Parallel BVH Generation via GPU

Denise Yang (denisey@andrew.cmu.edu)

Jai Madisetty (jmadiset@andrew.cmu.edu)

Github: https://github.com/Denise-Yang/BVHGPU

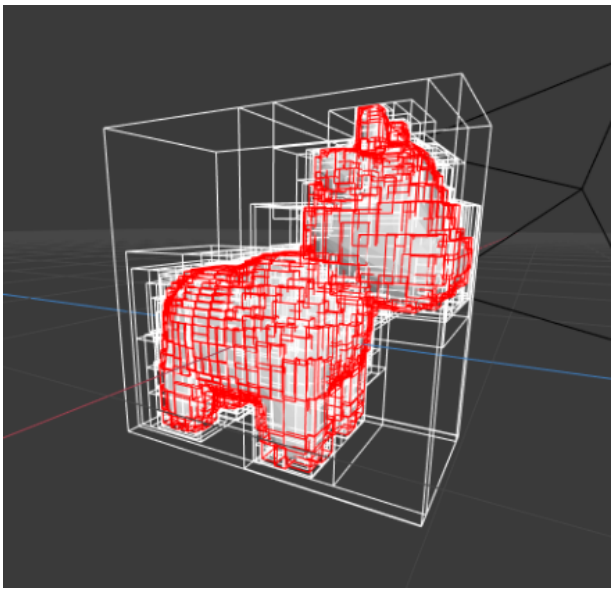


Figure 1. Bounding Volume Hierarchy on cow.dae

SUMMARY:

We have parallelized building the Bounding Volume Hierarchy (BVH) data structure used to improve the performance of ray tracing-based rendering algorithms. We shifted from our original means of parallelizing (CUDA) to OpenMP to parallelize the formation of the BVH tree. There are two main portions of the sequential code we focused on parallelizing: the recursive calls made to generate each node of the binary tree and the work within each node's recursive call. We were able to use the sequential and parallelized BVH tree implementations to render 3D objects on the Scotty3D package. We measured our parallelized versions' performance against the sequential version and analyzed the results of our experiments.

BACKGROUND:

The tree is structured using a global `root_idx` variable and a global `nodes` vector, where each node `n` in the vector stores information regarding its own bounding box and the indices of its left and right nodes in that same `nodes` vector, as well as a `size` field that indicates how many primitives are within the nodel.

Our program will output a binary BVH tree given an input vector of primitives i.e. shapes to render. In order to find the optimal partition to split the primitives in a scene into and recursively continue this operation within each respective partition until we reach the minimum node size i.e. the minimum amount of primitives within a node for it to be considered a leaf. To do so, we first sorted the primitives into eight bins for each of the eight possible partitions based on the barycenter of each primitive. This is done in our function `sort_into_buckets();` After that we would find the optimal partition by creating a bounding box around all of the possible groupings of consecutive bins which is performed by `get_opt_partition()`. For example, given four bins our partitions would be in groups of bounding boxes of bins 1, and 2-4, bins 1-2, and 3-4, bins 1-3 and 4, and bins 1-4 in one partition. Once all the different partitions have been formed, we iterate through each pair and calculate their surface area heuristic i.e. the sum of the surface areas of each primitive within a bounding box divided by the total surface area of the bounding box. We repeat this process for the remaining two axes in a 3D coordinate system and then sort the primitives vector such that all the primitives in one partition are on the left half of the vector and the primitives on the other side of the partition are on the right half. We then create two child nodes and recurse whilst limiting the vector of primitives to iterate through to the respective half i.e. the left child will continue to process the left side of the now sorted primitives vector and the right child will process the right half. This process will continue until the number of primitives that a child must go through is less than four.

There are many opportunities for parallelism within this program. Since we need to build a tree, the creation of each child is dependent on the creation of each parent node; however, nodes on the same level will operate on an individual section of the primitives vector such that we could process the creation of nodes on the same level in parallel. Within the process of sorting nodes, we can also perform the check to determine which bin the primitive belongs to independently of each other. This check could be done data-parallel with SIMD instructions; however there is likely to be an uneven workload distribution, and the main focus of this project is to identify if we can speed up BVH creation utilizing multi-threading. Currently within the structure of the code, each axis is computed sequentially however it would be possible to parallelize across all axes as well.

APPROACHES:

Sequential: Recursively generating BVH Nodes

In order to generate our BVH tree, we have `build` responsible for assigning the root to the generated tree. We use `build_helper` as a wrapper responsible for all computation, including finding the optimal partitions and recursively generating the left and right children of a node.

```
template<typename Primitive>
void BVH<Primitive>::build(std::vector<Primitive>&& prims, size_t max_leaf_size) {
    float B = 8.0;
    nodes.clear();
    primitives = std::move(prims);
    root_idx = build_helper(primitives, B, 0, primitives.size(), max_leaf_size);
}
```

```
l_index = build_helper(primitives, B, start, start + opt_A.size, max_leaf_size);
r_index = build_helper(primitives, B, start + opt_A.size, end, max_leaf_size);
return new_node(box, start, end-start, l_index, r_index);
```

Approach 1: Parallelizing across Primitives

Due to the fact that we were not able to install the packages needed to run Scotty3D on the GHC machines we relied on our personal computers i.e. a 4 GHz Quad-Core Intel Core i5 to utilize multi-core parallelism via OpenMP. The functions that we will parallelize are sort_into_buckets and get_opt_partitions .

Parallelizing sort_into_buckets():

We first began by parallelizing over the primitives in sort_into_buckets(), which sorts primitives into bounding boxes based on their barycenter, since a majority of the work within this function involved in determining which binning a primitive is independent. However the next step to find the optimal partition requires that all of the primitives must be sorted amongst the bins, creating a dependency. Furthermore there will be contention amongst the threads to access the bins. In order to avoid the overhead from locking, we assigned each thread a vector of eight bins in such that they can write to their respective bins without the possibility of conflict. For this approach we created a std::vector<Nodes> of size `total threads x 8 bins`. Once all the primitives have been sorted, we will then iterate through the bins within each thread and combine them with their respective bins of other threads. However the results, which will be discussed later in this paper, were not what we expected so we instrumented our code to identify which functions were the most computationally intensive, and as shown in **Table 1** below, while sort_into_buckets() certainly comprises a noteworthy

portion of the execution time, get_opt_partitions and the builtin vector::sort() encompass a even more significant portion.

	Time (ms)
sort_into_buckets()	0.206
get_opt_partitions()	0.349
vector::sort()	0.462
Total time	1.076

Table 1. Execution Times of Functions in build()

Parallelizing get_opt_partitions():

We then aimed to parallelize get_opt_partitions(); however since the partition formation of the sequential implementation relies on the previously formed partitions, we needed to refactor our code such that the formation of each partition was independent of each other. To do this, we looped through the possible partitions and for each partition, we iterated through all of the bins and would either bound them in a left or right half. In order to parallelize this we had each thread be assigned a partition to build via `#pragma omp parallel for`. The work that each thread must do is roughly evenly distributed since each thread must iterate through all the bins.

Parallelizing by combining axes loops:

We aimed for a third optimization by removing the separate for loop per axis, since our original code found the optimal partition for divides along a single axis, before trying to find the next optimal partition of the other axis and then comparing them. In order to perform this we created a shared global variable of size `total threads x 8 bins x 3 axis`, and modified the indexing of the sort_into_buckets for loops such that it iterated through the list of primitives three times, and would sort and store the partitions into a designated spot within the shared global variable.

Approach 2: Parallelizing across Nodes

In order to parallelize across the nodes of the BVH tree, it was necessary to convert the recursive implementation into an iterative implementation. Our original plan was to parallelize the calls using CUDA, which would have been difficult but possible; however, due to dependencies and limited permissions for installing certain packages, we went with an OpenMP implementation instead.
The tree is structured using a global `root_idx` variable and a global `nodes` vector, where each node `n` in the vector stores information regarding its own bounding box and the indices of its left and right nodes

in that same `nodes` vector. The parallelizing across the many nodes of the BVH tree took 3 iterations of significant code modifications, the first being algorithmic.

- Part 1:

First, I had to switch from a recursive algorithm that uses recursive calls to build the left and right trees to an iterative algorithm that builds nodes level by level, from top to bottom. This is unlike the recursive implementation, where the BVH tree is built from the bottom up. In other words, the two recursive calls return to the parent the indices of the left and right children. So, there is no need to store a hierarchy of nodes as the recursion implicitly defines this. The key to switching to an iterative implementation was with the use of a task queue, which we unload level by level.

We can use the root node as an example for how this algorithm works.

We start by creating a queue with solely the root node, containing the initial parameters associated with the root node we must pass to the `build_helper` function (which does all the computation to create the current node and determine the ideal partitions for its children). We then remove the root node (level 0) from this queue and pass the associated parameters to the `build_helper` function. The values computed in `build_helper` are then returned via a passed-in parameter. The values computed include parameters to be passed in for the left and right childrens' calls to `build_helper` as well as the current node's assigned index in the `nodes` vector. From there, we can label the returned `root_idx` with the returned `nodes` index.

Due to the top-down nature of this algorithm, when we create a node, we do not yet know the information about its children. Thus, we need to update the parent node's fields once a child's computation is completed. Since this is the root node, we do not have to reassign a parent's left/right field. We can then check if it hit the base case, and if it did not, we can add the children nodes with the returned parameters to the queue. For a better understanding, here is what the code looks like:

```
//initialize parameters
float B = 8.0;
nodes.clear();
primitives = std::move(prims);
struct q *q = (struct q *)malloc(sizeof(struct q));     // initialize queue
q->start = NULL;
q->end = NULL;
q->size = 0;
q_add(q, 0, primitives.size(), -1, none);               // add root node to queue
while (q->size > 0) {                                   // loop while there exists more levels to the tree
    int len = q->size;
    for (int i = 0; i < len; i++) {                     // loop through current level
        struct v v;                                     // v contains current node's parameters for solve_helper
        q_rem(q,v);
        struct solve_ret ret;                           // ret contains parameters for left and right children
        solve_helper(primitives, B, v.start, v.end, max_leaf_size, ret);

        if (v.parent_idx == -1) {                       // if root
            root_idx = ret.node_idx;
        }
        if (v.lab == left) {                            // if left child, update parent's left field
            nodes[v.parent_idx].l = ret.node_idx;
        }
        if (v.lab == right) {                           // if right child, update parent's right field
            nodes[v.parent_idx].r = ret.node_idx;
        }
        if (nodes[ret.node_idx].l == 0 && nodes[ret.node_idx].r == 0) {     // if base case hit, continue
            continue;
        }
        q_add(q, ret.start_L, ret.end_L, ret.node_idx, left);               // else add children to queue
        q_add(q, ret.start_R, ret.end_R, ret.node_idx, right);
    }
}
```

- Part 2:

After making this algorithmic change, it was time to add the OpenMP code. We decided we'd parallelize amongst the nodes per each level of the tree. Thus, we inserted a `#pragma omp for` above the for loop in the above code snippet. Since the queue and `nodes` vector are now shared variables, we needed to insert critical sections in our code whenever we access either the queue or the `nodes` vector. Specifically, we need to insert whenever we add or remove from the queue or whenever we push a node to the `nodes` vector. Thus, we added critical sections around the `q_rem` and two `q_add` calls, and also one around the `nodes.push_back()` call.

We were getting very little speedup with this approach. This made sense as every thread was in contention for the queue and `nodes` vector.

- Part 3:

We determined that the reason for such low speedup was due to every thread being in contention for the removing from the queue, adding to the queue, and pushing to the `nodes` vector. In this case, the threads would have to operate somewhat sequentially due to the large number of critical sections.

Because of this, we decided to assign each thread its own queue. This way, we'd greatly reduce the contention amongst threads. Instead of all threads waiting on a singular critical section, we could

create locks for each thread and make assignment of tasks to each thread's queue as evenly distributed as possible. This way, in the presence of many threads, there would ideally only be a handful of threads potentially assigning/removing a task to the same queue. Additionally, instead of pushing to the `nodes` vector, we decided to instead create unique indices for each of the tasks and resize the `nodes` vector according to how many nodes have been created. So, instead of pushing to the vector, we index into the vector. This way, we only enter a `#pragma omp critical` section when the index we want to insert is greater than or equal to the size of the `nodes` vector. While we would need a critical section for creating these unique indices, it would not have to be as costly as the queue addition/removal.

Here is the modified code:

Initializing parameters, queues, and locks:

```
// init code
float B = 8.0;
nodes.clear();
primitives = std::move(prims);

std::vector<struct q*> q(omp_get_max_threads());
std::vector<omp_lock_t> locks(omp_get_max_threads());
for (int i = 0; i < omp_get_max_threads(); i++) {
    q[i] = (struct q *)malloc(sizeof(struct q));
    q[i]->start = NULL;
    q[i]->end = NULL;
    q[i]->size = 0;
    omp_init_lock(&locks[i]);
}
```

Algorithm for assigning tasks to queues and computing tasks from queues:

```
while (total_size > 0) {                        // case on total number of items in all queues
    int thread_id;
    bool q_rem_res;
    struct v v;
    struct solve_ret ret;
    int left_ch, right_ch;
    #pragma omp parallel private(thread_id, q_rem_res, v, ret, left_ch, right_ch) shared(q, total_size, nodes_i)
    {
        thread_id = omp_get_thread_num();

        omp_set_lock(&locks[thread_id]);        // per thread lock
        q_rem_res = q_rem(q[thread_id],v);      // per thread queue removal
        omp_unset_lock(&locks[thread_id]);      // per thread unlock

        if (q_rem_res) {                        // if queue was not empty
            #pragma omp atomic                  // removed from queue => decrease by 1
            total_size -= 1;

             #pragma omp critical               // create unique indices
            {
                left_ch = nodes_i;
                right_ch = nodes_i+1;
                nodes_i += 2;
            }

            solve_helper(primitives, B, v.start, v.end, max_leaf_size, v.nodes_i, ret);
            if (v.parent_idx == -1) {
                root_idx = v.nodes_i;
            }
            if (v.lab == left) {
                nodes[v.parent_idx].l = v.nodes_i;
            }
            if (v.lab == right) {
                nodes[v.parent_idx].r = v.nodes_i;
            }
            if (nodes[v.nodes_i].l != 0 || nodes[v.nodes_i].r != 0) {   // assign children to some queue

                omp_set_lock(&locks[left_ch%omp_get_max_threads()]);
                q_add(q[left_ch%omp_get_max_threads()], ret.start_L, ret.end_L, v.nodes_i, left, left_ch);
                omp_unset_lock(&locks[left_ch%omp_get_max_threads()]);

                omp_set_lock(&locks[right_ch%omp_get_max_threads()]);
                q_add(q[right_ch%omp_get_max_threads()], ret.start_R, ret.end_R, v.nodes_i, right, right_ch);
                omp_unset_lock(&locks[right_ch%omp_get_max_threads()]);

                #pragma omp atomic              // added to queue => increase by 2
                total_size += 2;
            }

        }
    }
}
```

Since we want each thread to have access to its own queue, we decided to use `#pragma omp parallel` so it'd be easier to manage and reason about the parallel code. We determined that level by level computation was not necessary either since the children are only dependent on its parent node and not the rest of the tree.

RESULTS:

Results for Parallelizing across Primitives:

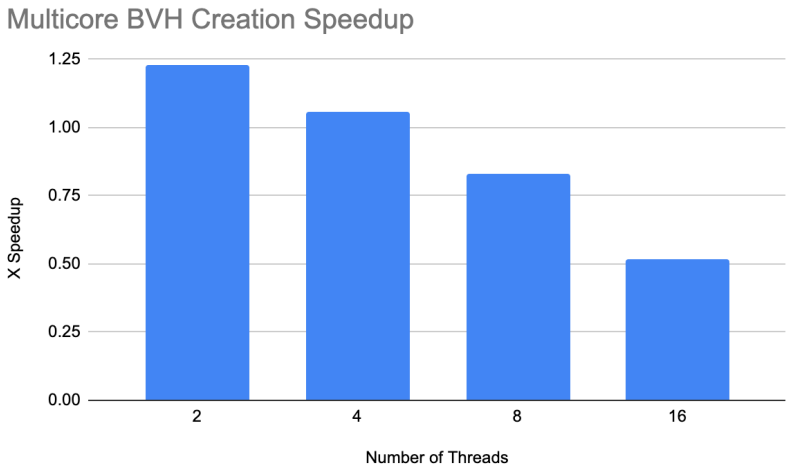


Figure 1. Speedup measured by P thread execution time/single thread execution time. The baseline was the original sequential Scotty3D BVH runtime.

As shown by **Figure 1**, we did achieve a significant speedup with two threads; however as we increased the thread count our performances became worse than the sequential implementation. We ran `top` to profile code and identified that a likely cause for these results is that Scotty3D is normally run on two cores due to the fact that it handles many moving components such as displaying an interactive scene, detecting user inputs, and supporting additional windows to render scenes. When we run the BVH generation program, `top` reveals that we do begin to use all four cores, however since the interactive Scotty3D window and render window are open; it is likely that two of the cores are still used to support other Scotty3D functionality and as a result we really only have two cores to run our BVH generation code. This means that as we try to increase the number of threads, we will need to perform more context swaps and aggregating all the results of shared arrays will take longer since these data structures will be larger as well thus leading to less undesirable speedups.

As shown in **Figure 2**, sort_into_buckets() and get_opt_partitions() do encompass a significant portion of the execution time however the largest portion of the program is the builtin vector::sort function that we use to sort the primitives vector before each recursive call. It would be difficult to parallelize this in-place sort because the placement of each primitive would be somewhat dependent on its neighboring primitives. Furthermore even though we parallelized get_opt_partitions, we are only generating 8 tasks amongst the threads, meaning that there is not much work to be distributed overall, and

the number of requests generated by sort_into_buckets() depends on what scene we're rendering but for testing cow.dae, we were generating 5856 tasks. Another possibility as to why our parallelization do not accrue significant speedup is due to the fact that as we recurse into higher levels of the tree, the list of primitives will shrink such that we do not get as much parallelization per node, and there are more nodes as well meaning that our parallel code will begin becoming very similar to our sequential code.

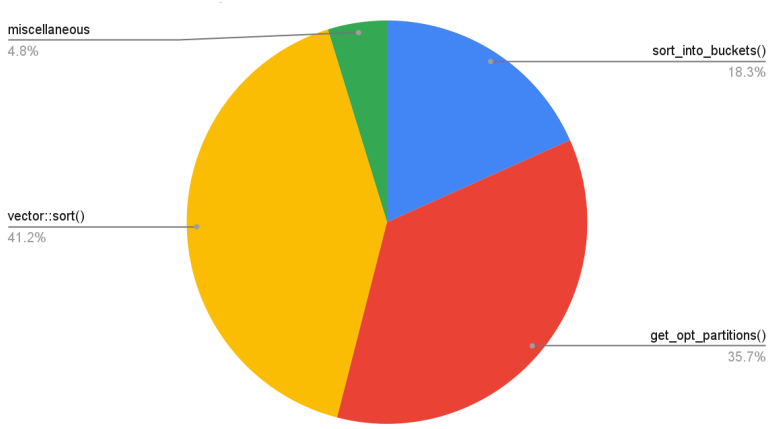


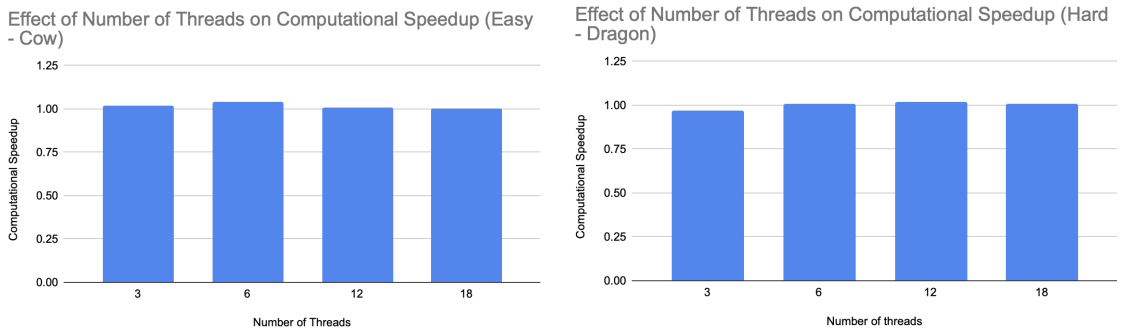
Figure 2. Distribution of Sequential BVH Execution Times

Results for Parallelizing across Nodes:

- Machine used: 2.9 GHz 6-Core Intel Core i9 Processor

As aforementioned, we were not able to yield good speedup during our first attempt at parallelizing. Here are the graphs associated with Part 2 of the Approach section:

- Speedup without per-thread queues:



As shown above, little to no speedup was possible for the cow (0.78 seconds baseline, single threaded) nor the dragon (84.5 seconds baseline, single threaded). As aforementioned, we believe the main reason for the lack of speedup even though we use `#pragma omp for` is due to the 3 large critical sections in

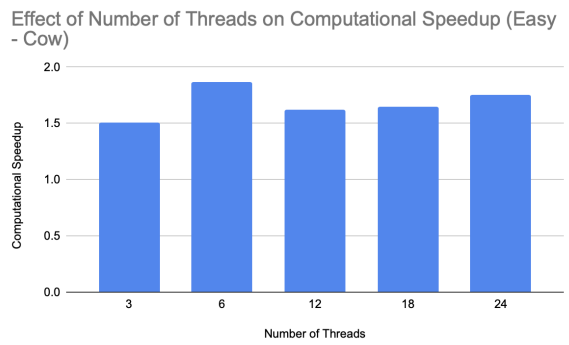
our code. This goes hand in hand with the fact that the queue and `nodes` vector are contended for by each and every thread. Essentially, the two main reasons for poor speedup with this version are:

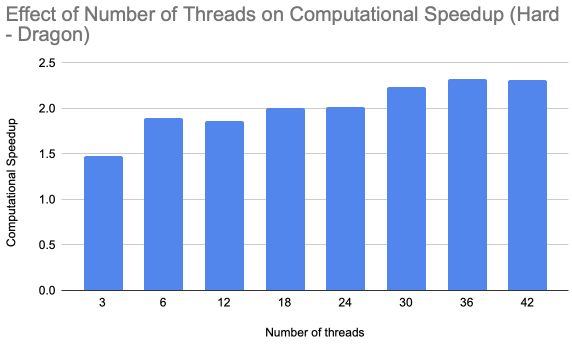
1. A good portion of the computation is wrapped within critical sections
2. There is overhead assigning threads to iterations of the for loop since the number of iterations at deeper levels can reach numbers significantly greater than the number of threads.

With the second iteration of parallelizing however, we were able to yield drastic speedup compared to the first.

Here are the graphs associated with Part 3 of the Approach section:

- Speedup with per-thread queues:





As shown above, we were able to yield up to 1.87x speedup for the cow, and up to 2.32x speedup for the dragon.

There are multiple reasons for this improvement. One reason is due to the significantly lower contention of any one single queue. Since we are now adding to other threads' queues in an interleaved-assignment fashion (to enable even workload distribution), we are greatly reducing the number of threads that would add to any one queue at the same time or remove from any one queue at the same time. Another reason is due to a more evenly distributed workload as the number of threads increases. This is because we are using interleaved assignments to assign tasks to queues, as aforementioned. One last reason is because we do not have contention when adding to the `nodes` vector either. Instead of pushing, we index into the vector to add a node. We only enter a critical section when we have to resize the vector, which does not happen often since it is resized to 4 times than what is needed.

It is interesting to observe that the dragon performed better at its best with 2.32x speedup compared to 1.87x with the cow. This is due to the fact that there are a lot more levels. Since the workload grows

exponentially as the number of the levels increases, there is more room for parallelism and better workload distribution.

We do believe that if we had more cores, we would have been able to yield better results. Running htop simultaneously with Scotty3D showed that some of the CPU was used for other applications such as VSCode and Safari. Thus, not all of my 6 cores were used towards creating the BVH tree. We definitely would have needed more CPUs since we were not programming in CUDA.

REFERENCES:

https://cmu-graphics.github.io/Scotty3D/

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT:

STUDENT	RESPONSIBILITIES	DISTRIBUTION
Jai Madisetty	Helping each other debug Approach 2 Writing Report Compiling Video	40%
Denise Yang	Helping each other debug Approach 1 Compiling OpenMP Writing Report Compiling Video	60%