Coherent GPU Path Tracing: Improving Data Locality

Matheus S. Faria* Edson A. Costa Jr† Renato C. Sampaio‡

Universidade de Brasília, Faculdade do Gama, Brasil

ABSTRACT

Path Tracing algorithms can be used to render photo-realistic images in a considerable amount of hours. The efforts to make this process faster have received more attention with the popularization of the GPU. One of the main issues is the poor cache use when the rendered scenes are complex. This work presents a novel approach that aims to improve its use. Our method performs an iterative tiled BVH traversal, where the BVH is split into sub-BVHs and reorganized in a breadth first l ayout. At e ach s tep, t he r ays t raverse a sub-BVH, then they are sorted based on the last node intersected. Which increases the data coherence and the GPU usage, leveraging the performance.

Keywords: ray tracing. ray coherence. GPU. BVH. ray sorting.

1 INTRODUCTION

The results of the rendering algorithms research over the years, since Ray Tracing creation [22], have shown that it is possible to achieve images indistinguishable from reality [17]. Ray Tracing algorithms use rays to find and intersect geometry in a scene, and each geometry has specific properties that will determine to where the ray should go from that intersection [22]. A large amount of these rays is required to get a realistic image, and this takes a long time. To reduce the rendering time, in the last two decades, some accelerations have been proposed, focusing on parallel processors, e.g. the Graphical Processing Units (GPU).

The GPU is a graphical card optimized for rendering using rasterization [5], which relies on several buffers to render the objects and defines what is going to be displayed based on their depth [21]. Although rasterization is a rendering technique that runs in real time, it does not produce directly any photo-realistic effects, like soft shadows, reflections, refractions, caustics, color bleeding and more [6]. To render these effects it is necessary to compute the Global Illumination (GI), which is the illumination that depends on everything that is on the scene [8]. To simulate the GI on rasterization, it is necessary to use shaders and external programs [14, 23], differently from Ray Tracing, that easily renders these effects without any shader [6].

In Ray Tracing, a common spatial data structure used is the Bounding Volume Hierarchy (BVH). It divides the 3D space based on the bounds of a set of objects. This structure reduces from $O(n)$ to $O(log\, n)$ the complexity of querying the scene object set to find an intersection.

Recent studies achieved interactive frame rates – between 1 to 20 FPS – by using parallel devices to accelerate the Ray Tracing and the BVH traversal [19, 6, 12]. However, most of their scenes do not include more complex effects, like reflections and refractions, not exploring most of the GI effects. The reason for using simpler scenes is that diffuse (or Lambertian) objects produce more coherent rays. These rays follow the same path under the BVH fetching the same memory regions, doing a heavy use of caches, which accelerates the render. However, when objects with reflective and refractive properties are added to the scene, the rays will hardly match their destination generating the incoherent rays [16].

Incoherent rays access different parts of the memory, which is known as spatial incoherence. This causes performance loss because the GPU is forced to query multiple memory lines, using only a small part of each line. On an ideal spatial coherent access, the data comes in sequence, using the same line or, if it is bigger than a line, a sequence of lines.

If it could be possible to trace incoherent rays avoiding the data incoherence, the GPU would greatly accelerate the algorithm execution. However, to achieve data coherence it is necessary to spatially organize the rays, and this can be done with a better spatial data structures and sorting routines.

In this work, a novel approach is presented. It handles incoherent ray data on the GPU by using a packet-less BVH traversal. Where the BVH is arranged in a breadth-first layout to reduce memory incoherent accesses. It uses a tiled pattern for loading sub-BVHs and traverses them to find the last intersected node in this sub-BVH. To traverse other sub-BVHs given the last intersection, it uses a radix sort routine to reorder the rays, improving data coherence at each iteration.

2 RELATED WORK

Efficient ray traversal is one of the main concerns on Ray Tracers [9]. Traversal of coherent and incoherent rays has been explored by several researchers over the last two decades. Coherent rays are those that have some spatial consistency, following the same path on the acceleration structure traversal [13]. Different from incoherent rays, their path diverge inside the acceleration structure.

The first attempt to improve the ray traversal efficiency was to use ray packets, introduced by Wald et al. (2001). The main idea is to encapsulate several rays into a pack and traverse it on the acceleration structure. In their work, a 4-ray pack was used to traverse a kd-Tree, in a depth-first manner. And to perform the ray-box intersection test, a 4-wide SIMD unit was used. Their results show that the processing time and the memory bandwidth were reduced by a factor of 4 because the kd-Tree nodes are only fetched from memory once for each pack [20].

As hardware got better and SIMD units wider, the idea of packets was improved by Reshetov et al. (2005). They group rays in a frustum, called ray beam. Those frustums hold 16 rays each and traverse the kd-Tree performing a reverse frustum culling algorithm, where the node AABB is tested against the frustum. These beams are subdivided into sub-frustums if the rays inside it diverge. And the process is repeated until they get into a kd-Tree leaf, where each ray is tested against the geometry. They show that for primary and shadow rays their method has doubled the performance [13].

Boulos et al. (2007) were the first to work with the performance of secondary rays using ray packets. The main problem identified by them was the performance loss as the rays of the same packet lose their coherence. Their solution was to pack the rays by their type, and with this modification they achieve a 2x speedup over single-ray tracing, using 8 x 8 packet, achieving an interactive frame rate on secondary rays [6].

*e-mail: matheusfaria@unb.br
†e-mail: edsonalves@unb.br
‡e-mail: renatocoral@unb.br

Overbeck et al. (2008) create a new method, called partition traversal, which improves the ray coherence. In their method, they use the same technique presented on [19], but when they select the rays they put the active rays in the first positions of the array of rays, and keep track of last active ray index on the traversal stack. With that, it traverses only the active rays inside each node. This method shows a good scaling when the packet size varies up to 32 x 32, and also outperforms the methods of Wald et al. (2001 and 2007) achieving interactive frame rates for scenes with refractions and reflections [12]. Gribble et al. (2008) present a similar idea for ray selection, the stream filtering, where they show that this approach is independent of the acceleration structure [11].

Wald et al. (2008) present a new technique for ray traversal that does not rely on packets. They use wide SIMD units to perform single-ray front-to-back traversal, but they also use a BVH with a high branching factor to match the SIMD size. They measured the same performance compared to the use of packets for primary rays. However, for secondary rays they had a performance gain [18].

Garanzha et al. (2010) use packets with ray reordering on the GPU. They present a Compress-Sort-Decompress (CSD) scheme to increase the coherence of ray packets. On the compress stage, a hash is extracted from every ray based on their origin and direction, then, rays with the same hash are grouped together. Afterwards a radix sort is performed on this compressed array and all data is decompressed. During the decompression, the rays will be already reordered in right position. Their results show that the CSD takes a little amount of time, not negligible, but the results are only shown for primary and shadow rays, which suits better in packets [10].

Benthin et al. (2012) combine packet traversal with single-ray traversal on a hybrid approach. It performs the packet traversal for primary rays, measuring in each traversal step a number of active rays. When this measure is below a defined threshold, the packet traversal switches to single-ray traversal using SIMD. Since they target a very specific platform, Intel Multiple Integrated Coprocessor (MIC), their architecture is very specific. But they show an almost interactive frame rate for very complex scenes in a full path tracer, achieving almost 2x more performance than the packet tracing alone [4].

Barringer et al. (2014) perform the single-ray traversal using SIMD units, to avoid packets. Their methods maintain three stacks of rays, one for each node of the BVH, left and right, and one for the rays that hit both. Besides, there is a global stack that keeps a number of rays added to each stack. In this way, it can traverse in a depth-first manner keeping the ray coherence. They show a 30% improvement over the hybrid approach [3].

Aila et al. (2009 and 2012) present their study about ray tracing on several NVIDIA's GPU architectures: Tesla, Fermi, and Kepler. Given several methods on their bibliography, they showed for each architecture some improvement to achieve more performance. One of the improvements is the use of textures to load triangles and nodes data, which improves the access speed because textures allow the incoherent access. Moreover, their results show that the power of these architectures is doubling from the oldest to the newest, which allows more rays to be processed [1, 2].

3 Proposed Model

The proposed model improves the ray coherence on the GPU by guaranteeing that rays residing in the same warp follow the same traversal path inside the acceleration structure [10]. If the rays are coherent, they access the same memory locations during traversal, reducing memory bandwidth. However, it is possible for rays in the same warp to traverse a distinct path keeping the low memory bandwidth.

Exploiting this fact the proposed model uses a tiled BVH traversal algorithm to improve the traversal of incoherent rays. The basic idea is to load a smaller sub-BVH into the shared memory and traverse it. When the rays reach the sub-BVH leaves, which we are going to reference as inner-leaves, they record which inner-leaf they've reached. When all rays finish the traversal, they are reordered based on inner-leaf index. The reordered rays can be used to restart the process. When all rays reach a leaf from the BVH, all the geometry stored on it is tested against the rays.

The algorithm is divided into four parts, where the first one describes how to properly organize the BVH, called Layer BVH. The second shows how to load and traverse a sub-BVH tree, the third presents the ray reorder strategy, and the fourth shows the final ray-triangle intersection.

The complete model can be overviewed at Figure 1. The model start its process with a BVH Tree, then it organizes the tree in the Layer BVH layout. In an iterative step, the model traverses the sub trees extracted from the Layer BVH, then it sorts the resulting intersections. If there is any intersection to be process yet, the model restarts at the traversal part. Otherwise, the leaf intersections are performed, and the rendered image is presented.

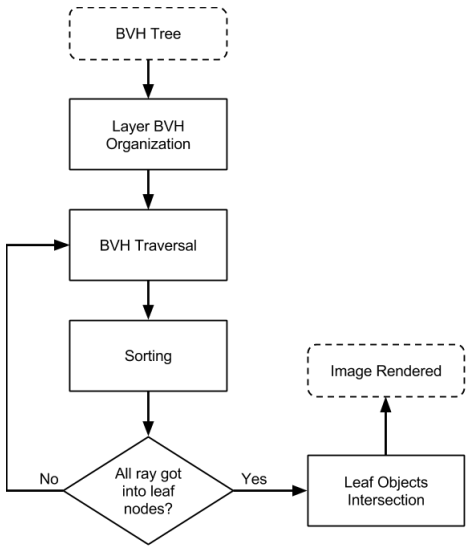

Figura 1: Flat Reorganized BVH

3.1 Layer BVH Organization

The BVH can be built using any algorithm, but the final array containing the tree will be reorganized into smaller trees, that will be placed sequentially in an array. Those smaller trees will be formed by a set of n tree layers.

Given the n tree layers, we can calculate the maximum subtree height H. Then, the algorithm performs a breadth-first traversal, appending the nodes in a subtree until its height does not exceed H. When it does, the subtree is written at the Layer BVH array. And the process is repeated until all nodes are attached to their subtree.

The exception of this method are the BVH leaf nodes. They should be processed in separate because they do not belong to any subtree, since they will be traversed in a different part. And just like the subtrees, they need to be placed sequentially in the Layer BVH array. Since the tree levels and leaves are contiguously located in memory, this new tree array is a cache friendly arrangement for a breadth-first traversal in the subtrees.

An example of Layer BVH can be seen in Figure 2, where the maximum subtree height is 2. It is important to notice that this method does not require a well-balanced tree since most of the BVHs aren't balanced. For that reason, it is required to keep track of the number of nodes in each layer.

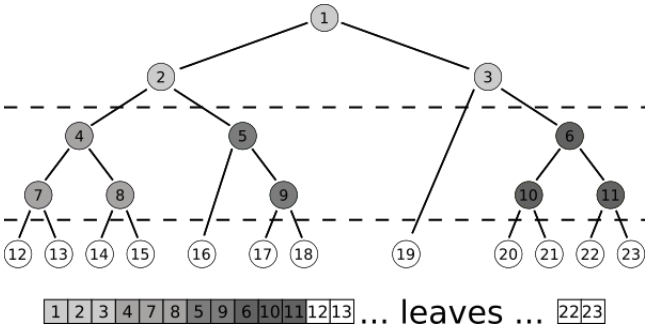

Figura 2: Flat Reorganized BVH

3.2 BVH Traversal

The tree traversal is divided into subtree traversals, where each subtree is traversed by several rays using one or more blocks in the GPU. The goal of this part is to determine which inner-leaves or leaves the rays will intersect at the end of subtree traversal.

The main difference between this algorithm and many other GPU tree traversal algorithms, like Garanzha et. al. (2010), is the use of the tiled pattern, where the data used in the kernel execution is gathered and stored in the shared memory. Following the pattern, rays and tree nodes are placed into the shared memory. Since both are spatially coherent located into the global memory, it is possible to load them using minimal overhead.

When the whole data is loaded, the algorithm can perform a non-recursive breadth-first traversal routine. The output data, the last inner-leaf intersections, is simultaneously stored in the global memory. However, this data will be spatially incoherent, which requires an extra sorting part. Otherwise, it won't be suitable for the next algorithm iteration.

3.3 Sorting

The traversal kernel needs sorted data to launch. Even though all coherent rays traverse the same subtree in the first iteration, this is not guaranteed in the subsequent iterations. When the algorithm gets to this part the rays array is filled with the rays and their inner-leaf intersection index. The sorting method needs to group the rays with the same index. To perform this task, the radix sort [15] is used, since it is the sorting algorithm that outperforms on the GPU compared to other sorting algorithms [7]. In this sorting routine, the index is used as sorting key, and the ray as the value. Leaf nodes must have the lowest priority during the sorting because they will not be considered on the next traversal iteration.

3.4 Leaf Objects Intersection

The last part is to traverse the leaf nodes found, which contain the scene geometry. Since the data will be sorted at this point, it is possible to use the data coherence to improve the intersection. This part of the algorithm uses a thread per ray to check its intersection against the geometry inside the leaf node and outputs the closest intersection. With this information, the Ray Tracing can proceed and compute the color related to that intersect or generate more rays.

4 RESULTS

The algorithm two main parts are the Layer BVH Organization and the tiled pattern applied on the BVH traversal. The other two parts, the ray reordering and the ray-object intersection, do not have a complex implementation.

The Layer BVH Organization, implemented in this work, uses two queues, one to keep track of the subtrees root nodes and another to perform the breadth-first traversal on the subtree. The traversal stops when the subtree size is greater than the maximum subtree size, and at this point all the nodes that are in the second queue are enqueued on the first queue. This transference from one queue to another happens because all nodes that are out of the subtree become a root node to another tree. Another implementation decision was to keep a vector of subtree size, so it is possible to know how many nodes each subtree has.

The tiled traversal is done in a CUDA kernel and uses the first thread of the block to determine which subtree that block is going to load. Since the data is ordered, most of the threads in the block point to the same tree. The threads that point to another tree are scheduled for a later iteration. When the subtree is determined, the kernel uses all threads in the block to load the subtree into the shared memory, even those that do not belong to that tree. This way the load operation can use the full potential of the block. Most of these load operations from global to shared memory require a barrier synchronization after them, which avoids threads using invalid data.

An example of the traversal kernel execution is shown at Figure 3, that shows the active threads during the execution. This example uses a 3 by 3 block structure, required by the tiled pattern, with subtrees of 7 nodes at maximum, with height 3. When the traversal kernel starts, all threads are active, the first thread on block stores 0 as the subtree root index. When the subtree is going to be loaded from the global memory to the shared memory, the 9 threads are used, the whole block. After this loading phase, the traversal starts, but since only the first 4 threads have the same subtree root index, all the other threads are inactivated and scheduled for later. If the root index is different, it means that the thread points to another subtree.

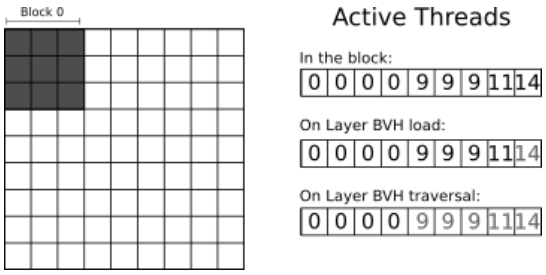

Figura 3: Threads state during the traversal

The traversal kernel is executed in a loop because some of the threads are scheduled for later. To know how many times this loop needs to run, an atomic counter of active threads is maintained through kernel calls. When a thread hits a BVH leaf this counter is decreased, and when it hits zero the algorithm can proceed to the ray-object intersection.

The two major downsides of the current implementation are the loop of active threads and the first thread of the block that selects the subtree. The loop is a downside because there is no guarantee of how fast the active threads are going to decrease. The first thread selecting the subtree is a downside too, because in some loops the selected subtree does not represent the most present subtree in that block. It may happen that only the first threads, or a few threads, need that subtree delaying the other threads to latter. This associated with the loop at the end of active threads causes kernel calls to process a very tiny amount of rays at each call. And it results in a longer time executing, because of the kernel call overhead and the sorting step that happens on each loop.

The traversal algorithm does not handle the intersection case where the ray hits both children yet. It only accepts one hit, which makes for most of the rays a bad decision because it is ignoring part of the tree. To handle these children overlap, it is necessary to allo-

cate more memory in the GPU. And in the worst case, where a ray intersects all n subtree leaves, it is necessary to allocate n times the space to store collision data. This can be very memory inefficient, and would increase by a factor of n the workload size. However, a workaround for this inefficiency is a better BVH partition algorithm.

The algorithm still is under development, with minor adaptations, and test. It's performance and efficiency will be evaluated when it is done in an environment that has an 8 GB memory, an Intel i5-4210H with 3.2GHz, and a NVIDIA GeForce GTX 965M with 2GB GDDR5 using CUDA 7.5[1]. The code and results can be found at the Github[2].

5 Conclusion

Incoherent ray traversal was already widely explored on the CPU with several types of research and algorithms. However, most of them do not work directly on the GPU, and this needs to be explored. The GPU has several particular issues that need to be addressed while you are developing an algorithm. And this puts several drawbacks on the discovery and adaptation of algorithms.

Our method is a novel way to organize the BVH trees and traverse them. It attempts to solve performance issues on the incoherent traversal. By organizing the BVH in several sub-trees, to use the benefits of the GPU's shared memory, which is faster than the commonly used global memory.

The implementation shows that the method can be very memory consuming, with several memory allocations for auxiliary data. And the partition method needs to be chosen carefully, because the algorithm can't solve correctly the case where there are overlapping nodes. The algorithm is currently under tests and adaptations. Our goal is to get into a model where it consumes near the same amount of memory that the naive BVH traversal and has a better performance.

5.1 Future Work

The algorithm presented on this work can be adapted for any binary tree. It presents a novel way to organize these trees and traverse it. In a context where nodes can't overlap each other, it can be very efficient.

As presented by Aila et. al. (2009), it is possible to use persistent threads to accelerate the traversal. These threads would persist in memory through several kernel calls. And with that in mind, a possible extension of this work is to load all subtrees into persistent threads, and use a pipeline pattern to traverse the tree. This pipeline would not need to load the subtrees again, and all traversal stages would be present on the GPU.

Referências

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149. ACM, 2009.

[2] T. Aila, S. Laine, and T. Karras. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[3] R. Barringer and T. Akenine-Möller. Dynamic ray stream traversal. *ACM Transactions on Graphics (TOG)*, 33(4):151, 2014.

[4] C. Benthin, I. Wald, S. Woop, M. Ernst, and W. R. Mark. Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(9):1438–1448, 2012.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

[6] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007*, pages 177–184. ACM, 2007.

[7] D. Bozidar and T. Dobravec. Comparison of parallel sorting algorithms. *arXiv preprint arXiv:1511.03404*, 2015.

[8] P. Dutre, P. Bekaert, and K. Bala. *Advanced global illumination*. CRC Press, 2006.

[9] C. Eisenacher, G. Nichols, A. Selle, and B. Burley. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, volume 32, pages 125–132. Wiley Online Library, 2013.

[10] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, volume 29, pages 289–298. Wiley Online Library, 2010.

[11] C. P. Gribble and K. Ramani. Coherent ray tracing via stream filtering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 59–66. IEEE, 2008.

[12] R. Overbeck, R. Ramamoorthi, and W. R. Mark. Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 41–48. IEEE, 2008.

[13] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (TOG)*, 24(3):1176–1185, 2005.

[14] T. Ritschel, T. Grosch, and H.-P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82. ACM, 2009.

[15] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[16] J. A. Tsakok. Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 151–158. ACM, 2009.

[17] E. Veach and L. J. Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

[18] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 49–57. IEEE, 2008.

[19] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1):6, 2007.

[20] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001*, pages 277–288. Springer, 2001.

[21] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 214–222. ACM, 1977.

[22] T. Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH Computer Graphics*, volume 13, page 14. ACM, 1979.

[23] C. Wyman and S. Davis. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 153–160. ACM, 2006.

[1] https://developer.nvidia.com/cuda-toolkit

[2] http://www.github.com/MatheusFaria/cuda_path_tracer