GPU BASED BVH CONSTRUCTION AND TRAVERSAL ALGORITHMS FOR RAY-TRACING

Armon Carigiet, Alessandro Maissen, Bo Li and Haitong Shi

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We explore ray tracing using Bounding Volume Hierarchies (BVHs) on GPU. Parallel BVH construction as well as several BVH traversal algorithms are implemented using CUDA. We experiment and benchmark our implementation. Furthermore, some potential improvements are proposed and analyzed.

1. INTRODUCTION

Ray tracing is a fundamental operation in computer graphics for emulating the transport of light. Rays are cast into a scene of geometric primitives, where the goal is to find all primitives that the given ray intersects. Many important problems are solved using this technique, such as global illumination and visibility tests.

Traditionally, offline rendering using ray-tracing is slow on CPUs and can take several minutes to several days per frame. With the development of many-core devices like GPUs, real-time graphic applications for ray-tracing are becoming more popular. However, they have a harsh requirement for performance, typically with a budget of tens of milliseconds per frame. Therefore the efficiency of ray-tracing algorithms has become even more important.

Normally all primitives would have to be tested against a ray to decide if there is an intersection or not, however Bounding Volume Hierarchies (BVHs) can be used to accelerate this process. A BVH organize a scene into a tree, where primitives are stored in the leafs and the internal nodes contain bounding boxes that encapsulate all the primitives in their subtrees. This allows a ray-tracing algorithm to simply skip a subtree where the ray does not hit its bounding box. In this paper we will discuss both the construction and traversal of BVHs on GPUs.

BVHs can be constructed in different ways. There are algorithms that aim for fast construction [1, 2], while others try to generate tree hierarchies with better properties [2] to improve performance of traversal. We take a look at a fast construction algorithm [1], which increases parallelism by computing the position of each node in the tree hierarchy independently of its parent. Furthermore, we evaluate the algorithm's scalability, test how well it performs on more recent GPU hardware and compare against a ray-tracing engine named Optix [3]. Finally we try to apply CUDA's concept of shared memory to optimize performance even more.

Parallel BVH traversal is often carried out recursively. This however requires enough memory to store a traversal stack for each ray that is processed in parallel [4]. To alleviate this problem several stackless algorithms have been proposed [4, 5, 6]. For stack-based traversal an extensive evaluation was performed by Aila et al. [7, 8], considering different traversal and GPU optimization strategies. Unfortunately, to the best of our knowledge, no such broad evaluation has been done for these stackless traversal algorithms. Therefore we want to expand the analysis of a stackless algorithm by Hapala et al. [4] in this paper. To accomplish this we implement their algorithm using the different strategies proposed by Aila et al. and benchmark it against several stack-based implementations.

2. BACKGROUND

In this section we give some background knowledge about the construction and traversal of BVHs.

BVH Construction. Most BVH construction algorithms follow a top-down approach: the aggregated axis-aligned bounding box (AABB) of all primitives in the root node is first calculated, then recursively split into two bounds such that the primitives whose centroids inside either bound formulate a subtree. The splitting may employ different strategies. For example, in a way that the count of primitives in two subtrees is equal. To favor efficient intersections, a more complex strategy called surface area heuristic (SAH) [2] is proposed. It provides a cost model, based on which the optimal split is found among a number of possible positions.

While high-quality BVHs can be constructed using SAH, it is extremely slow because of the large amount of work required to compute SAH costs. Moreover, the splitting of subtrees can only start after their parent node is constructed. This recursion imposes level-by-level data dependency and makes parallelization difficult. On the other hand, Linear Bounding Volume Hierarchies (LBVHs) are more suitable to implement on massively parallel GPUs. A LBVH arranges primitives in 3D space into a 1D array sorted using Morton codes [1]. The sorted linear array of primitives can be divided into segments that are processed in parallel, where each subtree is guaranteed to correspond to a continuous segment. We will mainly focus on the parallel construction of LBVHs in this work.

BVH Traversal. We refer to the process of finding a ray intersection against some primitives using a BVH tree as Ray Traversal. To do this, the tree is traversed top-down while only visiting the relevant parts. Generally this is done by first checking the bounding boxes of the two children of the root node. If the ray does not hit the bounding box of a node, one can be sure that no primitives contained in the node's subtree are hit by the ray, allowing the procedure to skip this subtree. Otherwise, if the bounding box was hit, the traversal continues recursively on that node. When the recursion arrives at a leaf of the BVH tree, the ray is tested against the leaf's primitives and intersections are stored accordingly. This process is done until all the relevant nodes have been processed, returning either an intersection or an indication that no intersection is found.

3. FAST PARALLEL BVH CONSTRUCTION

Karras [1] proposed a parallel algorithm for LBVH construction. We implement his algorithm for more recent GPU hardware, analyze performance and scaling of different stages of the algorithm. In this section we will first explain the basic steps of the algorithm. Afterwards we introduce a concrete idea about how to apply shared memory since the original paper briefly proposed this as a possible optimization. Finally, we shortly introduce our corresponding codebase.

Algorithm. In this section we briefly explain the main stages of the algorithm, while we refer to the literature for a detailed description. Basically, Karras algorithm can be divided into four stages:

Stage 1: Morton code generation. Given a 3D primitive and the corresponding centroid (x, y, z) of its AABB as binary fractions, the corresponding Morton code can be formed as $x_0y_0z_0x_1y_1z_1\dots$, where $x = 0.x_0x_1x_2\dots$ and for y,z respectively. Observe that this can be done independently for each primitive.

Stage 2: Sorting primitives. The primitives are sorted according to their Morton codes, which can efficiently be done with a parallel version of radix sort. This has the effect that primitives close to each other in 3D space are likely to end up nearby in the sorted sequence.

Stage 3: Building tree hierarchy. A tree hierarchy is generated for the primitives by constructing a binary radix tree and treating their Morton codes as keys. Observe that the leafs represent the keys and each internal node represents the longest common prefix among all keys in its subgraph. Moreover, note that for n keys the tree consists of exactly $n - 1$ internal nodes and n leaf nodes. To efficiently construct this hierarchy in parallel, the problem boils down to compute the following for each internal node independently: (1) Determine the range of covered keys. (2) Find the position where to split the range into sub-ranges. (3) And finally compute the pointers to the corresponding child nodes.

Stage 4: Bounding box computation. Given the tree hierarchy from the previous stage, it remains to compute the bounding boxes. Observe that the bounding box of each node is the union of its children's bounding boxes. Obviously those computations can not be done completely independently. To aim for as much parallelization as possible, Karras suggests the following strategy: a thread is initiated at each leaf node and walks towards the root using parent pointers which are set during the tree hierarchy construction stage. The first thread to visit an untouched node will mark the node as visited using atomic counters, then it terminates and lets the second thread compute the bounding box. Finally, only one thread will reach the root node and all nodes will be processed exactly once.

Shared memory optimization. The CUDA memory hierarchy distinguishes between global memory and shared memory, where the former is visible to all threads and the latter is shared among threads in the same thread block. Since shared memory is located on-chip, its latency is about 100x times lower than uncached global memory. Therefore we would like to take advantage of shared memory to improve the computation of bounding boxes (Stage 4 of the construction algorithm) as Karras suggested. Recall that a bounding box is computed by the union of its children's bounding boxes. Since the tree hierarchy completely lies in global memory, this computation requires to access global memory many times. Hence the idea is to allocate shared memory such that each thread additionally stores the recently computed bounding box in shared memory. In the case that both children were processed by threads of the same thread block, one can load corresponding bounding boxes directly from shared memory instead of loading them from global memory. Nevertheless, one have to keep in mind that usage of shared memory introduces thread synchronization overhead and that CUDA's caching strategy might outperform our shared memory strategy. This is indeed the case as we will see in Sec. 5.

Codebase. To facilitate the implementation of the algorithm, we borrowed some fundamental parts like basic vector, matrix and triangle data types from the open-source renderer pbrt [9]. Our implementation comes with unit tests, a benchmarking system, a plotting infrastructure and is publicly available[1].

4. BVH TRAVERSAL

BVH traversal is often carried out in a recursive manner using a stack. However, these approaches need enough memory to maintain a stack for each ray that is being traced at the same time [4]. This can be a problem in settings where a lot of rays are processed in parallel [4]. Several algorithms have been proposed to lower this memory requirement by performing a *stackless* BVH traversal [4, 5, 6]. The approach by Hapala et al. performs the same amount of bounding box intersections and also allows for the same traversal order as a stack-based variant would [4]. However, since it doesn't use a stack, it needs to visit each BVH node twice. It was benchmarked against a stack-based implementation of Aila [7] on a Monte Carlo path tracer [10] in their original paper, but a more detailed analysis in the same fashion as the evaluation of Aila et al. was not carried out.

One of our main goals is to expand the analysis of this stackless algorithm to the methodology of Aila et al., considering different ray types, traversal strategies and optimizations. Next we also wanted to see how this stackless algorithm performs in comparison to the stack-based implementations of Aila et al. on newer GPU hardware, since both original papers runs their experiments on older architectures like Kepler and Fermi.

Codebase. Since no code artifact was made available for the paper of Hapala et al. [4], we implemented their stackless algorithm directly into the framework of Aila et al. [11] as a CUDA kernel. Moreover we adapted existing benchmark infrastructure and added an automatic plotting infrastructure. Our implementation is publicly available[2].

Stackless Algorithm. Here we want to shortly describe this stackless BVH traversal algorithm. For a more detailed explanation we refer to the original paper [4]. The algorithm defines for each internal node a "near child" c_n and a "far child" c_f. By always first visiting c_n a deterministic traversal order is induced. Then there are three states which can occur during the traversal when we arrive at a node p, each of them uniquely defining the next node in the traversal order (See Fig. 1), which allows us to traverse the BVH tree using a simple state machine.

Traversal Strategies & Optimizations. Next, as mentioned at the start of this section, we discuss the different optimization approaches that we explored for the stackless traversal algorithm and discuss potential advantages and drawbacks for each strategy. We implemented several stackless traversal kernels using one or multiple of these strategies to evaluate them empirically.

[1]https://github.com/DPHPC-BVH/BVH-Construction
[2]https://github.com/DPHPC-BVH/BVH-Traversal

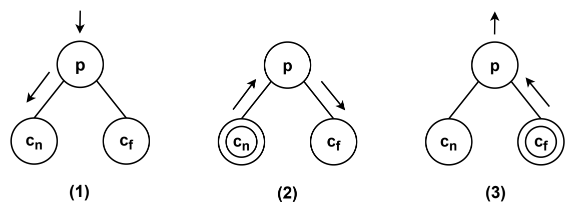

Fig. 1. States of the stackless traversal algorithm at the node p. Arrows indicate the traversal order before and after p. A double circle indicates the last node the algorithm visited before p. In state (1) the algorithm has not visited both of p's children, implying that the near child c_n should be visited next. In state (2) the traversal returned from c_n, which leaves the subtree of c_f to be explored. When we are in state (3) both subtrees of c_n and c_f have been explored and the algorithm can return form p to its parent.

Speculative Traversal (SP): Since different rays intersect with different bounding boxes of the BVH tree, different GPU threads will take different amounts of time to reach a leaf during traversal. GPU threads are bundled intro groups of 32 called warps. At the hardware level, instructions of threads in a warp are then executed in a SIMT fashion. On architectures before Volta all threads in a warp shared the same program counter, implying that an entire warp has to wait for all of its threads to arrive at a leaf before it can start the primitive intersections. Therefore Aila et al. proposed to let threads speculatively continue the traversal for additional leafs, until all of the other threads in the warp have found at least one [7]. These speculative kernels can be especially beneficial if one searches for geometry intersections that are closest to the ray origin. But they also need to use warp-level primitives [12] to synchronize the traversal and intersection stages, which introduces an overhead.

Now on later architectures "Independent Thread Scheduling" was introduced giving each thread a separate program counter, therefore allowing them to diverge in execution. Since this significantly changes the way how warp-level primitives work [12] we had to specify the virtual architecture as pre-Volta when compiling to make the baselines and our speculative kernels work on newer GPUs.

Traversal Order (I): The order in which a BVH is traversed (i.e. which of the two children is visited first) can also impact the traversal performance due to memory accesses being different. Hapala et al. suggested, for each internal node, to use the axis in which the two child-nodes' bounding boxes are the farthest apart. The order is then decided based on the ray's sign along that axis. Alternatively one can also just use the order that is implicitly given during construction. This eliminates some additional computation that the first approach needs during traversal, but may lead to worse memory accesses depending on the rays that are being traced.

Texture Caches (T): On GPUs global memory and texture memory are loaded through different caches using different memory lanes. This allows texture buffers to be used to relieve pressure on the global memory bandwidth [13]. We implement two versions of the speculative stackless kernels. One version that uses only texture buffers (T) and another which uses global memory buffers to read the BVH nodes itself and texture buffers to read the primitives that are stored with the leafs.

Persistent Threads (P/NP): Aila et al. used persistent threads to implement their stack-based kernels. This technique can be used to bypass the thread scheduling by the hardware. Instead of assigning one thread to one ray that has to be traced, a fixed number of threads are created (usually as many that are needed to fill all SMs), which continuously fetch new rays to process from a queue. To see if this also improves the stackless algorithm we have implemented two kernel versions with persistent threads (P) and one without (NP). Additionally we have also made persistent versions of the speculative stackless kernel and the stackless kernels using the independent traversal order, to analyze how persistent threads impacts the performance of these different traversal strategies.

Dynamic Ray Fetching (DF): Another performance problem can be that exceptionally long running threads in a warp can stall most of the SIMD lanes of a SM [7]. When using persistent threads this problem can be addressed by dynamically fetching new rays for a warp if to many of its threads have already been terminated [7, 8]. Again this introduces an overhead since warp-level primitives [12] are needed to find out when a warp has not enough active threads [7].

5. EXPERIMENTAL RESULTS

In this section we first briefly talk about the setup for our experiments and how the benchmarking was performed. Next we introduce the state-of-the-art ray tracing framework Optix [3] with which we compare our implementations. Then we will present the results of our experiments on the implemented BVH construction algorithm and continue with the BVH traversal experiments in a separate paragraph.

Experiment setup. All of our experiments have been performed on a machine with a Nvidia RTX 2060 GPU and a AMD Ryzen 4800H CPU. The implementations use CUDA 11.4 on Windows 10. The C++ code is compiled using MSVC 14.28 with optimization `/O2`. The CUDA code is compiled with flags `-O3`, `-use_fast_math`, `-code sm_75` (binary target hardware) and `-arch compute_62` (virtual architecture, only for traversal kernels, see Sec. 4).

Measurements. Because GPUs are asynchronous computing devices, instead of measuring time on CPU we use a GPU timer based on CUDA event timers [14]. Moreover following benchmarking guidelines [15], we tested the normality of all measurements by either inspecting a Q-Q plot or by the Shapiro–Wilk test. We found that our measurements are not normally distributed and therefore use the median and 95% confidence interval (CI) around the median to reason about performance. Additionally, if variance is large enough we prefer boxplots over barplots, which is only the case for traversal. In any case, we display the relative maximum error between the median and bounds of the 95% CI among all data series in the plot. We do 50 time measurements and 2 warm-up rounds which resulted in reasonably tight CIs. Depending on the experiment, we choose one or more of the following scenes from the Nvidia framework [11]: sibenik, fairyforest and conference with 80.131K, 174.177K and 282.759K primitives, respectively.

External baseline. We compare our implementations with a state-of-the-art GPU ray tracing engine Optix [3]. Optix is a programmable pipeline, allowing the user to construct the BVH and define the behaviour of rays (how primary rays are spawned, how to deal with intersections, etc.) in a general way through shaders. We use Optix to build BVHs for comparison of the construction part, and implement an ambient occlusion renderer for comparison of the traversal part.

Construction. As mentioned in Sec. 3 the construction algorithm can be divided into four stages. Therefore in our first experiment, we benchmark each stage separately. Fig. 2 shows absolute and relative execution time of each stage for different scenes. One can clearly see that sorting the Morton codes and computing the bounding boxes account for the most of the execution time (around $70-85\%$). This is actually what we expected since both stages have thread dependencies while the remaining stages have not. Since we used a parallel radix sort implementation of CUB [16], there's not much we could do to improve performance of sorting the Morton codes.

On the other hand as mentioned in Sec. 3, we implemented a kernel that makes use of shared memory to compute the bounding boxes. Therefore in the next experiment we compare both implementations to explore the benefits of using shared memory in newer GPU hardware. On the right hand side in Fig. 3 this comparison is shown. We observe that the shared memory implementation is about $60\% - 100\%$ slower. Remember that shared memory is only faster than uncached global memory. Apparently CUDA's caching strategy already does a great job, while usage of shared memory additionally introduces synchronization overhead. This might be the reason why the shared memory implementation is slower. Moreover this result gives as well a clue about the improvement of CUDAs caching strategies compared to older versions used in the original paper.

Following experiment compares our implementation with the state-of-the-art ray tracing engine Optix. The plot on the left of Fig. 3 shows that our implementation seems to be faster than Optix but there are some considerations we need

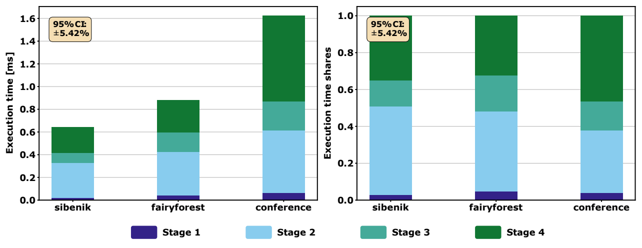

Fig. 2. Absolute execution time (left) and execution time shares (right) of all stages for different scenes for our implementation of the LBVH construction algorithm.

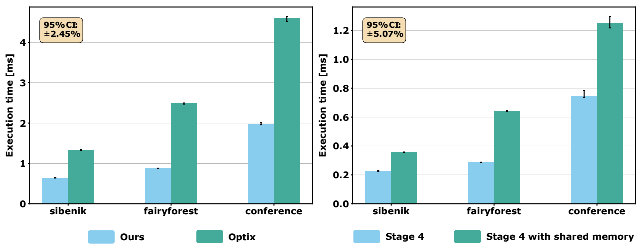

Fig. 3. Comparison with Optix (left) and impact of shared memory on Stage 4 (right) on different scenes. The vertical bars indicate the 95% confidence interval around the median.

to make. Firstly, in our implementation we stop after a tree hierarchy represented with pointers is created, while Optix may further flatten the tree to a pointerless representation. Moreover, our algorithm creates a standard LBVH, which is rather slow for ray-tracing. As mentioned in Sec. 2, BVHs are often combined with the SAH strategy, which makes construction slower but on the other hand accelerates BVH traversals. Optix most likely aims to optimize the entire ray-tracing pipeline, which includes BVH construction and BVH traversal. This might be one of the main reasons why our construction algorithm is faster than Optix. Nevertheless, the key take-away point of the comparison with Optix in Fig. 3 is that we show that our performance is at the same magnitude as state-of-the-art libraries.

As a final experiment we also examined the strong scaling behavior of the construction. There is no direct way to control the number of cores to use on a GPU, but we know that a CUDA thread block is attached to one SM [17]. So we can specify the number of blocks to launch, and then assign each thread $\lceil \text{nPrimitives}/(\text{nThreadsPerBlock} \times \text{nBlocks}) \rceil$ tasks (the denominator in the formula is simply the number of all threads launched). The result is shown in Fig. 4. The Nvidia RTX 2060 has 30 SMs, but the scaling seems to stop when the number of blocks approaches 15. We suspect that some resources are shared among the SMs and hence saturated (e.g. global memory bandwidth). Using a small number of blocks, the stage that builds the tree hierarchy scales linearly with the number of cores. The entire algorithm does

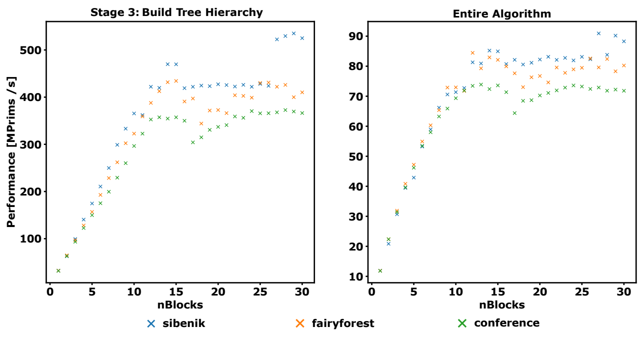

Fig. 4. Scaling of construction performance with the number of CUDA thread blocks. Note that the scaling of adding more blocks (SMs) stops when nBlocks approaches 15. Thereafter, the performance is not directly related to nBlocks but to how many tasks each thread processes (not shown in the figure). The optimal performance is finally achieved when one thread processes one task under large enough nBlocks, so that the hardware has maximum freedom to schedule threads.

not, because radix sort and the bounding box computation are not linearly scalable.

Traversal. For the traversal we evaluated each stackless kernel that we implemented (and the stack-based reference kernels) by rendering all test scenes over multiple camera angles for primary, ambient occlusion and diffuse interreflection rays. Due to space constraints we present results only for the conference scene, but note that the benchmarks for the remaining scenes are similar. We use the benchmarking approach from the framework of Aila et al. [8] where the rays to be traced for one frame are grouped into batches, which are then traced multiple times to receive multiple time measurements. The ray tracing performance, reported in million rays per second (Mrays/s), is then calculated by summing up all the rays over different camera angles for one frame and dividing them by the total amount time it took to trace these rays. Furthermore, we also compared the traversal performance against Optix, but omitted the detailed data to save space.

In general we can see that the stackless kernels are around 30-50% less performant than the stack-based implementations of Aila et al. (Fig. 5) due to the additional amount of computation that is performed. This is in line with the results of Hapala et al. that reported the stackless algorithm being around 30% slower than the stack-based variants when evaluated on a path tracer. Similar to the GPU that Hapala et al. used for their experiments, it does not seem to be a problem for a RTX 2060 to maintain one traversal stack per ray. In Fig. 5 we can also see that among the stackless kernels the versions using speculative traversal (SP) seem to perform better than the regular variants (N/NP), while the dynamic fetch (DF) variant seems to perform worse. The results in Fig. 5 are given for ambient

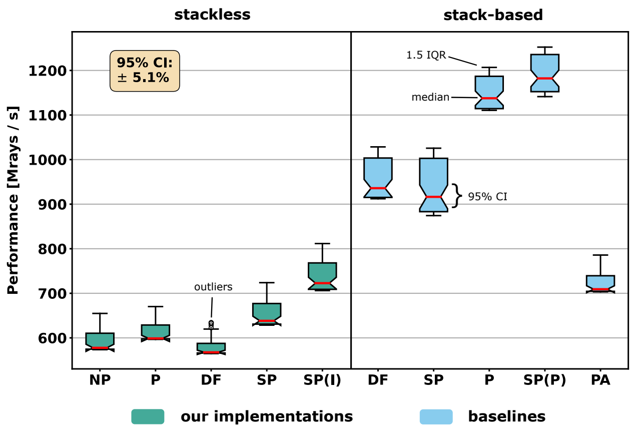

Fig. 5. **Performance comparison** between the implemented **stackless** kernels (left) and the **stack-based reference kernels** of Aila et al. [7, 8] (right) for conference. The kernels using different optimizations and traversal strategies are labeled as follows: **P**: using persistent threads. **NP**: non-persistent threads. **DF**: dynamic fetch. **SP**: speculative. **SP(P)**: persistent + speculative. **SP(I)**: speculative + independent traversal order. **PA** is a packet tracing kernel [18] from [7], included for completeness.

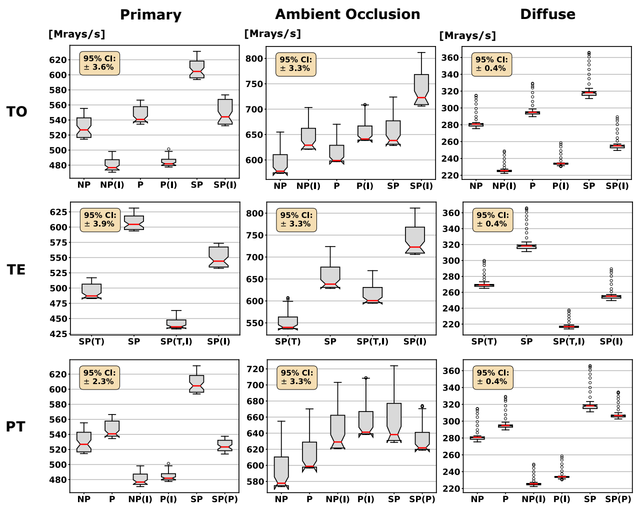

Fig. 6. **Performance of different optimized stackless kernels** for conference. We use the same labels for the kernels as in Fig. 5. To disambiguate the different optimizations, the identifiers (I,T,P) are appended. **TO**: Benchmarks for ray dependent and independent *(I)* traversal orders. **TE**: Benchmarks using only texture buffers *(T)* and splitting loads between global and texture memory. **PT**: Benchmarks with *(P)* and without persistent threads.

occlusion rays. However primary and diffuse rays follow a similar pattern, which we omitted for brevity.

Next we discuss the results for the optimization techniques from Sec. 2, which are summarized in Fig. 6 for all ray types.

Profiling showed that the kernel versions using only the texture cache (T) had a low computation throughput due to having to wait for memory loads to be completed. Splitting memory loads between the global and texture cache improved the computation throughput and also the ray-tracing performance that we measured (see Fig. 6, TE).

Then the usage of persistent threads seemed to improve the performance of the stackless kernels, with the exception of the speculative (SP) variant where we actually noticed a decrease in performance especially for primary rays (see Fig. 6, PT).

Having a traversal order that is ray independent, and therefore avoids additional computations during the traversal, seems to boost the performance for ambient occlusion rays. However, for Primary and Diffuse rays it seems to be detrimental (see Fig. 6, TO).

Finally we also tried to compare our traversal performance against Optix. Our implementation takes 4-6x longer to render a frame. It is however not very comparable to Optix for two reasons. Firstly, there are many other factors affecting the final frame rate like the generation of rays and management of ray buffers besides the traversal itself, which are out of our control since Optix is closed-source. Secondly, the Turing architecture of the GPU we used has dedicated RT Cores for accelerating ray tracing [19]. While RT Cores benefit Optix at the driver level, they are not applicable to our pure CUDA implementations.

6. CONCLUSIONS

Our project focused on several aspects of parallel construction and traversal of BVHs on GPUs. We implemented a parallel LBVH construction algorithm on recent GPU hardware and showed that it can compete with state-of-the-art frameworks like Optix. Our results for the scalability analysis and stage analysis match the results of the original paper. However, we showed that the use of shared memory does not improve performance on newer GPUs as it did in the implementation in the original paper. On the other hand for BVH Traversal we have implemented a stackless algorithm using different traversal & optimization strategies to (1) explore the impact of these strategies and (2) compare them against existing stack-based implementations. We showed that most explored optimizations show an improvement in performance and that this improvement is not always the same for different types of rays. Further we extended the analysis of Hapala et al. and showed that our implementations perform similarly to what they reported.

Our work covers the cornerstones of GPU ray tracing, however it is not comparable to a practical ray tracing engine like Optix. While outside the scope of this paper, the balance of traversal and construction speed for BVHs could be explored in future work.

7. REFERENCES

[1] Tero Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Goslar, DEU, 2012, EGGH-HPG'12, p. 33–37, Eurographics Association.

[2] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[3] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich, "Optix: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, jul 2010.

[4] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek, "Efficient stack-less bvh traversal for ray tracing," in *Proceedings of the 27th Spring Conference on Computer Graphics*, New York, NY, USA, 2011, SCCG '11, p. 7–12, Association for Computing Machinery.

[5] Samuli Laine, "Restart trail for stackless bvh traversal," in *Proceedings of the Conference on High Performance Graphics*, Goslar, DEU, 2010, HPG '10, p. 107–111, Eurographics Association.

[6] Brian Smits, "Efficiency issues for ray tracing," *Journal of Graphics Tools*, vol. 3, no. 2, pp. 1–14, 1998.

[7] Timo Aila and Samuli Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the Conference on High Performance Graphics 2009*, New York, NY, USA, 2009, HPG '09, p. 145–149, Association for Computing Machinery.

[8] Timo Aila, Samuli Laine, and Tero Karras, "Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum," .

[9] Matt Pharr, Wenzel Jakob, and Greg Humphreys, "pbrt-v3," `https://github.com/mmp/pbrt-v3`, 2018.

[10] James T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1986, SIGGRAPH '86, p. 143–150, Association for Computing Machinery.

[11] Tero Karras, Timo Aila, and Samuli Laine, "Understanding the efficiency of ray traversal on gpus," `http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/`.

[12] Yuan Lin and Vinod Grover, "Using cuda warp-level primitives," `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives`, 2018.

[13] "Chapter 5 - cuda memory," in *CUDA Application Design and Development*, Rob Farber, Ed., pp. 109–131. Morgan Kaufmann, Boston, 2011.

[14] Mark Harris, "How to implement performance metrics in cuda c/c++," `https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/`, Aug 2020.

[15] Torsten Hoefler and Roberto Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[16] Nvida Cooperation, "cub," `https://github.com/nvidia/cub`, 2021.

[17] Pradeep Gupta, "Cuda refresher: The cuda programming model," `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`, Jun 2020.

[18] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 113–118.

[19] NVIDIA Corporation, "NVIDIA Turing GPU Architecture Whitebook," 2018.

8. CONTRIBUTIONS OF TEAM MEMBERS

Because of very unequal contribution of team members to the project and on behalf of our supervisor Timo Schneider this section lists the contributions of each team member. Authors marked with a * have equally contributed to the project.

Armon Carigiet*. Implemented the stackless algorithm and all its versions using different optimizations and traversal strategies except early break. Wrote the plotting and data analysis framework for traversal. Modified and extended the existing benchmarking environment of the Nvidia framework for traversal. Performed the experimental evaluation of the traversal part. Wrote about the corresponding parts in the report together with Bo.

Alessandro Maissen*. Implemented all versions of the BVH construction algorithm, one with shared memory and one version without. Implemented the entire benchmark infrastructure for the construction part, and added an automated plotting infrastructure for construction. Implemented all the experiments for construction with its plots except the scalability experiment which was performed by Bo. Wrote the parts in the report concerning above listed work.

Bo Li*. Implemented the fundamental framework based on pbrt and the scaling experiment of the BVH construction. Explored warp intrinsics issues and the early break strategy for the traversal part. Implemented the Optix baseline. Helped Alessandro with design decisions concerning the construction algorithm. Helped debugging on traversal and construction. Moreover ran all implemented benchmarks for the data in the report. Wrote both on the traversal and construction part of the report.

Haitong Shi. Added a Q-Q plot to the automated plotting infrastructure and was responsible for making the scalability plot. Helped with checking the report for grammatical errors.