Traversal methods for GPU ray tracing

Marek Vinkler*
Supervised by: doc. Ing. Jiří Sochor, CSc.†

Faculty of Informatics Masaryk University
Brno / Czech Republic

Abstract

Ways of exploiting the raw performance of GPUs for computing ray tracing have been a hot research topic recently. Performance similar to the multi-core CPU ray tracing engines has been achieved. In this paper we present a new traversal method created by combining two of the existing methods. The proposed method is less sensitive to performance loss due to certain object scene distribution. It can also be faster than the methods from which it originated. Several possibilities how to create such a method exist and even better methods can be constructed with the addition of a single instruction to the next generation of GPUs. The resulting ray tracer achieves 50+ fps for primary rays on moderately complex scenes running on current mainstream GPUs.

Keywords: Ray tracing, GPU, CUDA

1 Introduction

The graphics hardware has witnessed enormous growth in both performance and programming flexibility recently [5, 7]. This development enabled creation of general purpose parallel computing architectures such as NVIDIA CUDA [8]. Many applications have been ported to this platform to take advantage of its raw performance. The GPU ray tracing engines became a serious alternative to CPU-based ones. Several mappings of the ray tracing algorithm to the graphics hardware were proposed [4, 1]. In this paper we compare some of the existing methods focusing on their performance characteristics with regard to the object scene distribution. It shows that where one method works well the other often performs poorly. This leads to the idea to create a hybrid method that addresses the drawbacks of both of the existent methods.

There are several promising mappings of the ray tracing algorithm to the graphics hardware. The packet traversal method is described in [4]. In this mapping all rays in a packet follow the same path during the traversal. To leverage the power of the GPU architecture rays are mapped to threads and packet size is chosen as warp (group of parallel threads) size. This way the threads within the warp can cooperate in loading the node and triangle data. Also there is less branching as all the rays, by definition, follow the same code path. This leads to almost perfect utilization of the hardware but effective parallelism decreases each time the threads within the warp want to take different paths.

Another approach named "if-if" traversal can be found in [1]. In this mapping once again one thread computes one ray but these rays follow their individual paths. Thus there is no cooperation among the threads. This approach sacrifices coalesced loads and coherent branching for higher parallelism. It achieves higher performance than packet traversal method when rays take different paths frequently e.g. near the leaf nodes. On the other hand its performance is inferior in places where rays traverse the acceleration data structure coherently e.g. near the tree root.

It is not a coincidence that both of the articles mentioned above use AABB BVH (Axis Aligned Bounding Boxes Bounding Volume Hierarchy [3, 9]) as an acceleration data structure. It has been chosen for several reasons. First, the acceleration data structure should be small as there is considerably less memory available on the GPU [6]. It should also allow fast reconstruction thus supporting dynamic scenes. Both of these criteria favour grids and BVHs. However grids often perform poorly on scenes with non-uniform object scene distribution making BVH the acceleration data structure of choice. Detailed description of both of these traversal methods as well as of the hybrid method constructed from them is given in section 3.

2 Test setup

For the reasons listed above we have chosen AABB BVH as an acceleration data structure for our ray tracing engine. Hierarchy construction is handled by a third-party code from Arauna ray tracer [2]. The maximum number of primitives in a single leaf node is set to 6. The engine creates the hierarchy just once at the beginning allowing

*xvinkl@mail.muni.cz
†sochor@fi.muni.cz

only static geometry to be rendered. However with the change of the underlined BVH construction algorithm dynamic scenes could be rendered as well.

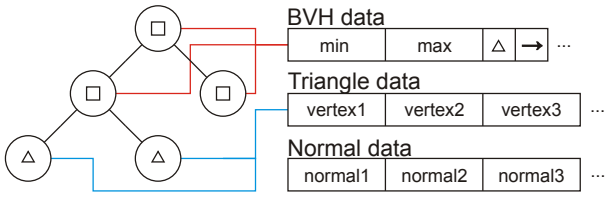


Figure 1: Memory layout

Each node is a 32-byte wide block (12 bytes min coordinates, 12 bytes max coordinates, 4 bytes number of contained primitives and 4 bytes pointer to the children). The latter two are integer values promoted to single precision floating point values for the ease of storing. The left and right BVH children are stored in continuous 64 bytes and are always loaded together. All data are stored in 1D float arrays. There are separate arrays for BVH data, triangle data and vertices normals as shown in 1. The BVH data are always fetched through texture while triangle data use texture only for the packet traversal method. Normal data are always loaded directly from global memory. This way the engine achieves maximum performance.

Four popular test scenes were chosen for performance measurement. Table 1 lists important information about the models. No hand tuning was applied to the assembly code produced by the complier.

Scene	Triangles	Split triangles	Nodes
Conference	283k	424k	190k
Fairy	174k	261k	117k
Sibenik	80k	121k	56k
Sponza	76k	114k	52k

Table 1: Triangle counts, triangle counts after split and number of nodes for the four test scenes. Rendered images of these scenes are in Figure 2.

3 Methods

To gain maximum ray tracing performance from the GPU one needs to address several key aspects. First, the register count per thread must be kept small enough to allow sufficient parallelism. Some thread data (e.g. stack, next node pointer, etc.) must be stored in shared or local memory instead of registers. Second, the code for data loading, stack handling and traversal decisions should be kept as simple as possible not to waste instructions. The kernels are completely compute bound (the memory access latency is completely hidden with computation) and therefore any extra instructions lead to noticeable performance loss.

We have implemented and compared three ray tracing traversal methods: packet traversal, if-if traversal and hybrid traversal. All of these methods share a common pattern and differ only in how the traversal is done. They all use optimization techniques described in [1]. Namely persistent warps and assigning ray indices based on morton code are employed. Description of each traversal method as well as its advantages and disadvantages follows.

3.1 Packet traversal

The basic characteristic of packet traversal is that a group of rays follows exactly the same path in the BVH tree. This is achieved by sharing the traversal stack among the rays in the packet. Each time the rays want to decide which node to traverse next they have to vote. There are two options how to do that using current hardware. The first one is to do reduction in shared memory. The second is to make use of warp vote functions. These functions evaluate a predicate for each thread and then return the same boolean value (computed from those evaluations) to every thread in the warp. Currently vote functions *__any* and *__all* are supported. While the former method gives a precise answer to where the majority of threads wants to go it takes a lot of instructions to compute. The latter method is only approximate but uses just a few instructions leading to higher performance. Another advantage of coherent traversal is that it leads to perfect memory loads. Every node intersected by the packet is loaded from the global memory only once (not multiple times for every ray in the packet) and all loads are coalesced. Also only one (two for triangle data) memory instruction per thread is issued to load all of the data for all of the threads. However when the rays within the packet want to take different paths in the tree these paths must be serialized using the shared stack. Parallelism is therefore lost with each such branching and rays visit (potentially many) nodes which they do not intersect.

The packet traversal method is the fastest possible method when the rays want to take nearly the same path, for example if the first intersection for all rays within the packet lies in the same leaf. This is however seldom even for highly coherent primary rays. To exploit the architecture best the packet size is chosen as the warp size. Our implementation of packet traversal kernel uses 25 registers per thread and about 3kB of shared memory per block. This leads to 50% occupancy on devices of compute capability 1.2 or higher. This is more than sufficient as the kernels have high arithmetic intensity (ratio between arithmetic and memory operations).

3.2 If-if traversal

In contrast to packet traversal in the if-if traversal method each ray follows its own path. This is done by keeping separate stack for each ray. The stack is currently allocated in thread's local memory as shared memory is too small. Even though the local memory is as slow as the global memory, stack loads and stores latencies seem to be hidden by other threads computation. What actually hurts the performance is thread serialization during these memory operations. Different threads often happen to access different stack indices. In such a case separate memory instructions must be issued for each stack index following the coalescing rules. Another performance loss is due to branching within a warp - some threads want to intersect nodes while others want to intersect triangles. In this case both execution paths are serialized as described in [8]. Clearly this increases the number of issued operations and reduces performance. Another drawback of this method is the way how loading of both node and triangle data is handled. Each thread must issue several memory load instructions to gain the data it needs.

The final kernel consumes 24 registers per thread and no shared memory is needed leading to 63% occupancy. Interestingly there is little performance loss if one extra register is consumed and occupancy drops to 50 %. This method's performance is superior to the one of packet traversal if threads within the warp take different paths frequently. A perfect example is the tracing of secondary rays.

3.3 Hybrid traversal

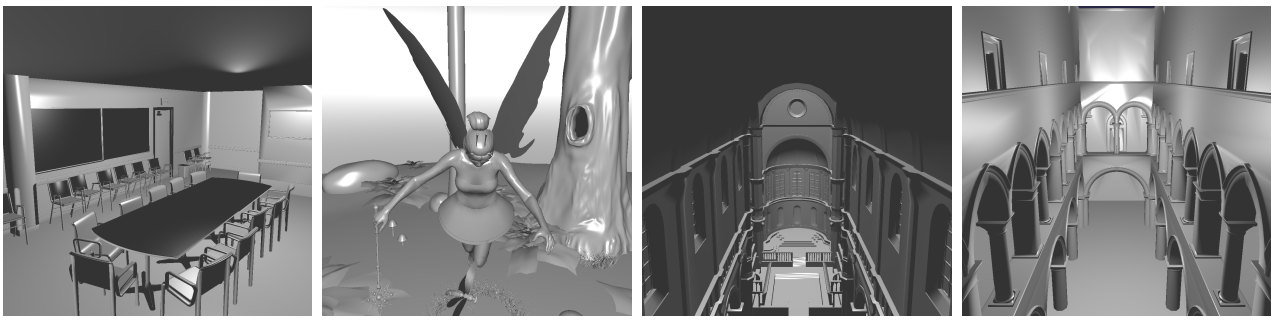
As mentioned above this method is a combination of the packet and if-if traversal ones. The idea is that packet traversal performs best near the tree root where rays are coherent whereas if-if traversal is better suited for traversing nodes near the leaves. It is, however, unclear when and how to switch between the two of the methods. We can divide the methods into groups based on how the switching is done.

The straightforward idea is to switch the packet and if-if traversal each time a certain condition is met. We start tracing the rays with the packet traversal method and when the condition is triggered we switch to if-if traversal. Then after all rays have finished traversing the current path they load another node from the shared stack and start tracing it again with packet traversal. This continues until the shared stack is empty. The traversal thus follows the classical depth-first search scheme but uses different methods to trace different parts of the tree. This method turns out to be slow as the next packet traversal phase cannot start until all of the rays have finished their if-if phase. Thus all the rays are idle until the longest running one ends and this happens multiple times. If, however, there were an effective algorithm for loading work per ray as discussed in [1] the majority of rays would not be idling and the method could be interesting.

Another option is to switch between the packet and if-if traversal only once. This method seems to be more promising and so we have developed several conditions to rule the switching. The easiest one and currently achieving best performance is the one we call "stack-max" traversal method. In this method packet traversal ends when the shared stack size is bigger than a predefined threshold. In this moment if-if traversal starts from the last visited node and later on visits each of the nodes on the shared stack. We will discuss the performance characteristics later in section 4. The register usage for this kernel is 26 registers per thread and the same amount of shared memory as for the packet traversal is used. The occupancy is thus at 50 %.

There is also the possibility of counting how many times rays wanted to take different paths. If this number exceeds some threshold we make the switch. This method is not very sophisticated yet it takes quite a lot of extra instructions leading to poor performance. This is why we will not mention it in the result section.

The last developed method is the most sophisticated one. It stops traversing a path with packet traversal if too few rays want to take that path. If this happens it pushes the address of the last node on that path to the local stack of threads which want to take that path. Then it takes another node from the shared stack and the process continues. This way it traverses all coherent nodes (nodes which a significant amount of rays want to visit) before switching to if-if traversal. Such a traversal no longer follows the classical depth-first search traversal scheme. It resembles the depth-limited search but the limit is different for each branch. The if-if traversal then traverses the sub trees defined by nodes saved on the local stack. The main advantage of this method is that it is not dependent on some predefined threshold and should classify coherent/incoherent parts of the tree well. The drawback of this method is that computing how many rays want to go to the left and right children of the current node is expensive. With current generation of hardware the reduction in shared memory must be employed to obtain such a number. However, if a new instruction - returning the number of threads in a warp satisfying a predicate - is introduced in next generation of GPUs the method could be the fastest hybrid method. In the result section this method is denoted as "cut" traversal method. This method has higher register and shared memory demands. It consumes 28 registers and about 3.4kB of shared memory per block. Nonetheless, that is still low enough to keep the 50% occupancy.



(a) Conference (b) Fairy (c) Sibenik (d) Sponza

Figure 2: Test scenes

4 Results

Images in figure 2 were obtained at output resolution of 1024x1024 using NVIDIA GTX 280. Each FPS count in table 2 is an average over five different viewpoints. These viewpoints were chosen randomly so that most of the scene triangles were visible from them. The timings include the whole kernel execution (ray generation, traversal, shading, etc.).

As it can be seen in the upper part of table 2 the if-if traversal is the second slowest of all methods for the primary rays. This is interesting since it is reported to be faster than packet traversal [1]. The reason for this discrepancy is unknown to us. Possible candidates are a worse BVH tree construction algorithm or simply a poor implementation of the mentioned method. It is important to realize that poor performance of the if-if traversal reflects also in performance numbers for the hybrid methods.

The stack-max traversal is slightly faster than the packet traversal but the speedup is not interesting by itself. The noteworthy thing is that it has much more balanced performance with different types of rays. The method depends on a constant denoting the maximal number of items in the shared stack. Its performance slightly varies with the change of the constant. When rendering the conference scene the constant for the best performance was mostly so high that the if-if part of the traversal was skipped. The important thing to notice here is that even when the stack-max traversal collapses into the packet traversal it is not much slower than the specialized kernel. Only one extra if-statement is evaluated in the main traversal loop. For the other two scenes a lower choice of the constant leads to higher performance. The constant value 11 was chosen for separate measurements because it achieved reasonable performance in all of the test scenes. This number is, however, scene specific and cannot be considered a general guideline. From some special viewpoints the stack-max achieved noteworthy speedups. This encourages the hypothesis that the tree can be divided into parts of coherent and incoherent traversal. Though the stack-max is not the right condition to guide this division.

The cut method is the slowest method for the primary rays. This is mainly because its performance is limited by the cost of its traversal decision code. It is cheaper to intersect one or two nodes than to decide which node to take next. This makes it a poor choice for current generation of hardware. As discussed above there is potential to change this state making it interesting to benchmark.

The results for the secondary rays (reflected and refracted primary rays) are given in the bottom part of table 2. The important thing about these numbers is that the measurement corresponds to tracing twice as many rays than for the primary rays. The if-if traversal method is the fastest for the incoherent secondary rays as predicted. Interestingly the other methods are not lacking far behind.

The stack-max traversal method performs quite well on the secondary rays. However, as discussed above it needs the right constant for the switch criterion to be fast. Here constant value 7 turned out to give reasonable performance. As one might expect this number is lower than for the primary rays because the rays tend to take different paths often. Thus the number of rays within the warp that want to take the same path drops rapidly with increasing depth. Unfortunately no choice of the constant for the stack-max method forces a collapse into the if-if traversal method. The packet part of the traversal always takes place. This explains why the method cannot achieve performance as good as the one of the if-if traversal method for the Sibenik and Sponza scenes. As discussed above the stack-max traversal method is more versatile with regard to object scene distribution. This can be observed from the speedups against packet traversal for the secondary rays.

The packet traversal method is a poor choice for the incoherent secondary rays. The rays traverse a great amount of nodes they do not intersect and have to finish paths that the majority of the rays want to take before taking their own path. This is why it is almost as slow as the cut method.

The results for the cut method show some promise. It is still the slowest of all the compared methods but the dif-

ference is not as abysmal as for the primary rays. This is a sign that the reduction of the amount of needlessly traversed nodes (due to better traversal criteria) outweigh the criteria cost.

5 Conclusions

The hybrid traversal methods presented in this paper are comparative to the fastest known traversal methods. They are able to benefit from coherent traversal of rays while they sustain good performance in incoherent setting as well. This is achieved by utilizing most of the GPU resources. Advantageously optimizations presented in other papers can be used with this traversal method as well.

The performance of the hybrid methods is strongly influenced by the employed switching criterion. The proposed criteria use heuristics to divide the acceleration data structure into coherent and incoherent parts. Better heuristics may be found in the future. They should be fast, consume as little resources as possible and precise in classification of coherent/incoherent nodes. With the progress in graphics hardware some known criteria may become more efficient.

6 Acknowledgments

Jacco Bikker (http://igad.nhtv.nl/∼bikker/) for the Arauna engine. Marko Dabrovic (www.rna.hr) for the Sibenik cathedral model. University of Utah for the Fairy scene. This work was supported by Ministry of Education of The Czech Republic, Contract No. LC06008 and by The Grant Agency of The Czech Republic, Contract No. P202/10/1435.

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.

[2] Jacco Bikker. Real-time ray tracing through the eyes of a game developer. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[4] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Mark Harris. Many-core gpu computing with nvidia cuda. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 1–1, New York, NY, USA, 2008. ACM.

[6] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Manocha Dinesh. Fast bvh construction on gpus. In *Computer Graphics Forum Volume 28, Issue 2*, pages 375–384. The Eurographics Association and Blackwell Publishing Ltd., 2009.

[7] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[8] NVIDIA. *NVIDIA CUDA Programming Guide Version 2.3.*, 2009.

[9] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object partitioning considered harmful: space subdivision for bvhs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 15–22, New York, NY, USA, 2009. ACM.

Primary rays								
Method	Conference	Speed-up	Fairy	Speed-up	Sibenik	Speed-up	Sponza	Speed-up
packet	68.12	0%	60.44	0%	49.48	0%	47.46	0%
if-if	54.80	-19.55%	54.54	-9.76%	49.48	0%	47.80	+0.72%
stack-max(11)	67.44	-1.00%	60.64	+0.33%	49.93	+0.91%	47.40	-0.13%
stack-max(best)	67.92	-0.29%	61.20	+1.26%	49.94	+0.93%	47.56	+0.21%
cut	61.98	-9.01%	53.04	-12.24%	44.96	-9.14%	42.28	-10.91%
Secondary rays								
packet	21.22	0%	18.84	0%	14.94	0%	15.26	0%
if-if	21.80	+2.73%	19.50	+3.50%	16.80	+12.45%	19.18	+25.69%
stack-max(7)	21.72	+2.36%	19.98	+6.05%	15.40	+3.08%	16.26	+6.55%
stack-max(best)	21.78	+2.64%	20.04	+6.37%	15.80	+5.76%	17.32	+13.50%
cut	21.66	+2.07%	18.30	-2.87%	13.82	-7.50%	15.00	-1.70%

Table 2: FPS counts from four static scenes averaged over five viewpoints each. Text in parentheses denotes the constant used for that method. The speed-up against packet traversal method is given for each test scene.