a hybrid mode. The experiments below are all conducted under the setting of `NUM_THREADS = 1024`. This number is chosen because 1024 is the maximum compute work group invocations or maximum number of threads per block for the device.

File/mesh names	# of primitives
teapot.dae	2,464
cow.dae	5,856
beetle.dae	7,558
bunnyNoBox.dae	30,338
bunny.dae	33,696
peter.dae	40,018
maxplanck.dae	50,801
beast.dae	64,618
lucy.dae	100,000
dragonNoBox.dae	100,000
dragon.dae	105,120
tyra.dae	200,000
armadillo.dae	212,574
wall-e.dae	240,326
happyBuddha.dae	600,000

Table 1: 15 experiment assets and its corresponding number of primitives (triangles).

4.2.1 Run-time Comparison Among Different Implementations

There are four different implemented configurations throughout the project, which is specified in Table 2. In later experiments and plots, they are presented in the form of its configuration number (e.g. Config 1). The descriptions for the configurations below are based on the four steps we discussed in Sec. 2, but it should be mentioned that `root bounding box computation` is a sub-task in the 1^{st} step of the algorithm. Step 1 of the algorithm consists of two main parts: root bounding box computation and other bounding boxes computation. We parallelize the two parts in different configurations. In Config 2, both parts are on CPU; in Config 3, bounding boxes of all nodes except the root node is calculated through a GPU kernel launch; in Config 4, bounding box of the root is computed by using `thrust::reduce` which was discussed in Sec. 3.2.

	root bbox computation on GPU	other nodes bbox computation on GPU	sort Morton code on GPU	Tree Construction on GPU
Config 1				
Config 2				✓
Config 3		✓	✓	✓
Config 4	✓	✓	✓	✓

Table 2: Configuration number and demonstration of whether the sub-task are implemented on GPU (if no ✓, it means on CPU)

Before starting analyzing the experiment results, it should be mentioned that, in the figures below, CPU and GPU memory exchange overhead is not taken into account in calculating the total run-time nor the run-time speed up (`Config1 run-time / Config4 run-time`). This is mainly due to the following two reasons. Firstly, all computations of the original path tracer are computed on CPU, including the interface design for visualizing the BVH tree and the traversal operations during the ray-tracing stage. In other words, if we finish constructing a BVH tree on GPU, we need

to convert it into the correct the CPU version to be correctly visualized and traversed. Therefore, this portion of time should not be taken into account. Secondly, this project focuses on constructing a BVH tree on GPU with a parallelized algorithm and the memory exchange should not be taken into account as, in reality, a parallelized BVH construction would be used in a GPU-based path tracer which would not have such overhead.

In Fig. 1 and Fig. 5, we can qualitatively and quantitatively evaluate our implementations and conclude that Config 4 implementation shows a significant run-time speed up ranging from 2.3 to **40**, compared with the CPU version (Config 1). It is also the same to state that, compared with our serial version of BVH construction on CPU, the run-time of our full GPU implementation (Config 4) is at most **40** times faster with the given experiment assets. Meanwhile, as can be seen in Fig. 3, the run-time of both CPU and GPU implementations increases almost linearly with the number of primitives which is reasonable and correspondent with what we discussed in Sec. 2.

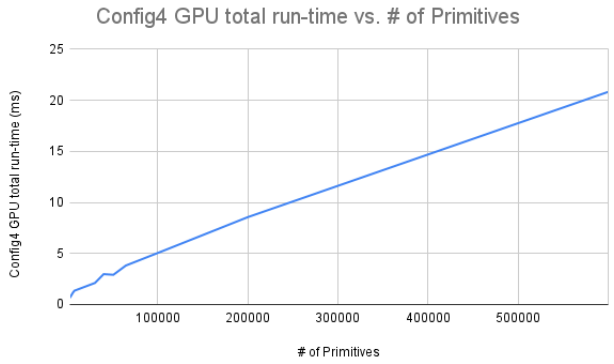

Figure 3: Config 4 GPU Run-time vs. # of Primitives

However, as can be noticed in Fig. 5 and Appendix A, the speed up of Config 4 drops when the problem size (the number of primitives) is very small (e.g. `teapot.dae` and `cow.dae`) and they are even lower than Config 2 and Config 3 in some experiment assets. Also, even if the numbers of primitives of two assets are the same (e.g. `lucy.dae` and `dragonNoBox.dae`), they show a slightly different run-time in some sub-tasks. These two phenomena that we observed from the run-time experiments would be reasoned in the following perspectives: the number of primitives and the spatial distribution of primitives.

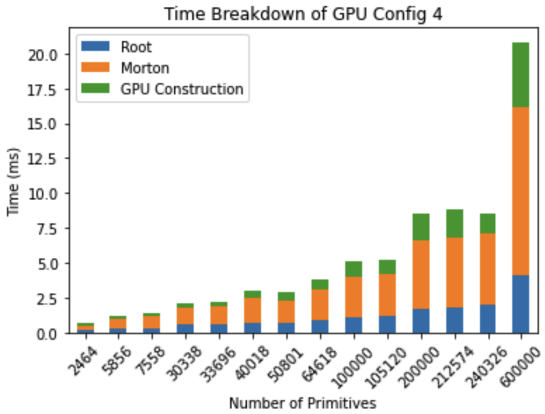

Figure 4: Config 4 GPU Run-time Breakdown

Filenames	# of Primitives	Config1	Config4 root	Config4 Morton	Config4 GPUconstruction	Config4 GPU total	Speed Up
teapot.dae	2464	1.5	0.1495	0.3211	0.1791	**0.6497**	2.308757888
cow.dae	5856	3.8	0.2886	0.6757	0.1916	**1.1559**	3.287481616
beetle.dae	7558	4.9	0.2611	0.8942	0.1962	**1.3515**	3.625601184
bunnyNoBox.dae	30338	24.9	0.5464	1.2248	0.3385	**2.1097**	11.80262597
bunny.dae	33696	29.9	0.5522	1.2944	0.3872	**2.2338**	13.38526278
peter.dae	40018	32.6	0.6647	1.8627	0.4465	**2.9739**	10.96203638
maxplanck.dae	50801	42.2	0.6793	1.6095	0.6213	**2.9101**	14.50121989
beast.dae	64618	54.1	0.8561	2.2817	0.6775	**3.8153**	14.17974995
lucy.dae	100000	92.2	1.1124	2.909	1.1093	**5.1307**	17.97025747
dragonNoBox.dae	100000	108.6	1.2179	2.6363	1.0909	**4.9451**	21.96113324
dragon.dae	105120	111.6	1.2014	2.9974	1.0242	**5.223**	21.36703044
tyra.dae	200000	231.7	1.6934	4.9071	1.9561	**8.5566**	27.07851249
armadillo.dae	212574	246.5	1.7945	5.035	1.9851	**8.8146**	27.96496721
wall-e.dae	240326	251.5	1.9494	5.1568	1.4592	**8.5654**	29.36231816
happyBuddha.dae	600000	834	4.0578	12.0783	4.6887	**20.8248**	40.04840383

Figure 5: Table of run-time of Config 4, its corresponding sub-task run-time and overall speed up compared with Config 1

Number of Primitives The run-time speed up is smaller when dealing with assets with fewer number of primitives, especially for Config 4 (See Fig. 4, Fig. 5 `teapot.dae` and `cow.dae`). This is mainly due to the overhead of kernel launches. It experiences a diminishing return in using parallelized method to get the bounding box of the root node, for instance, comparing Fig. 5 and 9. This is same as saying that probably Config 3 is a better choice when the number of primitives in the scene is smaller than a threshold (around 8,000 triangles in this case). Thus, the benefits brought by the parallelism is compromised by the kernel launch overhead when the problem size is small.

Spatial Distribution of Primitives In this section, we focus on these two assets `lucy.dae` and `dragonNoBox.dae` (See them in Appendix B) which have the same number of primitives (100,000) but show slightly distinct run-time speed up under both Config 3 and Config 4. This is caused by the different spatial distribution of the two meshes. Compared with `lucy.dae`, the triangles of the `dragonNoBox.dae` are spatially closer to each other, resulting in relatively similar Morton code values, and leading to a shorter sorting time on GPU using parallelized radix sort. Meanwhile, it would also cause a less-balanced tree if the primitives are partially clustered or show a wider range of spatial distribution, causing lower run-time performance in GPU construction when trying to build the structure from sorted Morton code.

Overall, smaller number of primitives would, to some extent, compromise the benefits brought by the parallelization due to kernel launch overhead, whereas the spatial distribution of primitives would slightly affect the sorting and construction time.

4.2.2 Run-time Breakdown

We also conduct a run-time breakdown analysis on Config 4 with the `happyBuddha.dae` because it owns the largest number of primitives in our dataset. For this analysis, we use NVIDIA NSight Compute because the device uses CUDA 11.5 and another profiler tool `nvprof` is not supported on devices with compute capability 8.0 and higher.

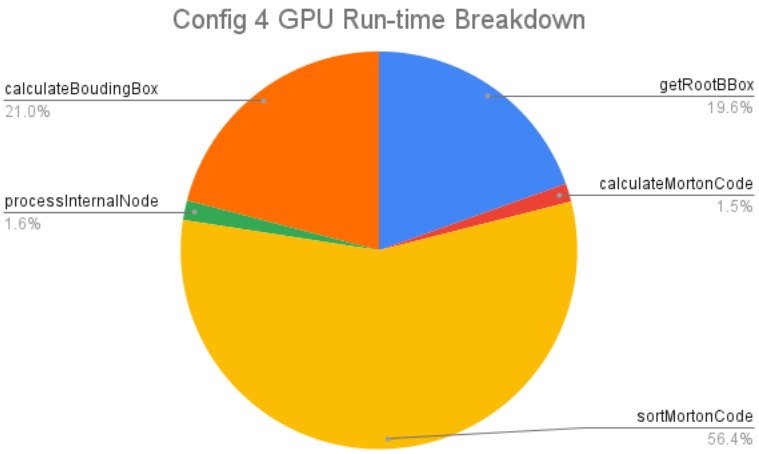

Figure 6: Config 4 GPU Run-time Breakdown

As discussed in Sec 3.2, there are five essential functions for Config 4 implementation, including `getRootBBox`, `calculateMortonCode`, `sortMortonCode`, `processInternalNode`, and `calculateBoundingBox`. Each of them either calls a kernel function or itself is a kernel function. According to the profiling run-time summary provided by NSight Compute, we can get this pie chart representing the run-time breakdown of Config 4 (See Fig. 6). As can be seen, the function `sortMortonCode` accounts for more than half of the total run-time. This result also helps to explain why the run-time increases as the number of primitives is larger. The percentages of function `getRootBBox` and function `calculateMortonCode` are similar because both of them traverse through all the bounding boxes of primitives.

Float and Double Object Selection When profiling Config 4, we also notice an interesting result from NSight Compute (See Fig. 7). This graph shows that the FP64 usage is quite high in function `calculateBoundingBox`. FP64 is the highest-utilized pipeline (86.3%). It executes 64-bit floating point operations. The pipeline is over-utilized and likely a performance bottleneck. Also, there is also a warning suggesting that the ratio of peak float (FP32) to double (FP64) performance on this device is 64:1. In other words, FP64 efficiency on this device (NVIDIA GeForce 3080 Laptop) is worse than FP32, and it is a general case for NVIDIA gaming graphics cards. Indeed, we notice this property of the device. And in order to reduce the effect, in the function `calculateBoundingBox`, we convert each `Vector3D` (3 doubles representing the 3D position) into floats and then do the calculation to get the Morton Code of each primitive. This conversion (from double to float), in fact, takes a large amount of time. Although this could be solved better if we use float in all functions and data, it would be a bit out of the scope of this project. In addition, it could possibly cause other issues if changing all data types from double to float. But it could be potential future work instead.

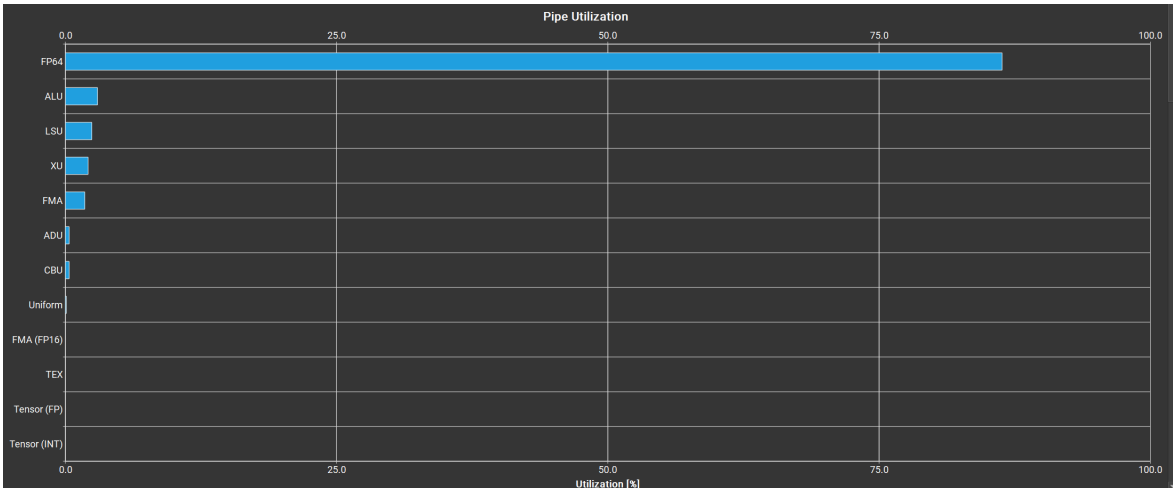

Figure 7: Screenshot of the FP64/32 Utilization analysis of "calculateBoundingBox" from NSight Compute Profiler

4.2.3 Improvement on Rendering Run-time

Additionally, we want to understand the quality of the GPU-built BVH tree as it is built differently compared to the CPU-built version. The quality of a BVH tree refers to its ability to reduce ray-object intersection checks (while ensuring correctness); the fewer checks the program performs, the faster the rendering is. Hence, we render several scenes with a CPU-built tree and a GPU-built tree to understand the difference; smaller rendering time will indicate higher quality.

The quality of a BVH tree is highly dependent on the scene itself, including the lighting and assets. Hence, we find the performance varies a lot with assets in our dataset. Nonetheless, we generally observe similar or improved quality, ranging from no speed up to around 20%. In the best case, when rendering `CBbunny.dae`, the GPU version (Config 4) takes 79.6023 seconds to render when the CPU version (Config 1) takes 100.0016 seconds; a 20.399% reduction in rendering time. Thus, we find our GPU approach produces BVH trees with similar or even higher quality. Note that although our CPU implementation is not using state-of-the-art techniques like surface area heuristics (SAHs) for higher quality, these techniques are expected to take much more time to run. Hence, our approach may show a better speed advantage when comparing to a CPU version with better quality. It then falls to the user's discretion to consider the trade-off between speed and quality.

5 Impact and Future Work

To conclude, our GPU-based BVH construction implementation improves the efficiency of the CS284A Path Tracer. We not only achieve **40** times faster run-time in BVH construction but also improve the quality of the resulting BVH tree for the ray-tracing stage by around 20% on the selected scene. In addition, we also find that GPU shows greater advantages in building BVH trees when the number of primitives is large. This could lead to another possible direction for future work, where we could develop an adaptive configuration and run only selected parts of the pipeline on GPU depending on the number of primitives in the scene. Last but not lease, we find that, when designing a new path tracer for running on gaming graphics cards, using float data type instead of double could be a better choice in computational efficiency.

References

[1] CS184 Staff, "Cs184 assignment 3-1 spring 2022." [Online]. Available: https://cs184.eecs.berkeley.edu/sp22/docs/proj3-1

[2] C. Apetrei, "Fast and simple agglomerative lbvh construction," in *TPCG*, 2014.

[3] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, 2009.

[4] "Thinking parallel, part iii: Tree construction on the gpu," Aug 2020. [Online]. Available: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/

Appendix A Running Time Data

Filenames	# of Primitives	Config1	Config2 root	Config2 Morton	Config2 GPUconstruction	Config2 GPU total	Speed Up
teapot.dae	2464	1.5	0.0345	0.3319	0.2232	**0.5896**	2.544097693
cow.dae	5856	3.8	0.1414	0.7409	0.266	**1.1483**	3.309239746
beetle.dae	7558	4.9	0.2064	1.0034	0.3965	**1.6063**	3.050488701
bunnyNoBox.dae	30338	24.9	0.9238	4.2766	0.4821	**5.6825**	4.381874175
bunny.dae	33696	29.9	1.091	7.5012	0.4924	**9.0846**	3.291284151
peter.dae	40018	32.6	1.62	9.4221	0.7542	**11.7963**	2.76357841
maxplanck.dae	50801	42.2	1.6499	9.1248	0.7604	**11.5351**	3.658399147
beast.dae	64618	54.1	2.1368	12.2528	0.7918	**15.1814**	3.563571212
lucy.dae	100000	92.2	3.6769	23.8571	1.2044	**28.7384**	3.208250981
dragonNoBox.dae	100000	108.6	3.6719	24.588	1.1788	**29.4387**	3.689021594
dragon.dae	105120	111.6	3.7165	26.7827	1.1329	**31.6321**	3.528061684
tyra.dae	200000	231.7	7.1609	53.314	2.0569	**62.5318**	3.705314736
armadillo.dae	212574	246.5	7.9329	57.3694	2.1015	**67.4038**	3.657063845
wall-e.dae	240326	251.5	8.98	56.2905	1.5841	**66.8546**	3.761895217
happyBuddha.dae	600000	834	20.8132	169.6996	4.8038	**195.3166**	4.269990364

Figure 8: Table of run-time of Config 2, its corresponding sub-task run-time and overall speed up compared with Config 1

Filenames	# of Primitives	Config1	Config3 root	Config3 Morton	Config3 GPUconstruction	Config3 GPU total	Speed Up
teapot.dae	2464	1.5	0.0369	0.3273	0.1926	**0.5568**	2.693965517
cow.dae	5856	3.8	0.0825	0.8697	0.2088	**1.161**	3.273040482
beetle.dae	7558	4.9	0.1307	0.9058	0.2137	**1.2502**	3.9193729
bunnyNoBox.dae	30338	24.9	0.5533	1.2962	0.3668	**2.2163**	11.23494112
bunny.dae	33696	29.9	0.6372	1.4379	0.4003	**2.4754**	12.07885594
peter.dae	40018	32.6	1.4032	1.6791	0.6432	**3.7255**	8.750503288
maxplanck.dae	50801	42.2	1.2278	1.7996	0.6412	**3.6686**	11.50302568
beast.dae	64618	54.1	1.8998	2.3189	0.6765	**4.8952**	11.05164243
lucy.dae	100000	92.2	3.2222	2.9186	1.1001	**7.2409**	12.73322377
dragonNoBox.dae	100000	108.6	3.141	2.6849	1.0973	**6.9232**	15.6863878
dragon.dae	105120	111.6	3.4922	2.6388	1.0325	**7.1635**	15.57897676
tyra.dae	200000	231.7	7.1852	4.5077	1.9635	**13.6564**	16.96640403
armadillo.dae	212574	246.5	7.7011	5.1402	1.9902	**14.8315**	16.62003169
wall-e.dae	240326	251.5	8.5319	5.5812	1.4686	**15.5817**	16.14072919
happyBuddha.dae	600000	834	20.7531	11.956	4.6843	**37.3934**	22.30340113

Figure 9: Table of run-time of Config 3, its corresponding sub-task run-time and overall speed up compared with Config 1

Appendix B Mesh Views

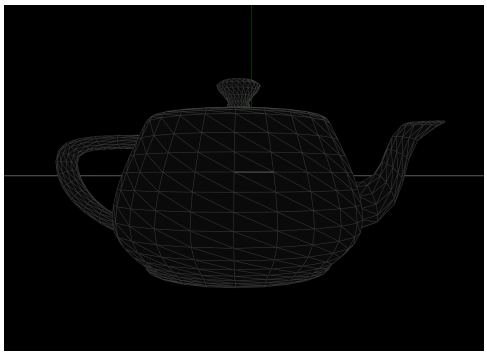

Figure 10: Mesh view: teapot.dae (2,464 primitives)

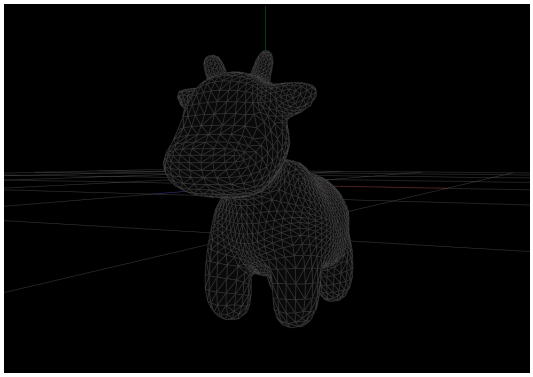

Figure 11: Mesh view: cow.dae (5,856 primitives)

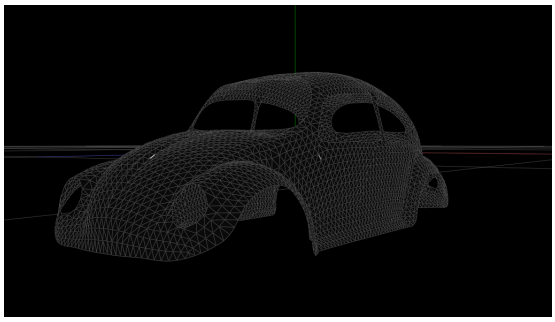

Figure 12: Mesh view: beetle.dae (7,558 primitives)

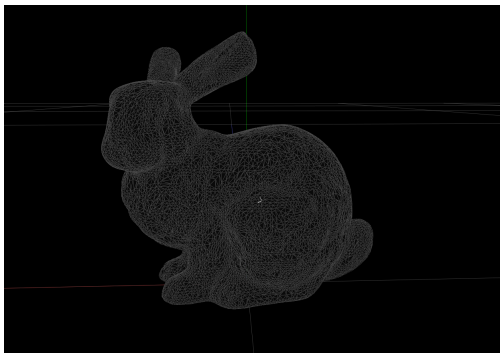

Figure 13: Mesh view: bunnyNoBox.dae (30,338 primitives)

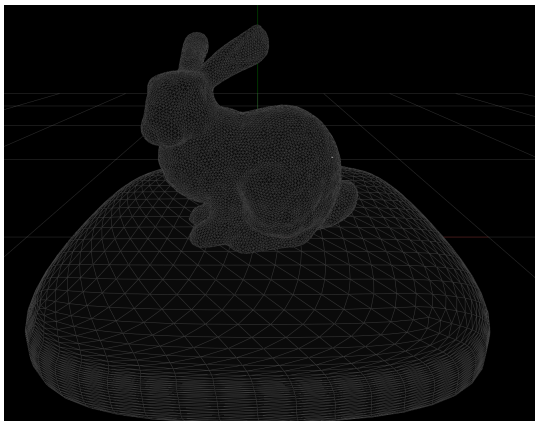

Figure 14: Mesh view: bunny.dae (33,696 primitives)

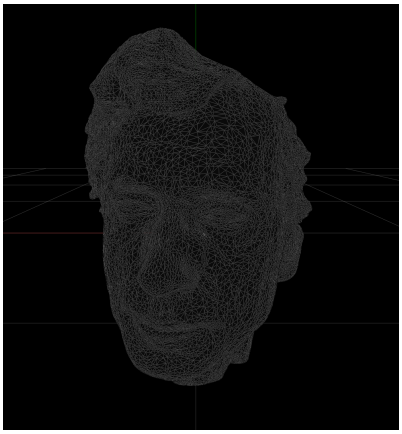

Figure 15: Mesh view: peter.dae (40,018 primitives)

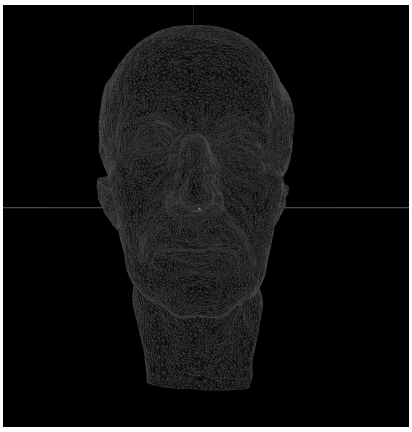

Figure 16: Mesh view: peter.dae (50,801 primitives)

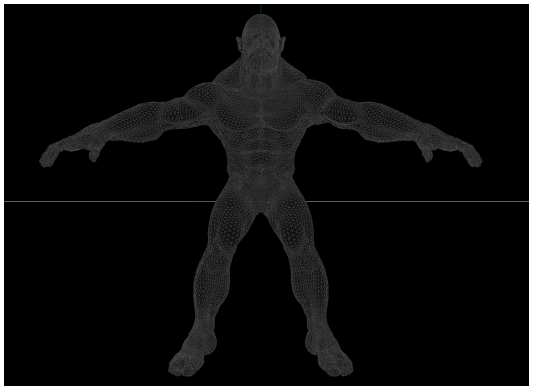

Figure 17: Mesh view: beast.dae (64,618 primitives)

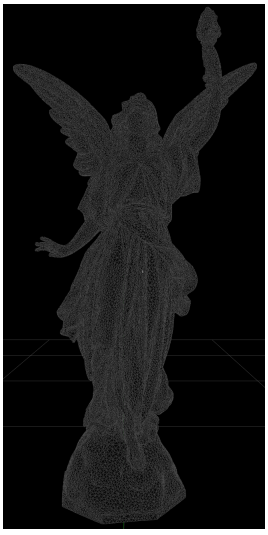

Figure 18: Mesh view: lucy.dae (100,000 primitives)

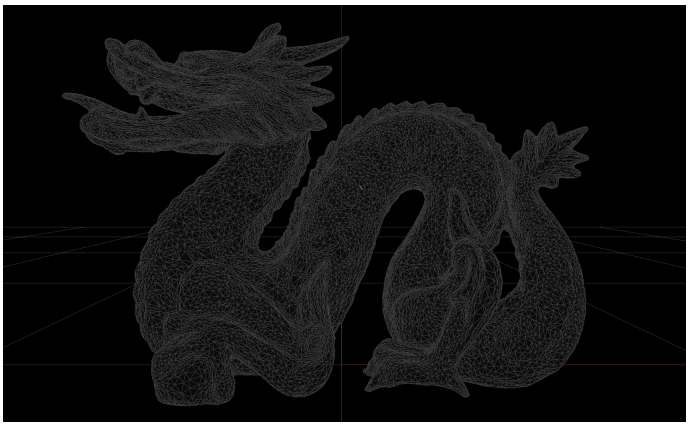

Figure 19: Mesh view: dragonNoBox.dae (100,000 primitives)

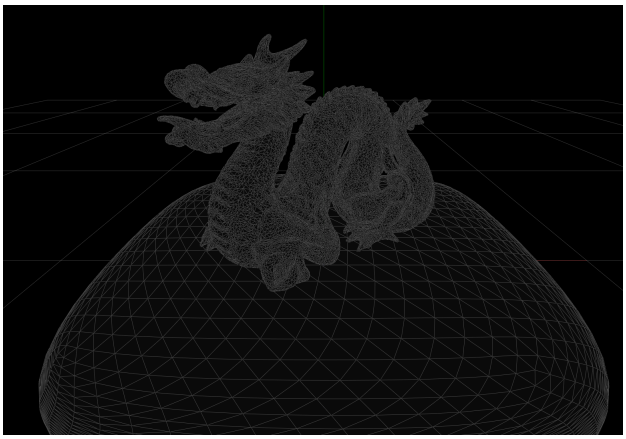

Figure 20: Mesh view: dragon.dae (105,120 primitives)

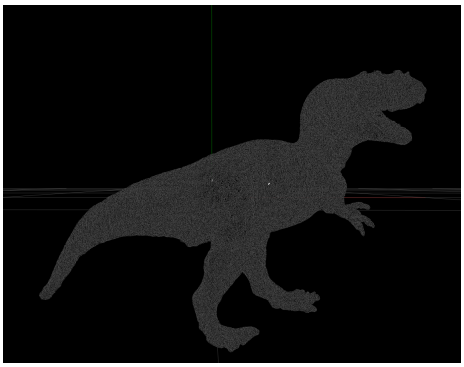

Figure 21: Mesh view: tyra.dae (200,000 primitives)

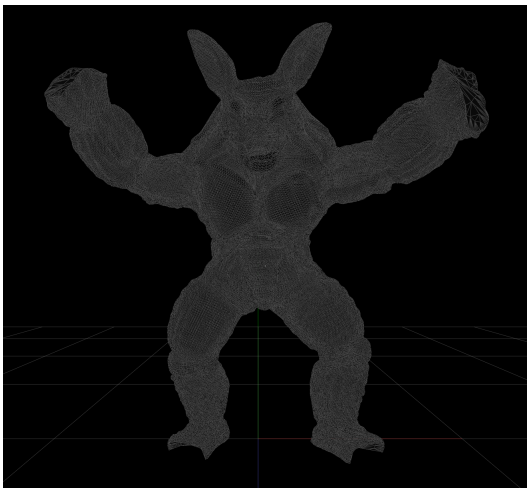

Figure 22: Mesh view: armadillo.dae (212,574 primitives)

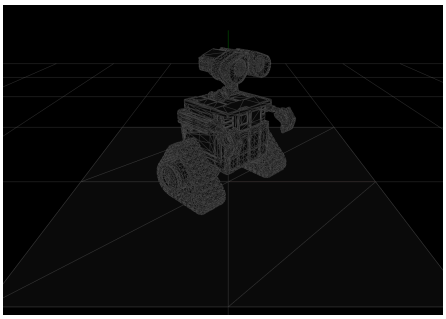

Figure 23: Mesh view: wall-e.dae (240,326 primitives)

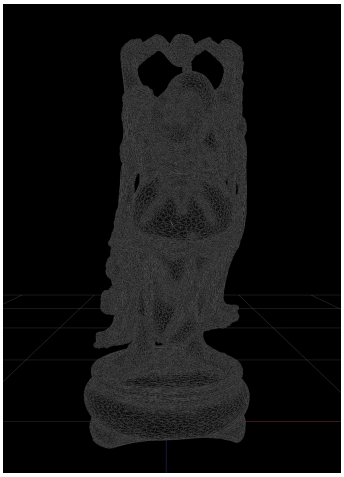

Figure 24: Mesh view: happyBuddha.dae (600,000 primitives)