GPU Assignment
Ray Tracing on GPU with BVH-based Packet Traversal

Natálie Kaslová[1]
Department of Computer Graphics and Interaction,
Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract

Implement a ray tracer with bounding volume hierarchies (BVH) that uses a cost model based on surface area heuristics. Implement ray-packet traversal on the GPU, assuming that the rays are formed by casting coherent primary rays.

Test the implementation on scenes consisting of triangles with varying numbers of objects. Report the performance for primary rays and shadow rays (using point light source).

Keywords: ray tracing, BVH, bounding volume hierarchy, packet traversal, GPU, CUDA

1. Introduction

The data structure is built based on the paper 'Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies' [1]. As described in the paper, it builds Bounding Volume Hierarchies using a variant of the Surface Area Heuristic (SAH). The ray tracing process originally employs ray packets. However, this part has been replaced with ray tracing based on the paper 'Realtime Ray Tracing on GPU with BVH-based Packet Traversal' [2].

The implementation also utilizes packet traversal, with the traversal algorithm accelerated through GPU parallelization using a shared stack.

2. Algorithm Description

The top-down build of the BVH structure is performed on CPU with a cost model based on SAH. It produces a structure of tree nodes requiring 32 bytes. This structure is then traversed with ray packets.

2.1. BVH Construction

The construction of a BVH using the SAH begins by calculating the bounding box for the entire set of objects to establish the root node. Each node in the hierarchy is evaluated to determine whether it should be a leaf node or undergo further subdivision. If the number of objects within a node falls below a predefined threshold, it is designated as a leaf node, and no additional splitting occurs. Otherwise, the algorithm evaluates potential split candidates (centroids of each object) along the three axes (x, y, and z) to identify an optimal division.

[1]B4M39GPU – Natálie Kaslová, winter semester 2024/25

For each axis, the algorithm assesses split planes to divide the objects into two groups—left and right—based on a cost function derived from the SAH. This cost function considers the surface areas of the bounding boxes for the left and right subtrees, the distribution of objects between the groups, and a probability factor that reflects traversal efficiency. The split yielding the lowest cost is selected, and the objects are partitioned accordingly to create two child nodes.

This recursive process is applied to each child node until the object count within a node is small enough to form a leaf node. The result is a BVH structure comprising a root node and a hierarchy of recursively partitioned child nodes. Each node contains references to its associated objects and their bounding boxes, ensuring efficient spatial organization for subsequent operations.

2.2. Ray Tracing

The algorithm maps one thread to one ray and a block to a packet. The whole packet traverse synchronously one node, therefore stack is places to shared memory.

If the node is a leaf, it intersects the rays in the packet with the contained geometry. Each thread stores the distance to the nearest found intersection. If the processed node is not a leaf, the algorithm loads its two children and intersects the packet with both of them to determine the traversal order. Each ray determines which of the two nodes it intersects and in which it wants to go first by comparing the signed entry distances of both children. If an entry distance of a node is beyond the current nearest intersection, the ray considers the node as not being intersected. The algorithm then makes a decision in which node to descend with the packet first by taking the one that has more rays wanting to enter it. If at least one ray wants to visit the other node then the address of this other node is pushed onto stack. In case all rays do not want to visit both nodes or after the algorithm has processed a leaf, the next node is taken from the top of the stack and its children are traversed. If the stack is empty, the algorithm terminates. The decision, which node has more rays wanting to traverse it first, is made using a parallel sum reduction. Each thread writes a 1 in an own location in the shared memory if its ray wants to visit the right one first, and -1 otherwise. The packet takes the left node if the sum is smaller than 1 and the right one otherwise.

2.3. Shadows

Additionally shadow rays were implemented. The point light is places above camera. The parameter defined by user is used as multiplier of scene height, how much above camera position should the light be placed. After traversal of primary rays, the shadow ray is initialized, the stack is restarted and the packet traverse packet of shadow rays. After first intersection is found, the thread sets flag and does not participate in showing interest to visiting other nodes.

For better visualization of the shadows, the phong reflectance model was implemented.

3. Implementation details

The assignment was implemented in C++ and CUDA based on the framework nanogolem 0.97 [3]. A rought estimation of the time spent on the project is approximately 130 hours.

3.1. BVH Construction

I created my own struct to be used during the construction process, storing the object centroid and bounding box. This resulted in a significant improvement in performance. Another optimization was to pre-order objects for each axis into separate vectors and, during the creation of children, only divide these vectors—there was no need to sort them again.

The threshold for determining when a leaf is small enough to not be partitioned was set to the average number of objects per leaf to be between 1 and 2, as these values provided the best results.

3.2. CPU traversal

Since nanogolem framework [3] initially works with BVH, the original CPU-based basic ray tracing was used and only modified by adding shadow processing.

3.3. GPU initialization

The GPU processing code required mirrored functions and structures corresponding to the CPU definitions necessary for traversal.

Once the BVH construction was completed, an array of primitives and nodes was created using data types designed for GPU traversal and copied into the GPU's global memory. Similarly, data defining the scene parameters—such as camera parameters, point light positions, background color, shadow usage flags, and Phong model parameters—were also transferred to the GPU. Finally, an output array for storing the computed color values was initialized.

The size of shared memory is 366 bytes composed by:

- Stack: size of BVH node * 50 (it is fixely defined size)
- Children pointers: size of BVH node * 2
- Reduction memory: warp width * warp height * size of int, used for parallel sum reduction during the decision process to determine which child should be visited first

The first step in the algorithm requires all threads to decompose the shared stack, as it is composed of different types.

3.4. GPU traversal

The project contains one kernel with the name *SharedStackTraversal* located in a file *traceGPU.cu*.

The size of a ray packet is limited by the WARP size, which consists of 32 threads. My initial idea was to use only 30 of them, creating a packet of 5x6 rays. Another option was a packet of 8x4 rays, utilizing all 32 threads. I initially thought that using a smaller but more compact packet would be beneficial, even though it involved using 2 fewer threads. However, after conducting some tests, I concluded that the larger and narrower 8x4 packet performs slightly better, as can be seen in figure 1. As mentioned in the paper about BVH construction [1], during their implementation of ray-packet traversal, the best performance was achieved with ray packets of 8x8 and 16x16 rays. Therefore, my concern about an inappropriate packet shape turned out to be unfounded.

Since the image is ray traced in packets, they do not always fit perfectly within the image resolution. Therefore, the image is aligned to fit the packet layout, and after traversal the additional padding is ignored.

In the paper about GPU ray tracing [2], there is an error in the pseudocode. The selected child to be entered next is replaced with the other child, resulting in incorect results.

Since the ordered traversal is determined by the number of threads attempting to enter each child, the variables that determine the order can be excluded from the BVH node definition. Despite this exclusion, the size of the node remains the same due to memory alignment.

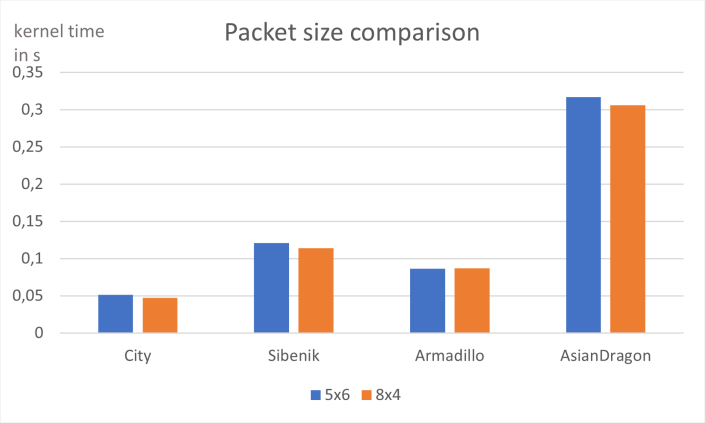

Figure 1: Measured kernel time with different packet size for different scenes.

3.5. Shadows

I faced the biggest difficulties during shadow implementation. My generated images had shadow acne, that i could not get rid of. Moving illuminated point slightly above suface or ignoring intersections lower than some epsilon value did not help. My resolution was to trace the shadow ray from light to the illuminated point and ignoring intersections, that are beyond said illuminated point.

Other problem was found during implementation shadows on GPU. Since almost all thread initialize some data during traversal, each of them is required to enter the traversal algorithm, even if their ray did not hit any object, thus not generated any shadow ray. It was not intuitive for me and had to realize after some time spent on debugging.

4. Results

The results were measured on my personal laptop with the following parameters:

- Computer Model: HP Victus 16-r0026nt
- Processor: Intel Core i7-13700H (13th Gen)
- Frequencies: 3,7 GHz
- Memory: 16 GB DDR5 RAM
- Graphics Card: NVIDIA GeForce RTX 4060 Laptop GPU

- Cuda Capability: 8.9
- Operating system: Windows 11 Home (application was run on WSL)

Generated pictures with shadows for some scenes can be seen in Figure 3.

The correctness was checked with an online tool designed to compare differences between pictures. The CPU-generated image and GPU-generated images differed by a maximum of 10 pixels, mostly occurring at the edges of the model. With a resolution of 800x800 pixels, the resulting difference is less than 0.01%.

Statistics that were measured are described below:

T_B	time to build the data structure, in seconds
T_CU	time to initialize and copy GPU data, in seconds
N_L	number of leaves
D_AVG	average leaf depth
D_MAX	maximal leaf depth
O_AVG	average number of objects in leaf
T_R	average time per query for many queries, in microseconds
T_F	in seconds
PERF	the performance in MRays/s (including CUDA initialization time)
N_IT	the average number of incidence operations per query with basic objects
N_TR	the average number of traversal steps through the data structures
N_Q	the count of queries used for a particular test case

The size of packets was set to 8x4, utilizing all 32 threads in WARP were used.

In terms of the time taken to build the data structure, the performance was particularly poor for scene AsianDragon, which took almost one minute to build. As described above, the split planes were placed on object centroids and the algorithm selected the one with the lowest cost. This can be improved by uniformly sampling only few objects to test for split plane placement. The parameters for this selection are described in detail in 'Realtime Ray Tracing on GPU with BVH-based Packet Traversal paper' [2].

Data structure statistics are showed in Table 1.

When comparing performace (for GPU includes time to initialize CUDA), displayed in figure 2, the difference is significant, aproximatelly the ray tracing on GPU is one order of magnitute faster than ray tracing on CPU. Except for the AsianDragon scene, again due to the lange scene size and the high cost of data copying operations.

If we focus on the average traversal time per query (as shown in Table 2, excluding the GPU data initialization time, the GPU traversal was one or two orders of magnitude faster than CPU traversal.

Scene	# of triangles	N_L	D_AVG	D_MAX	O_AVG	T_B
City	68 497	39 645	21.723	25	1.728	0.344
Sibenik	80 479	44 202	20.947	29	1.821	0.409
Armadillo	345 944	206 196	18.292	22	1.321	2.090
AsianDragon	7 219 045	190 147	23.462	29	1.016	59.504

Table 1: Number of leaves (N_L), average leaf depth (D_AVG), max leaf depth (D_MAX), average number of objects in leaf (O_AVG), measured values are time to build data structure (T_B [s]).

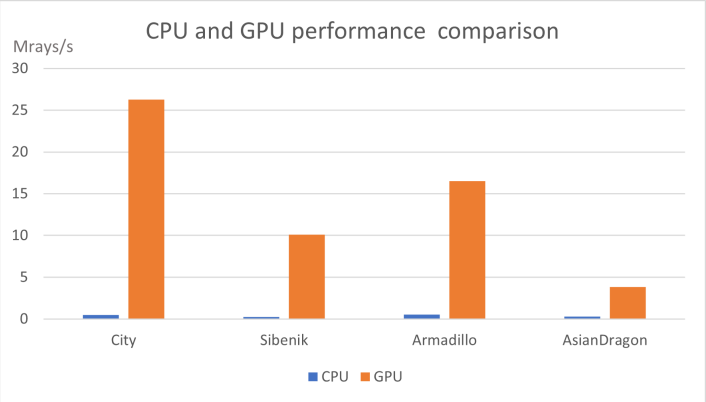

Figure 2: CPU and GPU performance in milion rays per second.

5. Conclusion

Ray tracing is significantly faster on GPU due to parallelization, but transferring data between the CPU and GPU consumes a large portion of the processing time. A major improvement to the current implementation would involve moving the BVH construction process to the GPU. This change would eliminate the need for data transfers between the CPU and GPU, potentially resulting in substantial speed-ups for the entire algorithm.

Additionally, another enhancement would be to modify the BVH construction algorithm to evaluate only a limited number of candidates for the split plane. This optimization would ensure that the algorithm performs efficiently, even for scenes containing millions of objects.

References

[1] Wald, I., Boulos, S. and Shirley, P. (2007). Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies.

[2] Gunther, J., Popov, S. and Seidel, H. (2007). Realtime Ray Tracing on GPU with BVH-based Packet Traversal.

[3] Havran V. (2024). Nanogolem 0.97. `https://cw.fel.cvut.cz/wiki/courses/b4m39dpg/parttime/start`

Scene	T_R	N_IT	N_TR	N_Q	T_F	T_CU	T_K	PERF
City CPU	2.026	10.82	16.48	1 017 966	2.062	-	-	0.493
City GPU	0.037	19.40	12.67	1 286 400	0.350	0.301	0.049	26.253
Sibenik CPU	4.254	15.47	13.10	1 279 989	5.446	-	-	0.235
Sibenik GPU	0.108	20.85	48.95	1 286 400	0.412	0.285	0.128	10.089
Armadillo CPU	1.864	3.74	21.24	789572	1.472	-	-	0.536
Armadillo GPU	0.064	8.82	22.79	1 286 400	0.385	0.307	0.078	16.492
AsianDragon CPU	3.321	3.91	21.29	787 837	2.616	-	-	0.301
AsianDragon GPU	0.243	23.47	102.59	1 286 400	1.260	0.923	0.998	3.812

Table 2: Comparison of CPU and GPU performance for primary and shadow rays, measured values are average time per query (T_R [per ray, in µs]), average number of incidence operations (N_IT), average number of traversal steps per query (N_TR), number of queries (N_Q), full time of traversal (T_F [s]), time to initialize CUDA data (T_CU), kernel execution time (T_K [s]) and PERF [Mray/s].

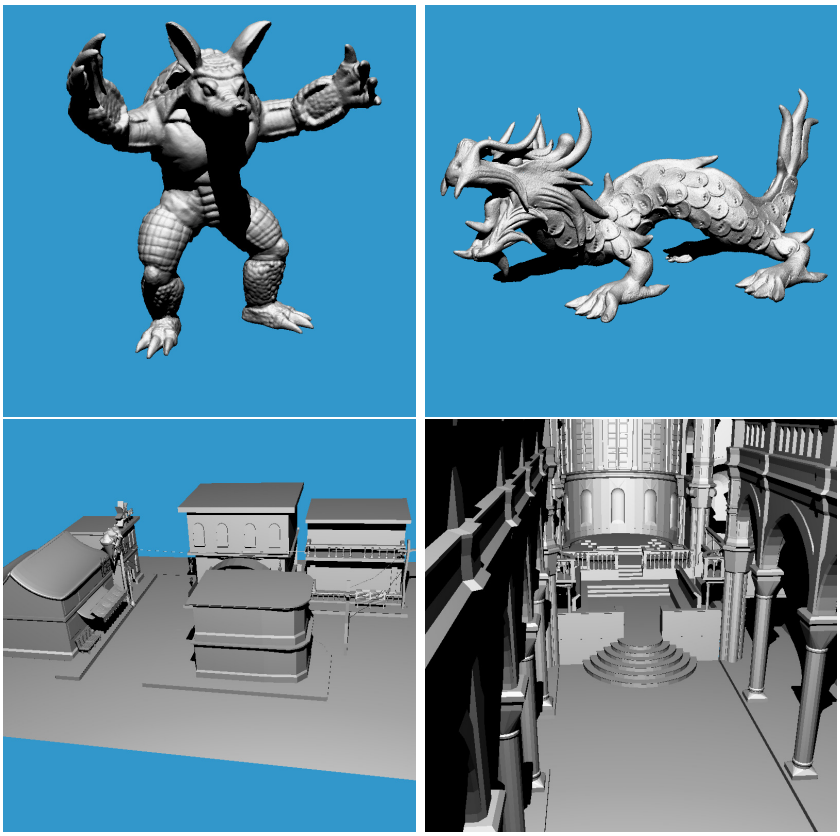

Figure 3: Example or result images rendered with shadows.
The scenes are: Armadillo, Asian Dragon, City, and Sibenik.