Build and Travel KD-Tree with CUDA

Meng Zhou
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
MengZhou@cmail.carleton.ca

April 24, 2014

1 Introduction

Ray tracing is an important and widely used tool in computer graphic. Entertainment and game industry have already benefit a lot from ray tracing. However, designers and end-users are forced to use off-line ray tracing tools for a long time due to the high computation load.

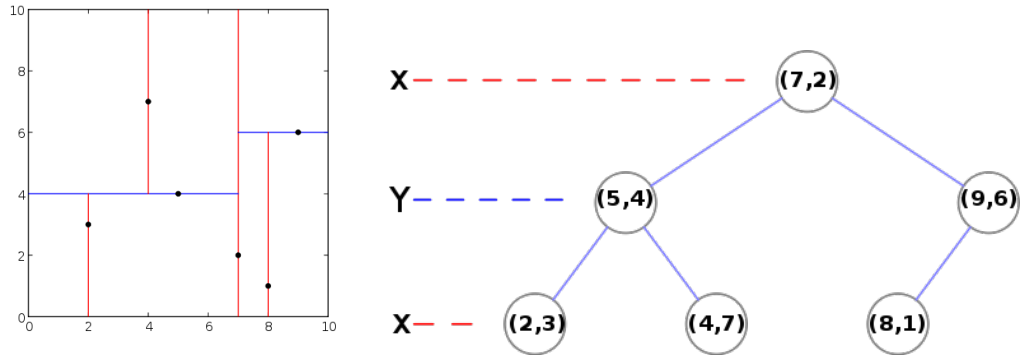
In ray tracing, most of the computation is concentrated on whether hundreds of millions of rays hit objects in a scene. The naive algorithm calculate intersection between every ray and every triangle in the scene, which yields $O(nm)$ computation time with n rays and m triangles. Inspired by binary search in one dimension space, one of the most popular and efficient improvement is partition the scene and build a tree to represent it. With this tree we have algorithm with $O(nlog(m))$. Travelling kd-tree with n rays is a natural parallel problem with each ray independent to each other. Thus if we have a kd-tree for a scene, the intersection problem can be calculated with $O(log(m))$ on n processors, which is an optimal algorithm with $nm/log(m)$ accelerator. In practice, the accelerator is much lower due to hardware limitation, which we will discuss later.

In this case kd-tree(k-dimensional tree)is developed with many other spacial partition algorithms. Despite animation with dynamic scene, kd-tree has been proved to be the most efficient data structure for static scene[2][16]. There are two tasks for us to use kd-tree in ray tracing: 1.Build kd-tree with triangle soup; 2.Travel kd-tree to find intersection points. This paper will first review state-of-the-art algorithms to build and travel kd-tree, both serial and parallel. Then implement an algorithm to build kd-tree with CUDA. Finally analyse this algorithm and try to make it faster and more memory efficient.

2 Literature Review

In this section, a brief introduction to kd-tree is first provided. Then some sequential algorithm will be introduced, followed by parallel algorithms, both on building and travelling kd-tree.

A kd-tree is a space-partitioning data structure for organizing points in a k-dimensional space. It is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right



(a) k-d tree decomposition for the point set (2,3), (5,4), (9,6), (4,7), (8,1), (7,2).

(b) The resulting k-d tree.

Figure 1: 2D kd-tree partition

subtree[14]. The hyperplane is perpendicular to axes, which makes kd-tree different from BSP(binary space partition) tree. Figure 4 illustrates a simple example of 2D kd-tree.

2.1 On building kd-tree

In the following, we first give the general algorithm of build a kd-tree[13], which is Algorithm 1. Consider $\mathcal{S}$ made up of N triangles. A kd-tree over $\mathcal{S}$ is a binary tree that recursively subdivides the space covered by S:The root corresponds to the axis-aligned bounding box(AABB) of $\mathcal{S}$;interior nodes represent planes $p_{k,\zeta} : x_k = \zeta$ that recursively subdivide space perpendicular to the coordinate axis;leaf nodes store references to all the triangles overlapping the corresponding voxel. Obviously, the structure of a given kd-tree(where exactly the planes are placed) directly influences how many traversal steps and triangles intersections the ray tracer has to perform. The problem is simple, how to find a "good" plane p to split V.

There are basically two kinds of strategies to choose p:

1. Median splitting[8]. To be specific, consider triangle soup in 3-dimensional space. We choose one of the three coordinates(x, y, z) each time in a circle, once selected we ignore the other two until next level. Each triangle's AABB has two projection on the current coordinate(which is also referred as events or candidates). Each time we need a p, we choose the median coordinate, and split the current triangle set into two subsets, which is the left and right child of the current node on kd-tree. We perform the same operation on the next level until the number of triangle is below threshold. We can run this algorithm in $O(nlogn)$ time, but the traversal performance is bad, because the median split strategy does not take triangle distribution into account.

2. Heuristic search[6][13][10][9]. Instead of split current node from the median point, heuristic algorithm search for the best p in the solution space. We can place p on any place between AABB bounding of current node. The best p gives the lowest travel cost on the kd-tree rooted by current node. The cost function $cost(x)$[3] is defined as follows:

$$cost(x) = C_I + C_L(x)\frac{SA_L(t,x)}{SA(t)} + C_R(x)\frac{SA_R(t,x)}{SA(t)} \tag{1}$$

Algorithm 1 Recursive KD-tree build

```
if Terminate(T, V) then
return new leaf node(T)
function RecBuild(triangles T, voxel V)return node
    if Terminate(T, V) then
        return new leaf node(T)
    end if
    p = FindPlane(T, V) Find a good plane p to split V
    (V_L, V_R) = Split V with p
    T_L = {t ∈ T|(t ∩ V_L) ≠ ∅
    T_R = {t ∈ T|(t ∩ V_R) ≠ ∅
    return new node(p, RecBuild(T_L, V_L), RecBuild(T_R, V_R))
end function

function BuildKDTree(triangles[] T))return root node
    V = B(T){start with full scene}
    return RecBuild(T, V)
end function
```

Where C_I is the cost of traversing the node itself, $C_L(t)$ is the cost of the left child given a split position x and $C_R(x)$ is the cost of the right child given the same split. $SA_L(t,x)$ and $SA_R(t,x)$ are the surface areas of the left and right children respectively. $SA(v)$ is the surface area of the triangle currently being considered for splitting. We can approximate $C_L(x)$ and $C_R(x)$ by just count number of triangles in the left and right sub-tree respectively. This evaluation is also called surface area heuristic(SAH) and widely used not only by kd-tree but also by other spatial partition algorithms such as bounding volume hierarchy(BVH), which has been proved to be faster than kd-tree in dynamic scenes[16][9].

In order to calculate number of triangles line on the left and right side of the plane at each x, we can iterate over all candidates to see if it belongs to the left or the right. This takes$O(n^2)$ time for n candidates. It is obvious that $O(n^2)$ algorithms are impractical except for trivially small n. We can get immediate speed up by sort t first at a cost of $O(nlogn)$. And then sweep t again to calculate $C_L(x)$ and $C_R(x)$ at a cost of $O(n)$. This algorithm results in $O(nlog^2n)$ for the kd-tree as a whole.

Wald has proved that the lower bound for construction of a kd-tree is $O(nlogn)$[13]. To achieve this goal, researchers have posted many solutions.

Wald himself describe an optimal sequential $O(nlogn)$ SAH kd-trss construction algorithm[13] that initially sorts the candidates extents in the thress coordinate axes, performs linear-time sorted-order coordinate sweeps to compute the SAH just like the $O(nlog^2n)$ method while maintains sorted order as the bounding boxes and their constituent geometries are moved and subdivided. It actually maintains 3 sorted array for 3 axes. When p is chosen, it puts left and right child to new sorted array for 3 axes individually, and sort all the candidates that cross p. Since the maximum number of triangles across p is no larger than$\sqrt{n}$ [13][10], sorting them takes $O(\sqrt{n}log\sqrt{n}) < O(n)$, so the overall cost is $O(nlogn)$.

To allow a tradeoff between tree quality and construction speed, Hunt[6] gives an approximation of equation 1. Instead of calculate all the possible p, they calculate only 16 samples, then use piecewise quadratic function approximately find the lowest $cost(x)$ for p.

They proved that the error is bounded.

Leaving the *cost* function alone, Popov play with the construction function on a lower level[10]. Popov's approach is based on Wald's theory but exploit memory usage. They try to use BFS to construct kd-tree instead of DFS, thus yield high local memory visit and memory coherence. But not all of their algorithm runs in BFS. On the lower level of kd-tree they still use DFS. We know that DFS is p-complete problem and thus not easy to be paralleled.

We have talked about build SAH kd-tree sequentially, now we are moving to the parallel situation. First Shevtsov developed a parallel kd-tree construction approach on multi-core CPU[12]. They divide the whole scene into several sets that have almost the same number of triangles, and then process them independently in parallel. Another idea they come up to is the Max-Min binning scheme. Instead of computing the precise SAH, they approximate the SAH at some equally sized bins. But this approach is not suit for GPU because GPU has thousands of threads while cutting the scene in such manner will digress the optimal solution.

Zhou developed the first kd-tree construction runnning entirely on GPU[16]. They classify the nodes into large nodes and small nodes by the number of triangles in the nodes or, on the other hand, level of nodes in the tree. They use median split for large nodes and SAH for small nodes. Also they use tight data structure to achieve high local memory hit. Their result shows that their algorithm is outstanding. However, large nodes in this algorithm may degrade the kd-tree quality if there are too many of them.

Choi[1] and Wu[15] present algorithms on GPU without degrade the quality of kd-tree respectively. Wu use GPU scan tto count triangle number of the child nodes and a bucket-based algorithm to sort the AABBs.

2.2 On travel kd-tree

For the sequential travel algorithm, one can refer to [2]. The idea is simple,The algorithm takes as input a tree and a ray, and searches for the rest primitive in the tree that is intersected by the ray. The tree is traversed starting at the root, and a stack is used as a priority-ordered list of nodes left to visit. Each node on the stack is closer to the ray origin than all nodes below it, and the node currently being traversed is closer than all nodes on the stack. A (tmin; tmax) range limits the part of the ray under consideration to that which intersects the current node.

When an internal node is encountered during the traversal, the(tmin,tmax)range of the ray is classified with respect to the splitting plane of the node. If the range lies entirely to one side of the plane, the traversal simply moves to the appropriate child. If, instead, the range straddles the plane then traversal will continue to the rest child hit by the ray, while the second child is pushed onto the stack along with its appropriate(tmin,tmax)range. In this way the traversal proceeds down the tree, occasionally pushing items onto the stack, until a leaf node is reached.

If the ray intersects one of the primitives in the leaf within the(tmin,tmax)range, then the closest intersection in the leaf is guaranteed to be the rest intersection along the ray, and the traversal terminates and yields this result. If no intersection is found then we pop a work item consisting of a node and a (tmin, tmax) range from the stack and continue searching. If the stack is empty then there is no intersection along the ray and the search terminates.

The main challenge for parallel traversal kd-tree is eliminate the stack operation. GPU

is not good at stack operations mainly because the local memory is too small. If a stack operation needs to visit global memory, it can be very slow. Fortunately there are already kd-tree traversal algorithms that do not need stack, and the most basic ones are KD-Restart and KD-Backtrack[2].Horn[4] and Popov[11] also provide stack-less kd-tree traversal algorithm.

3 Project Report

This project implements Zhou's Real-time KD-Tree construction algorithm[16] on both CPU and GPU. The following section of this paper will first describe the sequential version of the algorithm and then shows details about how to implement it with CUDA. At the end of this section, there will be a performance comparison between CPU and GPU versions.

3.1 Sequential Algorithm

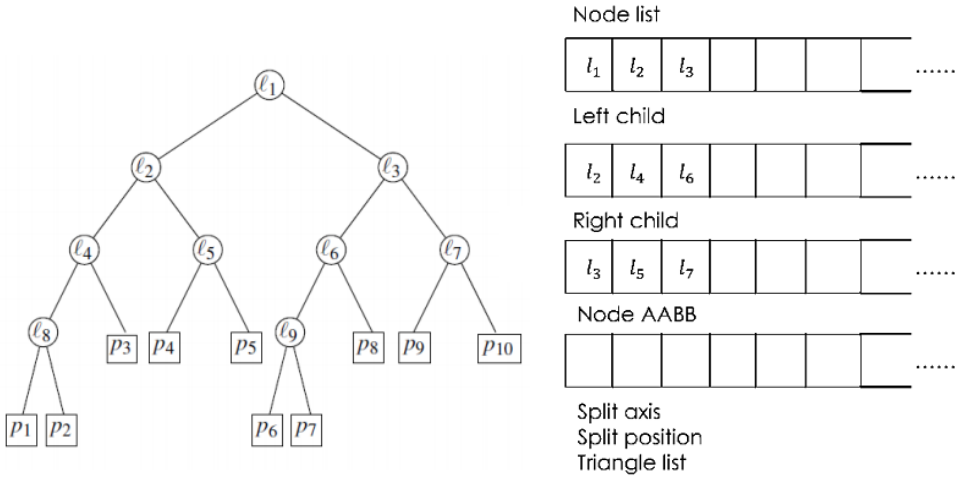
In order to compare the running speed of exactly the same algorithm on CPU and GPU, this paper does not use regular building algorithm for CPU introduced in section 2. The parallel version divide regular recursion building algorithm into two parts: big node stage and small node stage. The consideration is that in large node stage the work for each thread is heavy, however the number of working thread is low. There will be large numbers of stream processors idle in GPU while only few of them are hard-working. So we use mid split for large node stage to save time. In small node stage, there will be much more working threads, they will fit into stream processors and increase hardware utilization.

Algorithm 27 give the algorithm of the whole process. Note compare with the algorithm we give in section 2, we use iteration rather than recursion.

Algorithm 2 Sequential KD-tree construction

```
function BuildTree(triangles:list)
    nodelist ← new list
    activelist ← new list
    smalllist ← new list
    nextlist ← new list
    Create rootnode
    activelist.add(rootnode)
    for each input triangle t do
        Compute AABB for triangle t
    end for
    //large node stage
    while !activelist.empty() do
        nodelist.append(activelist)
        nextlist.clear()
        PROCESSLARGENODES(activelist, smalllist, nextlist)
        Swap nextlist and activelist
    end while
    //small node stage
    PROCESSSMALLNODES(smalllist)
    activelist ← smalllist
```

```
    while !activelist.empty() do
        nodelist.append(activelist)
        nextlist.clear()
        PROCESSSMALLNODES(activelist, smalllist, nextlist)
        Swap nextlist and activelist
    end while
end function
```



(a) Original tree. (b) Corresponding Array.

Figure 2: Array structure for KD-Tree.

The input of the algorithm is a list of triangles. First it calculate AABB of each triangle. The algorithm build KD-Tree in a BFS manner. Every while loop add new level to the tree. The only difference between large node and small node stage is they split parent node in different way.

A very important thing when implement an algorithm is data structure. To make a fair comparison this project use the same list data structure for KD-Tree. All the nodes are stored in several arrays. Instead of store pointers, we record the index of child nodes when creating KD-tree. The information we store for each node is its left and right child index, node AABB, split axis, split position and triangles that belong to the node. We only store triangle list for leaves. See figure 2 for better illustration.

For the large node stage, this project use a simplified version of Zhou's algorithm. This project does not use chunk, though it gives better performance. For each node in *activelist*, we first find the longest axis of its AABB, and mid split this AABB. If the number of triangles in child node is less than pre-set threshold, then put this new node into *smalllist*, otherwise we put this child node in *activelist* and continue doing mid split until all nodes in *activelist* is in *smalllist*.

The idea of using list is the key to eliminate recursion. When we implement sequential version on CPU, we use vector in STL library. There are multiple properties for one node, and two ways can be used to implement this array structure, one is struct of array(SOA) and another is array of struct(AOS). There is no doubt that we should use SOA for CPU. However for GPU, the conclusion cannot be draw so quickly. Of course access the same block of data will be faster within a group of stream processor(SMX for fermi and kepler),

but each thread in a block may access data faraway from each other in the array. This is brought by the tree structure. This project use AOS on both CPU and GPU. We haven't compare SOA and AOS on GPU yet, this probably will give us enhancement in performance.

Algorithm 3 Large node stage of KD-tree construction

```
function PROCESSLARGENODE(in activelist:list out smallllist, nextlist:list)
    for each node i in activelist do
        Mid split node i of the longest axis and generate newnode
        if Count of triangles in newnode larger than THRESHOLD then
            activelist.add(newnode)
        else
            smalllist.add(newnode)
        end if
    end for
end function
```

The small node stage is very similar to the large node stage. Instead of mid split current node, we find the best splitting candidate by calculate SAH of each candidate. The formula we use to calculate is exact formula 1, where $C_L(x)$ is the count of triangle in the left child and $C_R(x)$ is the count of triangle in the right child, $SA_L(t,x)$ is the area of the left child after split, $SA_R(t,x)$ is the area of the right child after split. We choose splitting axis the same way with the large node stage. In this project we choose 10 candidates regularly distributed on the splitting axis. This process is done until the number of triangles in current node is less than some pre-set threshold. See illustration in figure 3.

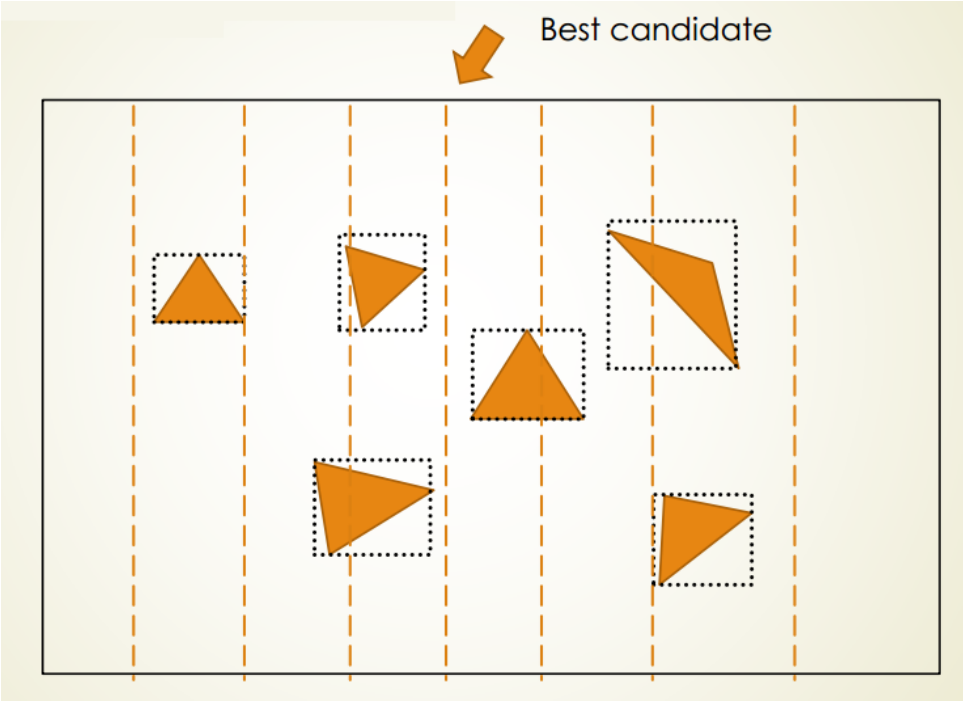


Figure 3: Small node stage of constructing KD-Tree.

Algorithm 4 Small node stage of KD-tree construction

```
function PROCESSSMALLNODE(in activelist:list)
    for each node i in activelist do
        for each splitting candidate j on the longest axis do
            Calculate SAH for j
        end for
        Split node i by j and generate newnode
        if Count of triangles in newnode larger than THRESHOLD then
            activelist.add(newnode)
        end if
    end for
end function
```

3.2 Parallel Algorithm

The parallel algorithm this project use is pretty straight forward. The basic idea is when we processing node split for *activelist*, we make each of the element in the list parallel. This is possible because we construct KD-Tree in BFS manner. Each node in the same level is responsible for different area of the space, so there is no conflict when we split them at the same time. We could also calculate AABB of each triangle in parallel without any conflict. We only gives the parallel for the large node stage.

Algorithm 5 Large node stage of KD-tree construction

```
function PROCESSLARGENODE(in activelist:list out smallllist, nextlist:list)
    for each node i in activelist in parallel do
        Mid split node i of the longest axis and generate newnode
        if Count of triangles in newnode larger than THRESHOLD then
            activelist.add(newnode)
        else
            smalllist.add(newnode)
        end if
    end for
end function
```

However for the data structure, there will be conflict when more than one thread try to add *newnode* to the array. Error may occur when threads read and update index of the array. The solution this project use is atomic operator provided by CUDA. The atomic operator make sure no read-write conflict happen when updating the index of the array. However the atomic operator will harm performance. One possible solution to this problem is like Ramin Azarmehr's solution to his line detection project. We store only piece of array in shared memory and combine them after synchronization. The performance may be better, but this only limits the conflict in block level instead of eliminate them. The following code shows how we use atomic add.

```
int push_back(T*d, unsighed int* ptr, T& t){
        int i = atomicAdd(ptr, 1);
        d[i] = t;
        return i;
```

}

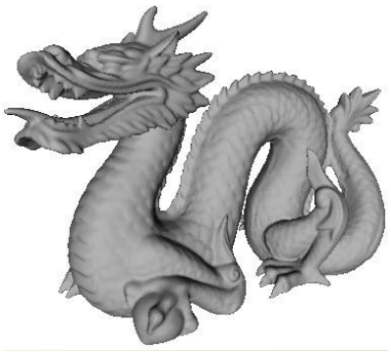
The purpose of building KD-Tree in ray tracing is quickly figure out ray-scene intersection points. Given a large set of rays, our GPU KD-Tree algorithm can calculate intersection fully parallel. The idea of accelerating intersection is branch cutting. Given a ray and a KD-Tree, we travel the tree in DFS manner. At each node we calculate whether the ray intersect the node's left child bounding box and right child bounding box. If it's not we does not have to travel the branch. Triangle-ray intersection is only necessary in leaf node.

Algorithm 6 Large node stage of KD-tree construction

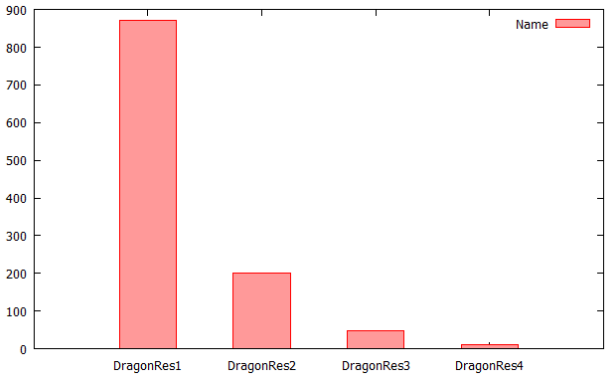
```
function INTERSECTION(in:RayList:list Root:KD-Tree out:intersection:list)
    for each ray in RayList in parallel do
        Travel root in DFS manner
        ...
        If ray intersect bounding box of child
        search child
        EndIf
        ...
    end for
end function
```

3.3 Experiment Results

The following section gives the experiment results. First we have to prove our GPU algorithm has the same result with the CPU version. Then we compare their construction and travelling time. All the experiments are done on Intel i5-4670k with 16GB memory and Nvidia GeForce GTX 780Ti. The test scenes we use ara a dragon with different detail levels.



(a) Test scene visualization



(b) Number of triangles

Figure 4: Test scene

3.3.1 Correctness Proof

Given a scene and a ray, we first give the proof that our program give the right intersection point. We implement brute force algorithm that compare the ray with every triangles in the

scene and use the result as the correct answer. Then use CPU KD-Tree and GPU KD-Tree to calculate intersection points. We select the closest point and see if it's the same. In the experiment the program use random ray to do this test. The result is as follows:

$Rayfirstintersecttriangle$

$3708distance : 0.56725Time : 2$

$KD - Treereport : Rayfirstintersecttriangle$

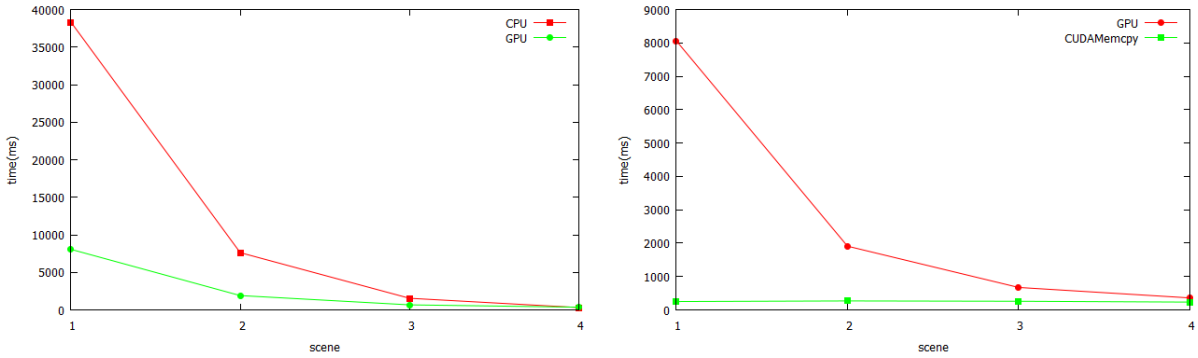
$3708withdistance0.56725Time : 1$

$GPUKD - Treereport : Rayfirstintersecttriangle$

$3708withdistance0.56725Time : 6$

This shows our KD-Trees calculate the correct result. The CPU KD-Tree has the best performance in the single thread test.

3.3.2 Construction KD-Tree Performance



(a) CPU vs GPU performance

(b) CUDAMemcpy time

Figure 5: Test scene

Figure 5 shows the performance of CPU and GPU KD-Tree construction. From the left one we can see that within large scene, GPU KD-Tree has better performance. The larger the scene goes, the performance become better. In small scene like scene 4 with only 11k triangles, CPU beats GPU in the experiment. This comes from mainly two reasons: First as the result in the correctness proof section, single thread on GPU takes more time than GPU. When the number of thread overcome this single thread shortcoming, GPU beats CPU. Second data for GPU computing is initially in the main memory. Because the latency of PCI-E bus, GPU is sitting idle waiting for data.

3.3.3 Travelling KD-Tree Performance

Figure 6 shows calculation time of 50000 random rays intersecting with test scene. When the size of the scene grows, both CPU and GPU running time grows in nearly linear way, which makes the acceleration ratio of GPU algorithm stable around 9. The reason for this is that both of CPU and GPU is fully working under this circumstance. Thus the running time is linearly related to input size.

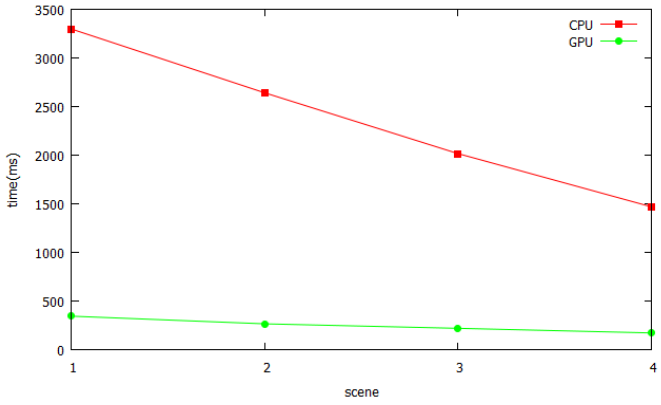


Figure 6: KD-Tree travel time

4 Conclusion

This project mainly consist of two part, one is the literature review of KD-Tree construction and travelling algorithm on GPU and the other is implementation of one of the paper. From the implementation we demonstrate the effectiveness of GPU KD-Tree compared with CPU KD-Tree and the possibility of accelerate ray tracing and nearest neighbour problems.

The implementation of the algorithm is using CUDA, one of the most popular GPU programming language invented by NVIDIA. This project helps to learn that GPU programming has huge difference with CPU and one needs to know details about the hardware to write high quality code.

One unfinished part of this project is integration with ray tracer. There is a folder in the source code zip package named "smallpt", which is a CUDA version of the well known 99 line ray tracer. The first step which terns smallpt into CUDA is done. The future of the work may be mix GPU KD-Tree with smallpt.

References

[1] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[2] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[3] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, May 1987.

[4] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[5] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011.

[6] W. Hunt, W.R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. *Symposium on Interactive Ray Tracing*, 0:81–88, 2006.

[7] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[8] Matt Pharr and Greg Humphreys, editors. *Physically Based Rendering, Second Edition:From Thery To Implementation.* Morgan Kaufmann, San Francisco, 2010.

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

[10] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, September 2006.

[11] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[12] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, 2007.

[13] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[14] Wikipedia. k-d tree. `http://en.wikipedia.org/wiki/K-d_tree`, 2014. [Online; accessed 07-Feb-2014].

[15] Zhefeng Wu, Fukai Zhao, and Xinguo Liu. Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 71–78, New York, NY, USA, 2011. ACM.

[16] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.