Comparing Acceleration Data Structures for Real-time Ray Tracing on GPU

by

Feng Ge, B.Sc.(Hons)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2013

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Feng Ge

September 2, 2013

Permission to Lend and/or Copy

Acknowledgments

I would like to thank my supervisor Michael Manzke for his advice and suggestions throughout this project.

Feng Ge

University of Dublin, Trinity College
September 2013

Comparing Acceleration Data Structures for Real-time Ray Tracing on GPU

Feng Ge
University of Dublin, Trinity College, 2013

Supervisor: Michael Manzke

As one of the most significant techniques for the next generation video games, real-time ray tracing has become a hot research topic over the past few years. Ray tracing which represents the high realistic image producing algorithm has an expensive cost on both rendering time and memory consumption. However, the state of the art GPU is able to handle these issues due to the massively parallel processing power. On the other hand, more and more types of the acceleration data structure such as KD-Trees, Bounding Volume Hierarchies, Octrees... have been researched and proved can efficiently speed up the ray tracing process. The construction and traversal process for each these data structures exhibit different characteristics in terms of scalability, SIMD utilization, bandwidth usage, memory footprint, data structure quality, real time performance, memory layout concerns, cache performance etc. This project is proposed to implement, investigate and compare the characteristics of the construction process of some of these data structures on modern GPUs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rasterisation based real-time rendering pipeline is widely used in modern video game industry and computer graphics research. Given the transformed and projected vertices with their associated shading data, the goal of a rasterizer is to compute and set colors for the pixels covered by the object[1] . This field has been heavily researched for decades which leads to efficient, fast, satisfying rendering result. Optimizing techniques such as HDR effects , dynamic shadows, screen-space ambient occlusion can produce relatively realistic image in most cases. However, drawbacks of rasterisation process are still obvious, mainly when simulating reflection and refraction between scene objects.

Ray tracing is an ideal solution to generate high realistic image by simulating reflection and refraction of lights more close to nature. It accurately replicates the path of light through the scene, trace the light to collect light intensity on the object which hit by light rays. Modern research in ray tracing by means of computer was initiated by Appel in 1968 which limited by both the hardware and software resolutions for years[2]. With the development of modern GPU and the improvement of acceleration data structures like kd-tree and bounding volume hierarchy, ray-tracing technique is possible to be implemented in real-time. The most significant progress is to take advantage of the parallel computing capability of modern GPU to map the construction and traversal process to it. Many research work has proved that this approach can dramatically reduce time consumption and improve real-time performance. This project is proposed to further compare and evaluate the construction algorithm for real-time ray tracing in terms of construction time, bandwidth utilization, cache performance and branch efficiency.

The construction algorithm used in this project is implemented in CUDA which is a parallel computation platform designed by NVIDIA. Both of the kd-tree and BVH construction code are from third party's library to ensure quality of the algorithm and accuracy of the evaluation.

This introduction chapter will first give the motivation of this project. Following this, the objective is presented followed by layout of this report.

1.1 Motivation and Objective

As one of the most significant techniques for computer graphics and the next time generation video games, real-time ray tracing have become a hot research topic over past few years. Ray tracing which represents high realistic image generating algorithm is known as its expensive rendering time and memory consumption. The core concept of any kind of ray tracing algorithm is to efficiently find intersections of a ray with geometry primitives in a 3D scene. In order to achieve this, different types of acceleration data structures are employed to improve ray tracing in terms of real-time performance, time consuming, memory usage and so on. These data structures play a significant role in speeding up ray tracing and represents high efficiency on rendering.

Over the last few years, the GPU has become much more powerful on computation and handling parallel process. Modern GPUs are no long just a graphics processing unit but also has been proved the capability of sharing workloads with CPU. Developers exploit the features of the GPU to construct and traversal the acceleration data structures used in ray tracing. These data structures exhibits different characteristics in terms of scalability, SIMD utilization, bandwidth usage, memory footprint, data structure quality, real time performance, memory layout concerns, cache performance etc.

The construction and traversal algorithm implemented on GPU has been proved that can reduce rendering time and improve real-time performance. Research paper also show testing data for their implementation. However most of the research on acceleration data structures only gives construction time/traversal time and frames per second as their testing benchmark. This project aims to more detailed analysis on construction process to acceleration data structures for real-time ray tracing. Bounding volume hierarchy and kd-tree are picked as the tested data structures which typically represent object based scene description and space partitioning. We are interested in the construction algorithm for kd-tree and BVH in terms of construction time, bandwidth utilization, cache performance and Branch efficiency.

1.2 This project

This thesis contains a review of the current state of the art in ray tracing and relevant acceleration data structures followed by an introduction of CUDA and NVIDIA profiling tools. Chapter 3 covers the major design decisions made during project. This includes codebase selection, GPU performance metrics description and construction algorithms introduction. Chapter 4 discusses the setup for the project environment in terms of both hardware and software. Then the approaches used for evaluating the construction algorithms are explained. Chapter 5 evaluates the construction algorithms used to building acceleration data structures based on the collected data. We compare and analyze the construction process of different acceleration data structures in detail in this part. Chapter 6 concludes the results and gives the possible future works.

Chapter 2

State of the Art

2.1 Ray Tracing

In order to determine the color of a given point in a rendering pipeline, two models can be used for this purpose: local illumination and global illumination. Local illumination computes the intensity of a pixel by determining how much light is transmitted directly from the light source to the point of interest[3]. However a global illumination model takes not only the transmitted light but also the light indirectly reflected from other object surface. As the fact that in real world, most of the light doesn't come from the light source directly, the global illumination can simulate the real light more closely and generate a photo-realistic image. Each models uses different shading equation. Local illumination solved by Phong method at most of the time[4] while global illumination can be obtained by solving Whitted's illumination equation[5].

Ray tracing is a typical technique to achieve global illumination. According to the Whitted method, in order to compute the color of a pixel on the image plane of the virtual camera, a ray is cast from the camera into the scene. After determining the first hit point of this primary ray, the ray tracer computes the color being reflected into the direction of that ray by first computing the incident illumination at the hit point[6].

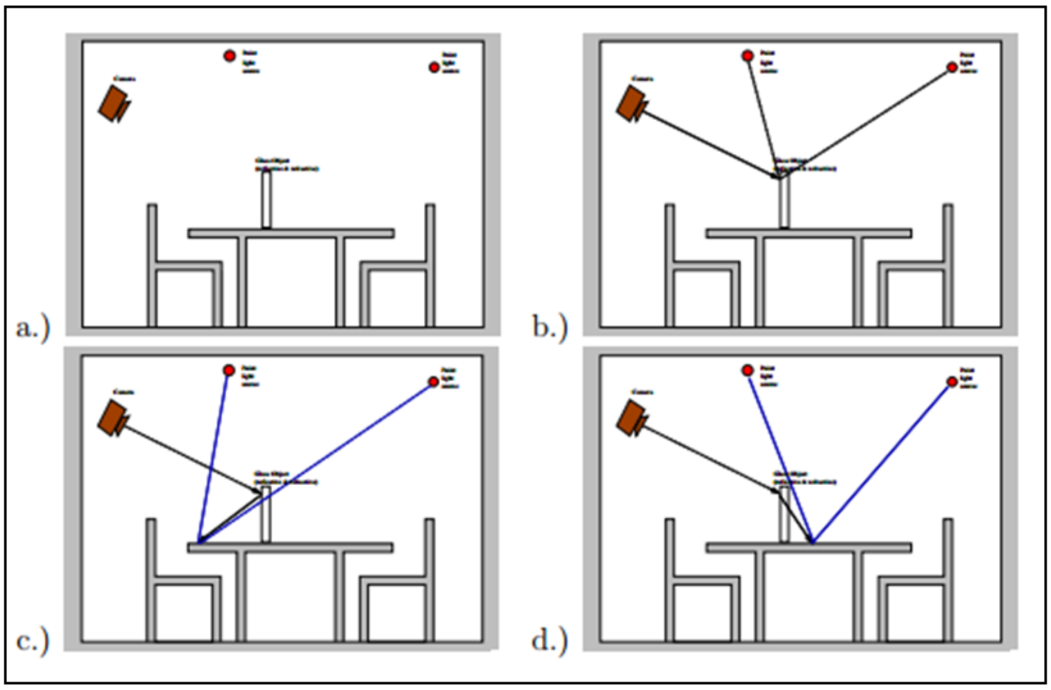


Figure 2.1: Recursive ray tracing[6]

Figure 2.1 shows a recursive ray tracing process. A camera, two light source and some objects are located in the scene (a). A ray is shot into the scene from camera and hit the object on the table. Two shadow rays send towards the light sources to determine the visibility (b). A secondary ray reflects from the hit point is traced recursively to compute the amount of intensity of incoming light (c). A refraction ray is traced recursively as well to computer refraction direction (d).

A modern graphics pipeline which is illustrated in Figure 2.2 is widely used in current game industry[7]. Many rendering pipeline such as OpenGL and DirectX are all based on this model which rasterizing primitives into fragments.

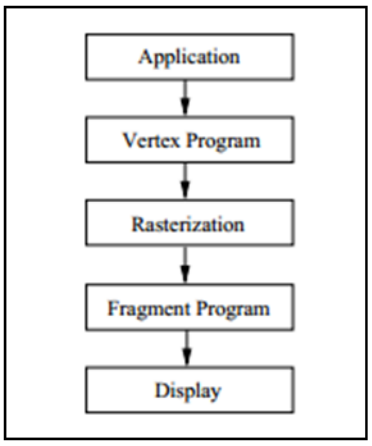


Figure 2.2: The programmable graphics pipeline

A typical way of implementing a ray tracing based rendering process is the rendering pipeline displayed in Figure 2.3. It contains 4 stages to render a image on the screen.

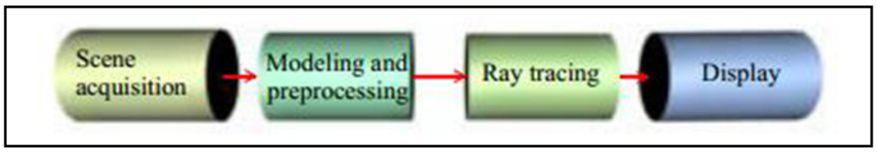


Figure 2.3: Ray tracing pipeline[3]

The first step is to acquire data from the scene description file. A ray tracer can define its own scene description language (SDL) to represent the objects in the environment. Some popular SDLs include POV files from Persistence, Inc. [8], RAY files for Rayshade from Stanford University[9], and VRML file format. The NFF file format proposed by Haines [10] is also commonly used in ray tracing literature. During the second stage, a data structure is constructed to speed up ray tracing algorithm which is most interested in this project and the third steps traverse the scene to find a object hit by a ray.

2.2 Acceleration Data structure

As we mentioned before, ray tracing based global illumination is very time consuming due to enormous amount of rays need to be shot and scene complexity is another significant factor which slows the process down. Considering a scene with thousands of objects, intersection test between ray and primitives will extremely slow if we use a brute-force approach. Therefore, the acceleration of a ray tracing algorithm is always an major topic over the past two decades. Many approaches have been proposed such as vertex tracing[11], the render cache[12], shadow caching[13] etc to speed up ray tracing.

The most successful approach to accelerate ray tracing is to reduce the ray-primitive intersection operations[6]. This is usually achieved by building an index data structures to find the close primitives to a given ray, and skip primitives that are far away. Many data structures have been proposed and proved can efficiently accelerate ray tracing like uniform grid, Octrees, Bounding Volume Hierarchies, kd-trees. In principle, all these techniques mainly differ in whether they can organize primitives hierarchically, or whether they can subdivide the scene into a set of voxels. This is project focus on two representative data structures: Bounding Volume Hierarchies and kd-trees.

2.2.1 Bounding Volume Hierarchy

Bounding volume hierarchies are widely used in ray tracing, physics simulation, visibility culling and similar applications to accelerate intersection tests. A Bounding Volume Hierarchy (BVH) is a rooted tree that each node in the tree is a bounding volume[5]. The internal tree nodes represent the bounding volumes that enclose all their children nodes. The leaf nodes are bounding volumes enclose scene primitives. A two layer Bounding Volume Hierarchy is illustrated in Figure reffig:bvh.

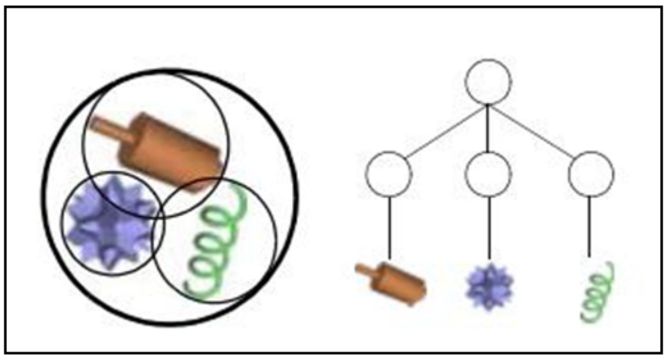


Figure 2.4: Bounding volume hierarchy

2.2.2 BVH Construction

Adding bounding volume to a primitive can speed up the intersection test. But the worst case running time for ray traversal is still O(n). Building a tree structure can reduce the ray intersection test by skipping all the uninteresting part of the tree and accelerate the time consuming of traversal to O(log n). When the test starts, the ray is first tested against the root level of the tree to determine whether there is intersection between this ray and largest bounding volume, then the next lower level of the tree will be tested until it reaches the end of the tree.

There are two conventional approaches to construct a BVH: bottom-up and top-down. Bottom-up approach is straightforward which encloses a fixed number of objects into a larger group until all the scene is enclosed. The root of the tree is the bounding volume of the whole scene. This approach requires a O(n) preprocessing time and space. Bottom-up method is a easy way to implement BVH, however it has a significant drawback that overlapped place is too much which slows down the traversal process.

Some other approaches are suggested to construct a BVH tree. Weghorst sorted objects by their x-coordinates before building the tree. This way reduces the overlapped area by grouping the objects closed to each other[14]. More time consuming is required when using this approach in construction and it takes O(n log n) to build a tree but it is more efficient in traversal. Also, Kay and Kajiya present another method based on different bounding volumes which improves BVH construction[15].

Kay and Kajiya introduced a top-down construction method for BVH tree[15]. They sorted the x-coordinates of the scene objects before construction and partitioned a tree node into two equal sized subnodes. The key point is to find the splitting plane of the bounding volume where the subdivision happens. For their approach, they simply cut the node at the median plane. The splitting process is recursively executed until each node contains only one object or empty. This top-down approach can create a balanced binary tree.

2.2.3 BVH Construction on GPU

The construction of bounding volume hierarchies are both successfully achieved on multicore CPUs and massively parallel GPUs. Lauterbach introduced a parallel construction algorithm for BVH based on spatial Morton codes[16]. The algorithm so-called linear BVH(LBVH). The idea is to simplify the problem by first choosing the order in which the leaf nodes appear in the tree, and then generating the internal nodes in a way that respects this order. In order to cluster objects which are close to each other, the algorithm sorts them along a space-filling curve which is defined in terms of Morton code and building the tree structure by recursively splitting intervals and creating internal nodes.

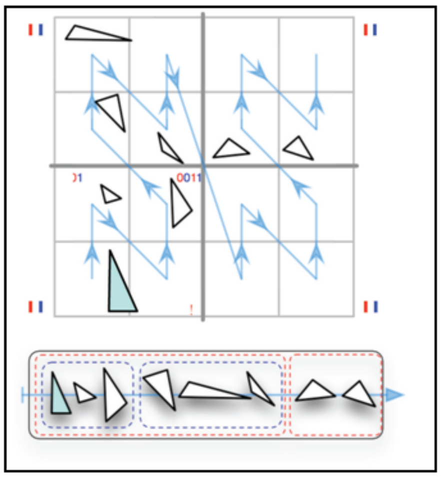


Figure 2.5: 2-D Morton code ordering primitives[16]

LBVH transforms the construction problem into a problem that sorting points along a curve. Once we have fixed the order of the leaf nodes, we can think of each node as just a linear range over them.

The main disadvantage of LBVH construction algorithm is that it simply splits the bounding volume at spatial median which leads to the inefficiency in traversal. This can be solved by surface area heuristic known as SAH[17]. The basic concept of any types of SAH algorithm is to calculate the cost for each splitting condition and pick the one with minimum cost. SAH requires extra cost for compute splitting position, but this can be accelerated by GPU as well. Some other approaches such Hierarchical LBVH construction given by J.Pantaleoni[18] can also take advantage of parallel processing capability of modern GPU and hybrid SAH and fast BVH construction within acceptable time scope.

2.2.4 Kd-tree

A kd-tree is a space partitioning data structure which is first introduced by Bentley[19]. A kd-tree is a special case of BSP-tree[3]. The difference between a kd-tree and a BSP-tree is the direction of the splitting plane. For a BSP-tree, the splitting plane can be oriented while the splitting plane for a kd-tree must be axis-aligned. Differing from BVH, kd-tree organizes 3D scene by subdividing space instead of grouping objects into clusters. A leaf node of a BVH tree should at most contain one object in the scene as it is a object based scene description. Whereas a kd-tree must stop splitting by given some level limitation. SAH algorithm is always used to construct a kd-tree to improve the traversal performance. Kd-tree can solve the space overlapping better than BVH even the latter can also use SAH approach to optimize tree structure. Kd-tree is an unbalanced binary tree requiring dynamic allocation on memory space which leads to more construction time than BVH.

2.2.5 Kd-tree Construction

A simple 2D example of construction of a kd-tree is illustrated in Figure 2.6.

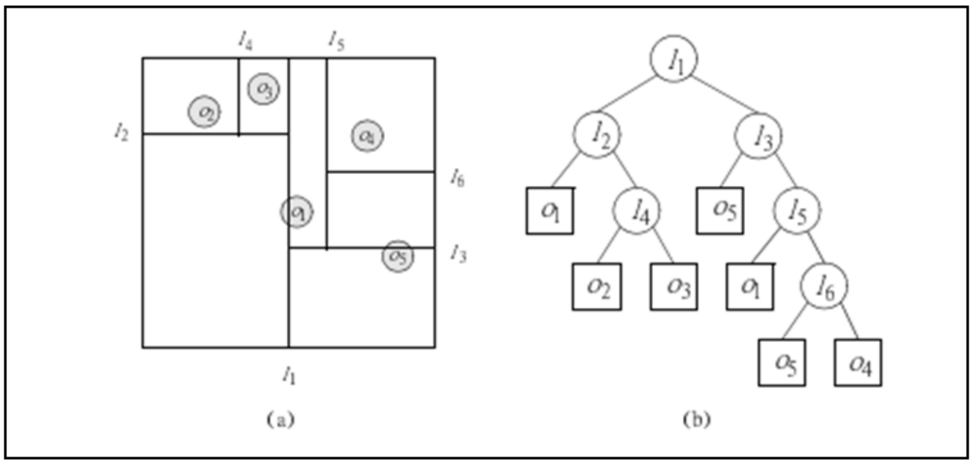


Figure 2.6: Basic kd-tree construction[3]

The entire scene is partitioned into 2 region by the line L1 which is parallel to the y-axis. L2 subdivide the region on the left side. As the region below L2 only contains part of the O1, we just need to keep subdividing the region above. This is the same for the region on the L1's right side. The tree is recursively built until no more than a single object is in each region. Figure 6(b) shows the constructed tree based on our subdivision process. As we can see, a kd-tree is always an unbalanced tree. Setting the threshold of a kd-tree can determine the height of a tree. In this case, the threshold is one. Choosing a reasonable tree height can affect the performance of kd-tree directly. A kd-tree with a heigh hierarchy may cause memory waste and end up many expensive vertical movements and a low height tree may cause more ray intersection test during traversal process.

The most important factor affects the performance of a kd-tree is to choose the splitting plane. The most naive approach is to split a region in its median point. This is accepted by some space partition algorithm like Kaplan BSP[20] or Samet PR kd tree[21]. This method is easy to implement and do not need any extra time to choose a splitting plane. Median splitting performs well if objects are well-distributed in the scene. But the disadvantage is obvious as well. Some region with few or none objects also will be subdivided which is meaningless and result in time and memory waste in construction. Analdi proposed a method that picking an axis-aligned splitting plane arbitrarily[22]. Suppose an object is chosen, the space can be subdivided by a plane that passes through the rightmost point of this object. His method can efficiently reduce the number of the object fragments.

A more successful approach is to optimize the positioning of the splitting plane by cost prediction functions proposed by Goldman and Salmon, MacDonald, Booth, and Subramanian[23, 17, 24]. A cost function is used to estimate how cost a split would be. The most famous of these cost prediction functions is the SAH as mentioned above introduced by MacDonald and Booth. The function assume that rays are equally distributed in the space, and are not blocked by objects[6]. Then the probability with which a ray hitting a voxel also hits any of its sub-voxel can be calculated. Once the respective probabilities are know, one can estimate the cost of a split.

2.2.6 Kd-tree construction on GPU

The first real-time kd-tree construction algorithm on GPU is provided by Zhou[25]. Following their algorithm, they build a kd-tree in a greedy, top-down manner. The splitting process uses SAH to evaluate costs and pick the plane with lowest cost. The algorithm exploits parallelism of modern GPUs heavily and has been proved much faster than the CPU based kd-tree construction algorithm.

Just similar to the BVH construction, kd-tree construction can be easily extended to the parallel construction in multiple threads[26]. This is due to both the primitives sorting and node splitting can be mapped perfectly in parallel where each single thread is assigned with equal number of primitives.

2.3 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA[27]. Modern GPUs are no longer limited to handle graphics application. Developers tend to expose GPU computing for general purpose. In order to make GPUs to easily program for both graphics and non-graphics applications, CUDA was introduced by NVIDIA since February 2007 and starts to be widely used to develop application for GPUs. The construction algorithm used in this project is fully implemented in CUDA. The CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications including,

1. CUDA C/C++ which is a GPU programming language based on industry-standard C/C++.
2. Straightforward APIs to manage devices, memory etc.
3. NVCC compiler to compile and link CUDA C/C++ code.
4. Hardware support drivers.

2.3.1 CUDA Programming model

Kernels

The most basic and significant concept in CUDA programming is kernel. A CUDA kernel is the function programmed in CUDA C running in parallel by CUDA threads. The following code implements a simple HellowWorld kernel.

```
__global__ void mykernel(void){
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

A CUDA kernel is defined using the __global__ declaration specifier. The number of threads that executes that kernel is specified by <<< ... >>> execution configuration syntax. Each thread will be allocated a unique thread id that can be accessed while the kernel calling[28].

Thread Hierarchy

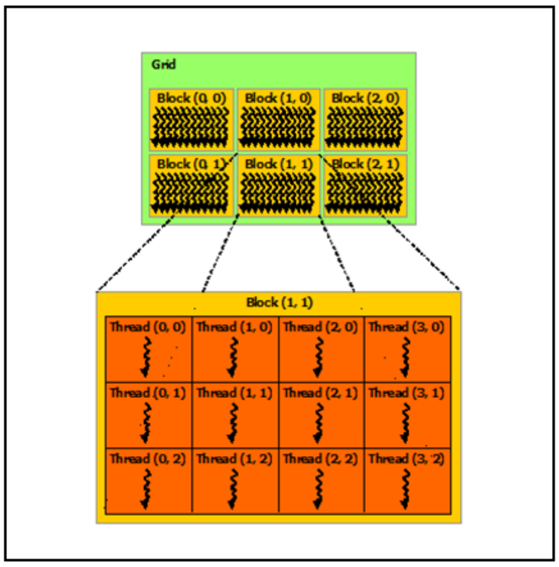


Figure 2.7: CUDA thread hierarchy[28]

CUDA defines a thread id in a 3D vector manner, so that threads can be defined using one-dimensional, two-dimensional or three-dimensional index forming a one-dimensional, two-dimensional or three dimensional thread block. The number of threads in each single block is limited as each thread in a block is expected to run on the same processor. Blocks are organized into a one-dimensional, two-dimensional or three-dimensional grid which is shown in Figure 2.7.

In modern GPU such GTX and GTX series cards generally supports 1024 threads per block. The total threads executes a kernel will be the number of threads per block times number of blocks. This is important because we always want to fully take advantage of GPUs' throughput and maximum processors' occupancy to improve performance of the application.

Memory Hierarchy

CUDA devices use several memory space. These memory spaces include global, local, shared, texture, and registers, as shown in Figure 2.8.

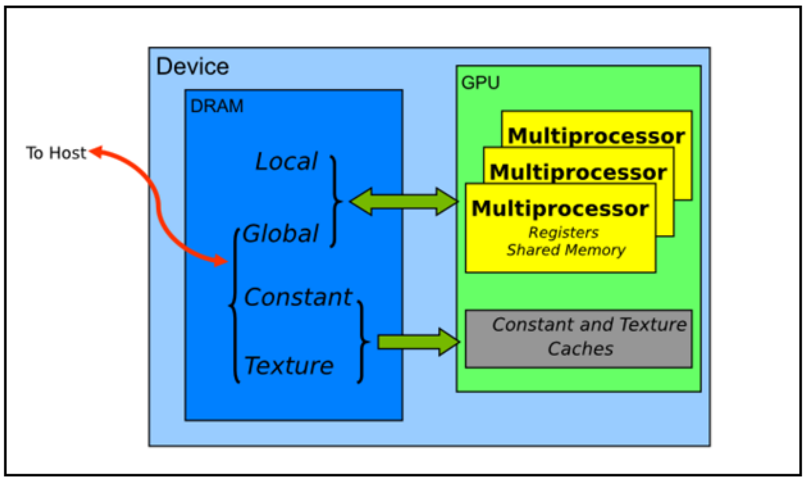


Figure 2.8: Memory space on a CUDA device[28]

CUDA threads access data from different memory space during execution. Each thread has private local memory. Each block has shared memory available to all threads in that block. All thread can have access to the same global memory.

Heterogeneous Programming

CUDA uses heterogeneous parallel computing to run code on both CPU(Host) and GPU(Device) concurrently. GPUs are powerful engines capable of running thousands of lightweight threads in parallel. This capability makes them well suited to computations that can leverage parallel execution. However the GPU device is distinctly different design from CPUs. It is important to understand those difference and how they can determine the performance of CUDA application.

1. CPU is optimized for fast single-thread execution. Host systems can only support a limited number concurrent threads. Each thread on a CPU is generally heavyweight. A CPU is designed for low-latency cache accesses and large cache size usually is used to hide DRAM access times.

2. GPU is optimized for high multi-thread throughput. Threads on GPUs are extremely lightweight. However thousands of threads are queued up for work. GPUs are designed to handle large number of concurrent, lightweight threads in order to maximize throughput.

Both the host and the device maintain their own memory spaces and connected by PCI-E bus. Serial code executes on the host while parallel code executes on the device. The processing flow for one cycle of a kernel call can be concluded by three steps[29]:

1. Copy input data from CPU memory to GPU memory through PCI-E bus.

2. Load kernel to GPU and execute, caching data on chip for performance.

3. Copy results from GPU memory to CPU memory through PCI-E bus.

Chapter 3

Design

In this section, the major design decision made during project will be introduced. First of all, the choice of codebase to implement acceleration data structures construction will be introduced. After that, the performance metrics are described as the evaluations of kd-tree and BVH construction will be undertaken in terms of these metrics. Thirdly, a detailed introduction of the construction algorithm used in this project will be given.

3.1 Choosing Codebase

The objective of this project is to compare and evaluate the construction process of kd-tree and BVH used for real-time ray tracing in terms of some of the GPU architectural characteristics such as memory throughput, cache performance or branching efficiency. The evaluation requires a complete and accurate implementation for kd-tree and BVH construction algorithm. This is the major reason why we decided to use third party's codebase in this project.

Choosing a codebase for the evaluation is one of the most challenge task during this project. The best option is to find an paper-supported implementation which can provide a high quality application for acceleration data structure construction and leads to justified and convinced evaluation results later. Unfortunately, most of the research does not release their implementation to public. Only several open source implementations correspond to real-time ray tracing are found.

The first project presents an implementation of the work of Kirill Garanzha and

Carles Loop entitled "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing"[29]. This project focuses on ray traversal on GPU and uses Optix as its ray tracing framework. According to Optix official documentation, Optix doesn't provide open source acceleration data structures builder, whereas only pre-defined data structures can be chose[30]. There is no entry point for building a specific data structure which makes impossible to test and evaluate the construction process while the application is running.

The second choice is the ray tracing framework developed by Martin Zlatuka supported by their research paper "Ray Tracing on a GPU with CUDA-Comparative Study of Three Algorithms"[31]. This is a completed implementation of ray tracing rendering pipeline. The framework contains acceleration data structures construction, ray traversal and Phong shading. The problem of this application is that they construct the acceleration data structures on CPU before sending to the ray tracer. This is obviously not meet the project's goal which requires a GPU construction for acceleration data structures.

The last one is the NIH library supported by NVIDIA[32]. NIH is a simple library for acceleration data structures construction. It is a fully CUDA based library implemented several kd-tree and BVH construction algorithms. The drawback of NIH library is that it doesn't contain any ray traversal and shading functions. The library is completely individual and using their implementation means it is barely possible to evaluate tree construction with real-time performance. However this may be the only quality ensured open source codebase can be found at present.

3.2 Performance Metrics

As mentioned before, we are interested in how the kd-tree and BVH construction can take advantage of modern GPU architecture and measuring performance for the construction process based on some performance metrics. In this project, four metrics are considered and they are described in detail in this section.

3.2.1 Timming

To measure the elapsed time of the construction process of kd-tree and BVH is the most basic analysis for performance. Most of the research paper for acceleration data structures will evaluate their algorithm by measuring the execution time and this project will also give a timed evaluation for kd-tree and BVH construction. It is straightforward to compare the efficiency for the construction of these two acceleration data structures.

Either the CPU or GPU timers can be used for CUDA calls and kernel executions. As all CUDA kernel launches are asynchronous, using a CPU timer requires to synchronize the CPU thread with GPU by calling cudaDeviceSynchronize(). Alternatively, CUDA event API provides GPU timers for time measure. This is achieved by calling record events before and after the kernel calls.

3.2.2 Memory Bandwidth

Bandwidth is defined as the rate at which data can be transferred which is one of the most important and commonly used gating factors for performance. When evaluating bandwidth efficiency, we use both the theoretical peak bandwidth and the observed or effective memory bandwidth. When the latter is much lower than the former, performance is drastically reduced.

Theoretical bandwidth can be calculated using hardware specifications available in the product literature. Effective bandwidth is computed by timing the execution of the program and measuring the amount of data which is processes:

$$Effectivebandwidth = (Br + Bw)/Tp \tag{3.1}$$

where Br is the number of bytes read per kernel, Bw is the number of bytes write per kernel and Tp is the execution time on p processor[33].

3.2.3 Cache Performance

The GPU cache hierarchy consists of per-core private L1 data caches and a shared L2 cache[34]. All modern CUDA capable cards (Fermi architecture and later) have a fully coherent L2 cache with much higher bandwidth than a typical CPU's L2 or L3 cache.

However, unlike CPU's L1 cache, the L1 caches in CUDA capable card are not coherent. This means that if two different streaming multiprocessors are reading and writing to the same memory location, there is no guarantee that one SM will immediately see the changes from other SM.

We are interested in the several cache performance metrics explained as follows:

1. L1 cache local hit rate, percentage of local memory loads that hit L1 cache.

2. L2 cache hit rate, hit rate at L2 cache for all read requests from L1 cache.

The higher cache hit rate during a construction process will reduce the time consuming for accessing memory space and the efficiency of loading and storing data. This will directly affect the memory throughput. Cache performance is one of the critical metrics we are considered in this project.

3.2.4 Branch Efficiency

Once a block is assigned to an streaming processor, it is divided into units called warps in CUDA. For current GPU, each warp contains 32 threads and executed in a SIMD fashion which means all threads within a warp must execute the same instruction at any given time. This may cause the issue that threads inside warps branches to different execution paths(Different instruction execute in the same warp such as thread 0 executes addition operation while others are doing multiplication). SIMD branching allows branching and looping a program, but because all processors must execute identical instructions, divergent branching can result in reduced performance[35].

Branch efficiency is a significant performance metric in this project. When comparing kd-tree and BVH construction, branch efficiency may become a potential factor that affects the construction time.

3.3 Tested Algorithms

According to the construction algorithms provided by NIH library, three algorithms are compared and evaluated including a kd-tree construction algorithm based on Morton code construction, a LBVH construction by sorting points over the Morton space-filling curve, and using middle spatial splits to separate them into clusters and a binned SAH algorithm for a middle-quality BVH builder.

3.3.1 Morton Code Based Fast Kd-tree Construction

The kd-tree construction algorithm implemented in NIH library is quite abstract and simple. The input of the algorithm is a set of random points which defined as a 4D vectors in float. Each point in set can be treated as a barycenter of a primitive, represents the bounding box of this primitive. The construction algorithm uses these point to determine which voxel a primitive should belong to.

The algorithm is started by computing Morton code for each point. The space-filling Morton curve or z-order curve is used to order primitives in any dimension space. Morton code is well-known because it determines a primitive's order along the curve which can be computed directly from its geometric coordinates, whereas some other space-filling curves require more expensive constructions to determine the ordering of primitives[16].

To calculate a Morton code for the given 3D point, we start by looking at the binary fixed-point representation of its coordinates, as shown in the Figure 3.1. To construct a $2^k \times 2^k \times 2^k$ lattice within the enclosing space, firstly take the fractional part of each coordinate and expand it by inserting two gaps after each bit. Second, interleave the bits of all three coordinates together to form a single binary number. If we step through the Morton codes obtained this way in increasing order, we are effectively stepping along the z-order curve in 3D.

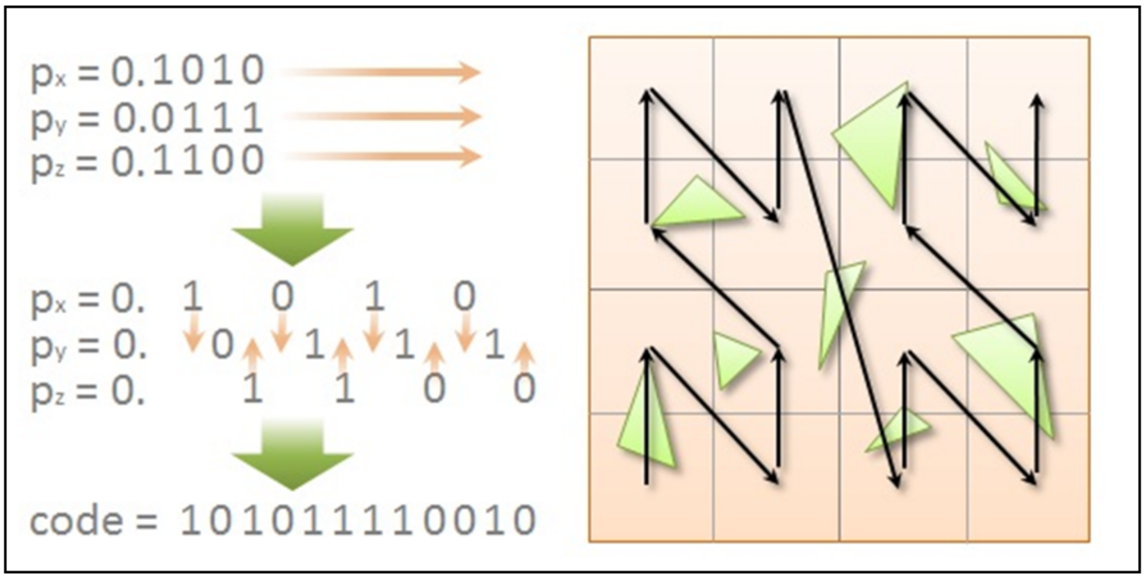


Figure 3.1: Morton code calculation for given 3D point[36]

Go back to the kd-tree construction algorithm, Morton code is computed for each point and point indices are set, which can perfectly take advantage of the GPU parallelism. The second step is to sort the objects accordingly. This can be achieved by parallel radix sort as we have successfully compute the Morton code to each point. Radix sorting is a very simple and powerful in terms of both logic and implementation[37]. The radix sorting method works by iterating over the d-bit digit-places of the keys from least-significant to most-significant. For each digit-place, the method performs a stable distribution sort of the keys based on their digit at the digit-place.

Assuming we have two points P(x_p, y_p, z_p) and Q(x_q, y_q, z_q), if the ith bit of point P's Morton code is the first that differs in point Q's Morton code then point P and point Q lie on different sides the ith cutting plane. The NIH's kd-tree construction algorithm simply use middle spatial splits and the sorted point makes it very fast to determine where a specific point should belong to.

3.3.2 LBVH construction

Linear Bounding Volume Hierarchies are a kind of BVHs built sorting points over the Morton space-filling curve, and using middle spatial splits to separate them into clusters. Although the resulting hierarchies are not very tight, but this is probably the fastest known method for building BVHs. This is because the LBVH is inherently parallel. As noted in the kd-tree construction, the Morton code can be computed to order primitives along space-filling curve. LBVH uses the same Morton code approach to order primitives. Morton code contains all necessary information where is a tree node should go when building the hierarchy. In the sorted sequence, two adjacent primitives' Morton code uniquely store the path from the root node to the leaf that contains only the primitive. Their least common ancestor and the farthest node from the root whose subtree contains both can be determined by the differ between the most significant bit of in their Morton code[16].

One of the great things about LBVH is that once we have fixed the order of the leaf nodes, we can think of each internal nodes as just a linear range over them. Suppose that we have n leaf nodes in total. The root node contains all of the leaf nodes and covers a range [0, n-1]. For some appropriate choice of m, the left child of the root node must cover the range [0, m], while the right child of the root must cover the range [m+1, n-1]. Therefore, start with a range that covers all primitives(from 0 to n-1), and choose a appropriate position to split the range in two. Recursively splitting the range in this manner to generate the whole hierarchy. The recursion terminates when the splitting process encounter a range that contains only one object, in which case we create a leaf node.

The remaining task is to determine the interval dividing lines in other word, for objects i and i+1. we want to know at what level they should be split into separate node. LBVH determines split according to the highest bit that differs between the Morton codes within the given range. To make it simple, the objective is to divide objects so that the highest differing bit will be zero for all objects in left child, and one for all objects in right child. Dividing objects based on the highest bit of Morton codes can classify objects on either side of an axis-aligned plane. Figure 3.2 illustrates splitting process according to the Morton codes.

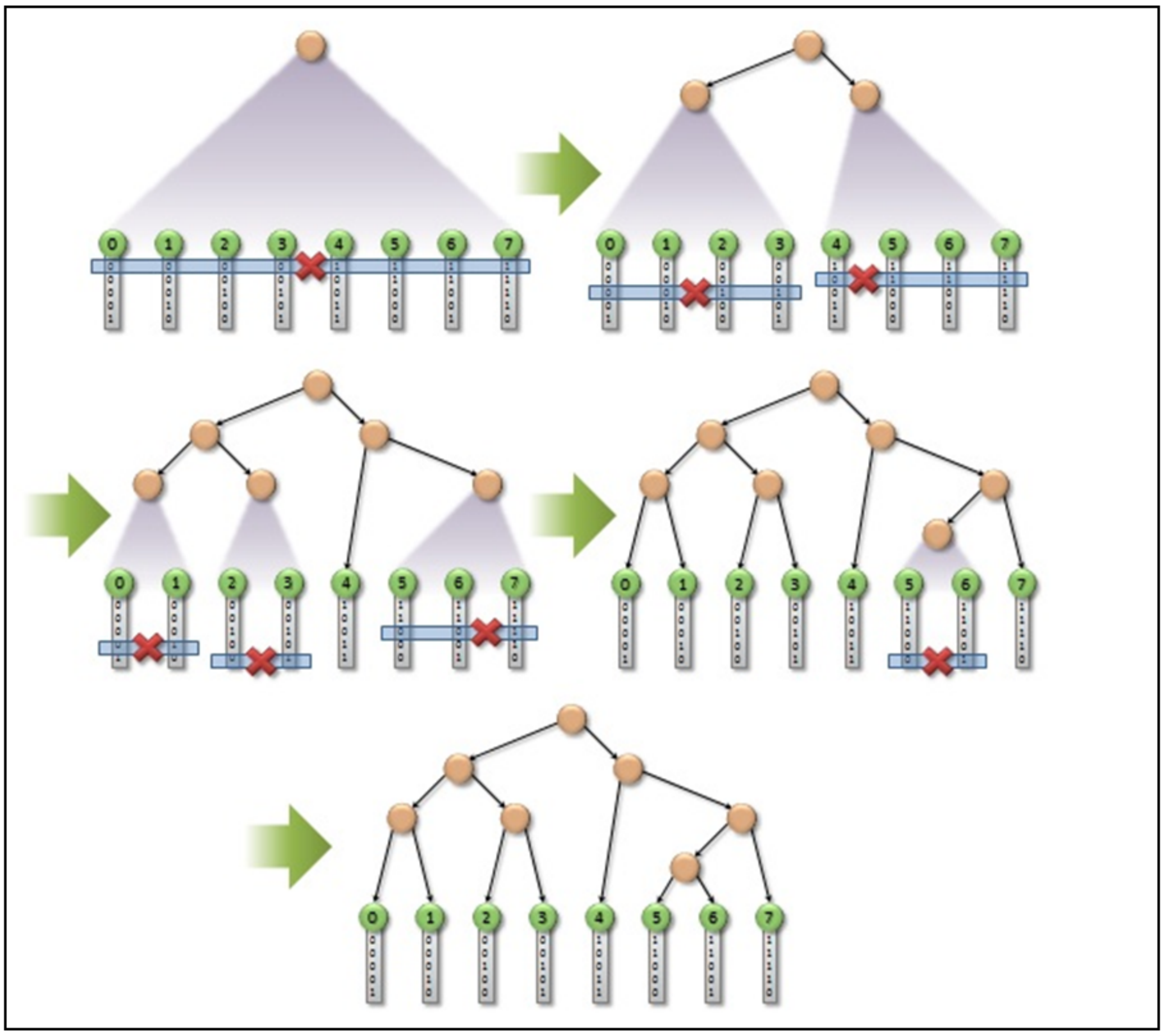


Figure 3.2: LBVH generation based on Morton code splitting[36]

In practice, binary search are used to find where the next bit differs. The question is how to parallelize this kind of recursive algorithm. According to the paper "Fast BVH construction on GPUs", they maintain a growing array of nodes in a breadth-first order, so that every level in the hierarchy corresponds to a linear range of nodes. Once a level is given, one thread is launched for each node that falls into this range. The thread will call the split function and append the resulting child nodes to the same node array and writes out their corresponding sub-ranges. Iterating this process so that each level outputs the nodes contained on the next level and then operated in the next round.

3.3.3 Binned SAH BVH construction

NIH library provides a binned SAH BVH builder, which can generate a middle-quality BVH supported by paper "Simpler and Faster HLBVH with Work Queues"[38]. This algorithm is implemented based on the LBVH construction algorithm described above. Furthermore several improvements are made to optimize the tree quality and better utilize the parallelism.

The first contribution of the algorithm is implemented a fast work queue generation method. When building the tree hierarchy, each node per level will be scanned if it needs to split. The splitting process will generate new nodes which will be treated as new input data for next round partition. The problem is how to generate a new fitted new queue for new nodes whose quantity is different from their ancestor. They introduce an approach that hybrids local prefix sum and atomic increment on global counter for each element of input queue, which is illustrated in Figure 3.3.

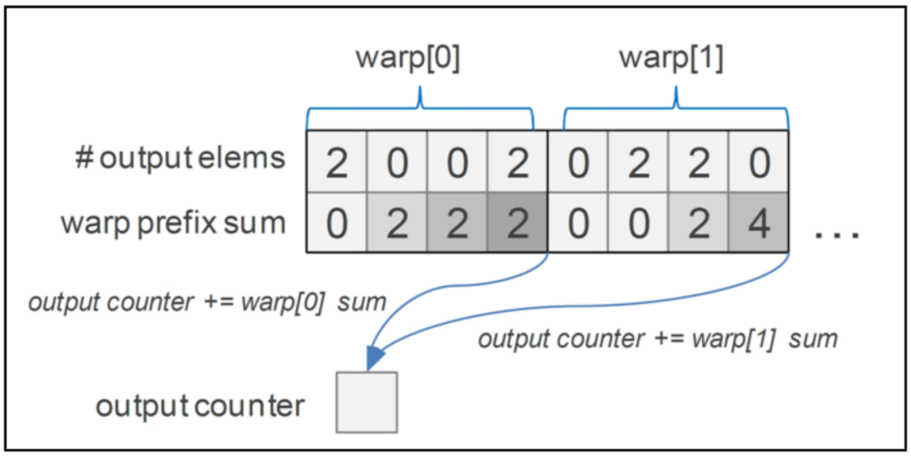


Figure 3.3: Efficient queuing with warp-wide reductions

Each warp keep fetching tasks from input data queue and each thread in the same warp processes one task. After each thread finishes its task and knows the number of elements in the output queue, it computes the offset of their output tasks according to the warp prefix sum. This approach can process each level of the hierarchy in a single kernel that leads to a satisfied improvement in terms of latency.

The algorithm starts from the root node noted as split task 0. Then the algorithm processes 3 step for each loop:

Binning: many research has proved that using approximate binning techniques can improve the construction process of SAH based tree structure[41]. The bounding box for each node is divided into eight bins and a bin stores the original bounding and a count. Clusters are assigned to corresponding bins in parallel and counter for each bin continues increasing.

SAH evaluation: the algorithm splits cluster which is the best one from the distributed bins. For each split-task, if a single cluster is encountered, the task stops. Otherwise, two output split-tasks are created with bounding boxes corresponding to the left and right subspaces determined by the SAH split.

Cluster distribution: after computing the best splits, clusters are distributed to their new split-task by comparing the bin id of the clusters to the id of the best split.

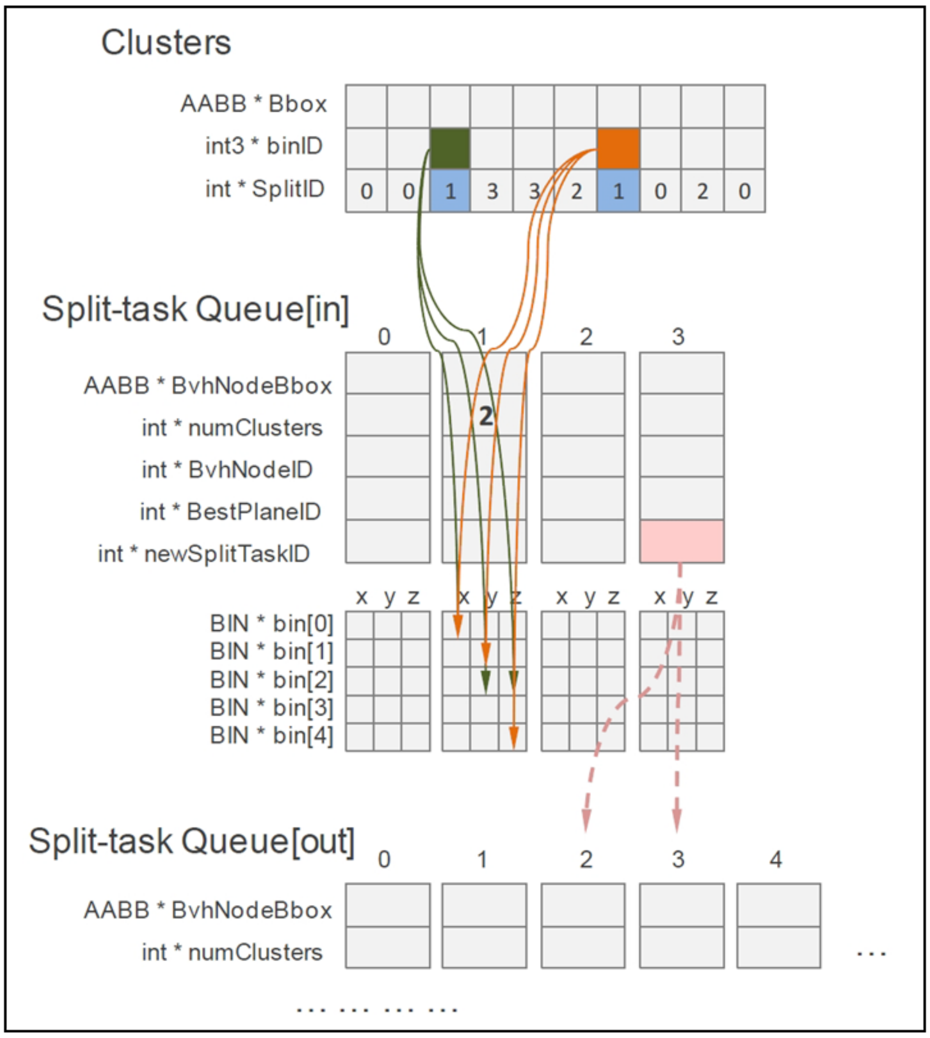


Figure 3.4: Binned SAH split task[38]

Figure 3.4 illustrates the SAH binning procedure from clusters distribution to node split in input work queue and two corresponding new split-tasks are generated at last.

Chapter 4

Implementation

This chapter will discuss the implementation of this project following the design decision made in previous chapter. As this is not a programming based project, this section will firstly introduce the setup for the testing and evaluation environment in terms of both software and hardware. Secondly, the approaches used for evaluating the kd-tree and BVH construction algorithm will be talked about.

4.1 GPU Setup

The construction algorithm for acceleration data structures are tested on NVIDA GeForce GTX 680 graphics card. Table 4.1displays some basic device information.

Device	NVIDIA GeForce GTX 680
CUDA capability	3.0
Total amount of global memory	2048 MBytes
8 multiprocessors	1536 CUDA Cores (192 for each)
GPU clock rate	1059 MHz
Memory clock rate	3004 MHz
Memory bus width	256-bit
L2 cache size	512 KBytes
Total amount of shared memory per block	48 KBytes
Total amount of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Maximum sizes of each dimension of a block	$1024 \times 1024 \times 64$
Maximum sizes of each dimension of a grid	$2147483647 \times 65535 \times 65535$

Table 4.1: Device information

4.2 CUDA and IDE Setup

This project uses the newest CUDA 5.5 toolkit and Microsoft Visual Studio 2010 IDE to compile the codebase for construction algorithms. CUDA 5.x provides a good support for Visual Studio 2010 and 2012, which is fully integrated. This simplifies the configuration procedure that users only need to install CUDA toolkit and it will do the rest of the work automatically.

4.3 Code Compile

This is one of the most challenge tasks in this project. The NIH library is developed about 3-4 years ago. The project is based on an old version of CUDA SDK and some third party library such as Thrust. CUDA SDK contains its own compiler called nvcc to compile CUDA C/C++ code. Each CUDA based project will define its compiler version and this causes a different version of nvcc cannot compile the code which assigned to another version of nvcc compiler. So that a new CUDA 5.5 project is created to hold the source code for the implementation.

Another issue is their implementation depends upon Thrust library which is a high level wrapper for CUDA C which pre-defined some frequently used kernels such as data copy or radix sort kernels. The library has been modified over several versions and many header files and static libraries are discarded or integrated into CUDA runtime API. In order to make the codebase compiled, I trace back to the old version of Thrust in their trunk and download the missing headers and include it to the project path.

4.4 Evaluation Tools

The evaluation for building kd-tree, LBVH and binned SAH BVH relies on two evaluation tools Nsight and Visual Profiler provided by NVIDIA.

Nsight, a powerful debugger and CUDA application profiler integrated with Visual Studio. I take advantage of Nsight to debug the codebase and have a insight of detailed memory usage of the algorithm. Nsight gives a visualized memory access evaluation of the tested application, however using Nsight to profile the application always requires a long testing period.

Visual Profiler, another application profiler for CUDA based project. Visual profiler is integrated in Eclipse. Visual profiler generates the timeline to trace the CUDA code and gives detailed analysis such as global memory throughput or instruction throughput for each launched kernel. A great advantage of Visual Profiler is that it displays profiling result in form which can be exported into CSV format easily.

4.4.1 Construction Algorithms Evaluation

Most of the evaluation tasks are achieved using NVIDIA Visual Profiler. When launching Visual Profile, the first thing is to generate a timeline for the profiled application. For each construction algorithm, eight released executable files are made corresponding to different point sets. In Visual Profiler, a new session is created to hold the tested app and generate a timeline for showing GPU and CPU activities. This is illustrated in Figure 4.1:

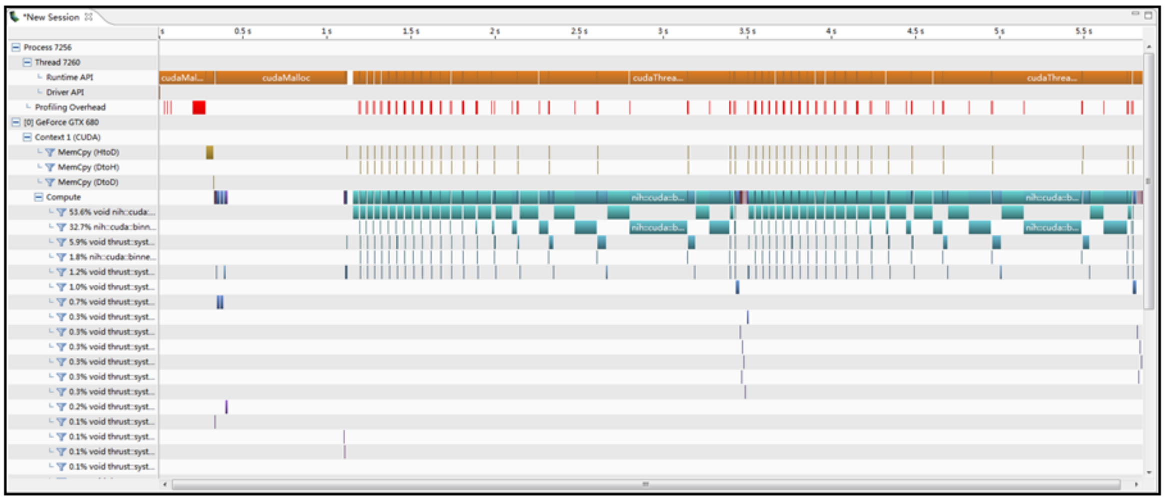


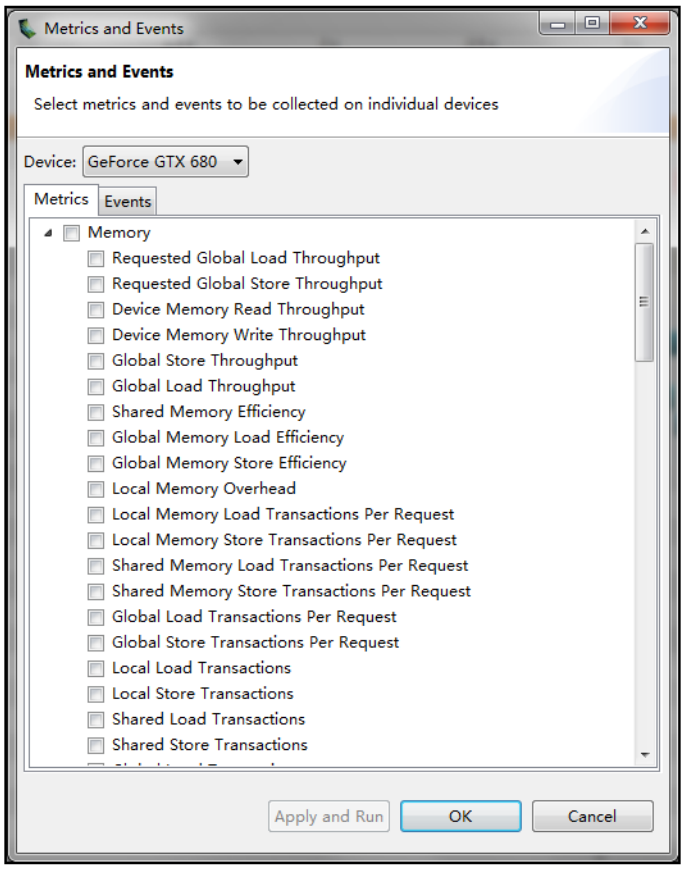
Figure 4.1: Timeline generated in Visual Profiler

The next step is to configure the metrics and events need to be collected. We are interested in the GPU metrics and event listed as follows:

Metrics	Events
Global memory load/store throughput	Branch
Local memory load/store throughput	Divergent branch
Shared memory load/store throughput	Branch efficiency
DRAM read/write throughput	
L1 cache local hit rate	
L1 cache local hit rate	

Table 4.2: Collected GPU metrics and events

These metrics and events can be selected in the ”Metrics and events” pop-up menu in Visual Profiler which is shown in Figure 4.2. In order to collect these information, it requires to re-run the application for several times.



Metrics and Events

Metrics and Events

Select metrics and events to be collected on individual devices

Device: GeForce GTX 680

Metrics | Events

- Memory
  - Requested Global Load Throughput
  - Requested Global Store Throughput
  - Device Memory Read Throughput
  - Device Memory Write Throughput
  - Global Store Throughput
  - Global Load Throughput
  - Shared Memory Efficiency
  - Global Memory Load Efficiency
  - Global Memory Store Efficiency
  - Local Memory Overhead
  - Local Memory Load Transactions Per Request
  - Local Memory Store Transactions Per Request
  - Shared Memory Load Transactions Per Request
  - Shared Memory Store Transactions Per Request
  - Global Load Transactions Per Request
  - Global Store Transactions Per Request
  - Local Load Transactions
  - Local Store Transactions
  - Shared Load Transactions
  - Shared Store Transactions

Figure 4.2: Metrics and events configuration in Visual Studio

The final step is to export the collected GPU metrics and events to CVS format which can be read by Excel to generate form and chart. As this is a evaluation based project, the implementation is all about how to setup testing environment and how to collect necessary data through profiling tools. More detailed comparison of constructing kd-tree, LBVH and binned SAH BVH will be discussed in next chapter.

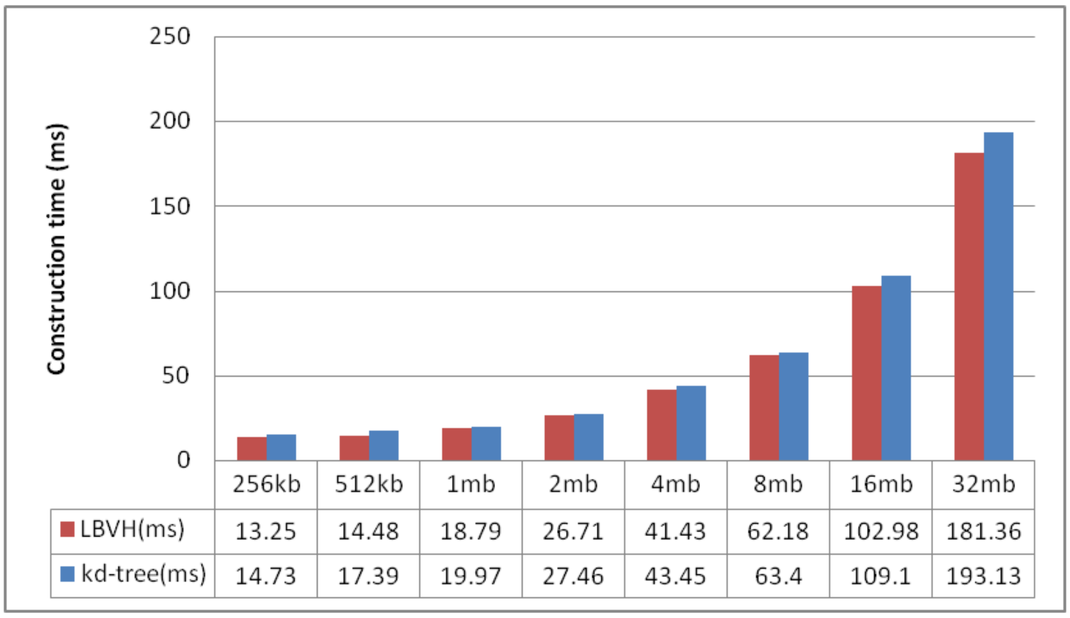
Chapter 5

Evaluation

In this chapter, we evaluate three construction algorithms based on the data collected from Visual Profiler. Construction time, memory throughput, cache performance and branch efficiency of building kd-tree, LBVH, binned SAH BVH are compared and analyzed. We are interested in how memory bandwidth utlization, cache hit rate and branch divergent will affect the construction time of these three acceleration data structures.

5.1 Construction Time

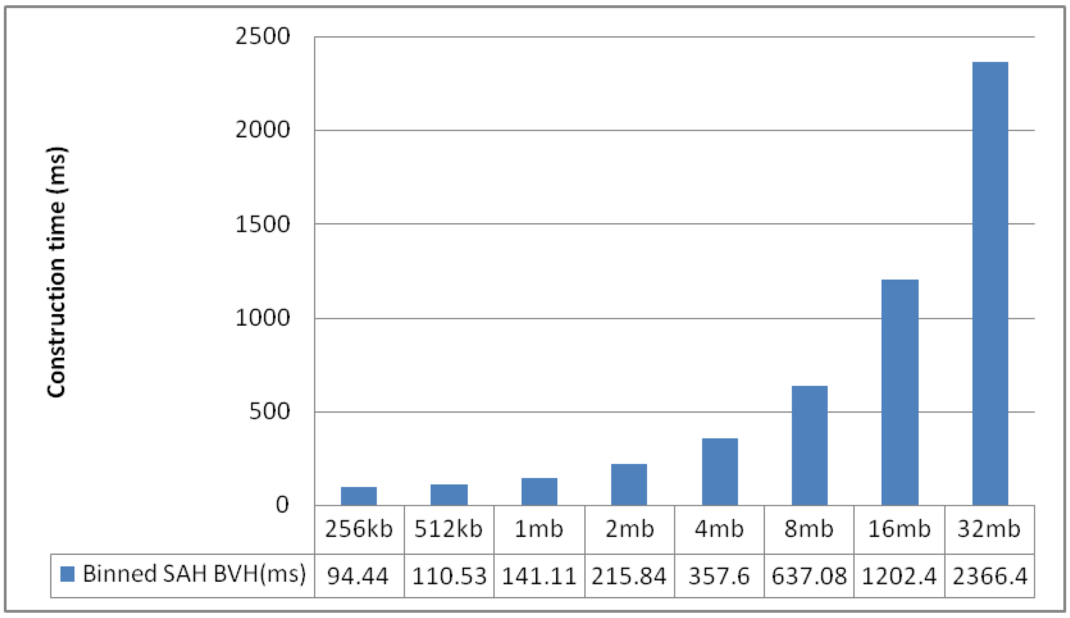
The evaluation will start by comparing the construction time of kd-tree, LBVH and binned SAH BVH. As the time consuming of building binned SAH BVH is much higher than the time cost by constructing kd-tree and LBVH, it will be plotted in a separate chart.



	256kb	512kb	1mb	2mb	4mb	8mb	16mb	32mb
🟥 LBVH(ms)	13.25	14.48	18.79	26.71	41.43	62.18	102.98	181.36
🟦 kd-tree(ms)	14.73	17.39	19.97	27.46	43.45	63.4	109.1	193.13

Figure 5.1: Construction time for LBVH and kd-tree

Figure 5.1 illustrates the time cost for constructing BVH and kd-tree based on sorting Morton code. The horizontal axis gives the size of point sets while the vertical axis represents time in millisecond. Each point is defined as a 4D vector in 4 bytes float. Therefore 256kb size of points represents 128×128 number of points. According to the statistics, it can be easily discovered that building a bounding volume hierarchy is cheaper than building a kd-tree in terms of time.



	256kb	512kb	1mb	2mb	4mb	8mb	16mb	32mb
■ Binned SAH BVH(ms)	94.44	110.53	141.11	215.84	357.6	637.08	1202.4	2366.4

Figure 5.2: Construction time for binned SAH BVH

Binned SAH builder can build a middle-quality bounding volume hierarchy. The time consuming is much higher than the previous two algorithms without using surface area heuristic. Just the same as the kd-tree and LBVH construction algorithm, binned SAH accepts a set of random points to represent a set of bounding box as input data and generates a tree hierarchy.

5.2 Memory Bandwidth

The construction time of kd-tree, LBVH and binned SAH BVH has been displayed and compared in the previous section. In this part, effective memory bandwidth of these three algorithms is plotted. We are interested in global memory space, local memory space, shared memory space and DRAM device memory access by the algorithms. The implementation by NIH library take advantage of CUDA global memory space, local memory space and shared memory space to store and load data, this is why these three memory spaces are evaluated. And DRAM usage also will be illustrated so that it will clearly show the memory bandwidth usage between GPU's processors and physical memory space.

Before analyzing each memory space, it is necessary to know how many kernels are launched in each algorithm. And for each single kernel, does it access global memory space or shared memory space or local memory space. These are given in Table 5.1. In the test, the kernel which does not access to a specific memory space will not be considered in the statistic.

	Kd-tree	LBVH	Binned SAH BVH
Launched kernel	44	44	34
Access global memory	40	40	30
Access local memory	44	44	33
Access shared memory	34	34	18
Access DRAM	44	44	34

Table 5.1: A statistics of how many kernels access different memory space

First column shows how many kernels are launched in total when the application is running. The rest of four columns give how many launched kernels request for a specific memory space access. These values are used to calculate the average memory throughput for each memory space.

5.2.1 Global Memory Access

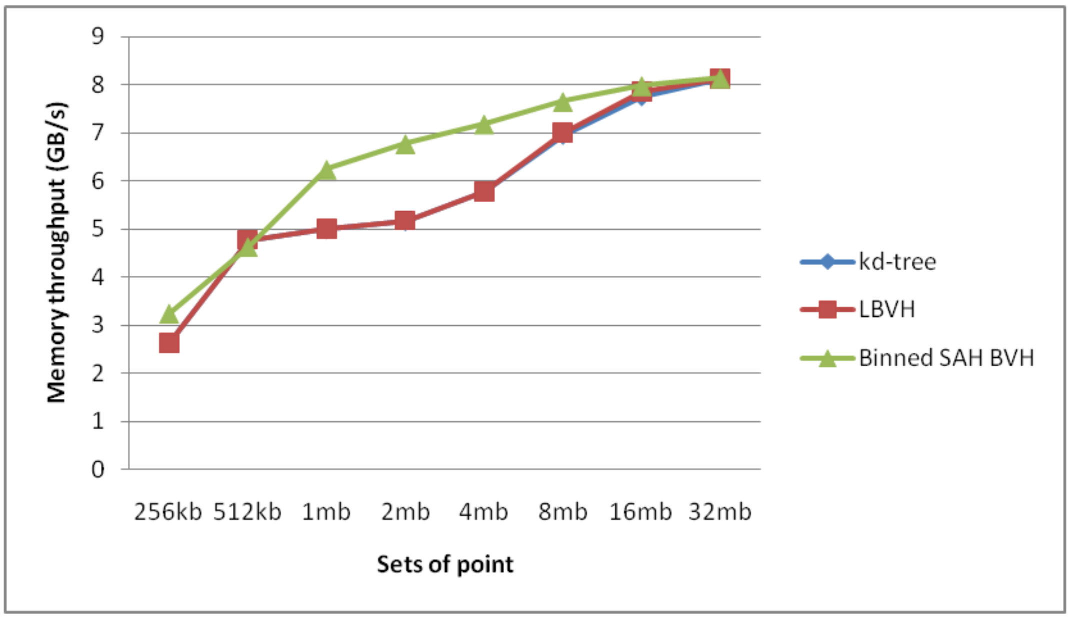


Figure 5.3: Global memory throughput

Figure 5.3 illustrates the global memory throughput for kd-tree, LBVH and binned SAH BVH. The y-axis represents the effective bandwidth in GB per second and the x-axis gives the size of point sets. From the chart, it clearly indicates that the global memory bandwidth usage of kd-tree and LBVH construction is almost the same. Whereas the binned SAH BVH has a higher memory throughput when using the point sets whose size is greater than 512KB and less than 16MB. As global memory loads/stores go through caches, the throughput will be directly affected by cache hit rate. This will be discussed in the next evaluation section. Access global memory is very expensive because it is a off-chip memory. As global memory is the most common used memory space when kernel is being executed, low global memory throughput means high latency on memory access. Binned SAH BVH has a better utilizing of global memory bandwidth than other two algorithms.

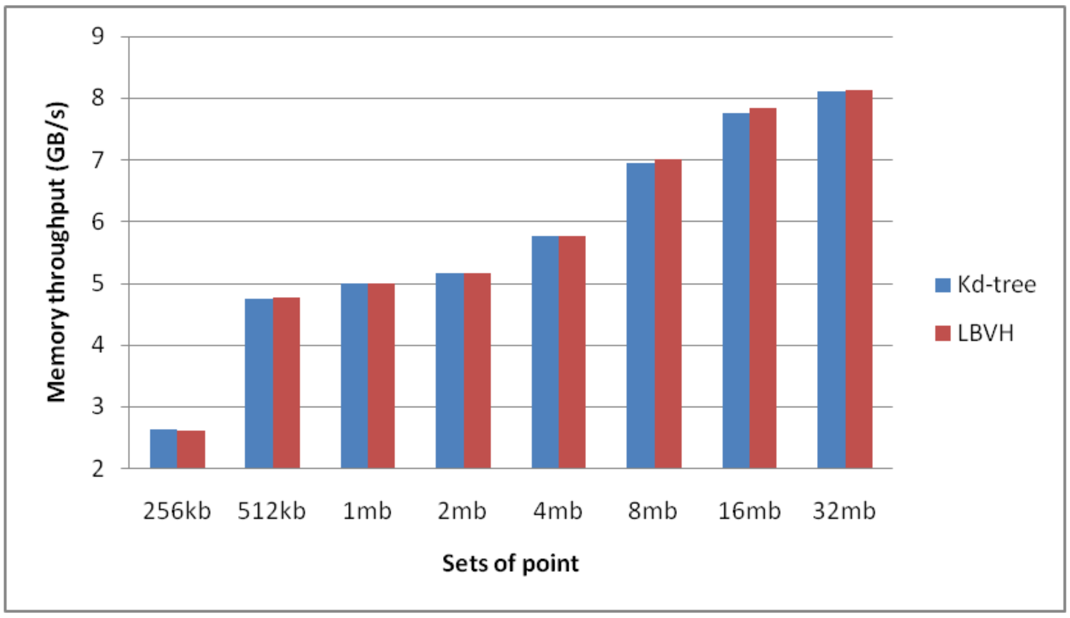


Figure 5.4: Global memory throughput of kd-tree and LBVH construction

A global memory throughput comparing chart for building kd-tree and LBVH are given as well. This two algorithms use the same Morton code based description to order points, and building a BVH costs less time than building a kd-tree. After comparing the global memory bandwidth usage of these two process, we can see from the chart that BVH construction has a tiny advantage on utilizing global memory bandwidth. This can affect the whole construction process because of the latency when access to global memory.

5.2.2 Local Memory Access

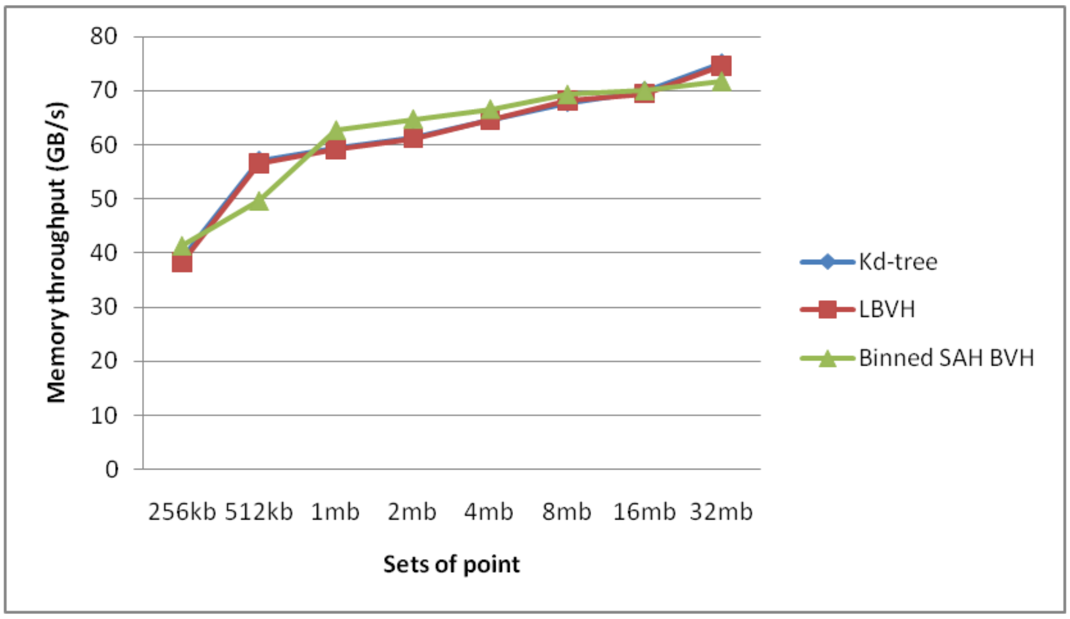


Figure 5.5: Local memory throughput

Access to local memory is as expensive as access to global memory. This because local memory is off-chip memory space. The local memory throughput of three construction algorithms appears to be similar given a set of points. Local memory is used only to hold automatic variables which is done by nvcc compiler. Therefore a further analysis will not be given.

5.2.3 Shared Memory Access

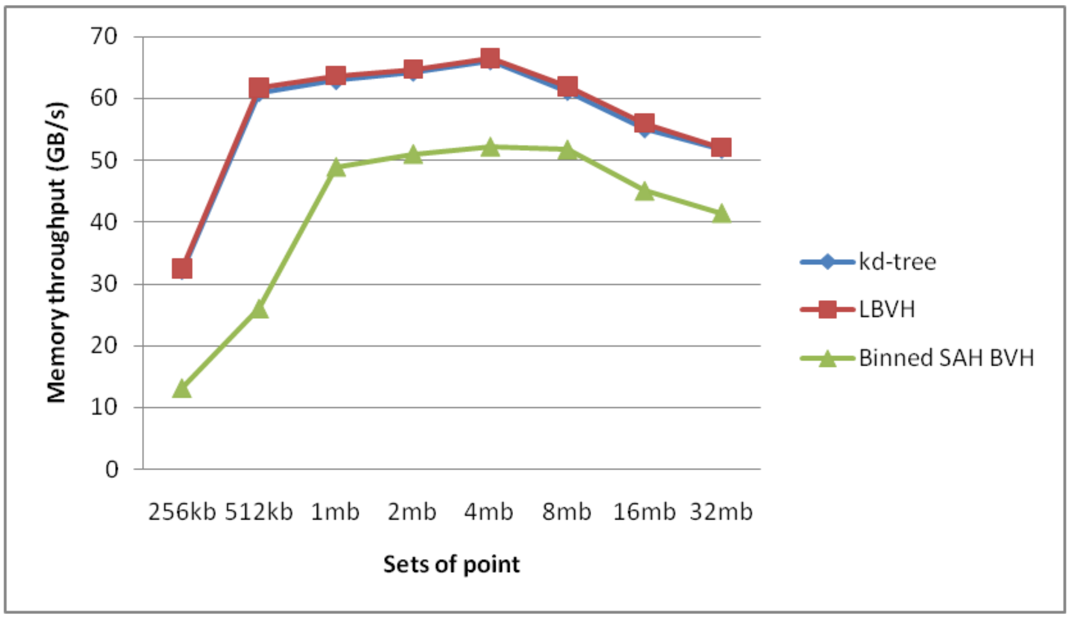


Figure 5.6: Shared memory throughput

Shared memory is the on-chip memory space which is much faster than global memory or local memory. Shared memory has low latency which means utilizing shared memory can make high efficiency. Just like the situation in global memory space, the shared memory throughput of building kd-tree and LBVH is similar. However building binned SAH BVH seems to has a lower shared memory throughput. This can be caused by bank conflicts in shared memory that multiple addresses of a memory request map to the same memory bank.

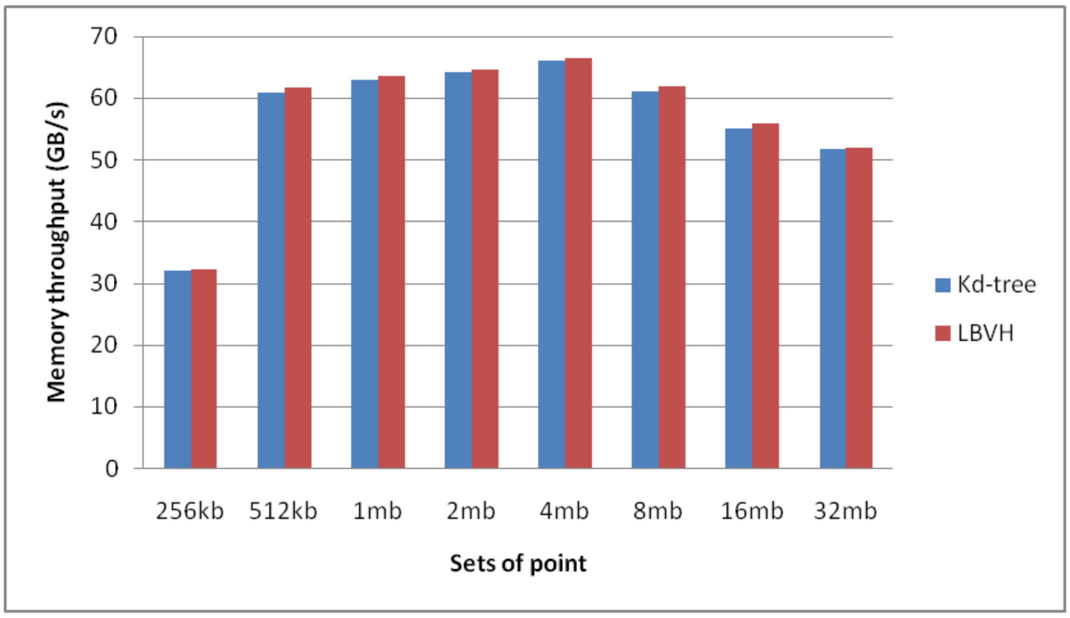


Figure 5.7: Shared memory throughput of kd-tree and LBVH construction

A further comparison of building kd-tree and LBVH is given by Figure 5.7. In most case, the shared memory throughput of building the BVH is higher than the one of building a kd-tree. This can be a factor that why the time consumption of BVH construction is cheaper than building a kd-tree.

5.2.4 Device Memory Access

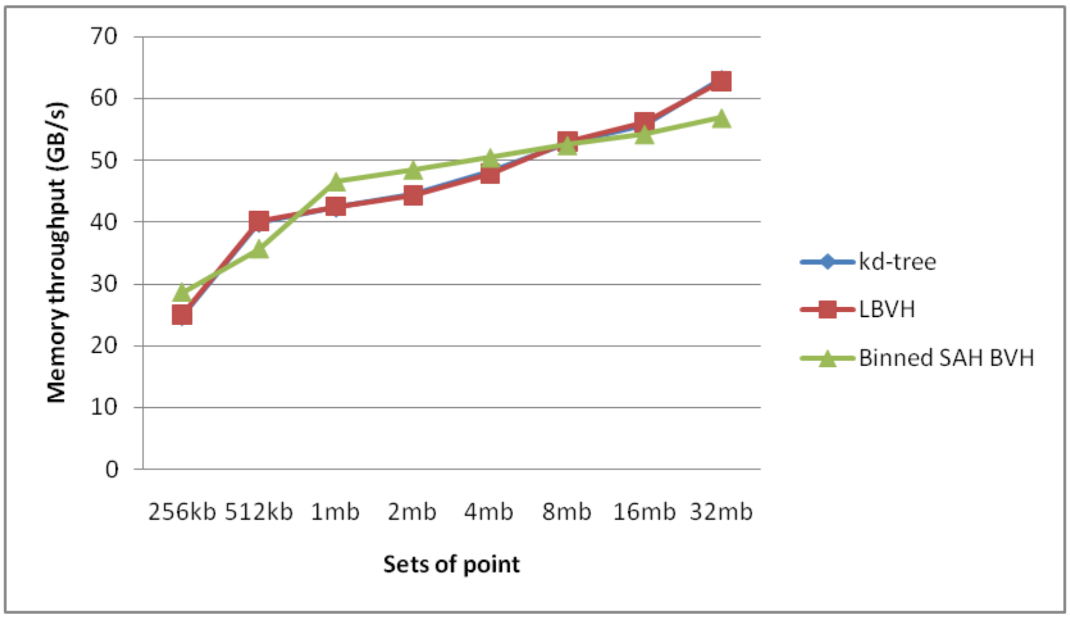


Figure 5.8: Device memory throughput

The device memory is the physical memory space different from global space. Global memory space is virtual address space whose data will be finally mapped to DRAM. DRAM throughput is solely calculated based upon DRAM access without any cache effects. The device memory throughput of three algorithms are close and performs a same increasing tendency.

5.3 Cache Performance

In CUDA memory space, there are two memory go through the cache, global and local memory. According to NVIDIA's official documentation, global memory access for devices of compute capability 3.x are cached in L2, however L1 cache is for local memory only. Therefore, two cache metrics are evaluated including L1 cache local hit rate and L2 cache hit rate for all read requests from L1 cache. The latter is the hit rate for both global memory access and read requests for local memory after L1 misses.

Cache performance requires a kernel requested corresponding memory space access.

This can be derived from Table 5.1. Similar to the memory throughput calculation, the average cache hit rates at L1 and L2 are given in this part. The x-axis represents the size of point set and y-axis shows the cache hit rate in percentage.

subsectionL1 Cache Local Hit Rate

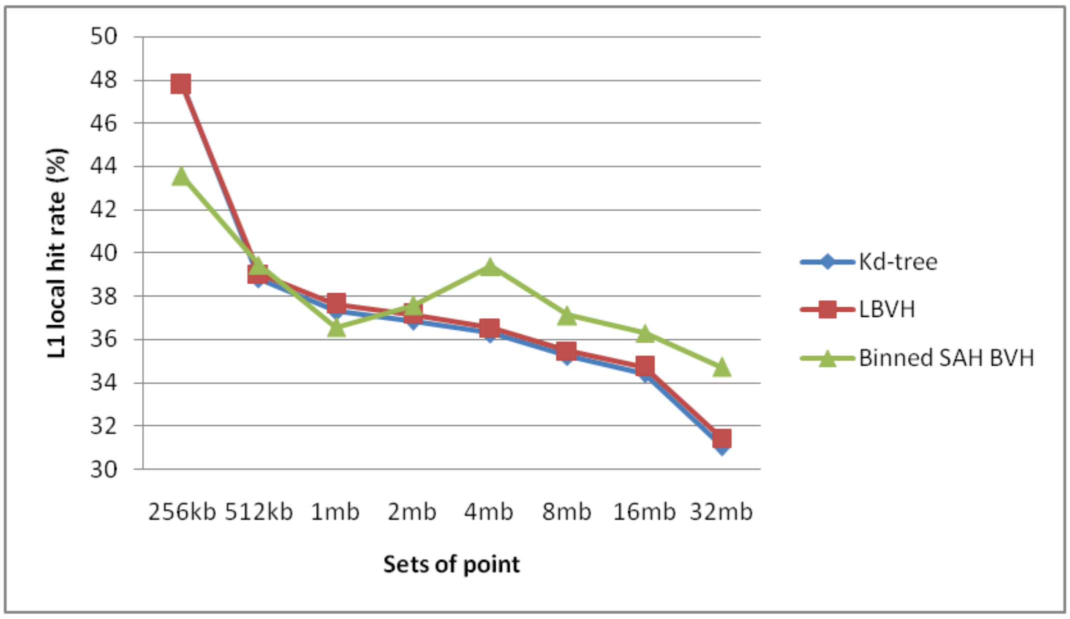


Figure 5.9: L1 cache local hit rate

For CUDA device of compute capability 3.x, only local memory is cached at L1. Local memory is used to hold automatic variables when the compiler determines that there is insufficient register space to hold the variable. How a L1 cache performance affects the whole construction process is not very clearly in this research. So just roughly comparing the L1 local hit rate for three construction algorithms. LBVH has a little bit higher hit rate than kd-tree.SAH BVH has a relative higher L1 cache hit rate when using big size of points. The tendency is very interesting, the larger the set of point is, the less L1 local hit rate the algorithm has.

5.3.1 L2 Cache Hit Rate

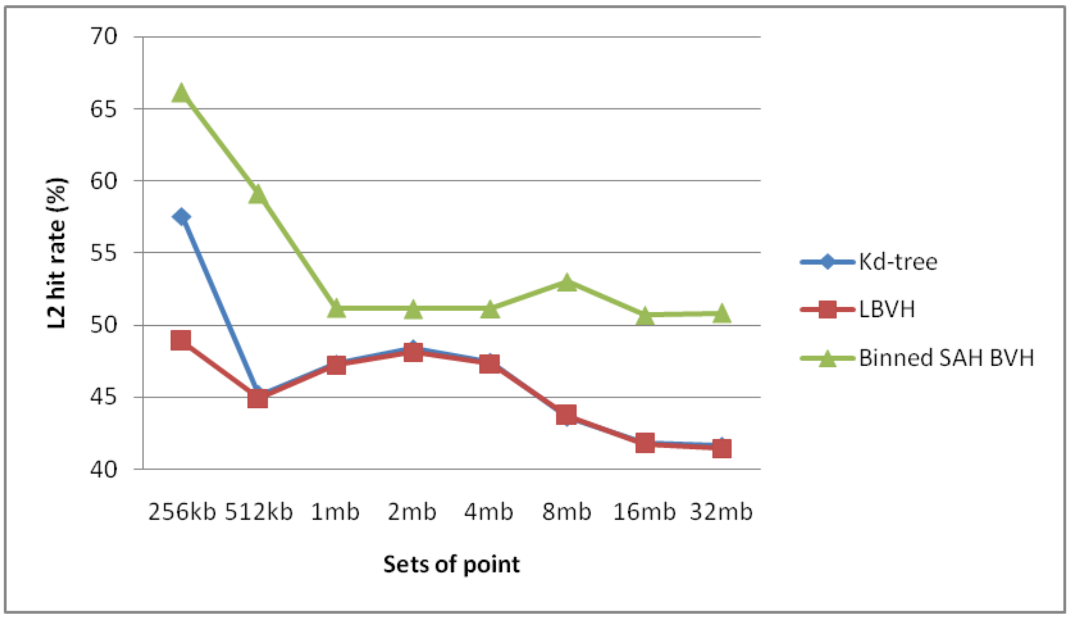


Figure 5.10: L2 cache hit rate

Global memory access will be cached in L2 for CUDA device of compute 3.x. Hence, the L2 hit rate will affect the global memory throughput directly. A higher L2 cache hit rate will result in a low latency of memory access. My guess is LBVH construction may have higher L2 cache hit rate than building kd-tree. The fact is that the L2 cache hit rate for kd-tree building is higher than building bounding volume hierarchy. The reason is possibly because both global memory and local memory access are cached in L2. However, local memory access is cached in L1 as well. So the lower cache hit rate in L1 for kd-tree in the previous test may cause the construction process of kd-tree will fetch more data in L2 cache which increases its cache hit rate in L2. The BVH builder using binned SAH has a much higher L2 cache hit rate that the rest of two builder which can explain its higher global memory throughput.

5.4 Branch Efficiency

Branch efficiency is a measure of how many branches diverged. One hundred percentage means no branch diverged whereas a branch diverges the warp thread will result in low efficiency on execution. In this scenario, The branch efficiency of the major kernels for each construction algorithms are tested.

For kd-tree and LBVH construction, the only different kernel of these two implementation is the split_kernel. They are compared in terms of how many branches happened, how many branch diverged and the branch efficiency where

$$BranchEfficiency = (Branch - nondivergedBranch)/Branch \tag{5.1}$$

For binned SAH BVH, a separate analysis is made for update_bins_kernel, sah_split_-kernel, setup_leaves_kernel and distribute_objects_kernel. These four kernels correspond to the building steps introduced in the design chapter.

5.4.1 Branch efficiency for building kd-tree and LBVH

We compare the split kernel for building kd-tree and LBVH. Other kernels used in the algorithms are pre-defined in thrust library which has the same branch efficiency regardless how large the point set is. Therefore we only compare the split kernel to show the distinction between kd-tree builder and LBVH builder.

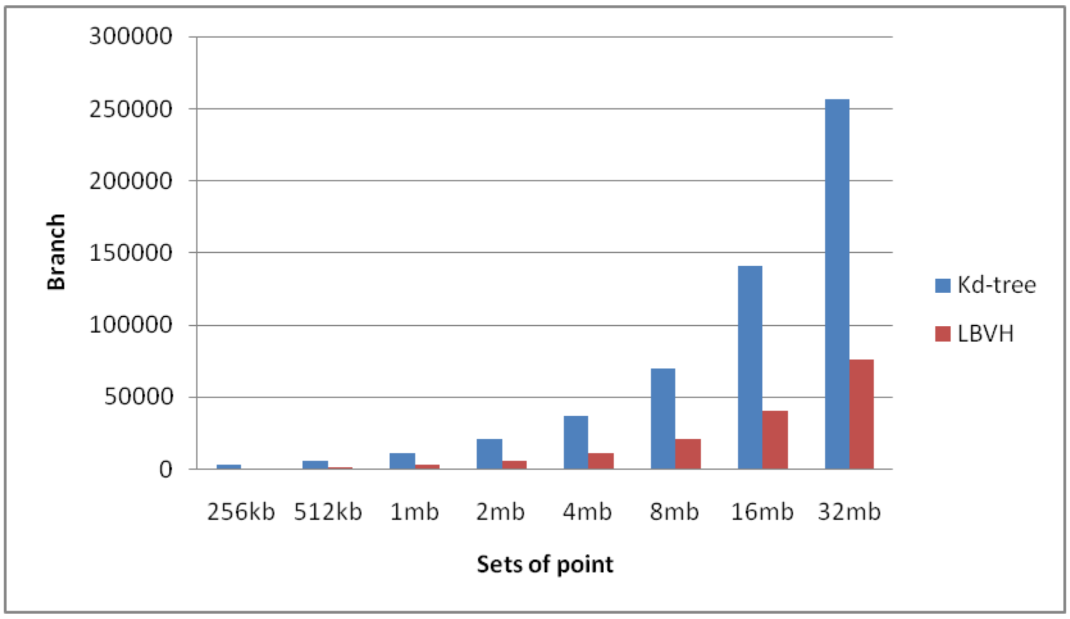


Figure 5.11: Number of branch

Before comparing the branch efficiency, we first look at the number of branch happened in both construction algorithm for different point sets. The result chart shows that two to three times branch happens in kd-tree construction. Which means more flow control instructions are used in building kd-tree.

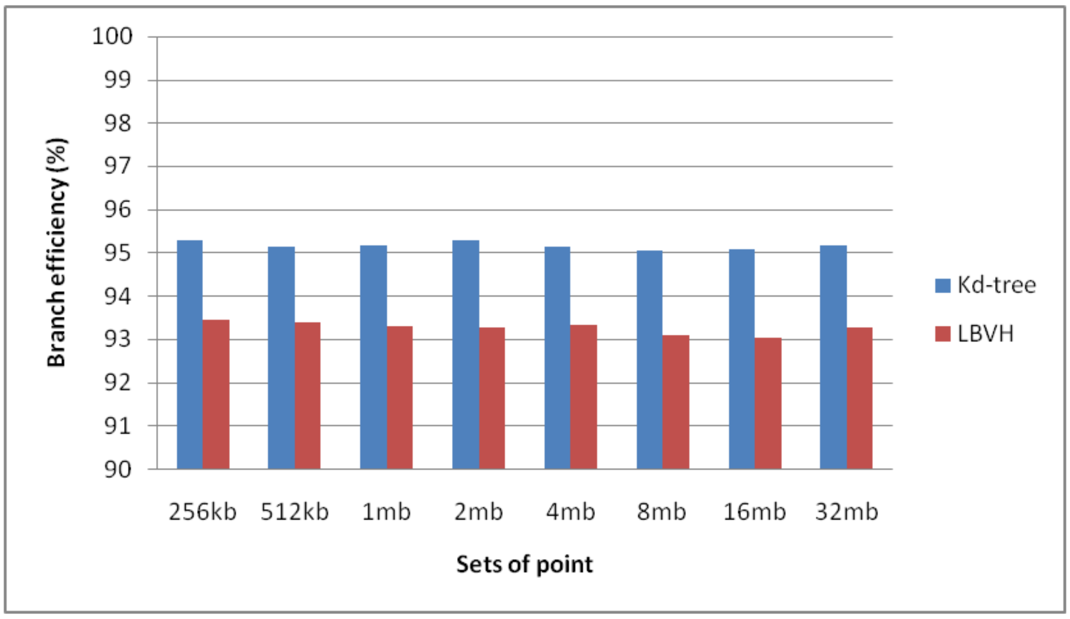


Figure 5.12: Branch efficiency

The branch efficiency comparison is plotted in Figure 5.12. Kd-tree construction exhibits higher branch efficiency, however requires more flow control instruction. In contrast, LBVH construction has much lower branch required along with about 2 percentage less branch efficiency in average.

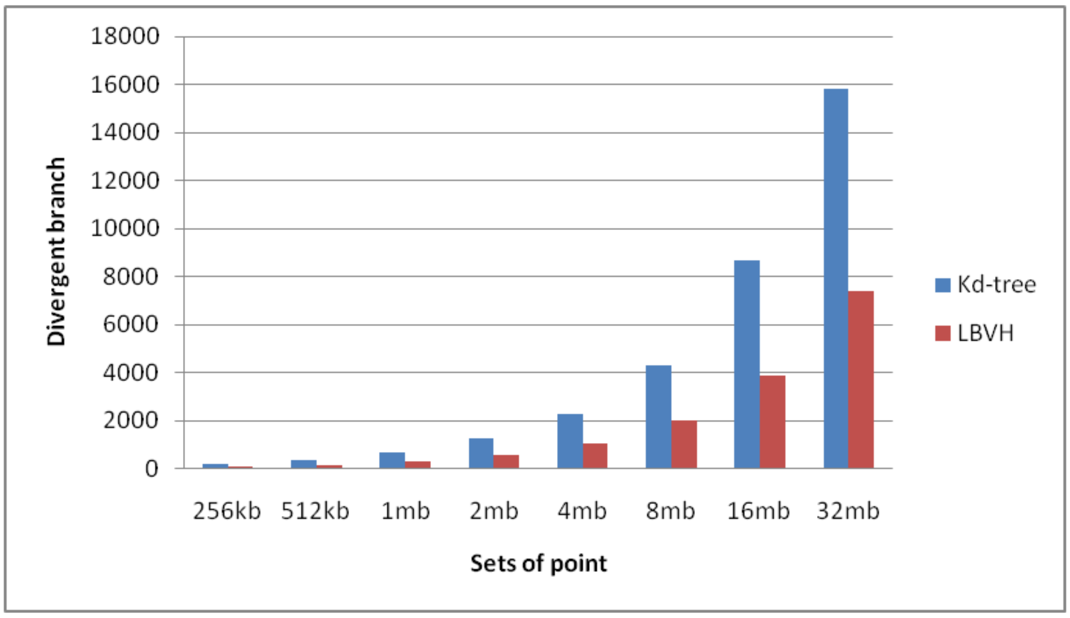


Figure 5.13: Divergent branch

Although kd-tree construction performs a higher branch efficiency than LBVH, too many branch required in building kd-tree directly leading to more divergent branch occurs when the application is running. This can significantly affect the instruction throughput by causing threads of the same warp to diverge. Followed by SIMD architecture, threads in the warp will be diverged to different execution paths. All threads in the same warp converge back to the same execution path until all the execution paths have completed which cause high instruction latency.

5.4.2 Branch efficiency for building binned SAH BVH

The situation is much more complicated in building binned SAH BVH. Four self-defined kernels are used to implement the algorithm including update_bin_kernel, sah_split_-kernel, setup_leaves_kernel and distribute_kernel.

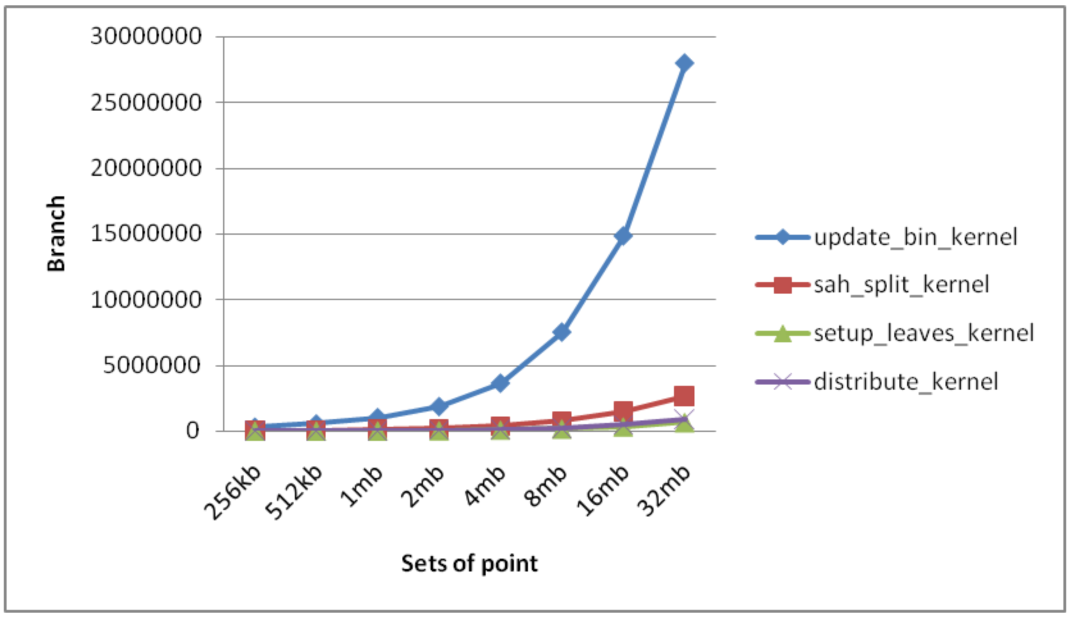


Figure 5.14: Branch comparison for major kernels of binned SAH BVH construction

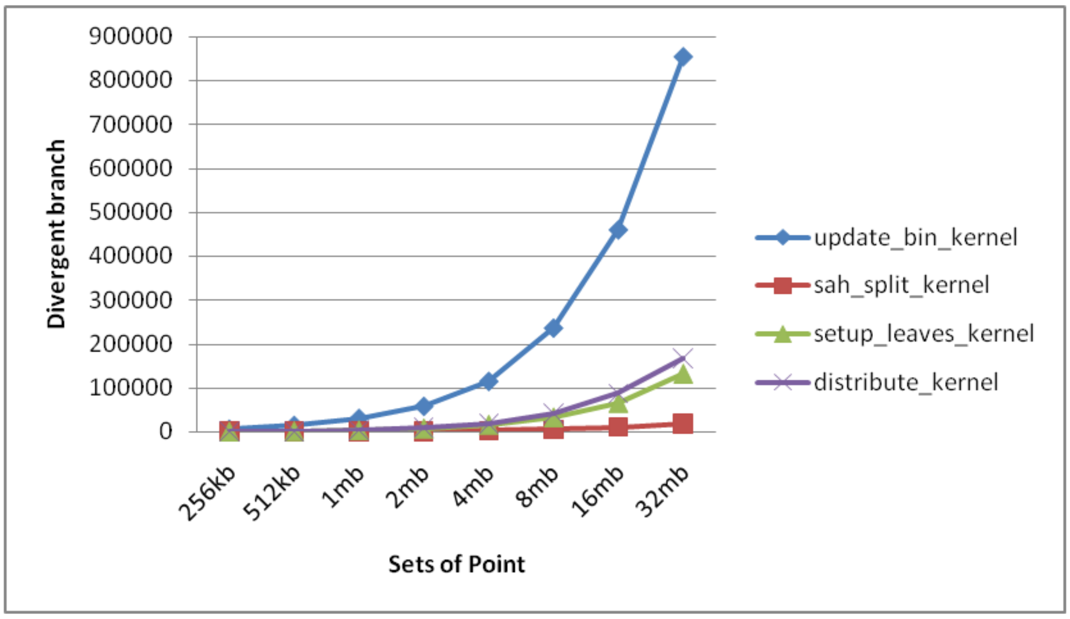


Figure 5.15: Divergent branch comparison for major kernels of binned SAH BVH construction

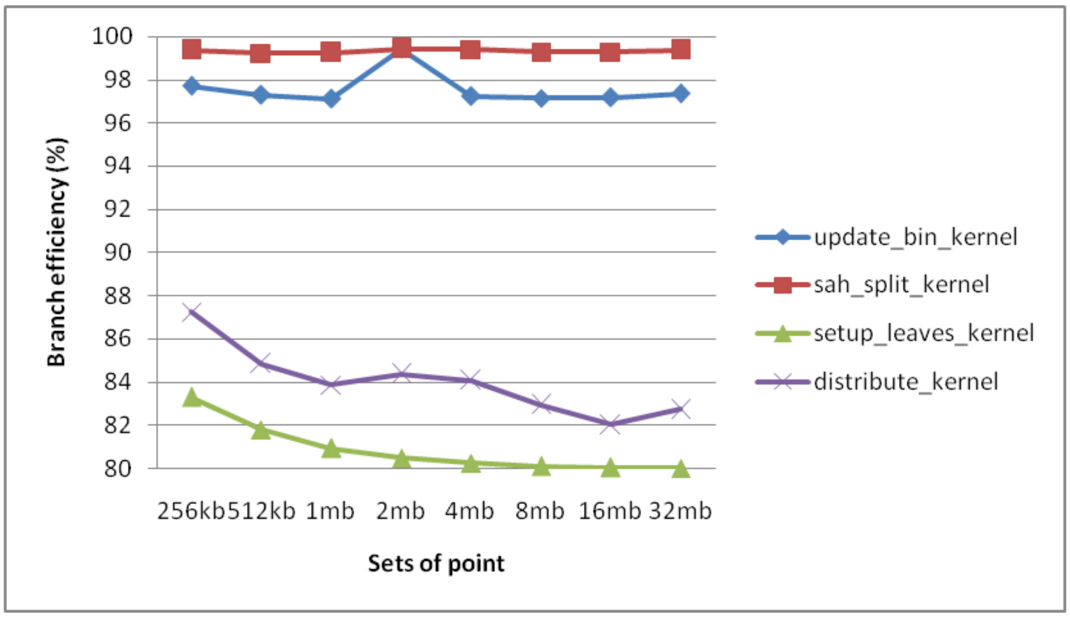


Figure 5.16: Branch efficiency comparison for major kernels of binned SAH BVH construction

Figure 5.14, Figure 5.15, and Figure 5.16 illustrate the branch, divergent branch and branch efficiency comparison for four functional kernels used to build SAH BVH. The binned SAH based BVH construction algorithm exhibits very high flow-control instructions usage according to the diagrams, which leads to more frequent threads divergent. This may be one of the factors that the algorithm has more expensive time cost.

Chapter 6

Conclusion and Future Works

We have compared the construction algorithms for building kd-tree, LBVH and binned SAH BVH in terms of construction time, memory bandwidth usage, cache performance and branch efficiency. The LBVH construction process exhibits a cheaper time consumption than building a kd-tree based on the same Morton code primitives description. We have explored that LBVH construction has higher memory throughput both in global memory space and shared memory space which may lead to a lower memory latency. This could be one of the reasons why building a BVH is faster than building a kd-tree.

The cache hit rate test gives that kd-tree has a lower L1 cache local hit rate while its L2 cache hit rate is higher than LBVH during tree building. As mentioned before, for CUDA compute 3.x device, only global memory access is cached in L2. However the higher L2 cache rate cannot explain why LBVH take a better advantage of global memory bandwidth. But a guess is made which points that both L1 cache for local memory access and L2 cache for global memory access affect the performance of memory throughput. This makes the situation much more complicated to analysis.

Kd-tree construction algorithm performs a very high flow-control instructions dependency. This directly results in more branch diverging happens even building kd-tree has a higher branch efficiency.

For the most of the tests, building a binned SAH BVH is not concluded with kd-tree and BVH. Because binned SAH BVH exhibits a distinct performance for some metrics such as branching. Limited to the codebase, we cannot find a high quality

SAH builder for kd-tree which can be compared to SAH BVH construction. But SAH BVH construction does have a better performance on global memory throughput and cache hit rate both in L1 and L2.

There are two unfinished tasks left. One of them is to compare SAH kd-tree construction algorithm with SAH BVH. This is limited by lack of the code resources. The other thing is to test the construction algorithm in a dynamic scene. A ray tracer is required to achieve this and the rebuilding and refitting comparison for these acceleration data structures will be a valuable evidence to guide which approach should be chose in real-time ray tracing.

Bibliography

[1] T. Akenine-Moller, T. Moller, and E. Haines, *Real-time rendering.* AK Peters, Ltd., 2002.

[2] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 37–45, ACM, 1968.

[3] A. Y. Chang, "A survey of geometric data structures for ray tracing," *Polytechnic University (New York), Tech. Rep. TR-CIS-2001-06*, 2001.

[4] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, 1975.

[5] T. Whitted, "An improved illumination model for shaded display," in *ACM SIGGRAPH 2005 Courses*, p. 4, ACM, 2005.

[6] I. Wald, *Realtime ray tracing and interactive global illumination.* PhD thesis, Universität Saarbrücken, 2004.

[7] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 703–712, 2002.

[8] K. C. Schreck, "Persistence of vision," 2011.

[9] C. Kolb and R. Bogart, "Rayshade 4.0," *Available from princeton. edu (128.112. 128.1)*, 1991.

[10] E. Hoines, “A proposal for standard graphics environments,” *Computer Graphics and Applications, IEEE*, vol. 7, no. 11, pp. 3–5, 1987.

[11] T. Ullmann, T. Preidel, and B. Bruderlin, “Efficient sampling of textured scenes in vertex tracing,”

[12] B. Walter, G. Drettakis, and S. Parker, “Interactive rendering using the render cache,” in *Rendering techniques 99*, pp. 19–30, Springer, 1999.

[13] E. A. Haines and D. P. Greenberg, “The light buffer: A shadow-testing accelerator,” *Computer Graphics and Applications, IEEE*, vol. 6, no. 9, pp. 6–16, 1986.

[14] H. Weghorst, G. Hooper, and D. P. Greenberg, “Improved computational methods for ray tracing,” *ACM Transactions on Graphics (TOG)*, vol. 3, no. 1, pp. 52–69, 1984.

[15] T. L. Kay and J. T. Kajiya, “Ray tracing complex scenes,” in *ACM SIGGRAPH Computer Graphics*, vol. 20, pp. 269–278, ACM, 1986.

[16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, “Fast bvh construction on gpus,” in *Computer Graphics Forum*, vol. 28, pp. 375–384, Wiley Online Library, 2009.

[17] J. D. MacDonald and K. S. Booth, “Heuristics for ray tracing using space subdivision,” *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990.

[18] 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*, pp. 87–95, Eurographics Association, 2010.

[19] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[20] M. R. Kaplan, “The use of spatial coherence in ray tracing,” *Techniques for Computer Graphics*, pp. 173–193, 1987.

[21] H. Samet, *The design and analysis of spatial data structures*, vol. 199. Addison-Wesley Reading, MA, 1990.

[22] B. Arnaldi, T. Priol, and K. Bouatouch, “A new space subdivision method for ray tracing csg modelled scenes,” *The Visual Computer*, vol. 3, no. 2, pp. 98–108, 1987.

[23] J. Goldsmith and J. Salmon, “Automatic creation of object hierarchies for ray tracing,” *Computer Graphics and Applications, IEEE*, vol. 7, no. 5, pp. 14–20, 1987.

[24] K. Subramanian and D. Fussel, *A search structure based on kd trees for efficient ray tracing.* PhD thesis, PhD thesis, The University of Texas at Austin, 1990.

[25] K. Zhou, Q. Hou, R. Wang, and B. Guo, “Real-time kd-tree construction on graphics hardware,” in *ACM Transactions on Graphics (TOG)*, vol. 27, p. 126, ACM, 2008.

[26] M. Shevtsov, A. Soupikov, and A. Kapustin, “Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes,” in *Computer Graphics Forum*, vol. 26, pp. 395–404, Wiley Online Library, 2007.

[27] NVIDIA. http://www.nvidia.com/object/cuda_home_new.html.

[28] NVIDIA, “Cuda toolkit documentation.” http://docs.nvidia.com/cuda/index.html.

[29] K. Garanzha and C. Loop, “Fast ray sorting and breadth-first packet traversal for gpu ray tracing,” in *Computer Graphics Forum*, vol. 29, pp. 289–298, Wiley Online Library, 2010.

[30] NVIDIA. http://www.nvidia.com/object/optix.html.

[31] M. Zlatuška and V. Havran, “Ray tracing on a gpu with cuda–comparative study of three algorithms,” *Informe Técnico, Czech Technical University in Prague Faculty of Electrical Engineering*, 2009.

[32] NIH. https://code.google.com/p/slash-sandbox/wiki/NIH.

[33] A. Resios and V. Holdermans, “Gpu performance prediction using parametrized models,” *Master’s thesis, Utrecht University, The Netherlands*, 2011.

[34] D. Wodniok, A. Schulz, S. Widmer, and M. Goesele, “Analysis of cache behavior and performance of different bvh memory layouts for tracing incoherent rays.,” in *EGPGV*, pp. 57–64, 2013.

[35] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, 2005.

[36] P. Forall, “Thinking parallel.” https://developer.nvidia.com/content/thinking-parallel-part-iii-tree-construction-gpu.

[37] A. Sohn and Y. Kodama, “Load balanced parallel radix sort,” in *Proceedings of the 12th international conference on Supercomputing*, pp. 305–312, ACM, 1998.

[38] 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*, pp. 59–64, ACM, 2011.