

Leonardo Rodrigo Domingues

GPU Optimization of Bounding Volume Hierarchies for Ray Tracing

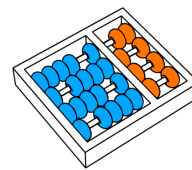
*Otimização em GPU de Bounding Volume
Hierarchies para Ray Tracing*

CAMPINAS
2015



University of Campinas
Institute of Computing

Universidade Estadual de Campinas
Instituto de Computação



Leonardo Rodrigo Domingues

GPU Optimization of Bounding Volume Hierarchies for Ray Tracing

Supervisor: **Prof. Dr. Helio Pedrini**
Orientador:

Otimização em GPU de Bounding Volume Hierarchies para Ray Tracing

MSc Dissertation presented to the Graduate Program of the Institute of Computing of the University of Campinas to obtain a Mestre degree in Computer Science.

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

THIS VOLUME CORRESPONDS TO THE FINAL VERSION OF THE DISSERTATION DEFENDED BY Leonardo Rodrigo Domingues, under the supervision of Prof. Dr. Helio Pedrini.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA POR Leonardo Rodrigo Domingues, sob orientação de Prof. Dr. Helio Pedrini.

Supervisor's signature / *Assinatura do Orientador*

CAMPINAS
2015

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

D713g Domingues, Leonardo Rodrigo, 1985-
GPU optimization of bounding volume hierarchies for ray tracing / Leonardo Rodrigo Domingues. – Campinas, SP : [s.n.], 2015.

Orientador: Hélio Pedrini.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Computação gráfica. 2. Algoritmos Ray tracing. 3. Estruturas de dados (Computação). I. Pedrini, Hélio, 1963-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Otimização em GPU de bounding volume hierarchies para ray tracing

Palavras-chave em inglês:

Computer graphics

Ray tracing algorithms

Data structures (Computer science)

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

Hélio Pedrini [Orientador]

Esteban Walter Gonzalez Clua

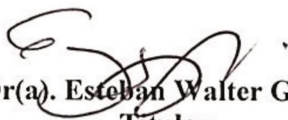
Jorge Stolfi

Data de defesa: 03-07-2015

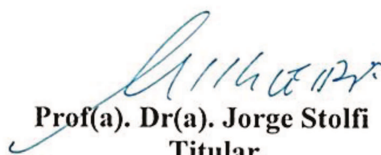
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Defesa de Dissertação de Mestrado em Ciência da Computação, apresentada pelo(a) Mestrando(a) **Leonardo Rodrigo Domingues**, aprovado(a) em **03 de julho de 2015**, pela Banca examinadora composta pelos Professores(as) Doutores(as):



Prof(a). Dr(a). Esteban Walter Gonzalez Clua
Titular



Prof(a). Dr(a). Jorge Stolfi
Titular



Prof(a). Dr(a). Hélio Pedrini
Presidente

GPU Optimization of Bounding Volume Hierarchies for Ray Tracing

Leonardo Rodrigo Domingues

July 03, 2015

Examiner Board / *Banca Examinadora:*

- Prof. Dr. Helio Pedrini (Supervisor / *Orientador*)
- Prof. Dr. Esteban Walter Gonzalez Clua
Instituto de Computação - UFF
- Prof. Dr. Jorge Stolfi
Instituto de Computação - UNICAMP
- Prof. Dr. David Menotti
Departamento de Computação - UFOP (Suplente)
- Prof. Dr. André Santanchè
Instituto de Computação - UNICAMP (Suplente)

Abstract

Ray tracing methods are well known for producing very realistic images at the expense of a high computational effort. Most of the cost associated with those methods comes from finding the intersection between the massive number of rays that need to be traced and the scene geometry. Special data structures were proposed to speed up those calculations by indexing and organizing the geometry so that only a subset of it has to be effectively checked for intersections. One such construct is the Bounding Volume Hierarchy (BVH), which is a tree-like structure used to group 3D objects hierarchically. Recently, a significant amount of effort has been put into accelerating the construction of those structures and increasing their quality. We present a new method for building high-quality BVHs on manycore systems. Our method is an extension of the current state-of-the-art on GPU BVH construction, Treelet Restructuring Bounding Volume Hierarchy (TRBVH), and consists of optimizing an already existing tree by rearranging subsets of its nodes using an agglomerative clustering approach. We implemented our solution for the NVIDIA Kepler architecture using CUDA and tested it on sixteen distinct scenes that are commonly used to evaluate the performance of acceleration structures. We show that our implementation is capable of producing trees whose quality is equivalent to the ones generated by TRBVH for those scenes, while being about 30% faster to do so.

Resumo

Métodos de *Ray Tracing* são conhecidos por produzir imagens extremamente realistas ao custo de um alto esforço computacional. Pouco após terem surgido, percebeu-se que a maior parte do custo associado a estes métodos está relacionada a encontrar a intersecção entre o grande número de raios que precisam ser traçados e a geometria da cena. Estruturas de dados especiais que indexam e organizam a geometria foram propostas para acelerar estes cálculos, de forma que apenas um subconjunto da geometria precise ser verificado para encontrar as intersecções. Dentre elas, podemos destacar as *Bounding Volume Hierarchies* (BVH), que são estruturas usadas para agrupar objetos 3D hierarquicamente. Recentemente, uma grande quantidade de esforços foi aplicada para acelerar a construção destas estruturas e aumentar sua qualidade. Este trabalho apresenta um novo método para a construção de BVHs de alta qualidade em sistemas *manycore*. O método em questão é uma extensão do atual estado da arte na construção de BVHs em GPU, *Treelet Restructuring Bounding Volume Hierarchy* (TRBVH), e consiste em otimizar uma árvore já existente reorganizando subconjuntos de seus nós por meio de uma abordagem de agrupamento aglomerativo. A implementação deste método foi feita para a arquitetura Kepler utilizando CUDA e foi testada em dezesseis cenas que são comumente usadas para avaliar o desempenho de estruturas aceleradoras. É demonstrado que esta implementação é capaz de produzir árvores com qualidade comparável às geradas utilizando TRBVH para aquelas cenas, além de ser 30% mais rápida.

Acknowledgements

First of all, I would like to thank my parents, for always encouraging me to keep going during difficult times. I also want to express my gratitude to my advisor, Helio Pedrini, for giving me the opportunity to finish my research and leading me through the right paths. My gratitude extends to my friends, for all the encouragement and letting me know early on how difficult this road would be. I thank the Eldorado Research Institute as well, for granting me time to dedicate to my studies.

Lastly, I would also like to thank Timo Aila, Samuli Laine, and Tero Karras for making their GPU ray tracing framework available, as well as the Stanford University Computer Graphics Laboratory for the scenes Armadillo, Buddha, Bunny and Dragon, the Stereolithography Archive at Clemson University for Skeleton Hand, the Georgia Institute of Technology for Turbine Blade, Anat Grynberg and Greg Ward for Conference, Ingo Wald for Fairy Forest, Samuli Laine for Hairball, Marko Dabrovic for Sibenik, Frank Mehl for Sponza, Jonathan Good for Arabic, Babylonian and Italian, Hameed Nawaz for Time Machine and Bangor University, UK for Welsh Dragon.

Contents

Abstract	6
Resumo	7
Acknowledgements	8
1 Introduction	1
1.1 Problem Description and Motivation	2
1.2 Objectives and Contributions	3
1.3 Text Structure	4
2 Related Work	5
2.1 Ray Tracing	5
2.2 Acceleration Structures	7
2.2.1 Kd-tree	8
2.2.2 Bounding Volume Hierarchy	8
2.3 BVH Optimization	10
3 CUDA and the Kepler Architecture	13
3.1 The Kepler Architecture	13
3.2 CUDA	16
3.3 Parameters to Optimize	17
3.3.1 Division of Work	17
3.3.2 Memory Access	18
3.3.3 Registers	18
3.3.4 Divergence	19
4 ATRBVH	20
4.1 Agglomerative Treelet Restructuring	20
4.1.1 Overview	20
4.1.2 Treelet Formation	21
4.1.3 Distance Metric	22
4.1.4 Merge Clusters	22
4.1.5 Treelet Reconstruction	23

4.1.6	Post-processing	23
4.1.7	Parameters	24
4.2	Implementation and Optimizations	24
4.2.1	Tree Traversal	25
4.2.2	Treelet Restructuring	25
4.2.3	Distance Matrix	27
4.2.4	Global Memory Usage	28
5	Results	30
5.1	Parameter Choice	31
5.2	Ray Tracer Analysis	34
6	Conclusions and Future Work	37
	Bibliography	39
A	Other Approaches	44
A.1	Tree Rotation	44
A.1.1	Parameters	45
A.2	Divisive Treelet Restructuring	45
A.2.1	Parameters	48
A.3	Results	48

List of Tables

5.1	Test results for ATRBVH	33
5.2	Results averaged over all test scenes	34
A.1	Test results for other approaches	47

List of Figures

2.1	Rendering a scene with Ray Casting	6
2.2	Rendering a scene with Recursive Ray Tracing	7
2.3	BVH for small data set of triangles	9
3.1	Kepler architecture SMX	14
3.2	Kepler architecture overview	15
4.1	Agglomerative treelet restructuring process for a treelet formed from six leaves	21
4.2	Updating the distance matrix	27
5.1	Percentage of time spent on each step of ATRBVH for six test scenes .	35

Acronyms

AoS Array of Structures. 18

ATRBVH Agglomerative Treelet Restructuring Bounding Volume Hierarchy. 34, 35, 37

BSP Binary Space Partitioning. 8

BVH Bounding Volume Hierarchy. vi, 1–3, 8–12, 20, 23–25, 30, 31, 34, 35, 37, 44, 49

CPU Central Processing Unit. 2, 10, 11

CUDA Compute Unified Device Architecture. 13, 14, 16, 24, 25, 37, 44

DRAM Dynamic Random-Access Memory. 14

DTRBVH Divisive Treelet Restructuring Bounding Volume Hierarchy. 48, 49

GPGPU General Purpose Computing on Graphics Processing Units. 13

GPU Graphics Processing Unit. 2, 3, 8, 10, 11, 13, 14, 16–19, 31, 34, 37, 44, 48, 49

HPC High Performance Computing. 13

ILP Instruction-Level Parallelism. 19

LBVH Linear Bounding Volume Hierarchy. 10, 30, 34, 44, 48

LRU Least Recently Used. 14

SAH Surface Area Heuristic. 2, 3, 9–12, 20–24, 26, 27, 29–31, 34, 38, 44, 45, 48

SIMT Single-Instruction, Multiple Thread. 2

SM Streaming Multiprocessors. 13, 16, 19

SoA Structure of Arrays. 18

TRBVH Treelet Restructuring Bounding Volume Hierarchy. vi, 30, 31, 34–37, 44, 48, 49

WLP Warp-Level Parallelism. 17

Chapter 1

Introduction

Ray tracing is a rendering technique that was originated in the sixties and is still used today to produce realistic images from computer models [6, 12, 47]. It consists in simulating the complex paths that rays of light take while traveling from an object to its observers eyes. A large number of rendering methods have evolved from ray tracing in order to improve the fidelity of images produced and enable the representation of phenomena such as depth of field and motion blur. Nowadays, ray tracing variations are widely used in fields such as architecture, cinema and engineering.

Rendering a high quality image through ray tracing is often a time demanding process, which requires a high computational effort. Most of that effort is redirected toward calculating the intersection between rays and the scene triangles. In order to accelerate those calculations, special data structures are employed to organize the geometry in such a way that, for each ray, only a subset of the triangles has to be tested.

The acceleration of ray tracing is a topic that has recently received sizable attention, and many different data structures have been applied for that purpose. Among those, BVHs are currently the most used, in great part due to being better at handling animated scenes and having a low memory footprint [18, 43]. Vinkler et al. [40] recently published a study comparing the ray tracing performance of BVHs and kd-trees on manycore architectures, showing that the former outperformed the latter for the majority of the tested scenes, with the exception of those with very high triangle

counts and significant levels of occlusion.

1.1 Problem Description and Motivation

In order to optimize a ray tracing application, it is important to reduce the construction time of the acceleration structures, especially for interactive applications and rendering animated scenes, since those structures must be rebuilt every frame due to changes in the geometry. Graphics Processing Units (GPUs) can be extremely helpful toward that end, since they are accessible and affordable high performance computing devices. In this research, we focused on using BVHs as our acceleration structure.

Building a high quality BVH is a computationally expensive process, which until recently has been performed exclusively on Central Processing Units (CPUs). Most algorithms that produce those structures work by processing the data in either a top-down [31] or a bottom-up order [45], and would take a considerable amount of time to execute. When GPUs began being employed to build BVHs, alternative algorithms were proposed so as to make better usage of the Single-Instruction, Multiple Thread (SIMT) architecture. Some of these algorithms could generate trees much faster than the traditional approaches, however, at the cost of producing structures with an inferior quality [14, 24, 30, 35].

BVH quality also plays an important part in the rendering time of ray tracing applications, since higher quality structures are more efficient in reducing the number of ray-triangle intersections that are required to render an image, thus speeding the process up. The quality of an acceleration structure is directly measured by the amount of rays per second that can be traced using it. However definitive, that value can only be obtained after the structure has already been built and is very susceptible to changes in the hardware used. Goldsmith and Salmon [16] and later MacDonald and Booth [31] introduced the idea that the quality of a structure can be derived from the number of intersections that are required to trace an arbitrary non-terminating ray through it. They also demonstrated how to calculate an estimate to that value, which they called Surface Area Heuristic (SAH) cost. It is important to notice that although that value has a high correlation with the performance of BVHs

when comparing trees built by using the same method with different parameter values, the results are not always consistent when comparing structures constructed through different methods [1].

The challenges involved in optimizing BVHs revolve around finding a balance between construction speed and structure quality. If we were to calculate node combinations in order to form all possible hierarchies for a given set of triangles, we could find the structure with the highest quality achievable. However, finding such a tree would require an enormous amount of time. By approximating the optimization stage, we can produce BVHs that still have a good quality but are faster to produce, so that when we combine the time required to construct the tree and trace the rays, the overall performance is maximized.

1.2 Objectives and Contributions

Karras and Aila [25] proposed a method for constructing high quality BVHs on GPU that is fast enough to be used in real time applications while achieving ray tracing performance competitive with the most time demanding algorithms. Their solution consists in optimizing an existing tree by looking at treelets, small local subsets of tree nodes, and rearranging their nodes in order to minimize the overall tree SAH cost. Since execution time and memory requirements grow rapidly with treelet sizes, only small treelets can be viably used.

Our objective in this research is to reduce build times for high quality BVHs even further. To that purpose, we expand on Karras and Aila’s original work by proposing a new method for rearranging treelet nodes in a greedy, bottom-up fashion, enabling the usage of larger treelet sizes while keeping construction times competitive. In fact, by using treelet sizes slightly larger than the original method, we were able to produce BVHs with equivalent quality in about 30% less time. The quality of the generated trees can be increased by using larger treelets, at the expense of higher construction times. The obtained results also show that, even when using the same treelet size, the dynamic programming solution employed originally to find the optimal treelet structure can be replaced by an approximated agglomerative search without a

significant reduction in tree quality.

1.3 Text Structure

The remainder of this text is organized as follows: Chapter 2 provides more detail about ray tracing and reviews previous approaches used to create and optimize bounding volume hierarchies, Chapter 3 presents a brief overview of CUDA and the Kepler architecture, Chapter 4 describes our tree construction algorithm and details our implementation and optimizations that were performed, Chapter 5 displays and discusses the results of our tests and Chapter 6 presents conclusions and lists some possibilities for future work. Appendix A lists other approaches that were tried during the course of this research.

Chapter 2

Related Work

This chapter provides a brief overview of the publications that led up to the development of this research. Section 2.1 reviews commonly used rendering methods that can be dramatically accelerated by optimizing the process of tracing rays through a scene. Section 2.2 discusses data structures that can be employed to accelerate ray tracing. Section 2.3 lists previous methods that were used to optimize Bounding Volume Hierarchies.

2.1 Ray Tracing

In 1968, Appel [6] proposed a technique for drawing and shading vivid two-dimensional images from computer models of 3D objects. His method, which would later be known as Ray Casting, consists in calculating the trajectory of several light rays emitted from the scene viewpoint in the direction of the 3D objects. After finding the range of coordinates for the discrete projection plane that must be rendered, the algorithm generates rays coming from the observer's position and going through each of those coordinates. When one of those rays intersects with one of the objects being rendered, another check is necessary to determine if the collision point is visible from the light source; if that is the case, the corresponding plane coordinate receives a light shade, and if not, it will not be illuminated. By using this method, Appel was able to generate shaded images that had clear shadow boundaries visible. The Ray Casting process is illustrated in Figure 2.1.

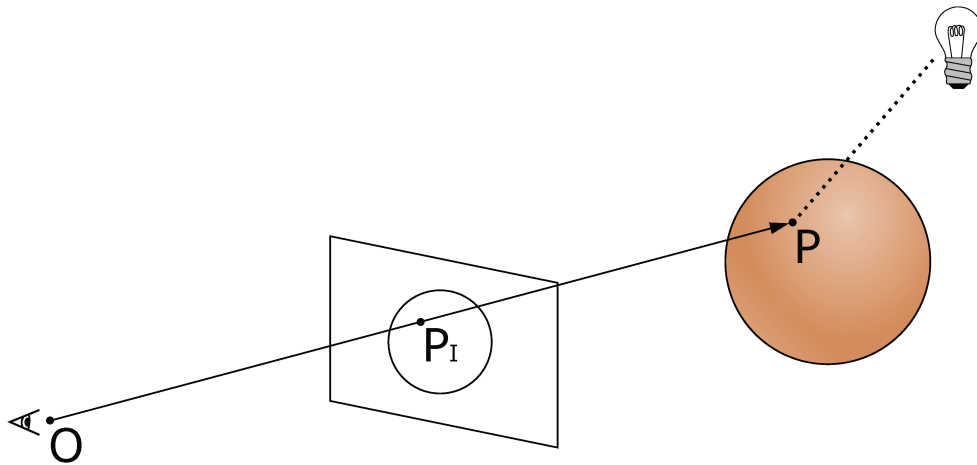


Figure 2.1: In Ray Casting, rays are generated from the observer (O) in the direction of coordinates of the projection plane. For each ray, the algorithm determines the point of intersection between that ray and the object being rendered (P) and attributes the color of that object to the corresponding coordinate in the projection plane (P_I). Shadow Rays (illustrated with a dotted line) are also traced from the points of intersection to the light source, to determine if those positions are beyond the shadow boundary.

Whitted [47] extended on the existing ray casting algorithm by taking global illumination information into account during the rendering process and enabling the generation of images depicting shadows, refraction and reflection for perfectly smooth surfaces. Instead of terminating the visibility calculation when a ray hits a surface, each intersection generates up to three additional rays that needed to be traced: one for the surface reflection, one for refraction and another to determine if the intersection point lays in shadow. The direction of reflected and refracted rays is calculated through classic ray optics. This method became known as Recursive Ray Tracing or Whitted Ray Tracing.

In his work, Whitted also pointed that up to 95% of the time required to render a scene would be spent on ray-surface intersection calculations, and if that part of the algorithm could be optimized, great performance gains could be expected.

In Ray Casting and Recursive Ray Tracing, the direction of rays is determined precisely through ray optics calculations. Cook [12] proposed a method called Distributed Ray Tracing, in which the direction of rays is calculated by sampling an

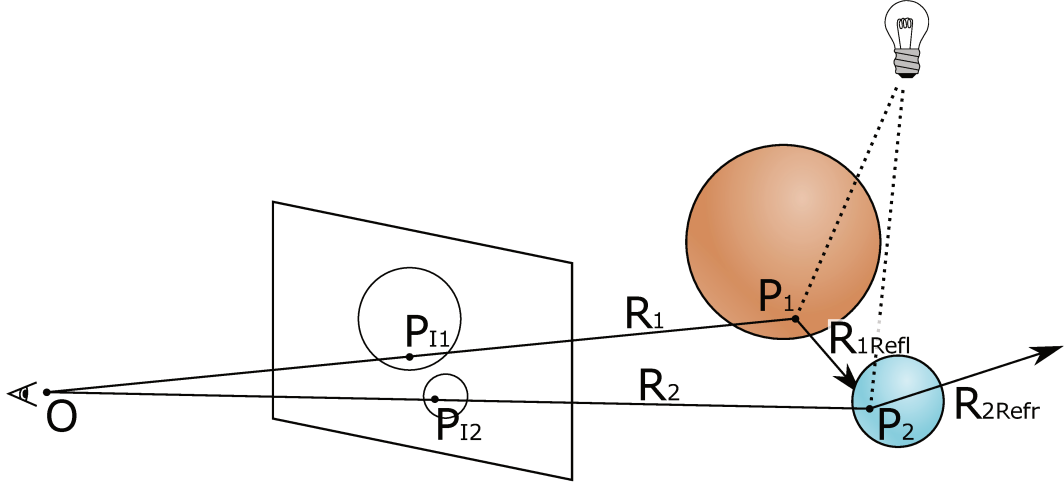


Figure 2.2: Recursive Ray Tracing expands on Ray Casting (Figure 2.1) by bouncing incident rays multiple times to account for reflection and refraction effects. The ray R_1 hits the larger sphere and is reflected in the direction R_{1Refl} , while R_2 is refracted in the direction R_{2Refr} . Shadow rays are also traced from the points of intersection P_1 and P_2 to the light source, to determine if those points are in shadow. Only one bounce of light is depicted here.

analytic function. By sampling different functions, such as time, the camera lens or even the solid angle of the light sources, his rendering algorithm is able to include effects such as motion blur, depth of field, translucency, penumbra and fuzzy reflections to rendered images, at no additional cost when compared to traditional ray tracing.

More recently, a series of stochastic rendering methods, such as Bi-Directional Path Tracing [29], Metropolis Light Transport [39] and Photon Mapping [23], were devised. Even though they are not direct variations of Ray Tracing, these methods also rely heavily on ray-surface intersection calculations and can benefit from the results of this research.

2.2 Acceleration Structures

During ray tracing, we need to find the intersection points between a number of rays and the scene being rendered. The naïve solution of testing each ray against all of the scene geometry has a time complexity of $O(MN)$, where M is the number of rays traced and N is the number of objects in the scene. Since both these values tend

to be very high, rendering complex scenes in a reasonable time is often not possible using this approach.

In order to reduce the number of ray-primitive intersections, special data structures can be used to index the scene geometry and limit the set of primitives that have to be checked for each given ray. These structures can be classified into two groups as to whether they organize the scene geometry in a hierarchy or subdivide the scene space [27, 41].

Some commonly used acceleration structures are Bounding Volume Hierarchies [36], Kd-trees [7], Octrees [15] and Grids [4].

2.2.1 Kd-tree

Kd-trees [7] are a generalization of Binary Space Partitioning (BSP) trees [20]. In these structures, data is organized by subsequently dividing the search space into two using axis-aligned hyperplanes. Because of that property, kd-trees can be used as binary search trees to locate objects in the scene and efficient divide-and-conquer and branch-and-bound traversal algorithms can be implemented for them [22].

In his thesis [20], Havran developed a methodology for comparing the use of different acceleration structures in ray tracing. His results showed that, for his set of tests, kd-trees were the most efficient data structures for organizing the scene geometry.

Even though kd-trees have been replaced by BVHs as the most used acceleration structure on GPU ray tracers [38], they are still able to achieve good results and new methods for creating and traversing them are still being researched [10, 11, 19, 37, 48].

For kd-tree traversal algorithms, we redirect readers to the recently published review by Hapala and Havran [19].

2.2.2 Bounding Volume Hierarchy

A bounding volume hierarchy [36] is a data structure used to group 3D objects hierarchically. It has the form of a tree structure, where each leaf represents one or more of the objects being stored and internal nodes represent a grouping of those objects. Each node holds a conservative bounding volume of all objects that descend from that

node, usually in the form of an axis-aligned bounding box. Weghorst et al. [46] investigated the criteria that are involved in the selection of the optimal bounding volume used. Figure 2.3 illustrates a two-dimensional BVH for a small set of triangles.

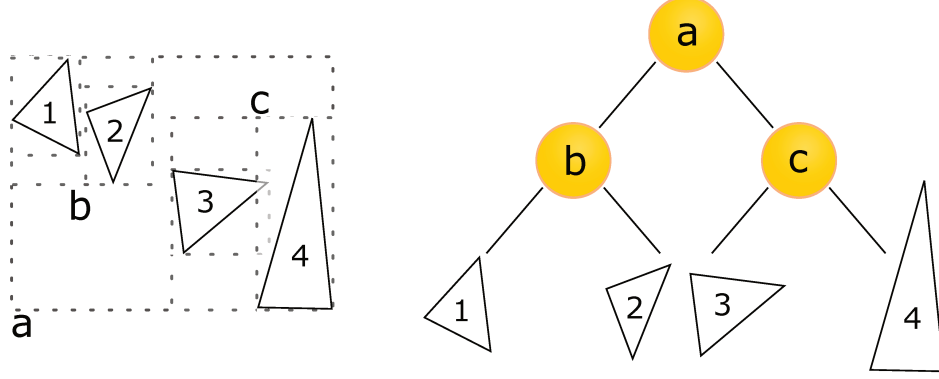


Figure 2.3: BVH constructed from a small data set of four triangles. Each internal node represents a subset of the input containing two or more triangles and stores the bounding box of that subset.

A common approach to generating BVHs that optimize ray tracing performance is to minimize their SAH cost [16, 31] during the tree construction. That value can be defined as follows

$$\text{SAH} = \frac{1}{A_t} \left(C_i \sum_{n \in I} A(n) + C_t \sum_{n \in L} A(n) N(n) \right) \quad (2.1)$$

where A_t corresponds to the surface area of the root node, $A(n)$ is the surface area of node n , C_i and C_t are relative costs for traversing an internal node and for performing a ray-triangle intersection, respectively, I is the set of internal nodes, L is the set of leaves and $N(n)$ is the number of triangles referenced by leaf n .

The first method for the automatic generation of BVHs was created by Goldsmith and Salmon [16]; it consists in adding nodes to the tree one at a time, in a position that minimizes the overall SAH cost of the tree. MacDonald and Booth [31] later proposed an algorithm for constructing BVHs based on a top-down partitioning of the triangles that also has the objective of minimizing the SAH cost of the generated tree. Walter et al. [45] went in the opposite direction by creating BVHs in a bottom-up, agglomerative approach and was able to produce trees with SAH costs lower than

those for top-down methods, at the expense of longer processing times.

Until a few years ago, acceleration structures would almost exclusively be built on CPUs and, as a result, the algorithms used were mostly serial. When GPUs began being used to accelerate ray tracing, new algorithms were developed for taking advantage of the massive level of parallelism available at manycore processors. Lauterbach et al. [30] proposed a new method called Linear Bounding Volume Hierarchy (LBVH) for constructing BVHs in parallel, by first sorting the scene triangles using a space-filling curve and then recursively splitting that data to create the internal nodes of tree. Although being able to run extremely fast, the quality of the trees produced with that method was not on par with that of the structures produced with traditional approaches.

Pantaleoni and Luebke [35] and Garanzha et al. [14] proposed a new method that could still run efficiently on GPUs and produce trees of higher quality than LBVH by using a hierarchical grouping of the input data and using the SAH to optimize the top levels of the structure. Karras [24] improved on LBVH by showing that the tree construction stage can be completely parallelized by processing all nodes simultaneously. Apetrei [5] also improved on LBVH by performing both the tree construction and the bounding box calculation in a single bottom-up traversal.

A BVH can be traversed by checking a ray against the bounding volume of the BVH root node: if they intersect each other, then the left and right children of that node are recursively processed. This step is repeated until a leaf node is reached, and at that time the ray is checked for collisions against each object represented by that leaf. Aila and Laine [2, 3] identified the gaps between previous acceleration structure traversal algorithms and the theoretical optimum, and introduced the algorithm which is the current state-of-the-art for BVH traversal on GPUs.

2.3 BVH Optimization

There are two common approaches for optimizing the SAH cost of a BVH during its construction stage. The first and most used is to build the tree in a top-down, divisive manner [31]: at each iteration of the algorithm, a split plane is chosen to create an

internal node that divides the triangles into two groups, which are then recursively divided themselves. The split plane is chosen so as to minimize the SAH cost at each step. Since finding the best split planes is a very computationally expensive task, various optimizations have been proposed which provide a trade-off between construction time and quality by approximating that search [21, 42, 44].

The second approach to constructing a BVH while optimizing its SAH cost is to perform a bottom-up, agglomerative tree construction [45]. For this method, each triangle is regarded as a cluster of size one. At each iteration, the pair or clusters for which a dissimilarity function returns the lowest value is merged; this process is repeated until all triangles are grouped under a single cluster. Agglomerative algorithms often lead to the construction of trees that perform better than their divisive counterparts, however, at the expense of longer construction times. Gu et al. [17] later proposed a method for efficiently building BVHs on multicore CPUs using an approximate agglomerative approach.

Kensler [28] optimized already existing BVH structures by making local modifications derived from tree rotation operations to them, which lead to the generation of trees with very high quality. Bittner et al. [8] proposed another optimization technique which performed global modifications to an existing tree: by relocating certain tree nodes, their method is able to produce the current gold standard in tree quality.

Karras and Aila [25] introduced the idea of optimizing existing trees on GPU by rearranging subsets of their nodes in order to minimize the overall SAH cost. Their algorithm performs a bottom-up traversal of the tree and, for each internal node that is encountered, a treelet is formed by using that node as its root. Treelets can be thought of as small binary trees themselves, containing nodes that are connected in the original tree. The best topology is then found for each treelet by testing the cost of all possible rearrangements of its leaves using a dynamic programming algorithm. Among the methods used to produce BVHs on GPUs, this one generates the highest quality structures. Karras and Aila [26] also published a patent describing an agglomerative approach to restructuring treelets. Our work was developed independently and without prior knowledge of that patent.

Recently, Bittner et al. [9] proposed a new algorithm for constructing BVHs incre-

mentally by sequentially adding new triangles to the existing tree at specific positions that minimize the global SAH cost. Their method is able to create BVHs with quality comparable to the commonly used SAH top-down builders.

Stich et al. [38] introduced a new construction algorithm that improves the ray tracing performance of BVHs, specially for scenes that are not regularly tessellated, by performing spatial splits in the input geometry. Karras and Aila [25] also used spatial splits in their method and proposed an approach to performing triangle splitting on GPUs. In this paper, we do not employ triangle splitting, since our objective is to focus on the treelet restructuring stage.

Chapter 3

CUDA and the Kepler Architecture

In this chapter, we introduce readers to General Purpose Computing on Graphics Processing Units (GPGPU) by presenting an overview of Compute Unified Device Architecture (CUDA) and the Kepler architecture, which is the base for the NVIDIA GTX 770 GPU used during our experiments. We focus on the aspects of parallel programming that were necessary to reach the results presented in this dissertation. For more information about the Kepler architecture and other possible optimizations, readers can check the CUDA C Programming Guide [34] and CUDA C Best Practices Guide [33].

3.1 The Kepler Architecture

GPUs are very complex in nature and have constantly evolved over time to become faster and more power efficient. The architecture for the iteration of NVIDIA GPUs used in this research is called Kepler, and it was designed to provide a high double precision performance for High Performance Computing (HPC) [32].

Modern NVIDIA GPUs consist of a number of Streaming Multiprocessors (SMs), which are elaborated structures capable of executing a great volume of tasks in parallel. The current generation of SMs, employed in the Kepler architecture, is called

SMX. Each SMX contains 192 single-precision cores, each having its own floating point and integer arithmetic logic units. Cores in a SMX share a common instruction cache, register file, read-only cache and a shared memory / L1 cache. Different Kepler GPUs have different numbers of SMXs, however, the structure of those SMXs remains the same throughout all models. A simplified version of the SMX is depicted in Figure 3.1.

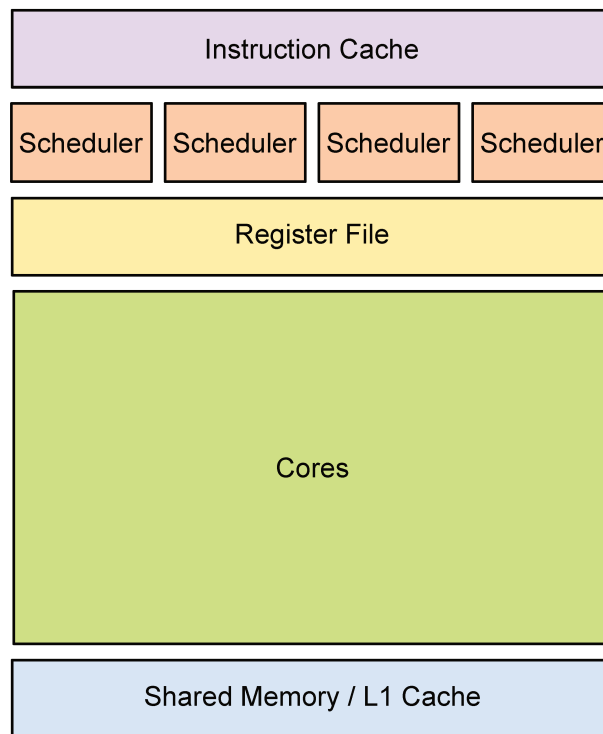


Figure 3.1: The Kepler architecture SMX combines the power of 192 CUDA cores into a single structure. With four schedules, the SMX can dispatch up to eight instructions per warp per cycle. A single on-chip memory block of 64KB is used for the shared memory and L1 cache, and it can be divided evenly between the two or split as 48KB/16KB or 16KB/48KB. The register file and instruction cache are also common for all cores in the SMX.

The amount of Dynamic Random-Access Memory (DRAM) found on Kepler GPUs is different for each model. Alongside the DRAM, a Kepler GPU also has available a 1536KB L2 cache that works in a Least Recently Used (LRU) policy, and is the same size for all GPU models. When a value is requested from memory, it is first searched in the SMX L1 cache, and then in the GPU L2 cache. If it is not found, a load from the DRAM is issued.

A SMX executes groups of 32 parallel threads that are called warps. Threads in a warp are always in synchrony, that is, at any given time, the same instruction is being executed for all threads in that warp. SMXs can have up to 64 resident warps, and they alternate the execution of those warps to maximize the usage of their various pipelines. Each SMX has four warp schedulers, and each of those can dispatch up to two instructions per warp per cycle.

The Kepler architecture, which serves as base for the NVIDIA GTX 770 used in our tests, is illustrated in Figure 3.2.

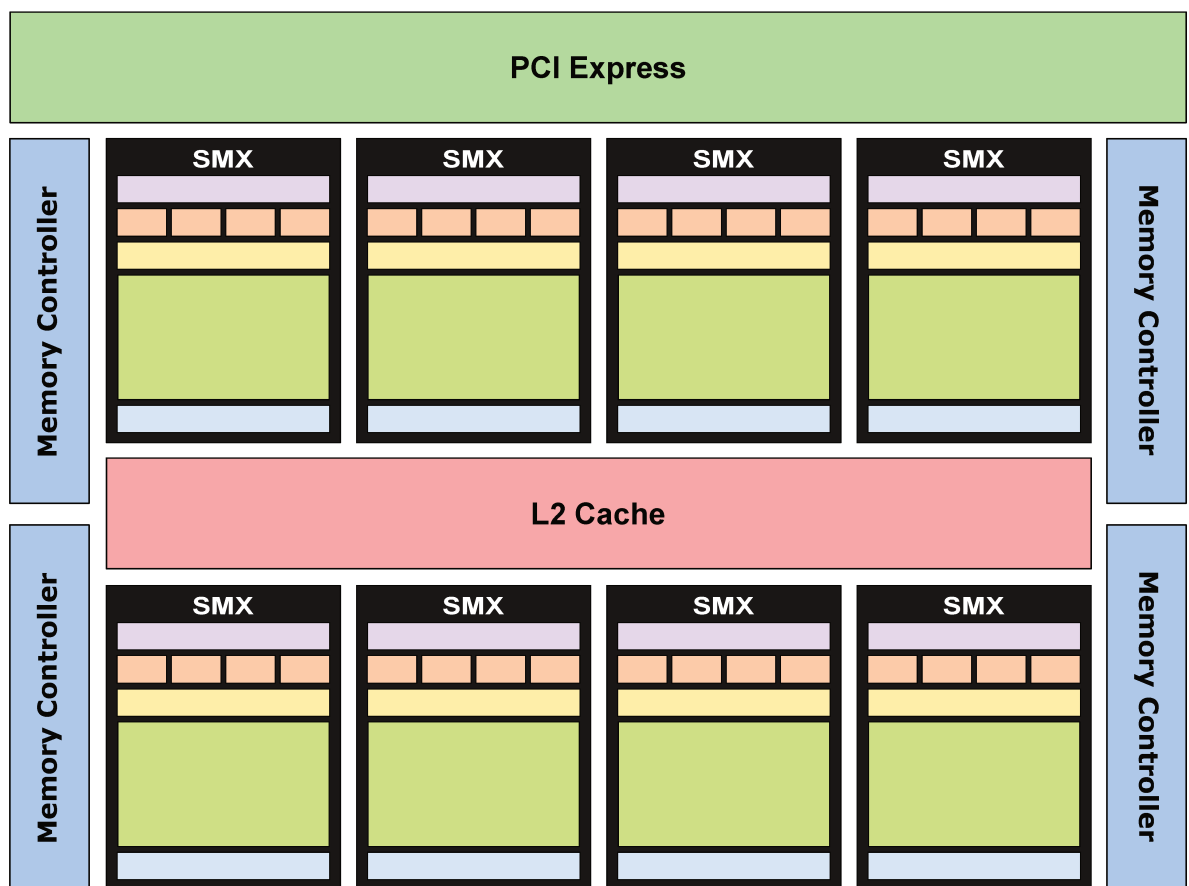


Figure 3.2: The NVIDIA GTX 770 has eight SMXs, each capable of handling up to 64 warps. When a memory address is requested by one of the four memory controllers, it first goes through a L2 cache that is shared among all SMXs.

3.2 CUDA

The CUDA programming model was created to harness the power of GPUs for general purpose processing and enable the creation of applications that can transparently scale with an increased number of cores, just as graphics applications do. Differently from traditional CPU development, when creating applications using CUDA one should focus on achieving high levels of parallelism instead of executing tasks sequentially as fast as possible.

CUDA applications are separated into the host part, which runs on CPU, and the device part being executed on the GPUs. Computer programs that are created using CUDA and run on GPUs are called kernels. They are commonly written through a high-level programming API using the C language, although wrappers for several other languages exist.

Parallelism is achieved in CUDA by dividing the work into small chunks that are executed by threads. Those threads are organized into blocks, and each block is assigned to a different SM, where it will be executed. Threads in a block can share data by using the shared memory that is available to each SM.

The grid configuration, that is, the number of blocks that will be used to run a kernel and the dimension of those blocks, can be determined by the developer. Although sometimes kernels are developed with a specific hardware in mind and implicitly contain some of that hardware characteristics, most often the grid configuration can be changed during runtime and set according to the exact board that is being used, enabling the same program to be optimized for a wide variety of GPUs.

When programming using CUDA, one has access to several different memory types that must be balanced to reach optimal performance: global memory is the most abundant and is accessible to all threads, but is also the one with the highest latency; texture memory is read-only and optimized for a 2D access pattern; constant memory is read-only and used to store constants and kernel arguments; shared memory can be used to share data between threads in a block and is almost as fast as using registers, although very limited in size; local memory is a portion of the global memory to where registers are spilled; registers are the fastest place to store data, but are limited to

65536 32-bit registers per SMX.

3.3 Parameters to Optimize

This section presents a number of parameters that have to be balanced to reach optimal performance. The values listed here are specific for the Kepler architecture, and may be different in other GPUs.

3.3.1 Division of Work

The amount of threads created for each kernel and how those threads are grouped are parameters that can be determined during runtime by arguments specified using a special kernel syntax. The application can configure the block size, that is, the number of threads that compose each block, and the grid size, which is the total number of blocks that will be launched.

Block size and grid size are three-dimensional values. Since the SMX handles warps and not individual threads, the number of threads per block should ideally be a multiple of 32. Because a block has to be run entirely by one SMX and needs to share resources with the other blocks that are allocated to that SMX, its maximum number of threads is limited to 1024.

Each SMX can be responsible for up to 64 warps at any given time. The occupancy value indicates how many warps each SMX is handling in relation to that maximum value. In general, the higher the occupancy, the more work a SMX scheduler has available at each cycle to choose from. Having a high occupancy is important because it helps hide the latency of instructions, but achieving 100% is not always possible, seeing that the number of warps that can be allocated in a SMX depends on the amount of shared memory used by each block and the number of registers used per thread. The term Warp-Level Parallelism (WLP) refers to the use of multiple warps in parallel to hide instruction latency.

The objective of tuning these values is guaranteeing that the GPU will have enough work that can be executed in parallel to keep it as busy as possible.

3.3.2 Memory Access

Copying memory between the host and the device is one of the operations that incur the greatest latency in GPU applications. Whenever possible, it is best to implement all steps of an algorithm on the GPU, so data does not have to be copied back and forth to the main system memory. Memory allocation also accounts for a significant percentage of execution times, and should be avoided as often as possible with measures such as allocating memory at the application start instead of every frame and reusing allocated memory for different structures.

Most GPU applications are bound by the speed at which data from the global memory can be accessed, and so understanding how the L1 cache works plays a fundamental role in optimizing them. Data is not accessed per thread; instead, requests for all threads of a warp are combined and the required memory is read using 128-byte wide transactions into the L1 cache. That way, if threads in a warp access memory positions that are stored sequentially, fewer read transactions have to be issued, and the threads can just read their values from the L1 cache. When a set of threads accesses memory positions that are adjacent to each other, we say those accesses are coalesced. Making memory accesses coalesced reduce the number of read transaction that have to be issued, thus increasing performance.

Preventing cache thrashing, that is, the removal from the cache of data that is commonly used, is also important to optimizing performance. To that end, using the correct memory layout for data structures is essential. When storing a collection of some data structure in memory, it is common to create a Array of Structures (AoS) type layout; however, it may be beneficial to consider using a Structure of Arrays (SoA) layout to make memory accesses more coalesced and avoid cache thrashing.

Shared memory should be used whenever possible to prevent accessing the more expensive global memory or to explicitly cache values that are used frequently.

3.3.3 Registers

The number of registers used by each thread directly affects the occupancy, so it is important to keep that value as low as possible. If there is a high Register Pressure,

that is, a thread requires more registers than are available in the SMX, the exceeding variables will be spilled to local memory. The `--launch_bounds--()` qualifier can be employed to force the compiler to use a specific amount of registers, causing the remaining registers to spill to local memory. While this qualifier can be useful to reach a certain occupancy goal, it has to be used with care, since oftentimes it leads to decrease performance.

The Kepler architecture introduced the shuffle intrinsic operations, which allow a thread to read registers from other threads that are in the same warp at a very low cost. Using these instructions, it is possible to store strategic values that are accessed by more than one thread in a warp in registers, in order to save shared memory.

Instruction-Level Parallelism (ILP) can also be employed to hide instruction latency on Kepler GPUs. Although the order of instructions is mostly determined by the compiler, there are cases where switching the order of statements in the C code can lead the compiler to producing machine code with higher ILP.

3.3.4 Divergence

When there is a branch in the kernel, the SM needs to execute separately each divergent path, while all other threads are stalled. This means that branching should be kept to a minimum, especially if both paths are taken by different threads in a warp.

Chapter 4

ATRBVH

This chapter describes the methodology used in our experiments. In Section 4.1, we provide a brief overview of the proposed algorithm and then give a detailed description of each of its steps. Section 4.2 presents implementation details that we believe are useful for understanding and replicating this work.

4.1 Agglomerative Treelet Restructuring

Our method [13] improves the one presented by Karras and Aila [25] by modifying its treelet restructuring stage: instead of evaluating all possible node topologies to find the one with the lowest SAH cost, we use a greedy algorithm based on agglomerative clustering to rearrange our treelets (Figure 4.1). While this process results in the generation of treelets that might not have the lowest cost possible, it runs faster than the original, therefore enabling the usage of larger treelet sizes without drastically changing execution times. We were motivated by the idea that, by increasing the treelet size, changes to the BVH would be more global in scope, thus allowing it to be modified to a greater extent.

4.1.1 Overview

The algorithm starts with a bottom-up traversal of the tree. Upon reaching each node, a treelet is formed by using that node as its root and then restructured to have

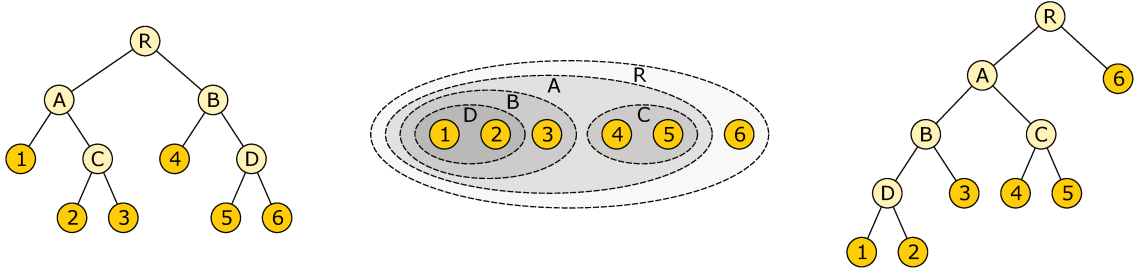


Figure 4.1: Agglomerative treelet restructuring process for a treelet of size six. Leaf nodes are represented through numbers, whereas internal nodes are represented through letters. Left: original treelet, before optimization takes place. Center: treelet leaves grouped by using agglomerative clustering. At each step, the two clusters closer to each other, given a certain distance function, are merged. Right: the treelet is reconstructed. At the end of each clustering step, each pair of elements that were grouped are connected using an internal node. This process does not require the allocation of new nodes, since the original ones can be re-utilized by just changing their pointers.

its SAH cost minimized. For the treelet optimization process, each leaf is regarded as a cluster of size one, and a dissimilarity value is calculated for each cluster pair. The pair with the smallest dissimilarity value is chosen to be merged, and an internal node is added to the treelet, with the nodes corresponding to the original clusters as its children. Both clusters are removed from the group, and a new one is added to represent the newly added internal node. These steps are repeated until only one cluster remains. The pseudocode is shown in Algorithm 1 and its main stages are detailed in the following sections.

4.1.2 Treelet Formation

At the treelet formation stage (line 2), we follow the same idea used by Karras and Aila [25] and greedily choose the nodes with the largest surface area, expecting them to be the ones with the greatest optimization potential. At the beginning of the process, the root node is added to the internal node set and its children are added to the set of leaves. At each iteration, the leaf with the largest surface area is moved to the internal node set and its children are added as leaves. This process is repeated until the treelet has grown to the desired size.

Algorithm 1: RearrangeTreelets

```

1 for internal node i in BVH do
2   treelet  $\leftarrow$  FormTreelet(i)
3   clusters  $\leftarrow$  treeletLeaves
4   while length(clusters) > 1 do
5     distances  $\leftarrow$  []
6     foreach pair of clusters (x, y) do
7       d  $\leftarrow$  Dissimilarity(x, y)
8       distances  $\leftarrow$  (d, x, y)
9     end
10    (m, n)  $\leftarrow$  FindMinimumDistance(distances)
11    o  $\leftarrow$  MergeClusters(m, n)
12    clusters.remove(m)
13    clusters.remove(n)
14    clusters.add(o)
15  end
16 end

```

4.1.3 Distance Metric

The agglomerative clustering process requires a dissimilarity or distance metric to be specified. At each iteration, the pair of nodes that are closer to each other using the given metric will be merged. Since the objective is to minimize the overall SAH cost of the tree, we chose the distance between two clusters to be the surface area of the bounding box containing them.

Calculating the distance between two clusters (line 7) is an expensive operation. In order to prevent it from being repeated for each cluster pair at each step, we borrow the idea of caching those values from Gu et al. [17]. At the beginning of the optimization process, the distances between all cluster pairs are pre-calculated and stored in a triangular matrix. We chose to use a lower triangular matrix, so the distance between clusters i and j will be stored at position (i, j) if $i > j$ or (j, i) otherwise.

4.1.4 Merge Clusters

After the distances have been calculated, the next step is to find the pair of clusters that are closest to each other (line 10) and merge them (line 11). When two clusters

are merged, the corresponding treelet nodes, which can be either internal nodes or leaves, must be connected by a new internal node that references them as children. In practice, one of the original internal nodes of the treelet is updated to represent this new node, so no extra memory has to be allocated during the merge process.

At each step, the distance matrix will also have to be updated by removing the two clusters that were merged and adding the newly formed one (lines 12–14). In order to keep the matrix compact, we replace the first removed cluster with the new one, and move the last valid cluster to the other vacant spot, as described by Gu et al. [17]. The distances between the new cluster and the others also need to be calculated so the matrix is complete once again.

4.1.5 Treelet Reconstruction

Instead of performing changes to the treelet at each iteration, the proposed modifications are stored in a list. Each list entry contains the index of the internal node that should be modified and the indices of its two children (the nodes corresponding to the clusters that were merged to originate it). After the whole treelet has been processed, the SAH cost of the newly generated topology is compared with the original treelet cost: only if the cost has decreased will the changes stored in the list be applied to the treelet, thus preventing the BVH from receiving changes that might increase its overall SAH cost.

4.1.6 Post-processing

Throughout the optimization process, each of the tree leaf nodes reference only one triangle. However, the SAH cost of a BVH can generally be reduced further by collapsing some of its subtrees, that is, replacing all the internal nodes corresponding to a particular subtree by a single leaf node that indexes all the triangles that could be reached from the root of that subtree. This collapsing step is performed after all treelets have been restructured.

In order to decide which subtrees should be collapsed, the SAH cost of that subtree

should be compared with the cost of the collapsed subtree

$$c = C_t A(n) N(n) \quad (4.1)$$

where $A(n)$ corresponds to the surface area of the subtree root, $N(n)$ is the number of triangles contained in the subtree and C_t corresponds to the relative cost for performing a ray-triangle intersection, and should be the same constant that was used to calculate the SAH cost (Equation 2.1). If the cost of the collapsed subtree is smaller, then the entire subtree is replaced by a leaf which references all its triangles.

Differently from the collapsing method used by Karras and Aila [25], the cost of the collapsed subtree does not have to be considered during the treelet restructuring process, since the only criterion used to determine which nodes will be merged is the surface area.

4.1.7 Parameters

There are three parameters that have to be set for our method: treelet size, number of iterations and γ . Treelet size corresponds to the number of leaf nodes of a treelet, and larger treelets generally produce better structures, at the expense of greater build times. Number of iterations corresponds to the number of times that a full bottom-up sweep of the tree is executed, with treelets being assembled and restructured at each step. Parameter γ determines how many leaves a node must have as descendants so that it can be used as a treelet root, and it is used to balance execution time and treelet quality, with higher values favoring build speed and lower values favoring quality.

4.2 Implementation and Optimizations

Our algorithm was implemented using CUDA and was designed to take advantage of the NVIDIA Kepler architecture. Being inspired mostly on Karras and Aila's [25] original work on BVH optimization via treelet restructuring, we used an agglomerative clustering method for recreating treelet topologies, with the objective of investigating

how larger treelet sizes affect the resulting BVHs when an approximate method is used to find the optimal solution. By using slightly larger treelets than the original method, we were able to produce BVHs with comparable quality in considerable less time, and also to drastically reduce the amount of temporary memory used.

The CUDA implementation makes heavy use of the shuffle intrinsic operations, which enables registers from other threads in the same warp to be read at a very low cost. Using those instructions, values that are accessed often within a warp can be cached in registers, both providing a fast access to those values and saving shared memory that could otherwise be used to store them. The shuffle intrinsics are also used to efficiently implement reductions, for example when comparing clusters to find which pair is closer to each other.

During the implementation stage, we tried to reach 100% occupancy while keeping register spills to local memory to a minimum. However, our design decision to rely heavily on shuffle operations requires some values to be stored throughout the whole treelet restructuring phase, leading to a high register usage. Therefore, we ended up opting for a 75% occupancy, which seems to be the optimal value for our implementation, leaving 40 registers available to each thread and 1024B of shared memory available to each warp when using the Kepler architecture.

4.2.1 Tree Traversal

For the bottom-up traversal of the tree, we use the method described by Karras [24]. Each thread starts by processing a leaf node; after it is done, it goes on to process the parent of that node. Since we are dealing with binary trees, at each iteration two threads will reach each node. By making the first thread inactive and allowing only the second one to continue, a node is guaranteed to be processed after both its children, thus avoiding race conditions. The pseudocode is shown in Algorithm 2.

4.2.2 Treelet Restructuring

During the bottom-up traversal, the number of active threads per warp is quickly reduced. In order to keep a high level of parallelism, each treelet restructuring task

Algorithm 2: TreeTraversal

```

1 for leaf  $i$  in  $BVH$  do in parallel
2    $current \leftarrow i$ 
3   while  $current \neq null$  do
4      $counter \leftarrow atomicInc(counters[current])$ 
5     if  $counter = 0$  then
6        $return$ 
7     end
8      $ProcessNode()$ 
9      $current \leftarrow parents[current]$ 
10  end
11 end

```

is distributed among all threads from the corresponding warp, even those that were inactive during the traversal. When alternating between bottom-up traversal and treelet restructuring, we use the `--ballot()`¹ voting intrinsic to find out which threads are active and referencing a valid treelet root. The returned values are analyzed, and each warp sequentially processes its treelets.

Three arrays are allocated in shared memory to store each treelet: one for leaf indices, another for internal node indices and a third for leaf surface areas. Leaf surface areas are only used when the treelet is being formed, which means that array can safely be repurposed to store other values during the treelet reconstruction. The array of treelet leaves can also be reused during that stage: when optimizing a treelet, each thread in the warp is responsible for storing at registers the variables of a cluster, such as node index and bounding box, so those values can be read efficiently by other threads using the shuffle instructions; after reading those values, their sources are no longer required. Lastly, after a pair of clusters is merged, the registers from the thread corresponding to the last cluster themselves can be used to store other values as well. Combining the free registers and shared memory, we can create an implicit list to hold the modifications required to restructure the tree without the need of allocating extra memory. When two clusters are merged, the last unused internal node is chosen to be modified and unite the nodes corresponding to each cluster, and the parent, left and right indices, together with the new bounding box and SAH cost are stored in that

¹<http://docs.nvidia.com/cuda/cuda-c-programming-guide>

implicit list.

If the SAH cost of the topology generated using the agglomerative clustering method is less than that of the original treelet, we proceed to commit the modifications found in the implicit list. This operation can be executed completely in parallel, with each thread processing one treelet node.

4.2.3 Distance Matrix

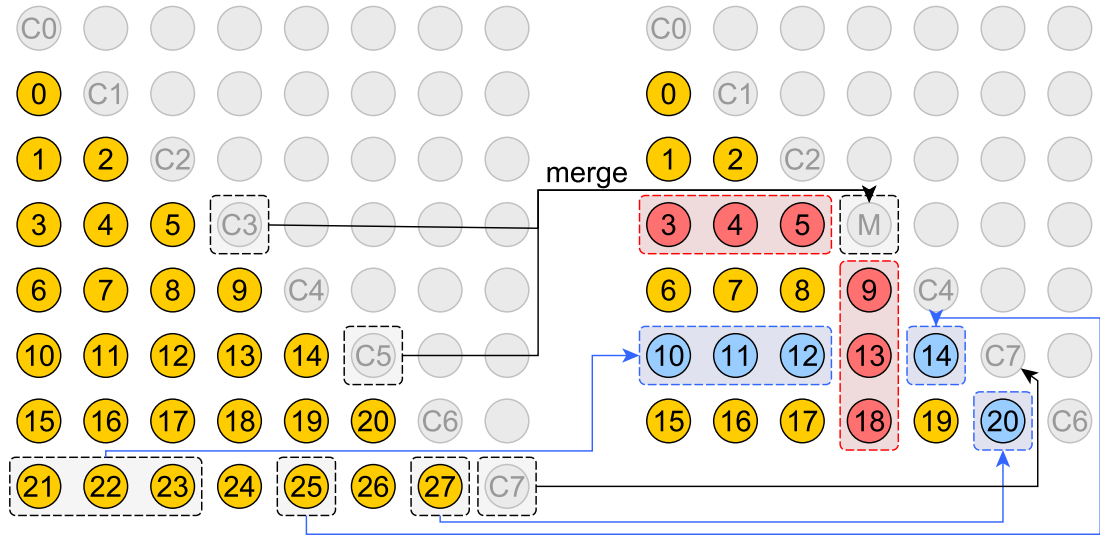


Figure 4.2: Updating the distance matrix after merging clusters three and five. The numbers inside elements below the main diagonal indicate the indices of those elements in the linear representation of the distance matrix. Clusters are represented by the main diagonal elements, and prior to the merge eight of them existed. The newly formed cluster M replaces cluster $C3$, while the last cluster is moved in the position of cluster $C5$. Elements highlighted in red represent the distance between M and all the other clusters and need to be recalculated. Elements highlighted in blue are copied from the last row.

To avoid redundant computations of distances between clusters, those distances are pre-calculated and cached in a triangular matrix just before the treelet restructuring. Instead of keeping the entire matrix in memory, we only store the elements that are below the main diagonal. These elements can be represented sequentially in an array, and the corresponding matrix row and column numbers can be calculated from that

array index as follows

$$r = 1 + \left\lfloor \frac{\sqrt{8i+1}-1}{2} \right\rfloor \quad (4.2)$$

$$c = i - \frac{r(r-1)}{2} \quad (4.3)$$

where r is the row number, c is the column number and i is the array index. We chose to use a lower triangular matrix, since its coordinates are slightly cheaper to extract from the array index than those of an upper triangular matrix.

In our implementation, the best performance is achieved by storing the distance matrix for treelets of sizes up to 20 in shared memory, since that can be done while maintaining an occupancy of 75%. For larger treelets, the distance matrix is stored entirely in global memory. The construction of the initial distance matrix is performed with the help of a precomputed schedule, specifying what pairs of clusters will be analyzed by each thread. Distances are calculated in the order that they appear in the distance array, so as to keep memory accesses coalesced for matrices that are stored in global memory. The use of a schedule is not required, since the indices for cluster pairs can be extracted by using Equations 4.2 and 4.3 for each array index, but it results in a minor performance increase.

After merging clusters i and j , the newly formed cluster will take i 's place, and the elements corresponding to the distance between it and all other clusters will have to be recalculated. This operation can be completely parallelized, with each thread calculating the distance between a cluster pair. The last cluster from the distance matrix will have to be moved to the j -th position as well. This can also be executed in parallel, with each thread moving an element. The process of updating the distance matrix is illustrated in Figure 4.2.

4.2.4 Global Memory Usage

The BVH used in our implementation requires 52 bytes of global memory per node: 5 scalars totaling 20B (parent, left and right pointers, data index and surface area) and one 32B bounding box (minimum and maximum values stored as float4). Aside from

the tree structure, we also store in global memory the atomic counters used during tree traversal (4B per node), the number of triangles descending from each node (4B per node), the SAH cost of each node (4B per node), the agglomerative schedule (256B for a treelet size of 32) and the distance matrix, if a treelet size of 21 or more is used (number of cluster combinations $\times$ 4 bytes per warp launched).

Chapter 5

Results

Given that our objective was to modify and improve on Karras and Aila’s [25] original work (TRBVH), we tried to replicate their experiments as closely as possible. In order to measure the performance of BVHs, we used Aila’s et al. [2, 3] ray tracing framework. Our tests were run on 16 different static scenes, most of which are commonly used in the area; their names, number of triangles and screenshots can be found in Table 5.1.

The structures that served as base for the optimization step were built using the LBVH algorithm [24, 30], as it is able to produce usable BVHs very quickly. All tested trees were collapsed using a post-processing kernel so as to further reduce their SAH cost.

Since a public implementation of TRBVH is not available, we implemented it ourselves by following the description available in the original paper [25]. The BVH build times achieved by our version are about 3 times higher than those reported by the authors, and that discrepancy cannot be explained by differences in the hardware used alone. However, we believe the comparisons made here are fair, seeing that both tested implementations share a large similarity and were created by the same authors. It is very likely that additional optimizations applied to one method will also benefit the other. The results reported here do not take triangle splitting into account for any method.

In our implementation of TRBVH, some values that were reported by Karras and Aila [25] as being stored in registers, such as SAH costs, triangle counts and

bounding boxes, had to be stored in global memory due to a lack of registers available. Because of that, it is safe to assume that our implementation of TRBVH is not using an optimal amount of temporary memory. The listed temporary memory values are implementation-dependent, and were only reported here to provide clarity and reproducibility.

The temporary memory values reported for TRBVH, ATRBVH and ATRBVH* (ATRBVH using the same parameter values as TRBVH) include the amount of memory required by LBVH. Since the tree construction and optimization are performed separately, these values can be further optimized by allocating a single block of memory large enough for both steps and reusing it.

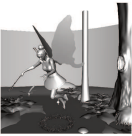
The results presented here were obtained on a PC with a Intel Core i5 4670 CPU, 12GB of RAM and a NVIDIA GTX 770 GPU. The implementation used in this research has been made publicly available¹.

5.1 Parameter Choice

In order to find the optimal treelet size and number of iterations, we tested all possible combinations of values, with treelet sizes ranging from 3 to 32 and number of iterations ranging from 1 to 5. We noticed that while increasing both parameters led to the generation of better performing trees, the performance gains that could be obtained in relation to TRBVH were very modest. Therefore, we opted to choose values which would minimize the build time, while keeping the performance within 0.5% of TRBVH. Using this heuristic, we determined a treelet size of 9 and a number of iterations of 2 to be the optimal values.

When evaluating the SAH cost of treelets or BVHs, we consider the relative costs C_i and C_t to be 1.2 and 1.0, respectively. As for γ , we started its value as the treelet size and doubled it at each iteration.

¹<https://github.com/leonardo-domingues/atrbvh>

 Arabic (412K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	41.73	10.61	127.15	11	65.27
	TRBVH	63.93	56.75	77.89	73	100.00
	ATRBVH	61.50	38.44	77.45	17	96.20
	ATRBVH*	62.57	43.23	77.44	17	97.87
 Armadillo (346K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	110.15	10.01	41.90	9	91.66
	TRBVH	120.17	43.87	35.51	62	100.00
	ATRBVH	118.64	31.82	35.93	15	98.73
	ATRBVH*	120.02	35.34	36.19	15	99.88
 Babylonian (499K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	55.83	12.19	104.23	13	60.20
	TRBVH	92.74	69.21	55.13	90	100.00
	ATRBVH	91.26	46.84	56.65	21	98.40
	ATRBVH*	95.33	52.88	55.14	21	102.79
 Buddha (1.1M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	75.31	23.53	89.14	29	85.11
	TRBVH	88.49	134.64	70.38	196	100.00
	ATRBVH	87.67	90.33	70.73	46	99.07
	ATRBVH*	87.69	104.98	71.31	46	99.10
 Bunny (69K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	169.19	3.79	44.89	2	89.09
	TRBVH	189.90	12.42	39.13	13	100.00
	ATRBVH	188.98	8.86	39.56	3	99.52
	ATRBVH*	188.56	9.57	39.72	3	99.29
 Conference (282K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	98.14	8.59	65.53	8	71.45
	TRBVH	137.36	39.25	39.53	51	100.00
	ATRBVH	139.67	27.76	38.93	12	101.68
	ATRBVH*	142.63	30.16	38.56	12	103.84
 Dragon (870K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	84.13	19.64	75.50	23	89.19
	TRBVH	94.33	105.61	62.04	157	100.00
	ATRBVH	94.23	72.21	62.10	37	99.89
	ATRBVH*	93.72	88.63	62.82	37	99.35
 Fairy (174K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	106.57	6.40	38.72	5	77.23
	TRBVH	137.99	27.11	33.21	31	100.00
	ATRBVH	135.72	19.03	33.84	7	98.35
	ATRBVH*	136.73	21.34	33.55	7	99.09

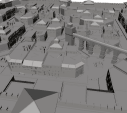
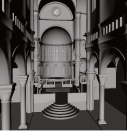
 Hairball (2.9M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	15.33	65.71	541.90	77	92.41
	TRBVH	16.59	374.22	478.08	520	100.00
	ATRBVH	16.44	255.24	475.72	121	99.10
	ATRBVH*	16.44	289.69	477.46	121	99.10
 Italian (368K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	53.31	10.58	121.79	10	61.23
	TRBVH	87.06	55.47	61.19	67	100.00
	ATRBVH	85.06	37.55	61.40	16	97.70
	ATRBVH*	84.49	42.37	61.02	16	97.05
 Sibenik (80K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	99.19	4.19	66.79	2	80.22
	TRBVH	123.65	14.77	51.85	14	100.00
	ATRBVH	122.67	10.31	52.19	3	99.21
	ATRBVH*	123.52	11.62	52.49	3	99.89
 Skeleton Hand (655K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	113.43	14.70	37.00	17	91.59
	TRBVH	123.84	79.89	30.72	118	100.00
	ATRBVH	123.68	55.45	31.09	27	99.87
	ATRBVH*	124.70	62.47	31.18	27	100.69
 Sponza (262K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	56.17	7.91	114.42	7	69.23
	TRBVH	81.13	37.86	75.75	50	100.00
	ATRBVH	85.99	26.43	75.42	12	105.99
	ATRBVH*	77.80	30.36	75.88	12	95.90
 Time Machine (4.7M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	9.43	95.29	308.61	125	86.91
	TRBVH	10.85	583.55	248.99	844	100.00
	ATRBVH	11.21	415.50	246.78	196	103.32
	ATRBVH*	11.53	448.57	248.08	196	106.27
 Turbine Blade (1.8M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	123.45	36.56	91.41	47	93.03
	TRBVH	132.70	213.27	78.99	319	100.00
	ATRBVH	131.16	151.76	79.79	74	98.84
	ATRBVH*	131.70	171.43	79.87	74	99.25
 Welsh Dragon (2.2M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	58.59	40.92	90.11	59	89.23
	TRBVH	65.66	256.53	73.51	399	100.00
	ATRBVH	65.18	176.62	74.38	93	99.27
	ATRBVH*	64.96	200.09	74.52	93	98.93

Table 5.1: Test results for 16 scenes. The Relative column expresses the performance of each method relative to TRBVH. The reported times for all methods include the time required to create the original tree, optimize it when applicable and collapse the structure.

5.2 Ray Tracer Analysis

The test results are summarized in Table 5.1. For each scene, we obtained measures using LBVH, TRBVH and two versions of Agglomerative Treelet Restructuring Bounding Volume Hierarchy (ATRBVH): ATRBVH* is the method described here, but using the same parameter values as TRBVH (treelet size = 7, number of iterations = 3); ATRBVH corresponds to the same method, but using the optimized parameter values (treelet size = 9, number of iterations = 2). We recorded BVH build times, number of rays traced per second, SAH cost and temporary memory required for each method. Memory allocation times are not reported.

The reported results are the average values obtained from ray tracing each scene through multiple viewpoints to better capture their details. We used five viewpoints for each scene, except for Italian, Babylonian and Arabic, where 10 viewpoints were employed to better represent the large areas of those scenes. Performance measurements were obtained by tracing diffuse rays, since those are less dependent on camera position than primary and ambient occlusion rays.

Considering the average results among all scenes, available in Table 5.2, ATRBVH was able to produce BVHs 30.5% faster than TRBVH, while keeping the ray tracing performance of those trees on par with the ones produced by the latter. This represents a large speedup over a method that already pushes current GPUs to their limit. When analyzing results for individual scenes, we can see that the variations in performance are very subtle, with no method being considerably better than the other in any given scene.

Method	Performance (%)	Time (%)
LBVH	80.8	20.3
TRBVH	100	100
ATRBVH	99.7	69.5
ATRBVH*	99.9	78.4

Table 5.2: Average among all scenes for the relative performance and build time of each method compared to TRBVH.

It is important to notice that when averaging the results among all scenes, even

though TRBVH considers all possible treelet topologies, by greedily choosing one treelet structure, ATRBVH* manages to achieve virtually the same ray tracing performance. This reinforces the capability of agglomerative bottom-up build methods to produce high quality trees.

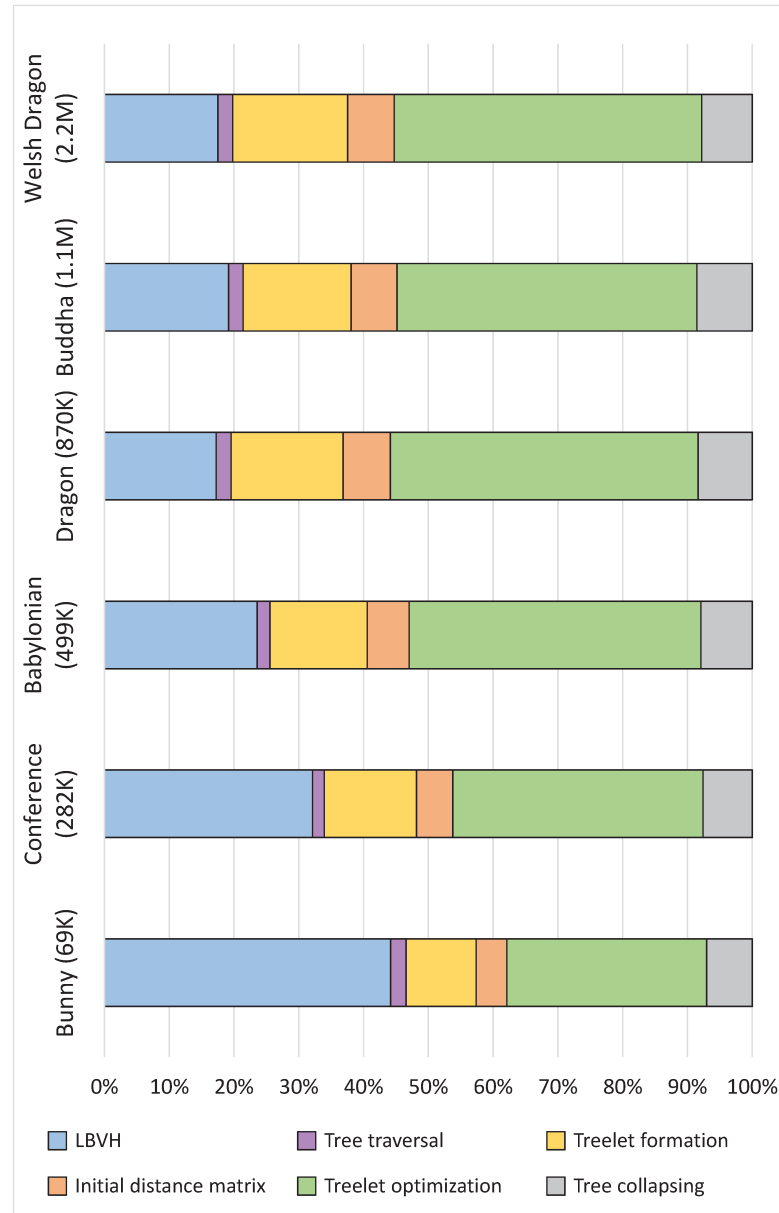


Figure 5.1: Percentage of time spent on each step of ATRBVH for six test scenes. These results were obtained by using a treelet size of 9 and a number of iterations of 2.

Figure 5.1 shows a breakdown of the total time required to construct BVHs using ATRBVH for six test scenes. The code for all steps but the initial distance matrix

construction and treelet optimization is the same as the one used for TRBVH in our tests.

Chapter 6

Conclusions and Future Work

During the course of this research, we investigated the effect that acceleration structures have in ray tracing performance. In particular, we focused on different techniques for constructing and optimizing BVHs on GPU. We realized that the main challenge in generating the optimal structure is to maintain a balance between quality and construction speed, and that the scale moves one way or another, depending on the kind of application that will use those structures.

Our contribution is ATRBVH, a method for optimizing existing BVHs which extends the current state-of-the-art in GPU BVH optimization, TRBVH. Instead of considering all possible node combinations when restructuring a treelet, our method makes greedy choices through agglomerative clustering. It can be efficiently implemented for the NVIDIA Kepler architecture using CUDA and is able to construct a high-quality tree from the ground up in a matter of milliseconds.

We have shown that our method produces structures that provide virtually the same ray tracing performance as TRBVH, while spending about 30% less time and requiring just a fraction of its temporary memory to do so. It is our understanding that any application that currently uses TRBVH can benefit from our algorithm. We have also made our implementation of both TRBVH and ATRBVH available as open-source, so other authors can easily compare against them.

For future work, we intend to investigate how treelet sizes greater than 32 affect the quality of produced BVHs. Another possibility would be to dynamically adjust

the treelet size used, provided that we can estimate how much the SAH cost can be reduced by adding a node to a treelet prior to restructuring it. It also remains to be seen how our method behaves when triangle splitting is used and how to adapt it to handle animated scenes. If the process of committing a set of modifications to a treelet could be performed atomically, the tree would be consistent at all points during the optimization, and the effect of setting a time budget and only stopping the optimization after that time has run out could be investigated. Lastly, a more thorough analysis of the scenes used could be performed to identify which characteristics affect the optimization, such as vertex degree and triangle sizes.

Bibliography

- [1] T. Aila, T. Karras, and S. Laine. On quality metrics of bounding volume hierarchies. In *Proceedings of the High-Performance Graphics Conference*, pages 101–107, Anaheim, California, USA, 2013. ACM.
- [2] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the High-Performance Graphics Conference*, pages 145–149, New Orleans, Louisiana, USA, 2009. ACM.
- [3] T. Aila, S. Laine, and T. Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.
- [4] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, pages 3–10, 1987.
- [5] C. Apetrei. Fast and simple agglomerative LBVH construction. In *Computer Graphics and Visual Computing*. The Eurographics Association, 2014.
- [6] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference*, pages 37–45, Atlantic City, New Jersey, USA, 1968. ACM.
- [7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.
- [8] J. Bittner, M. Hapala, and V. Havran. Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum*, 32(1):85–100, Feb. 2013.
- [9] J. Bittner, M. Hapala, and V. Havran. Incremental BVH construction for ray tracing. *Computers & Graphics*, 47:135 – 144, Apr. 2015.
- [10] B. Chang, W. Seo, and I. Ihm. On the efficient implementation of a real-time kd-tree construction algorithm. In *GPU Computing and Applications*, pages 207–219. Springer Singapore, 2015.

- [11] B. Choi, B. Chang, and I. Ihm. Improving memory space efficiency of kd-tree for real-time ray tracing. *Computer Graphics Forum*, 32(7):335–344, 2013.
- [12] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, 18(3):137–145, Jan. 1984.
- [13] L. R. Domingues and H. Pedrini. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the High-Performance Graphics Conference*, Los Angeles, California, USA, 2015. ACM.
- [14] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster HLBVH with work queues. In *Proceedings of the High-Performance Graphics Conference*, pages 59–64, Vancouver, British Columbia, Canada, 2011. ACM.
- [15] A. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24, Oct 1984.
- [16] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.
- [17] Y. Gu, Y. He, K. Fatahalian, and G. Bluelloch. Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the High-Performance Graphics Conference*, pages 81–88, Anaheim, California, USA, 2013. ACM.
- [18] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 113–118. IEEE Computer Society, 2007.
- [19] M. Hapala and V. Havran. Review: Kd-tree traversal algorithms for ray tracing. *Computer Graphics Forum*, 30(1):199–213, 2011.
- [20] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
- [21] V. Havran, R. Herzog, and H.-P. Seidel. On the fast construction of spatial hierarchies for ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 71–80. IEEE Computer Society, Sept. 2006.
- [22] T. Ize, I. Wald, and S. Parker. Ray tracing with the BSP tree. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 159–166, Aug 2008.
- [23] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.

- [24] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the High-Performance Graphics Conference*, pages 33–37, Paris, France, 2012. Eurographics Association.
- [25] T. Karras and T. Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the High-Performance Graphics Conference*, pages 89–99, Anaheim, California, USA, 2013. ACM.
- [26] T. Karras and T. Aila. Agglomerative treelet restructuring for bounding volume hierarchies. U.S. Patent Applications Publication No. US20140365529 A1, filed 10/28/2013, 2014.
- [27] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 269–278. ACM, 1986.
- [28] A. Kensler. Tree rotations for improving bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 73–76. IEEE Computer Society, Aug. 2008.
- [29] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques*, pages 145–153, 1993.
- [30] C. Lauterbach, M. Garland, S. Sengupta, D. P. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, Apr. 2009.
- [31] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics*, 6(3):153–166, May 1990.
- [32] NVIDIA. NVIDIA’s next generation CUDA compute architecture: Kepler GK110/210. Technical report, 2014.
- [33] NVIDIA. CUDA C Best Practices Guide. <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>, 2015. [Online; accessed 20-April-2015].
- [34] NVIDIA. CUDA C Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>, 2015. [Online; accessed 20-April-2015].
- [35] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the High-Performance Graphics Conference*, pages 87–95, Saarbrücken, Germany, 2010. Eurographics Association.

- [36] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Proceedings of the Seventh Annual Conference on Computer Graphics and Interactive Techniques*, 14(3):110–116, July 1980.
- [37] A. Santos, J. Teixeira, T. Farias, V. Teichrieb, and J. Kelner. Understanding the efficiency of kd-tree ray-traversal techniques over a GPGPU architecture. *International Journal of Parallel Programming*, 40(3):331–352, 2012.
- [38] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the High-Performance Graphics Conference*, pages 7–13, New Orleans, Louisiana, USA, 2009. ACM.
- [39] 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, 1997.
- [40] M. Vinkler, V. Havran, and J. Bittner. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In *Proceedings of the 30th Spring Conference on Computer Graphics*, pages 29–36, Smolenice, Slovakia, 2014. ACM.
- [41] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.
- [42] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 33–40. IEEE Computer Society, 2007.
- [43] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transaction on Graphics*, 26(1):1–18, Jan. 2007.
- [44] I. Wald, T. Ize, and S. G. Parker. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics*, 32(1):3 – 13, Feb. 2008.
- [45] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast agglomerative clustering for rendering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 81–86. IEEE Computer Society, Aug 2008.
- [46] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, Jan. 1984.
- [47] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

- [48] Z. Wu, F. Zhao, and X. Liu. SAH KD-tree construction on GPU. In *Proceedings of the High-Performance Graphics Conference*, pages 71–78, Vancouver, British Columbia, Canada, 2011. ACM.

Appendix A

Other Approaches

In this appendix, we describe two other approaches to optimizing BVHs on GPU that were tested during this research. Although both methods managed to improve the quality of trees created using LBVH, their results were not good enough to justify their usage over TRBVH.

A.1 Tree Rotation

Our first attempt at optimizing BVHs was to extend Kenler’s work on tree rotations [28] by implementing it on GPU using CUDA. In his paper, Kensler proposed performing operations similar to the tree rotations used to balance red-black trees in order to minimize the SAH cost of a tree. By recursively traversing the tree, his method checks which is the best of four possible rotations that can be applied to that node in order to minimize its local cost.

Instead of traversing the tree recursively, we opted to use the bottom-up traversal algorithm proposed by Karras [24]. Each thread would be responsible for checking one node to determine its best rotation, and then applying the necessary modifications. Since by using this bottom-up traversal one node is only processed after both its children, it is guaranteed that no race conditions will occur.

Simply iterating through the tree and performing rotation on its nodes is not enough to produce trees that have quality competitive with TRBVH. Kensler achieved his best results by using Simulated Annealing optimization to avoid local minima.

This technique, however, requires a large number of iterations to converge, and as such is not ideal for the fast optimization of trees that was our goal.

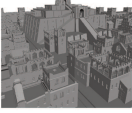
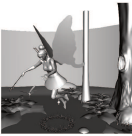
In order to try to minimize the number of iterations required by Simulated Annealing, we experimented with a simpler algorithm to introduce perturbations during the search for the optimal tree: when evaluating a node, instead of always greedily executing the rotation that minimizes the local SAH cost, we introduced a small chance that a rotation would be chosen at random for that node. This modification was sufficient to cause the SAH cost to decrease significantly when compared to greedily choosing the best rotation, however it still required a large number of iterations to converge, resulting in large optimization times.

A.1.1 Parameters

During the evaluation of each node, the probability of choosing a random rotation instead of the one that minimized the local SAH cost was 0.5%. To produce the results presented here, the algorithm was repeated for a total of 1000 iterations.

A.2 Divisive Treelet Restructuring

Karras and Aila [25] indicated in the future work section of their paper that it would be possible to use approximate methods to restructure larger treelets than those used in TRBVH, in order to perform more extensive modifications to treelets. Inspired by this idea, we implemented a version of their method that uses a greedy top-down optimization instead of extensive searching through all possible node combinations to determine the best treelet structure.

 Arabic (412K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	41.73	10.61	127.15	11	65.27
	TRBVH	63.93	56.75	77.89	73	100.00
	Rotation	60.33	2755.25	83.90	29	94.37
	DTRBVH	48.07	191.09	100.88	115	75.19
 Armadillo (346K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	110.15	10.01	41.90	9	91.66
	TRBVH	120.17	43.87	35.51	62	100.00
	Rotation	116.12	2122.67	37.55	25	96.63
	DTRBVH	116.56	144.20	38.41	98	97.00
 Babylonian (499K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	55.83	12.19	104.23	13	60.20
	TRBVH	92.74	69.21	55.13	90	100.00
	Rotation	83.39	3379.41	62.07	36	89.92
	DTRBVH	63.33	242.66	83.07	141	68.29
 Buddha (1.1M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	75.31	23.53	89.14	29	85.11
	TRBVH	88.49	134.64	70.38	196	100.00
	Rotation	83.80	6996.25	74.88	79	94.70
	DTRBVH	74.57	510.70	87.70	307	84.27
 Bunny (69K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	169.19	3.79	44.89	2	89.09
	TRBVH	189.90	12.42	39.13	13	100.00
	Rotation	183.03	476.03	40.94	5	96.38
	DTRBVH	170.46	37.23	43.94	20	89.76
 Conference (282K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	98.14	8.59	65.53	8	71.45
	TRBVH	137.36	39.25	39.53	51	100.00
	Rotation	128.97	1870.10	43.39	20	93.89
	DTRBVH	123.18	118.02	47.47	80	89.68
 Dragon (870K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	84.13	19.64	75.50	23	89.19
	TRBVH	94.33	105.61	62.04	157	100.00
	Rotation	88.71	5553.61	65.38	63	94.04
	DTRBVH	87.15	386.72	70.41	246	92.39
 Fairy (174K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	106.57	6.40	38.72	5	77.23
	TRBVH	137.99	27.11	33.21	31	100.00
	Rotation	129.95	1204.35	34.14	13	94.17
	DTRBVH	121.49	83.87	34.85	49	88.04

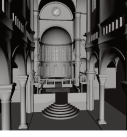
 Hairball (2.9M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	15.33	65.71	541.90	77	92.41
	TRBVH	16.59	374.22	478.08	520	100.00
	Rotation	15.93	17933.43	491.60	209	96.02
	DTRBVH	15.47	1288.38	514.86	813	93.25
 Italian (368K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	53.31	10.58	121.79	10	61.23
	TRBVH	87.06	55.47	61.19	67	100.00
	Rotation	81.13	2652.58	66.93	27	93.19
	DTRBVH	58.52	192.29	93.81	106	67.22
 Sibenik (80K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	99.19	4.19	66.79	2	80.22
	TRBVH	123.65	14.77	51.85	14	100.00
	Rotation	115.09	620.31	55.10	6	93.08
	DTRBVH	107.35	44.10	60.49	23	86.82
 Skeleton Hand (655K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	113.43	14.70	37.00	17	91.59
	TRBVH	123.84	79.89	30.72	118	100.00
	Rotation	117.42	4000.86	32.69	47	94.82
	DTRBVH	115.69	273.73	34.71	185	93.42
 Sponza (262K)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	56.17	7.91	114.42	7	69.23
	TRBVH	81.13	37.86	75.75	50	100.00
	Rotation	70.11	1808.60	79.31	20	86.42
	DTRBVH	61.45	122.70	88.56	79	75.74
 Time Machine (4.7M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	9.43	95.29	308.61	125	86.91
	TRBVH	10.85	583.55	248.99	844	100.00
	Rotation	10.68	30494.63	258.62	339	98.43
	DTRBVH	9.37	2617.31	298.66	1320	86.36
 Turbine Blade (1.8M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	123.45	36.56	91.41	47	93.03
	TRBVH	132.70	213.27	78.99	319	100.00
	Rotation	127.45	10473.06	82.81	128	96.04
	DTRBVH	126.48	671.29	84.63	498	95.31
 Welsh Dragon (2.2M)	Method	Mrays/s	Time (ms)	SAH	Memory (MB)	Relative (%)
	LBVH	58.59	40.92	90.11	59	89.23
	TRBVH	65.66	256.53	73.51	399	100.00
	Rotation	62.90	13263.17	77.16	160	95.80
	DTRBVH	62.79	804.13	77.77	624	95.63

Table A.1: Test results for DTRBVH and Tree Rotation. The Relative column expresses the performance of each method relative to TRBVH. The reported times for all methods include the time required to create the original tree, optimize it when applicable and collapse the structure.

To optimize the treelets, we used the binned SAH strategy, which is commonly used to construct BVHs on CPU [42]. It consists in discretizing the space into bins and iteratively distributing the tree nodes among those bins. After all nodes have been assigned, the bounds of the bins are used as split planes, and the split that minimizes the tree SAH cost is chosen. The algorithm is recursively repeated for each of the two subsets formed that have at least two nodes.

Another strategy that was tested to restructure treelets was to always split triangles in the largest dimension of their bounding boxes. The triangles would be sorted along that dimension using insertion sort, and the boundary between each triangle would be used as a potential split plane.

Although this method was able to improve the quality of the processed tree, it was not capable of reaching results comparable with TRBVH, and the required optimization time was much higher. We attribute the high optimization time to the lack of sufficient work to keep the GPU busy. This happens mostly because this heuristic is inherently sequential in nature, with each step only having a small amount of work that can be parallelized.

A.2.1 Parameters

We used the same parameters that were used on TRBVH, that is, a treelet size of 7 and a number of iterations of 3. The γ value was not updated, resulting in all tree nodes being evaluated during each iteration.

A.3 Results

The test results are summarized in Table A.1. Although both methods managed to improve the quality of trees constructed using LBVH, the performance increase was consistently lower than that produced by TRBVH. Tree Rotation in particular obtained results that were close to those of TRBVH in quality, but never superior. Divisive Treelet Restructuring Bounding Volume Hierarchy (DTRBVH), on the other hand, only increased the quality by a modest amount. The memory consumption of

DTRBVH was also quite higher than that of TRBVH, making it unpractical for larger scenes.

Optimizing BVHs using these algorithms took significantly longer than doing so using TRBVH. Due to the many iterations that are necessary for it to converge, Tree Rotation requires seconds of optimization, which is completely unacceptable considering that it runs on a GPU. It is important to notice that neither method was fully optimized for speed, since the quality of trees produced would not justify using them anyway. Even though the speed of these methods can be improved by a more careful optimization, we believe that those improvements would still not be enough to make them competitive with the alternatives.