By: Krunal Patel - k5patel@me.com

Prof. Hanspeter Pfister

TF: Kevin Dale

CS264 - Massively Parallel Computing

December 4, 2009

Regular-Grid Accelerated Ray-Tracing Using CUDA

Problem Definition

Ray-tracing is a method used to draw realistic images on a computer. One can find examples of ray-traced images when viewing Disney/Pixar movies like Toy Story. The underlying algorithm behind ray-tracing is to decompose a picture into pixels, send rays through the pixels, intersect the rays with the objects in the scene, launch any secondary rays, and emit a final color for the original pixel.

Ray-tracing is known to be an embarrassingly parallel problem. Each pixel can be completely calculated independently of any other neighboring pixels. So a mechanism that allows spawning of multiple threads to parallelize the ray-tracing tasks would prove quite useful.

Hence we have employed NVIDIA's CUDA based technology to exemplify the performance gains to be had with GPU hardware acceleration for ray-tracing. The goal is not to implement a full-blown ray-tracer, but instead, highlight the intricacies of the CUDA technology. Ray-tracing is a very branchy algorithm, and we wanted to see how well CUDA stands up to the task.

Our ray-tracer supports spheres, rectangles, and triangles as the basic primitives. Multiple light sources as well as shadows are available. Anti-aliasing via regular sampling is available too. Phong materials are utilized with support for glossy specular highlights. All world objects are composed into build scenes. Nine such scenes are available for the reader to test. There are command-line options to the “gpuraytracer” executable to further alter the rendering behavior of the build scenes. Support for 3D model meshes by utilizing the PLY file format is available too.

For comparison purposes, a CPU version of the application is also included. This is embedded within the gpuraytracer executable. We also provide an error checker that calculates the differences between the output of the CPU and GPU execution.

Finally, though we get substantial improvements in rendering time taking a brute approach to ray-tracing, we also employ an acceleration scheme to the whole process. Obviously the brute approach is to traverse the whole scene-graph per pixel and the ray-tracing steps to find any intersections with world objects. While this is naive, I will show later that it still yielded tremendous performance over a CPU. Our regular grid acceleration scheme subdivides the world into axis-aligned cells. These cells contain the world primitive objects that are within it’s bounding box. The grid creation is done on the CPU. GPU grid traversal (and CPU for that matter) involve shooting a ray and intersecting with the cells and primitives that come in it’s path. Once such an intersection is found, it is no longer necessary to continue with finding another intersection as we are guaranteed that the one just found is the closest. Thereafter shading proceeds as normal. More information about regular grid acceleration scheme can be found in

the book “Ray Tracing from the Ground Up”[1] (herein known as RTGU). We will show later that there was even more gains to be had using regular grids.

[1] http://www.raytracegroundup.com/

Data Details

The data for this project comprised of the build scenes previously mentioned. To highlight the pros and cons of CUDA, we required **lots** of data to be sent to the GPU. To achieve this task, our test scenes contain thousands of polygons and primitive objects (known as GeometricObject in code-speak). In some of our scenes, we send over 50MB of data to the GPU to be processed. Our most highest quality meshes are the Stanford Bunny and Horse model each weighing in at 69K polygons and 97K polygons respectively. The Stanford Bunny model was actually acquired from a real-life bunny using a 3D-scanner. Triangle meshes were essential in allowing to test for the arithmetic intensity of our GPU ray-tracer.

Spheres not only look pretty, but also helped us throughly test the validity of the grid acceleration scheme. With that in mind, we have support for tens of thousands of spheres in our build scenes. The data here would highlight different aspects of brute versus grid acceleration and how the CPU fairs against the GPU due to the high amount of data transfers involved. One issue with spheres was it's intersection code. Ideally, a ray-tracer likes to do all intersections using double-precision and shading can be done with single-precision. Since we are limited to only using single-precision floats for all our CUDA work, there were odd artifacts as the rims of the spheres which the author believes most likely has to do with tangent ray intersections not being handled fully correctly with single-precision. Build01 exemplifies this issue.

Originally there were plans to support Planes, but it was later dropped because they couldn't fit into a grid acceleration scheme correctly. Since planes are infinite, and don't have any bounding box, they couldn't exist in the cells of a grid. As a future improve-

ment, we could handle such special objects outside of the grid in a separate data-structure. So instead we opted to do the next best thing, and that was to handle rectangle primitives.

I obtained the sources for my data from the RTGU website. This greatly assisted me in development since I had a source reference to compare my progress with. The PLY file reading support had to be updated to take into account platform-dependent line-endings. I had to write code to support saving of BMP files. While this was already somewhat in place from the previous assignments, I had to modify it so that there wasn't an assumption of a source input file to base the BMP header information on.

The most interesting aspect of the data was the transfer of it from host to device. It was the author's opinion to use as pinned memory as much as possible, guaranteeing faster transfer of data to device memory. This worked just fine on the author's development environment. Though on Resonance, the pinned memory transfers took extensively longer when compared with our development box (iMac 3.0 Ghz, 4 Gigs Ram, 8800GS). So it was decided to not use pinned memory by default, and then we got much better performance on Resonance. An option is available at runtime to allow pinned memory (-p=1).

Design Decisions

Off the bat, the most important aspect of our ray-tracing was that it be used as a comparative tool. We wanted to highlight the gains of using NVIDIA CUDA hardware technology when compared with CPU. We also wanted to show some of the downfalls with GPU computing too, namely memory operations. With all this in mind, we based our code from that of RTGU. We reorganized, optimized, and extended the original ray-tracer to fit into the mould of the CUDA run-time SDK.

An important design decision was to allow the sharing of code/execution paths between CPU and GPU. Hence the code for intersections and shading computation is shared by both the CPU and GPU e.g. declare a method with __host__ __device__ attributes so that it can be called from both parties. This approach was taken because the actual math computations couldn't be further optimized with GPU code since they were quite simple. But a more important reason was to highlight the awesome throughput of the GPU even when the underlying algorithm hadn't changed. Obviously though, the GPU global kernel was highly tuned and important data structures had to be shuffled around in order to work effectively on the GPU.

The hybrid code approach mentioned above allowed for better software engineering practices. We were able to use standard C++ classes, methods, and member variables interchangeably. An important diversion from the base RTGU code was how we handled multiple GeometricObjects (our primitives). The most sense full thing to do here is to use C++ inheritance and polymorphism. A base class than can be used in all the different containers e.g. grid cells, without worrying about the specific instance of the primitive. Intersection and shading code would just follow naturally based on the primi-

tive too. Anyways, such a flexible solution wasn't an option for us since CUDA 2.3 SDK didn't support C++ polymorphic behavior or inheritance. We had to resort to old-school methodologies and impose a Type field in our GeometricObject. Hence there is only really one GeometricObject that encapsulates a sphere, mesh, rectangle, etc. Same applies for Materials.

The obvious drawback to this sort of approach is that it unnecessarily increased the memory footprint. For example, a scene with only spheres in it doesn't really require all the information for mesh and vice-versa. To mitigate this problem, we highly optimized the sizeof(GeometricObject) and brought it down from it's original peak of 340 bytes to 128 bytes. It was essential that we only deal with one class of objects so that it would make our lives easier when it came to traversing the scene-graph and with containers. It made coding that more bit easier, but performance (memory usage/bandwidth) that much worse. Later we discuss future optimizations that can be made to this approach.

The GPU kernel is quite straight-forward. A thread is launched for each pixel of the output. We also support anti-aliasing, so a thread actually samples multiple pixels for the final color output. To allow for such sampling, we just loop over the required sample size in the kernel. This allows for better thread coherency, though another approach would have been to share the outputs of the neighboring sampled pixels between the threads of a block, though it is quite non-trivial to do so (we propose a better solution later). Next, the block size is 256 threads. We originally were working with 128 due to extensive register usage. But after much optimizations, we were able to double the number of threads per block. This yielded some performance gains. Note that, a block size of 256 is too large for most devices and really only works on Resonance. On our

development box, we reduce it back to 128. Obviously the number of blocks in the grid is proportional to the output width and height.

Originally we were utilizing much shared memory to store such things as pixel data, rays, and shading information. Each thread would have it's own store in the shared array. After many iterations, we found the best balance of shared memory usage, register count, and SM occupancy. The final implementation only allocates shared data for the rays. This yields a higher SM occupancy and drives up the performance of the kernel.

The ray-tracing kernel reads from global memory. We have a World object that encapsulates all the necessary scene graph information. While texture memory may have been a better choice for performance, it would have required a substantially different software design e.g. Structure of Array (SoA) approach. It would also be less flexible since it would limit the size of our scene graph too. Using texture would also make the code for indexing into the scene graph containers for traversal purposes different on CPU and GPU. We would have to introduce a "isGPU/isCPU" concept making our code get real ugly, divergent, and redundant really fast. Though we have heard that in the 3.0 SDK there are preprocessor defines to determine if the current execution path is on the CPU or GPU. This could prove quite useful in the future.

The next concern is what exactly the kernel is reading from memory. To that end, we need to focus in on the data structures employed in our application. We will first discuss the brute, non-grid, approach, since the grid approach stems from it. Our scene graph is composed of multiple light sources, multiple primitives, many materials, and a single camera. All of these are encapsulated within the World object. Other than the camera, the rest are actually container of objects e.g. lists/vectors. We had to allow for an un-

known # of such elements at runtime. So we used STL vectors as appropriate. Though the instant side-effect was that such containers can't co-exist with CUDA on the device. Moreover, one can't even declare a static instance in a class member variable. So to that extent, what we did was declare all such STL vectors as globals existing outside of the class they help to form. This wasn't such a bad thing since really there is only ever one list of objects, one list of materials, etc. Though for meshes, we do want to support many list lists of vertices & normals. Our system doesn't currently allow for that, though one that does can be easily developed in the future. Anyways, declaring the STL vectors outside of the scope of the class is fine, but we still needed a way to store the objects **in** the class so that they can later be processed during traversal. To that extent, we simply created a dynamic array of objects whose size can be determined at runtime from the given equivalent STL vector input. For example, vector<GeometricObject*> objects would become GeometricObject *cached_objects as a member variable in the World class. The actual ray tracing code touches cached_objects, and the scene-graphic construction code only deals with the host side STL vector.

Note that both CPU and GPU code touches the cached_objects. Next order of business was how to prepare that array when the transition from CPU to GPU execution occurs. This proved more difficult than first meets the eye. Basically the goal was to make a particular cached array point to device memory. Please refer to World::packGPU and Grid::packGPU to see how we do this in code. The basic idea is to make the member variable array pointer which is pointing to a host array, direct it to memory allocated on the device. At this point, the actual World object has some member data pointing to device and some to host. Continue this processing of 'packing the

GPU' until all such pointers are looking at the device. Then finally, in World::packGPU line 196, we copy over the data from "this" host World to the device World. This copies all the automatic (non-pointer) member variables and also the pointers to device memory over to the device World. At that point the device World object contains all the relevant information to be processed by the kernel.

Grid creation is done on the CPU right before the world is about to be rendered. The algorithm borrows heavily from RTGU with adjustments made to the container objects. Basically we needed to allow for the above such caching of object containers. The difficulty with grid and cells is that now we have array of arrays of GeometricObjects. This just makes the programming a little more cumbersome, but not impossible. The world packs the grid, the grid packs the cells, and the cells pack the list of cells, etc. At the end, we form one solid World object that encapsulates everything.

Luckily, regular grid traversal is a very iterative solution and doesn't call for recursion as some of the other acceleration schemes do. Borrowing from the learned concepts mentioned earlier, we shared the traversal code between the CPU & GPU. There really would have been no further optimization that could be made for the GPU since the core of the traversal code contains an infinite loop! The coolest thing about regular grids is that off the bat, if the ray misses the grid's bounding box there are no further intersections undertaken. Obviously this is a big advantage for the GPU code cause it avoids further branchy code. The next big win is that the ray is only intersected with cells that it goes "through". This saves a lot of time especially when there are large number of objects in the grid. Hence the rays will only be tested with a fraction of the objects, something that is demonstrated in the performance section. Finally, the cells are only tested

against in increasing order of the ray parameter “t”. Meaning, the first object that the ray intersects with is the closest one and no further investigation is required.

Memory usage for the grid traversal was not done in the most efficient manner. Instead of storing pointers to actual GeometricObjects in the cells, we instead stored an instance of the object (copy) itself. This proved to be a bad design decision in the end since it increased the memory bandwidth requirements when transferring to the GPU and also probably disallowed us from using pinned memory since we were asking for too much. But this was a decision made in the beginning when the idea of how to transfer scene graph data to device wasn’t quite clear. In hindsight, we could have easily declared GeometricObject **cached_objects. The cells are should contain pointers to the objects not the objects themselves as that results in substantial duplication. More than anything else in our implementation, this was the worst design decision. Luckily for us, our performance gains still outshine in majority of our build scenes even with this bad choice.

I just mentioned a bad design decision. Let me mention a good one now. The vertices and normals for the mesh are all encapsulated in the Mesh object. The Mesh object itself is not renderable. However, the SmoothMeshTraingle object which is a type of GeometricObject contains a pointer to the (one) Mesh object and 3 indices into the Mesh’s vertex and normal containers. That’s a total of 4+12=16 bytes of data per triangle. We could have instead just stored the vertices and normals right in the SmoothMeshTriangle, but that would have resulted in lots of duplicated data since adjacent triangles share the same information. Hence this resulted in less memory being used overall.

The only trick then was how to assign the mesh pointer in an instance of SmoothMeshTriangle when we are packing for the GPU. The task for this is done by a separate kernel called arrange_smt_pointers whose only job is to traverse the grid and reassign the necessary mesh pointers. The author spent some time contemplating on the necessity for launching such an auxiliary kernel. We couldn't figure out any way to dereference a device pointer on the host. Obviously that shouldn't work. We tried cudaMemcpyToSymbol, but that didn't do the job either. Then thinking outside the box, we came up with the idea of just launching a pre-kernel that does some last-minute setup work before the actual ray-tracing kernel is called. Again, lucky for us that this didn't really affect the overall execution time, since we launch 512 threads and as many blocks as necessary. It's a highly parallel kernel with proper memory coalescing (each thread indexes a separate cell).

Shadows were implemented by separating the calls to shading and intersection into separate methods to avoid any unnecessary recursion. Reflections could have been achieved in the same manner, but was omitted due to other priorities.

Another thing I do right is not store copies of the Materials inside of the GeometricObject. Most of the times, materials are shared amongst objects in a scene in your average 3D package. In the case of our PLY mesh, we only allow for one material for the whole mesh. If we stored the material information in a SmoothMeshTriangle object itself, our memory usage would skyrocket, and all the triangles would have the exact copy of the Material. Instead we store indices to an array of materials. This would be the similar approach I would have taken with the grid cells in the future.

Application Usage

It is important to run the application from the root directory where the executable is placed in order for the software to find the PLY models. There are 2 Makefiles supported, one for Mac, and the default for Resonance. Here are some useful commands:

1) make all # cleans and makes the project, only really need to do this once. It deletes *bmp

2) Then there are a bunch of useful make commands that easily run the application with the right command-line parameters for the convenience of the user e.g. make build03-g1, will run build03 (scene) with grid acceleration on.

On top of the 9 provided build files, I have allowed for much customization for the user from the command-line. They can control such things as resolution, whether or not to use the grid acceleration, how many samples to take for anti-aliasing, etc. Here is a print out of the usage menu (e.g. gpuraytracer -help)

```
crystal:gpuraytracer k5patel$ ./gpuraytracer -help
USAGE:  gpuraytracer {-b=build01} {-o=build01.bmp} {-g=1} {-w=400} {-h=400} {-n=1} {-z=1.0} {-m=0} {-s=1}
{-p=1} {-help}
All arugments are optional. By default, build01 test scene is executed.
The defaults are determined by the build scene.
-b  The build files supported by the sytem and overridden by the ones below.
-o: output (GPU) image saved to disk.  By default the naming scheme:
"build01_g-1_p-0_wh-600x600_n-1_z-1.0_cpu-64.74_gpumem-1.30_gpu-0.01_gputotal-1.31_error-0.000050.bm
p"
    which represents the build name, if grid was used, the resolution,
    number of samples, zoom, cpu time, gpu memory time, gpu computation time, gpu total time, error.
    NOTE: Your directory may become full of BMPs :)
-g: (0,1) representing whether or not grid acceleration is used. 1 by default.
```

-w: width of final image (max=16,384).

-h: height of final image (max=16,384).

-n: number of samples used per pixel for anti-aliasing (integer, max=256).

-z: camera zoom factor (float).

-m: (0,1,2) 0 to render on both GPU & CPU. 1 to render on GPU only. 2 to render on CPU only. 1 & 2 imply no error checking. 0 by default.

-s: (0,1) 0 to not save to file, 1 to save to file. Note, that either way, we do create a GPU/CPU buffers to store pixel data so as to be a fair test. 1 by default.

-p: (0,1) 0 to not use host pinned memory, 1 to use pinned memory. Default 0. Since lots of our scenes have a lot of mesh data, resonance really chokes on asking for too much pinned memory.

-help: this menu.

Example: ./gpuraytracer -b=build01 -g=1 -s=0 -m=1

would result in:

-bash-3.2$ ./gpuraytracer -b=build01 -g=1 -s=0 -m=1

*** Configuration - b: build01, o: auto-gen, g: 1, w: 4000, h: 4000, n: 16, z: 13.00, m: 1, s: 0, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

Not going to use host pinned memory...

Using Grid acceleration...

Organizing grid cells...

Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

num_cells = 45

numZeroes = 0, numOnes = 3, numTwos = 5

numThrees = 7 numGreater = 30

Finished grid cell organization.

Finished Consolidation.

World Stats:

Total # of GeometricObjects: 307

```
  Approximate bytes of data: 49624 bytes
Starting GPU ray tracing...
  # of threads in a block: 16 x 8 (128)
  # of blocks in a grid  : 250 x 500 (125000)
  GPU memory access time: 143.837997 (ms)
  GPU computation time  : 5233.520996 (ms)
  GPU processing time   : 5377.358887 (ms)
Finished ray-tracing...
```

This means we requested build01, with an automatic filename output (BMP), with grid, 4000x4000 and 16 samples per pixel, zoom of 16, mode of 1 (meaning only render on GPU not CPU), s=0 don't save the file output since we don't want to generate lots of unnecessary BMP and are just interested in the runtime, and finally with no pinned memory being used.

So you should play with m=1, different resolutions, zoom level, and sample size to try out variety of tests. Also running one of the mesh builds with p=1 on Resonance would show you that it performs worse than p=0.

Insights and Extensions

I learned a lot about how to adjust your thought process when solving a problem in a new infrastructure. CUDA 2.3 limits lots of your favorite C++ techniques, and one has to program around it. This was by far my most challenging obstacle. I invented not so fancy ideas just to get around these issues. That being said, it was more just learning about the nuts and bolts of the SDK than the SDK being that bad. I'm really hoping that the 3.0 version of CUDA performs better from a programmer's productivity standpoint.

At the start of this course I was always bothered when I heard people complain about the lack of double-precision support. I wondered to myself why they needed it so badly, and couldn't come up with a scenario where I would really want to use it. I started off doing this project with such ignorance and was quickly overcome by the lack of double-precision support! My sphere intersection code wasn't working as well as it should. Epsilon checks could have been made smaller if better precision was allowed. And my bounding box could have been made smaller if I could have used double-precision. This in turn would have reduced my memory usage since I wouldn't have more objects unnecessarily crossing over to cells they don't belong in since their bounding box doesn't fully represent the object's true geometry. I will never again make fun of people wanting double-precision!

I've already outlined my thoughts on future improvements and extensions that I can perform. At the top of the list, and I can't re-iterate this enough, would be to store pointers to GeometricObjects in my grid. Next, instead of this whole mess of STL vectors and cached_objects, I would now truly appreciate the need for the Thrust library. I did look into this at the start of the project, but I am glad that I didn't utilize it for my project.

I really learned about the essentials of GPU data structures, and can now understand why the Thrust library could be so useful. That being said, I did look over the source code for Thrust and it was the source of inspiration for the idea of the one Compilation Unit structure of my code. You will notice that there is only 1 .cu file in my project and the rest are just .inl files which are included and assembled in main.cu. This was a real pain in the beginning because I didn't really appreciate the true nature of what .cu meant. It means compilation unit (not CUDA!), and that all the code required for that unit to work has to be inside of it. Moreover, you can't have the Sphere class included in two .cu files or your program would not link and would get duplicate symbol errors. Issues like these were very annoying, but I learned from just studying the Thrust code-base.

Another neat idea I have is to remove the double for loop for sampling from the kernel and basically launch just that many more threads from the beginning. The dimensions of the grid would scale accordingly to the output dimensions by a factor of the sample size. After the ray tracing is done, a separate kernel would be launched to sum up the samples for the final image. The 2nd kernel can just use the output buffer from the ray-tracer and the host doesn't have to recopy it for the execution of the 2nd kernel. Since each thread would represent similar ray paths, I believe this would even result in better thread convergence.

The grid has a lot of empty cells. I would look into creating some sort of hash data structure that would remove these redundant cells and still allow for direct indexing. Obviously, this would further minimize the amount of memory required for grid management.

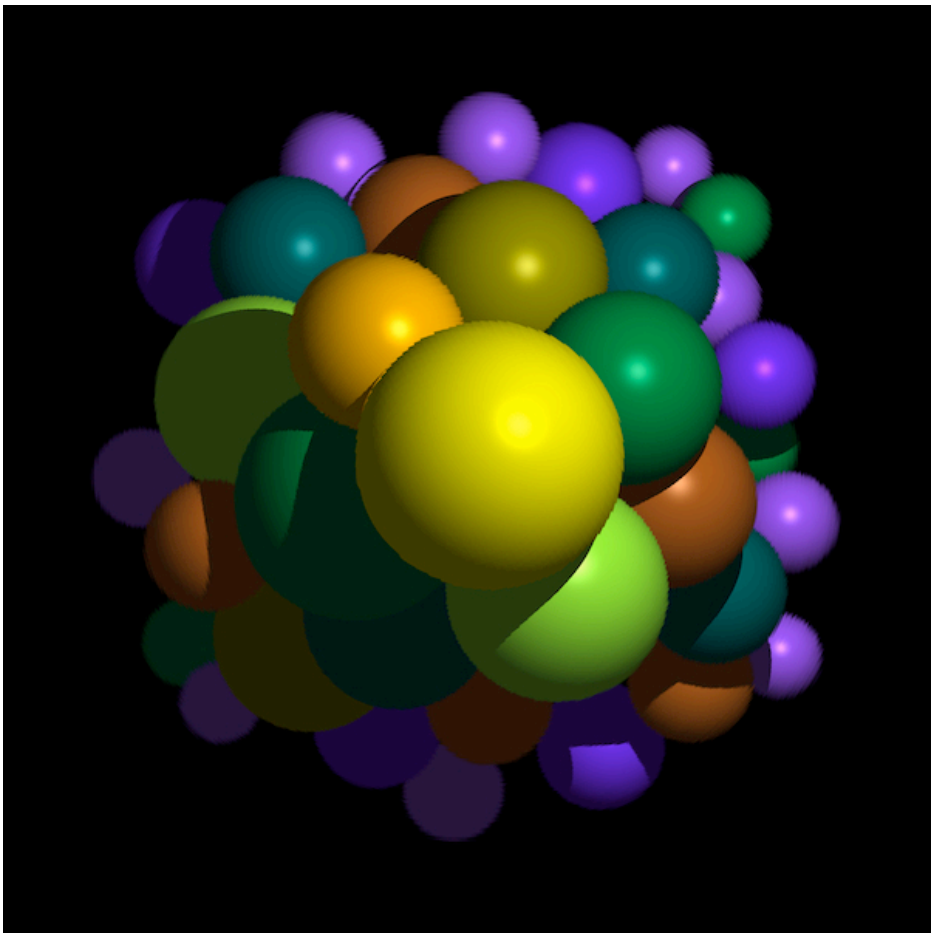
My final change would be to remove the encapsulation of different primitive type into one array, and separate them all. This would solidify the notion of Structure of Arrays that is propagated for best usage of CUDA. Again, Thrust would have proved real useful here.

I throughly enjoyed the memory requirements of the application. Though I didn't get it right the first time, I really understand now why so much stress is put on it. I found myself fighting a lot with the SDK, but NVIDIA has done phenomenal job developing the whole CUDA architecture. I have not looked into the 3.0 SDK yet, but I can only hope that they have better support for C++ and double-precision. Debugging the application was hard obviously. Emulation mode, printfs, and the -keep option proved essential here.

Performance Analysis

We now outline the performance of our application by demonstrating the different build scenes. We differentiate between grid and non-grid execution. In some of our test cases, we don't even show the non-grid execution since the time was unbounded. We also differentiate between GPU and CPU execution. You can see the produced images and their relevant log information that follows it. I've also included my own notes for your reference.

Build01



./gpuraytracer -b=build01 -g=0

*** Configuration - b: build01, o: auto-gen, g: 0, w: 4000, h: 4000, n: 16, z: 13.00, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

  Not going to use host pinned memory...

  Not using Grid accelration...

Finished Consolidation.

World Stats:

  Total # of GeometricObjects: 35

  Approximate bytes of data: 6104 bytes

Starting CPU ray tracing...

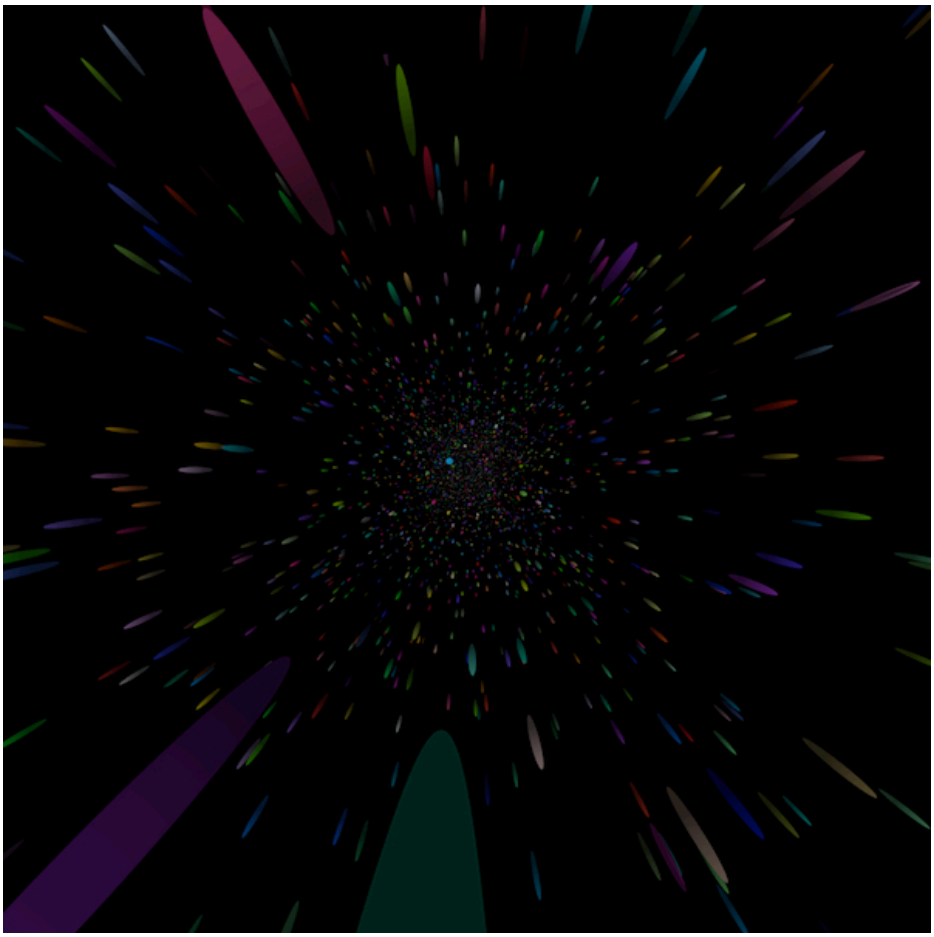
```
  CPU processing time   : 183745.546875 (ms)
Starting GPU ray tracing...
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 250 x 250 (62500)
  GPU memory access time: 139.876999 (ms)
  GPU computation time  : 3655.775879 (ms)
  GPU processing time   : 3795.652832 (ms)
Calculating accuracy...
  Error : 2.944392
Finished ray-tracing...


./gpuraytracer -b=build01 -g=1
*** Configuration - b: build01, o: auto-gen, g: 1, w: 4000, h: 4000, n: 16, z: 13.00, m: 0, s: 1, p: 0 ****
Using device 0: Tesla C1060
Starting ray-tracing...
Reorganizing Data Structures...
  Not going to use host pinned memory...
  Using Grid acceleration...
    Organizing grid cells...
      Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...
      num_cells = 45
      numZeroes = 0, numOnes = 3, numTwos = 5
      numThrees = 7 numGreater = 30
    Finished grid cell organization.
Finished Consolidation.
World Stats:
  Total # of GeometricObjects: 307
  Approximate bytes of data: 49624 bytes
Starting CPU ray tracing...
```

```
  CPU processing time   : 161656.328125 (ms)
Starting GPU ray tracing...
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 250 x 250 (62500)
  GPU memory access time: 143.811005 (ms)
  GPU computation time  : 4702.103027 (ms)
  GPU processing time   : 4845.914062 (ms)
Calculating accuracy...
  Error : 2.944392
Finished ray-tracing...
```

Notes: The high error can be attributed to single floating point imprecisions. I mentioned already that I had issues with tangent rays and hence the fuzzy rims of the spheres. It should be noted that both the CPU and GPU implementations are exhibiting this behavior. This is a very simple scene with just 35 spheres. Just outlines that our algorithm works. In both traversal schemes, the GPU beats the CPU by more than a factor of 50x. Though you will notice that the non-grid version slightly outperformed the grid version. This is mostly due to the fact that the overhead of traversing the grid with such low amount of model outweighs any performance gains.

GPU out performs CPU by 50x

Build02



./gpuraytracer -b=build02 -g=1

*** Configuration - b: build02, o: auto-gen, g: 1, w: 3000, h: 3000, n: 32, z: 1.00, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

Not going to use host pinned memory...

Using Grid acceleration...

Organizing grid cells...

Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

num_cells = 5832

numZeroes = 302, numOnes = 759, numTwos = 1115

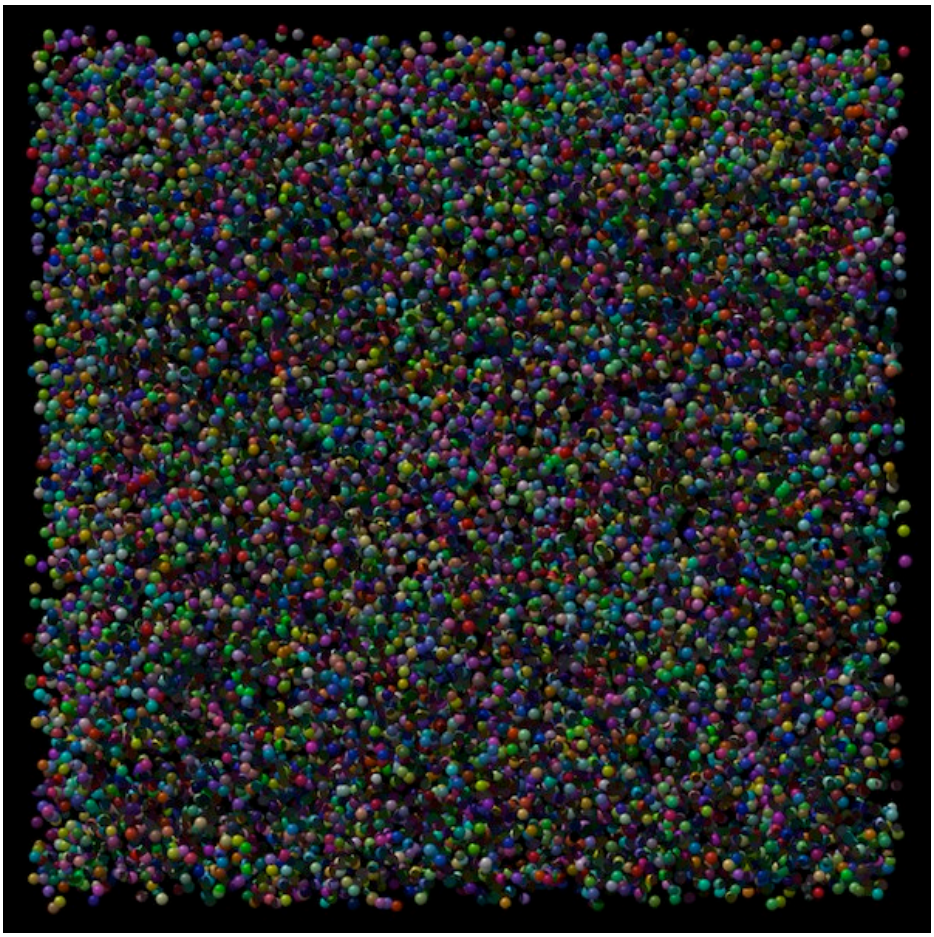
numThrees = 1232 numGreater = 2424

```
    Finished grid cell organization.
  Finished Consolidation.
  World Stats:
   Total # of GeometricObjects: 24379
   Approximate bytes of data: 3901144 bytes
  Starting CPU ray tracing...
   CPU processing time   : 197754.734375 (ms)
  Starting GPU ray tracing...
   # of threads in a block: 16 x 16 (256)
   # of blocks in a grid  : 188 x 188 (35344)
   GPU memory access time: 318.101990 (ms)
   GPU computation time  : 7326.841797 (ms)
   GPU processing time   : 7644.943848 (ms)
  Calculating accuracy...
   Error : 0.006260
  Finished ray-tracing...
```

Notes: This scene demonstrates close to 25K spheres in a scene. The non-grid version was unbounded in execution time, so is not listed here. The GPU outperforms the CPU by a factor of 25x. If we stored the spheres in a separate data structure as mentioned above, then the memory requirements would have been much less. Also, again if we stored pointers to actual spheres in the cells instead of duplicate objects, we could further bring down the GPU memory access time.

GPU out performs CPU by 25x

Build03



./gpuraytracer -b=build03

*** Configuration - b: build03, o: auto-gen, g: 1, w: 1000, h: 1000, n: 32, z: 2.50, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

Not going to use host pinned memory...

Using Grid acceleration...

Organizing grid cells...

Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

num_cells = 21952

numZeroes = 58, numOnes = 258, numTwos = 506

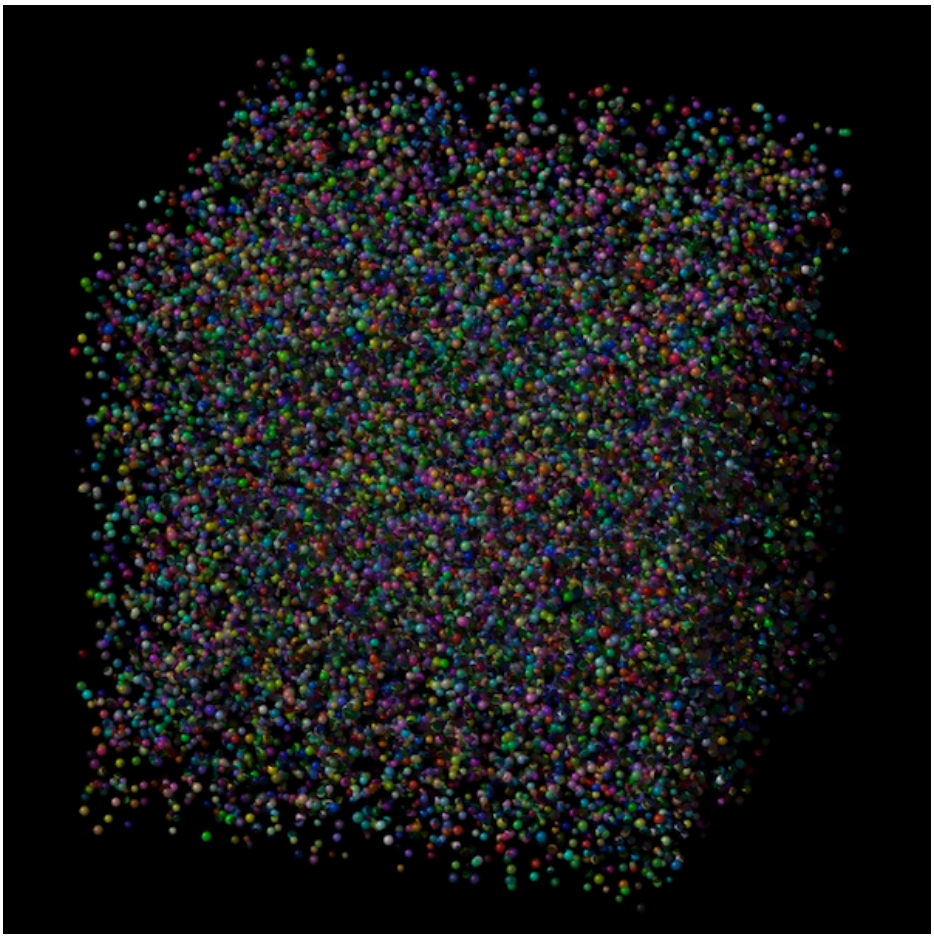
numThrees = 874 numGreater = 20256

```
    Finished grid cell organization.
  Finished Consolidation.
  World Stats:
   Total # of GeometricObjects: 216069
   Approximate bytes of data: 34571544 bytes
  Starting CPU ray tracing...
   CPU processing time   : 59133.652344 (ms)
  Starting GPU ray tracing...
   # of threads in a block: 16 x 16 (256)
   # of blocks in a grid  : 63 x 63 (3969)
   GPU memory access time: 6335.658203 (ms)
   GPU computation time  : 9074.992188 (ms)
   GPU processing time   : 15410.650391 (ms)
  Calculating accuracy...
   Error : 10.157168
  Finished ray-tracing...
```

Notes: Again, non-grid time is not demonstrated. Here we are instantiating over 200K spheres. The performance gains here is not great since we are succumbed to the memory transfer of 34MB of data. The GPU memory time itself is almost half the total processing time. Again better usage of data structures would have greatly decreased this. If we look at only the GPU computation time, then we have gains of 7x improvement. Though since we must take the total time into consideration, we have something closer to only 4x improvement.

GPU out performs CPU by 4x

Build04



./gpuraytracer -b=build04

*** Configuration - b: build04, o: auto-gen, g: 1, w: 2400, h: 2400, n: 32, z: 6.00, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

  Not going to use host pinned memory...

  Using Grid acceleration...

    Organizing grid cells...

      Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

      num_cells = 21952

      numZeroes = 58, numOnes = 258, numTwos = 506

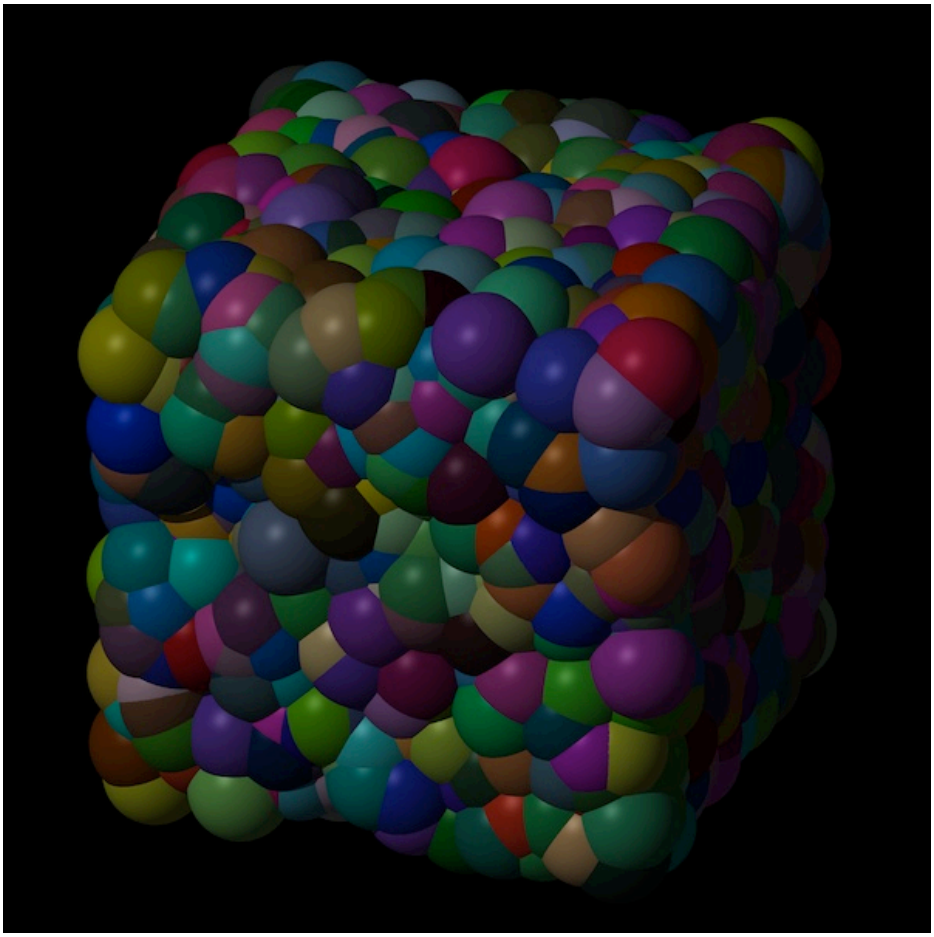
      numThrees = 874 numGreater = 20256

```
    Finished grid cell organization.
  Finished Consolidation.
  World Stats:
   Total # of GeometricObjects: 216069
   Approximate bytes of data: 34571544 bytes
  Starting CPU ray tracing...
   CPU processing time   : 266908.968750 (ms)
  Starting GPU ray tracing...
   # of threads in a block: 16 x 16 (256)
   # of blocks in a grid  : 150 x 150 (22500)
   GPU memory access time: 6394.527832 (ms)
   GPU computation time  : 39441.500000 (ms)
   GPU processing time   : 45836.027344 (ms)
  Calculating accuracy...
   Error : 17.188444
  Finished ray-tracing...
```

Notes: This is a special grid setup, exploiting the advantages of such an acceleration scheme. Since the close to 200K objects are positioned in an almost cube like manner, rays can be opted out much earlier without getting too deep into the cells. Hence we see big gains in our GPU version even though we are still plagued by the nasty memory access time. The GPU outperforms the CPU by a factor of 6x.

GPU out performs CPU by 6x

Build05



-bash-3.2$ ./gpuraytracer -b=build05 -g=0 -m=1

*** Configuration - b: build05, o: auto-gen, g: 0, w: 2400, h: 2400, n: 8, z: 6.00, m: 1, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

  Not going to use host pinned memory...

  Not using Grid accelration...

Finished Consolidation.

World Stats:

  Total # of GeometricObjects: 3000

  Approximate bytes of data: 480504 bytes

Starting GPU ray tracing...

```
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 150 x 150 (22500)
  GPU memory access time: 93.487000 (ms)
  GPU computation time  : 24981.437500 (ms)
  GPU processing time   : 25074.923828 (ms)
Finished ray-tracing...
./gpuraytracer -b=build05 -g=1
*** Configuration - b: build05, o: auto-gen, g: 1, w: 2400, h: 2400, n: 8, z: 6.00, m: 0, s: 1, p: 0 ****
Using device 0: Tesla C1060
Starting ray-tracing...
Reorganizing Data Structures...
  Not going to use host pinned memory...
  Using Grid acceleration...
    Organizing grid cells...
      Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...
      num_cells = 3375
      numZeroes = 0, numOnes = 3, numTwos = 3
      numThrees = 8 numGreater = 3361
    Finished grid cell organization.
Finished Consolidation.
World Stats:
  Total # of GeometricObjects: 440481
  Approximate bytes of data: 70477464 bytes
Starting CPU ray tracing...
  CPU processing time   : 141253.453125 (ms)
Starting GPU ray tracing...
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 150 x 150 (22500)
  GPU memory access time: 553.368958 (ms)
```

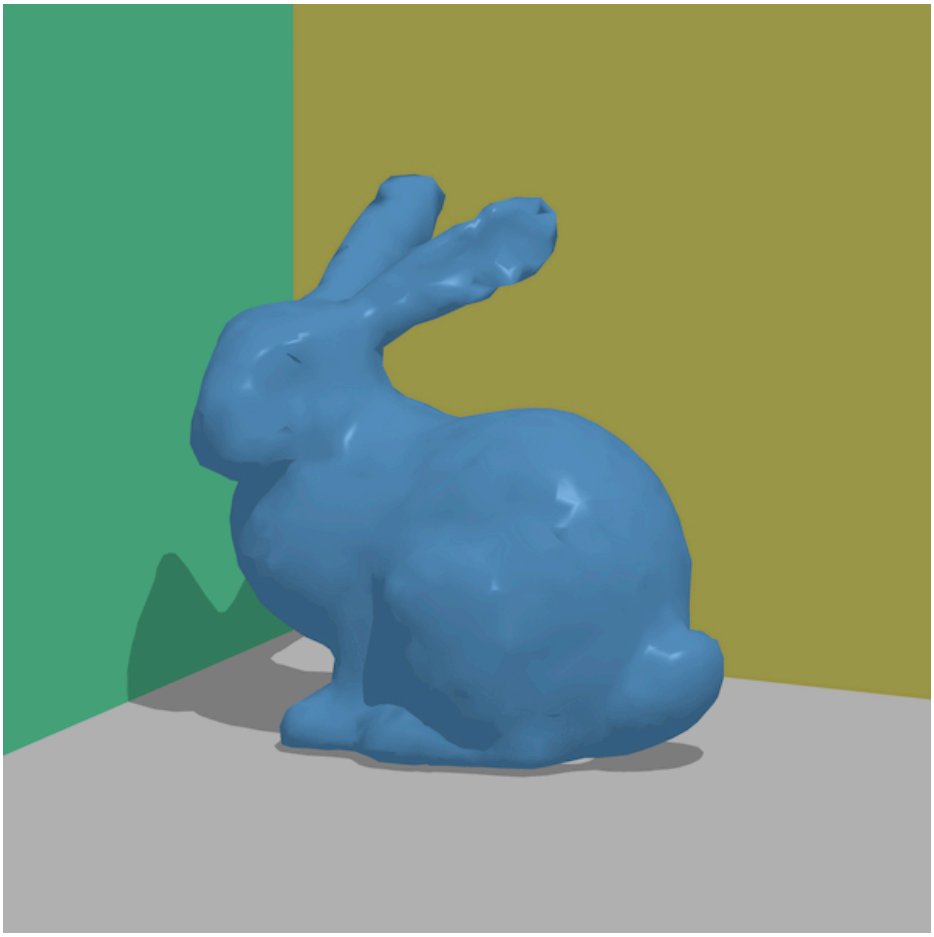
```
GPU computation time  : 7443.520996 (ms)
GPU processing time   : 7996.890137 (ms)
Calculating accuracy...
Error : 0.266910
Finished ray-tracing...
```

Note: Another special grid setup, but here highlighting the fact that any given cell has a larger list of contained objects since the spheres themselves would crossover many more cell boundaries. This results in the CPU code having to traverse much more and hence it's longer execution time. In the grid version, the performance gains of using the GPU are 18x improvement. We also provide the timings for the non-grid execution on the GPU. The CPU non-grid version was not listed because it took unbounded time. Hence the potential gains are undetermined, though we are assured that it's more than 10x. Also, if you compare the non-grid GPU and grid GPU times, you will notice though while the grid was still faster, just not a whole lot. This sort of setup is bad for the grid, and basically requires finer-grained cells, which we don't employ because of memory constraints.

GPU out performs CPU by more than 10x in non-grid

GPU out performs CPU by 18x in grid

Build06



-bash-3.2$ make build06-g0

./gpuraytracer -b=build06 -g=0 -m=1

Parsing PLY model: models/Bunny4K.ply

*** Configuration - b: build06, o: auto-gen, g: 0, w: 2400, h: 2400, n: 4, z: 6.00, m: 1, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

  Not going to use host pinned memory...

  Not using Grid accelration...

Finished Consolidation.

World Stats:

  Total # of GeometricObjects: 3854

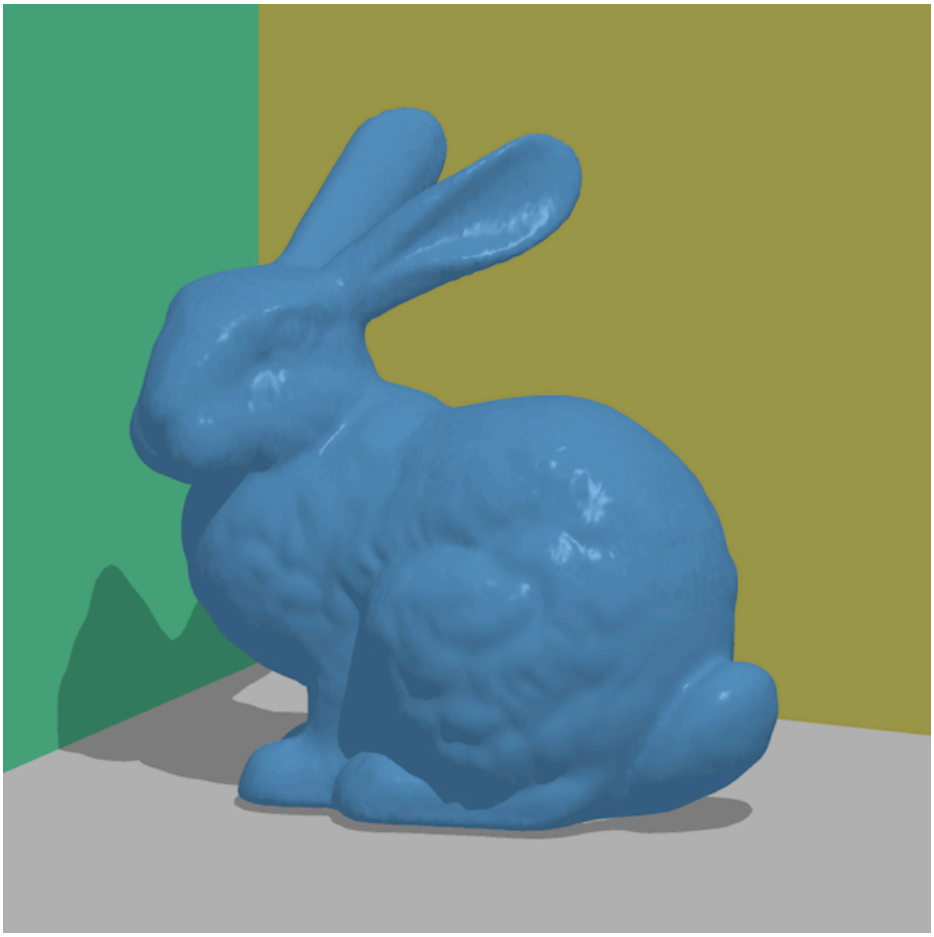
```
  Approximate bytes of data: 662480 bytes
  # of triangles in mesh: 3851
Starting GPU ray tracing...
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 150 x 150 (22500)
  GPU memory access time: 107.730003 (ms)
  GPU computation time  : 313763.375000 (ms)
  GPU processing time   : 313871.093750 (ms)
Finished ray-tracing...
./gpuraytracer -b=build06 -g=1 -m=1
Parsing PLY model: models/Bunny4K.ply
*** Configuration - b: build06, o: auto-gen, g: 1, w: 2400, h: 2400, n: 4, z: 6.00, m: 1, s: 1, p: 0 ****
Using device 0: Tesla C1060
Starting ray-tracing...
Reorganizing Data Structures...
  Not going to use host pinned memory...
  Using Grid acceleration...
    Organizing grid cells...
      Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...
      num_cells = 4335
      numZeroes = 75, numOnes = 660, numTwos = 1872
      numThrees = 1714 numGreater = 14
    Finished grid cell organization.
Finished Consolidation.
World Stats:
  Total # of GeometricObjects: 18205
  Approximate bytes of data: 2958640 bytes
  # of triangles in mesh: 3851
Starting GPU ray tracing...
```

```
# of threads in a block: 16 x 16 (256)
# of blocks in a grid  : 150 x 150 (22500)
GPU memory access time: 254.091995 (ms)
GPU computation time  : 60455.207031 (ms)
GPU processing time   : 60709.300781 (ms)
Finished ray-tracing...
```

Notes: This is the 4K version of the Stanford Bunny. You can tell that the specular highlights reveal the low-poly count of the model itself. CPU times for both schemes are not available due to the longevity of their execution times. The grid GPU outperforms the non-grid GPU by a factor of 5x.

GPU easily outperforms by over a factor of 30x.

Build07



-bash-3.2$ ./gpuraytracer -b=build07 -g=0 -m=1

Parsing PLY model: models/Bunny69K.ply

*** Configuration - b: build07, o: auto-gen, g: 0, w: 1000, h: 1000, n: 4, z: 3.00, m: 1, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

  Not going to use host pinned memory...

  Not using Grid accelration...

Finished Consolidation.

World Stats:

  Total # of GeometricObjects: 69454

  Approximate bytes of data: 11975872 bytes

```
  # of triangles in mesh: 69451
 Starting GPU ray tracing...
  # of threads in a block: 16 x 8 (128)
  # of blocks in a grid  : 63 x 125 (7875)


  GPU memory access time: 151.362000 (ms)
  GPU computation time  : 1030644.875000 (ms)
  GPU processing time   : 1030796.250000 (ms)
 Imaged saved:
build07_g-0_p-0_wh-1000x1000_n-4_z-3.0_gpumem-151.36_gpu-1030644.88_gputotal-1030796.25.bmp
 Finished ray-tracing...
 ./gpuraytracer -b=build07 -g=1
 Parsing PLY model: models/Bunny69K.ply
 *** Configuration - b: build07, o: auto-gen, g: 1, w: 1000, h: 1000, n: 4, z: 3.00, m: 0, s: 1, p: 0 ****
 Using device 0: Tesla C1060
 Starting ray-tracing...
 Reorganizing Data Structures...
  Not going to use host pinned memory...
  Using Grid acceleration...
   Organizing grid cells...
    Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...
    num_cells = 73568
    numZeroes = 1183, numOnes = 10881, numTwos = 31713
    numThrees = 29724 numGreater = 67
   Finished grid cell organization.
 Finished Consolidation.
 World Stats:
  Total # of GeometricObjects: 314876
  Approximate bytes of data: 51243392 bytes
```

```
  # of triangles in mesh: 69451
Starting CPU ray tracing...
  CPU processing time   : 407167.406250 (ms)
Starting GPU ray tracing...
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 63 x 63 (3969)
  GPU memory access time: 260599.312500 (ms)
  GPU computation time  : 51246.035156 (ms)
  GPU processing time   : 311845.343750 (ms)
Calculating accuracy...
  Error : 0.039692
Finished ray-tracing...
```

Notes: Now with 69K of polys. Here we see once again the memory creep of the grid version. It took more than a factor of 5 to transfer the grid information to the GPU. This is unacceptable and will be easily addressed in the future. Though the GPU grid still out performs the non-grid GPU by a factor of 3x. If we only compare the raw processing times, then the performance gains are more like 20x. Again the CPU times were not accounted for since they were unbounded.

GPU easily outperforms by over a factor of 50x.

Build08

./gpuraytracer -b=build08 -g=1

Parsing PLY model: models/Horse97K.ply

*** Configuration - b: build08, o: auto-gen, g: 1, w: 3000, h: 3000, n: 8, z: 5.00, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

Not going to use host pinned memory...

Using Grid acceleration...

Organizing grid cells...

Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

num_cells = 101760

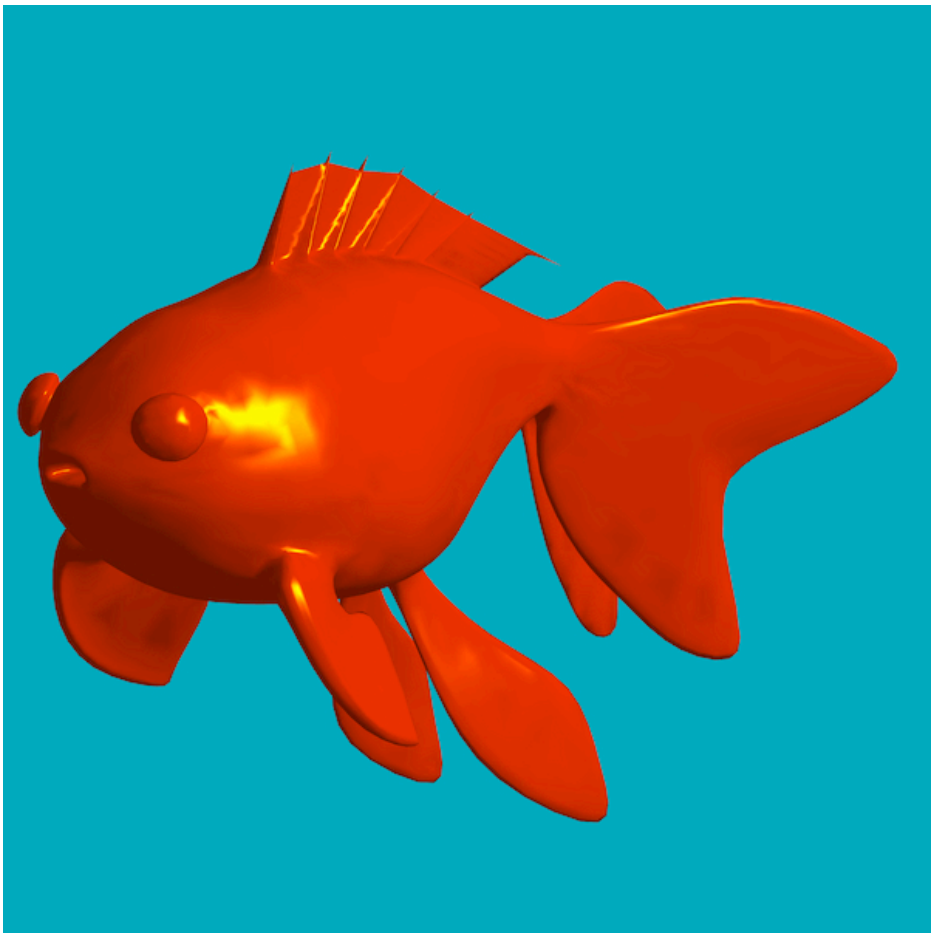
numZeroes = 93654, numOnes = 239, numTwos = 156

```
        numThrees = 192 numGreater = 7519
      Finished grid cell organization.
    Finished Consolidation.
    World Stats:
     Total # of GeometricObjects: 379137
     Approximate bytes of data: 61826064 bytes
     # of triangles in mesh: 96966
    Starting CPU ray tracing...
     CPU processing time   : 76605.570312 (ms)
    Starting GPU ray tracing...
     # of threads in a block: 16 x 16 (256)
     # of blocks in a grid  : 188 x 188 (35344)
     GPU memory access time: 1392.593018 (ms)
     GPU computation time  : 13985.824219 (ms)
     GPU processing time   : 15378.416992 (ms)
    Calculating accuracy...
     Error : 0.000105
    Finished ray-tracing...
```

Notes: This is a 97K poly horse. Non-grid execution time is not provided. The GPU performs 5x better than the CPU in the grid version. Also notice that the error count is something much more acceptable with our implementation of triangle meshes, since the intersection code is trivial and more immune to floating point imprecisions.

GPU out performs CPU by 5x in grid

Build09



./gpuraytracer -b=build09 -g=1

Parsing PLY model: models/goldfish_high_res.ply

*** Configuration - b: build09, o: auto-gen, g: 1, w: 2400, h: 2400, n: 16, z: 7.00, m: 0, s: 1, p: 0 ****

Using device 0: Tesla C1060

Starting ray-tracing...

Reorganizing Data Structures...

Not going to use host pinned memory...

Using Grid acceleration...

Organizing grid cells...

Stats: # of cells that have 0 objects, 1 object, 2 objects, etc...

num_cells = 22950

numZeroes = 19629, numOnes = 142, numTwos = 154

```
      numThrees = 149 numGreater = 2876
    Finished grid cell organization.
  Finished Consolidation.
  World Stats:
    Total # of GeometricObjects: 79294
    Approximate bytes of data: 12953656 bytes
    # of triangles in mesh: 22048
  Starting CPU ray tracing...
    CPU processing time   : 114595.484375 (ms)
  Starting GPU ray tracing...
    # of threads in a block: 16 x 16 (256)
    # of blocks in a grid  : 150 x 150 (22500)
    GPU memory access time: 275.122009 (ms)
    GPU computation time  : 18418.689453 (ms)
    GPU processing time   : 18693.810547 (ms)
  Calculating accuracy...
    Error : 0.001519
  Finished ray-tracing...
```

Notes: Non-grid not provided since the times were unbounded.

GPU out performs CPU by 6x

Acknowledgements

I would whole-heartily like to thank Kevin Suffern, the author of Ray Tracing from the Ground Up. We exchanged many emails during the course of this project, and he was essential in helping me isolate imprecision issues. His book is an awesome read, and the exercise questions inspired me to make some of my most relevant design decisions. I would like to also formally acknowledge that I based a lot of my code from his sample ray tracer in his book.

Finally I would also like to thank my TF, Kevin Dale, for guiding me on the right track for this project. I originally had proposed an MPI based Ray-Tracer using no acceleration scheme. He was the one who convinced me that it would be more interesting to use some sort of acceleration scheme (to offset the branchy ray-tracing algorithm). I can now see that an MPI based solution would have been more leg-work, and wouldn't have best exemplified CUDA/GPU without a proper acceleration scheme.