High Performance Ray Tracing

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Spring 2017
Rendering Performance

- Lets assume we are using a general purpose path tracer to render an image
- Lets also assume that we are making use of efficient BRDF sampling, light sampling, etc.
- We still need to generate at least 100 or more paths per pixel to produce a reasonable quality image
- If each path has an average of 5 bounces, and generates one direct and one indirect sample per bounce, then a path uses an average of 11 rays
- If we have a 2000 x 1500 image, we have 3 million pixels
- Altogether, we would expect to need to trace 3.3 billion rays to generate a decent image
- If we want to render at 30 Hz, that’s about 100 billion rays per second
Rendering Performance

• There are several strategies for improving ray/path tracing performance
• Good quality pixel sampling as well as good importance sampling of the BRDFs helps a lot
• There are various other algorithmic improvements that can be done, including the use of hybrid approaches like photon mapping
• Beyond fundamental algorithmic improvements, there are various hardware-specific approaches we can take
Hardware Based Optimization

- Assuming we are using good algorithms, we need to turn our attention to various hardware and implementation details that can make ray tracing faster.
- One fundamental issue that often comes up in this discussion is the subject of CPU vs. GPU ray tracing.
CPU vs. GPU

- The movie industry tends to use CPU based rendering exclusively.
- This is mainly because of huge memory requirements, as well as lots of previous time invested in CPU code.
- Consider that a current high-end GPU has maybe 16 GB of video memory.
- Typical movie scenes often require 50 GB of memory just for geometry, plus maybe another 1-2 TB of memory for textures.
- This is obviously way outside the range that GPU’s can store on board, and so CPU based techniques combined with memory paging are required.
CPU vs. GPU

• For non-movie applications, such as real time, or at least interactive applications, GPUs may be a good choice

• State of the art GPU rendering algorithms will generally outperform state of the art CPU rendering for reasonably complex scenes, but not by huge amounts

• For simple test cases though, GPUs can significantly outperform CPUs
GPU Ray Tracing

• One of the biggest challenges with ray tracing on the GPU is that GPU memory architectures are not nearly as well set up for random access as CPU memories

• The GPU performs well on predictable, regular memory access patterns

• Standard path tracing is particularly bad on memory access patterns, as rays can bounce around randomly throughout the entire environment

• Much of the research on optimizing GPU ray tracing is focused on GPU-specific spatial data structures (variants on BVH, KD trees, etc.)
GPU Ray Tracing

• nVidia offers a ray tracing system called OptiX, that can trace up to 350M rays/second in a complex scene, on a GTX Titan

• It is a modular system that has routines for building spatial data structures for meshes, tracing rays against those data structures, and more

• There are various other GPU ray tracing systems out there as well, including some open source packages
Multi-Processor Rendering

• One way to speed up rendering with either CPUs or GPUs is to use more than one
• As ray based rendering tends to be very parallel friendly, you can split the work up across multiple chips within a computer, or even multiple computers on a network
• This is common for movie production, where they may have a local cluster of 100 CPUs and any particular artist who needs to render a test image can send a job over to the cluster and get an image back quickly
• One can even distribute the job to various different pieces of hardware, such as a mix of CPUs and GPUs
Multi-Core Rendering

- Many modern CPUs contain multiple independent compute cores
- It is common for Intel chips to have 2 or 4 cores these days, although some have up to 8, each with its own cache
- The caches are synced up so that if one core writes to a memory location, it will (very quickly) be available to the other cores, even if they already have the memory location cached
- This allows for very easy multi-threaded programming, as all threads can access whatever memory they want
- As ray tracing involves reading from a large pool of memory, but then writing to a single pixel at a time, it is no problem to have different threads rendering neighboring pixels into the same image
Modern Intel processors also have a technology called HyperThreading that allows each core to rapidly switch between two different threads to allow it to switch when one stalls for even a single clock cycle.

With HyperThreading on a 4 core CPU, you would typically have 8 threads rendering (each thread will have half the cache it would have without HT though).

HyperThreading definitely won’t double the performance, but it usually improves it by 10% or so.

In some cases, however, the reduced cache size can cause the performance to decrease.

It’s very easy to enable/disable, so if you’re already multithreading, it’s very simple to try it out.
Multi-Thread Rendering

• When rendering with multiple threads on a single CPU, it is important to decide how to divide the work between the threads
• A common approach is to divide the image up into tiles, of maybe 8x8 pixels or so, and let threads render entire tiles
• When a tile is complete, the thread just renders the next available tile
• This accomplishes a few things
  – Gives each thread a reasonable sized task to reduce the overhead of the thread management
  – Keeps the tasks small enough so that no one tile should end up holding up the entire render
  – Helps improve caching performance, as rendering a tile will typically involve traversing a subset of the geometry (This is more true for camera rays, shadow rays and specular reflections. It is less true for path-traced diffuse surfaces, which scatter rays all over)
Multi-Thread Rendering

- There are various ways to implement this in code
- A straightforward way is to use standard library threads, `std::thread`
- This can be combined with atomic variables using `std::atomic`
- Atomic variables are protected at the hardware level to prevent different threads from trying to change the variable at the same time
- This allows a single global atomic int to indicate the next tile to render
- Each individual thread reads and decrements the atomic (in a single hardware operation). The read value tells the thread which tile to render. The process decrements the value, so that another thread will later read it and render the next tile. When the atomic value gets to zero, there are no more to render, and the image is finished when all threads finish their current tile
- There is also a ‘master’ thread that runs the main program and kicks off the multi-thread render. However, this master thread uses no resources once the render starts, as it will just wait for the render threads to finish
std::atomic<int> tile=100; // assuming 100 tiles
BitMap image;

void RenderImage() {
    const int numThreads=4;
    std::thread *thread[numThreads];
    for(int i=0;i<4;i++)
        thread[i]=new std::thread(ThreadFunction);

    for(int i=0;i<4;i++)
        thread[i]->join();
}

void ThreadFunction() {
    while(1) {
        int tileNum=tile--;
        if(tileNum<=0) return;

        // Render tile ‘tileNum’ and store data in ‘image’
    }
}
SIMD Processing

- In addition to taking advantage of multicore CPUs and multiple threads per core, we can also use the SIMD computing capabilities (single instruction, multiple data)
- Intel chips from the early 2000s have a technology called SSE, or Streaming SIMD Extension
- SSE extends the standard floating point capability to operate on vectors of 4 floats (or 2 double-precision floats)
- It has its own set of 128-bit registers, as well as a set of instructions for add, subtract, multiply, divide, sqrt, min, max, dot, and much more
- In the 2010’s, Intel added the AVX, AVX2, and AVX512 instruction sets, which extend the vector operations to vectors of 8, 16, and ultimately 32 floats
SIMD Programming

• In order to use the SIMD instructions, you can hope that your compiler is smart enough to figure it out by itself and use the SIMD where it feels it should.

• Unfortunately, compilers are never as smart as you want them to be, so this will not work.

• In Windows/Intel programming, you can do one of two things to use the SIMD instructions:
  – Use compiler ‘intrinsics’ from C++
  – Use inline assembler
SIMD Intrinsics

• Some operating systems (like Windows/Intel) provide special C functions that directly compile to single assembly instructions (or sometimes 2 instructions)
• These are called ‘intrinsics’ and they look like any other function, but they typically have ugly names like:

  ___m128 ___mm_add_ps(___m128 a, ___m128 b);
Inline Assembly

• In VisualStudio, you can do things like this:

```c
void Function() {
    __asm {
        addps       xmm0,xmm1
    }
}
```
Linear Tone Mapping

• The next example is linear tone mapper that takes a color as a vector of 4 floats (RGBA) and produces a 32 bit integer
• It performs a gamma correction assuming a gamma of 2.0, then scales the color by 255 and clamps it in the 0-255 range
• It then gets converted to 4 32-bit integers, then packed into 4 16-bit values, and finally into 4 8-bit values in a single int
Linear Tone Mapping

```c
int ToneMap(const rtColor &color) {
    static float load[2] = {255.0f, 255.0f};

    __asm {
        // Set up variables
        move    ax, dword ptr [color]  // Load address of color
        movups  xmm2, xmmword ptr [eax]  // Set xmm2 to input color
        sqrtps  xmm2, xmm2  // Gamma correction (gamma=2)
        movddup xmm1, xmmword ptr ds:[load]  // Set xmm1 to all 255's
        xorps   xmm0, xmm0  // Set xmm0 to all 0's

        // Scale & clamp color (xmm2) to [0...255] range
        maxps   xmm2, xmm0  // Clip color at 0
        mulps   xmm2, xmm1  // Multiply color by 255
        minps   xmm2, xmm1  // Clip color at 255

        // Convert 4 floats to 32 bit int
        cvtps2dq xmm2, xmm2  // Convert 4 floats to 4 32-bit ints
        packssdw xmm2, xmm2  // Pack 4 32-bit ints into 4 16-bit ints
        packuswb xmm2, xmm2  // Pack 4 16-bit ints into 4 8-bit ints

        // Finish
        movde   ax, xmm2  // Extract lower 32 bit int to return
    }
}
```

Note that this could be even faster if it operated on an array of colors instead of one at a time. This would save having to load the 255.0 into registers each time, as well as remove function call & return overhead. It would also allow for the Intel out-of-order execution scheduling for flexibility.
Ray Tracing with SIMD Instructions

• Dispatching multiple floating point operations per clock cycle is a big win for heavy computations
• The trick however is to be able to use all of the parallel floats
• This isn’t so hard with 4, but gets harder with 8, 16, or 32
• For example, using SIMD operations on 4 floats at a time, one can build a box-tree data structure with 4 boxes contained per box (instead of 2 as we’ve done so far). To test for ray intersection, the ray can test all 4 boxes at once. This tends to have a big payoff, as it performs multiple operations in parallel and cuts the tree depth in half
• The payoff isn’t as big with 8, 16, or more though
Cache Optimization

• Even if we are very effective in using every possible clock cycle on every core, we still often run into a big challenge: memory bottlenecks
• With path tracing, random memory access is one of the biggest challenges to efficient rendering
• Any particular pixel might create paths that bounce around and hit any particular triangle in the whole scene
• This is about the worst possible case for cache performance
• This is one of the areas of current research in high performance rendering, and many movie studios struggle with this on their own internal rendering systems
Cache Optimization

• One strategy to improve this is to try to group nearby rays together, as it is expected that they will need to test the same geometry (and thus examine the same regions of memory)
• This adds a lot of complexity to the rendering process. Instead of a pixel just rendering all of the rays/paths it needs, rays get added to a big queue
• A ray can be thought of as a point in 5D space (3 for the origin, and 2 more for the direction)
• Similar rays will be nearby in 5D space
• As we queue up all the rays we need to trace, we group them into batches that are close in 5D space
• Rays are then traced in batches, hoping to minimize cache misses
• When a ray hits and wants to generate reflections, etc., those are just added to the same queue and batched with other rays