Thinking Parallel, Part I: Collision Detection on the GPU

This series of posts aims to highlight some of the main differences between conventional programming and parallel programming on the algorithmic level, using broad-phase collision detection as an example. The first part will give some background, discuss two commonly used approaches, and introduce the concept of divergence. The second part will switch gears to hierarchical tree traversal in order to show how a good single-core algorithm can turn out to be a poor choice in a parallel setting, and vice versa. The third and final part will discuss parallel tree construction, introduce the concept of occupancy, and present a recently published algorithm that has specifically been designed with massive parallelism in mind.

Why Go Parallel?

The computing world is changing. In the past, Moore's law meant that the performance of integrated circuits would roughly double every two years, and that you could expect any program to automatically run faster on newer processors. However, ever since processor architectures hit the Power Wall around 2002, opportunities for improving the raw performance of individual processor cores have become very limited. Today, Moore's law no longer means you get faster cores—it means you get more of them. As a result, programs will not get any faster unless they can effectively utilize the ever-increasing number of cores.

Out of the current consumer-level processors, GPUs represent one extreme of this development. NVIDIA GeForce GTX 480, for example, can execute 23,040 threads in parallel, and in practice requires at least 15,000 threads to reach full performance. The benefit of this design point is that individual threads are very lightweight, but together they can achieve extremely high instruction throughput.

One might argue that GPUs are somewhat esoteric processors that are only interesting to scientists and performance enthusiasts working on specialized applications. While this may be true to some extent, the general direction towards more and more parallelism seems inevitable. Learning to write efficient GPU programs not only helps you get a substantial performance boost, but it also highlights some of the fundamental algorithmic considerations that I believe will eventually become relevant for all types of computing.

Many people think that parallel programming is hard, and they are partly right. Even though parallel programming languages, libraries, and tools have taken huge leaps forward in quality and productivity in recent years, they still lag behind their single-core counterparts. This is not surprising; those counterparts have evolved for decades while mainstream massively parallel general-purpose computing has been around only a few short years.

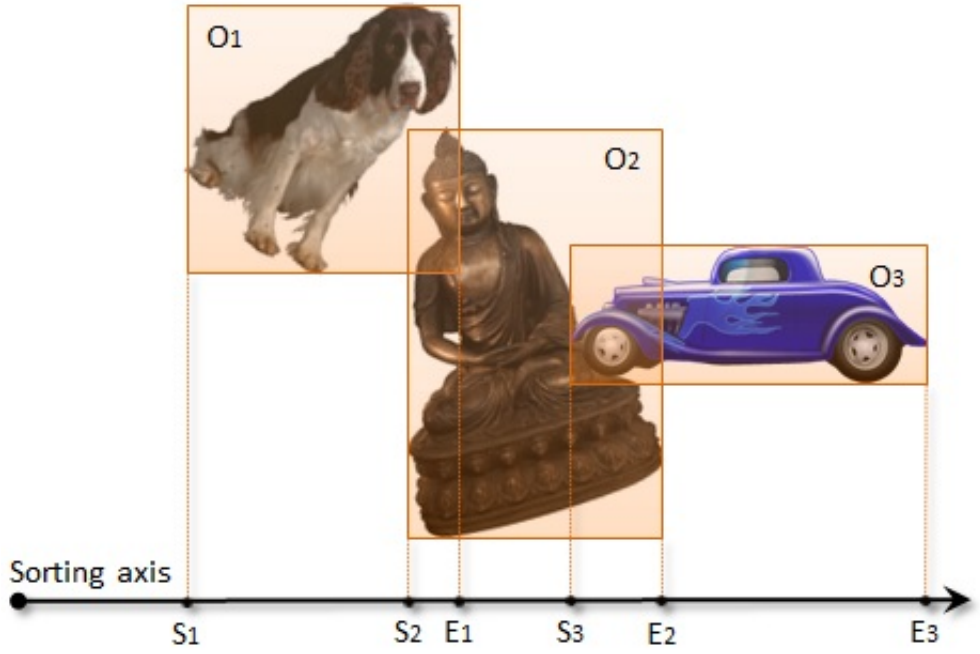
More importantly, parallel programming *feels* hard because the rules are different. We programmers have learned about algorithms, data structures, and complexity analysis, and we have developed intuition for what kinds of algorithms to consider in order to solve a particular problem. When programming for a massively parallel processor, some of this knowledge and intuition is not only inaccurate, but it may even be completely wrong. It's all about learning to "think parallel".

Sort and Sweep

Collision detection is an important ingredient in almost any physics simulation, including special effects in games and movies, as well as scientific research. The problem is usually divided into two phases, the broad phase and the narrow phase. The broad phase is responsible for quickly finding pairs of 3D objects than can potentially collide with each other, whereas the narrow phase looks more closely at each potentially colliding pair found in the broad phase to see whether a collision actually occurs. We will concentrate on the broad

phase, for which there are a number of well-known algorithms. The most straightforward one is sort and sweep.

The sort and sweep algorithm works by assigning an axis-aligned bounding box (AABB) for each 3D object, and projecting the bounding boxes to a chosen one-dimensional axis. After projection, each object will correspond to a 1D range on the axis, so that two objects can collide only if their ranges overlap each other.



In order to find the overlapping ranges, the algorithm collects the start points (S_1, S_2, S_3) and end points (E_1, E_2, E_3) of the ranges into an array, and sorts them along the axis. For each object, it then sweeps the list from the object's start and end points (e.g. S_2 and E_2) and identifies all objects whose start point lies between them (e.g. S_3). For each pair of objects found this way, the algorithm further checks their 3D bounding boxes for overlap, and reports the overlapping pairs as potential collisions.

Sort and sweep is very easy to implement on a parallel processor in three processing steps, synchronizing the execution after each step. In the first step, we launch one thread per object to calculate its bounding box, project it to the chosen axis, and write the start and end points of the projected range into fixed locations in the output array. In the second step, we sort the array into ascending order. Parallel sorting is an interesting topic in its own right, but the bottom line is that we can use parallel radix sort to yield a linear execution time with respect to the number of objects (given that there is enough work to fill the GPU). In the third step, we launch one thread per array element. If the element indicates an end point, the thread simply exits. If the element indicates a start point, the thread walks the array forward, performing overlap tests, until it encounters the corresponding end point.

The most obvious downside of this algorithm is that it may need to perform up to $O(n^2)$ overlap tests in the third step, regardless of how many bounding boxes actually overlap in 3D. Even if we choose the projection axis as intelligently as we can, the worst case is still prohibitively slow as we may need to test any given object against other objects that are arbitrarily far away. While this effect hurts both the serial and parallel implementations of the sort and sweep algorithm equally, there is another factor that we need to take into account when analyzing the parallel implementation: divergence.

Divergence

Divergence is a measure of whether nearby threads are doing the same thing or different things. There are two flavors: execution divergence means that the threads are executing different code or making different control flow decisions, while data divergence means that they are reading or writing disparate locations in memory. Both are bad for performance on parallel machines. Moreover, these are not just artifacts of current GPU architectures—performance of any sufficiently parallel processor will necessarily suffer from divergence to some degree.

On traditional CPUs we have learned to rely on the large data caches built right next to the execution pipeline. In most cases, this makes memory accesses practically free: the data we want to access is almost always already present in the cache. However, increasing the amount of parallelism by multiple orders of magnitude changes the picture completely. We cannot really add a lot more on-chip memory due to power and area constraints, so we will have to serve a much larger group of threads from a similar size cache. This is not a problem if the threads are doing more or less the same thing at the same time, since their combined working set is still likely to stay reasonably small. But if they are doing something completely different, the working set explodes, and the effective cache size from the perspective of one thread may shrink to just a handful of bytes.

The third step of the sort and sweep algorithm suffers mainly from execution divergence. In the implementation described above, the threads that happen to land on an end point will terminate immediately, and the remaining ones will walk the array for a variable number of steps. Threads responsible for large objects will generally perform more work than those responsible for smaller ones. If the scene contains a mixture of different object sizes, the execution times of nearby threads will vary wildly. In other words, the execution divergence will be high and the performance will suffer.

Uniform Grid

Another algorithm that lends itself to parallel implementation is uniform grid collision detection. In its basic form, the algorithm assumes that all objects are roughly equal in size. The idea is to construct a uniform 3D grid whose cells are at least the same size as the largest object.

In the first step, we assign each object to one cell of the grid according to the centroid of its bounding box. In

the second step, we look at the 3x3x3 neighborhood in the grid (highlighted). For every object we find in the neighboring cells (green check mark), we check the corresponding bounding box for overlap.

If all objects are indeed roughly the same size, they are more or less evenly distributed, the grid is not too large to fit in memory, and objects near each other in 3D happen to get assigned to nearby threads, this simple algorithm is actually very efficient. Every thread executes roughly the same amount of work, so the execution divergence is low. Every thread also accesses roughly the same memory locations as the nearby ones, so the data divergence is also low. But it took many assumptions to get this far. While the assumptions may hold in certain special cases, like the simulation of fluid particles in a box, the algorithm suffers from the so-called teapot-in-a-stadium problem that is common in many real-world scenarios.

The teapot-in-a-stadium problem arises when you have a large scene with small details. There are objects of all different sizes, and a lot of empty space. If you place a uniform grid over the entire scene, the cells will be too large to handle the small objects efficiently, and most of the storage will get wasted on empty space. To handle such a case efficiently, we need a hierarchical approach. And that's where things get interesting.

Discussion

We have so far seen two simple algorithms that illustrate the basics of parallel programming quite well. The common way to think about parallel algorithms is to divide them into multiple steps, so that each step is executed independently for a large number of items (objects, array elements, etc.). By virtue of synchronization, subsequent steps are free to access the results of the previous ones in any way they please.

We have also identified divergence as one of the most important things to keep in mind when comparing different ways to parallelize a given computation. Based on the examples presented so far, it would seem that minimizing divergence tilts the scales in favor of "dumb" or "brute force" algorithms—the uniform grid, which reaches low divergence, has to indeed rely on many simplifying assumptions in order to do that.

In my next post I will focus on hierarchical tree traversal as a means of performing collision detection, with an aim to shed more light on what it really means to optimize for low divergence.

∎∀

About the author: Tero Karras is a graphics research scientist at NVIDIA Research.

Parallel Forall is the NVIDIA Parallel Programming blog. If you enjoyed this post, subscribe to the Parallel Forall RSS feed!

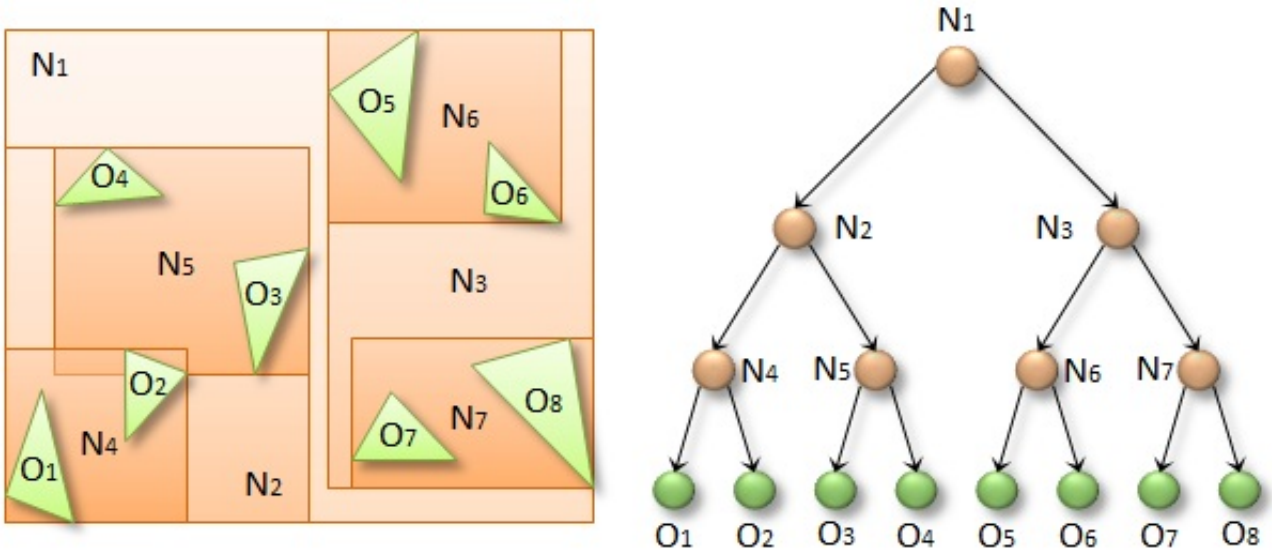
Thinking Parallel, Part II: Tree Traversal on the GPU

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)) );

        // Query overlaps a leaf node => report collision.
        if (overlapL && bvh.isLeaf(childL))
            list.add(queryObjectIdx, bvh.getObjectIdx(childL));

        if (overlapR && bvh.isLeaf(childR))
            list.add(queryObjectIdx, bvh.getObjectIdx(childR));

        // Query overlaps an internal node => traverse.
        bool traverseL = (overlapL && !bvh.isLeaf(childL));
        bool traverseR = (overlapR && !bvh.isLeaf(childR));

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

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.

∥∀

About the author: Tero Karras is a graphics research scientist at NVIDIA Research.

Parallel Forall is the NVIDIA Parallel Programming blog. If you enjoyed this post, subscribe to the Parallel Forall RSS feed!

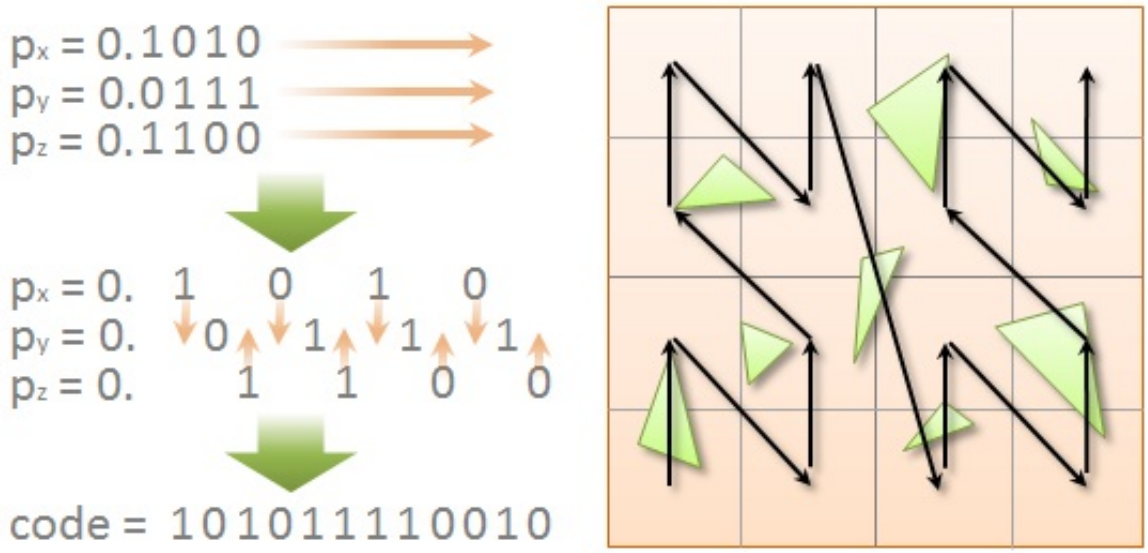
Thinking Parallel, Part III: Tree Construction on the GPU

In part II of this series, we looked at hierarchical tree traversal as a means of quickly identifying pairs of potentially colliding 3D objects and we demonstrated how optimizing for low divergence can result in substantial performance gains on massively parallel processors. Having a fast traversal algorithm is not very useful, though, unless we also have a tree to go with it. In this part, we will close the circle by looking at tree building; specifically, parallel bounding volume hierarchy (BVH) construction. We will also see an example of an algorithmic optimization that would be completely pointless on a single-core processor, but leads to substantial gains in a parallel setting.

There are many use cases for BVHs, and also many ways of constructing them. In our case, construction speed is of the essence. In a physics simulation, objects keep moving from one time step to the next, so we will need a different BVH for each step. Furthermore, we know that we are going to spend only about 0.25 milliseconds in traversing the BVH, so it makes little sense to spend much more on constructing it. One well-known approach for handling dynamic scenes is to essentially recycle the same BVH over and over. The basic idea is to only recalculate the bounding boxes of the nodes according to the new object locations while keeping the hierarchical structure of nodes the same. It is also possible to make small incremental modifications to improve the node structure around objects that have moved the most. However, the main problem plaguing these algorithms is that the tree can deteriorate in unpredictable ways over time, which can result in arbitrarily bad traversal performance in the worst case. To ensure predictable worst-case behavior, we instead choose to build a new tree from scratch every time step. Let's look at how.

EXPLOITING THE Z-ORDER CURVE

The most promising current parallel BVH construction approach is to use a so-called linear BVH (LBVH). The idea is to simplify the problem by first choosing the order in which the leaf nodes (each corresponding to one object) appear in the tree, and then generating the internal nodes in a way that respects this order. We generally want objects that located close to each other in 3D space to also reside nearby in the hierarchy, so a reasonable choice is to sort them along a space-filling curve. We will use the Z-order curve for simplicity.



The Z-order curve is defined in terms of *Morton codes*. 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 top left part of the figure. First, we take the fractional part of each coordinate and expand it by inserting two "gaps" after each bit. Second, we 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 (a 2D representation is shown on the right-hand side of the figure). In practice, we can determine the order of the leaf nodes by assigning a Morton code for each object and then sorting the objects accordingly. As mentioned in the context of sort and sweep in part I, parallel radix sort is just the right tool for this job. A good way to assign the Morton code for a given object is to use the centroid point of its bounding box, and express it relative to the bounding box of the scene. The expansion and interleaving of bits can then be performed efficiently by exploiting the arcane bit-swizzling properties of integer multiplication, as shown in the following code.

```
// Expands a 10-bit integer into 30 bits
// by inserting 2 zeros after each bit.
unsigned int expandBits(unsigned int v)
{
    v = (v * 0x00010001u) & 0xFF0000FFu;
    v = (v * 0x00000101u) & 0x0F00F00Fu;
    v = (v * 0x00000011u) & 0xC30C30C3u;
    v = (v * 0x00000005u) & 0x49249249u;
    return v;
}

// Calculates a 30-bit Morton code for the
// given 3D point located within the unit cube [0,1].
unsigned int morton3D(float x, float y, float z)
{
    x = min(max(x * 1024.0f, 0.0f), 1023.0f);
    y = min(max(y * 1024.0f, 0.0f), 1023.0f);
    z = min(max(z * 1024.0f, 0.0f), 1023.0f);
    unsigned int xx = expandBits((unsigned int)x);
    unsigned int yy = expandBits((unsigned int)y);
    unsigned int zz = expandBits((unsigned int)z);
    return xx * 4 + yy * 2 + zz;
}
```

In our example dataset with 12,486 objects, assigning the Morton codes this way takes **0.02** milliseconds on GeForce GTX 690, whereas sorting the objects takes **0.18** ms. So far so good, but we still have a tree to build.

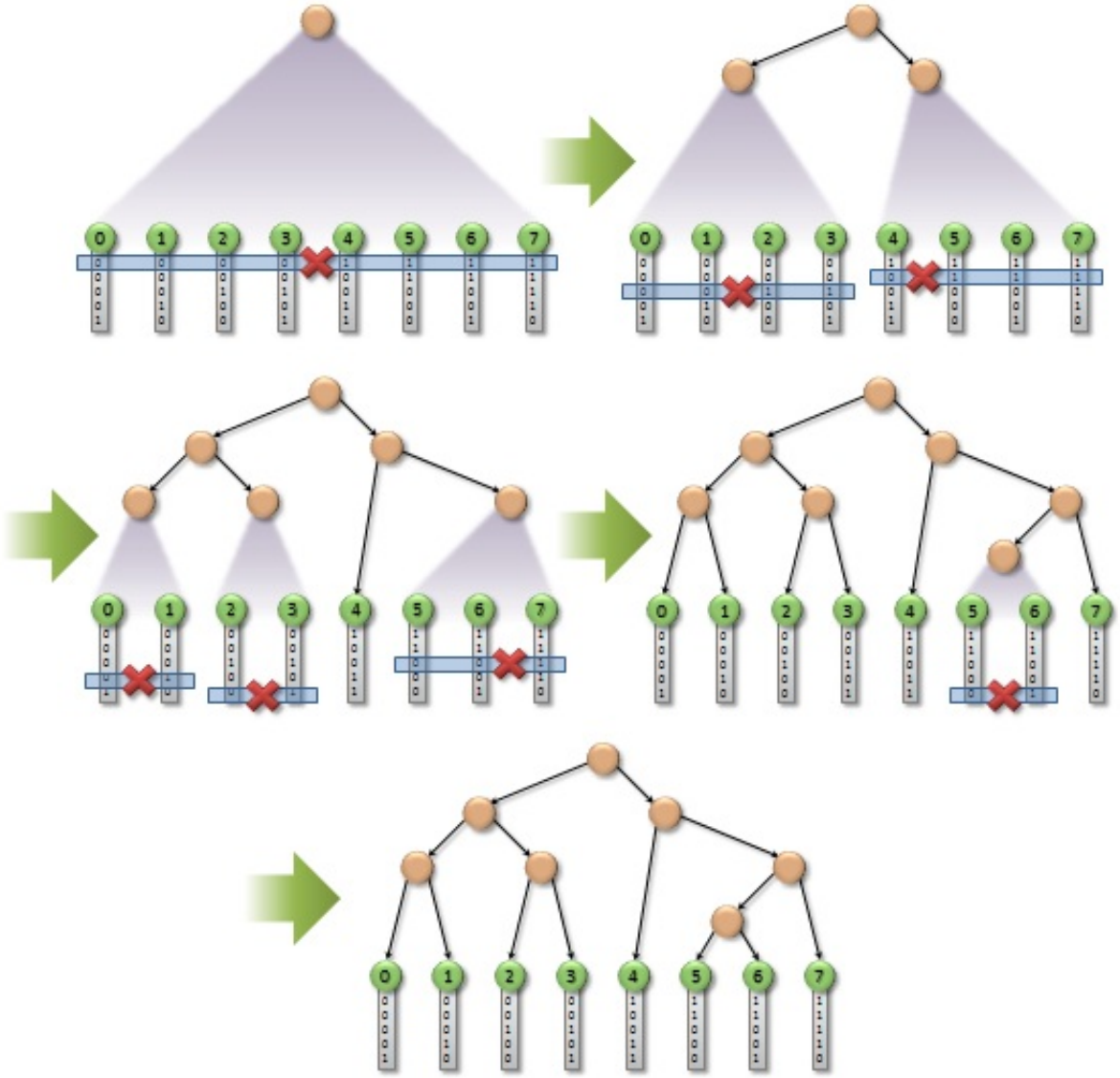
TOP-DOWN HIERARCHY GENERATION

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 node as just a linear range over them. To illustrate this, suppose that we have N leaf nodes in total. The root node contains all of them, i.e. it covers the range $[0,N-1]$. The left child of the root must then cover the range $[0,\gamma]$, for some appropriate choice of γ, and the right child covers the range $[\gamma+1,N-1]$. We can continue this all the way down to obtain the following recursive algorithm.

```
Node* generateHierarchy( unsigned int* sortedMortonCodes,
                         int*          sortedObjectIDs,
                         int           first,
                         int           last)
{
```

```
    // Single object => create a leaf node.

    if (first == last)
        return new LeafNode(&sortedObjectIDs[first]);

    // Determine where to split the range.

    int split = findSplit(sortedMortonCodes, first, last);

    // Process the resulting sub-ranges recursively.

    Node* childA = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                     first, split);
    Node* childB = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                     split + 1, last);
    return new InternalNode(childA, childB);
}
```

We start with a range that covers all objects (`first`=0, `last`=N-1), and determine an appropriate position to split the range in two (`split`=γ). We then repeat the same thing for the resulting sub-ranges, and generate a hierarchy where each such split corresponds to one internal node. The recursion terminates when we encounter a range that contains only one item, in which case we create a leaf node.



The only remaining question is how to choose γ. LBVH determines γ according to the highest bit that differs between the Morton codes within the given range. In other words, we aim to partition the objects so that the highest differing bit will be zero for all objects in `childA`, and one for all objects in `childB`. The intuitive reason that this is a good idea is that partitioning objects by the highest differing bit in their Morton codes corresponds to classifying them on either side of an axis-aligned plane in 3D. In practice, the most efficient way to find out where the highest bit changes is to use binary search. The idea is to maintain a current best guess for the position, and try to advance it in exponentially decreasing steps. On each step, we check whether the proposed new position would violate the requirements for `childA`, and either accept or reject it accordingly. This is illustrated by the following code, which uses the `__clz()`intrinsic function available in NVIDIA Fermi and Kepler GPUs to count the number of leading zero bits in a 32-bit integer.

```
int findSplit( unsigned int* sortedMortonCodes,
               int           first,
               int           last)
{
    // Identical Morton codes => split the range in the middle.

    unsigned int firstCode = sortedMortonCodes[first];
    unsigned int lastCode = sortedMortonCodes[last];
```

```
    if (firstCode == lastCode)
        return (first + last) >> 1;

    // Calculate the number of highest bits that are the same
    // for all objects, using the count-leading-zeros intrinsic.

    int commonPrefix = __clz(firstCode ^ lastCode);

    // Use binary search to find where the next bit differs.
    // Specifically, we are looking for the highest object that
    // shares more than commonPrefix bits with the first one.

    int split = first; // initial guess
    int step = last - first;

    do
    {
        step = (step + 1) >> 1; // exponential decrease
        int newSplit = split + step; // proposed new position

        if (newSplit < last)
        {
            unsigned int splitCode = sortedMortonCodes[newSplit];
            int splitPrefix = __clz(firstCode ^ splitCode);
            if (splitPrefix > commonPrefix)
                split = newSplit; // accept proposal
        }
    }
    while (step > 1);

    return split;
}
```

How should we go about parallelizing this kind of recursive algorithm? One way is to use the approach presented by Garanzha et al., which processes the levels of nodes sequentially, starting from the root. The idea is to 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. On a given level, we launch one thread for each node that falls into this range. The thread starts by reading `first`and `last`from the node array and calling `findSplit()`. It then appends the resulting child nodes to the same node array using an atomic counter and writes out their corresponding subranges. This process iterates so that each level outputs the nodes contained on the next level, which then get processed in the next round.

OCCUPANCY

The algorithm just described (Garanzha et al.) is surprisingly fast when there are millions of objects. The algorithm spends most of the execution time at the bottom levels of the tree, which contain more than enough work to fully employ the GPU. There is some amount of data divergence on the higher levels, as the threads

are accessing distinct parts of the Morton code array. But those levels are also less significant considering the overall execution time, since they do not contain as many nodes to begin with. In our case, however, there are only 12K objects (recall the example seen from the last post). Note that this is actually less than the number of threads we would need to fill our GTX 690, even if we were able to parallelize everything perfectly. GTX 690 is a dual-GPU card, where each of the two GPUs can run up to 16K threads in parallel. Even if we restrict ourselves to only one of the GPUs—the other one can e.g. handle rendering while we do physics—we are still in danger of running low on parallelism.

The top-down algorithm takes **1.04** ms to process our workload, which is more than twice the total time taken by all the other processing steps. To explain this, we need to consider another metric in addition to divergence: *occupancy*. Occupancy is a measure of how many threads are executing on average at any given time, relative to the maximum number of threads that the processor can theoretically support. When occupancy is low, it translates directly to performance: dropping occupancy in half will reduce performance by 2x. This dependence gets gradually weaker as the number of active threads increases. The reason is that when occupancy is high enough, the overall performance starts to become limited by other factors, such as instruction throughput and memory bandwidth.

To illustrate, consider the case with 12K objects and 16K threads. If we launch one thread per object, our occupancy is at most 75%. A bit low, but not by any means catastrophic. How does the top-down hierarchy generation algorithm compare to this? There is only one node on the first level, so we launch only one thread. This means that the first level will run at 0.006% occupancy! The second level has two nodes, so it runs at 0.013% occupancy. Assuming a balanced hierarchy, the third level runs at 0.025% and the fourth at 0.05%. Only when we get to the 13th level, can we even hope to reach a reasonable occupancy of 25%. But right after that we will already run out of work. These numbers are somewhat discouraging—due to the low occupancy, the first level will cost roughly as much as the 13th level, even though there are 4096 times fewer nodes to process.

FULLY PARALLEL HIERARCHY GENERATION

There is no way to avoid this problem without somehow changing the algorithm in a fundamental way. Even if our GPU supports dynamic parallelism (as NVIDIA Tesla K20 does), we cannot avoid the fact that every node is dependent on the results of its parent. We have to finish processing the root before we know which ranges its children cover, and we cannot even hope to start processing them until we do. In other words, regardless of how we implement top-down hierarchy generation, the first level is doomed to run at 0.006% occupancy. Is there a way to break the dependency between nodes?

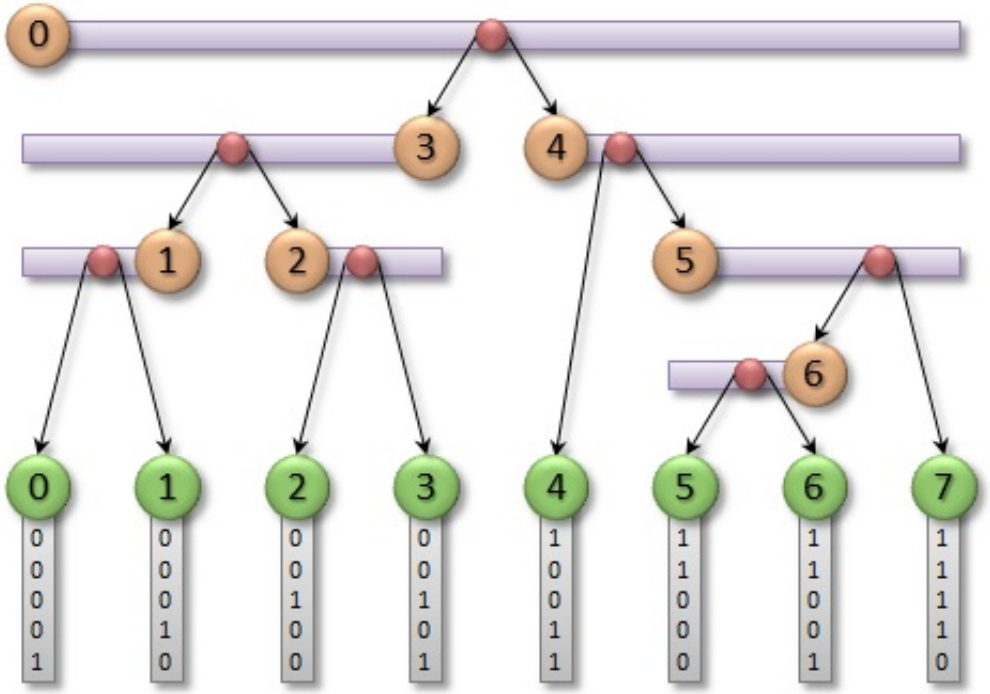
In fact there is, and I recently presented the solution at High Performance Graphics 2012 (paper, slides). The idea is to number the internal nodes in a very specific way that allows us to find out which range of objects any given node corresponds to, without having to know anything about the rest of the tree. Utilizing the fact that any binary tree with *N* leaf nodes always has exactly *N-1* internal nodes, we can then generate the entire hierarchy as illustrated by the following pseudocode.

```
Node* generateHierarchy( unsigned int* sortedMortonCodes,
                         int*          sortedObjectIDs,
                         int           numObjects)
{
    LeafNode* leafNodes = new LeafNode[numObjects];
    InternalNode* internalNodes = new InternalNode[numObjects - 1];

    // Construct leaf nodes.
```

```
// Note: This step can be avoided by storing
// the tree in a slightly different way.

for (int idx = 0; idx < numObjects; idx++) // in parallel
    leafNodes[idx].objectID = sortedObjectIDs[idx];

// Construct internal nodes.

for (int idx = 0; idx < numObjects - 1; idx++) // in parallel
{
    // Find out which range of objects the node corresponds to.
    // (This is where the magic happens!)

    int2 range = determineRange(sortedMortonCodes, numObjects, idx);
    int first = range.x;
    int last = range.y;

    // Determine where to split the range.

    int split = findSplit(sortedMortonCodes, first, last);

    // Select childA.

    Node* childA;
    if (split == first)
        childA = &leafNodes[split];
    else
        childA = &internalNodes[split];

    // Select childB.

    Node* childB;
    if (split + 1 == last)
        childB = &leafNodes[split + 1];
    else
        childB = &internalNodes[split + 1];

    // Record parent-child relationships.

    internalNodes[idx].childA = childA;
    internalNodes[idx].childB = childB;
    childA->parent = &internalNodes[idx];
    childB->parent = &internalNodes[idx];
}

// Node 0 is the root.

return &internalNodes[0];
```

```
}
```

The algorithm simply allocates an array of *N*-1 internal nodes, and then processes all of them in parallel. Each thread starts by determining which range of objects its node corresponds to, with a bit of magic, and then proceeds to split the range as usual. Finally, it selects children for the node according to their respective sub-ranges. If a sub-range has only one object, the child must be a leaf so we reference the corresponding leaf node directly. Otherwise, we reference another internal node from the array.



The way the internal nodes are numbered is already evident from the pseudocode. The root has index 0, and the children of every node are located on either side of its split position. Due to some nice properties of the sorted Morton codes, this numbering scheme never results in any duplicates or gaps. Furthermore, it turns out that we can implement `determineRange()`in much the same way as `findSplit()`, using two similar binary searches over the nearby Morton codes. For further details on how and why this works, please see the paper.

How does this algorithm compare to the recursive top-down approach? It clearly performs more work—we now need three binary searches per node, whereas the top-down approach only needs one. But it does all of the work completely in parallel, reaching the full 75% occupancy in our example, and this makes a huge difference. The execution time of parallel hierarchy generation is merely **0.02** ms—a 50x improvement over the top-down algorithm!

You might think that the top-down algorithm should start to win when the number of objects is sufficiently high, since the lack of occupancy is no longer a problem. However, this is not the case in practice—the parallel algorithm consistently performs better on all problem sizes. The explanation for this is, as always, divergence. In the parallel algorithm, nearby threads are always accessing nearby Morton codes, whereas the top-down algorithm spreads out the accesses over a wider area.

BOUNDING BOX CALCULATION

Now that we have a hierarchy of nodes in place, the only thing left to do is to assign a conservative bounding box for each of them. The approach I adopt in my paper is to do a parallel bottom-up reduction, where each

thread starts from a single leaf node and walks toward the root. To find the bounding box of a given node, the thread simply looks up the bounding boxes of its children and calculates their union. To avoid duplicate work, the idea is to use an atomic flag per node to terminate the first thread that enters it, while letting the second one through. This ensures that every node gets processed only once, and not before both of its children are processed. The bounding box calculation has high execution divergence—only half of the threads remain active after processing one node, one quarter after processing two nodes, one eigth after processing three nodes, and so on. However, this is not really a problem in practice because of two reasons. First, bounding box calculation takes only **0.06** ms, which is still reasonably low compared to e.g. sorting the objects. Second, the processing mainly consists of reading and writing bounding boxes, and the amount of computation is minimal. This means that the execution time is almost entirely dictated by the available memory bandwidth, and reducing execution divergence would not really help that much.

SUMMARY

Our algorithm for finding potential collisions among a set of 3D objects consists of the following 5 steps (times are for the 12K object scene used in the previous post).

1. **0.02** ms, one thread per object: Calculate bounding box and assign Morton code.
2. **0.18** ms, parallel radix sort: Sort the objects according to their Morton codes.
3. **0.02** ms, one thread per internal node: Generate BVH node hierarchy.
4. **0.06** ms, one thread per object: Calculate node bounding boxes by walking the hierarchy toward the root.
5. **0.25** ms, one thread per object: Find potential collisions by traversing the BVH.

The complete algorithm takes **0.53** ms, out of which 53% goes to tree construction and 47% to tree traversal.

DISCUSSION

We have presented a number of algorithms of varying complexity in the context of broad-phase collision detection, and identified some of the most important considerations when designing and implementing them on a massively parallel processor. The comparison between independent traversal and simultaneous traversal illustrates the importance of divergence in algorithm design—the best single-core algorithm may easily turn out to be the worst one in a parallel setting. Relying on time complexity as the main indicator of a good algorithm can sometimes be misleading or even harmful—it may actually be beneficial to do more work if it helps in reducing divergence.

The parallel algorithm for BVH hierarchy generation brings up another interesting point. In the traditional sense, the algorithm is completely pointless—on a single-core processor, the dependencies between nodes were not a problem to begin with, and doing more work per node only makes the algorithm run slower. This shows that parallel programming is indeed fundamentally different from traditional single-core programming: it is not so much about porting existing algorithms to run on a parallel processor; it is about re-thinking some of the things that we usually take for granted, and coming up with new algorithms specifically designed with massive parallelism in mind.

And there is still a lot to be accomplished on this front.

∥∀

About the author: Tero Karras is a graphics research scientist at NVIDIA Research.
