Parallelising Ray Tracing on GPUs

Isaac Chandler - Daniel Cutfield - Yogesh Dangwal

Introduction

The evolution of digital graphics rendering has necessitated the increase in processing power and speed. A graphics processing unit (GPU) is one of the tools that has provided better performance for these processes. Modern graphical effects such as ray tracing, create a more photorealistic looking image but they require large amounts of computation, hence they take a long time to render. A GPU can speedup this process with parallelisation on its many cores. GPUs are used to increase performance in many other areas such as gaming, machine learning, cryptocurrency mining and statistical modelling.

Background

GPUs

A graphics processing unit (GPU) is a type of processor that specialises in performing computationally intensive tasks efficiently. A GPU contains thousands of cores which can be used in parallel to perform operations quickly and efficiently. This makes it more efficient than a CPU to perform these intensive tasks as a CPU generally only has 4 - 16 cores. It was initially designed to accelerate the rendering of images but over time, it has been utilised in many other processes such as scientific modelling and machine learning. With the increasing demands of graphics rendering, the GPU has become an integral part of modern rendering systems.

GPUs excel at rendering 3D images with high fidelity as it can operate on large amounts of data simultaneously. With the additional processing power, more visual effects can be added to increase the photorealism of the image such as ray tracing and textures. The computations required for these effects can be parallelised and performed very efficiently on a GPU due to its high number of cores. These effects are also commonly used in gaming.

Architecture of GPUs

Parallelism in GPU architecture is a characteristic feature of efficient data processing. The Single Instruction, Multiple Thread (SIMT) execution model is important in this regard. SIMT allows the simultaneous processing of different data sets with the exact instructions, which is fundamental in applications requiring high computing power, such as matrix processing, image display, and neural network computations [1].

The primary design component of GPUs is Streaming Multiprocessors (SMs), which have multiple CUDA cores(for Nvidia gpu) acting as processing units. Each SM consists of arithmetic logic units (ALUs), texture units, and shared memory, where each component is built to maximise performance and efficiency [2]. This structure allows the SM to operate independently and parallel execution of instructions, so that computing performance increases dramatically.

The concept of the memory hierarchy in GPU architecture addresses the memory requirements of parallel computing. There are often global, shared, and local memories, each with unique characteristics that serve specific needs. Global memory provides more storage capacity but suffers

from high latency. In contrast, shared memory that can be accessed by threads in SM offers lower latency and higher bandwidth. Local memory is allocated for private data storage for each thread.

Thread execution and scheduling are fundamental to the GPU architecture for efficient parallelism exploitation. Thread blocks cover thousands of threads running in parallel. Thread scheduling by the hardware scheduler in each SM is essential to ensure efficient resource utilisation. This scheduling determines the allocation of thread blocks to SMs and individual threads to execution resources.

Memory coalescing and caching are strategies GPUs use to optimise memory access patterns. Coalesced memory permits multiple threads to access contiguous memory places, decreasing memory latency and maximising memory throughput. GPUs also use diverse caching mechanisms, which include texture caches and consistent caches, to lower memory access latency and capitalise on data locality.

The GPU's structure is bolstered by using programming frameworks and Application Programming Interfaces (APIs), like CUDA and OpenCL. These APIs present a high-level interface that allows developers to utilise GPU parallelism effectively, ensuring efficient programming and usage of GPU resources [3]. They expose GPU abilities to developers and help translate these abilities into realistic solutions.

GTX 1080

The Nvidia GeForce GTX 1080, with 2,560 CUDA cores and a boost clock of 1733 MHz, facilitates highly parallel processing. The GTX 1080 features GDDR5X memory with a high bandwidth of 320 GB/s. High memory bandwidth is essential in parallel computing, ensuring fast data transfer between the GPU core and memory. By optimising throughput by reducing latency, higher memory bandwidth improves the performance of parallel computations.

Additionally, the GTX 1080 has a computing power of cuda 6.1, indicating its proficiency in managing complex parallel tasks. This capability has been extended to support advanced features such as dynamic parallelism, which allows the kernel to start new kernels and execute multiple kernels simultaneously.). The Nvidia CUDA programming model supported by the GTX 1080 provides developers with powerful tools and libraries for parallel computing applications. By allowing developers to write parallel code using high-level languages such as CUDA C/C++, the CUDA's based programming model makes it easy to use the GTX 1080's power more efficiently.

CUDA

Nvidia's CUDA (Compute Unified Device Architecture) C/C++ is a parallel computing platform and programming style. It has gained widespread acceptance in clinical computing and general-purpose computing on graphics processing units because of its ability to leverage the parallel processing capabilities of Nvidia GPUs.CUDA is an extension to C/C++ that allows developers to perform capabilities (known as kernels) on Nvidia GPUs.It includes key constructs for designing and launching kernels, thread synchronisation, and GPU memory management. Using CUDA C/C++, developers may take advantage of GPUs' high parallelism by developing code that executes on several threads at the same time [4].

The CUDA API includes a comprehensive set of features for handling GPU memory, controlling device execution, and managing errors. These APIs provide developers with low-level access to the GPU, allowing them to optimise their code for optimal performance. Developers can use CUDA's memory management APIs, for example, to create and deallocate memory on the GPU, transmit data

between the host (CPU) and the device (GPU), and manage GPU memory caches. Another important aspect of CUDA C/C++ is its ability to optimise GPU code by various programming techniques. This includes shared memory programming, which allows threads in the same block to share data more quickly; warp-level primitives, which improve warp-level synchronisation; and dynamic parallelism, which allows a GPU kernel to launch another kernel. CUDA C/C++ and its API represent a powerful tool for leveraging the processing power of GPUs for a wide range of applications. They provide the means to tap into the immense parallel processing capabilities of modern GPUs, enabling developers to accelerate their applications beyond what is possible with CPUs alone

Ray Tracing

Ray tracing is a lighting effect used in graphics rendering to simulate the movement of light rays and how they react to objects in an image. It is used to make the image appear photorealistic with the use of reflections and refractions. It is very computationally intensive, hence the use of a GPU is essential to increase the speed of the process. It traces the path of light rays individually as they travel between a light source and the camera. This allows for a more accurate representation of the light when compared with other rendering methods such as rasterization.

The process of ray tracing generally contains the following steps:

- Ray Generation - Primary rays are cast through the image from the camera's point of view.
- Intersection Testing - The rays are tested to see whether or not they intersect with objects in the scene.
- Shading and Illumination - When a ray intersects with an object, more rays are cast to the point of intersection, which provides data on how the light interacts with the surface.
- Reflection and Refraction - These rays provide data on how light bounces off reflective surfaces or how light passes through transparent objects

  .

Bloom

Bloom is a post-processing effect that simulates the actual camera artefact known as blooming or glow, which is generally caused by brightly lit parts in an image. This is caused by diffraction which is not simulated in ray tracing. It offers an important aesthetic touch, increasing visual attractiveness and realism, especially when used with sophisticated lighting techniques like ray tracing. Numerous studies have shed light on how bloom, when used successfully after ray tracing, may improve overall graphic fidelity, providing the user with a more immersive and aesthetically compelling experience.

Implementation

For our ray tracing implementation we use the CUDA C++ API this allows us to write our CPU and GPU code in essentially the same language and even allows reuse of most of it. Our implementation has the functionality to load test scenes from a simple custom file format, construct bounding volume hierarchies that can accelerate the process of raytracing, perform the ray tracing itself and output a final image. For this project we have implemented GPU acceleration for the entire ray tracing step and part of the image rendering step. To measure the speedup that the GPU is able to provide to ray tracing we also run the same ray tracing kernels on the CPU in parallel using OpenMP.

Bounding volume hierarchy

The most critical data structure for high performance ray tracing is the bounding volume hierarchy (BVH). BVHs are a binary tree structure that partitions the geometry in the scene so that when a ray

needs to find out which object it hits it doesn't need to test against every object in the scene. Each node within the bounding volume hierarchy contains a box which surrounds its children. If a ray doesn't intersect this box, tests with all its children can be skipped. The leaf nodes of this tree contain all the primitive geometry in the scene (triangles in our case).

Ray tracing phase

As the ray tracing phase is the most time consuming step of our application this is the section that is the section that is currently parallelised on the GPU. To manage this timing and keep complexity under control this is broken down into multiple substeps. These steps are shown in figure 1.

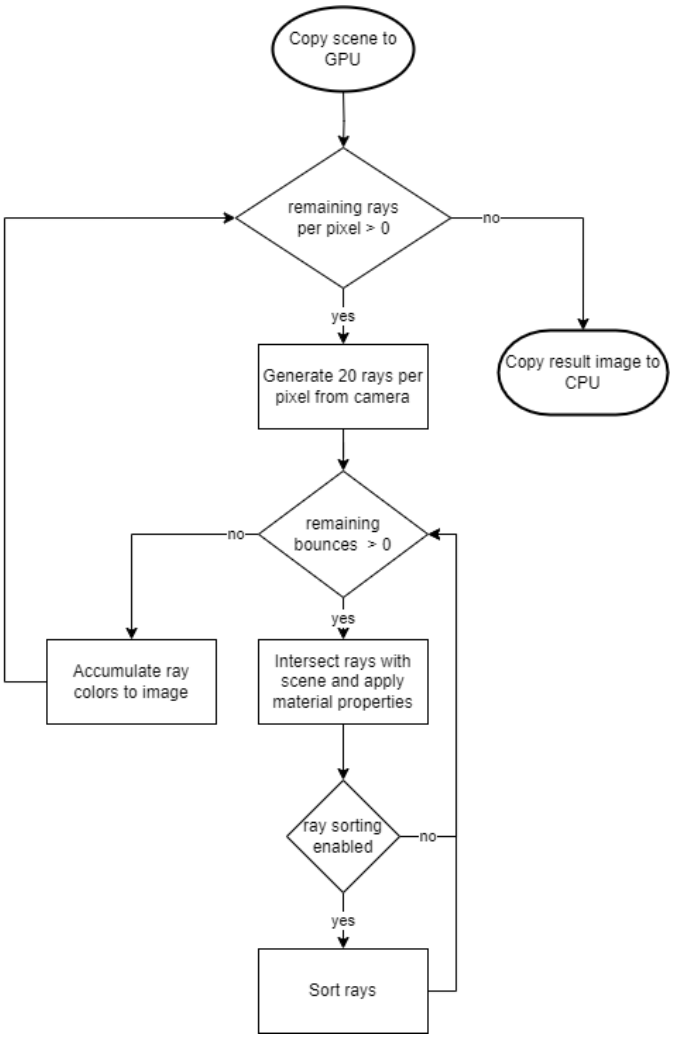

Figure 1 - Flowchart showing ray tracing process

CPU to GPU copy

The first stage is copying the scene data from CPU memory to GPU memory. The scene data includes information about the camera location, output image resolution, scene geometry, BVH, surface material properties and the environmental lighting texture. The next step which is initial ray generation is not dependent on all of this data. Ray generation only requires the basic camera camera and output image information to start executing. We take advantage of this by performing the copy of the scene geometry, surface material properties, bounding volume hierarchy and the environmental lighting texture on a background CUDA stream and the copy of camera and image metadata on the

main CUDA stream. All operations in a single CUDA stream will execute serially but multiple CUDA streams can execute in parallel. This allows the ray generation step to start after only the very small metadata has been copied to the GPU and operate in parallel with the copy of the much larger scene data.

The scene metadata is placed in constant memory since it will never be written on the GPU, is used very frequently during the ray tracing process and is very small. Although the rest of the scene data is also read only, it is placed in global memory since not all parts of it will be accessed by every thread and it is too large to fit into constant memory anyway.

Ray generation

Ray generation is responsible for initially creating the rays emitted from the camera that will be used for later processing. Since there could be thousands of rays for each of the millions of pixels in the output image which would exceed the GPUs memory capacity we generate and then process rays in batches of 20 rays per pixel. This also prevents an issue encountered during testing where the OS would terminate any kernel running for longer than 5 seconds since the test GPU was also the primary display GPU by breaking the computations into short chunks. For ray generation each GPU thread generates one ray and there are 128 threads per block. This is also parallelised on the CPU implementation using an OpenMP parallel for loop.

Ray propagation

Propagating in itself consists of a few substeps. These are traversing the BVH to determine which geometry to test for intersections against, performing intersections against geometry and then applying material properties of the hit geometry to update a ray's colour and direction. These steps are all performed in a single kernel so the output values of each step can be kept stored in registers for the next step for extremely quick access.

For traversing the tree of the BVH it is critical that we use an iterative method with a stack to perform depth first search instead of a recursive implementation. With a recursive implementation different threads within the same warp may or may not enter into different branches of the recursive traversal causing execution to diverge. With an iterative implementation all threads will always return to the top of the loop and just load different data from the stack. The exact speedup this gives hasn't been measured since the recursive implementation hits the five second cutoff from the OS discussed above.

Ray intersection testing uses the Möller-Trumbore algorithm [5] for testing ray intersections with triangles in the scene. This is a standard algorithm used to perform ray-triangle intersection with no special adjustments made to run on the GPU.

Once an intersection has been determined the material properties of the triangle that was intersected will be looked up based on an array of material indices which runs parallel to the array of triangles. Using a parallel array for material indices avoids polluting the very small cache of the GPU with material indices when just performing triangle intersections since most triangles won't intersect and therefore the material index won't be needed. The material's colour is then applied to the ray and the material determines how the light will reflect (randomly for diffuse surfaces and mirror-like for metallic surfaces) or refract for transparent surfaces. Different materials mean that different threads will take different branches in this computation so best performance will be achieved when rays in one warp hit surfaces with the same material.

The parallelisation strategy for this step is the same as with ray generation. On the GPU we use one thread per ray and 128 threads per block. On the CPU we use OpenMP parallel for.

Ray sorting

In order to reduce the issues with divergent execution discussed in the previous step it is desirable for similar rays to be grouped together since these rays are likely to traverse the same part of the BVH, intersect similar triangles and use similar materials. This can be achieved by sorting according to some criterion which will be similar for similar rays [6]. Ray sorting is never performed when raytracing on the CPU since it wouldn't provide much benefit so isn't worth the cost.

We use a 30 bit origin-direction Morton coding for this. Morton coding assigns an integer index to each point on a Z-curve which will map points which are close together in 3D space to integers which are close together. Computing this code involves quantizing the x, y and z components of the origin and direction vectors to 5 bits and then interleaving the bits of the thread components to get a 15-bit morton code for the origin and the direction. The origin code is used as the top 15 bits of key and the direction code for the bottom 15 bits. This computation can be done with only a handful of bitwise operations performed at the end of the ray propagation step.

For the actual sorting operation radix sort from the cub library was used since this provides the fastest sorting implementation for our target GPU and in order for the architectural benefits of ray coherence to matter the sorting must be very fast. Since radix sort is linear in the number of elements the more rays we trace each pass the better since this will reduce the overhead of calling the sort kernel and will mean there are more rays to choose from when trying to find a similar ray.

Ray accumulation

Ray accumulation takes an insignificant amount of time compared to the other two steps since it's only 3 additions per ray. This is still worth implementing on the GPU since performing the reduction there avoids copying data back to the CPU for every ray processed, instead only each pixel must be copied. Since performance is not critical here we use the simplest approach of doing an atomic add for each ray to its corresponding pixel value. This could be further accelerated by performing intrawarp and intrablock reductions before doing an atomic add but this wouldn't provide much overall speedup to the program.

Bloom

For bloom we first apply a high pass filter to isolate the image's bright areas. The minimum brightness required for a pixel to be considered 'bright' is determined by a threshold that is proportional to the scene's ray count. After isolating the bright parts, we smooth their appearance with a two-pass box blur. The first pass blurs the image horizontally, while the second blurs the image vertically. This two-pass method approximates a two-dimensional blur while reducing computational complexity. Finally, the blurred, bright parts are added back to the original image, creating the bloom effect's signature glow.

Results

The ray tracing phase shown in figure 1 was timed on both the CPU and GPU with a variety of different scenes and ray tracing configurations. The GPU tests were performed on an Nvidia GTX 1080 with the CPU tests on an AMD Ryzen 5 1600. All scenes were rendered at a resolution of 1000x1000 pixels with a bounce limit of 10 per ray. The results of this testing are summarised in table 1.

Scene	Primitive count	CPU time (s)			GPU time without reordering (s)			GPU time with reordering (s)		
Rays per pixel		1	10	100	1	10	100	1	10	100
Spheres	4	0.1	0.8	6.6	0.2	0.2	0.5	0.2	0.2	0.9
Cornell	32	1.7	17.4	168.9	0.2	0.3	1.5	0.3	0.4	1.8
Cornell plus	34	1.7	17.6	171	0.2	0.3	1.5	0.2	0.4	1.8
Teapot [7]	126050	1.1	11.2	109.7	0.2	0.3	1.3	0.2	0.3	1.4
Glass teapot [7]	126050	1.6	16.0	156.3	0.2	0.4	2.0	0.2	0.3	1.9
Lamp [7]	619350	1.9	18.7	186.3	0.2	0.4	2.0	0.2	0.4	1.9

Table 1 - Time taken to perform ray tracing for different scenes

The much more complex teapot scene is able to outperform the simpler Cornell box scenes because in the teapot scene many rays hit nothing immediately or after a single bounce. This allows the majority of rays to exit early. Ray reordering only shows a benefit in the glass teapot and lamp scenes which are also the 2 most costly scenes to ray trace. The best GPU versus CPU speedups shown for 1, 10 and 100 samples per pixel respectively are 9.5, 37.4 and 98.0.

Conclusions

The investigation of ray tracing performance across CPU and GPU platforms produced intriguing findings. The GPU greatly outperforms the CPU in all evaluated cases, proving the usefulness of parallel computing architecture for such computationally heavy workloads. Rendering times may be lowered by two orders of magnitude when using the GPU, demonstrating its promise for real-time applications and high-quality graphics rendering. An increase in the number of rays per pixel has a linear effect on the CPU time while the GPU time initially shows sublinear performance scaling due to fixed costs to copy the scene to the GPU and JIT compile GPU kernels.

Furthermore, scene complexity, as measured by the number of primitives, is not a good indicator of rendering time. The Cornell box scenes are slower to render than the much more geometrically complex teapot scene. This indicates the effectiveness with which BVHs are able to prevent tests with large amounts of geometry with low overhead. Ray reordering is only able to give a slight speedup in the most complex glass teapot and lamp scenes tested. This is because ray reordering does provide some benefit to the ray tracing pass of all scenes but the fixed overhead for sorting is too large for the simpler scenes.

Future Work

Despite these positive findings, there is still plenty of room for further research and development. The existing approach might be improved by investigating more complex ways for ray reordering and

dealing with divergent execution. A more efficient method might greatly enhance performance. One approach would be to use machine learning techniques to forecast and optimise ray reordering, which might lead to more effective resource allocation and quicker rendering. Alternatives to this would be to avoid doing a full sort on the rays when reordering and doing a much cheaper approximate sort. In terms of hardware, the performance of ray tracing on newer GPU architectures with ary tracing hardware acceleration might be examined. Currently the bloom effect has been implemented in GPU only but it could be implemented in CPU and overall improvement in performance of GPU can be compared with CPU. Finally, it would be interesting to evaluate this ray tracing for more complicated scenarios with varied sorts of materials and lighting conditions on GPU performance, which might provide insights into the current approach's limitations and potential remedies.

Further performance improvements in overall runtime could also be made by parallelising the bounding volume hierarchy construction on the GPU since this has currently not been implemented and takes a significant amount of time for the more complex scenes.

Contributions

Section	Isaac Chandler	Daniel Cutfield	Yogesh Dangwal
Introduction	☐	☑	☐
Background	☐	☑	☑
Implementation Ray tracing	☑	☐	☐
Implementation Bloom	☐	☐	☑
Results	☑	☐	☐
Conclusions and future work	☑	☐	☑

References

1. J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?," Queue, vol. 6, no. 2, pp. 40-53, Mar.-Apr. 2008.T. Möller and B. Trumbore, ‘Fast, Minimum Storage Ray/Triangle Intersection’, in *ACM SIGGRAPH 2005 Courses*, Los Angeles, California, 2005, pp. 7-es.

2. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," IEEE Micro, vol. 28, no. 2, pp. 39-55, Mar.-Apr. 2008.

3. J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," Computing in Science & Engineering, vol. 12, no. 3, pp. 66-78, May-Jun. 2010.

4. J. Nickolls and W. J. Dally, "The GPU computing era," IEEE Micro, vol. 30, no. 2, pp. 56-69, Mar.-Apr. 2010.

5. T. Möller and B. Trumbore, ‘Fast, Minimum Storage Ray/Triangle Intersection’, in *ACM SIGGRAPH 2005 Courses*, Los Angeles, California, 2005, pp. 7-es.

6. D. Meister, J. Boksansky, M. Guthe, and J. Bittner, ‘On Ray Reordering Techniques for Faster GPU Ray Tracing’, in *Symposium on Interactive 3D Graphics and Games*, San Francisco, CA, USA, 2020.

7. B. Bitterli, ‘Rendering resources’. 2016. [Online]. Available: https://benedikt-bitterli.me/resources/.