ACCELERATED COMPUTING GRAPHICS / SIMULATION HPC

Thinking Parallel, Part II: Tree Traversal on the GPU

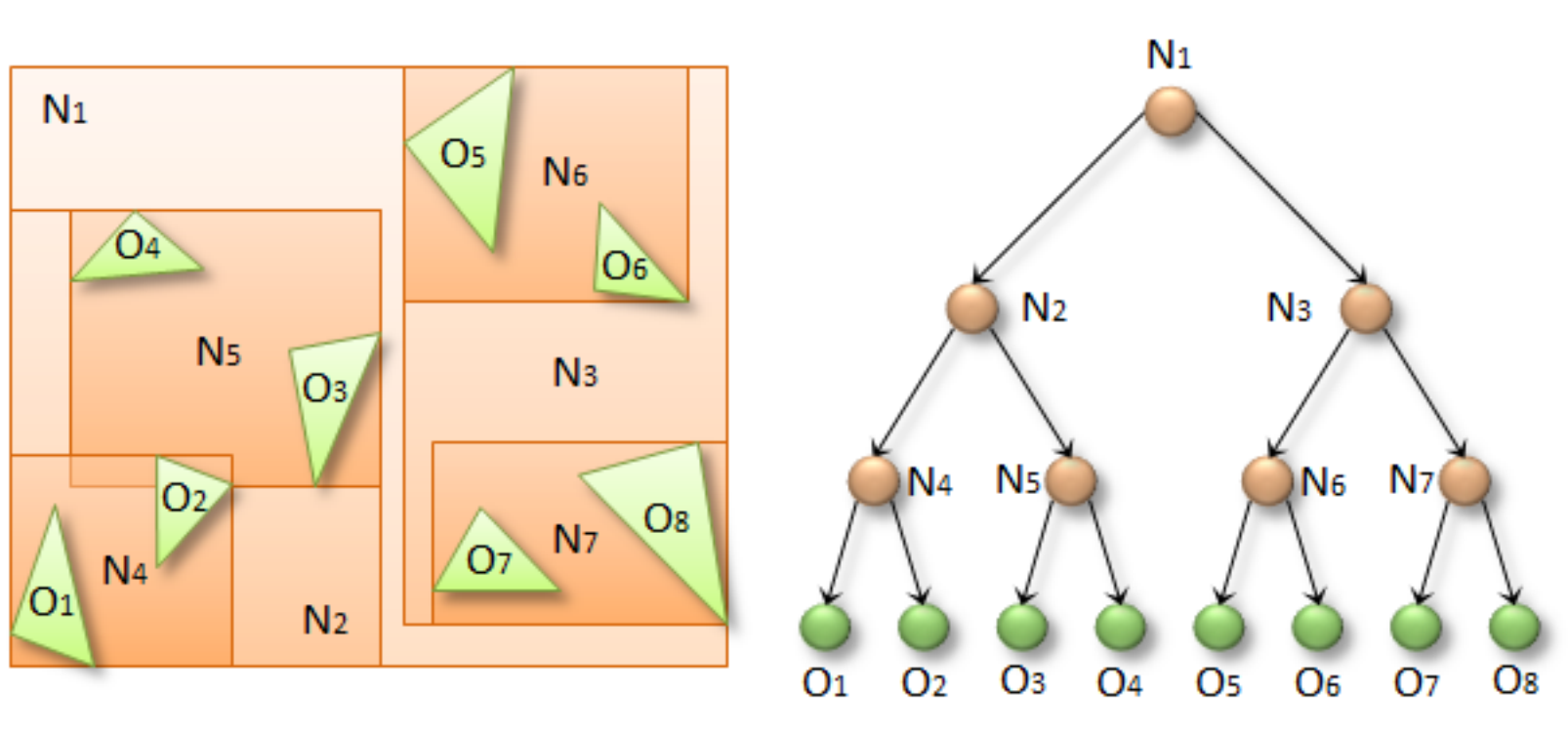
By Tero Karras | November 26, 2012 Tags: Algorithms, Parallel Programming

In the first part of this series, we looked at collision detection on the GPU and discussed two commonly used algorithms that find potentially colliding pairs in a set of 3D objects using their axis-aligned bounding boxes (AABBs). Each of the two algorithms has its weaknesses: *sort and sweep* suffers from high execution divergence, while *uniform grid* relies on too many simplifying assumptions that limit its applicability in practice.

In this part we will turn our attention to a more sophisticated approach, *hierarchical tree traversal*, that avoids these issues to a large extent. In the process, we will further explore the role of divergence in parallel programming, and show a couple of practical examples of how to improve it.

Bounding Volume Hierarchy

We will build our approach around a bounding volume hierarchy (BVH), which is a commonly used acceleration structure in ray tracing (for example). A bounding volume hierarchy is essentially a hierarchical grouping of 3D objects, where each group is associated with a conservative bounding box.



Suppose we have eight objects, O_1-O_8, the green triangles in the figure above. In a BVH, individual objects are represented by leaf nodes (green spheres in the figure), groups of objects by internal nodes (N_1-N_7, orange spheres), and the entire scene by the root node (N_1). Each internal node (e.g. N_2) has two children (N_4 and N_5), and is associated with a bounding volume (orange rectangle) that fully contains all the underlying objects (O_1-O_4). The bounding volumes can basically be any 3D shapes, but we will use axis-aligned bounding boxes (AABBs) for simplicity.

Our overall approach is to first construct a BVH over the given set of 3D objects, and then use it to accelerate the search for potentially colliding pairs. We will postpone the discussion of efficient hierarchy construction to the third part of this series. For now, let's just assume that we already have the BVH in place.

Independent Traversal

Given the bounding box of a particular object, it is straightforward to formulate a recursive algorithm to query all the objects whose bounding boxes it overlaps. The following function takes a BVH in the parameter `bvh` and an AABB to query against it in the parameter `queryAABB`. It tests the AABB against the BVH recursively and returns a `list` of potential collisions.

```
void traverseRecursive( CollisionList& list,
                        const BVH&     bvh,
                        const AABB&    queryAABB,
                        int            queryObjectIdx,
                        NodePtr        node)
{
    // Bounding box overlaps the query => process node.
    if (checkOverlap(bvh.getAABB(node), queryAABB))
    {
        // Leaf node => report collision.
        if (bvh.isLeaf(node))
            list.add(queryObjectIdx, bvh.getObjectIdx(node));

        // Internal node => recurse to children.
        else
        {
            NodePtr childL = bvh.getLeftChild(node);
            NodePtr childR = bvh.getRightChild(node);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childL);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childR);
```

The idea is to traverse the hierarchy in a top-down manner, starting from the root. For each node, we first check whether its bounding box overlaps with the query. If not, we know that none of the underlying leaf nodes will overlap it either, so we can skip the entire subtree. Otherwise, we check whether the node is a leaf or an internal node. If it is a leaf, we report a potential collision with the corresponding object. If it is an internal node, we proceed to test each of its children in a recursive fashion.

To find collisions between all objects, we can simply execute one such query for each object in parallel. Let's turn the above code into CUDA C++ and see what happens.

```
__device__ void traverseRecursive( CollisionList& list,
                                   const BVH&     bvh,
                                   const AABB&    queryAABB,
                                   int            queryObjectIdx,
                                   NodePtr        node)
{
    // same as before...
}

__global__ void findPotentialCollisions( CollisionList list,
                                         BVH           bvh,
                                         AABB*         objectAABBs,
                                         int           numObjects)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < numObjects)
        traverseRecursive(bvh, list, objectAABBs[idx],
                          idx, bvh.getRoot());
}
```

Here, we have added the `__device__` keyword to the declaration of `traverseRecursive()`, to indicate that the code is to be executed on the GPU. We have also added a `__global__` kernel function that we can launch from the CPU side. The `BVH` and `CollisionList` objects are convenience wrappers that store the GPU memory pointers needed to access BVH nodes and report collisions. We set them up on the CPU side, and pass them to the kernel by value.

The first line of the kernel computes a linear 1D index for the current thread. We do not make any assumptions about the block and grid sizes. It is enough to launch at least `numObjects` threads in one way or another—any excess threads will get terminated by the second line. The third line fetches the bounding box of the corresponding object, and calls our function to perform recursive traversal, passing the objects index and the pointer to the root node of the BVH in the last two arguments.

To test our implementation, we will run a dataset taken from APEX Destruction using a GeForce GTX 690 GPU. The data set contains 12,486 objects representing debris falling from the walls of a corridor, and 73,704 pairs of potentially colliding objects, as shown in the following screenshot.

The total execution time of our kernel for this dataset is **3.8** milliseconds. Not very good considering that this kernel is just one part of collision detection, which is only one part of a simulation that we would ideally like to run at 60 FPS (16 ms). We should be able to do better.

Minimizing Divergence

The most obvious problem with our recursive implementation is high execution divergence. The decision of whether to skip a given node or recurse to its children is made independently by each thread, and there is nothing to guarantee that nearby threads will remain in sync once they have made different decisions. We can fix this by performing the traversal in an iterative fashion, and managing the recursion stack explicitly, as in the following function.

```
__device__ void traverseIterative( CollisionList& list,
                                   BVH& bvh,
                                   AABB& queryAABB,
                                   int queryObjectIdx)
{
    // Allocate traversal stack from thread-local memory,
    // and push NULL to indicate that there are no postponed nodes.
    NodePtr stack[64];
    NodePtr* stackPtr = stack;
    *stackPtr++ = NULL; // push

    // Traverse nodes starting from the root.
    NodePtr node = bvh.getRoot();
    do
    {
        // Check each child node for overlap.
        NodePtr childL = bvh.getLeftChild(node);
        NodePtr childR = bvh.getRightChild(node);
        bool overlapL = ( checkOverlap(queryAABB,
                                       bvh.getAABB(childL)) );
        bool overlapR = ( checkOverlap(queryAABB,
                                       bvh.getAABB(childR)) );
```

The loop is executed once for every internal node that overlaps the query box. We begin by checking the children of the current node for overlap, and report an intersection if one of them is a leaf. We then check whether the overlapped children are internal nodes that need to be processed in a subsequent iteration. If there is only one child, we simply set it as the current node and start over. If there are two children, we set the left child as the current node and push the right child onto the stack. If there are no children to be traversed, we pop a node that was previously pushed to the stack. The traversal ends when we pop `NULL`, which indicates that there are no more nodes to process.

The total execution time of this kernel is **0.91** milliseconds—a rather substantial improvement over **3.8 ms** for the recursive kernel! The reason for the improvement is that each thread is now simply executing the same loop over and over, regardless of which traversal decisions it ends up making. This means that nearby threads execute every iteration in sync with each other, even if they are traversing completely different parts of the tree.

But what if threads are indeed traversing completely different parts of the tree? That means that they are accessing different nodes (*data divergence*) and executing a different number of iterations (*execution divergence*). In our current algorithm, there is nothing to guarantee that nearby threads will actually process objects that are nearby in 3D space. The amount of divergence is therefore very sensitive to the order in which the objects are specified.

Fortunately, we can exploit the fact that the objects we want to query are the same objects from which we constructed the BVH. Due to the hierarchical nature of the BVH, objects close to each other in 3D are also likely to be located in nearby leaf nodes. So let's order our queries the same way, as shown in the following kernel code.

```
__global__ void findPotentialCollisions( CollisionList list,
                                         BVH           bvh)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < bvh.getNumLeaves())
    {
        NodePtr leaf = bvh.getLeaf(idx);
        traverseIterative(list, bvh,
                          bvh.getAABB(leaf),
                          bvh.getObjectIdx(leaf));
    }
}
```

Instead of launching one thread per object, as we did previously, we are now launching one thread per leaf node. This does not affect the behavior of the kernel, since each object will still get processed exactly once. However, it changes the ordering of the threads to minimize both execution and data divergence. The total execution time is now **0.43** milliseconds—this trivial change improved the performance of our algorithm by another 2x!

There is still one minor problem with our algorithm: every potential collision will be reported twice—once by each participating object—and objects will also report collisions with themselves. Reporting twice as many collisions also means that we have to perform twice as much work. Fortunately, this can be avoided through a simple modification to the algorithm. In order for object A to report a collision with object B, we require that A must appear before B in the tree.

To avoid traversing the hierarchy all the way to the leaves in order to find out whether this is the case, we can store two additional pointers for every internal node, to indicate the rightmost leaf that can be reached through each of its children. During the traversal, we can then skip a node whenever we notice that it cannot be used to reach any leaves that would be located after our query node in the tree.

```
__device__ void traverseIterative( CollisionList& list,
                                   BVH&           bvh,
                                   AABB&          queryAABB,
                                   int            queryObjectIdx,
                                   NodePtr        queryLeaf)
{
    ...

    // Ignore overlap if the subtree is fully on the
    // left-hand side of the query.

    if (bvh.getRightmostLeafInLeftSubtree(node) <= queryLeaf)
        overlapL = false;

    if (bvh.getRightmostLeafInRightSubtree(node) <= queryLeaf)
        overlapR = false;

    ...
}
```

After this modification, the algorithm runs in **0.25** milliseconds. That is a 15x improvement over our starting point, and most of our optimizations were only aimed at minimizing divergence.

Simultaneous Traversal

In independent traversal, we are traversing the BVH for each object independently, which means that no work we perform for a given object is ever utilized by the others. Can we improve upon this? If many small objects happen to be located nearby in 3D, each one of them will essentially end up performing almost the same traversal steps. What if we grouped the nearby objects together and performed a single query for the entire group?

This line of thought leads to an algorithm called simultaneous traversal. Instead of looking at individual nodes, the idea is to consider pairs of nodes. If the bounding boxes of the nodes do not overlap, we know that there will be no overlap anywhere in their respective subtrees, either. If, on the other hand, the nodes do overlap, we can proceed to test all possible pairings between their children. Continuing this in a recursive fashion, we will eventually reach pairs of overlapping leaf nodes, which correspond to potential collisions.

On a single-core processor, simultaneous traversal works really well. We can start from the root, paired with itself, and perform one big traversal to find all the potential collisions in one go. The algorithm performs significantly less work than independent traversal, and there really is no downside to it—the implementation of one traversal step looks roughly the same in both algorithms, but there are simply less steps to execute in simultaneous traversal (60% less in our example). It's a better algorithm, right?

To parallelize simultaneous traversal, we must find enough independent work to fill the entire GPU. One easy way to accomplish this is to start the traversal a few levels deeper in the hierarchy. We could, for example, identify an appropriate cut of 256 nodes near the root, and launch one thread for each pairing of the nodes (32,896 in total). This would result in sufficient parallelism without increasing the total amount of work too much. The only source of extra work is that we need to perform at least one overlap test for each initial pair, whereas the single-core implementation would avoid some of the pairs altogether.

So, the parallel implementation of simultaneous traversal does less work than independent traversal, and it does not lack in parallelism, either. Sounds good, right? Wrong. It actually performs a lot worse than independent traversal. How is that possible?

The answer is—you guessed it—divergence. In simultaneous traversal, each thread is working on a completely different portion of the tree, so the data divergence is high. There is no correlation between the traversal decisions made by nearby threads, so the execution divergence is also high. To make matters even worse, the execution times of the individual threads vary wildly—threads that are given a non-overlapping initial pair will exit immediately, whereas the ones given a node paired with itself are likely to execute the longest.

Maybe there is a way to organize the computation differently so that simultaneous traversal would yield better results, similar to what we did with independent traversal? There have been many attempts to accomplish something like this in other contexts, using clever work assignment, packet traversal, warp-synchronous programming, dynamic load balancing, and so on. Long story short, you can get pretty close to the performance of independent traversal, but it is extremely difficult to actually beat it.

Discussion

We have looked at two ways of performing broad-phase collision detection by traversing a hierarchical data structure in parallel, and we have seen that minimizing divergence through relatively simple algorithmic modifications can lead to substantial performance improvements.

Comparing independent traversal and simultaneous traversal is interesting because it highlights an important lesson about parallel programming. Independent traversal is a simple algorithm, but it performs more work than necessary. overall. Simultaneous traversal, on the other hand, is more intelligent about the work it performs, but this comes at the price of increased complexity. Complex algorithms tend to be harder to parallelize, are more susceptible to divergence, and offer less flexibility when it comes to optimization. In our example, these effects end up completely nullifying the benefits of reduced overall computation.

Parallel programming is often less about how much work the program performs as it is about whether that work is divergent or not. Algorithmic complexity often leads to divergence, so it is important to try the simplest algorithm first. Chances are that after a few rounds of optimization, the algorithm runs so well that more complex alternatives have a hard time competing with it.

In my next post, I will focus on parallel BVH construction, talk about the problem of occupancy, and present a recently published algorithm that explicitly aims to maximize it.

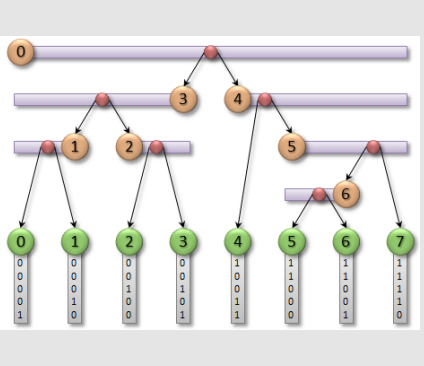
4 Comments

About the Authors

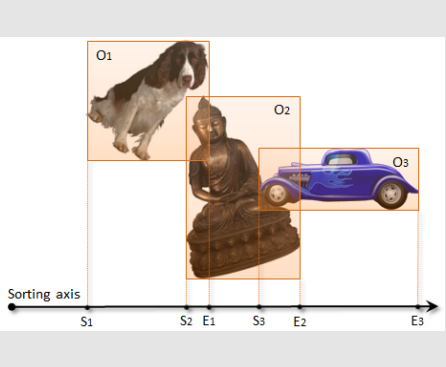
About Tero Karras
Tero Karras joined NVIDIA Research in 2009, after having worked for 3 years with NVIDIA's Tegra business. His research interests include real time ray tracing, representations for detailed 3D content, parallel algorithms, and GPU computing.
View all posts by Tero Karras »

Related posts

Thinking Parallel, Part III: Tree Construction on the GPU
By Tero Karras | December 19, 2012

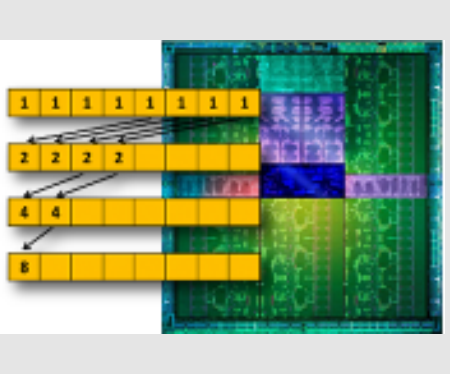


Thinking Parallel, Part I: Collision Detection on the GPU
By Tero Karras | November 12, 2012



Cooperative Groups: Flexible CUDA Thread Programming
By Mark Harris and Kyrylo Perelygin | October 4, 2017

Faster Parallel Reductions on Kepler
By Justin Luitjens | February 13, 2014



Comments

4 Comments NVIDIA Developer Blog Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

hwdong • 3 years ago
Why the if-else in "traverseIterative" does not result in divergent? I can't understand this.
Reply • Share ›

Hurrrrrrrrrrrrrr → hwdong • 3 years ago
Are you referring to the piece I quote below?

if (!traverseL && !traverseR)
node = *--stackPtr; // pop
else
{
node = (traverseL) ? childL : childR;
if (traverseL && traverseR)
*stackPtr++ = childR; // push
}

I think that this causes execution to diverge, but then probably it can converge again at the end of the else block. That is only my speculation though... I too would like to see somebody explain more fully when execution divergence does or does not happen. I imagine it will vary from one piece of hardware to another, and/or from one compiler to another...
Reply • Share ›

Olga • 4 years ago • edited
Hi! Have anybody managed to get close to 0.25ms for BVH tree traversal timings stated above? My implementation of traverseIterative on GPU is extremely slow (>100ms) which is very surprising as implementation of algorithms from part iii gives very close timings to those declared in paper. I'm testing a 18K triangles scene, which has about 50K potential intersections. If anybody can help with advice, I can provide my source code.
Reply • Share ›

Hurrrrrrrrrrrrrr → Olga • 3 years ago
I'm down to 2.5 ms or so. Not what's claimed above (I need to optimize something it seems) but better than 100 ms.
Reply • Share ›