report

December 21, 2019

[64]:
```
%alias_magic t timeit -p "-o -n 1 -r 1 -q"
```

```
Created `%t` as an alias for `%timeit -o -n 1 -r 1 -q`.
Created `%%t` as an alias for `%%timeit -o -n 1 -r 1 -q`.
```

[41]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import math
```

1 MI-PRC - Path tracing report

Jiří Šebele

1.1 Problem definition

Path tracing is a Monte Carlo computer graphics rendering method, which produces scenes that are very faithful to reality. In theory, it's integrating the surface radiance function over all directions for every point in the scene. In practicality, it's simulating a ray of light through the scene, collecting info about everything it hits on the way. And it does it pseudorandomly, which is where the Monte Carlo part comes from. Therefore, there are the following parameters to each scene:

- Resolution in pixels
- Samples per pixel
- Amount of geometry in the scene

We'll try playing around with each of them and see where that takes us.

1.2 Sequential implementation

In it's core, the actual algorithm solves the rendering equation, which goes roughly as follows:

$L_o(x) = L_e(x) + \int_\Omega f_r(x) L_i(x)(\omega \cdot n) d\omega$

Where:

- L_o is the total radiance observed from point x
- L_e is the emitted radiance from point x

- Ω is the set of all possible outgoing rays from x
- f_r is the bidirectional scattering distribution function (or BSDF)
- L_i is the radience incoming to x
- n is the surface normal of x

In practice, that means that for every pixel on the screen we see what point of what object we can see and solve the rendering equation for that point. Since computing it completely would take an unreasonable amount of time for even a minimal scene, we make our life easier by sampling directions from Ω according to f_r and recursively computing the values of L_i by re-applying the rendering equation. Since this could go on forever, we limit the number of bounces through the scene (e.g. the maximum recursion depth).

To see which point is observed from where, we use a technique called raycasting, e.g. sending a ray through the scene and checking, if it intersects any of our objects. In our algorithm, we only allow spheres to be present, since that makes the checking of intersections to be fairly simple while still being an interesting shape to look at.

Since the algorithm is already very parallel in nature, we just have to iterate over every pixel and solve the render equation there.

```
for (uint32_t x = 0; x < width; x++) {
    for (uint32_t y = 0; y < height; y++) {
        render_equation(...);
    }
}
```

1.3 Adapting for the GPU

After implementing the sequential part of this assignment, it's time to parallelize it and run it on the GPU. The sequential implementation is already well suited to run in parallel. The only thing we have to change to get going is to replace two for-loops in the sequential implementation…

```
for (uint32_t x = 0; x < width; x++) {
    for (uint32_t y = 0; y < height; y++) {
        render_equation(...);
    }
}
```

…with a CUDA kernel…

```
__global__
void render_kernel(...) {
    uint64_t x = blockIdx.x * blockDim.x + threadIdx.x;
    uint64_t y = blockIdx.y * blockDim.y + threadIdx.y;

    render_equation(...);
}
```

…and an appropriate kernel call.

```
dim3 block(32, 32, 1);
dim3 grid(width / block.x, height / block.y, 1);
```

```
render_kernel<<<grid, block>>>(...);
```

We use a block of size 32 by 32 pixels, so we fill all of the 1024 threads in a block. Then we compute how many blocks we need to fit into the resolution and off we go.

Next up we'll try to curb the amount of global memory accesses by moving the scene into shared memory. We'll allocate enough space for the scene and use each thread to copy one sphere. That will help with memory coalescing and speed up subsequent intersection checks.

```
extern __shared__ Sphere shared_spheres[];

if (threadIdx.x < sphere_count) {
    shared_spheres[threadIdx.x] = device_spheres[threadIdx.x];
}
__syncthreads();
```

We also need to modify the kernel call to allocate enough shared memory.

```
render_kernel<<<grid, block, spheres.size() * sizeof(Sphere)>>>(...);
```

1.4 Time measurements

Let's see how much the changes helped and do some time measurements of the performance.

1.4.1 Sequential implementation

Let's see how the sequential implementation fares with minimal parameters (128 samples per pixel with a resolution of 512 by 512 pixels).

```
$ time ./main_cpu -w 512 -h 512 -s 128 -o output.ppm
Rendering...
Rendering 511, 511
Saved image to 'output.ppm'
./main  509.50s user 6.17s system 99% cpu 8:35.91 total
```

As we can see, that's laughably slow, so we're not going to even bother pushing it further.

1.4.2 GPU implementation

Let's see how the GPU version does with the same problem.

[70]:
```
!nvprof ./main_gpu -w 512 -h 512 -s 128 -o output.ppm
```

```
==11842== NVPROF is profiling process 11842, command: ./main_gpu -w 512 -h 512
-s 128 -o output.ppm
Rendering…
Saved image to 'output.ppm'
==11842== Profiling application: ./main_gpu -w 512 -h 512 -s 128 -o output.ppm
==11842== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max
Name
```

```
 GPU activities:   98.74%  71.231ms         1  71.231ms  71.231ms  71.231ms
render_kernel(float3*, unsigned int, unsigned int, unsigned int, Sphere*,
unsigned long)
                   1.26%  911.24us         1  911.24us  911.24us  911.24us
[CUDA memcpy DtoH]
                   0.00%     736ns         1     736ns     736ns     736ns
[CUDA memcpy HtoD]
      API calls:   71.64%  187.35ms         2  93.674ms  238.74us  187.11ms
cudaMalloc
                  28.04%  73.331ms         2  36.665ms  16.221us  73.315ms
cudaMemcpy
                   0.24%  633.59us         2  316.79us  244.69us  388.89us
cudaFree
                   0.04%  99.966us        97  1.0300us     103ns  40.289us
cuDeviceGetAttribute
                   0.02%  59.755us         1  59.755us  59.755us  59.755us
cudaLaunchKernel
                   0.01%  22.096us         1  22.096us  22.096us  22.096us
cuDeviceGetName
                   0.01%  17.924us         1  17.924us  17.924us  17.924us
cuDeviceTotalMem
                   0.00%  3.8410us         3  1.2800us     262ns  3.2750us
cuDeviceGetCount
                   0.00%  2.3360us         2  1.1680us     141ns  2.1950us
cuDeviceGet
                   0.00%     255ns         1     255ns     255ns     255ns
cuDeviceGetUuid
```

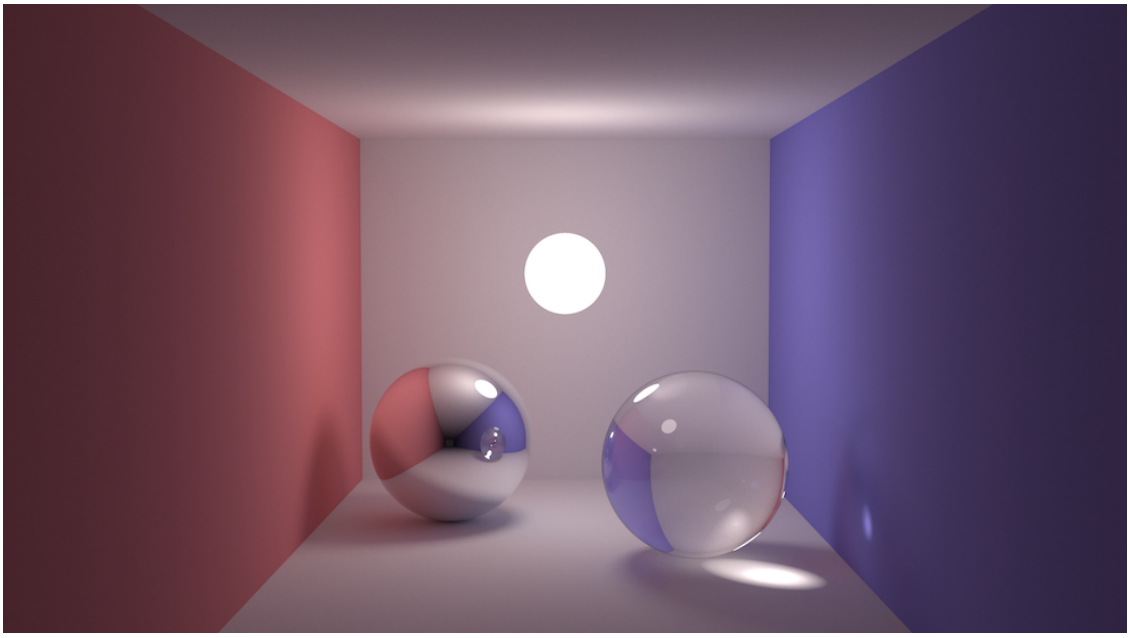
That's a *lot* faster. Now let's see how far we can push it.

[71]:
```
!nvprof ./main_gpu -w 3840 -h 2160 -s 8096 -o output.ppm
```

```
==11847== NVPROF is profiling process 11847, command: ./main_gpu -w 3840 -h 2160
-s 8096 -o output.ppm
Rendering…
Saved image to 'output.ppm'
==11847== Profiling application: ./main_gpu -w 3840 -h 2160 -s 8096 -o
output.ppm
==11847== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max
Name
 GPU activities:   99.97%  138.346s         1  138.346s  138.346s  138.346s
render_kernel(float3*, unsigned int, unsigned int, unsigned int, Sphere*,
unsigned long)
                   0.03%  43.683ms         1  43.683ms  43.683ms  43.683ms
[CUDA memcpy DtoH]
                   0.00%     800ns         1     800ns     800ns     800ns
[CUDA memcpy HtoD]
```

```
      API calls:   99.86%  138.391s         2  69.1953s  16.816us  138.391s
cudaMemcpy
                    0.14%  187.26ms         2  93.629ms  242.39us  187.01ms
cudaMalloc
                    0.00%  3.7191ms         2  1.8595ms  247.11us  3.4719ms
cudaFree
                    0.00%  100.62us        97  1.0370us     103ns  41.039us
cuDeviceGetAttribute
                    0.00%  69.221us         1  69.221us  69.221us  69.221us
cudaLaunchKernel
                    0.00%  17.859us         1  17.859us  17.859us  17.859us
cuDeviceTotalMem
                    0.00%  17.727us         1  17.727us  17.727us  17.727us
cuDeviceGetName
                    0.00%  3.4050us         3  1.1350us     262ns  2.8560us
cuDeviceGetCount
                    0.00%  2.2710us         2  1.1350us     152ns  2.1190us
cuDeviceGet
                    0.00%     278ns         1     278ns     278ns     278ns
cuDeviceGetUuid
```

Just over two minutes. That's *still* 4 times faster than the sequential implementation. Let's look at the result now.



Pretty good!

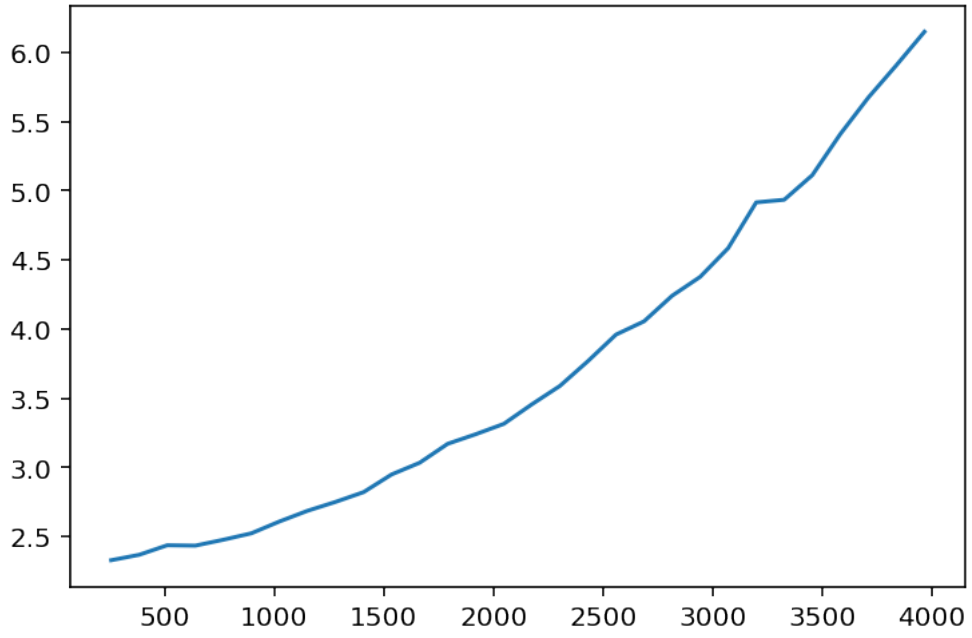
1.4.3 Parameter sensitivity

Next up let's check how our algorithm reacts to changes in the parameters.

Resolution Let's see how changing the resolution impacts the time.

[60]:
```
resolutions = []
results = []
for size in range(256, 4096, 128):
    result = %t !./main_gpu -w {size} -h {size} -s 128 > /dev/null
    results.append(result)
    resolutions.append(size)
```

[63]:
```
plt.plot(resolutions, list(map(lambda x: x.best, results)))
plt.show()
```

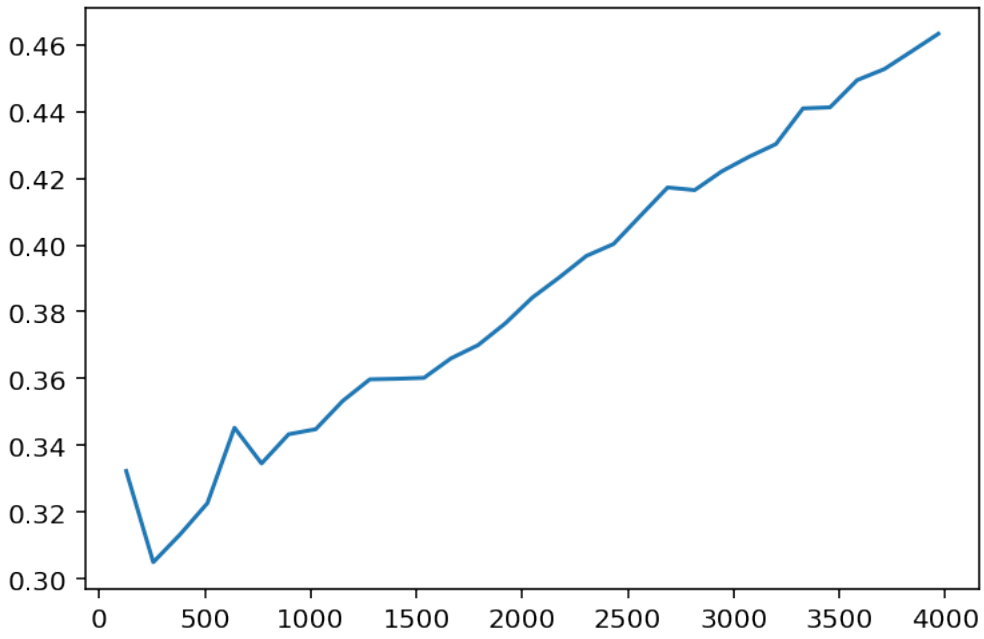


We can see a pretty predictable quadratic increase in complexity, since each step increases the total area. Nothing new under the sun there.

Samples per pixel Let's see how increasing the samples per pixel changes the complexity.

[73]:
```
spps = []
results = []
for spp in range(128, 4096, 128):
    result = %t !./main_gpu -w 128 -h 128 -s {spp} > /dev/null
    results.append(result)
    spps.append(spp)
```

[74]:
```
plt.plot(spps, list(map(lambda x: x.best, results)))
plt.show()
```

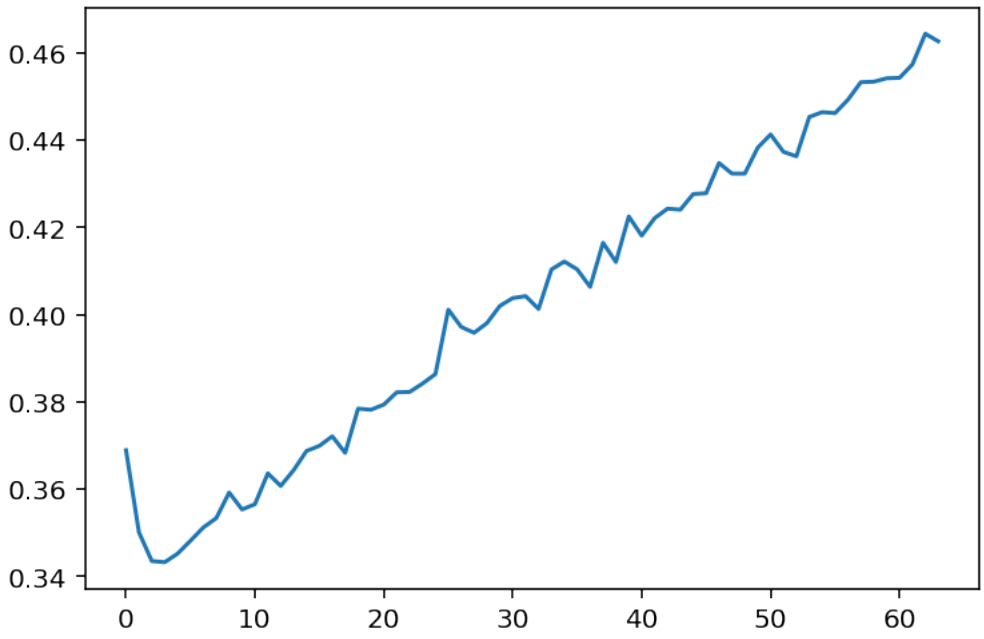


Here, the change is expectedly linear. Double the samples means double the work. So far so good.

Amount of geometry Let's see how adding additional spheres to the scene changes the complexity.

[79]:
```
counts = []
results = []
for count in range(0, 64):
    result = %t !./main_gpu -w 128 -h 128 -s 1024 -r {count} > /dev/null
    results.append(result)
    counts.append(count)
```

[80]:
```
plt.plot(counts, list(map(lambda x: x.best, results)))
plt.show()
```



Here, the change also seems linear. Which makes sense, because for each pixel the number of intersection checks is $|spheres| * bounces$.

1.5 Open Image Denoise

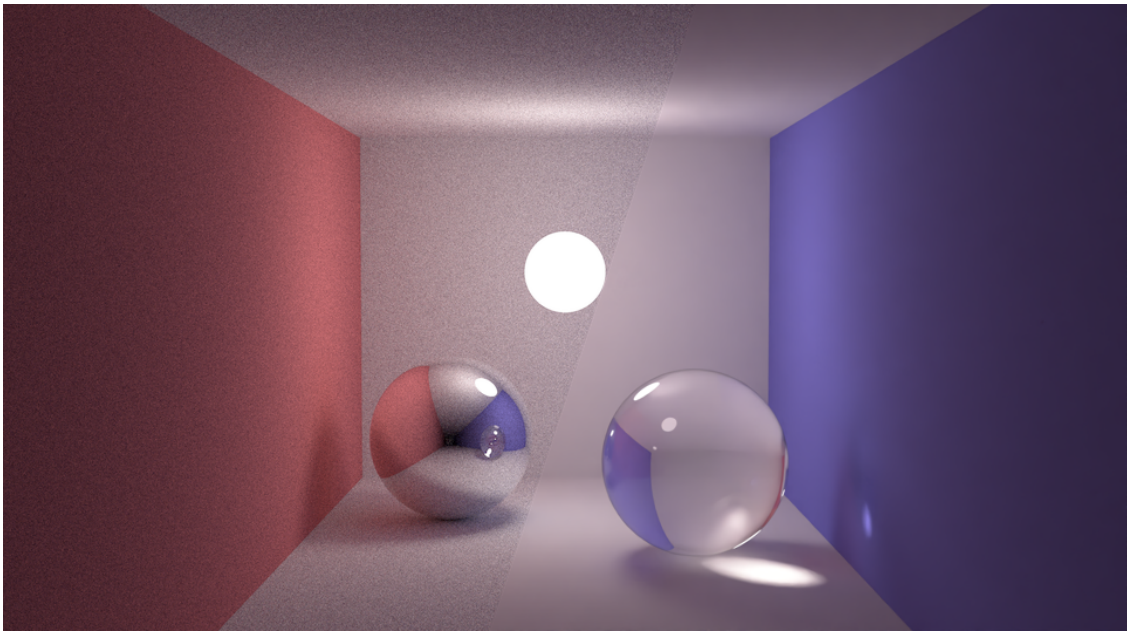
Using Intel's Open Image Denoise, we can drastically improve the quality of our image without increasing the number of samples. We just use a very noisy image as an input to a OIDN filter and then execute it.

```
oidn::DeviceRef device = oidn::newDevice();
device.commit();

oidn::FilterRef filter = device.newFilter("RT");
filter.setImage("color",  output_h,  oidn::Format::Float3, width, height);
filter.setImage("output", denoised, oidn::Format::Float3, width, height);
filter.commit();

filter.execute();
```

The result is astonishingly accurate, without almost any added computational time.



1.6 Conclusion

Since our algorithm was designed to be parallelized from the ground up, it required very little change to adapt to GPU. Running it in parallel proved to be an order of magnitude faster. This made it feasible to render images with less noise and higher resolution. Further optimizations could be made to further improve performance, such as:

- using nVidia's OptiX API to accelerate raycasting,
- adding a BVH hierarchy to decrease the number of collision checks.

For the sake of this assignment, I tried to keep it as simple as possible.