Bandwidth and Memory Efficiency in Real-Time Ray Tracing

Pedro Filipe Vitorino Castro Lousada
pedro.lousada@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2016

Abstract

Real time ray tracing has been given a lot of attention in recent years in the academic and research community. Several novel algorithm have appeared that parallelize different aspects of the ray tracing algorithm through the use of a GPU. Among theses, the creation of BVHs. We believe that recent approaches have failed to consider the performance impact of memory accesses in GPU and how their cost affects the overall performance of the application. In this work we show that by reducing memory bandwidth and footprint we are able to achieve significant improvements in BVH traversal times. We do this by compressing the BVH and the triangle mesh a in parallel manner after its creation in each frame and then decompressing then as needed while traversing the BVH.
Keywords: *Ray Tracing*, *Bounding Volume Hierarchy*, *Parallelization*, *GPU*, *Quantization*, *Memory*

1. Introduction

Real-time rendering concerns itself with the generation of synthetic images at a rate fast enough so that the viewer can interact with a virtual environment. An image appears on screen, the viewer acts or reacts, and this feedback affects what is generated next. Two of the more popular approaches to synthetic image generation are Rasterization and Ray-Tracing. Both have been used in computer graphics for the past decades. Each method allow us to generate 2D images from 3D scenes composed of virtual objects.

Ray-Tracing received little attention outside the academic world mainly due to its high computation costs made it a much more expensive and slow approach compared to rasterization. Ray tracing offers a fairly long list of advantages over rasterization. Ray-tracing can easily simulate non-local effects such as shadows, reflections and refraction. In Rasterization reflections and shadows are hard to compute; refractions being very hard.

Due to its mathematical correctness, ray tracing can make generated images look more realistic. It's only a matter of being able to generate them at a rate fast enough required by the application.

1.1. Problem Description

At its core, the ray tracing algorithm follows the following logic: for each pixel of the display, cast a ray that propagates in a straight line until it intersects an element of the scene being rendered. The intersection is then used to determine the color of the pixel as a function of the intersected elements surface. For a simple scene with no secondary rays (i.e. reflections, shadows, refractions, etc) this mean having at least NxM intersection tests (N being the number of rays and M the number of polygons in the scene). For its nature, ray tracing inherently leads to a high number of expensive floating point operations. Put this together with an irregular and high bandwidth memory access pattern and performance will be an issue when trying to achieve real time rendering.

One way to optimize the standard ray tracing algorithm is through the use of acceleration structures. Acceleration structures allow us to lower the number of intersections tests needed to render an image. Currently Bounding Volume Hierarchies show the most promise. The state of the art in this kind of structure has been targeted towards generating the structure at a rate fast enough to be usable in a real time application, typically creating a new BVH or adapting an existing one at each frame.

Most algorithms use a GPU to achieve the performance they need. Despite their capability of performing a high number of floating point operations per second, GPUs tend to suffer from slow memory accesses. Most algorithms that focus on constructing a BVH in parallel manner tend to overlook this and be careless with memory accesses and bandwidth, resulting in overall poorer performance than possible.

1.2. Contibutions

We believe that by concerning ourselves with GPU memory footprint and bandwidth efficiency we are able to improve performances. In this work we aim to develop a ray tracing application supporting one

of the latest state of the art algorithms in parallel BVH construction and improve it further through memory compression techniques.

Our contribution is a novel algorithm based in existing techniques for real time BVH construction with focus and improvements in BVH and triangle mesh compression. We are able to reduce the total size of memory used to store both the BVH and the triangle mesh as well as reduce the memory bandwidth of the application. We see improvements of up to 40% in occupied memory and reductions of up to 20% in memory bandwidth.

2. Bakcground and Related Work

2.1. Ray Tracing

First introduced by T. Whitted [11], ray tracing is a technique based in global illumination shading. It employs the use of ray casting [1] to intersect eye rays with objects in a computer simulated scene. Eye rays which intersect ibjects then lead to the creation of extra secondary rays: e.g. shadow, reflection and refraction rays.

The main difference between rasterization and ray tracing is the ability to calculate the graphical models of shadows, reflections and refractions. Whereas rasterization engines only try to simulate these effects through the use of texturing, ray tracing takes a more mathematical approach. Since these effects are highly geometry dependent, simulated methods look rather unrealistic when changing object position or viewer position. The difference in ability to simulate secondary visual effects between rasterization and ray tracing can be seen in Figure 1.

Figure 1: Difference between rasterization and ray tracing (source: Intel)

2.2. Bounding Volume Hierarchies

One of the major problems of ray tracing is the sheer number of intersection tests one must perform in order to check if a certain ray intersection an object of the scene. Acceleration structures help us reduce the number of intersection to test by organizing the geometry of the scene into a data structure that can easily be explored. The organization of acceleration structure is typically hierarchical, loosely meaning that the topmost level encloses the levels bellow it, and so on.

In the current state of the art Bounding Volume Hierarchies show the most promisse [9][6][4]. Bounding Volume Hierarchies are a tree-like structure that subdivide a scene into smaller segments. A Bounding Volume Hierarchy divides a scene around its objects. All objects are wrapped into individual Bounding Volumes that form the leaf nodes of the tree. Bounding Volumes are then recursively enclosed together by more embracing Bounding Volumes until we are left with a single Bounding Volume that wraps around the entire scene.

In a typical object hierarchy data structure, it's easy to update the data structure as an object moves because the object lives in just one node and the bounds for that node can be updated with relatively simple and localized update operations. For deformable scenes, just re-fitting a Bounding Volume Hierarchy i.e., recomputing the hierarchy node's bounding volumes, but not changing the hierarchy itself is sufficient to produce a valid Bounding Volume Hierarchy for the new frame. Bounding Volume Hierarchies also allow for incremental changes to the hierarchy[10].

In Bounding Volume Hierarchies, primitives are only referenced exactly once, allowing us to save GPU memory and bandwidth. Empty cells that frequently occur in spatial subdivision but simply do not exist in object hierarchies. The effectiveness of a Bounding Volume Hierarchy depends on what actual hierarchy the build algorithm produces.

2.3. GPU Computing

Modern GPUs are what we call SIMD (Single Instruction Multiple Data) devices, meaning that they can perform the same operation on multiple data points simultaneously. Even though a modern CPU core can dispatch more operations per second than a GPU core, the GPU as a whole can vastly outperform a CPU by having thousands of cores running the same operations versus the 4-8 cores present in the CPU. The idea behind GPU accelerated computing is to use a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally intensive part is run by the GPU [3]. The massive parallelism

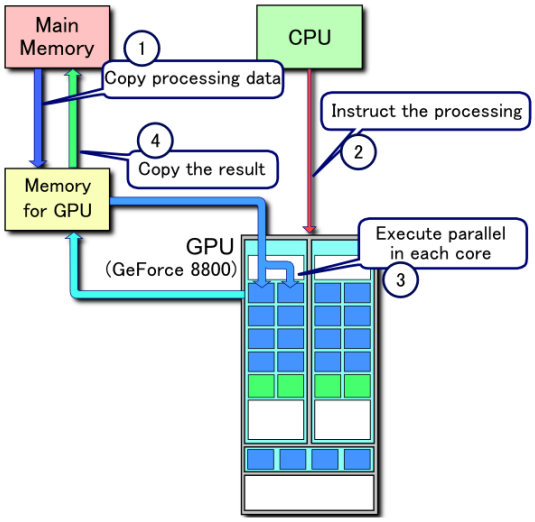

Figure 2: Example of CUDA processing flow (source: NVIDIA)

of programmable GPUs naturally lends itself to inherently parallel problems such as ray tracing

A GPU accelerated application consists of 2 main components: a host and a device. The host refers to the CPU and the systems memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernel functions which are executed in the device.

A typical sequence of operations for a CUDA program consist of (Figure 2):

1. Declare and allocate host and initialize the host's data
2. Transfer data from host memory to device memory
3. Execute kernel functions
4. Transfer results from device memory to host memory

A given kernel executes a single program across many parallel threads. Typically, each kernel completes execution before the next kernel begins, with an implicit barrier synchronization between kernels. GPUs have support for multiple, independent kernels to execute simultaneously, but many kernels are large enough to fill the entire device. Threads are decomposed into thread blocks; threads within a given block may efficiently synchronize with each and have shared access to per-block on-chip memory[6].

2.4. Parallel Construction of Bounding Volume Hierarchies

Generating an acceleration structure is a necessary step when trying to achieve real-time performances. One can take two approaches: To construct a new structure every frame or to reuse the same structure and adjust it at each frame. The latter option has been recently explored [5] but still seems to be lacking performance when processing large trees or large modifications are needed to the data structure.

Most approaches [2] that explore a construction of new hierarchy at each frame tended to rely on serial algorithms running on the CPU to construct the necessary hierarchical acceleration structures. While once a necessary restriction due to architectural limitations, modern GPUs provide all the facilities necessary to implement hierarchy construction directly. Doing so should provide a strong benefit, as building structures directly on the GPU avoids the need for the relatively expensive latency introduced by frequently copying data structures between CPU and GPU memory spaces.

The first to explore such method was Lauterbach et al.[6]. Lauterbach introduced a novel algorithm using spatial *Morton codes*[7] to reduce the construction of BVH a simple sorting problem. *Morton codes* are used to determine a primitives order along a space filling curve. They can be computed directly from a primitive's geometric coordinates. Lauterbach expects each input primitive to be represented by an AABB[1] and that the enclosing AABB of the entire input geometry is known. By taking the barycenter of each primitive's AABB as its representative point and by quantizing each of the 3 coordinates of the each representative points into k-bit integers, a $3k$-bit *Morton code* is constructed by interleaving the successive bits of these quantized coordinates. Figure 3 shows a 2D representation of this. Sorting the *Morton codes* will automatically lay the associated points in order along a *Morton curve*. It will also order the corresponding primitives in a spatially coherent way. Because of this, sorting geometric primitives according to their *Morton code* has been used to improve cache coherence as a ray that hits a certain primitive will likely hit the primitive adjacent to it.

The main problem of Lauterbach's work and others[2][8] is that in order to build the resulting BVH a series of sequential steps must be taken. Lauterbach et al create their BVH by sequentially observing each bit of each *Morton code* and grouping the primitives according to the value of the bit. At each level if said bit has value *0* then the primitive it is placed in a group, if the bit has value *1* then it is placed in the opposite group. Garanzha et al. takes a different approach by generate one level of nodes at a time, starting from the root. They process the nodes of the BVH on a given level in parallel. For this they use binary search to partition the primitives contained within each node. The

[1] AABB - Axis-aligned minimum bounding box

resulting child nodes are then enumerated using an atomic counter, and subsequently process them on the next round.

Karras[4] further developed this line of though by proving a way to build a BVH in a total parallel manner. Karras introduces an in-place algorithm for constructing a binary radix tree (also called a *Patricia* tree) which can directly be converted into a BVH.

He bases his approach on the fact that for a scene with N primitives we know we can make a *Patricia* tree with $N - 1$ internal nodes to represent it. He analyzes the similarities between each *Morton code* and its neighbours determine the position of each internal node in the Patricia Tree. The child-parent association is then calculated based on the range of *same-value-bits* with the rest of the *Morton codes*.

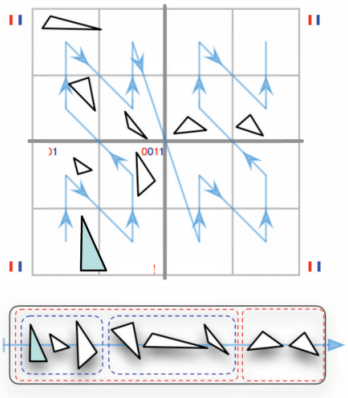

Figure 3: Example 2-D Morton code ordering of triangles with the first two levels of the hierarchy. Blue and red bits indicate x and y axes, respectively. Source: [6]

3. Approach and Architecture

In order to achieve a high frame rate one must reduce the time it takes to generate an image at each frame. For this we make use of a GPU to help us accelerate all the floating point operations required by the intersection process of a ray-tracer. We aim to reduce the memory bandwidth and memory footprint of our application. We make use of research made in the area of parallel BVH construction in GPU to naturally reduce the number of intersection tests performed (thus, reducing memory bandwidth as well). We then explore BVH and triangle meshes compression techniques. With a compressed BVH and triangle mesh we can expect our rendering kernels to be to achieve lesser rendering times as the data transfered between the GPU's global memory and the kernel's local memory will be smaller.

All of this, which define our proposed solution model, will be addressed in further detail in the following sections.

3.1. Binary Radix Tree Properties

Before we descreibe our algorithm there are a few Binary Radix Tree properties we should cover as these are important for our work. A radix tree is a space-optimized trie often used for indexing string data, although it can also be used to index any data divisible in smaller comparable chunks such as characters, binary numbers, etc. For simplicity, assume that from now on our radix tree only contains binary values and that they are in a lexicographical order as in our application each key will correspond to a sorted *Morton code*.

Given a set of keys $k_0, ..., k_{n-1}$ represented as bits, a radix tree can be seen as a hierarchical representation of the common bits of each key. The keys are represented in the leaf nodes of the tree, and each internal node corresponds to the longest common prefix shared between the keys in that subtree.

Assume the example referenced in Figure 4. As one would expect the root of the tree covers the full range of keys. At each level the keys are partitioned according to their first differing bit. The first difference occurs between keys k_3 and k_4, thus, the left child of the root node contains keys k_0 through k_3 and the right child contains k_4 through k_7. We continue this process to essentially get a hierarchical representation of the common prefixes between each key. At the bottom level of the tree we will find that each child references a key.

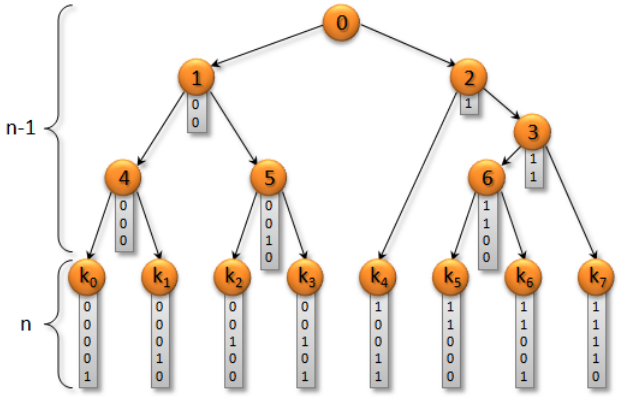

Figure 4: A visual representation of an ordered radix tree. The numbers 0-7 act as keys (and such as leaf nodes) of the tree. Each internal node represents a common range prefix of the binary value of all the leaf nodes bellow it. Notice we have $N - 1$ internal nodes for N keys.

A radix tree is considered a compact data structure as it omits nodes which only have one child, thus removing redundant information and decreasing the overall size of the tree in memory.

One property of a binary radix tree is that any

given tree with N keys will have $n-1$ internal nodes. This allows us to know even before we construct the tree how many nodes, and thus how many memory, we will require. Assuming that we have a lexicographically ordered tree allows us to express $[i, j]$ as the range of keys covered by any given internal node. We use $\delta(i, j)$ to denote the length of the longest common prefix between the keys k_i and k_j. The ordering of the keys automatically implies that $\delta(i', j') \geq \delta(i, j')$ for any $i', j' \in [i, j]$ [4]. We can then determine the common prefix shared between key under a given node by comparing the first and last key it covers - all the other keys in between are guaranteed to share the same or a larger prefix.

In practice each internal node partitions the keys under it on their first differing bit, the one following $\delta(i, j)$. We can safely assume that in the range $[k_i, k_j]$ part of the keys will have said bit set to 0 and others will have it set to 1. Since we are working with an ordered tree all of the keys with the bit set to 0 will be presented before the keys with the bit set to 1. We call the position of the last key where this bit has value 0 the split position, denoted by $\gamma \in [i, j-1]$. Since the split position is where the first bit differs between the keys in the range we can say that $\delta(\gamma, \gamma+1) = \delta(i, j)$. The ranges $[k_i, k_\gamma]$ and $[k_{\gamma+1}, k_j]$ will be subdivided at the next differing bit. We can thus say for sure is that $\delta(i, \gamma) > \delta(i, j)$ and $\delta(\gamma+1, j) > \delta(i, j)$. Taking Figure 4 once more as reference, we can see that the first differing bit happens between keys k_3 and k_4, the range is then split at $\gamma = 3$ resulting in the subranges $[k_0, k_3]$ and $[k_4, k_7]$. The left child then splits its range $[k_0, k_3]$ at the third bit, $\gamma = 1$. The right child splits the range $[k_4, k_7]$ at $\gamma = 4$, at the second bit, and so on.

3.2. Morton Codes

We start our process with a series of primitives represented by 3D points. In order to generate our BVH we first start by deciding in which order each leaf node will be represented in the tree. A good approach is to sort the leafs according to their position, generally we will want primitives close to each other in 3D space to appear close to each other on the tree. In order to do this we sort them via a space-filling curve, more specifically a Z-order curve. For this we take the centroid of each triangleand express it relative to the bounding volume of the entire scene, known after loading the scene's data in the Host code.

Let bvh_{min} be defined as the minimum value of the scene's bounding volume and bvh_{max} as its maximum. If we define c as the centroid point of a given triangle then we can express q, the same point but now in coordinates relative to the bounding volume of the scene through the following formula:

$$q = \frac{c - bvh_{min}}{bvh_{min} - bvh_{max}} \tag{1}$$

q's coordinate values now vary between 0 and 1. We can think of it as a point inside the 3D space delimited by the scene's bounding volume. The closer each coordinate is to 0 the closer the point will be to the minimum point of the bounding volume, and, the closer it is to 1 the closer it will be to its maximum point.

We now want to express this 3D point as a *Morton code*. The first step in this process is to transform our point from a continuous space into a discrete one. We achieve this by quantizing each floating point coordinate into a range created by the difference between the scene's bounding volume maximum and minimum extremes. *Morton codes* are best expressed as a single integer so to represent a 3D point in a single integer value we will have to make some precision sacrifices. We assume an architecture of 32 bit sized integers, meaning that in order to represent 3 distinct values in 32 bits we will have 10 bits for each of the 3 Cartesian coordinates. We are thus left with the following quantization equation:

$$q = \frac{(c - bvh_{min}) * 1024}{bvh_{min} - bvh_{max}} \tag{2}$$

Where q is now a 3D point who's coordinates are composed of 10 bit integers. To correctly represent this point along a Z-order curve we interleave the bits of all three coordinates together to form a single binary number. We take the value of each coordinate, expand the bits by inserting 2 bit "gaps" after each bit and then criss-cross them. Beyond this point it's simply a matter of calculation the *Morton codes* for each primitive and sort them via a parallel sorting algorithm, we used radix sort for this.

3.3. Parallel Construction of BVH

This section follow the same line of thought described by Karras in [4]. The idea is to assign indexes for the internal nodes in a way that enables finding their children without depending on earlier results, this way we can fully parallelize the construction of the BVH.

We create 2 separate arrays to store our nodes, $\mathbf{L}$ for the leaf nodes and $\mathbf{I}$ for the internal nodes. We define the layout of $\mathbf{I}$ as having the root node at the index 0, denoted by $\mathbf{I_0}$. The index of each internal node will be defined by its split position. For any given node the left child will be defined in $\mathbf{I}_\gamma$ if it covers more than one key and at $\mathbf{L}_\gamma$ if it doesn't. Similarly the right child will be located either at $\mathbf{I}_{\gamma+\mathbf{1}}$ or at $\mathbf{L}_{\gamma+\mathbf{1}}$. Assuming this layout has an important property, the index of every internal node will either coincide to the first or the last key

it covers. Take for example the root node, it covers the entire range of keys $[0, n-1]$ and is located at position $\mathbf{I_0}$. A node covers the range $[i, j]$; its left child will be located at the end of the range $[i, \gamma]$ and its right child located at the beginning of the range $[\gamma+1, j]$.

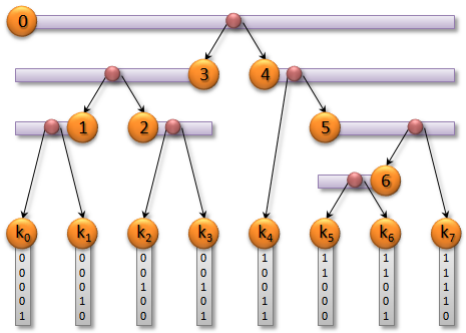

Figure 5: Bar representation of the keys covered by each internal node.

In Figure 5 we present a visual example of this property. Node 0 covers the entire range $[k_0, k_7]$ and is therefore located at $\mathbf{I_0}$. Its children, node 3 and 4 cover the ranges $[k_0, k_3]$ and $[k_4, k_7]$ and are placed at $\mathbf{I_3}$ and $\mathbf{I_4}$, respectively. Interestingly, this process will never result in gaps or duplicates when populating the internal node array. An advantage of using this scheme is that each internal node is conveniently placed next to a sibling node in memory. And since the internal nodes run from 0 to $n-1$ we can use them to directly address the nodes in memory.

In order to construct the BVH, we need to know more than simply what nodes cover which keys, we need to know how the nodes connect amongst themselves, i.e. the parent-child relations.

Lets say we want to determine the direction in which the range of keys covered by node $\mathbf{I_i}$ extends in. We call this direction d. In order to determine d we compare the length of the common prefix between the keys k_{i-1}, k_i, and k_{i+1}. Iif $\delta(i-1, i) > \delta(i, i+1)$ then $d = -1$. And if $\delta(i-1, i) < \delta(i, i+1)$, then $d = +1$.

Knowing this we can say that k_i and k_{i+d} both belong to node $\mathbf{I_i}$, and that k_{i-d} belongs to node $\mathbf{I_{i-d}}$.

We now need to know how far the range of each node extents. Since k_{i-d} does not belong to the range of keys covered by node $\mathbf{I_i}$, we can safely assume that the keys which are share amongst themselves a larger common prefix than k_i and k_{i-d} do. We call this common prefix δ_{min} so that $\delta_{min} = \delta(i, d-1)$.

$\delta(i, j) > \delta_{min}$ for any k_i belonging to $\mathbf{I_i}$. This being said all we need to do to find the other end of the range is to search for the largest l that satisfies the equation $\delta(i, i+ld) > \delta_{min}$. The fastest way to do this is to start at k_i and in powers of 2 increase the value of l until it no longer satisfies $\delta(i, i+ld) > \delta_{min}$. Once this happens we know that we have gone too far and we have left the range of keys covered by $\mathbf{I_i}$. Lets call this upper bound l_{max}. We know for sure the correct value of l is somewhere in the range $[l_{max}/2, l_{max}-1]$. Now it's only a matter of using a binary search to find the value of l under which $\delta(i, d(l+1)+i) \leq \delta_{min}$.

After finding the value of l we can use $j = i + ld$ to specify the other end of the range.

Lets call δ_{node} the length of the common prefix shared between k_i and k_j, given by $\delta(i, j)$. We use δ_{node} to search the split position γ that partitions the keys covered by $\mathbf{I_i}$. What we do now is perform a search for the largest $s \in [0, l-1]$ that satisfies $\delta(i, i+sd) > \delta_{node}$. I.e., we need to find the furthest key which shares a larger prefix with k_i than k_j does.

Discovering γ allows us to determine the ranges covered by each children. The left child will have a range covering $[min(i, j]), \gamma]$ and the right child will cover $[\gamma+1, max(i, j)]$.

For the final step we analyze the values of i, j and γ. If $i = \gamma$ we know $\mathbf{I_i}$'s left child is the leaf node $\mathbf{L}_\gamma$, otherwise it's the internal node $\mathbf{I}_\gamma$. Correspondingly if $j = \gamma + 1$ we say $\mathbf{I_i}$'s right child is the leaf node $\mathbf{L_{\gamma+1}}$, otherwise it's internal node $\mathbf{I_{\gamma+1}}$. Pseudocode for this algorithm can be found in Figure 6.

3.4. Compression

Reducing memory footprint and bandwidth has been our goal with this work. A certain amount of bandwidth is automatically reduced by the simple use of a BVH. We can further improve our gains by compressing both the BVH and the triangle mesh.

3.4.1 BVH Compression

We start a thread at each internal node and make our way up to the top of the tree. Once we reach the top of the tree we start making the same path in the opposite direction, that is, from the root node to the internal node in question. As we descend each level we take the bounding box of the previous parent and subdivide each dimension in 1024 segments. We treat this 1024x1024x1024 grid as a voxel space and map the minimum and maximum points of the current level's node bounding box into the nearest corresponding voxels.

It becomes clear we lose some precision as we descend each level since we are going from floating point coordinates into integer indexes. Each index will be contained in the range $[0, 1024[$, this is so we can store 3 of these indexes in a single 32 bit integer. Each bounding box can then be stored as

```
1:  for each internal node with index i ∈ [0, n − 2] in parallel
2:      // Determine direction of the range (+1 or -1)
3:      d ← sign(δ(i, i + 1) − δ(i, i − 1))
4:      // Compute upper bound for the length of the range
5:      δ_min ← δ(i, i − d)
6:      l_max ← 2
7:      while δ(i, i + l_max · d) > δ_min do
8:          l_max ← l_max · 2
9:      // Find the other end using binary search
10:     l ← 0
11:     for t ← {l_max/2, l_max/4, ..., 1} do
12:         if δ(i, i + (l + t) · d) > δ_min then
13:             l ← l + t
14:     j ← i + l · d
15:     // Find the split position using binary search
16:     δ_node ← δ(i, j)
17:     s ← 0
18:     for t ← {⌈l/2⌉, ⌈l/4⌉, ..., 1} do
19:         if δ(i, i + (s + t) · d) > δ_node then
20:             s ← s + t
21:     γ ← i + s · d
22:     // Output child pointers
23:     if min(i, j) = γ then left ← L_γ else left ← I_γ
24:     if max(i, j) = γ + 1 then right ← L_{γ+1} else right ← I_{γ+1}
25:     I_i ← (left, right)
26: end for
```

Figure 6: Pseudocode for the parallel construction of a BVH (source:[4]).

a single 2 integer data structure instead of a 6 float data structure, reducing its size down from 24 bytes to 8 bytes. It is obvious from this method that out BVH will become more loosely coupled as we need to perform roundings when going from a continuous space (world coordinates) into a discrete one (voxel indexes). We round up when mapping the maximum point of a bounding volume to a voxel and round down when mapping the minimum so out bounding volumes remain coherent. We predict this loss of precision won't have a big impact in our intersection tests, we will most likely intersect a few more bounding volumes as they will be slightly larger in size but we don't expect a significant increase in the number of tests. One thing to take into account is that in each thread we must quantize the entire path from the root to each internal node otherwise our BVH will become incoherent.

3.4.2 Mesh Compression

Karras[4] uses each leaf node as a redirect to a primitive. We can remove an indirection level by having each leaf node envelop a single triangle and store said triangle in the node itself. We can also try and diminish the number of memory necessary to store a triangle by following the same line of thought as in section 3.4.1. By quantizing the position of each vertex of each triangle into the bounding box that encapsulates it, we can go from having to store 9 floats to just 3 integers. This will of course have an impact on the model itself as we will be losing the original coordinates of each vertex. When recalculating each vertex back to world coordinates we will be changing the final world positions of each triangle, resulting in slight mesh deformation. Interestingly enough, this effect isn't noticeable unless examining models very up close. In a practical application like a game or simulation where the camera and the objects are constantly moving this effect could pass up unnoticed. This compression step can easily be done in the same kernel as the quantization the internal nodes of the tree, thus having little to no effect on the post-processing time of the BVH.

4. Results

Testes were made using an NVIDIA GeForce GTX 970 with 4 GB of RAM. We chose to render our images in a 1280x720p as this is one of the more commonly used resolutions for multimedia content.

We tested our algorithm with three different scenes, Asian Dragon, Cornell and Stanford Bunny.

Cornell (152 triangles) is a scene representative of highly reflective and refractive scenes. It also represents a low polygon scene. It consists of an icosphere surrounded by five mirrors and a glass prism. This scene focuses on testing performance gains in scenes with a high number of secondary rays.

The Stanford Bunny scene (70K triangles) is what we would call a "medium" sized scenes. It serves as a midterm between the very low polygon scenes (Cornell) and the very high polygon scenes (Asian Dragon). This scene also only features eye and shadow rays.

Asian Dragon (7.2 M triangles) is representative of complex objects with a high number of triangles. It's a dense model with a great number of triangles in a small space. Our intend with this scene is measure the performance gains of BVH compression for very dense BVHs. This scene only features eye and shadow rays.

4.1. Number of Ray-Box Intersection Tests

Despite being a small, enclosed scene we notice that lack of precision of the bounding volume areas causes an increase of 9% in intersection tests. We assume this is likely caused by reflected and refracted rays that would normally pass tangent to the icosphere but are instead tested and categorized as a hit.

In Standford bunny we notice an increase of 7.1% in the number of ray-box intersection tests, making it the less affected of all the 3 scenes.

In Asian Dragon we notice an increase of

12.38% in the number of ray-box intersection tests. Since this scene has a high tree we expect rounding "errors" to accumulate over the levels and lead to this increase in the number of intersection tests.

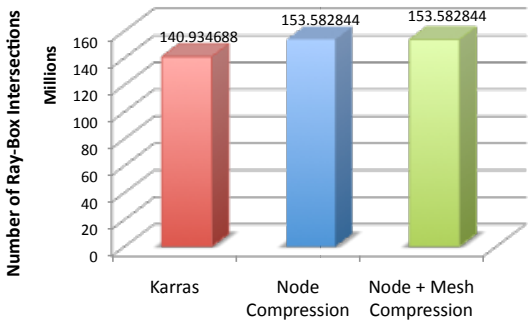

Figure 7: Number of ray-box intersection tests in Cornell.

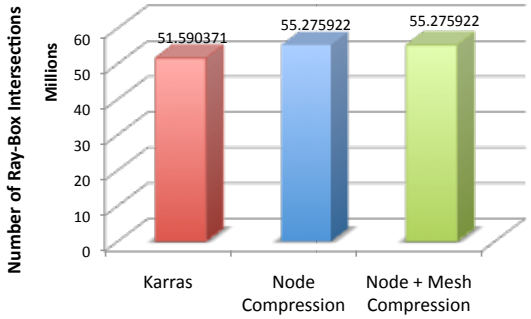

Figure 8: Number of ray-box intersection tests in Stanford Bunny.

4.2. Number of Ray-Triangle Intersection Tests

Cornell has an increase of 11.1% tests performed. We believe this number is greater in comparison to the other test scenes because in this scene almost every ray will hit an object and thus will have to traverse the entire BVH, making it more prone to the effects of loss of precision of the bounding volumes.

In Stanford Bunny we notice an increase of 7.4% in the number of tests performed. This increase in the number of ray-triangle test seems to be similar to the increase of ray-box intersection tests.

Asian Dragon has an increase of 10.75% tests performed. As in Stanford Bunny this number seems to go in hand with the increase of ray-box intersectiont tests, although slighty lower.

4.3. Global Memory Reads

The following results are based in the number of intersection tests performed in each case and the size of each internal node, leaf node and primitive.

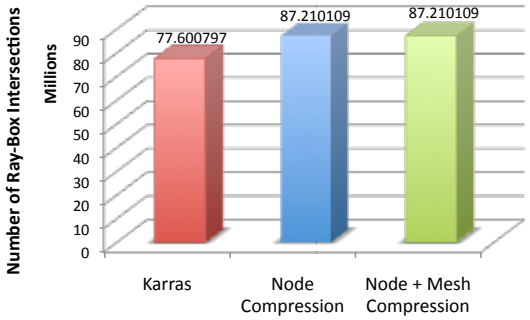

Figure 9: Number of ray-box intersection tests in Asian Dragon.

In Cornell we notice we are able to achieve a significant impact in the amount of memory we read when we implement node compression (14.5%), but we see little to no gains when implementing mesh compression. This is because in this scene the number of ray-box intersection dwarfs the number of ray-triangle intersection, hence optimizations made to the execution time of ray-triangle tests will have little impact.

Stanford Bunny reduces its memory accesses by 14% when compressing internal nodes alone and 22% when compressing the mesh alike. These results go in hand with the results obtained in Asian Dragon providing some insight that every model, be it high or low poly will be positively affected by our improvements.

In Asian Dragon we see a steady decrease of accessed memory as we implement our changes. We reduce memory accesses by 11.9% when using node compression and, by 20.3% when adding mesh compression

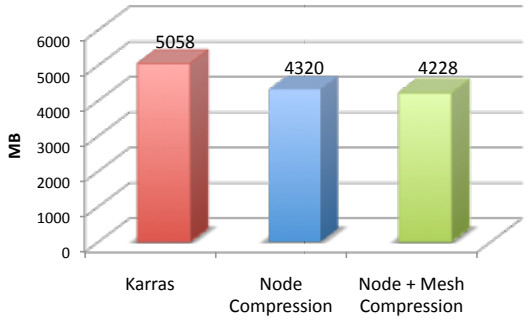

Figure 10: Estimated memory read during intersection tests in Cornell in each frame.

4.4. Kernel Execution Time

In this section we measure the execution time for each of the most relevant kernels. Karras repre-

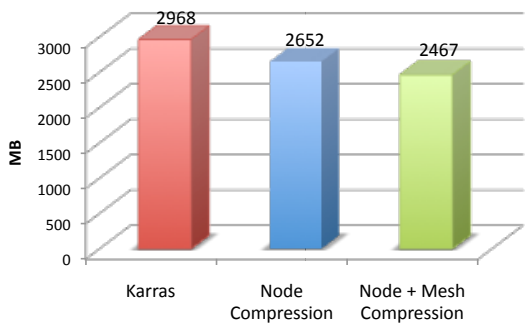

Figure 11: Estimated memory read during intersection tests in Asian Dragon in each frame.

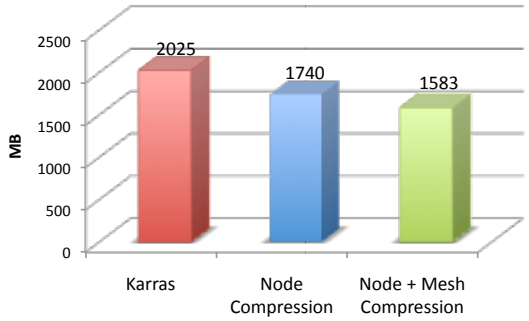

Figure 12: Estimated memory read during intersection tests in Stanford Bunny in each frame.

sents the control algorithm, as described by Karras in [4]. Node represents the same algorithm but with BVH compression. Node+Mesh represents the same algorithm but with BVH and triangle compression.

Cornell doesn't benefit much from our modifications. We notice a reduction in rendering time of 6.7% when compressing internal nodes and 10.7% when compressing the mesh alike. Both BVH construction time and compression are minimal. We believe our modifications do not have a great impact on this scene as almost every surface is either reflective or refractive so much of the rendering kernel's time will be spent generating secondary rays and performing floating point operations.

In this scene we notice most of the time is spent executing the rendering kernel as in Cornell . The creation of the BVH has a slight impact on the time

Cornell	Karras	Node	Node+Mesh
Kernel	Time (ms)	Time (ms)	Time (ms)
BVH Creation	2.168	2.168	2.168
Node Comp	0	0.009	0.009
Mesh Comp	0	0	0.011
Rendering	48.934	45.680	43.680

Table 1: Kernel execution time for Cornell frame.

bunny	Karras	Node	Node+Mesh
Kernel	Time (ms)	Time (ms)	Time (ms)
BVH Creation	7.815	7.815	7.815
Node Comp	0	0.243	0.243
Mesh Comp	0	0	0.346
Rendering	80.735	64.147	46.825

Table 2: Kernel execution time for Standford bunny frame.

required to render each frame and remains constant across all 3 variations. Compression times have little to no weight but greatly impact the time it takes to render the scene. Here we can see a reduction in rendering time of 20.55% when compressing just the BVH's internal nodes and a reduction of 41.01% when compressing both internal nodes and triangles. Overall we are only able to achieve a 37.33% increase in performance when considering the rest of the kernels.

Asian Dragon	Karras	Node	Node+Mesh
Kernel	Time (ms)	Time (ms)	Time (ms)
BVH Creation	424.639	424.639	424.639
Node Comp	0	16.328	16.328
Mesh Comp	0	0	24.433
Rendering	246.357	194.436	160.952

Table 3: Kernel execution time for Asian Dragon frame.

In Asian Dragon we see a time reduction 21.08% of the duration rendering kernel when compressing just the internal nodes and 34.67% when compressing both internal nodes and triangles. Here we can see our algorithm has a significat impact in high poly models as we are fetching a vast number of BVH nodes in this scene. Unfortunately the BVH Creation kernel takes up most of the time in each frame when dealing with such a high number of polygons. Overall we are only able to achieve a 6.6% increase in performance when we consider the rest of the kernels that make up the frame.

5. Conclusions

We talked about some of the problems of real time ray tracing and how they could approached. We proposed that by reducing memory footprint and memory bandwidth, an area often overlooked, we be able to increase performance of a real time ray tracing application. We looked into the state of the art in BVH parallel compression and BVH compression as our focus areas. We took the work of Karras[4] as our starting point and then proceeded to apply a series of compression steps to the BVH constructed at each frame.

In our tests we used the vanilla version of Karra's algorithm as a control benchmark. Tests showed all test scenes benefited from out approach. As we

hypothesized the number of intersection tests increased but the overall time it took to traverse the BVH and perform each intersection test decreased. Some scenes benefit more from our algorithm than others. We notice that in scenes with a high number of polygons the construction of the BVH becomes a bottleneck, taking most of the time in the frame. This affects our algorithm as much as the control algorithm. Gains from our approach were diluted by this bottleneck for this kind of scene.

As we expected real time ray tracing algorithm benefit from a reduction in memory footprint and memory bandwidth. Despite having to spend more time compressing and then decompressing both the BVH and the scene's primitives in each frame this penalty is compensated by the reduced time it takes to fetch the scene's data from global memory.

5.1. Future Work

BVH construction times reveal themselves to be a bottleneck in high polygon scenes. Further improvements in this area can make gains made from our contributions more relevant to the overall performance of the application. Using 64 bit types to store quantized values could also be a solution worth exploring. It's a middle term between the loss of precision caused the quantization values into 32 bits and the performance gained from reducing memory bandwidth.

References

[1] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[2] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 59–64. ACM, 2011.

[3] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[4] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[5] T. Karras and T. Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[6] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[7] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[8] J. Pantaleoni and D. Luebke. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.

[9] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques*, 2006:139–149, 2006.

[10] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *Computer Graphics Forum*, volume 28, pages 1691–1722. Wiley Online Library, 2009.

[11] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, Jun 1980.