Accelerating Ray Tracing

Wei-Lun Huang
Center for Data Science
New York University
wh2103@nyu.edu

Maksat Kuanyshbay
Department of Mathematics
New York University
mk7756@nyu.edu

YiTao Long
Department of Computer Science
New York University
yl8251@nyu.edu

Abstract

Recently, Metaverse has been widely discussed, which requires a 3D rendering technique, by major tech companies such as Nvidia, Meta, and Microsoft. Ray tracing is a photorealistic 3D rendering algorithm that simulates the physics of how humans see objects through lights. However, it suffers from expensive computations and load-balancing issues and has been considered nearly impossible for decades until Nvidia introduced RTX technology for hardware acceleration. In this report, we implemented a ray tracer, its parallelizations in OpenMP and CUDA, and hardware acceleration using RTX's Optix SDK for comparison. Although parallelizing a ray-tracing algorithm is simple, it is non-trivial to push its performance to the theoretical limit. From a software aspect, we found that scheduling and floating-point precision are crucial for the speedup of a ray tracer. From a hardware aspect, our results showed that Optix outperforms other implementations with a large margin. Therefore, we concluded that an efficient ray tracing implementation should coordinate the software and hardware well.

1 Introduction

Since the outbreak of the COVID-19 pandemic, people have been more familiar with connecting online and working remotely. With this trend, major tech companies such as Nvidia, Meta, and Microsoft brought the concept of virtual reality (VR) into discussion again. Ray tracing is a natural way of generating 3D images from the coordinates of the objects in the scene for the VR world. It is natural because it imitates the way our eyes see the world. As shown in Figure 1, a ray-tracing problem usually consists of 4 components: camera or the eyes, image or pixel grid, objects in the scene, and the light source. In the physics of the real world, humans can see a thing because it can reflect portions of the light that can get through people's pupils and project a 2D image to the retina upside down. Our brains can automatically flip the image back to its normal orientation. In computer graphics, the image plane is flipped to the front of the camera. For simplicity, a ray-tracing algorithm casts rays in a reverse direction, from each pixel in the image plane of the eyes to the object and then to the light sources instead. The color of the pixel is then determined by the intensity of the light, the number of bounces, the angle between the ray, and the normal vector of the surface and color (i.e., reflection or absorption rate) of the object itself. According to where the lights come from, two lightning models have been proposed: local illumination (i.e., direct lightning model) and global illumination (i.e., indirect lightning models) [Appel, 1968, Marschner and Shirley, 2016].

We only consider global illumination for this report since it is more similar to how the real world works and more computationally challenging. In this model, each pixel samples and shoots many rays in different directions for a given maximum number of bounces (i.e., depth) and averages the colors of the objects that the beams can hit as the final pixel value. This model is also natural for explicit light sources since it considers them as ordinary objects that can emit light colors instead of reflecting other colors. Although the global illumination can generate photorealistic views, its output image can be noisy without sufficient samples. It usually requires thousands of samples, which leads to a long-running time.

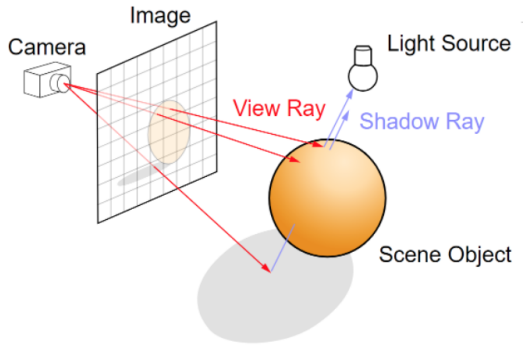

Figure 1: Illustration of a ray tracing problem [Henrik, 2008]

It is easy to naively parallelize the algorithm for each pixel due to inter-pixel independency. However, since each ray shooting from each pixel may take various bounces, a naive implementation can result in a load-balancing problem, which cannot maximize the occupancy of threads. We implemented both OpenMP and CUDA versions to examine this in different parallelisms. For the OpenMP version, we developed static and dynamic scheduling. For the CUDA version, we considered a naive static scheduling implementation and improved it by evenly distributing the workload to each thread. For both versions, we examined the speedup of `float` and `double` precisions. Besides the scheduling, to understand how further to improve the speed of the ray tracing algorithm, we also implemented a hardware-accelerated version with RTX's Optix SDK [Nvidia, 2021].

Our experimental results showed that the scheduling and floating-point precision affect the speedup of the algorithm, especially on GPUs. Additionally, we found that Optix is much faster than our fastest CUDA scheduling. We conclude that a highly efficient ray tracer should harmonically orchestrate both software and hardware sides.

2 Ray Tracing Algorithms

2.1 Serial Algorithm

To render an image plane with size $H \times W$, we can formalize the color of each pixel (y, x) by the global illumination model as

$$c(y, x) = \frac{1}{S} \sum_{s=1}^{S} L_s(y, x),$$

where S is the number of samples cast from the pixel (y, x) and $L_s(y, x)$ is the ray's color, which corresponds to the objects it can hit. To determine the light color $L_s(y, x)$, we can decompose it as

$$L_s(y, x) = L_s(y, x, 0) + L_s(y, x, 1) + \cdots, \tag{1}$$

where $L_s(y, x, 0)$ is the color of the object that it can directly hit, and $L_s(y, x, d)$ for $d >= 1$ means the colors of objects that it can hit with d bounces. Note that the ray direction after each bounce may change. We consider three kinds of object materials to determine the new ray direction: diffuse, specular, and dielectric materials. For diffuse materials, they can reflect a ray in any direction, so we randomly choose a reflected direction in the implementation. For specular materials, only reflect a ray to a specific direction by the reflection laws. For dielectric materials, they can both (partially) reflect and (partially) refract a ray, so we can cast two rays after hitting a dielectric object. Since (1) is a recursive process that may not stop, we can truncate it to only D terms. After each bounce, the ray's intensity usually decreases, and the last terms are eventually sufficiently small to be neglected. The overall serial algorithm is described in Algorithm 1 and 2.

Assume that whether a ray can hit an object can be determined in a constant time in 2, and it linearly checks each object in a list of objects O, the overall time complexity of Algorithm 1 is $\mathcal{O}(HWS|O| \cdot 2^D)$, where $|O|$ is the number of objects. To alleviate the computational difficulty introduced by the 2^D term, we can modify Algorithm 2 to a random version, Algorithm 3, with a single branch, and the new complexity is $\mathcal{O}(HWS|O| \cdot D)$.

Algorithm 1 `Render` function of a serial ray tracing

Require: image height H and width W,
Require: number of samples S, maximum depth D, list of objects O, output color array c
  for $y = 0 \cdots H-1$ **do**
    for $x = 0 \cdots W-1$ **do**
      tmp_$c \leftarrow (0,0,0)$
      for $s = 0 \cdots S-1$ **do**
        ray $\leftarrow$ random direction shooting from (y, x)
        tmp_$c \leftarrow$ tmp_c+ `Cast_ray`(ray, O, D) ▷ Algorithm 2
      end for
      $c(y, x) \leftarrow$ tmp_c/S.
    end for
  end for

Algorithm 2 `Cast_ray` function of serial ray tracing with recursion

Require: ray r, list of objects O, depth d
  if $d \leq 0$ **then** return $(0,0,0)$ ▷ If r is too deep, return black
  end if
  bool can_hit $\leftarrow$ false
  for $o = 0, \cdots |O| - 1$ **do** ▷ Check if r can hit any object
    can_hit $\leftarrow$ can_hit or r->hit(O[o])
  end for
  if can_hit **then**
    attenuation $\leftarrow$ attenuation rate of the hit object
    emission $\leftarrow$ emission of the hit object ▷ When the object is a light source, it is nonzero
    $p \leftarrow$ hit point of the object
    if the hit object is diffuse **then**
      reflected_ray $\leftarrow$ random direction starting from p
      return emission+attenuation·`Cast_ray`(reflected_ray, O, $d-1$)
    else if the object is specular **then**
      reflected_ray $\leftarrow$ the reflected direction starting from p
      return emission+attenuation·`Cast_ray`(reflected_ray, O, $d-1$)
    else if the object is dieletric **then**
      reflected_rate $\leftarrow$ reflection rate of the hit object
      refracted_rate $\leftarrow$ 1.0 - reflected_rate
      reflected_ray $\leftarrow$ the reflected direction starting from p
      refracted_ray $\leftarrow$ the refracted direction starting from p
      return emission+attenuation· ((reflected_rate·`Cast_ray`(reflected_ray, O, $d-1$)+
                (refracted_rate·`Cast_ray`(refracted_ray, O, $d-1$))
    end if
  else return (0,0,0) ▷ If r hits nothing, return background color
  end if

2.2 OpenMP Parallelization

Algorithm 1 utilizes three nested for loops. We tried three different implementations of OpenMP for parallelization and concluded that the simplest implementation is the best performing one testing on different devices with different precisions. The first two loops are used for casting rays from each pixel on our $H \times W$ image plane. Since each pixel casts rays to determine its color independently, it is intuitive to use OpenMP for parallelization. We can parallel the outermost loop, the second loop, or both loops simultaneously. The height H and width W usually do not differ significantly, so parallelizing the first loop is similar to parallelizing the second loop. However, parallelizing both loops together may potentially accelerate the process. The implementation of parallelizing the outer loop is in Algorithm 4. We can uncomment the `collapse(2)` in the same algorithm to parallelize two loops jointly. We used a loop collapsing via the collapse clause since the two loops are perfectly nested and independent.

Algorithm 3 `Cast_ray` function of serial ray tracing with randomized recursion

Require: ray r, list of objects O, depth d
  if $d \leq 0$ **then** return $(0, 0, 0)$ ▷ If r is too deep, return black
  end if
  bool can_hit ← false
  for $o = 0, \cdots |O| - 1$ **do** ▷ Check if r can hit any object
    can_hit ← can_hit or r->hit(O[o])
  end for
  if can_hit **then**
    attenuation ← attenuation rate of the hit object
    emission ← emission of the hit object ▷ When the object is a light source, it is nonzero
    p ← hit point of the object
    if the hit object is diffuse **then**
      scattered_ray ← random direction starting from p
    else if the object is specular **then**
      scattered_ray ← the reflected direction starting from p
    else if the object is dieletric **then**
      reflected_rate ← reflection rate of the hit object
      r ← random number between 0 and 1
      if r < reflected_rate **then**
        scattered_ray ← the reflected direction starting from p
      else
        scattered_ray ← the refracted direction starting from p
      end if
    end if
    return emission+attenuation·`Cast_ray`(scattered_ray,$O, d - 1$)
  else return (0,0,0) ▷ If r hits nothing, return background color
  end if

Algorithm 4 `Render` function of a OpenMP ray tracing

Require: image height H and width W,
Require: number of samples S, maximum depth D, list of objects O, output color array c
  #pragma omp parallel for schedule(static/dynamic)
  // #pragma omp parallel for collapse(2)
  for $y = 0 \cdots H - 1$ **do**
    for $x = 0 \cdots W - 1$ **do**
      tmp_$c \leftarrow (0, 0, 0)$
      // #pragma omp parallel for reduction(+: tmp_c)
      for $s = 0 \cdots S - 1$ **do**
        ray ← random direction shooting from (y, x)
        tmp_c ← tmp_c+ `Cast_ray(ray`$, O, D)$ ▷ Algorithm 3
      end for
      $c(y, x)$ ← tmp_c/S.
    end for
  end for

After testing both implementations, we concluded that collapsing the loops does not provide better performance. The main reason behind it is that loop ranges are too large compared to the available number of threads to accelerate the algorithm efficiently. Additionally, we can also parallelize the innermost loop of Algorithm 1 with a reduction on the shared variable tmp_c. However, it did not accelerate the performance significantly since it would add additional overhead to slow down the outer-loop parallelization.

From all of the above, we decided to keep only the outer-loop parallelization and focus more on differing the scheduling and the number of threads. Scheduling matters because the computational load on each pixel is different depending on whether the casted rays hit objects in the scene or not. Supporting evidence of the above can be found in the results section.

2.3 CUDA Parallelization

Since each pixel casts rays independently, we can parallelize Algorithm 1 by simply launching HW threads in total, each one of which renders a pixel in parallel. However, CUDA's default stack size per thread is smaller than the stack size in the serial version, so the recursive function in Algorithm 3 may blow up the stack. Although we can configure a larger CUDA's stack size, the implementation is not robust for a larger maximum depth D and different GPU architectures. For stability of the implementation, we replace the recursive function with an iterative one since it only contains one branch per call, resulting in Algorithm 5. The complete naive CUDA implementation is Algorithm 7.

Algorithm 5 Iterative `Cast_ray_iter` function

Require: ray r, list of objects O, depth D
  $c \leftarrow (0, 0, 0)$
  cur_attenuation $\leftarrow (1, 1, 1)$
  cur_ray $\leftarrow r$
  for $d = 0 \cdots D - 1$ **do**
    bool can_hit $\leftarrow$ false
    for $o = 0, \cdots |O| - 1$ **do** ▷ Check if r can hit any object
      can_hit $\leftarrow$ can_hit or r->hit(O[o])
    end for
    if can_hit **then**
      attenuation $\leftarrow$ attenuation rate of the hit object
      emission $\leftarrow$ emission of the hit object ▷ When the object is a light source, it is nonzero
      $p \leftarrow$ hit point of the object
      $c \leftarrow c+$ cur_attenuation$\cdot$emission
      cur_attenuation $\leftarrow$ cur_attenuation$\cdot$attenuation
      if the hit object is diffuse **then**
        cur_ray $\leftarrow$ random direction starting from p
      else if the object is specular **then**
        cur_ray $\leftarrow$ the reflected direction starting from p
      else if the object is dieletric **then**
        reflected_rate $\leftarrow$ reflection rate of the hit object
        $r \leftarrow$ random number between 0 and 1
        if $r <$ reflected_rate **then**
          cur_ray $\leftarrow$ the reflected direction starting from p
        else
          cur_ray $\leftarrow$ the refracted direction starting from p
        end if
      end if
    else return cur_attenuation$\cdot(0, 0, 0)$ ▷ If r hits nothing, return background color
    end if
  end for
  return c

Algorithm 6 `Render` kernel of a naive CUDA ray tracing

Require: image height H and width W,
Require: number of samples S, maximum depth D, list of objects O, output color array c
  $y \leftarrow$ blockDim.$y\times$blockIdx.y+threadIdx.y
  $x \leftarrow$ blockDim.$x\times$blockIdx.x+threadIdx.x
  tmp_$c \leftarrow (0, 0, 0)$
  for $s = 0 \cdots S - 1$ **do**
    ray $\leftarrow$ random direction shooting from (y, x)
    tmp_$c \leftarrow$ tmp_$c+$ `Cast_ray_iter`(ray, O, D) ▷ Algorithm 5
  end for
  $c(y, x) \leftarrow$ tmp_c/S.

However, the naive CUDA implementation has the load balancing issue since each ray in each thread may bounce at different times. A reasonable solution for this is to arrange each thread's workload

dynamically. If some threads finish earlier, they can keep taking other tasks. Unlike the OpenMP version, CUDA cannot launch each thread individually but 32 threads in a warp. Alternatively, we can use fewer threads, split the image pixels into several chunks of workload, and assign each thread to compute more than one pixel over these chunks. This method was mentioned in [Chen et al., 2009]. Since the physical number of total threads in CUDA is less than the number of pixels, the speed of naive implementation relies on CUDA's underlying thread scheduling algorithm and may not balance the workload. With the chunking implementation, we can force each thread to be in charge of different local regions of the image. We assume that the local workloads in each region are similar due to the color continuity, so distributing the uneven inter-region workloads to each thread may be helpful. Another reason for this method is that CUDA can reuse the existing threads without recreating new threads, as suggested in [Harris, 2013].

Algorithm 7 `Render` kernel of a chunking CUDA ray tracing

Require: image height H and width W,
Require: number of samples S, maximum depth D, list of objects O, output color array c
Require: number of chunks in height chunk_y, number of chunks in width chunk_x
  $y \leftarrow$ blockDim.$y\times$blockIdx.y+threadIdx.y
  $x \leftarrow$ blockDim.$x\times$blockIdx.x+threadIdx.x
  chunk_size_$y = H/$chunk_y
  chunk_size_$x = W/$chunk_x
  for chunk_id_$y = 0, \cdots$,chunk_$y - 1$ **do**
    for chunk_id_$x = 0, \cdots$,chunk_$x - 1$ **do**
      tmp_$c \leftarrow (0, 0, 0)$
      $y2 \leftarrow y$+chunk_size_$y\cdot$chunk_id_y
      $x2 \leftarrow x$+chunk_size_$x\cdot$chunk_id_x
      for $s = 0 \cdots S - 1$ **do**
        ray $\leftarrow$ random direction shooting from $(y2, x2)$
        tmp_$c \leftarrow$ tmp_c+ `Cast_ray_iter`(ray, O, D) ▷ Algorithm 5
      end for
      $c(y2, x2) \leftarrow$ tmp_c/S.
    end for
  end for

2.4 Optix Acceleration

Optix is a ray tracing API developed by NVIDIA, which can calculate needed mathematics equations in ray-tracing utilizing RT Cores in GPU. In this implementation, we use Optix API to do `context` initialization, which includes the initialization of ray generation and ray missing. Then, we invoke `RT_CALLABLE_PROGRAM` to create different objects, i.e., ground, light, moving sphere, metal sphere, and noise sphere and add those objects into `Optix::Group`. After that, we can use its `validate` function to render the image. Finally, we print out image from `Optix::Buffer` and destroy `Buffer` and `Context`.

3 Experiments

3.1 Ray Tracer Implementation and Compilation

For our implementation of the serial ray tracer, we directly modify the code from the first two books in `Ray Tracing in One Weekend` series [Shirley, 2016, Allen, 2018]. Since the objects are stored in a pointer of object pointers, each of which contains material or texture pointers, and a tree structure BVH for accelerating the hit test of objects, it is non-trivial to copy these objects to CUDA's global memory contiguously. For simplicity, we only considered spheres and left the CUDA-friendly implementation of generic objects as future work. To understand how floating point precision may affect the speedup, we implemented both `float` and `double` for both OpenMP and CUDA versions. For OptiX, we use only `float` since the hardware acceleration is only designed for `float`. For better performance, we enabled level-3 compiler optimizations with the flag `-O3` for all versions.

3.2 Experimental Settings

In our experiments, we choose the number of samples $S = 1,000$ and maximum depth $D = 10$ for our sphere scene as shown in Figure 2. For the testing machines, we use `cuda3` in Courant's severs since it is the fastest GPU among `cuda1` to `cuda5`. We also consider `cuda2` for experiments since it is the only RTX GPU that can run OptiX SDK in all the severs.

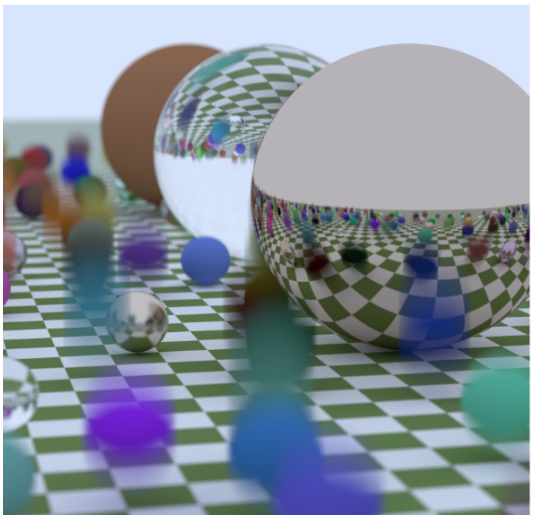

Figure 2: Sphere scene with $S = 1,000$ and $D = 10$.

4 Results

4.1 Serial Algorithm

From Table 1, we can observe that the running time of the serial implementation on both `cuda2` and `cuda3` vary with different floating-point precisions. On `cuda2`, `double` type is faster than `float` type, while the result on `cuda3` is the opposite. On `cuda2`, `double` is 9% faster than `float`, while for `cuda3`, `float` is 1% faster than `double`. Therefore, the difference of running time with different precisions is not very significant.

`cuda2 (double)`	`cuda2 (float)`	`cuda3 (double)`	`cuda3 (float)`
9,652	10,524	10,123	10,028

Table 1: Running time (s) of serial implementation with different floating-point precisions

4.2 OpenMP Parallelization

Table 2 validates the previous discussion of OpenMP implementations. For simplicity, we only present the results for `float` on `cuda2`. As was expected, the simple implementation, i.e., parallelizing the outer-loop only, overall outperforms other implementations.

`simple`	`collapse`	`simple+reduction`	`reduction`
3048	3220	3131	4672

Table 2: Running time (s) of different OpenMP implementations

As shown in Table 3, on average, the results for dynamic scheduling work better than for static scheduling since it can alleviate the load-balancing issue. We can notice that the impact of dynamic scheduling is smaller for a larger number of threads due to the thread creation and synchronization overhead. Similarly, more threads may slow down the performance, as in the results on `cuda3 (double)`.

NumThreads	`cuda2 (double)`	`cuda2 (float)`	`cuda3 (double)`	`cuda3 (float)`
4 (static)	4533 (2.13×)	4745 (2.22×)	4995 (2.03×)	4886 (2.05×)
4 (dynamic)	3797 (2.54×)	4020 (2.62×)	**4269 (2.37.×)**	5175 (1.94×)
8 (static)	3325 (2.90×)	3198 (3.29×)	4498 (2.25×)	4684 (2.14×)
8 (dynamic)	3127 (3.09×)	**3048 (3.45×)**	4409 (2.30×)	4552 (2.20×)
12 (static)	3232 (2.99×)	3506 (3.00×)	4912 (2.06×)	4210 (2.38×)
12 (dynamic)	**3116 (3.10×)**	3251 (3.24×)	4807 (2.11×)	**4060 (2.47×)**

Table 3: Running time (s) and speedup of simple OpenMP parallelization with different floating-point precisions. Each boldface number is the fastest one on the corresponding machine and precision.

4.3 CUDA Parallelization

As shown in Table 4, naive CUDA implementation with block size 16×16 or 16×32 performs the best on all machines and precisions. Compared to an 8×8 block, a 16×16 block consists of more warps, and CUDA's scheduler can reuse existing threads more times, so it has less overhead of thread initialization. Although a 32×32 can reuse more warps than a 16×16 block, it would request too many registers and, hence, regress the performance. For `double` precision, CUDA cannot even successfully launch the kernels due to the out-of-resource error. Therefore, a moderate block size like 16×16 or 16×32 outperforms others.

blockDim	`cuda2 (double)`	`cuda2 (float)`	`cuda3 (double)`	`cuda3 (float)`
8×8	390 (24.75×)	358 (29.40×)	237 (42.71×)	346 (28.98×)
16×16	**332 (29.07×)**	342 (30.77×)	**203 (49.87×)**	**336 (29.85×)**
16×32	339 (28.47×)	**326 (32.28×)**	210 (48.20×)	343 (29.24×)
32×32	N/A	372 (28.29×)	N/A	372 (26.96×)

Table 4: Running time (s) and speedup of naive CUDA parallelization with different floating-point precisions. The N/A entries are due to CUDA's out-of-resource error with too many registers. Each boldface number is the fastest one on the corresponding machine and precision.

From Table 5, we can observe that a smaller block size 8×8 with a larger chunk size 4×4 or 5×5 can boost the performance on `cuda2 (float)`, `cuda3 (double)`, and `cuda3 (float)`, which may be because a small block size 8×8 can only reuse two warps, and more chunks of tasks can force them to reuse more and decrease the imbalanced loads among threads. For `cuda2 (double)`, its best configuration of block and chunk sizes corresponds to the naive parallelization, which may imply that CUDA's default scheduler works well for this case. For the speedup, we can find that CUDA accelerates `float` type on `cuda2` and `cuda3` by more than 80 times, which implies CUDA may work better for `float` type and may explain why chunking is not very helpful for `cuda2 (double)`. Interestingly, we can also see that a chunking implementation with a chunk size 1×1 can be much faster than its equivalent naive implementation. Their only difference is that the former contains additional two loops that only iterate one chunk. A possible explanation may be such an implementation can suggest CUDA's compiler that reuses more existing threads than the default setting.

blockDim, chunkDim	`cuda2 (double)`	`cuda2 (float)`	`cuda3 (double)`	`cuda3 (float)`
8×8, 1×1	374 (25.81×)	179 (58.79×)	232 (43.63×)	239 (41.96×)
8×8, 2×2	417 (23.15×)	194 (54.25×)	240 (42.18×)	235 (42.67×)
8×8, 4×4	386 (25.01×)	**129 (81.58×)**	257 (39.39×)	**114 (87.96×)**
8×8, 5×5	348 (27.74×)	133 (79.13×)	**166 (60.98×)**	128 (78.34×)
16×16, 1×1	**329 (29.34×)**	163 (64.56×)	201 (50.36×)	230 (43.60×)
16×16, 2×2	357 (27.04×)	190 (55.39×)	213 (47.53×)	237 (42.31×)
16×16, 4×4	474 (20.36×)	142 (74.11×)	235 (43.08×)	116 (86.45×)
16×16, 5×5	465 (20.76×)	145 (72.58×)	188 (53.85×)	146 (68.68×)

Table 5: Running time (s) and speedup of chunking CUDA parallelization with different floating-point precisions. Each boldface number is the fastest one on the corresponding machine and precision.

To compare the workloads per thread before and after chunking, we visualize the average number of bounces (depths) of all threads in Figure 3. The workload of the naive implementation is imbalanced. Its top region is the background that only bounces once, while the bottom region can bounce at most 9 or 10 times. By contrast, the workload of the 4×4 chunking algorithm is more balanced, and the maximum average depth is around 4. The 2.5× reduction of average workload corresponds to the 2.95× speedup from naive to 4×4 chunking implementation on `cuda3 (float)` in Table 4 and 5.

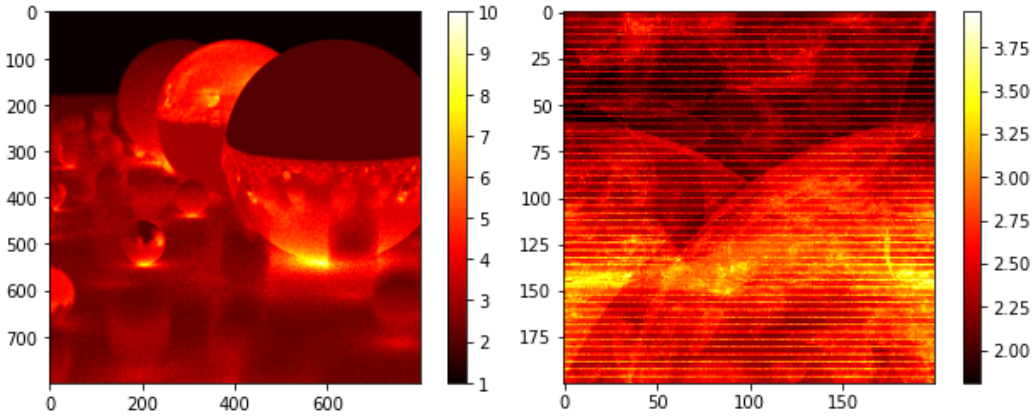

Figure 3: Average number of bounces (depths) of all threads with naive (left) and 4×4 chunking (right) CUDA algorithms

4.4 Optix Acceleration

The version of Optix we used is 6.5.0, and the data format is a single-precision floating-point. We only test it on `cuda2` because the GPU driver in other servers does not fit this version of Optix. To render the same image in Optix, it only costs 2 seconds, which is about 65× speedup when compared with `cuda2 (float)` since it takes full advantage of hardware resources, to specific, RT Cores. This result shows that hardware acceleration is very effective, but it entails the efforts of hardware craft and architecture.

5 Conclusion and Future Work

This report experimented with and compared a serial ray tracer, its OpenMP and CUDA parallelizations with different scheduling and floating-point precisions, and hardware acceleration with RTX Optix's SDK. In conclusion, both the scheduling and floating-point representations are vital for the performance gain from a software perspective. To maximize the efficiency of the ray tracer, it is also crucial to consider the hardware. For future work, we would like to experiment with different scenes to see how they would affect the speedup. Additionally, we may study how to implement CUDA-friendly storage of the objects. The current implementation with the pointer of object pointers is generic for various types of objects. Still, it is not easy to copy all the pointers contiguously into the GPU global and shared memory, which may regress the performance due to poor bandwidth.

References

R. Allen. `https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/`, 2018.

A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring)*, 1968.

L. Chen, H. Das, and S. Pan. An implementation of ray tracing in cuda cse. 2009.

M. Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. `https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/`, 2013.

Henrik. https://commons.wikimedia.org/w/index.php?curid=3869326, 2008.

S. Marschner and P. Shirley. Fundamentals of computer graphics, fourth edition. 2016.

Nvidia. NVIDIA OPTIX™ RAY TRACING ENGINE. https://developer.nvidia.com/rtx/ray-tracing/optix, 2021.

P. Shirley. Ray Tracing in One Weekend — The Book Series. https://raytracing.github.io/, 2016.