Department of Computer Science

Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC

Justin Lewis Salmon

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

August 2019 | CSMSC-19

Declaration:

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Justin Lewis Salmon, August 2019

EXECUTIVE SUMMARY

OpenMC is a CPU-based Monte Carlo particle transport simulation code recently developed in the Computational Reactor Physics Group (CRPG) at MIT, and which is currently being evaluated by the UK Atomic Energy Authority (UKAEA) for use on the ITER fusion reactor project. In this work, we present a novel port of OpenMC to run on the new Ray Tracing (RT) cores in NVIDIA's latest Turing GPUs. We show here that the OpenMC GPU port yields up to 9.8x speedup on a single node over a 16-core CPU using the native constructive solid geometry, and up to 13x speedup using approximate triangle mesh geometry. Furthermore, since the expensive 3D geometric operations required during particle transport simulation can be formulated as a ray tracing problem, there is an opportunity to gain even higher performance on triangle meshes by exploiting the RT cores in Turing GPUs to enable hardware-accelerated ray tracing. Extending the GPU port to support RT core acceleration yields between 2x and 20x additional speedup on triangle meshes. We note that topological complexity of the model mesh has a significant impact on performance, with RT core acceleration yielding comparatively greater speedups as complexity increases. The principal benefits of RT cores are shown to be the caching of triangle vertex data, and the concurrent relief of expensive ray tracing operations from streaming multiprocessors (SMs).

To the best of our knowledge, this is the first work showing that exploitation of RT cores for scientific workloads is possible. The OpenMC port developed here is comparable to, or better than, other ports of similar Monte Carlo particle transport codes in terms of performance. The UKAEA are supporting this research due to the need for modern high performance GPU-based simulation codes which can take advantage of many-GPU supercomputing resources. Furthermore, the MIT CRPG are evaluating the code for inclusion into the official OpenMC repository to undergo further development.

The main contributions of this research include:

- The first known study into the use of RT cores for scientific applications.
- An optimised port of OpenMC to NVIDIA GPUs, including detailed performance benchmarking on realistic problem sizes (Chapter 9).
- A rigorous study of RT cores on their native graphics workload (Chapter 5).
- A rare evaluation of RT cores in terms of wider applicability, limitations and performance portability, intended to facilitate future studies (Chapter 11).
- A full conference paper based on this work, submitted for publication at the International Conference for High Performance Computing, Networking, Storage and Analysis 2019 (SC19) (Appendix B).

ACKNOWLEDGMENTS

First and foremost I would like to thank Simon McIntosh-Smith for all his excellent guidance and encouragement, without which this project would not have been a success. I would also like to thank Tom Deakin, Patrick Atkinson, Andrei Poenaru and James Price from the University of Bristol HPC group for all their help, and for doing such a good job on keeping the Zoo up and running. Thanks to Andy Davis at UKAEA as well, for all his excellent advice around model geometries, and for working so hard to acquire models from the ITER project.

Finally I would like to thank Veronica for all her patience and support, without which I certainly would not have been able to achieve so much.

CONTENTS

1 INTRODUCTION

Particle transport algorithms simulate interactions, such as nuclear fission and electron scattering, between particles as they travel through 3D model environments. The Monte Carlo method is used to stochastically sample large numbers of particle trajectories and interactions to calculate the *average behaviour* of particles over these 3D models, producing highly accurate simulation results compared to deterministic methods [34]. Monte Carlo particle transport has found applications in diverse areas of science, such as fission and fusion reactor design, radiography, and accelerator design [5, 7].

The Monte Carlo particle transport algorithm is highly computationally intensive, partly due to the large number of particles which must be simulated to achieve the required degree of accuracy, and partly due to certain computational characteristics of the underlying algorithm that make it challenging to fully utilise modern computing resources [22]. This limits the scale and speed at which simulations can be performed in practice.

Previous studies have successfully proven that Monte Carlo particle transport is well suited to GPU-based parallelism [3, 13, 15]. Furthermore, targeting GPUs is becoming increasingly important in High-Performance Computing (HPC) due to their proliferation in modern supercomputer designs such as Summit [14]. This work aims to take advantage of GPU parallelism to improve Monte Carlo particle transport performance, broadly targeted towards the nuclear research domain.

1.1 OPENMC

OpenMC is a Monte Carlo particle transport code focussed on neutron criticality simulations, recently developed in the Computational Reactor Physics Group at MIT [33]. OpenMC is written in modern C++, and has been developed using high code quality standards to ensure maintainability and consistency. This is in contrast to many older codes, which are often written in obsolete versions of Fortran, or have otherwise grown to become highly complex and difficult to maintain. It is partly for this reason that the UK Atomic Energy Authority (UKAEA) is currently evaluating OpenMC as a tool for simulating the ITER nuclear fusion reactor [41] on future supercomputing platforms.

OpenMC natively represents 3D model geometries using a method called *constructive solid geometry (CSG)*, which supports exact models, but lacks the ability to represent complex surfaces such as toroids. An alternative method to CSG, called *triangle meshes*, are able to represent arbitrarily complex surfaces using tessellations of small triangles, but can only produce approximate models (the accuracy of which depend upon the tessellation size). Nevertheless, triangle meshes are gaining popularity in the field due to their representational ability and ease of use, and approximate models are often an

acceptable engineering tradeoff. Furthermore, wider standardisation efforts surrounding triangle mesh file formats can potentially reduce the engineering burden often required when duplicating CSG models between different simulation codes. This work will consider both methods when designing how to parallelise OpenMC on the GPU.

OpenMC currently runs on CPUs only, using OpenMP for on-node parallelism and MPI for inter-node parallelism, and has been well studied from a performance perspective [37]. The first major piece of this work will present a novel port of OpenMC to NVIDIA GPUs (using both CSG and triangle mesh models), hypothesising that significant on-node performance improvements can be obtained on the GPU compared to the CPU. This has potentially immediate real-world benefits for the UKAEA and other institutions seeking improved Monte Carlo particle transport performance.

1.2 HARDWARE-ACCELERATED RAY TRACING

As the memory bandwidth and general purpose compute capability improvements of recent GPUs begin to plateau, GPU manufacturers are increasingly turning towards specialised hardware solutions designed to accelerate specific tasks which commonly occur in certain types of applications. A recent example of this is the inclusion of Tensor cores in NVIDIA's Volta architecture, which are targeted towards accelerating matrix multiplications primarily for machine learning algorithms [21]. An earlier example is texture memory, designed to improve efficiency of certain memory access patterns in computer graphics applications [32]. These specialised features have often been repurposed and exploited by HPC researchers to accelerate problems beyond their initial design goals.

NVIDIA's latest architecture, codenamed Turing, includes a new type of fixed-function hardware unit called Ray Tracing (RT) cores, which are designed to accelerate the ray tracing algorithms used in graphical rendering of triangle mesh models. NVIDIA advertises potential speedups of up to 10x with RT cores, which will supposedly allow computer games designers to bring real-time ray tracing to their games, opening up new levels of photorealism.

The ray tracing algorithms used in graphical rendering are similar in nature to Monte Carlo particle transport algorithms, in the sense that both require large numbers of linear geometric queries to be executed over complex 3D geometric models. Based on this key parallel, it is entirely conceivable that RT cores can potentially be utilised for Monte Carlo particle transport. The second major piece of this work investigates that concept, by attempting to exploit RT cores to accelerate OpenMC using triangle meshes, hypothesising that the particle transport workload can achieve some worthwhile fraction of the 10x speedup obtained by graphical rendering applications.

1.3 AIMS AND OBJECTIVES

This project aims to port OpenMC to NVIDIA GPUs to improve on-node performance on both triangle mesh and CSG models, and subsequently exploit the hardware-accelerated ray tracing available on the Turing architecture to reach previously unattainable speeds on triangle meshes. In order to achieve this, the following objectives will be fulfilled:

- **Implement new GPU ports** supporting both triangle mesh and CSG models.
- **Evaluate performance relative to existing CPU implementations** using test models of realistic scale.
- **Extend the GPU port to support RT core acceleration** and re-evaluate performance relative to the original port.
- **Profile the implementation** in order to understand exactly how RT cores work in terms of computational resource usage.
- **Provide insight into RT cores** to facilitate future research in other scientific domains.

Furthermore, since RT cores are a very recent development, this work aims to provide a rigorous study of their performance capabilities from a general perspective by fulfilling the following additional objective:

- **Develop an RT core benchmarking tool** to evaluate them on a graphical rendering workload.

If successful, this project will directly benefit the UKAEA and other institutions undertaking research into GPU-based Monte Carlo particle transport, and will provide valuable insights for researchers in other scientific domain areas who wish to make use of RT core acceleration in their applications.

To the best of our knowledge, this work comprises the first study into the use of RT cores for scientific applications. As such, a full conference paper based on this work has been submitted for publication at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19) and is expected to be accepted following submission of this dissertation.

1.4 STRUCTURE OF THIS THESIS

This thesis is divided into five main parts:

PART I: BACKGROUND introduces Monte Carlo particle transport from a high-performance computing perspective, and introduces the concepts leveraged from ray tracing and computer graphics required to understand hardware-accelerated ray tracing and how it can be used.

PART II: RT CORE BENCHMARKING is a stand-alone analysis of RT core performance using a graphical rendering benchmark. The benchmark is developed as part of this work specifically to gain an accurate and rigorous understanding of RT core performance.

PART III: DESIGN AND IMPLEMENTATION describes the design and implementation of the OpenMC GPU port, and outlines the experimental methodology used to gather performance results.

PART IV: RESULTS, ANALYSIS AND CONCLUSIONS presents performance results of the OpenMC GPU port, including comparisons to the original CPU implementation, showing how RT cores affect performance.

PART V: APPENDIX lists full results of all experiments performed in this work, as well as relevant code samples. A copy of the conference paper which has been submitted for publication at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19) is also included.

Part I

BACKGROUND

2

MONTE CARLO PARTICLE TRANSPORT

Monte Carlo methods employ probabilistic sampling to obtain approximate solutions to systems of equations, relying on the central limit theorem to improve accuracy as sample size increases. Particle transport is a specific case describing the physics of particles travelling through some space of materials, and has application in diverse areas of science such as fission and fusion reactor design, radiography, and accelerator design [5, 7] with new application areas constantly being developed.

However, the method is computationally intensive and slow, which limits its practical usefulness and prevents it from being usable in certain desirable scenarios such as performing on-demand simulations in radiotherapy clinics [15]. This is partly due to the large number of particles which must be simulated to achieve the required degree of accuracy, and partly due to certain computational characteristics of the underlying algorithm that make it challenging to fully utilise modern computing resources. As modern HPC architectures approach exascale levels, researchers continue to search for ways to improve parallel efficiency of the algorithm to overcome these challenges.

2.1 HIGH-PERFORMANCE COMPUTING (HPC)

A variety of computational factors have been cited as general HPC characteristics which affect the parallel efficiency of Monte Carlo particle transport. These include load balancing [31], SIMD utilisation [11], random memory access patterns [22, 43] and atomic tallying performance [7] which have all been extensively studied in the literature. A number of previous studies have successfully proven that Monte Carlo particle transport is well suited to GPU-based parallelism [3, 13, 15] and can achieve significantly higher on-node performance than CPU-based codes. This is primarily due to the ability of massively parallel architectures to hide memory access latency through scheduling.

This project will carefully consider these parallel efficiency factors, specifically focussing on the use of GPU parallelism. However, we will also focus heavily on the less well studied concern of *geometric algorithm efficiency*. This relates to how the underlying geometric model is represented and accessed during simulation. When tracking particles travelling through complex geometric models, a large number of queries must be performed on those models, comprising a major proportion of overall runtime [3, 4, 36]. This becomes a critical factor in production-grade systems which simulate large scale models such as the ITER fusion reactor [40]. Optimising this aspect of the Monte Carlo particle transport algorithm has the potential to yield significant and wide-ranging benefits. The following sections introduce the concepts behind geometric algorithms, and how concepts from the field of *ray tracing* can be leveraged.

2.2 GEOMETRIC ALGORITHMS

All production-grade Monte Carlo particle transport codes support complex, arbitrary 3D geometric models as the substrate on which to simulate particle interactions. As a particle such as a neutron moves through its environment, it can move between materials of different densities and temperatures, such as from a fuel rod into water or concrete shielding. A major part of the simulation process involves knowing exactly where a particle is within the model, computing where it will be in the next timestep given its velocity, and determining which objects it will intersect with along its trajectory in order to record certain physical properties such as energy deposition. This process depends heavily on how the model is represented in software.

2.2.1 *Geometric Representation*

There are many ways to represent geometric information. The most prevalent representations in the Monte Carlo particle transport world are *constructive solid geometry (CSG)* and *triangle meshes*.

2.2.1.1 *Constructive Solid Geometry (CSG)*

The CSG representation essentially uses a tree of boolean operators (such as *intersection* and *union* applied to primitive objects (such as planes, spheres and cubes) to build complex

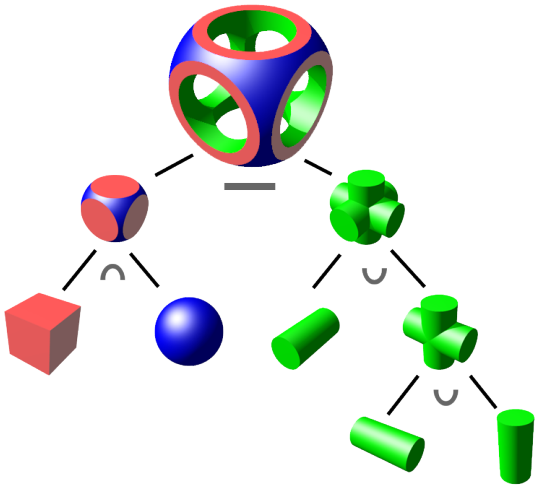


Figure 2.1: CSG models are created using boolean operators applied hierarchically to simple primitive objects.

objects that closely approximate reality. Figure 2.1 shows a basic illustration of this process. To track particles through CSG models, intersections between particles and objects are calculated using basic geometric functions on nodes in the tree, based on the primitive type of the shape node and its parameters (such as the radius of a sphere). CSG representations are relatively lightweight, since only the primitive shape parameters and operator tree need to be stored. Many institutions create model geometries using computer-aided design (CAD) tools, which often use CSG representations due to their accuracy. However, it is often time consuming to create complex models using CSG, and certain surfaces such as toroids cannot be represented. Furthermore, since there is no standard CSG file format, engineers are often forced to duplicate the model using the proprietary input formats of each Monte Carlo particle transport code.

2.2.1.2 *Triangle Meshes*

Triangle meshes are made up of tessellations of small triangles (represented as vertices in 3D space) which produce approximations to objects. Figure 2.2 shows an example of a simple triangle mesh model. Triangle meshes are generally less accurate than CSG

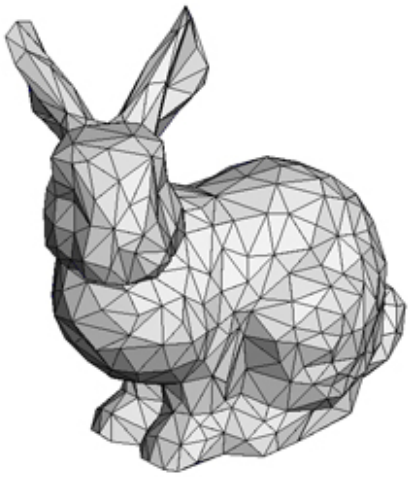


Figure 2.2: A simple triangle mesh model. This particular model is the classic Stanford Bunny, widely used as a reference in computer graphics.

models, although their accuracy can be improved by using a more highly faceted mesh (i.e. smaller triangles). This can result in relatively high storage costs if precise models are required. Furthermore, particle tracking through triangle meshes can be expensive, requiring potentially many triangles to be tested before finding the exact intersection point. However, it is often easier to create complex topological surfaces using triangle meshes compared to CSG models. There are several widely supported industry-standard formats

for storing triangle mesh data, which can potentially reduce the engineering burden of defining models multiple times in the specific input format of each code [2, 42]. It is for these reasons that triangle meshes are gaining interest in the Monte Carlo particle transport domain.

2.2.2 *Ray Tracing*

The technique of *ray tracing* is generally used in computer graphics to trace the movement of photons emanating from a light source, interacting with a scene, and eventually terminating in a virtual camera lens. This allows photorealistic 2D renderings of 3D models to be created, provided enough rays can be cast.

Going back to particle transport, a similar technique can be used, but for a different purpose. The particle transport algorithm dictates exactly where each particle is, how fast it is travelling, and in which direction. At each time step, the algorithm needs to check which material the particle lies within, and whether it will have transitioned into a different material at the next time step. To do this, it needs to query the geometry twice: once at the location of the particle to find out its current material, and once using the velocity of the particle to calculate the closest material transition (or "intersection" point).

Particle simulation can be formulated using ray tracing to perform these queries. Figure 2.3 shows a particle (or *ray*) can be cast along a straight trajectory from a point in the plane through a two dimensional object to determine whether the point is inside the object. If the total number of intersections is *odd* after the ray is extended to infinity, the

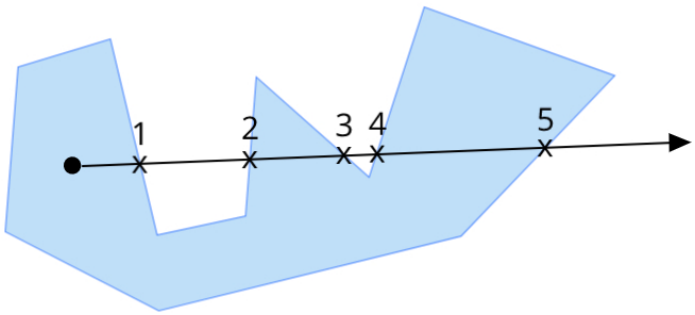


Figure 2.3: Using ray tracing to solve the point-in-polygon problem.

point must lie within the object. If the intersection count is *even*, the point is outside the object. The closest intersection point then simply becomes the first intersection along the trajectory. This method is trivially extended to 3 dimensions, and can be used with both CSG and triangle mesh geometries.

2.2.3 *Acceleration Structures*

A naïve way to determine the exact point at which a particle intersects with an object is to simply query every surface on the model. For large and complex geometries, this becomes prohibitively expensive. The graphics industry has dedicated a significant amount of time into optimising this kind of operation through the development of *acceleration structures*, which can dramatically reduce the cost of finding each intersection by quickly eliminating large numbers of surfaces from the search space.

Acceleration structures divide the model geometry into progressively smaller sub-regions, surrounding each with a bounding box. It is these bounding boxes which are then tested for intersection in a binary tree style search, massively reducing the number of objects or surfaces that need to be tested. The most prevalent types of acceleration structure are Bounding Volume Hierarchy (BVH) trees [19], Octrees [20] and Kd-trees [12], each of which contains tradeoffs between lookup performance, build time and runtime memory overhead. Figure 2.4 shows how a simple BVH tree is constructed. This work will

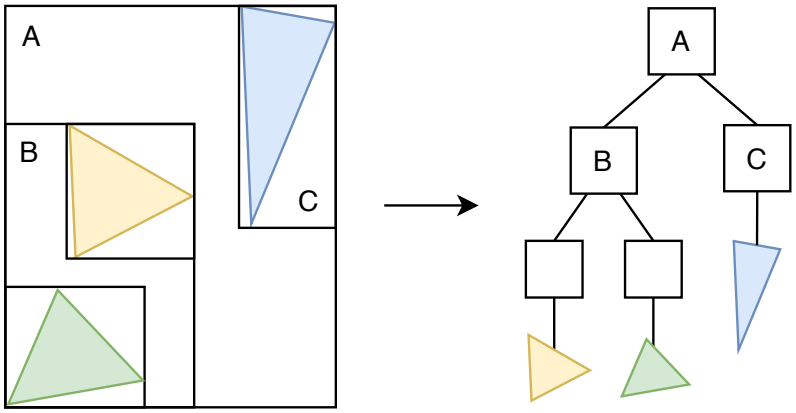


Figure 2.4: Construction of a BVH tree by recursively organising bounding boxes of decreasing size.

primarily focus on BVH trees, since they are currently a preferred approach in many fields which make use of them, as we will discover in the following sections.

2.3 RT CORES, OPTIX AND THE TURING ARCHITECTURE

The "holy grail" of computer graphics has always been *real-rime ray tracing*. If enough rays can be traced fast enough, fully photorealistic scenes can be rendered in real time. While not yet possible, NVIDIA's latest GPU architecture, codenamed Turing (described in Chapter 3), is specifically designed to be a step in this direction. It contains specialised

Ray Tracing (RT) cores, designed to offload ray-surface intersection tests and traversal of acceleration structures onto dedicated hardware, dramatically increasing ray tracing performance. NVIDIA advertises up to 10x speedup for ray tracing on graphical rendering workloads. However, this comes with two key caveats. Firstly, only *triangle mesh* models and *BVH trees* are eligible for hardware offloading. Secondly, in order to use the RT cores, a special library called *NVIDIA OptiX* (described in Chapter 4) must be used, which has major ramifications in terms of implementation architecture.

2.4 RELATED WORK

The promise of up to 10x speedup for ray tracing on RT cores is certainly an enticing prospect for Monte Carlo particle transport, despite the need to use the OptiX library. To explore the possibilities, a suitable code was sought which could be modified to make use of RT cores, and which could be representative in terms of real-world scale and functionality, in order to obtain realistic comparisons. Several other studies have attempted to use ray tracing techniques for particle-based applications, which we briefly review here in order to motivate the choice of code to study.

There have been a mixture of approaches focussing on both CPUs and GPUs, exploring different types of geometric representations and acceleration structures. In 2017, Bergmann et. al. presented results for the WARP code, a GPU-based Monte Carlo neutron transport code which uses OptiX to define CSG models and to perform intersection tests accelerated with BVH trees [3, 4]. Bergmann's code compared well with existing non-ray-traced CPU implementations. However, it does not use triangle mesh models, thus making it unsuitable for RT core acceleration. We did not choose to study the WARP code further, since it is not considered production-grade. Nevertheless, Bergmann's work sets important precedents for the use of both ray tracing and the OptiX library for Monte Carlo particle transport from which we can learn.

In 2018, Thomson claims to have implemented the first GPU-based astrophysics simulation code to use a ray-traced triangle mesh geometry accelerated with a BVH tree as part of the TARANIS code [39]. Thomson presented excellent performance results for simple test cases compared to existing non-ray-traced CPU-based astrophysics codes, although his implementation did not scale to more realistic test cases. This appears to be due to complexities with ionised particle behaviour which can occur with astrophysics simulations, which fortunately do not appear in most Monte Carlo particle transport simulation as commonly only neutral particles (such as neutrons or photons) are simulated. Despite being in quite a different algorithmic category, much insight can be drawn from Thomson's work, namely the further important precedents which are set for the use of both triangle mesh geometries and BVH trees for a particle-based code. However, the code itself is not available for direct study.

In 2010, Wilson et. al. published their work on the DAGMC toolkit [42], a generic CPU-based triangle mesh geometry traversal code that can be plugged in to a number of production-grade Monte Carlo particle transport codes such as MCNP [25], OpenMC [34] and SERPENT [40]. DAGMC replaces the native geometry representation with its own ray-traced triangle mesh geometry based on the MOAB library [38] accelerated with a BVH tree. Wilson's work was primarily focussed on conversion of CAD models to implementation-agnostic triangle meshes in an attempt to reduce the amount of effort required to port models between different codes. Initial performance results on complex models were adequate, although subsequent work by Shriwise in 2018 significantly improved performance on the MCNP code through the use of a more efficient BVH implementation [35]. It is important to note that although DAGMC runs only on CPUs, it proves the feasibility of using ray tracing over triangle mesh geometries with production-grade Monte Carlo particle transport codes.

2.5 OPENMC

OpenMC is a CPU-based Monte Carlo particle transport simulation code focussed on neutron criticality calculations, recently developed in the Computational Reactor Physics Group at MIT [33]. OpenMC natively uses CSG representations, and does not use ray tracing for geometry traversal (although the DAGMC plugin for OpenMC allows ray tracing over triangle meshes). OpenMC has been well studied from a performance perspective [37] and uses OpenMP for on-node parallelism and MPI for inter-node parallelism. However, it has not yet been ported to make use of GPU parallelism. Several other Monte Carlo particle transport codes have gained significant performance improvements from GPUs recently [15], [3], [13] therefore it is reasonable to assume that OpenMC may also benefit in similar ways.

OpenMC has been selected as the candidate for this study, as it is a freely available open-source implementation, unlike many other codes such as MCNP and SERPENT which are export restricted due to their sensitive links to nuclear research. OpenMC will be ported to the GPU, with support added for ray tracing over triangle meshes using the OptiX library. Following that, the implementation will be extended to support RT core acceleration.

2.6 SUMMARY

Existing work has proven that ray tracing techniques can improve the performance of particle-based scientific codes, particularly on GPUs, and that triangle mesh geometries and BVH trees are both feasible approaches. However, there is a clear opportunity to extend this work to utilise the hardware-accelerated ray tracing capabilities of the Turing

GPU architecture. To the best of our knowledge, there are currently no published studies investigating this opportunity.

This project seeks to address that gap by first porting OpenMC to the GPU, using the OptiX library to replace the native geometry representation with a ray-traced triangle mesh. Following that, we attempt to exploit the RT cores on a Turing-class GPU to gain even higher performance.

3 THE TURING GPU ARCHITECTURE

In August 2018, NVIDIA announced their eighth-generation GPU architecture, codenamed Turing [26]. The Turing architecture introduces a number of improvements over previous generations in terms of memory bandwidth, cache bandwidth, and latency reduction. It also includes a brand new type of fixed-function hardware unit called Ray Tracing (RT) cores, which are designed specifically for acceleration of ray tracing operations. This chapter aims to give a high-level overview of the generational improvements of Turing, before going into greater depth on RT cores and their capabilities.

3.1 COMPARISON TO PREVIOUS GENERATIONS

Table 3.1 compares the flagship Turing-class RTX 2080 Ti GPU with the GTX 1080 Ti, its sixth-generation Pascal-class equivalent. NVIDIA does not currently offer a GPU in the same range using the seventh-generation Volta architecture, so we also include comparisons of the TITAN range for posterity, which is available for all three generations.

Device	GTX 1080 Ti	RTX 2080 Ti	TITAN Xp	TITAN V	TITAN RTX
Architecture	Pascal	Turing	Pascal	Volta	Turing
SMs	28	68	30	80	72
CUDA cores/SM	128	64	128	64	64
Tensor cores/SM	-	8	-	8	8
RT cores	-	68	-	-	72
GPU Clock	1481 MHz	1350 MHz	1405 MHz	1200 MHz	1350 MHz
Memory	11GB GDDR5X	11GB GDDR6	12GB GDDR5X	12GB HBM2	24GB GDDR6
Memory Bandwidth	484.4 GB/s	616.0 GB/s	547.7 GB/s	651.3 GB/s	672.0 GB/s
L1 cache size/SM	48KB	64KB	48KB	96KB	64KB
L2 cache size	2.75MB	5.5MB	3MB	4.5MB	6MB
Giga Rays	1.1 GR/s	10+ GR/s	1.1 GR/s	1.1 GR/s	10+ GR/s
FP32 performance:	11.34 TFLOPS	13.45 TFLOPS	12.15 TFLOPS	14.90 TFLOPS	16.31 TFLOPS
FP64 performance:	354.4 GFLOPS	420.2 GFLOPS	379.7 GFLOPS	7.450 TFLOPS	509.8 GFLOPS

Table 3.1: Comparison of Pascal, Volta and Turing GPU architectures.

On paper, the Turing architecture provides notably higher memory bandwidth than both previous generations, and departs from HBM2 memory in favour of GDDR6. It also includes a larger L2 cache than has previously been seen, which should theoretically

provide advantages for a number of different workloads. In addition to these, the Turing architecture also contains several other generational improvements [26]:

- A new data path for integer operations that executes concurrently with floating-point operations;
- A new unified SM memory path which supposedly means twice as much bandwidth and twice as much L1 cache capacity;
- New reduced-precision INT8 and INT4 modes for Tensor cores.

An in-depth look at several of these features, as well as other computational characteristics of the Turing architecture, can be found in the microbenchmarking work of Jia et. al. [17]. As we are primarily concerned with ray tracing performance in this research, we now move on to examining RT cores.

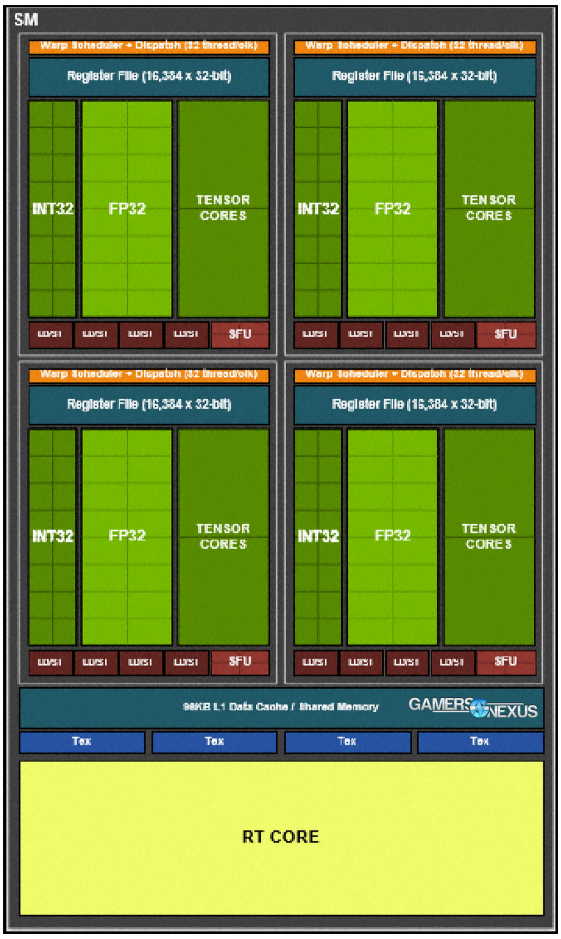


Figure 3.1: Block diagram of a Turing SM.

3.2 RT CORES

Triangle meshes are the prevalent representation in the graphics world, and the Turing architecture has been designed to provide specific support for them. Arguably one of the most anticipated features of the architecture, RT cores offload the critical part of a ray tracing loop; traversal of acceleration structures (specifically BVH trees) and ray-triangle intersection tests [9]. Each streaming multiprocessor (SM) on the GPU has access to its own RT core to which it can issue "ray probe" requests. Figure 3.1 shows a block diagram of a single Turing SM. The RTX 2080 Ti contains 72 of these units. Each RT core is made up of a triangle intersection unit and a BVH traversal unit, and is supposedly able to cache triangle vertex and BVH tree data in dedicated L0 caches to provide data locality for the two units, although the exact layout and functionality of the cores is not publicly known. The two units in the RT core perform the ray probe asynchronously, writing the result back to an

SM register once complete. Consequently, the SM is able to perform other work while the ray tracing is happening, saving potentially thousands of instruction cycles. Figure 3.2 illustrates the difference between this new approach to ray tracing compared with how previous generation GPU architectures emulate the process in software.

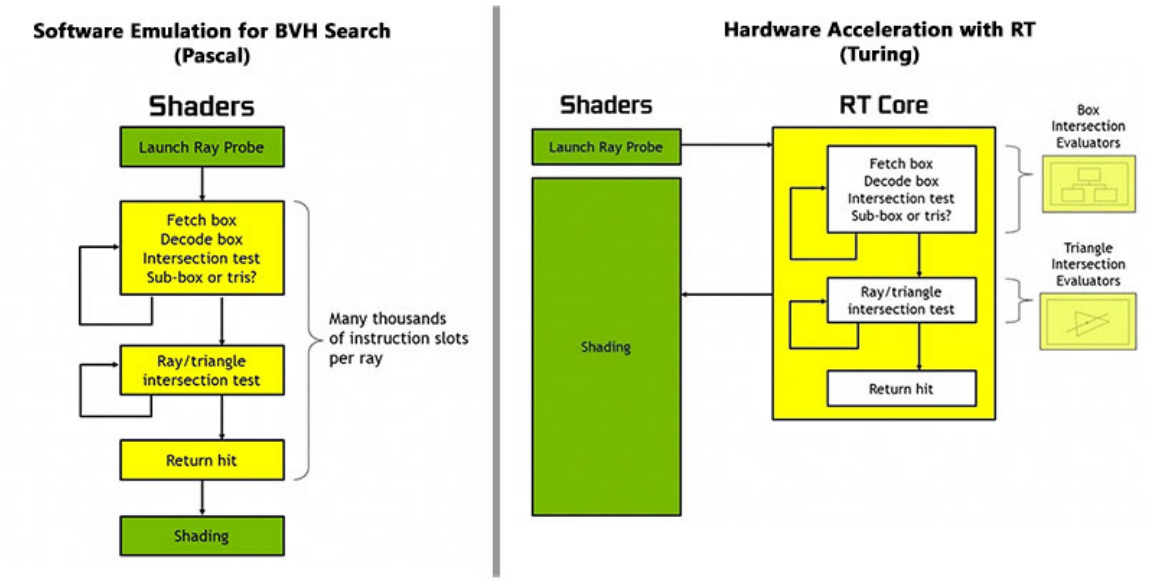


Figure 3.2: Emulating ray tracing in software (left) versus accelerating in hardware via RT cores.

3.2.1 *Programmability*

RT cores are not the first example of NVIDIA adding fixed-function hardware to its GPUs. The Volta architecture introduced Tensor cores [21], designed specifically to accelerate matrix multiplications as used heavily in deep learning. This is perhaps evidence of a turning point in GPU design, as memory bandwidth and general-purpose compute capability approach saturation.

However, while Tensor cores are directly programmable by the user (via the WMMA API), NVIDIA does not expose any method for directly accessing RT cores. They are only accessible indirectly, through one of three closed source libraries: Microsoft's DXR [24], Khronos Group's Vulkan [8] and NVIDIA's OptiX [29]. Each of these libraries handle RT core operations under the hood, on behalf of the user. The programming models prescribed by these libraries are undeniably easy to use and abstract away many of the complexities that come with ray tracing algorithms, albeit at the expense of being able to leverage RT cores for custom tasks. The inaccessibility of RT cores makes them difficult to directly

examine, which means that some amount of speculation is required to explain the precise reasons for their performance capabilities.

Since NVIDIA allows Microsoft and the Khronos Group to interface with RT cores through their libraries, it suggests that it may not be impossible to access them directly through reverse engineering, however such an exercise is out of scope of this research. It also suggests that future revisions of the RT core specification may be opened up further, once the technology reaches a more mature state. However, it does appear that NVIDIA have consciously chosen to restrict RT core usage to this particular use case due to the dominance of BVH trees and triangle geometries in graphics applications [1].

Chapter 4 introduces the OptiX API and its programming model, which will be the primary mechanism used in this work to leverage RT cores, due to its maturity and alignment with the familiar CUDA ecosystem.

3.3 SUMMARY

The Turing architecture contains multiple generational improvements over the Pascal and Volta architectures, and RT cores are a promising new addition which this research is keenly interested in leveraging for Monte Carlo Particle transport. Existing RT core benchmarks claim up to 10x speedup on pure graphical rendering workloads, given ideal conditions [28] [26]. These claims are investigated in Part II, where we develop our own benchmark to test the raw ray tracing performance of RT cores on a graphical rendering workload. Following that, in Part III we design and implement a GPU-based port of OpenMC which is ultimately capable of exploiting RT cores.

4

OPTIX

The OptiX library is a closed-source general purpose ray tracing library for NVIDIA GPUs [29]. OptiX is a high-level layer on top of CUDA, consisting of host-side and device-side APIs for defining geometric structures and launching ray tracing kernels. OptiX was originally designed with graphics applications in mind, but is generic enough to support any kind of ray tracing application, based on the observation that most ray tracing algorithms can be implemented using only a small set of operations [29]. It contains highly optimised implementations of BVH tree and Kd-tree build and traversal algorithms, and supports arbitrary 3D triangle mesh geometries as well as constructive solid geometries. The OptiX library has existed for more than five years, and has gone through a number of design iterations. The latest version of OptiX (version 6.0) includes support for hardware accelerated ray tracing via RT cores through the new so-called RTX execution strategy.

4.1 STRUCTURE OF AN OPTIX APPLICATION

OptiX is highly suited to graphical rendering applications in terms of API design, although it presents itself as being flexible enough to handle other types of application. This is true to an extent, provided the application is willing and able to adhere to the constraints of the API in terms of architectural design and flexibility. The OptiX programming model is a rather different experience for developers who are used to writing vanilla CUDA or OpenCL code. This section illustrates the basic structure of an OptiX application by walking through a simplified example of rendering a very basic 3D model to a 2D image.

An OptiX application essentially consists of two parts. Firstly, the *device-side code* comprises a set of user programs written in CUDA C which run on the GPU, which process a single ray (or particle) at a time and perform the actual computational work for that ray, writing result data to one or more output buffers. Secondly, the *host-side code* comprises a set of API calls which run on the CPU to initialise the scene geometry, load and assign various device programs, and handle movement of device buffers and variables to/from the device during execution.

4.1.1 *Device Code*

There are several device program types that can be provided by the user. These include *ray generation*, *closest hit*, *any hit*, *miss*, *intersection* and *bounding box* programs. The first two program types are mandatory for all OptiX applications, while the remaining types may be omitted depending on the particular application. The device programs are provided to

the OptiX host API as PTX strings, which are then dynamically compiled on-the-fly into CUDA object code by the OptiX runtime and weaved into a single "mega kernel". OptiX then handles scheduling of kernel launches internally, automatically balancing load across the GPU.

4.1.1.1 *Ray Generation Programs*

The *ray generation* program is the entry point to a ray tracing algorithm, and is the point from which rays are fired. When an OptiX context is launched, millions of instances of the ray generation program are created and scheduled on the GPU in an intelligent fashion, each one responsible for a single ray. Listing B.1 shows a basic example of an OptiX ray generation program which simply fires a ray from the origin in a fixed direction based on its launch index and writes the result to an output buffer.

The ray is allocated its own data payload containing arbitrary user data, which is passed to the synchronous `rtTrace` function. Device programs downstream in the ray tracing pipeline modify the payload, which is ready to copy to the output buffer once `rtTrace` returns. In this case, the payload result is interpreted directly as an RGB colour. Note the `rtBuffer` and `rtDeclareVariable` definitions; these must match the buffer names set on the OptiX context in the host code. The `launch_index` and `launch_dim` variables are made available automatically by the OptiX runtime, similar to the `blockDim` and `threadIdx` variables in a regular CUDA program.

4.1.1.2 *Closest Hit Programs*

The *closest hit* program is called when the closest geometric surface intersection is found after the ray has been fired, and is generally used to perform some calculation (such as determining a colour value) and store the result in the ray payload. Listing B.2 shows a very basic closest hit program that simply uses the *normal* value of the surface that has been hit and stores it in the payload belonging to the ray. This is known as a "normal" shader.

4.1.1.3 *Any Hit Programs*

The *any hit* program is called for every geometric surface intersection, not just the closest one. Common use cases for the any hit program are terminating rays in shadow, or accumulating values at each surface intersection. For this simple example, the any hit program is not required.

4.1.1.4 *Miss Programs*

The *miss* program is called when a ray does not intersect with any object i.e. its trajectory extends to infinity without hitting the model. Miss programs are commonly used to

perform tasks such as setting a constant background colour, or looking up a background colour from an environment map. Listing B.3 shows an example of the former.

4.1.1.5 *Intersection and Bounding Box Programs*

The *intersection program* is assigned on a per-object basis, and is responsible for calculating whether a ray at a given position and direction intersects with a particular object, and at what location. Intersection programs are highly object-specific; intersection testing on a sphere will look very different to a cube or a triangle, for example. OptiX supports arbitrary geometries, provided the user can implement intersection and bounding box programs for them. Listing B.4 shows what an intersection program for a triangle mesh might look like.

The *bounding box* program is also assigned per-object, and is called at context initialisation when the acceleration structure (e.g. BVH tree or Kd-tree) is being constructed. Bounding box programs are responsible for calculating the coordinates of a minimal bounding box for the object, which will be recorded in the acceleration structure for later use. Listing B.5 shows how a bounding box program might look for a single triangle.

4.2 RTX MODE

Since version 6.0, OptiX supports RT core acceleration through a new setting called *RTX mode*. This in fact refers to two separate concepts:

- A new `GeometryTriangles` API, which automatically offloads intersection and bounding box calculations to RT cores (if the device contains them, and as long as the intersection and bounding box programs are not overridden). Only triangle geometries are supported.
- A new optimised kernel compilation and execution pipeline, which can be used independently of the `GeometryTriangles` API and provides faster execution regardless of GPU generation or geometry type.

Note that acceleration structure traversal will only be offloaded to RT cores if a BVH tree is used (using other structures will fall back to a software traversal implementation). [27]. There are also options to set custom intersection and bounding box programs, although reasonable implementations of these are provided by default. Note that for RT core acceleration, the default implementations must be used. RT core acceleration is enabled in OptiX via the `RT_GLOBAL_ATTRIBUTE_ENABLE_RTX` setting.

4.2.1 *Host Code*

Listing B.6 shows what the host code might look like to configure and render our example scene on the device. There are a number of important steps:

- The scene geometry is created and copied into device buffers;
- The device-side programs are loaded as PTX strings and assigned to their respective OptiX API objects;
- The acceleration structure is specified;
- Output buffers and other variables are created on the device;
- The ray tracing kernel is launched with a particular size and dimensionality;
- The output buffers are copied back to the host, and in this case, interpreted as an image.

The experienced CUDA or OpenCL programmer will notice the familiar yet distinct structure of the host code, but may also initially be puzzled by the lack of a block size argument to the `context->launch()` method. Users can specify 1D, 2D or 3D launch dimensions depending on the problem, but these does not necessarily correspond to grid size, and block size dimensions are not specifiable at all. The OptiX runtime decides how to decompose the problem into blocks and threads itself, and is even able to schedule and balance load across multiple GPUs.

4.2.1.1 *GeometryTriangles API*

Listing B.7 shows an example usage of the `GeometryTriangles` API compared with the standard `Geometry` API, and how the former is conditionally activated based on the OptiX version in use, otherwise falling back to the latter. As previously mentioned, to qualify for RT core acceleration, the `GeometryTriangles` API must be used.

4.3 BUILD AND COMPILATION

OptiX requires that device programs be passed as PTX strings within the host code. These PTX strings are then compiled at runtime by the OptiX engine and instrumented into a single "mega kernel". This is necessary for the library to wire everything up properly behind the scenes, but it has the significant drawback that attaching a debugger to a kernel is no longer possible due to the loss of debug information. Another drawback to this approach is that OptiX will recompile the entire kernel at runtime if one of a number of conditions is activated, such as changing the value of a top-level variable. These

recompilations can be potentially expensive, and so care must be taken to avoid them wherever possible.

4.4 INTEROPERABILITY WITH CUDA

While OptiX does support interoperability with regular CUDA buffers to some degree, support in version 6 is limited, and it is recommended to use the OptiX buffer abstraction methods. However, the upcoming OptiX version 7 is much more closely aligned with traditional CUDA methods, and in fact does away with OptiX buffers entirely, in favour of regular CUDA buffers. This is potentially advantageous for existing CUDA-based codes which may want to experiment with OptiX.

Part II

RT CORE BENCHMARKING

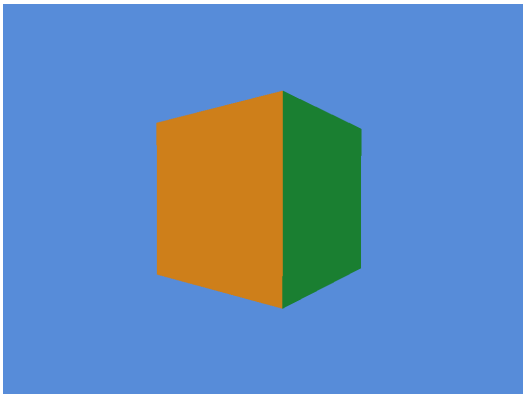
5 RT CORE BENCHMARK DESIGN

To the best of our knowledge, there are currently no published studies which specifically analyse the RT cores in the Turing GPU architecture in a rigorous manner. Jia et. al. used microbenchmarking to analyse a Turing-class GPU in 2019 [17], and discovered a number of interesting facts about the Turing architecture from a general perspective. However, RT cores were not specifically investigated. NVIDIA have included indicative results of the potential speedups achievable with RT core acceleration in their online marketing material [27], but the details of how these results were obtained have not been published. There are also existing 3rd-party benchmarks available online, such as Nyers [28] which have attempted to replicate and extend the official NVIDIA results. However, Nyers also does not describe his experimental methodology, and the two differ fairly significantly in terms of actual performance numbers. Therefore, this project will attempt to clarify the situation by replicating NVIDIA's original results in a structured way.

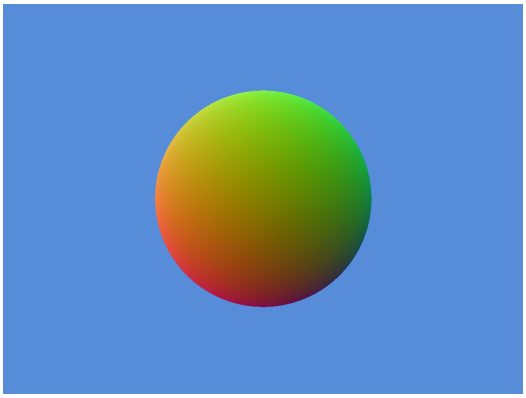
This chapter, together with the following chapter, comprises a rigorous stand-alone analysis into the performance of the hardware-accelerated ray tracing on the Turing architecture from a graphics rendering perspective. This enables realistic baseline performance results to be gathered, which can subsequently be used to build an expectation of the kind of speedups that are potentially achievable for Monte Carlo particle transport (and other scientific codes). This is achieved by developing a benchmarking tool specifically designed to test the raw performance of RT core acceleration. We first outline the design, implementation and experimental methodology of the RT core benchmark, before presenting performance results in Chapter 6.

5.1 MODELS

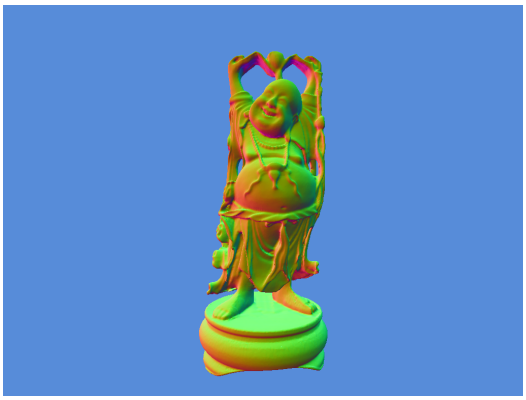
Both the official NVIDIA benchmarks and the Nyers benchmarks make use of a number of standard models, mostly taken from the McGuire Computer Graphics Archive [23], which are reasonably complex triangle meshes in Wavefront OBJ format. This work will also use three of the McGuire models, as well as two simpler models, spanning a wide range in terms of triangle count and topological complexity, with the intention of identifying different performance characteristics. The simplest model contains a trivial 12 triangles, while the largest contains over 7 million triangles. Most are single-cell watertight models, with the exception of the Hairball model which is a tightly packed weave of thin cells. Figure 5.1 shows 2D renderings of each of the five chosen models, as rendered by the RT core benchmark tool.



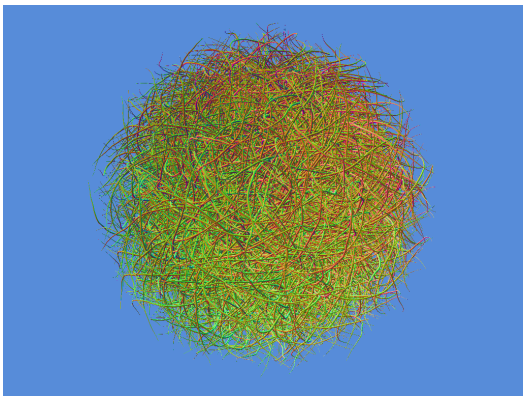
(a) Cube, 12 triangles



(b) Sphere, 89k triangles



(c) Happy Buddha, 1.1m triangles



(d) Hairball, 2.9m triangles

(e) Asian Dragon, 7.2m triangles

Figure 5.1: Renderings of the five 3D models produced by the RTX benchmark tool, with approximate triangle counts.

5.1.1 *Cube Model*

The simplest model of the set is a 5x5x5cm cube, where each face is faceted with two triangles, giving a total triangle count of 12. The Cube model should be trivially simple to render given its low triangle count, as there will be a maximum of two ray intersections to traverse the entire object irrespective of ray origin, since it is a single-volume solid object. The model structure should produce a BVH tree of low complexity.

5.1.2 *Sphere Model*

The Sphere model is a tessellated icosphere of radius 5cm, comprising 81,932 triangles. It should also be fairly simple to render given its regular structure and the fact that it is also a single-volume solid object. The generated BVH tree should also be of fairly low complexity.

5.1.3 *Happy Buddha Model*

The Happy Buddha model represents the middle ground of our model set in terms of triangle count and complexity, having 1,087,451 triangles and an irregular structure, measuring 17.5cm tall. Its BVH tree should be reasonably complex. However, it is still a single-volume solid object, meaning that the maximum number of ray intersections will be low (around three to four, depending on ray origin).

5.1.4 *Hairball Model*

The Hairball model comprises 2,880,012 triangles, which does not give it the highest triangle count of all our models; however, it is extremely topologically complex. It stands at 10cm tall, and is made up of many thin solid strands compacted tightly together. Its structure means that there are potentially a large number of ray intersections required to traverse the entire model, and that its generated BVH tree will also be highly complex.

5.1.5 *Asian Dragon Model*

The Asian Dragon model comprises 7,218,942 triangles, which is the highest count of all the models under study. It is also the largest model, measuring 200cm in width. Due to its sheer size, it is likely to be challenging to render in terms of memory requirements.

5.2 DEVICES

The Zoo testbed at the University of Bristol contains a Turing-class RTX 2080 Ti, which is classed as a consumer-grade GPU. There is currently no HPC-grade Turing GPU available, unlike the Pascal and Volta architectures (which have the HPC-grade P100 and V100 respectively). Therefore it makes sense to compare to another consumer-grade GPU. The Volta architecture is not currently available in a consumer-grade format, but the Pascal architecture is. The Zoo testbed also contains a Pascal-class GTX 1080 Ti, which is the same product tier as the RTX 2080 Ti. Therefore we will benchmark against these two GPUs, as they make for the most appropriate comparison.

5.3 IMPLEMENTATION

The benchmark has been implemented using the `optixMeshViewer` sample provided with the OptiX SDK as a baseline reference. The benchmark performs a fixed number of non-interactive static frame renders as fast as possible and measures the total amount of time taken (as opposed to launching an interactive window that is dynamically re-rendered as the user moves around the viewport with the mouse).

5.3.1 *Ray Generation*

Rays are launched in a two dimensional configuration, corresponding to one ray launched per pixel, in a prism shape emanating from an elevated viewpoint looking down on the 3D model. The launch dimensions are configurable, with the resolution of the resulting rendered image matching the launch dimensions.

5.3.2 *Shader*

The shader implementation is intentionally as simple as possible, in order to maximise the ratio of ray tracing work to other compute work, thereby exposing the raw performance of the ray tracer. Once a ray intersects with a particular triangle, its geometric normal is fetched from the normal buffer and directly interpreted as a colour in RGB space. This type of shader is known as a "normal" shader. The shader only considers primary rays, i.e. rays are terminated at the point of intersection and do not reflect or refract as secondary rays.

5.4 EXPERIMENTAL METHODOLOGY

As previously mentioned, neither the NVIDIA benchmarks nor the Nyers benchmarks state the launch dimensions (or "resolutions") which were used, therefore a strong scaling study has been designed to investigate the performance behaviour as the resolutions are scaled up. This should produce meaningful results which will subsequently allow solid conclusions to be drawn about the characteristics of RT core acceleration at different resolutions.

The scene viewpoint location (i.e. the origin from which rays are cast) is kept constant for each experiment, and the number of rendering iterations is maintained at 10^4. The launch dimensions (which directly correspond to output image resolution) are varied from 960x540 doubling each time up to 15360x8640, roughly corresponding to the range of a 540p SD image up to a 16k Ultra HD image. Each experiment is run five times, and the ray generation rate is taken as the average of the five runs. Each experiment is run with RTX mode enabled and disabled on both the Turing and Pascal GPUs. This is repeated for each model, thus allowing us to investigate the effects of varying both model complexity and launch dimensionality independently.

The main metric used to measure performance is *rays cast per second*, calculated as the total number of rays cast divided by the total runtime duration, measured in *GigaRays per second*.

5.4.1 *Hypotheses*

Based on the existing benchmarks, we can assume that the triangle count will have a measurable effect on performance. We hypothesise that as model complexity increases, performance will decrease as triangle intersection testing becomes the limiting factor. We also hypothesise that as launch dimensions are increased, there will be some sublinear scaling relationship with the number of rays cast per second.

5.5 SUMMARY

The benchmark described in this chapter has been designed to enable rigorous analysis of RT cores on a graphical rendering workload, using models chosen in order to represent a wide range in terms of triangle count and topological complexity. The models have been carefully chosen with the intention of uncovering a range of interesting computational characteristics of RT core acceleration.

6

RT CORE BENCHMARK RESULTS

This chapter presents and discusses the performance results obtained from the RT core benchmark experiments. We first consider the effect of model complexity on performance, before investigating the effect of varying the launch dimensionality.

6.1 RESULTS: MODEL COMPLEXITY

Figure 6.1 shows how each of the five models behave at a fixed launch resolution of 15360x8640 on a Turing-class RTX 2080 Ti GPU and a Pascal-class GTX 1080 Ti. Note that enabling RTX mode on the Pascal GPU is a valid configuration despite the lack of RT cores, as it will still use the improved OptiX execution pipeline.

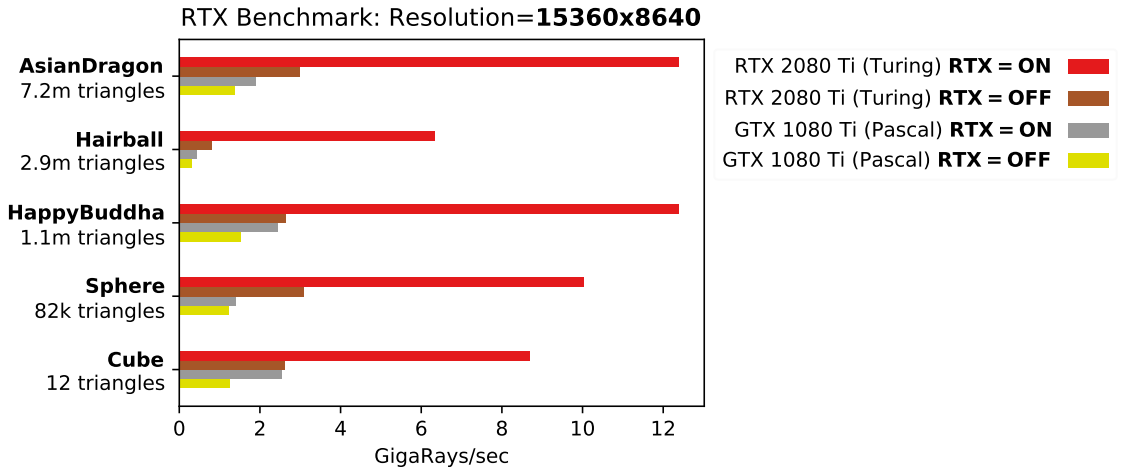


Figure 6.1: Results of the RTX-enabled graphics benchmark on each of the tested models and GPUs for a fixed launch resolution of 15360x8640. Higher is better.

Using RTX mode on the Turing GPU yields a 4.6x speedup on average at this resolution. The lowest speedup is 3.3x on the Cube model, and the highest is 11.8x on the Hairball model. A fraction of this speedup is attributable to the optimised execution pipeline in OptiX 6, which can be seen when comparing the two results on the Pascal GPU. The vast majority however is due to the RT cores themselves. The benchmark is able to produce over 12 GigaRays/sec for the Happy Buddha and Asian Dragon models, which corroborates NVIDIA's own benchmarks for these two models [26]. The result profile for these two

models appears quite similar, despite being almost an order of magnitude apart in terms of triangle count. This suggests that both are ideal workloads, and are probably reaching the optimal throughput of the RT cores.[1]

6.1.1 *Discussion*

It is clear to see that the number of triangles alone does not dictate performance. The Hairball model is clearly the heaviest workload, despite not being the largest model in terms of triangle count, and is 11.8x faster with RTX mode on the Turing GPU. This suggests that the higher topological complexity of the model is having a significant impact, most likely because its multi-volume topography results in a higher number of potential intersections along a single ray path, thereby requiring a greater number of vertex buffer lookups to complete the full traversal. This then becomes the dominant performance factor. Additionally, the generated BVH tree is likely to be deeper and therefore slower to search.

The reason why the Cube and Sphere models are slower than the larger Asian Dragon and Happy Buddha models under RT core acceleration is simply because ray-triangle intersection testing and BVH traversal operations comprise less of the overall runtime due to the simplicity of the models. As the models get larger, the overall runtime increases proportionally and the ray tracing operations become the dominant factor. The Hairball model shows the extreme case of this, where ray tracing takes such a large amount of effort that runtime increases to the detriment of ray casting throughput.

The Turing GPU also outperforms the Pascal GPU with RTX mode disabled. This simply suggests that the generational improvements of the Turing architecture are providing some benefit for this particular type of workload, such as larger L1 and L2 caches permitting better locality for vertex data.

6.2 RESULTS: LAUNCH DIMENSIONS

Figure 6.2 shows how the ray tracing throughput changes as the launch dimensions are varied for the Happy Buddha model. The other four models exhibit essentially the same behaviour, so have been omitted for brevity. The peak RT core speedup for this model is 5.5x on the Turing GPU. The best speedup obtained for any model was 15.6x on the Hairball model, which occurred at a resolution of 1920x1080.

6.2.1 *Discussion*

There is clearly a sublinear scaling profile as launch dimensions are increased, regardless of GPU architecture. This is most likely a result of the way that OptiX schedules kernel

1 It is perhaps not surprising that these models were used to produce the advertised performance numbers.

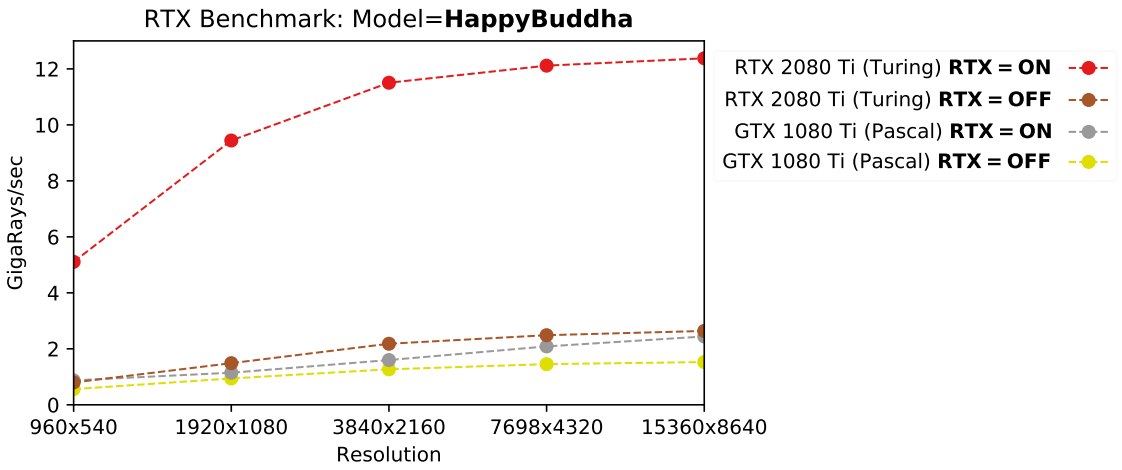


Figure 6.2: Results of the RTX-enabled graphics benchmark for the Happy Buddha model as the launch dimensions are increased. Higher is better.

launches and automatically balances load over the GPU, with larger launch sizes leading to more effective balancing and higher occupancy. It is interesting to note that the scaling profile with RT core acceleration follows the same shape, suggesting that it is being limited by the same resource, but that RT cores are providing a larger amount of the resource in question which results in the performance barrier being higher.

6.3 PROFILING

In order to gain a deeper understanding of how the GPU is being utilised, the NVIDIA Visual Profiler was used to profile the running benchmark. Unfortunately, at the time of writing, it is not possible to profile kernels running on Turing-class GPUs, due to support not yet having been added for Compute Capability 7.5. It is also not possible to profile with RTX mode enabled on any GPU, due to the new kernel mangling performed by OptiX leading to `nvprof` being unable to instrument the main kernel. Furthermore, OptiX makes things even more challenging to profile, since it combines all user kernels into one "mega kernel", resulting in debug symbols being unavailable. Nevertheless, some insight can still be gleaned by profiling on the Pascal GPU with RTX mode disabled, combined with a certain amount of speculation about what might be occurring on the RT cores.

Table 6.1 shows some key metrics obtained for each model at a launch resolution of 15360x8640 with RTX mode disabled on a GTX 1080 Ti. The following sections discuss these metrics in terms of parallel resource usage and efficiency.

Model	Cube	Sphere	Happy Buddha	Asian Dragon	Hairball
Achieved Occupancy (%)	56.2	56.2	56.2	56.2	56.2
Device Memory Throughput (GB/s)	310.5	235.5	254.5	254.33	88.7
Local Memory Overhead (%)	96.5	87.7	88.4	82.6	61.3
Memory Dependency Stalls (%)	79.39	67.2	67.1	75.12	63.32
Memory Utilisation (%)	65	65	55	55	75
Compute Utilisation (%)	23	38	35	28	55

Table 6.1: Selected profiling metrics obtained for the RT core benchmark at a launch resolution of 15360x8640 on a GTX 1080 Ti GPU with RTX mode disabled.

6.3.1 *Occupancy*

The RT core benchmark obtains a reasonably good occupancy of 56.2% for all models, using 54 registers per thread. An attempt was made to fix the number of registers at a lower level, which did not result in any significant performance improvement. This suggests that the OptiX scheduling mechanism is reasonably well optimised for this type of problem.

6.3.2 *Memory Bandwidth*

Performance is limited by memory bandwidth in all cases. It might be reasonable to initially guess that the smaller Cube and Sphere models would not be affected by memory bandwidth issues, since the models are easily small enough to fit entirely in L1/L2 cache. However, this is not the case when taking into account other factors such as output buffer writes, BVH traversal and internal bookkeeping done by OptiX. The combination of these factors leads to relatively poor device memory throughput and utilisation for all models. As the memory footprint of the models increase with complexity, bandwidth throughput deteriorates and local memory overhead decreases as more memory activity occurs at higher levels of the hierarchy.

6.3.3 *RT Cores*

The dramatic performance improvements when using RT cores can be explained partially in terms of memory bandwidth, and partially in terms of compute utilisation. Firstly, since the RT cores are able to cache triangle vertex and BVH tree data, they effectively alleviate some of the memory pressure caused by reading in this data from global memory.

Secondly, since the RT cores execute ray tracing probes concurrently, the SM is able to perform the shading work asynchronously.

6.4 SUMMARY

The results presented here have successfully managed to reproduce and corroborate the RT core performance numbers advertised by NVIDIA, and have found that there is a performance sweet spot in terms of model complexity and launch dimensionality. RT cores are capable of providing greater performance increases with highly topologically complex models. RT cores are indeed a powerful addition to the Turing architecture, and our findings serve as solid motivation for further investigating their potential use for Monte Carlo particle transport. To the best of our knowledge, this work constitutes the most rigorous study of RT cores yet to be published, and will provide useful information for future research into the topic area.

Part III

DESIGN AND IMPLEMENTATION

7 IMPLEMENTATION

This chapter describes the design and implementation processes undertaken during development of the OpenMC GPU port. OpenMC is a fully-featured production grade simulation code comprising over 100,000 lines of C++ code. Porting such a large and complex codebase to a completely new platform is a highly non-trivial task, therefore it is crucial to approach the implementation in a methodical and incremental way to improve the chances of success. The following sections chronologically outline the sequence of steps that were taken to create the GPU port.

7.1 UNDERSTANDING OPENMC

The first stage was to closely inspect the original codebase and documentation to understand how the OpenMC particle transport algorithm works on the CPU. OpenMC uses an *over-particles* parallelism approach, i.e. each CPU thread simulates a single particle from birth to death. There are three main code regions that are parallelised: the *source bank initialisation* loop, the main *particle transport* loop, and the *tally reduction/fission bank synchronisation* loops at the end of a generation. Listing 7.1 gives a simplified overview of the execution flow of the OpenMC algorithm in the default *eigenvalue* run mode.

7.1.1 *Source Bank Initialisation*

OpenMC supports multiple particle sources, each of which can have its own spatial and energy distributions depending on the scenario. For example, a small cubic high-energy source can be placed at the very centre of a geometry, while a large low-energy source is distributed over the entire problem space. The source bank initialisation loop (highlighted orange in Figure 7.1) is responsible for randomly sampling from these spatial and energy distributions, and storing the sampled particles in the `source_bank` buffer. By default, 16 CPU threads are spawned which concurrently sample particles until the desired number have been stored in the buffer.

7.1.2 *Particle Transport*

The main particle transport loop (highlighted pink in Figure 7.1) fetches a single particle from the source bank and performs the main physics simulation. Importantly, this includes the geometric calculations in order to know which cell the particle currently lies within, and the distance to the cell boundary given the current trajectory. Following that, the

```
initialise source_bank:
        (parallel) for i in num_particles:
                sample particle from distribution
                store particle in source_bank
begin simulation:
        foreach batch:
                foreach generation:
                        (parallel) for i in num_particles:
                                fetch particle from source_bank
                                transport particle:
                                        compute current_cell
                                        compute distance_to_boundary
                                        sample collision
                                        sample scattering
                                        sample absorption
                                        store tallies
                                        sample fission reactions
                                        store secondary particles in fission_bank
                        finalize generation:
                                (parallel) reduce tally buffers
                                calculate stats
                                (parallel) synchronise fission_bank
                                copy fission_bank to source_bank
                finalize_batch:
                        accumulate tallies
```

Listing 7.1: OpenMC eigenvalue pseudo-algorithm on the CPU, highlighting parallelised regions.

algorithm stochastically samples particle collisions, scattering, absorption, and accumulates various tallies. If a fission event occurs, secondary particles are generated and stored in the `fission_bank` buffer. By default, 16 CPU threads share the transport workload until each particle is terminated.

7.1.3 *Tally Reduction and Fission Bank Synchronisation*

The tally reduction loop (highlighted green in Figure 7.1) occurs at the end of a generation, which reduces the `threadprivate` tally buffers into a global buffer, from which end-of-generation statistics are calculated. Following that, the `threadprivate` fission banks are gathered into a master buffer, which is subsequently used as the source bank buffer in the next generation.

7.2 BUILDING AN MVP

Once the OpenMC algorithm was broadly understood, the process of designing an approach to parallelising these three regions on the GPU could begin. As a first step, a minimum viable product (MVP) was designed to validate the basic feasibility of using the OptiX API to create triangle mesh geometry on the GPU and having OpenMC use that geometry for traversal. An intentionally naïve but highly effective strategy was conceived to achieve this. The existing CPU code was simply modified at the point where the geometric calculations are invoked, to launch a GPU kernel to perform those calculations for *a single particle*. A ray is traced from the particle origin along its trajectory, and the intersection results are returned back to the CPU code. This is obviously an extremely inefficient use of GPU resources, but is an effective and direct route to validate the basic concept with minimal effort. Furthermore, this effort is not wasted, as a large proportion of the OptiX host code which sets up the geometry is reusable for later stages. At this point, the geometry itself is hard-coded to a basic nested cube structure.

OpenMC includes plotting capabilities, which can be used to generate slice plots of model geometry. Figure 7.1 shows a composite image from the ParaView tool, containing the initial nested cube geometry (the white wireframe lines) and a slice of a voxel plot

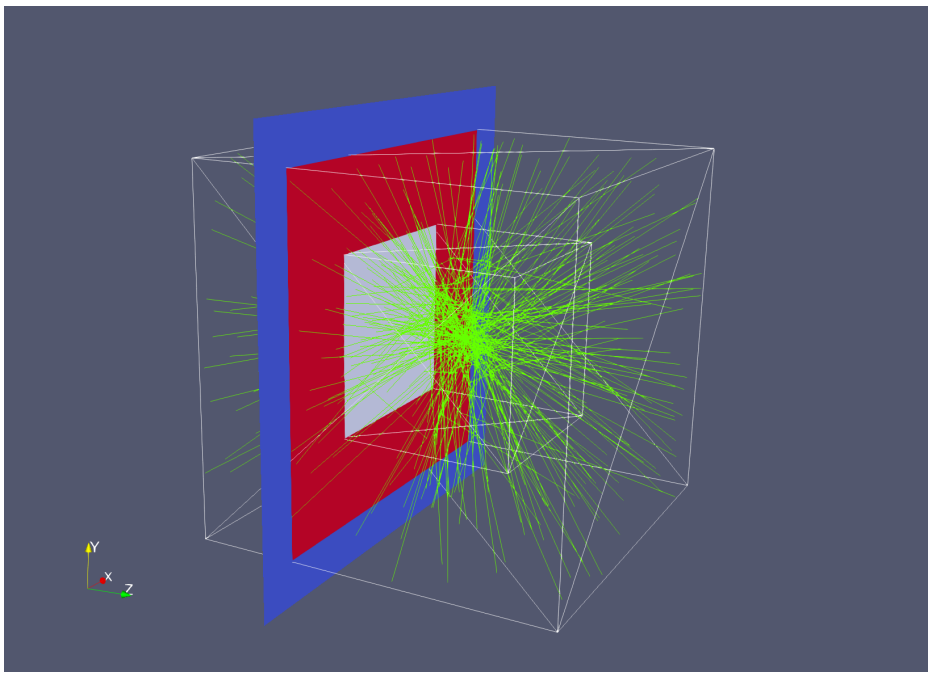


Figure 7.1: A simple nested cube geometry (white wireframe) overlaid with a generated OpenMC slice plot (blue/red/grey squares) and particle tracks (in green).

generated by OpenMC. This plot proves that the MVP is working correctly, as OpenMC uses the point-in-volume calculations (which are run on the GPU using the OptiX geometry) to colour each volume. The light blue region at the centre is the innermost cube, the red region is the space inside the outermost cube, and the dark blue region is the infinite space outside the model.

OpenMC also supports recording particle tracks (shown in green in Figure 7.1) which proves that our model geometry is configured correctly, as particles were configured to terminate once they reach the edge of the outermost cube.

7.3 PORTING THE MAIN TRANSPORT LOOP

Now that using OptiX-based geometry on the GPU to perform geometric calculations is proven to be feasible, we can begin to make proper use of GPU parallelism. Firstly, the main particle transport loop (shown in pink in Listing 7.1) was ported to CUDA and OptiX. This was by far the most time consuming and challenging aspect of this work, requiring many thousands of lines of code to be converted to the CUDA C language, and many thousands of lines of new code to be written using the OptiX API to configure model geometry and manage device buffer movement. The following section outlines some of the important decisions that were made and challenges that were faced during this process.

7.3.1 *Device Code*

The bulk of the main transport loop was implemented as an OptiX *ray generation* program, which broadly corresponds to a CUDA `__global__` function. This was initially done on a like-for-like basis, conserving all critical functionality. The regions where geometric calculations are performed were implemented using calls to the OptiX `rtTrace` function, firing rays according to the current particle velocity. These rays are then handled by an OptiX *closest hit* program which uses the point-in-polygon algorithm to calculate the current cell and the distance to the nearest boundary. This is supported by implementations of OptiX *intersection* and *bounding box* programs which perform the actual ray-triangle intersection tests in software. These implementations are reasonably long and complex, and are thus not shown here for brevity. However, Appendix B gives indicative samples of the implementation approach for the interested reader.

7.3.1.1 *Challenges*

As with any CUDA port, the programmer must manage the limitations of the platform in a way that ideally minimises the amount of code duplication between CPU and GPU versions. For newly written codes, this is often fairly simple to achieve with careful use of `#ifdef` guards, compilation flags, and restricted feature sets. However, for large pre-

existing codebases such as OpenMC which have not been ported to the GPU before, the situation becomes more difficult. OpenMC makes use of several C++ language features that are not supported in CUDA C, and therefore must be modified to run on the GPU. The two unsupported features which have the most impact on the GPU port are the use of *object orientation* and *C++ standard template library (STL)*.

OpenMC is exceptionally well written for a CPU-based scientific code, with great care having been taken to keep the codebase consistent and carefully engineered. This of course means an object-oriented approach has been followed. Unfortunately, it is not currently practical to use inheritance, virtual functions or polymorphism in CUDA C. Furthermore, it is recommended that all function calls in a CUDA kernel are explicitly `inlined` for performance. As a result, it was necessary to duplicate all class member functions as top-level functions, taking a context object as a parameter to those functions.

In terms of the C++ STL, all usage of `std::vector` was removed in favour of the OptiX `rtBuffer` mechanism. This is effectively syntactic sugar for a raw device pointer, with some extra useful functionality such as a `size()` function. Unfortunately, this required further code duplication, with each `class` needing a corresponding device copy with the `rtBuffer` types to enable compilation to succeed.

7.3.2 *Host Code*

On the host side, the bulk of the work lies in using the OptiX host API to define the model geometry, configure the various OptiX ray tracing programs, copy data buffers to and from the device, and launch the GPU kernels themselves. The model geometry definition code developed during the MVP build could fortunately be reused at this point, while still using the hard-coded nested cube geometry.

7.3.2.1 *Device Buffers*

There is a large amount of data that needs to be copied on to the device before the simulation begins. This includes the materials assigned to each cell in the model, individual nuclides which comprise those materials, and the nuclide cross section data tables corresponding to those nuclides. The native OpenMC code uses a nested object hierarchy to represent these structures, using `std::vector` instances. For this to work on the GPU, each one of these vectors needs to be converted to a top-level pointer array and copied into an OptiX `rtBuffer`, which remains on the device throughout the entire simulation. The hierarchy is reasonably complex, and results in around one thousand lines of code to configure correctly on the GPU (see `src/optix/optix_data.cpp` in the accompanying source code for reference). The `source_bank` and `fission_bank` buffers and the various global tally buffers are also configured as output buffers, which are subsequently copied back to the host at the end of each generation for further processing.

7.3.2.2 *Kernel Configuration and Launch*

OptiX does not allow specifying CUDA kernel block and grid sizes directly, but instead takes a one, two or three dimensional launch parameter and manages kernel launches internally. A one-dimensional launch parameter was chosen, despite the observation that a three-dimensional structure naturally maps on to the three-dimensional nature of the model space. This choice was made due to the fact that particles in a Monte Carlo simulation do not follow coherent trajectories, and therefore are not likely to receive any data locality benefits from using a three-dimensional structure. Furthermore, indexing strategies become much simpler in one dimension.

7.3.2.3 *Build and Configuration*

The OptiX API ships with a sample CMake configuration, which is fairly non-trivial. Fortunately, OpenMC also uses CMake as its build tool, so it was only necessary to combine these two configurations in order to include CUDA and OptiX into the OpenMC build chain. To enable the DAGMC plugin to be tested, it was also necessary to compile out-of-tree versions of DAGMC itself, as well as the MOAB and HDF5 libraries.

7.3.3 *Minimum Required Functionality*

Due to the relatively short timeframe of this project, it was understood that not all features of OpenMC will be supported by the initial GPU port. Any feature which does not lie on the critical *eigenvalue* simulation path can safely be scheduled for reimplementation at a later date. This guarantees that faithfully accurate simulations can be run, while minimising effort on developing features that are not due to be tested in this work. The set of omitted features includes: (i) Fixed-source simulation mode; (ii) Multiple particle sources; (iii) Periodic boundary conditions; (iv) Photon tracking support; (v) Lattice geometry support; (vi) Multiple nuclides per material; (vii) Custom mesh tallies.

The source bank initialisation loop (highlighted orange in Figure 7.1) was also parallelised, although it was not strictly necessary to do so.

7.4 SUPPORTING ARBITRARY GEOMETRIES

Once the bulk of the computational work was offloaded to the GPU, the next step was to move away from the fixed cube geometry and build support for loading arbitrary triangle meshes from disk. The Wavefront OBJ file format was selected from a number of candidate formats, since it is widely supported by the available tools in the graphics world, and is also supported by some other Monte Carlo particle transport codes. A mesh loader implementation was created which uses the open-source `tinyobjloader` library [6] to read

in the OBJ mesh, with the necessary OptiX device buffers and `Geometry` instances then being populated accordingly.

7.4.1 *Scene Epsilon*

During development of the mesh loader, it was discovered that in certain cases particles were being irrevocably lost during simulation, leading to incorrect results. This was found to happen only when particles moved very close to the surface of an object. When tracing rays with OptiX, a *scene epsilon* parameter is used to define the smallest distance a ray can travel before testing for intersection. If this value is too small, intersections can be miscalculated, resulting in either too many or too few intersections. This is a critical problem, since the point-in-polygon algorithm relies on knowing the exact number of surface crossings. An optimal value of $1e^{-3}$ was found to behave correctly for all models tested in this work.

7.5 OPTIMISATIONS

Once the code had been ported to the GPU in a like-for-like manner, it made sense to apply some basic optimisations in order to improve performance. The first of these optimisations was to switch to using single-precision `float` arithmetic instead of double-precision. This not only significantly improved speed, but also significantly reduced memory usage. Without this optimisation, the GPU port was only able to simulate up to 10^5 particles due to device memory size limitations, compared to the 10^7 particles possible using single-precision.

A second optimisation was to avoid dynamically allocating memory on the device wherever possible, as doing so can incur significant overhead. During particle source initialisation, new particle object instances are initialised and added to the `source_bank` buffer. Pre-allocating these object instances on the host and providing them in an input device buffer significantly improves performance.

An important caveat of the way OptiX manages device kernels relates to modification of top-level device variables between kernel launches. If a variable is modified, OptiX is forced to recompile the entire "mega kernel", potentially incurring a significant cost. A solution to this was to use a single-item buffer containing a single structure holding all necessary top-level variables, and subsequently update the values inside this structure between kernel launches.

7.6 ENABLING RT CORE ACCELERATION

As described in Section 4.2, RT core acceleration is only available when defining mesh geometry via the `GeometryTriangles` API. Furthermore, in order to actually offload triangle

intersection and BVH tree traversal onto the RT cores, it is essential that the built-in *intersection* and *bounding box* programs are used. The mesh loader implementation was extended to support this, as well as the standard `Geometry` API. A simple command line switch was added to facilitate experimentation with the two implementations and correctly assign the necessary OptiX programs. Listing B.7 gives an exemplary sample of conditionally using the `GeometryTriangles` API when available, otherwise falling back to the standard `Geometry` API.

7.7 CSG SUPPORT

To enable better comparison with the original CPU version, the native CSG traversal methods were also ported to the GPU. This required additional duplication and modification of a large amount of code, to enable support for the various primitive shapes provided by OpenMC. An additional command line switch was added to activate CSG mode on the GPU, and the CSG tree and its parameters were copied onto the device as buffers.

7.8 TOOLS

For the 3D model building work, a number of tools were utilised, including Blender, Trelis (with the DAGMC plugin configured) and ParaView. For debugging host code remotely, the ARM Forge suite was used, enabling easy step-through debugging and variable evaluation. To ensure the same set of dependencies and compilers were used on all machines under test, the `spack` tool was used to install exact versions in an isolated fashion.

7.9 SUMMARY

The OpenMC GPU port was developed over a number of steps, in order to validate the implementation incrementally and provide clear checkpoints. The stepped approach also helps to reduce cognitive burden and minimise debugging effort. The code analysis undertaken during the design stage provided reasonable confidence in the overall approach, however there were several significant challenges that needed to be overcome during the implementation process.

8

EXPERIMENTAL METHODOLOGY

This chapter describes the experimental methodology that was developed to ensure accurate and reliable performance results could be gathered to evaluate the OpenMC ports in a rigorous manner. The experimentation phase is split into two distinct stages. Firstly, the initial triangle mesh and CSG versions will be examined (ignoring RT cores), enabling an understanding of the general performance characteristics of OpenMC on the GPU to be formed for both representations. These versions will be compared to the existing OpenMC CSG version on the CPU, and the existing DAGMC triangle mesh version on the CPU. Following that, our extended RT core accelerated triangle mesh version will be examined to understand what effect RT cores have on OpenMC's particle transport workload. This will then be compared with the originally developed GPU versions.

8.1 EXPERIMENTS

Two main experiments will be conducted to evaluate the behaviour of the GPU ports compared to the existing CPU versions. Firstly, we use a number of 3D models of varying size to test how *model complexity* affects performance. Secondly, we vary the *launch dimensions*, i.e. the number of simulated particles, to test its effect on performance.

The main metrics used for evaluation are the particle *calculation rate* (measured in particles/sec) and wallclock time spent in particle transport (ignoring initialisation and finalisation).

8.1.1 *Model Complexity*

While it would be desirable to test our implementation on a model directly related to Monte Carlo particle transport, access to such models was not possible at the time of writing. Therefore, we use the same five models from the RT core graphics benchmark (as described in Section 5.1). Despite the slightly contrived nature of the models in this context, they are nevertheless realistic in scale compared to real simulation models (such as the ITER fusion reactor model). We hypothesise that, similar to the RT core benchmarking, models with higher triangle counts will exhibit lower calculation rates and longer amounts of time spent in particle transport due to the additional memory bandwidth pressure caused by cache rotation of triangle vertex data. Since OpenMC natively uses CSG representation, only the Cube and Sphere models can be directly compared between CPU and GPU versions, as it is not feasible to define the other geometries using CSG due to their complexity. However, all five triangle mesh models can be sensibly compared.

8.1.2 *Launch Dimensions*

By varying the particle count, we are effectively performing a strong scaling study for each model. We hypothesise that longer runtimes and lower particle calculation rates will be observed as the particle count increases, forming a sub-linear scaling profile, due to the higher chance of factors such as thread divergence, ray incoherence and memory pressure on fission bank and tally buffers.

8.2 CONFIGURATION

We run OpenMC in its default *eigenvalue* run mode, simulated for 2 generations using 2 batches of N particles, where N ranges in powers of 10 from 10^3 up to 10^7. The number of simulated particles is used as a one-dimensional launch parameter, with one OptiX context launch being launched per generation/batch.

Each model is configured such that the closed volume sections are filled with a fissionable material (^{235}U). A bounding cube of 5^3 surrounds the model, which is filled with a vacuum. A boundary condition is set on the bounding cube, such that particles are terminated if they hit the edge of the cube. A single particle source is placed at the centre of the model, configured with the default strength and temperature parameters.

8.3 DEVICES

The GPU port is tested on each of the models, using the same two GPUs as before. For comparison, OpenMC was tested natively on a single node containing a 16-core AMD Ryzen 7 2700 CPU (using GCC 7.4). For further comparison, the DAGMC plugin was also tested (which uses CPU-based ray tracing over triangle meshes).

8.4 TEST AUTOMATION

A set of Python scripts have been developed to assist the experimentation phase. These scripts are able to remotely compile the OpenMC code on the target machine, generate the required configuration settings and remotely execute the experiments. Once the required number of samples has been gathered, the results are returned back to the host machine and are automatically processed and visualised using the `matplotlib` library. This process not only reduces unnecessary manual effort, but also ensures that experimental bias is not accidentally introduced through human error.

8.5 SUMMARY

The experiments outlined here have been designed with the intention of uncovering a range of potentially interesting characteristics of the Monte Carlo particle transport algorithm on the GPU. It should be possible to accurately characterise the OpenMC GPU ports and draw useful conclusions from the results of these experiments. Despite being contrived in this context, the chosen models are realistic in terms of scale and complexity.

Part IV

RESULTS, ANALYSIS AND CONCLUSIONS

9 PERFORMANCE RESULTS

This chapter presents and discusses performance results obtained from the OpenMC GPU port. Firstly, we consider the port from a traditional HPC perspective (explicitly not considering RT cores) to form an initial understanding of the characteristics of how OpenMC behaves on the GPU. This enables performance comparison with existing CPU-based versions. Secondly, we compare the performance of the OpenMC GPU port with and without RT core acceleration, to understand the effect it has on the particle transport workload. Finally, we compare our implementation with other GPU ports of similar codes in order to fully contextualise the evaluation.

9.1 GPU VS CPU

The GPU port supports both triangle mesh and CSG models. As discussed in Chapter 8, it was only possible to define the Cube and Sphere models using the CSG format provided by OpenMC, therefore these are the only two models which can be directly compared between CPU and GPU versions and against the triangle mesh implementation.

9.1.1 *Model Complexity*

We first consider the model complexity experiment. Figure 9.1 shows the range in calculation rate performance on the Sphere and Cube models between GPU and CPU versions, for a fixed launch size of 10^6 particles. The GPU versions are significantly faster on average than the CPU on both models. In this case, the fastest GPU version is 13x faster than the native CPU version, and 47.5x faster than the DAGMC CPU version. It is not necessarily fair to compare CPU performance on a single node, since OpenMC is capable of scaling to thousands of processors, but it is nevertheless useful to get a sense of on-node scale. The GPU triangle mesh is the most performant version overall, being 1.22x times faster on average than the CSG version on the GPU.

9.1.1.1 *Discussion*

The Sphere model is quite significantly slower than the Cube model on the native and DAGMC versions. Since DAGMC uses a ray traced triangle mesh, this can be attributed to the increased triangle count of the Sphere model introducing extra overhead. The GPU versions show much less of a dependency on triangle count, suggesting that the overhead is likely due to memory access latency which the GPU is able to hide to an extent through

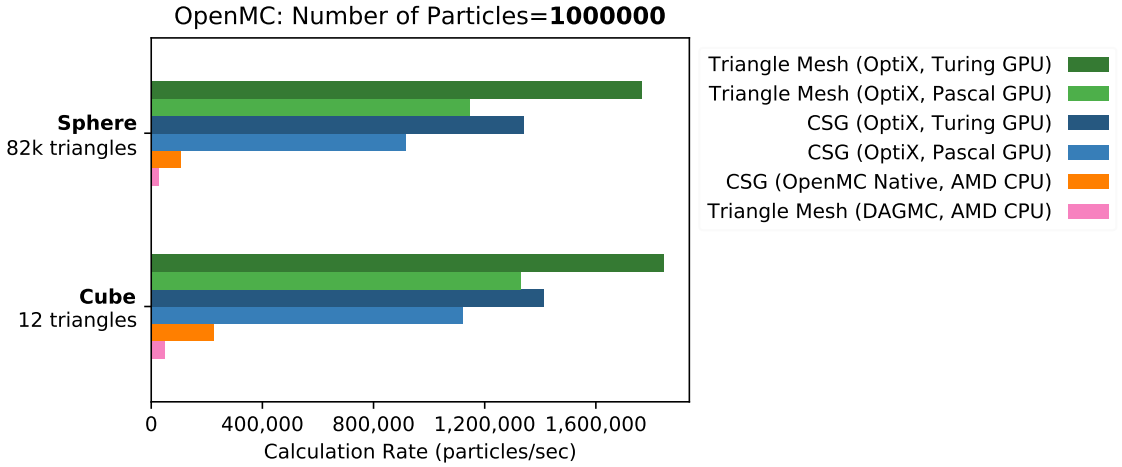


Figure 9.1: Range in particle calculation rate between CPU and GPU versions for the Sphere and Cube models with a particle count of 10^6. Higher is better.

increased parallelism. Note that the native CPU version is slower on the Sphere model simply because the CSG sphere intersection calculation is more complex than the CSG cube intersection. CSG intersection testing is almost entirely a computational problem, much unlike triangle mesh intersection which requires large volumes of vertex data to be read. The DAGMC version is the least performant, suggesting the efficiency of its ray tracing implementation over triangle meshes is comparatively poor.

A secondary observation of Figure 9.1 is the difference in performance between Turing and Pascal GPUs, with the former being 1.4x faster than the latter. This is most likely to be evidence of the memory subsystem improvements in the Turing architecture allowing greater throughput of triangle vertex data and other data buffers.

9.1.2 *Launch Dimensions*

We now consider the launch dimensionality experiment. Figure 9.2 shows how the calculation rate varies as the number of simulated particles is scaled up on the Sphere model. The Cube model displays roughly the same behaviour, so it omitted here for brevity, but can be found in Appendix A.

9.1.2.1 *Discussion*

The difference in performance becomes more pronounced as the number of particles is increased, with the GPU versions appearing to peak in performance at 10^6 particles. At this point, the GPU triangle mesh is 16.4x faster than the native CPU. This suggests that

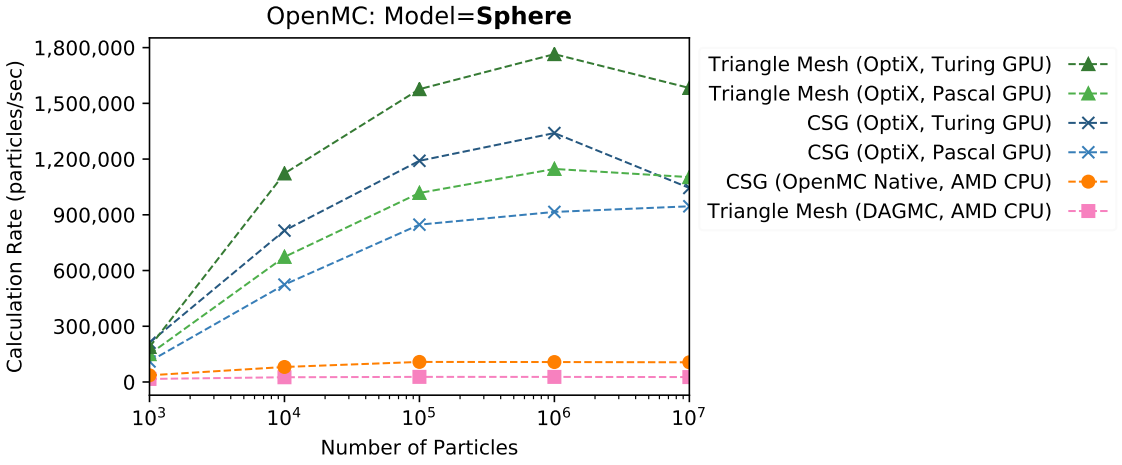


Figure 9.2: Particle calculation rate comparison between CPU and GPU versions when scaling up the particle count for the Sphere model. Higher is better.

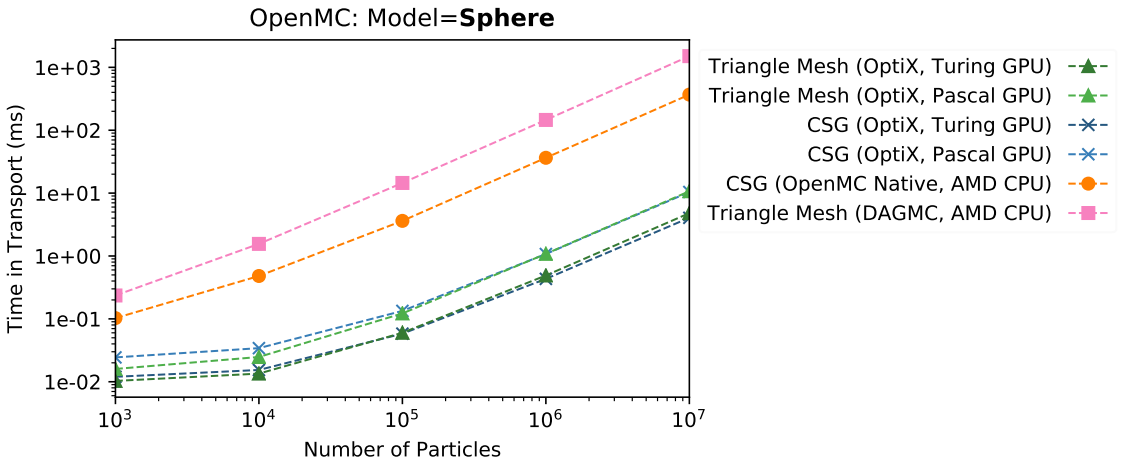


Figure 9.3: Wallclock time in transport comparison between CPU and GPU versions when scaling up the particle count for the Sphere model. Lower is better (note the logarithmic y-axis).

one or more resources on the GPU are being most efficiently used at that scale, and begin to deteriorate at larger scales. As discussed previously in Section 6.2, this is most likely a result of OptiX managing kernel grid and block sizes.

The CPU versions appear to have reasonably consistent calculation rates, regardless of number of particles, suggesting that the CPU throughput is simply saturated and cannot

process any faster without added parallelism, but does not deteriorate in performance. This behaviour is also observable in Figure 9.3, which shows how the wallclock time spent in particle transport (excluding initialisation and finalisation) increases as the particle count is scaled up. An interesting observation of Figure 9.3 is how the wallclock time increases almost linearly for the GPU versions, while the calculation rate actually deteriorates past the 10^6 mark.

9.1.3 *Summary*

We have shown here that the OpenMC GPU port already exhibits significantly better on-node performance compared to the existing CPU-based implementations, for both triangle meshes and CSG models. Our strong scaling study on particle count demonstrates that there is a performance peak at around 10^6 particles, confirming our original hypothesis that a sub-linear scaling profile would be observed.

9.2 RT CORE ACCELERATION

We now move on to the task of exploiting RT core acceleration for triangle mesh models. As described in Section 4.2, the RTX mode available with the OptiX 6 API enables RT core accelerated ray tracing on Turing GPUs, as well as a more efficient execution pipeline that simulates RT cores in software on older GPU architectures. The following sections present and discuss the results of the model complexity and particle count experiments for the modified OpenMC GPU version which supports RTX mode, as well as the originally developed GPU version.

At this point we will depart from the native CPU version and the GPU CSG version, since we will be benchmarking against triangle meshes that are not possible to define using the constructive solid geometry format provided by OpenMC. We will also depart from the DAGMC triangle mesh version, since it is in a much lower performance category and therefore not significantly valuable to consider any further.

9.2.1 *Model Complexity*

Figure 9.4 shows the calculation rates for each of the five models for a single launch of 10^6 particles. RTX mode on the Turing GPU is the fastest in all cases, being 6.0x faster on average than without. The Hairball model shows the biggest difference, being 20.1x faster with RTX mode on Turing, significantly skewing the average. On the Pascal GPU, RTX mode is 1.6x faster on average.

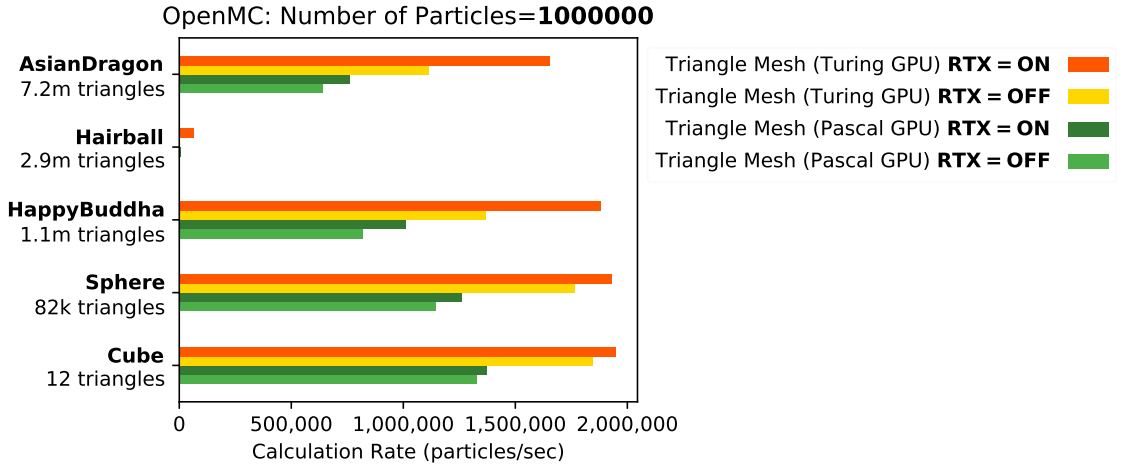


Figure 9.4: Effects of RTX mode (which uses RT cores on the Turing GPU) on all models for a particle count of 10^6. Higher is better.

9.2.1.1 *Discussion*

The number of triangles has a clear and simple effect on the calculation rate for the single solid volume models, which is all except the Hairball model. The Hairball model exhibits the most dramatic behaviour due to its high topological complexity, which has an even greater effect than on the graphics benchmark. This is because of the way the point-in-volume and boundary distance algorithm works; it traces a ray iteratively over all intersections at each layer of the model until it "misses" (i.e. extends to infinity). For the single cell models, most of the time there are only two hits to escape the model (one to escape the cell, then one to escape the bounding cube). This means that the ray tracer does not have to work very hard to calculate the full traversal path. For the Hairball model, there are potentially many more surface hits before the bounding cube is reached, meaning the ray tracer has to work much harder. This is in contrast to the graphics benchmark, which stops once the first surface intersection occurs. RT core acceleration is actually 20.1x faster in this case, allowing its superior ray tracing speed to show over the software implementation as ray tracing completely dominates the workload.

9.2.2 *Launch Dimensions*

Figures 9.5 and 9.6 show how the calculation rate varies as the number of simulated particles is scaled up for the Asian Dragon and Hairball models respectively. These two models have been selected as they represent the most realistic examples in terms of scale.

Figures for the remaining three models are omitted for brevity, but can be found in Appendix A.

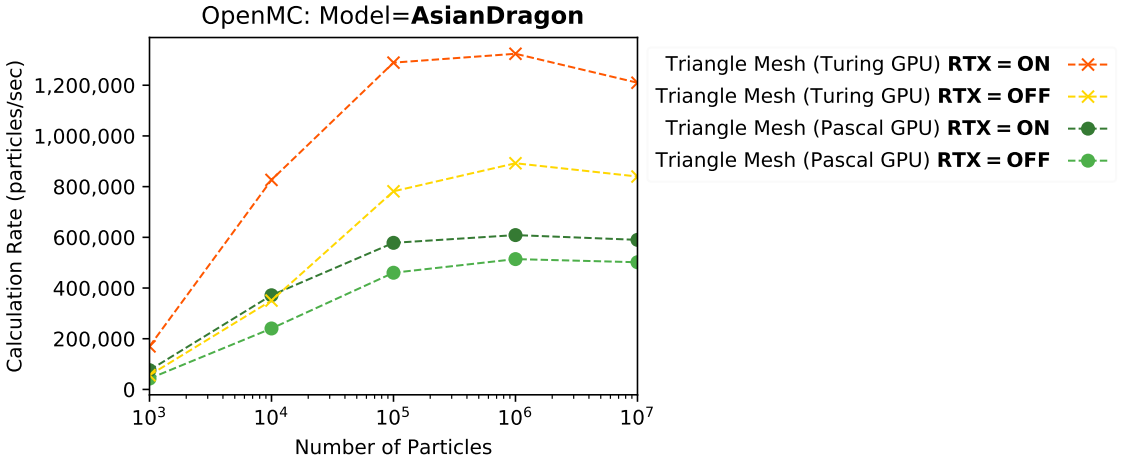


Figure 9.5: Effects of RTX mode on particle calculation rate when the particle count is scaled up for the Asian Dragon model. Higher is better.

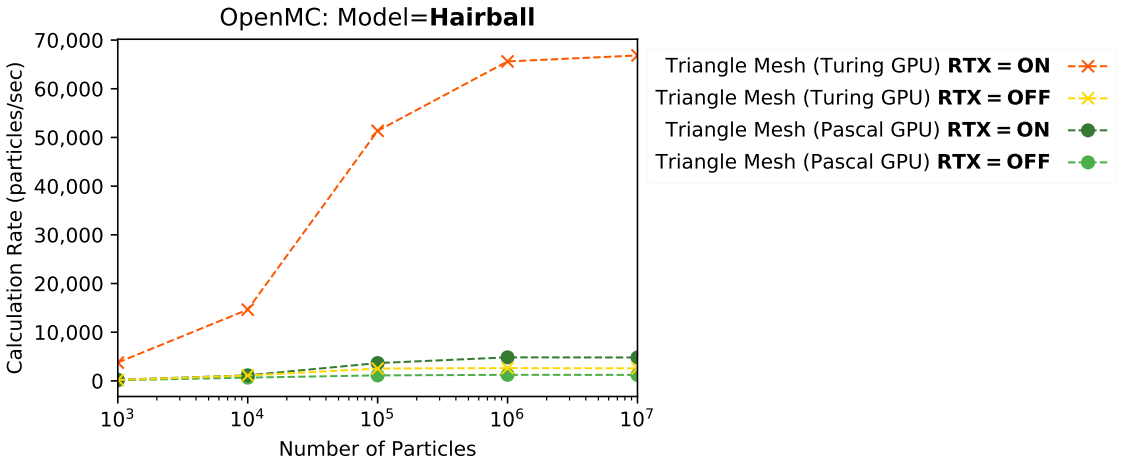


Figure 9.6: Effects of RTX mode on particle calculation rate when the particle count is scaled up for the Hairball model. Higher is better.

9.2.2.1 *Discussion*

The same performance peak at 10^6 particles is visible with RTX mode enabled (as seen in Section 9.1.2). This again is likely to be an artifact of the way OptiX internally manages

kernel launch dimensions to balance load, and that there is an optimal input problem size past which performance begins to deteriorate.

On average, the Asian Dragon and Hairball models are 1.99x and 20.1x times faster respectively with RT core acceleration on the Turing GPU than without. As the particle count is scaled up, RT core acceleration actually exhibits the same performance profile, just at a higher overall calculation rate. This suggests that the performance limiting resource is the same regardless of RT cores, but that RT cores are successfully providing more of the limiting resource which results in the overall performance gain.

Figures 9.7 and 9.8 show the wallclock time in transport for the same two models. From these figures we can observe the dominance of RT core acceleration from another angle. Additionally, it is interesting to note that using RTX mode on the Pascal GPU results in higher performance than the Turing GPU with RTX mode disabled. This is a commendation for the new execution pipeline in OptiX 6. Again we can see the extreme behaviour of the Hairball model, and how RT core acceleration yields outstanding performance when ray tracing becomes the dominating workload factor.

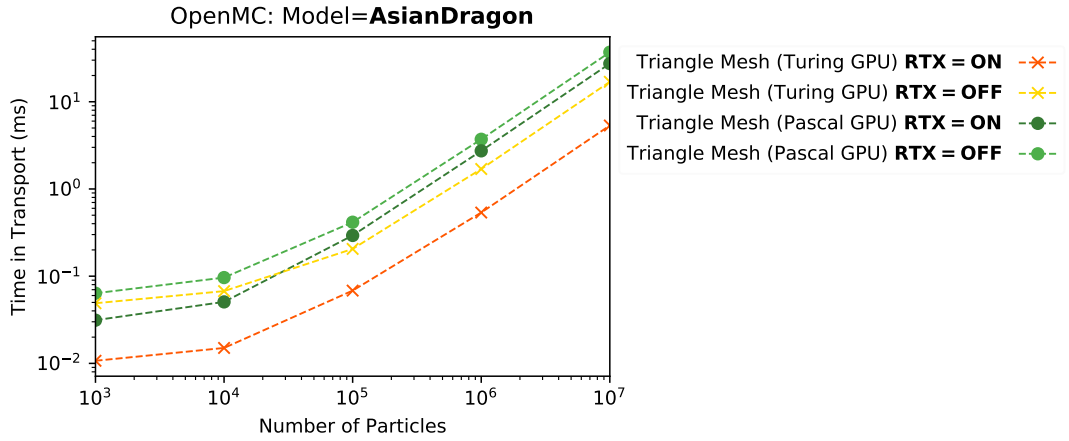


Figure 9.7: Effects of RTX mode on wallclock time in transport when the particle count is scaled up for the Asian Dragon model. Lower is better.

9.3 PROFILING

Now that we have seen how the OpenMC GPU port performs, we can begin to more closely inspect its computational characteristics through the use of profiling. This section discusses a range of factors that have been discovered through profiling which affect performance. As mentioned in Section 6.3, the NVIDIA Visual Profiler does not support the Turing architecture, nor does OptiX 6 support profiling of RTX-enabled kernels. We

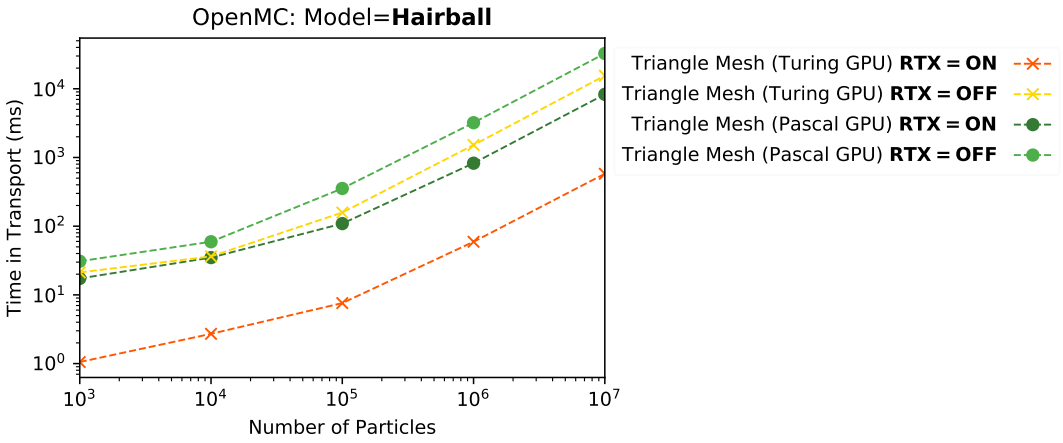


Figure 9.8: Effects of RTX mode on wallclock time in transport when the particle count is scaled up for the Hairball model. Lower is better.

attempt here to use the available profiling data from the Pascal GPU to speculate on how the main transport kernel might be behaving on the Turing GPU.

9.3.1 *Occupancy*

The kernel uses 80 registers per thread by default, which leads to an acceptable (albeit not spectacular) occupancy of 37.2% regardless of model. This is expectedly lower than the RT core benchmark, due to the greatly increased compute workload. An attempt was made to reduce the number of registers at compile time in order to improve occupancy, however this did not result in any significant performance improvement, suggesting that the OptiX scheduling mechanism is performing well in this scenario. The profiler calculates a theoretical maximum efficiency of 37.5% for this kernel, putting its achieved occupancy very close to optimal.

9.3.2 *Memory Bandwidth*

The OpenMC GPU port is memory bandwidth/memory latency bound in general, which is expected for this type of Monte Carlo particle transport code. Figure 9.9 shows the achieved device memory bandwidth for each model.

There is a noticeable drop in memory bandwidth throughput and utilisation as the models get larger. This is likely due to a number of possible reasons, which are discussed in the following subsections.

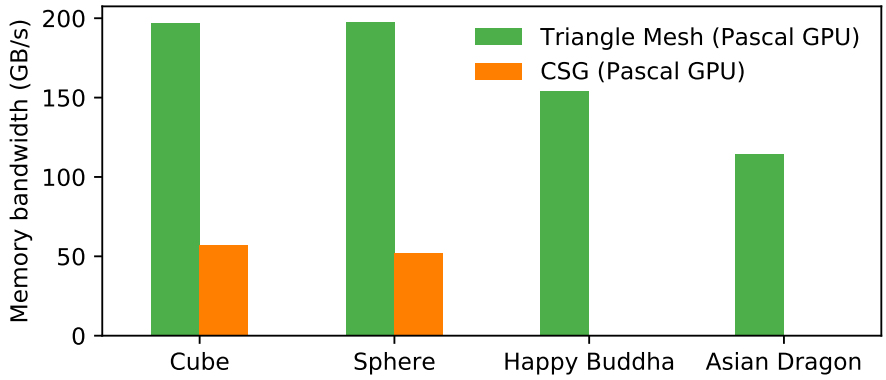


Figure 9.9: Variation in memory bandwidth throughput for different models. Note that bandwidth for the Hairball model could not be obtained, as it caused the profiler to crash due to its complexity. Only the Cube and Sphere models have CSG counterparts.

9.3.2.1 *Vertex/normal buffer lookup*

As the memory footprint of the kernel increases with model size, cache efficiency is reduced for the triangle mesh models as more vertex and normal data must be fetched from main memory during ray traversal. The Cube and Sphere models require 144 Bytes and 984 KBytes respectively, so can easily fit in L1/L2 cache, while the Buddha and Dragon models must undergo some amount of cache rotation at 13.2 MBytes and 82.4 MBytes respectively. This extra data movement puts further pressure on the memory subsystem. For CSG models, there is no vertex buffer lookup, as geometry traversal is done purely computationally, based on the boolean operator tree and primitive shape parameters.

9.3.2.2 *BVH tree traversal*

As the model size increases, the generated BVH tree becomes larger and more intricate as more bounding boxes are required to subdivide the geometry. Therefore, the ray tracing algorithm has more intricate BVH hierarchies to traverse, which again means more memory lookups and more pressure on the memory subsystem.

9.3.2.3 *Tally writes*

Each simulated particle maintains its own set of tallies, which are written into global tally buffers indexed by particle number. While this approach means that atomic writes are not necessary, the incoherent and stochastic movement of particles leads to a random write pattern on those buffers. This effect becomes more detrimental to performance as the launch dimensions are increased, but is not affected by model size.

9.3.2.4 *Particle banks*

Particles are initialised on the host and loaded onto the device at the beginning of the simulation. Each thread reads a particle to simulate based on its launch index. Subsequently, as particles collide with other particles in the material, new particles are created and stored in the fission and secondary particle banks. These reads introduce an additional source of memory pressure which scales with launch dimensionality, independent of model size.

9.3.2.5 *Workload imbalance and thread divergence*

Due to the random nature of particle histories, some particles may survive longer than others, introducing thread imbalance. This has been observed in other codes, such as [18] and [10]. Fortunately for this implementation, the OptiX scheduling mechanism does a reasonably good job of minimising the impact of the imbalance, being able to ensure that short-lived threads are efficiently replaced, thereby maintaining a good percentage of theoretical occupancy.

9.4 COMPARISON TO RELATED WORK

To fully contextualise the evaluation of our implementation, it is important to compare results with existing work on porting Monte Carlo particle transport codes to the GPU. Our results compare competitively with other studies such as [18] and [10] which observed ~3x and ~2.7x speedups respectively using similar methods on older GPU architectures. We expect that simply using newer GPU architectures would increase these numbers, but that our implementation is benefiting majorly from the optimisations inside the OptiX API, particularly the scheduling mechanism. Even without RT core acceleration, this port is comparable to, or better than, existing ports. Since this is the first known work to use RT core acceleration for a Monte Carlo particle transport code (or indeed any scientific code), there is no direct point of reference for comparison. Nevertheless, we can confidently say that a significant proportion of the ~10x speedup advertised by NVIDIA for graphics workloads has been obtained here.

9.5 SUMMARY

The results presented in this chapter demonstrate that OpenMC is well suited for GPU parallelism, as originally hypothesised. We have obtained significant on-node speedups of up to 13x using triangle meshes on a Turing GPU, and up to 10x using CSG models compared to the native CPU version. Furthermore, exploiting RT core acceleration on triangle meshes yields between ~2x and ~20x additional speedup. This proves that the Turing architecture is a formidable platform for Monte Carlo particle transport. Our

implementation compares extremely competitively with existing GPU ports of similar codes.

We have characterised OpenMC as being memory latency bound through profiling on a Pascal GPU. The case is almost certainly the same on the Turing GPU, despite the lack of available profiling data. The principal benefits of RT core acceleration are the ability for the SM to perform other computational work while the RT cores perform ray tracing probes, and the ability for RT cores to cache triangle vertex and BVH tree data which significantly alleviates device memory pressure.

10 CRITICAL EVALUATION

This chapter critically evaluates the extent to which this project has achieved its aims and objectives, and analyses the effectiveness of the approach that was taken during the developmental stages of the project.

10.1 AIMS AND OBJECTIVES

The overall aim of this project was to exploit RT core acceleration for a scientific application, namely Monte Carlo particle transport. As this is the first known attempt to do so (to the best of our knowledge), it was important to break down the work into logical pieces in order to validate the feasibility of the approach at each stage. The following subsections discuss each of the concrete objectives (as originally proposed in Section 1.3).

10.1.1 *RT Core Graphics Benchmarking*

It was originally proposed to benchmark the RT cores on a graphics workload at the end of the project, following the OpenMC GPU port. However, it was understood very early on that completing this task at the beginning would prove beneficial, as it would yield a performance baseline for which to subsequently aspire to. This also allowed deep inspection of RT cores at their most basic level, allowing critical understanding of their operation and behaviour to be gathered early on. During the OpenMC porting work, this allowed the separation and understanding of the different computational factors in isolation.

10.1.2 *Implementing new GPU ports of OpenMC*

A crucial aspect of the project was to validate the feasibility of using triangle mesh geometries with OpenMC, and traversing them using ray tracing on a GPU. The first stage of this was using the "single ray" approach with just the geometry traversal running on the GPU. This was developed reasonably quickly using a fixed geometry. Although not the final implementation (since it was an extremely inefficient use of GPU resources) this turned out to be a very efficient use of time, as a lot of the geometry related code could be reused.

10.1.2.1 *Porting the Main Transport Loop*

Once the geometry aspect was validated, the main loop could then be ported to the GPU. This was by far the most challenging aspect of this project, and took a lot of time and effort. The most significant challenge came with debugging the OptiX code, which is extremely difficult due to the "mega kernel" mechanism. A large amount of time was spent on profiling, looking for ways to get information from the Turing GPU, most of which was wasted as it transpired not to be possible with the currently available tools. Despite this limitation, the profiling results gleaned from the Pascal GPU turned out to be sufficient to characterise the application, and to enable reasonable assumptions to be made about what was happening on the Turing GPU.

10.1.2.2 *Supporting Complex Models*

Once the main transport loop was running on the GPU with a fixed geometry, support for arbitrarily complex geometries could be added. This required choosing a model format which made sense. The Wavefront OBJ format was chosen as it is widely used and supported by the tools in the industry. A fairly significant amount of time was spent recreating the same models in both OBJ and DAGMC format, so that accurate comparisons could be made between CPU and GPU versions. The Trelis tool used to make DAGMC files is not particularly user friendly, and would often crash or lock up, especially with larger models. It was later discovered that DAGMC files could be created directly from OBJ files without Trelis, which would have saved a great deal of time. An even better approach would have been to interface the OptiX geometry loader with the DAGMC API, removing the need to convert to OBJ format entirely. Ultimately, DAGMC turned out to perform worse than expected, calling into question whether the time spent getting it working was well spent.

While efforts were made to acquire realistic models early on in the project, this proved to be a slow and difficult process. Towards the very end of the project, an ITER model was acquired, but there was not enough time to gather and present results for it here. Nevertheless, the chosen models were effective in uncovering a range of performance characteristics from both the GPU port in general, and the RT core acceleration. It was unfortunate that it was not possible to test with a complex native OpenMC model that had a matching triangle mesh version. Nevertheless, the Cube and Sphere models were sufficient to show that triangle meshes are more generally performant overall under our experimental conditions.

10.1.3 *Results Gathering, Profiling and Performance Evaluation*

Significant upfront effort was spent scripting automating the results gathering and visualisation process. This proved to be a very good investment in time, since the need for a

lot of manual effort was removed and allowed results to be gathered while other tasks were being executed concurrently. Furthermore, as the GPU ports went through several iterations of development, results could be re-collected quickly and efficiently.

It was fortunate that there was little traffic on the HPC Zoo nodes containing the GPUs used in this work. However, there were a number of times when the nodes were down for maintenance. Despite this, it did not have a significant negative impact on the project overall, as the disruptions were minimal and communicated well in advance, allowing other tasks to be completed in the meantime.

10.1.4 *Providing Insight into RT Cores*

A key factor affecting the level of knowledge that could be gleaned about RT cores was the lack of profiling support for both the Turing architecture and the OptiX library. This effectively required a certain amount of speculation derived from profiling on the Pascal GPU. It was initially hoped that RT cores would be more flexible than they were ultimately discovered to be. The unavoidable necessity of using the OptiX API dramatically limits their potential usefulness for other applications. Nevertheless, several opportunities have been discovered, which are outlined in Section 11.1.

10.2 SUMMARY

This project has successfully implemented a novel and highly performant port of OpenMC to NVIDIA GPUs, supporting both CSG and triangle mesh geometries, which is further accelerated via RT cores on triangle meshes. Unique insights have been presented into RT cores and their potential applicability to other scientific research domains, which represents a valuable contribution to the wider research community. Despite the challenging set of objectives and several major implementation hurdles, this work has succeeded in achieving its goals, and has produced results above and beyond expectations. Both the UKAEA and the MIT CRPG are actively following up this work for future research and development.

11 CONCLUSIONS

In terms of on-node performance, the speedups obtained here of between ~10x on CSG models and ~33x on RT core accelerated triangle meshes show that the Turing architecture is a formidable platform for Monte Carlo particle transport. Our results compare competitively with other GPU-based ports of Monte Carlo particle transport codes such as [18] and [10] which observed ~3x and ~2.7x speedups respectively using similar methods on older GPU architectures.

11.1 WIDER APPLICABILITY OF RT CORES

A secondary goal in this paper was to gain an understanding, through implementation, of the kinds of scientific workloads which might be suitable candidates for RT core acceleration. It is clear from the OpenMC results that Monte Carlo particle transport is indeed a natural candidate to benefit from the accelerated BVH traversal and triangle intersection testing brought by RT cores, as hypothesised.

Any algorithm which spends a significant amount of its runtime tracing large numbers of straight lines through large 3D triangle mesh geometries should be able to exploit RT cores. If that is not the case, or the code cannot be reformulated to do so, then RT cores are not going to be useful. Despite this restriction, several potential candidates have been identified. In cancer radiotherapy for example, patient dose simulations are required before treatment, which use 3D biological models. Improving performance in that sector could potentially mean radiologists could provide diagnosis and treatment to patients on a much shorter timescale [16]. Another example can be found in the geoscience field, where seismic ray tracing is used to simulate the movement of seismic waves through the earth in order to gain a representation of the Earth's interior. This technique is useful for analysing earthquakes, and also has potential for oil and gas exploration.

11.2 PERFORMANCE PORTABILITY

It may be non-trivial for other codes to reap the benefits described above, due to the restrictions placed on how RT cores can be accessed. There are no PTX instructions available for the developer to issue instructions to RT cores (that have been made publicly known), necessitating the use of the OptiX library and all of its ramifications in terms of architectural design. As a result, existing codes would likely have to undergo significant development rework, which may or may not be feasible depending on the amount of investment in the original code and the hardware it has been designed to run on. The upcoming OptiX 7 release is much less opinionated than previous versions and is much closer in terms of

API design to vanilla CUDA, which could potentially ease the performance portability burden. Ideally, performance portable enabling languages such as OpenCL would support RT core acceleration, but that is currently not the case.

Furthermore, OpenMC is written in modern C++, and makes extensive use of the C++ STL, polymorphism, and other C++ features. Since many of these language features are not fully supported by CUDA code, a large amount of code duplication and reworking has been required to allow OpenMC to run on the GPU. This results in two distinct codebases, which is a major drawback in terms of maintainability. Performance portability is becoming increasingly important in HPC as the range of available compute platforms continues to grow [30], which means that these drawbacks are likely to heavily influence any decisions to port large-scale existing codebases.

There is a definite trend in HPC towards many-GPU designs, such as in the latest Summit [14] supercomputer which contains tens of thousands of Volta-class NVIDIA GPUs. It is conceivable that future supercomputer designs may feature GPUs containing RT cores, thus providing additional motivation to exploit them. It is also conceivable that RT cores may be opened up by NVIDIA in future iterations, and may eventually find support in more performance portable libraries.

11.3 FUTURE WORK

A number of potentially valuable extensions to this work have been identified. Firstly, it would be useful from an experimental perspective to compare the OptiX-based ports of OpenMC with a vanilla CUDA port. This would provide an additional point of comparison, and would allow us to understand to what extent the OptiX library is contributing to the achieved performance results. Secondly, the excellent work surrounding the DAGMC toolkit could be incorporated into the OpenMC GPU port. If models could be loaded from DAGMC files rather than OBJ files, the existing CAD workflows could be reused, removing the need for difficult model conversion stages. Thirdly, the built-in support for multi-GPU execution in the OptiX API could be utilised to further increase on-node performance.

OpenMC supports many rich features, not all of which are currently supported by the GPU port. In the near term, features such as lattices, photon tracking and custom mesh tallies would bring OpenMC closer to a production-ready state on the GPU.

It is useful to note that NVIDIA is not alone in putting dedicated ray tracing hardware in its GPUs - in fact, both Intel and AMD are planning similar fixed-function hardware units in their upcoming GPU architectures. This provides further suggestion that there may be increased motivation to implement support for RT core acceleration in the near future.

11.4 SUMMARY

A viable, novel and highly performant GPU port of OpenMC has been presented using the OptiX library, which supports both native CSG models and triangle mesh models. Triangle meshes provide the most significant performance improvements compared to CPU-based versions, and the presence of RT cores in the Turing architecture provides impressive additional speedup. We believe ours is the first work to use ray tracing hardware to accelerate Monte Carlo particle transport, and indeed any scientific application. A major factor contributing to performance is the geometric complexity of the model. Specifically, highly topologically complex models with many layers are the most computationally demanding. The speedup from RT core acceleration becomes even more significant at this scale, due to its ability to relieve the SM of large numbers of computations and its internal caching of triangle vertex and BVH tree data.

The necessity of using the OptiX API to acquire these speedups is a significant drawback. The architectural ramifications may prove to be prohibitive to other scientific codes with large and complex codebases. Nevertheless, while further work is required to support the entire OpenMC feature set, the developed code is already being evaluated by the UKAEA for ITER neutronics analysis, and is being considered for inclusion into the official OpenMC repository for further development.

This work also presented a novel in-depth analysis into RT cores from a graphical rendering perspective, used to rigorously corroborate the manufacturer's performance claims and to provide detailed understanding for future research into their use. A rare discussion of the wider applicability of RT cores has been given, intended to facilitate future studies.

Finally, a full conference paper based on this work has been submitted for peer-reviewed publication at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19) which is expected to be accepted shortly following submission of this dissertation.

BIBLIOGRAPHY

[1] Timo Aila and Tero Karras. 'Relative encoding for a block-based bounding volume hierarchy'. In: 1 (2018).

[2] Andreu Badal, Iacovos Kyprianou, Diem Phuc Banh, Aldo Badano and Josep Sempau. 'PenMesh-Monte Carlo radiation transport simulation in a triangle Mesh Geometry'. In: *IEEE Transactions on Medical Imaging* (2009). ISSN: 02780062. DOI: 10.1109/TMI.2009.2021615.

[3] Ryan M. Bergmann and Jasmina L. Vujić. 'Monte Carlo Neutron Transport on GPUs'. In: 2014, V004T11A002. DOI: 10.1115/icone22-30148.

[4] Ryan M. Bergmann, Kelly L. Rowland, Nikola Radnović, Rachel N. Slaybaugh and Jasmina L. Vujić. 'Performance and accuracy of criticality calculations performed using WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs'. In: *Annals of Nuclear Energy* 103 (2017), pp. 334–349. ISSN: 18732100. DOI: 10.1016/j.anucene.2017.01.027.

[5] E. D. Cashwell and C. J. Everett. 'A Practical Manual on the Monte Carlo Method for Random Walk Problems'. In: *Mathematics of Computation* (2006). ISSN: 00255718. DOI: 10.2307/2003194.

[6] Syoyo Fujita. *syoyo/tinyobjloader*. 2019. URL: https://github.com/syoyo/tinyobjloader.

[7] N A Gentile, R J Procassini and H A Scott. 'Monte Carlo Particle Transport: Algorithm and Performance Overview'. In: (2005).

[8] Khronos Group. *The Vulkan API Specification and related tools*. 2019. URL: https://github.com/KhronosGroup/Vulkan-Docs.

[9] Eric Haines. *Ray Tracing Gems*. 2019. ISBN: 9781484244265. DOI: 10.1007/978-1-4842-4427-2.

[10] Steven P. Hamilton and Thomas M. Evans. 'Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code'. In: *Annals of Nuclear Energy* 128 (2019), pp. 236–247. ISSN: 18732100. DOI: 10.1016/j.anucene.2019.01.012. URL: https://doi.org/10.1016/j.anucene.2019.01.012.

[11] Steven P. Hamilton, Stuart R. Slattery and Thomas M. Evans. 'Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms'. In: *Annals of Nuclear Energy* 113 (2018), pp. 506–518. ISSN: 18732100. DOI: 10.1016/j.anucene.2017.11.032. URL: https://doi.org/10.1016/j.anucene.2017.11.032.

[12] M. Hapala and V. Havran. 'Review: Kd-tree traversal algorithms for ray tracing'. In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213. ISSN: 14678659. DOI: 10.1111/j.1467-8659.2010.01844.x.

[13] A. Heimlich, A. C.A. Mol and C. M.N.A. Pereira. 'GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation'. In: *Progress in Nuclear Energy* 53.2 (2011), pp. 229–239. ISSN: 01491970. DOI: 10.1016/j.pnucene.2010.09.011. URL: http://dx.doi.org/10.1016/j.pnucene.2010.09.011.

[14] Jonathan Hines. 'Stepping up to summit'. In: *Computing in Science and Engineering* (2018). ISSN: 15219615. DOI: 10.1109/MCSE.2018.021651341.

[15] Xun Jia, Peter Ziegenhein and Steve B. Jiang. *GPU-based high-performance computing for radiation therapy*. 2014. DOI: 10.1088/0031-9155/59/4/R151.

[16] Xun Jia, Xuejun Gu, Yan Jiang Graves, Michael Folkerts and Steve B. Jiang. 'GPU-based fast Monte Carlo simulation for radiotherapy dose calculation'. In: (2011), p. 18. ISSN: 0031-9155. DOI: 10.1088/0031-9155/56/22/002. arXiv: 1107.3355. URL: http://arxiv.org/abs/1107.3355.

[17] Zhe Jia, Marco Maggioni, Jeffrey Smith and Daniele Paolo Scarpazza. 'Dissecting the NVidia Turing T4 GPU via Microbenchmarking'. In: (2019). arXiv: 1903.07486. URL: http://arxiv.org/abs/1903.07486.

[18] Daniel Karlsson and Zhao Yuer. 'Monte-Carlo neutron transport simulations on the GPU'. In: (2018).

[19] Tero Karras and Timo Aila. 'Fast parallel construction of high-quality bounding volume hierarchies'. In: (2013), p. 89. DOI: 10.1145/2492045.2492055.

[20] Tero Karras and Tero. 'Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees'. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics* (2012), pp. 33–37. DOI: 10.2312/eggh/hpg12/033-037. URL: https://dl.acm.org/citation.cfm?id=2383801.

[21] Matt Martineau, Patrick Atkinson and Simon McIntosh-Smith. 'Benchmarking the NVIDIA V100 GPU and tensor cores'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11339 LNCS. Springer Verlag, 2019, pp. 444–455. ISBN: 9783030105488. DOI: 10.1007/978-3-030-10549-5_35.

[22] Matt Martineau and Simon McIntosh-Smith. 'Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App'. In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC* 2017-Septe (2017), pp. 498–508. ISSN: 15525244. DOI: 10.1109/CLUSTER.2017.83.

[23] Morgan McGuire. *Computer Graphics Archive*. 2017. URL: https://casual-effects.com/data.

[24] Microsoft Inc. *Announcing Microsoft DirectX Raytracing*. 2019. URL: https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/.

[25] Monte Carlo Team. 'MCNP - A General Monte Carlo N-Particle Transport Code, Version 5'. In: *Los Alamos Nuclear Laboratory* (2005). DOI: LA-UR-03-1987.

[26] NVIDIA. 'NVIDIA Turing GPU'. In: *White Paper* (2018).

[27] NVIDIA. *NVIDIA OptiX 6.0 Programming Guide*. 2019. URL: https://raytracing-docs.nvidia.com/optix{_}6{_}0/guide{_}6{_}0/index.html (visited on 25/04/2019).

[28] Miklos Nyers. *GPU rendering RTX ray tracing benchmarks - RTX 2080 Ti*. 2019. URL: http://boostclock.com/show/000250/gpu-rendering-nv-rtxon-gtx1080ti-rtx2080ti-titanv.html (visited on 25/04/2019).

[29] Steven G. Parker et al. 'OptiX: A General Purpose Ray Tracing Engine'. In: *ACM Transactions on Graphics* 29.4 (2010), p. 1. ISSN: 07300301. DOI: 10.1145/1778765.1778803.

[30] S. J. Pennycook, J. D. Sewall and V. W. Lee. 'A Metric for Performance Portability'. In: (2016), pp. 1–7. arXiv: 1611.07409. URL: http://arxiv.org/abs/1611.07409.

[31] R. Procassini, M. O'Brien and J. Taylor. 'Load balancing of parallel Monte Carlo transport calculations'. In: *International topical meeting on mathematics and computation, supercomputing, reactor physics and nuclear and biological applications* (2005). URL: https://inis.iaea.org/search/search.aspx?orig{_}q=RN:40054813.

[32] Nunu Ren, Jimin Liang, Xiaochao Qu, Jianfeng Li, Bingjia Lu and Jie Tian. 'GPU-based Monte Carlo simulation for light propagation in complex heterogeneous tissues'. In: *Optics Express* (2010). DOI: 10.1364/oe.18.006811.

[33] Paul K. Romano and Benoit Forget. 'The OpenMC Monte Carlo particle transport code'. In: *Annals of Nuclear Energy* 51 (2013), pp. 274–281. ISSN: 03064549. DOI: 10.1016/j.anucene.2012.06.040.

[34] Paul Kollath Romano. 'Parallel Algorithms for Monte Carlo Particle Transport Simulation on Exascale Computing Architectures'. PhD thesis. Massachusetts Institute of Technology, 2013, p. 199.

[35] Patrick C. Shriwise. 'Geometry Query Optimizations in CAD-based Tessellations for Monte Carlo Radiation Transport'. PhD thesis. University of Wisconsin - Madison, 2018.

[36] Patrick C. Shriwise, Andrew Davis, Lucas J. Jacobson and Paul P.H. Wilson. 'Particle tracking acceleration via signed distance fields in direct-accelerated geometry Monte Carlo'. In: *Nuclear Engineering and Technology* 49.6 (2017), pp. 1189–1198. ISSN: 2234358X. DOI: 10.1016/j.net.2017.08.008. URL: https://doi.org/10.1016/j.net.2017.08.008.

[37] Andrew R. Siegel, Kord Smith, Paul K. Romano, Benoit Forget and Kyle G. Felker. 'Multi-core performance studies of a Monte Carlo neutron transport code'. In: *International Journal of High Performance Computing Applications* (2014). ISSN: 10943420. DOI: 10.1177/1094342013492179.

[38] Timothy J. Tautges. 'MOAB-SD: Integrated structured and unstructured mesh representation'. In: *Engineering with Computers* (2004). ISSN: 01770667. DOI: 10.1007/s00366-004-0296-0.

[39] Samuel Thomson. 'Ray-traced Radiative Transfer on Massively Threaded Architectures'. In: (2018). URL: https://www.era.lib.ed.ac.uk/bitstream/handle/1842/31277/Thomson2018.pdf.

[40] A. Turner, A. Burns, B. Colling and J. Leppänen. 'Applications of Serpent 2 Monte Carlo Code to ITER Neutronics Analysis'. In: *Fusion Science and Technology* 74.4 (2018), pp. 315–320. ISSN: 19437641. DOI: 10.1080/15361055.2018.1489660.

[41] Andrew Turner. 'Investigations into alternative radiation transport codes for ITER neutronics analysis'. In: *Transactions of the American Nuclear Society*. 2017.

[42] Paul P.H. Wilson, Timothy J. Tautges, Jason A. Kraftcheck, Brandon M. Smith and Douglass L. Henderson. 'Acceleration techniques for the direct use of CAD-based geometry in fusion neutronics analysis'. In: *Fusion Engineering and Design* (2010). ISSN: 09203796. DOI: 10.1016/j.fusengdes.2010.05.030.

[43] Kai Xiao, Danny Z. Chen, X. Sharon Hu and Bo Zhou. 'Monte Carlo Based Ray Tracing in CPU-GPU Heterogeneous Systems and Applications in Radiation Therapy'. In: June 2015 (2015), pp. 247–258. DOI: 10.1145/2749246.2749271.

Part V

APPENDIX

A

FULL RESULTS

A.1 RT CORE BENCHMARK

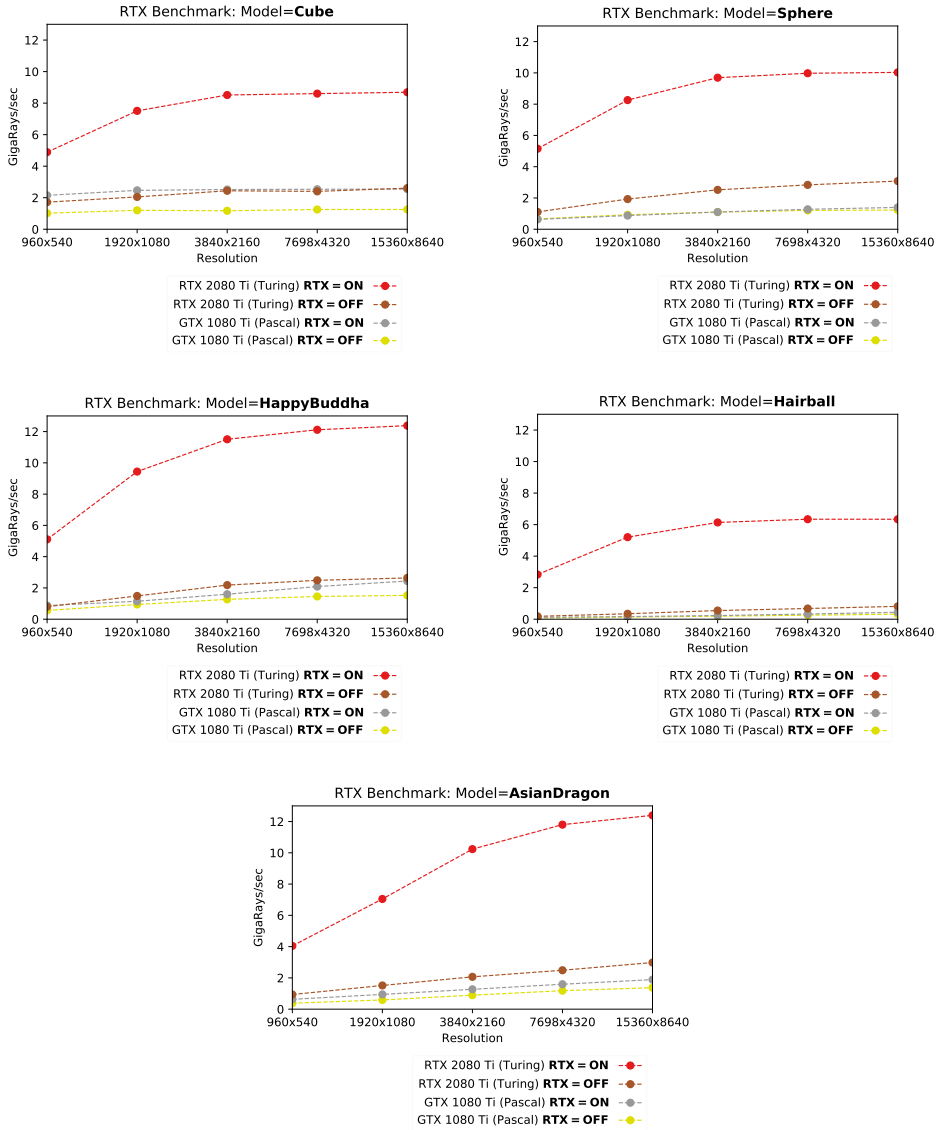


Figure A.1: Plots from the RT core benchmark, showing how increasing the image resolution affects performance for each of the five tested models.

A.2 OPENMC GPU PORT

A.2.1 *CPU vs GPU*

A.2.1.1 *Calculation Rate*

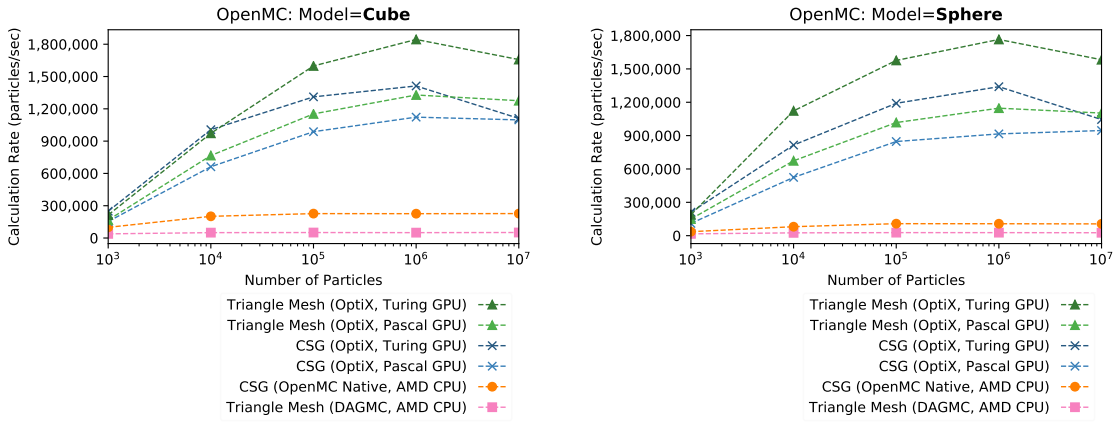


Figure A.2: Calculation rate vs particle count for all models which are directly comparable between CPU and GPU versions.

A.2.1.2 *Time in Transport*

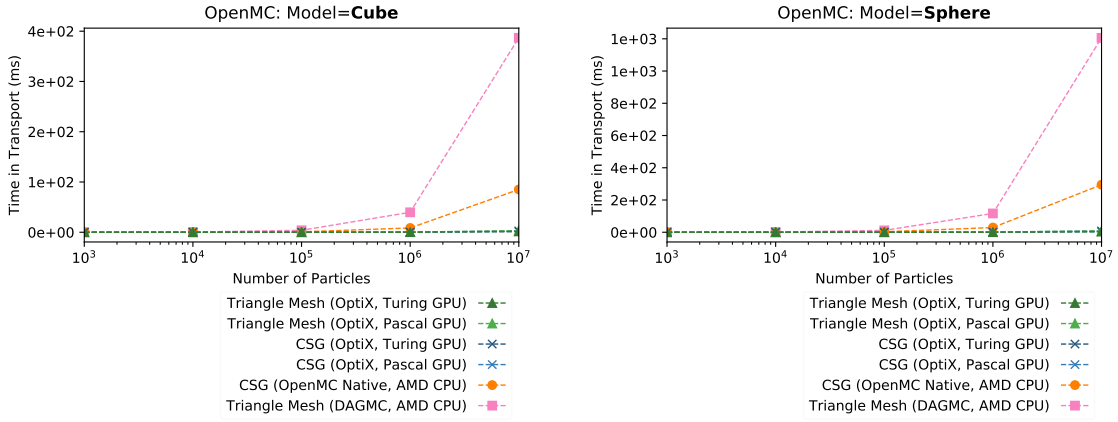


Figure A.3: Time in transport vs particle count for all models which are directly comparable between CPU and GPU versions.

A.2.2 *RT Core Acceleration*

A.2.2.1 *Calculation Rate*

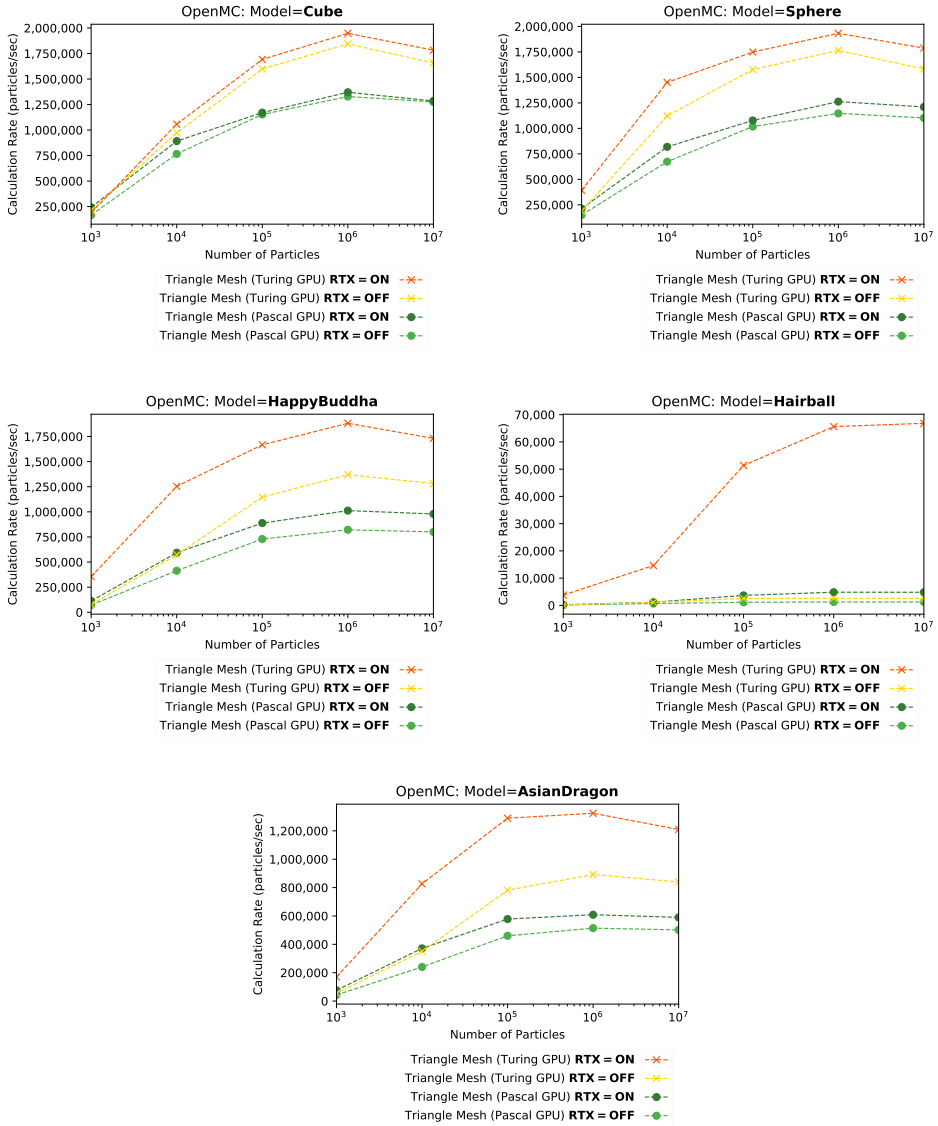


Figure A.4: Calculation rate vs particle count for all models, showing how RT core acceleration affects performance.

A.2.2.2 *Time in Transport*

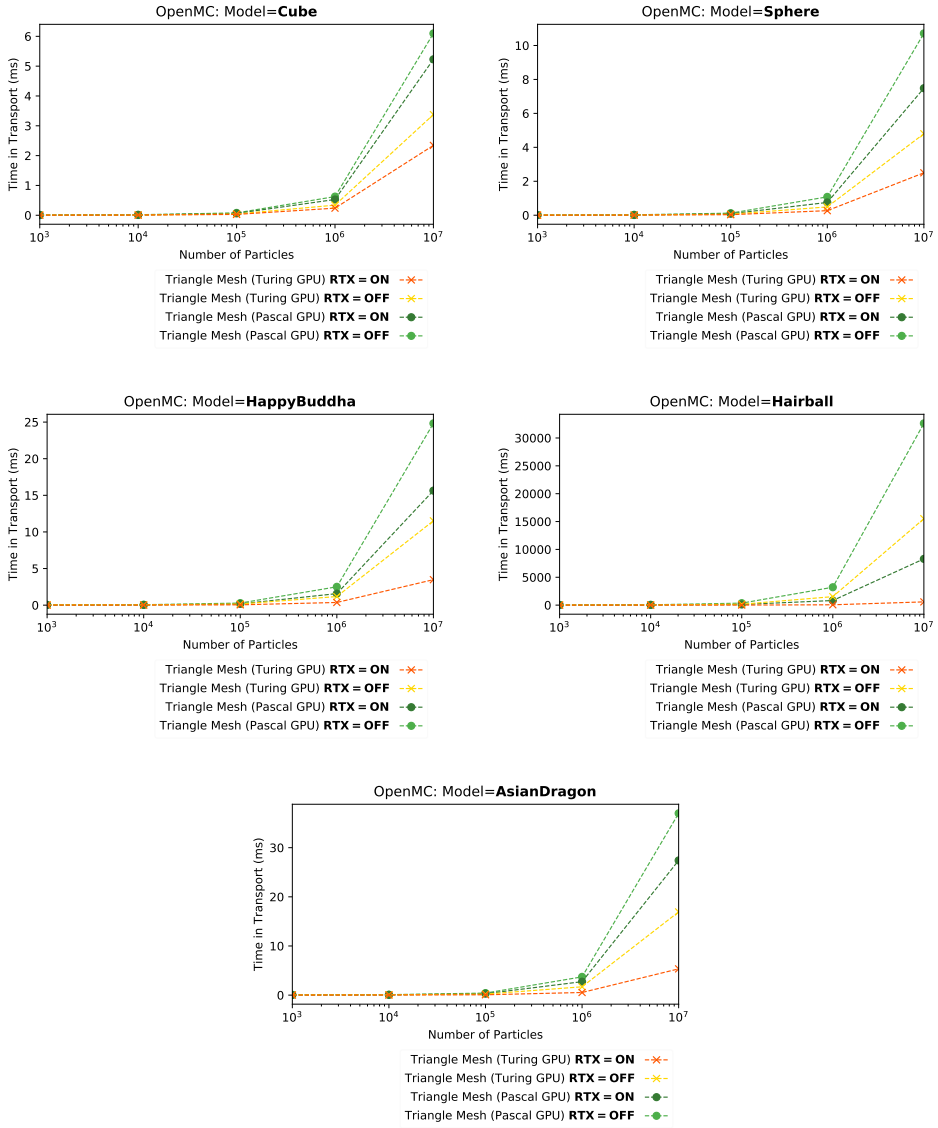


Figure A.5: Time in transport rate vs particle count for all models, showing how RT core acceleration affects performance.

B

CODE SAMPLES

B.1 OPTIX DEVICE CODE SAMPLES

```
struct PerRayData {
  float3 result;
}

rtBuffer<uchar4, 2> output_buffer;

rtDeclareVariable(uint2, launch_index, rtLaunchIndex, );
rtDeclareVariable(uint2, launch_dim,   rtLaunchDim, );

RT_PROGRAM void generate_ray() {
  float3 ray_origin = make_float3(0.f, 0.f, 0.f);
  float3 ray_direction = make_float3(1.f, 1.f, 1.f);

  optix::Ray ray = optix::make_Ray(ray_origin, ray_direction, 0, 0);

  PerRayData prd;
  rtTrace(top_object, ray, prd);

  output_buffer[launch_index] = make_color(prd.result);
}
```

Listing B.1: A sample OptiX *ray generation* program which launches a primary ray and writes the result field from the ray payload to an output buffer, interpreting it directly as a colour.

```
rtDeclareVariable(PerRayData, payload, rtPayload, );
rtDeclareVariable(float3, shading_normal, attribute shading_normal, );

RT_PROGRAM void closest_hit() {
  payload.result = normalize(rtTransformNormal(RT_OBJECT_TO_WORLD,
                             shading_normal)) * 0.5f + 0.5f;
}
```

Listing B.2: A sample OptiX *closest hit* program which writes the surface normal of the intersected surface into the ray payload, normalised to a colour value.

```
rtDeclareVariable(float3, bg_colour, , );

RT_PROGRAM void any_hit() {
  payload.result = bg_colour;
}
```

Listing B.3: A sample OptiX *any hit* program which simply assigns a fixed background colour to the payload result.

```
RT_PROGRAM void triangle_mesh_intersect(int primitive_idx) {
  const int3 v_idx = index_buffer[primitive_idx];
  const float3 p0 = vertex_buffer[v_idx.x];
  const float3 p1 = vertex_buffer[v_idx.y];
  const float3 p2 = vertex_buffer[v_idx.z];

  // Intersect ray with triangle
  float3 n;
  float t, beta, gamma;
  if (intersect_triangle(ray, p0, p1, p2, n, t, beta, gamma)) {
    if (rtPotentialIntersection(t)) {

      geometric_normal = normalize(n);
      if (normal_buffer.size() == 0) {
        shading_normal = geometric_normal;
      } else {
        float3 n0 = normal_buffer[v_idx.x];
        float3 n1 = normal_buffer[v_idx.y];
        float3 n2 = normal_buffer[v_idx.z];
        shading_normal = normalize(n1 * beta + n2 * gamma + n0 * (1.0f - beta - gamma));
      }

      // Report the intersection
      rtReportIntersection(material_buffer[primitive_idx]);
    }
  }
}
```

Listing B.4: A sample OptiX *intersection* program for a triangle mesh geometry.

```
RT_PROGRAM void mesh_bounds(int primitive_idx, float result[6]) {
  const int3 v_idx = index_buffer[primitive_idx];

  const float3 v0 = vertex_buffer[v_idx.x];
  const float3 v1 = vertex_buffer[v_idx.y];
  const float3 v2 = vertex_buffer[v_idx.z];
  const float area = length(cross(v1 - v0, v2 - v0));

  optix::Aabb *aabb = (optix::Aabb *) result;

  if (area > 0.0f && !isinf(area)) {
    aabb->m_min = fminf(fminf(v0, v1), v2);
    aabb->m_max = fmaxf(fmaxf(v0, v1), v2);
  } else {
    aabb->invalidate();
  }
}
```

Listing B.5: A sample OptiX *bounding box* program for a triangle mesh geometry.

B.2 OPTIX HOST CODE SAMPLES

```
// Create context object
optix::Context context = optix::Context::create();

// Create a single triangle in 3D space
unsigned int num_triangles = 1;
float3 vertices[3];
unsigned int indices[3] {0, 1, 2};

vertices[0] = optix::make_float3(-5.0f, 0.0f, 0.0f);
vertices[1] = optix::make_float3(5.0f, 0.0f, 0.0f);
vertices[2] = optix::make_float3(0.0f, 5.0f, 0.0f);

// Copy the vertices and indices to OptiX device buffers
optix::Buffer vertex_buffer = context->createBuffer(RT_BUFFER_INPUT, RT_FORMAT_FLOAT3);
vertex_buffer->setSize(sizeof(vertices));
memcpy(vertex_buffer->map(), &vertices, sizeof(float3) * sizeof(vertices));
vertex_buffer->unmap();

optix::Buffer index_buffer = context->createBuffer(RT_BUFFER_INPUT, RT_FORMAT_INT3);
index_buffer->setSize(1);
```

```
21 memcpy(index_buffer->map(), &indices, sizeof(indices));
22 index_buffer->unmap();
23
24 // Set the buffers on the context object, so they can be referenced on the device
25 context["vertex_buffer"]->setBuffer(vertex_buffer);
26 context["index_buffer" ]->setBuffer(index_buffer);
27
28 // Load user programs as a PTX string
29 const char *ptx = sutil::getPtxString("ray_generation.cu");
30
31 // Create ray generation program
32 context->setRayGenerationProgram(0, context->createProgramFromPTXString(ptx, "generate_ray");
33
34 // Create material and assign closest hit program
35 optix::Material mat = context->createMaterial();
36 mat->setClosestHitProgram(0, context->createProgramFromPTXString(ptx, "closest_hit"));
37
38 // Create geometry object and assign intersection and bounding box programs
39 optix::Geometry geometry = context->createGeometry();
40 geometry->setPrimitiveCount(num_triangles);
41 geometry->setIntersectionProgram(context->createProgramFromPTXString(ptx,"tri_intersect"));
42 geometry->setBoundingBoxProgram(context->createProgramFromPTXString(ptx, "tri_bounds"));
43
44 // Create geometry instance and assign material
45 optix::GeometryInstance gi = context->createGeometryInstance(geometry, material);
46
47 // Specify acceleration structure
48 gi->setAcceleration(context->createAcceleration("Trbvh"));
49
50 // Specify the root of the object hierarchy as a variable
51 context["top_object"]->set(gi);
52
53 Buffer output_buffer = context->createBuffer(RT_FORMAT_OUTPUT, RT_FORMAT_UNSIGNED_BYTE4);
54 context["output_buffer"]->set(output_buffer);
55
56 // Declare other device-side variables
57 context["bg_colour"]->setFloat(0.34f, 0.55f, 0.85f);
58
59 // Launch context with 1024x1024 2D grid structure
60 unsigned int width = 1024, height = 1024;
61 context->launch(0, width, height);
62
63 // Fetch output data and convert to image
```

```
64 float3 *output = static_cast<float3>(output_buffer->map());
65 rgbBufferToImage(output);
66 output_buffer->unmap();
```

Listing B.6: An OptiX host code sample, demonstrating how a simple triangle geometry is created as buffers on the device, how device programs and acceleration structures are assigned, how data buffers are copied to and from the device, and how the OptiX context is launched.

```
1  optix::GeometryInstance geom_instance;
2
3  #if OPTIX_VERSION / 10000 >= 6
4  optix::GeometryTriangles geom_tri = ctx->createGeometryTriangles();
5  geom_tri->setPrimitiveCount(mesh.num_triangles);
6  geom_tri->setTriangleIndices(mesh.tri_indices, RT_FORMAT_UNSIGNED_INT3);
7  geom_tri->setVertices(mesh.num_vertices, mesh.vertices, mesh.positions->getFormat());
8  geom_instance = ctx->createGeometryInstance();
9  geom_instance->setGeometryTriangles(geom_tri);
10 #else
11 optix::Geometry geom = ctx->createGeometry();
12 geom->setPrimitiveCount(mesh.num_triangles);
13 geom->setBoundingBoxProgram(createBoundingBoxProgram(ctx));
14 geom->setIntersectionProgram(createIntersectionProgram(ctx));
15 geom_instance = ctx->createGeometryInstance(geometry);
16 #endif
17
18 return geom_instance;
```

Listing B.7: An example of conditionally using the OptiX `GeometryTriangles` API when available, otherwise falling back to the standard `Geometry` API.

Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC

Justin Lewis Salmon, Simon McIntosh Smith
Department of Computer Science
University of Bristol
Bristol, U.K.
{justin.salmon.2018, S.McIntosh-Smith}@bristol.ac.uk

***Abstract*—OpenMC is a CPU-based Monte Carlo particle transport simulation code recently developed in the Computational Reactor Physics Group at MIT, and which is currently being evaluated by the UK Atomic Energy Authority for use on the ITER fusion reactor project. In this paper we present a novel port of OpenMC to run on the new ray tracing (RT) cores in NVIDIA's latest Turing GPUs. We show here that the OpenMC GPU port yields up to 9.8x speedup on a single node over a 16-core CPU using the native constructive solid geometry, and up to 13x speedup using approximate triangle mesh geometry. Furthermore, since the expensive 3D geometric operations required during particle transport simulation can be formulated as a ray tracing problem, there is an opportunity to gain even higher performance on triangle meshes by exploiting the RT cores in Turing GPUs to enable hardware-accelerated ray tracing. Extending the GPU port to support RT core acceleration yields between 2x and 20x additional speedup. We note that geometric model complexity has a significant impact on performance, with RT core acceleration yielding comparatively greater speedups as complexity increases. To the best of our knowledge, this is the first work showing that exploitation of RT cores for scientific workloads is possible. We finish by drawing conclusions about RT cores in terms of wider applicability, limitations and performance portability.**

***Index Terms*—HPC, Monte Carlo particle transport, ray tracing, GPUs**

I. Introduction

Particle transport algorithms simulate interactions between particles, such as nuclear fission and electron scattering, as they travel through some 3D geometric model. The Monte Carlo method can be applied to particle transport, using random sampling of particle trajectories and interactions to calculate the average behaviour of particles, producing highly accurate simulation results compared to deterministic methods. Monte Carlo particle transport has found applications in diverse areas of science such as fission and fusion reactor design, radiography, and accelerator design [1], [2]. The Monte Carlo particle transport algorithm is highly computationally intensive, partly due to the large number of particles which must be simulated to achieve the required degree of accuracy, and partly due to certain computational characteristics of the underlying algorithm that make it challenging to fully utilise modern computing resources.

Previous studies have successfully proven that Monte Carlo particle transport is well suited to GPU-based parallelism [3]–[5]. Targeting GPUs is becoming increasingly important in HPC due to their proliferation in modern supercomputer designs such as Summit [6].

A. OpenMC

OpenMC is a Monte Carlo particle transport code focussed on neutron criticality simulations, recently developed in the Computational Reactor Physics Group at MIT [7]. OpenMC is written in modern C++, and has been developed using high code quality standards to ensure maintainability and consistency. This is in contrast to many older codes, which are often written in obsolete versions of Fortran, and have grown to become highly complex and difficult to maintain. It is partly for this reason that the UK Atomic Energy Authority (UKAEA) is currently evaluating OpenMC as a tool for simulating the ITER nuclear fusion reactor [8].

OpenMC currently runs on CPUs only, using OpenMP for on-node parallelism and MPI for inter-node parallelism, and has been well studied from a performance perspective [9]. The first major piece of this work will present a novel port of OpenMC to NVIDIA GPUs, hypothesising that significant on-node performance improvements can be obtained over the CPU. This has potentially immediate real-world benefits for UKAEA and other institutions seeking improved Monte Carlo particle transport performance. We then go on to explore emerging ray-tracing hardware and its applicability to acceleration in this application area.

B. Hardware-Accelerated Ray Tracing

As the memory bandwidth and general purpose compute capability improvements of recent GPUs begin to plateau, GPU manufacturers are increasingly turning towards specialised hardware solutions designed to accelerate specific tasks which commonly occur in certain types of applications. A recent example of this is the inclusion of Tensor cores in NVIDIA's Volta architecture, which are targeted towards accelerating matrix multiplications primarily for machine learning algorithms [10]. An earlier example is texture memory, designed to improve efficiency of certain memory access patterns in computer graphics applications [11]. These specialised features have often been repurposed and exploited by HPC researchers to accelerate problems beyond their initial design goals.

NVIDIA's latest architecture, codenamed Turing, includes a new type of fixed-function hardware unit called Ray Tracing

(RT) cores, which are designed to accelerate the ray tracing algorithms used in graphical rendering. NVIDIA advertises potential speedups of up to 10x with RT cores, which will supposedly allow computer games designers to bring real-time ray tracing to their games, opening up new levels of photorealism.

The ray tracing algorithms used in graphical rendering are similar in nature to Monte Carlo particle transport algorithms, in the sense that both require large numbers of linear geometric queries to be executed over complex 3D geometric models. Based on this parallel, it is entirely conceivable that RT cores can potentially be utilised for Monte Carlo particle transport. The second part of this paper investigates this concept, by attempting to exploit RT cores to accelerate OpenMC, hypothesising that the particle transport workload can achieve some worthwhile fraction of the 10x speedup obtained by graphical rendering.

The contributions of this paper include:

- The first known study into the use of RT cores for scientific applications.
- An optimised port of OpenMC to GPUs, including detailed performance benchmarking.
- An evaluation of RT cores in terms of wider applicability, limitations and performance portability, intended to facilitate future studies.

II. Background

A. Monte Carlo Particle Transport Efficiency

There are several factors which affect the efficiency of the Monte Carlo particle transport method, most of which relate to utilisation of parallel resources. Load balancing [12], SIMD utilisation [13], random memory access patterns [14], [15] and atomic tallying performance [2] are all cited as general characteristics that affect parallel efficiency, and have all been extensively studied. However, this project will not focus primarily on parallel efficiency, but will instead focus on *geometric efficiency*. When tracking particles travelling through complex geometric models, a large number of queries must be performed on that model, comprising a major proportion of overall runtime [4], [16], [17]. Geometric efficiency is a critical factor in production-grade systems which simulate large scale models (such as the ITER fusion reactor), and optimising this aspect of the Monte Carlo particle transport algorithm could significantly improve overall performance.

B. Geometric Algorithms in Particle Transport

All production-grade Monte Carlo particle transport codes support complex, arbitrary 3D geometric models as the substrate on which to simulate particle interactions. A major part of the simulation process involves knowing exactly where a particle is within the model, computing where it will be in the next timestep given its velocity, and determining which objects it will intersect with along its trajectory. These computations depend on how the model is represented in software.

Geometric Representation: Some codes use a method called constructive solid geometry (CSG) to represent models, which essentially uses a tree of boolean operators (intersection, union, etc.) applied to primitive objects such as planes, spheres and cubes to build precise structures. In this case, intersections between particles and objects are calculated using basic geometric functions on nodes in the tree, based on the primitive type of the shape node and its parameters. CSG representations are relatively lightweight, since only the primitive shape parameters and operator tree need to be stored. Many institutions create model geometries using computer-aided design (CAD) tools, which often use CSG representations due to their accuracy. However, it is often time consuming to create complex models using CSG. Furthermore, engineers are often forced to duplicate the model using the proprietary input formats of each Monte Carlo particle transport code.

Other codes use meshes of small triangles (represented as vertices in 3D space) which produce approximations to objects. Triangle mesh models are generally less accurate than CSG models, although more highly faceted meshes lead to more accurate models. This can result in relatively high storage costs if precise models are required. However, it is often easier to create complex models using triangle meshes compared to CSG models. Intersection testing on triangle meshes can be expensive, requiring potentially many triangles to be tested before finding the exact intersection point. There are several industry-standard formats for storing triangle mesh data which are widely supported.

Ray Tracing: The geometric queries in particle transport simulation can be formulated using techniques from computer graphics. Ray tracing can be used to determine which volume a particle lies within and which surface it will intersect with given its current direction. Figure 1 shows how a particle (or "ray") can be cast along a straight trajectory from a point in the plane through a two dimensional object to determine whether the point is inside the object. If the total number of intersections is *odd* after the trajectory is extended to infinity, the particle must lie within the object. If the intersection count is *even*, the particle is outside the object. The closest intersection point then simply becomes the first intersection along the trajectory. This method is trivially extended to three dimensions, and can be used with both CSG and triangle mesh geometries.

Acceleration Structures: For large models, finding the intersection points is an expensive process. The graphics industry has dedicated a significant effort into optimising this kind of operation through the development of acceleration structures, which use a hierarchy of progressively smaller bounding boxes around model sub-regions. It is these boxes which are then tested for intersection in a binary tree style search, massively reducing the number of surfaces that need to be tested. The most prevalent types of acceleration structure are Bounding Volume Hierarchy (BVH) trees [18], Octrees [19] and Kd-trees [20], each of which contains tradeoffs between lookup performance, build time and runtime memory overhead. Figure 2 shows how a simple BVH tree is constructed.

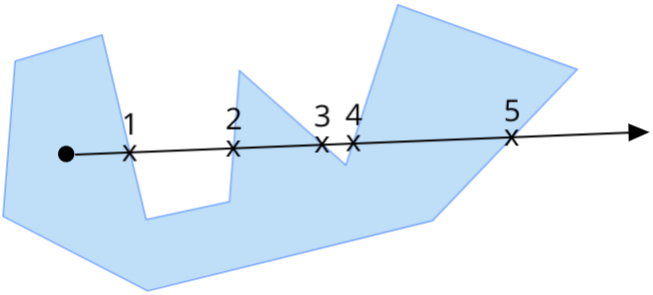


Fig. 1: Using ray tracing to solve the point-in-polygon problem.

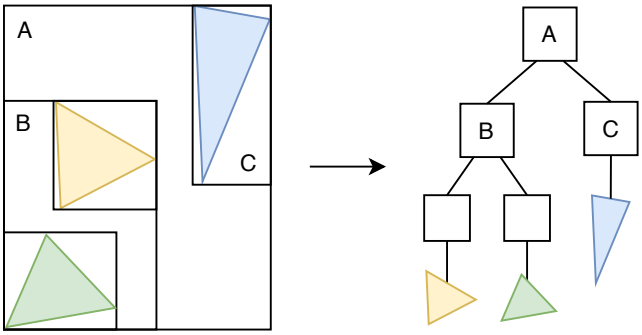


Fig. 2: Construction of a BVH tree by recursively organising bounding boxes of decreasing size.

C. The NVIDIA Turing Architecture

Triangle meshes are the prevalent representation in the graphics world, and the Turing architecture has been designed to provide specific support for using them with ray tracing algorithms. RT cores offload the critical part of a ray tracing loop; traversal of acceleration structures (specifically BVH trees) and ray-triangle intersection tests. Each streaming multiprocessor (SM) on the GPU has access to its own RT core to which it can issue "ray probe" requests. Each RT core is made up of a triangle intersection unit and a BVH traversal unit, and is supposedly able to cache triangle vertex and BVH tree data, although the exact layout and functionality of the cores is not publicly known. The two units in the RT core execute the ray probe asynchronously, writing the result back to an SM register once complete. Consequently, the SM is able to perform other work while ray tracing operations are happening, saving potentially thousands of instruction cycles. Existing RT core benchmarks claim up to 10x speedup on pure graphical rendering workloads, given ideal conditions [21] [22].

The Turing architecture also contains a number of generational improvements over the Pascal and Volta architectures, a detailed analysis of which can be found in [23].

D. OptiX

Unlike with Tensor cores, NVIDIA does not provide an API to allow developers to issue instructions to RT cores directly. Instead, support for them has been added to three existing ray tracing libraries: Microsoft's DXR [24], Khronos Group's Vulkan [25] and NVIDIA's OptiX [26]. Developers wishing to make use of RT cores must do so through one of these libraries. This work will focus on the OptiX library, due to its maturity and alignment with the familiar CUDA ecosystem.

OptiX is highly suited to graphical rendering applications in terms of API design, although it presents itself as being flexible enough to handle other types of application. This is true to an extent, provided the application is willing and able to adhere to the constraints of the API in terms of architectural design and flexibility. Those familiar with CUDA programming may find the OptiX development process somewhat restrictive. The user provides a set of CUDA-like kernel programs to the OptiX host API as PTX strings, each of which performs a specific function in the ray tracing pipeline, such as generating rays, handling intersections or handling rays which miss the geometry entirely. These programs are then compiled on-the-fly by OptiX and weaved into a single "mega kernel". OptiX then handles scheduling of kernel launches internally, automatically balancing load across the GPU.

RTX Mode: RT core acceleration is enabled in OptiX via the *RT_GLOBAL_ATTRIBUTE_ENABLE_RTX* setting, provided certain prerequisites are met. Only geometries defined using the `GeometryTriangles` API which use the default intersection and bounding box programs are eligible to use the RT core's triangle intersection unit (CSG models are not eligible). BVH trees are the only acceleration structure supported by the traversal unit. RTX mode also enables a new compilation pipeline introduced in version 6, which is designed to be more efficient on GPUs without RT cores.

E. Related Work

The promise of up to 10x speedup for ray tracing on RT cores is enticing, despite the drawbacks of needing to use the OptiX API to achieve it. To explore the possibilities for Monte Carlo particle transport, a suitable code was sought which can be representative in terms of real-world scale and functionality, in order to obtain realistic comparisons. Several other studies have attempted to use ray tracing techniques for particle-based applications, which we briefly review here in order to motivate the choice of code to study.

There have been a mixture of approaches focussing on both CPUs and GPUs, exploring different types of geometric mesh representations and acceleration structures. In 2017, Bergmann et. al. presented results for the WARP code, a GPU-based Monte Carlo neutron transport code which uses OptiX to define CSG models accelerated with a BVH tree [4], [16] and to perform intersection tests. Bergmann's code compared well with existing non-ray-traced CPU implementations. However, it does not use triangle mesh models, thus making it unsuitable for RT core acceleration. We did not choose to study the WARP code further, since it is not considered production-grade. Nevertheless, Bergmann's work sets important precedents for the use of both ray tracing and the OptiX library for Monte Carlo particle transport.

In 2018, Thomson claims to have implemented the first GPU-based astrophysics simulation code to use a ray-traced triangle mesh geometry accelerated with a BVH tree as part of the TARANIS code [27]. Thomson presented excellent performance results for simple test cases compared to existing non-ray-traced CPU-based astrophysics codes, although his implementation did not scale to more realistic test cases. This appears to be due to complexities with ionised particle behaviour which can occur with astrophysics simulations, which fortunately do not appear in most Monte Carlo particle transport simulation as commonly only neutral particles (such as neutrons or photons) are simulated. Despite being in quite a different algorithmic category, much insight can be drawn from Thomson's work, namely the further important precedents which are set for the use of both triangle mesh geometries and BVH trees for a particle-based code. However, the code itself is not available for direct study.

In 2010, Wilson et. al. published their work on the DAGMC toolkit [28], a generic CPU-based triangle mesh geometry traversal code that can be plugged in to a number of production-grade Monte Carlo particle transport codes such as MCNP [29], OpenMC [30] and SERPENT [31]. DAGMC replaces the native geometry representation with its own ray-traced triangle mesh geometry based on the MOAB library [32] accelerated with a BVH tree. Wilson's work was primarily focussed on conversion of CAD models to implementation-agnostic triangle meshes in an attempt to reduce the amount of effort required to port models between different codes. Initial performance results on complex models were adequate, although subsequent work by Shriwise in 2018 significantly improved performance on the MCNP code through the use of a more efficient BVH implementation [33]. It is important to note that although DAGMC runs only on CPUs, it proves the feasibility of using ray tracing over triangle mesh geometries with production-grade Monte Carlo particle transport codes.

Institutions which make use of Monte Carlo particle transport are increasingly becoming interested in triangle mesh geometries, and the DAGMC toolkit is a promising route to enable simulations to be performed on existing CAD models, potentially reducing the engineering burden of defining models multiple times in the specific input format of each code [28], [34].

F. OpenMC

OpenMC natively uses CSG representations, and does not use ray tracing for geometry traversal (although the DAGMC plugin for OpenMC allows ray tracing over triangle meshes). OpenMC has been well studied from a performance perspective [9] and uses OpenMP and MPI for parallelism. However it has not been ported to make use of GPU parallelism. Several other Monte Carlo particle transport codes have gained significant performance improvements from GPUs recently [3], [4], [5] therefore it is reasonable to assume that OpenMC may also benefit in similar ways.

OpenMC has been selected as the candidate for this study, as it is a freely available open-source implementation, unlike many other codes such as MCNP and SERPENT which are export restricted due to their sensitive links to nuclear research. OpenMC will be ported to the GPU, with support added for ray tracing over triangle meshes using the OptiX library (as well as the native CSG geometry). Following that, the implementation will be further extended to support RT core acceleration.

G. Summary

Existing work has proven that ray tracing techniques can improve the performance of particle-based scientific codes, particularly on GPUs, and that triangle mesh geometries and BVH trees are both feasible approaches. However, there is a clear opportunity to extend this work to utilise the hardware-accelerated ray tracing capabilities of the Turing GPU architecture. There are currently no published studies investigating this opportunity. This work seeks to address that gap by porting OpenMC to the GPU, using the OptiX library to replace the native geometry representation with a ray-traced triangle mesh backed by a BVH tree, enabling the exploitation of RT cores on a Turing-class GPU for the first time.

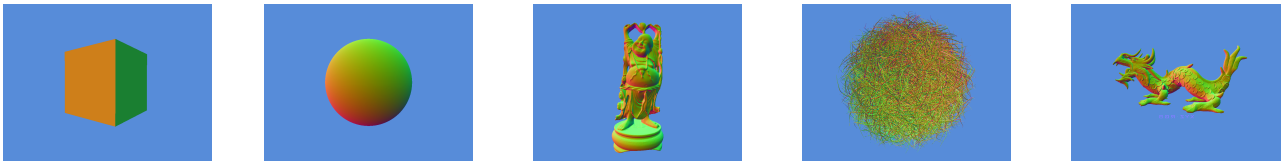
III. RT Core Benchmarking

Before attempting to directly exploit RT cores for Monte Carlo particle transport, we first measure their performance on the type of graphics workload for which they were designed. This helps to gain an understanding of their characteristics, and to form a baseline performance profile to work towards.

A simple benchmarking tool was developed to evaluate the raw ray tracing performance of RT cores, based on a sample from the OptiX library. The benchmark renders a fixed number of frames of a 3D triangle mesh scene as fast as possible. Each frame is rendered by launching a fixed number of primary rays in a 2D pixel grid orientation, using a very simple shading routine to maximise the ratio of ray tracing work to other compute work. Each thread handles a single ray and writes the computed pixel colour to an output buffer, which is then interpreted as an image. The benchmark supports both the standard OptiX geometry API, as well as the `GeometryTriangles` API which offloads intersection testing to RT cores when RTX mode is enabled.

A. Method

A number of standard 3D graphics models were selected to use as rendering targets, spanning a wide range in terms of triangle count and topological complexity, with the intention of identifying different performance characteristics. The simplest model contains a trivial 12 triangles, while the largest contains over 7 million triangles. Most are single-cell watertight models, with the exception of the Hairball model which is a tightly packed weave of thin cells. Figure 3 shows 2D renderings of each of the five chosen models, as rendered by the RT core benchmark tool. The scene viewpoint location (i.e. the origin from which rays are cast) is maintained, and the number of rendering iterations is maintained at 10^4. The



(a) Cube, 12 (b) Sphere, 89k (c) Happy Buddha, 1.1m (d) Hairball, 2.9m (e) Asian Dragon, 7.2m

Fig. 3: Renderings of the five 3D models produced by the RTX benchmark tool, with approximate triangle counts.

launch dimensions (which directly correspond to output image resolution) are varied from 960x540 doubling each time up to 15360x8640, roughly corresponding to the range of a 540p SD image up to a 16k Ultra HD image. This is repeated for each model, thus allowing us to investigate the effects of varying both model complexity and launch dimensionality independently.

The main metric used to measure performance is *rays cast per second*, calculated as the total number of rays cast divided by the total runtime duration, measured in GigaRays per second.

B. Results: Model Complexity

Figure 4 shows how each model behaves at a launch resolution of 15360x8640 on a Turing-class RTX 2080 Ti GPU and a Pascal-class GTX 1080 Ti. Note that enabling RTX mode on the Pascal GPU is a valid configuration despite the lack of RT cores, as it will still use the improved OptiX execution pipeline.

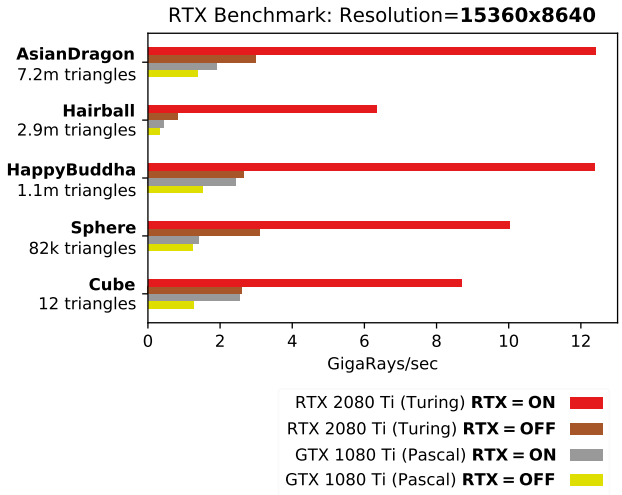


Fig. 4: Results of the RTX-enabled graphics benchmark on each of the tested models and GPUs for a fixed launch resolution of 15360x8640. Figures in GigaRays/s, higher is better.

Using RTX mode on the Turing GPU yields a 4.6x speedup on average at this resolution. A fraction of this is attributable to the optimised execution pipeline in OptiX 6, which can be seen when comparing the two results on the Pascal GPU. Most of the speedup however is due to the RT cores themselves. The benchmark is able to produce over 12 GigaRays/sec for the Happy Buddha and Asian Dragon models, which corroborates NVIDIA's own benchmarks for these two models [**?**]. The result profiles for these models appears quite similar, despite being almost an order of magnitude apart in terms of triangle count. This suggests that both are ideal workloads, and are probably reaching the optimal throughput of the RT cores.[1]

It is clear to see that the number of triangles alone does not dictate performance. The Hairball model is clearly the heaviest workload, despite not being the largest model in terms of triangle count. It is 11.8x faster with RTX mode on the Turing GPU. This suggests that the higher topological complexity of the model is having a significant impact, most likely because its multi-volume topography results in a higher number of potential intersections along a single ray path, thereby requiring a greater number of triangle vertex buffer lookups to complete the full traversal. This then becomes the dominant performance factor. Additionally, the generated BVH tree is likely to be deeper and therefore slower to search.

The reason why the Cube and Sphere models are slower than the larger Asian Dragon and Happy Buddha models under RT core acceleration is simply because ray-triangle intersection testing and BVH traversal operations comprise less of the overall runtime due to the simplicity of the models. As the models get larger, the overall runtime increases proportionally and the ray tracing operations become the dominant factor. The Hairball model shows the extreme case of this, where ray tracing takes such a large amount of effort that runtime increases to the detriment of ray casting throughput.

The Turing GPU also outperforms the Pascal GPU with RTX mode disabled. This simply suggests that the generational improvements of the Turing architecture are providing some benefit for this particular type of workload, such as the larger L1 and L2 caches permitting better locality for triangle vertex data.

C. Results: Launch Dimensions

Figure 5 shows how the ray tracing throughput changes as the launch dimensions are varied for the Happy Buddha model. For this model, the peak speedup is 6.4x, which occurs at the 1920x1080 resolution. The other four models exhibit essentially the same behaviour, so are omitted here for brevity. There is clearly a sublinear scaling profile as launch

[1]It is perhaps not surprising that these models were used to produce the advertised performance numbers.

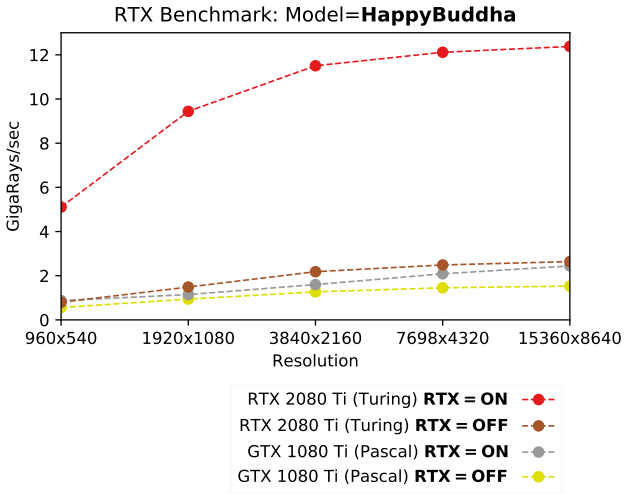


Fig. 5: Results of the RTX-enabled graphics benchmark for the Happy Buddha model as the launch dimensions are increased.

dimensions are increased, regardless of GPU architecture. This is most likely a result of the way that OptiX schedules kernel launches and automatically balances load over the GPU, with larger launch sizes leading to more effective balancing and higher occupancy. The peak speedup obtained overall is 15.6x for the Hairball model, which occurs at the 960x540 resolution.

D. Profiling

In order to gain a deeper understanding of how the GPU is being utilised, the NVIDIA Visual Profiler was used to profile the running benchmark. Unfortunately, at the time of writing, it is not possible to profile kernels running on Turing-class GPUs, due to support not yet having been added for Compute Capability 7.5. It is also not possible to profile with RTX mode enabled on any GPU, due to the new kernel mangling performed by OptiX leading to `nvprof` being unable to instrument the main kernel. Furthermore, OptiX makes things even more challenging to profile, since it combines all user kernels into one "mega kernel", resulting in line information being unavailable. Nevertheless, some insight can still be gleaned by profiling on the Pascal GPU with RTX mode disabled, combined with a certain amount of speculation about what might be occurring on the RT cores.

Performance is limited by memory bandwidth in all cases. It might be reasonable to initially guess that the smaller Cube and Sphere models would not be affected by memory bandwidth issues, since the models are easily small enough to fit entirely in L1/L2 cache. However, this is not the case when taking into account other factors such as output buffer writes, BVH traversal and internal bookkeeping done by OptiX.

E. Summary

We have managed to reproduce the performance numbers advertised by NVIDIA, and have found that there is a performance sweet spot in terms of model complexity and launch dimensionality. RT cores are indeed a powerful addition to the Turing architecture, and the results presented here serve as solid motivation for further investigating their potential use for Monte Carlo particle transport.

IV. OpenMC GPU Port

We now begin our efforts to exploit RT core acceleration for Monte Carlo particle transport. To the best of our knowledge, this is the first time that anyone has ported this class of application to ray tracing hardware. The first stage is to port OpenMC to the GPU and examine it from a traditional HPC perspective, initially ignoring RT cores, enabling an understanding of its overall performance characteristics to be formed and for comparisons to be drawn against existing CPU-based versions using both CSG and triangle mesh geometries. The second stage is to extend the GPU port to support RT core acceleration, to understand how well the particle transport workload maps onto the paradigm.

A. Method

The main particle transport kernel of OpenMC was ported to CUDA, using the OptiX API on the device to trace rays and store quantities in output buffers. On the host side, OptiX is used to create a triangle mesh geometry loaded from an external model file in OBJ format, and to handle device buffer movement. OptiX does not allow specifying kernel block and grid sizes directly, but instead takes a one, two or three dimensional launch parameter and manages kernel launches internally. We use the number of simulated particles as a one-dimensional launch parameter, and execute one OptiX context launch per generation/batch. A number of simple optimisations were made to the GPU port, such as aligning structure members and switching to single-precision arithmetic.

While it would be desirable to test the code on a model more related to Monte Carlo particle transport (such as a fission reactor), we do not currently have access to such models, so we use the same five models from the RTX benchmark. The model is filled with a fissionable material (^{235}U) and is surrounded by a bounding cube of 5^3 filled with a vacuum. Boundary conditions were set on the bounding cube so that particles are terminated if they hit the edge of the bounding cube. The particle source is set inside the model, using the default strength parameters. Each model was simulated for 2 generations using 2 batches of N particles, where N ranges in powers of 10 from 10^3 up to 10^7.

The GPU port was tested on each of the models, using the same two GPUs as before. For comparison, OpenMC was tested natively on a single node containing a 16-core AMD Ryzen 7 2700 CPU (using GCC 7.4). Since OpenMC natively uses CSG representation, only the Cube and Sphere models can be compared, as it is not feasible to define the other geometries using CSG due to their complexity. For further comparison, the DAGMC plugin was also tested, which uses CPU-based ray tracing over triangle meshes.

We collect the particle calculation rate (measured in particles/sec) and wallclock time spent in particle transport (ignoring initialisation and finalisation) as the main metrics for evaluation.

V. Results: GPU vs CPU

A. Model Complexity

Figure 6 shows the range in calculation rate performance on the Sphere and Cube models between GPU and CPU versions, for a launch size of 10^6 particles.

The GPU versions are significantly faster on average than the CPU on both models. In this case, the fastest GPU version is 13x faster than the native CPU version, and 47.5x faster than the DAGMC CPU version. It is not necessarily fair to compare CPU performance on a single node, since OpenMC is capable of scaling to thousands of processors, but it is nevertheless useful to get a sense of on-node scale.

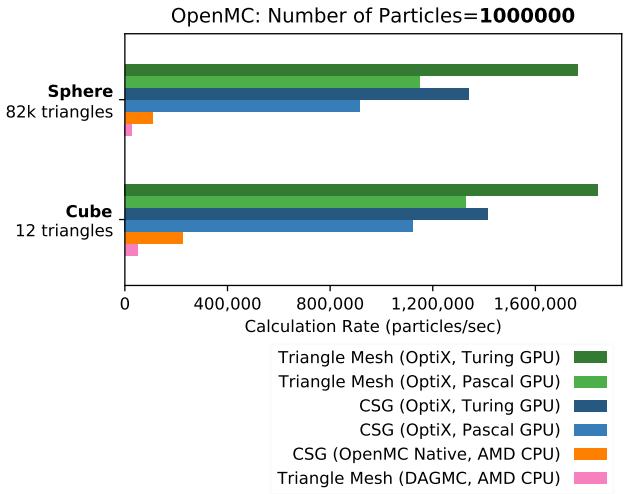


Fig. 6: Range in particle calculation rate between CPU and GPU versions for the Sphere and Cube models with a particle count of 10^6. Higher is better.

The increased triangle count of the Sphere model slows down the native and DAGMC versions quite significantly compared to the Cube model, whereas the GPU versions show much less of a dependency on triangle count. Note that the native CPU version is slower on the Sphere model simply because the CSG sphere intersection calculation is more complex than the CSG cube intersection. The DAGMC version is the least performant, suggesting the efficiency of its ray tracing implementation over triangle meshes is comparatively poor.

A secondary observation from Figure 6 is the difference in performance between Turing and Pascal GPUs, with the former being 1.4x faster than the latter. This is most likely to be evidence of the memory subsystem improvements in the Turing architecture.

B. Launch Dimensions

Figure 7 shows how the calculation rate varies as the number of simulated particles is scaled up on the Sphere model. The triangle mesh geometry on the GPU is the fastest in all cases, being 1.4x faster than the GPU CSG geometry.

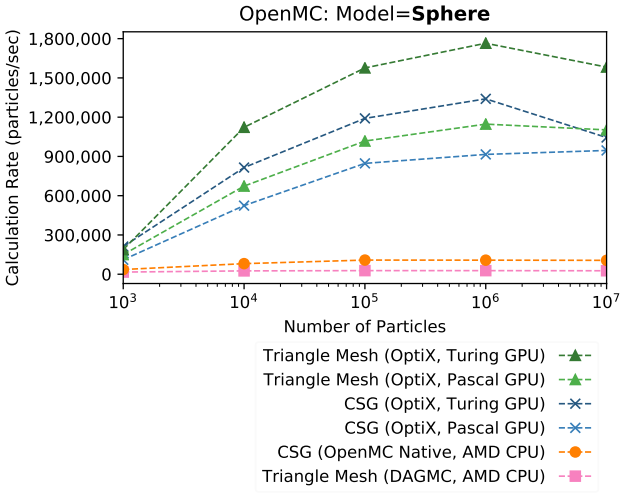


Fig. 7: Comparison between CPU and GPU versions when scaling up the particle count for the Sphere model.

The difference in performance becomes more pronounced as the number of particles is increased, with the GPU versions appearing to peak in performance at 10^6 particles. Peak speedup is 16.4x over native CPU. This peak suggests that one or more resources on the GPU are being most efficiently used at that scale, and begin to deteriorate at larger scales. As discussed previously in Section III-C, this is most likely a result of OptiX managing kernel grid and block sizes.

The CPU versions appear to have reasonably consistent calculation rates, regardless of number of particles, suggesting that the CPU throughput is simply saturated and cannot process any faster without added parallelism, but does not deteriorate in performance.

VI. Results: RT Core Acceleration

Having seen the performance improvements brought by porting OpenMC to the GPU, we can now move on to the task of exploiting RT core acceleration. As described in Section II-D, the RTX mode available with the OptiX 6 API enables RT core accelerated ray tracing on Turing GPUs, as well as a more efficient execution pipeline that simulates RT cores in software on older GPU architectures. The following sections present and discuss the results of the model complexity and particle count experiments for the extended OpenMC GPU version which supports RTX mode, as well as the original GPU versions.

At this point we will depart from the native CPU version and the CSG GPU version, since we will be benchmarking against triangle meshes that are not possible to define using the

constructive solid geometry format provided by OpenMC. We will also depart from the DAGMC version, since it is in a much lower performance category and therefore not significantly valuable to consider any further.

A. Model Complexity

Figure 8 shows the calculation rates for each of the five models for a single launch of 10^6 particles. RTX mode on the Turing GPU is the fastest in all cases, being 6.0x faster on average than without. On the Pascal GPU, RTX mode is 1.6x faster on average. The Hairball model shows the biggest difference, being 20.1x faster with RTX mode on Turing.

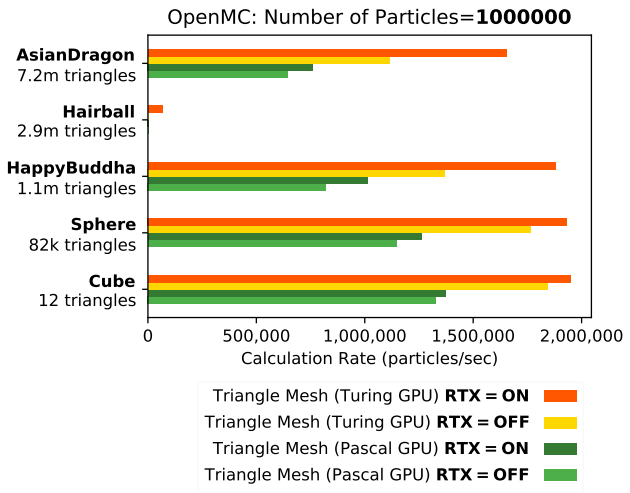


Fig. 8: Effects of RTX mode (which uses RT cores on the Turing GPU) on all models for a particle count of 10^6.

The number of triangles has a clear and simple effect on the calculation rate for the single solid volume models, which is all except the Hairball model. The Hairball model exhibits the most dramatic behaviour due to its high topological complexity, which has an even greater effect than on the graphics benchmark. This is because of the way the point-in-volume and boundary distance algorithm works; it traces a ray iteratively over all intersections at each layer of the model until it misses. For the single cell models, most of the time there are only two hits to escape the model (one to escape the cell, then one to escape the bounding cube). This means that the ray tracer does not have to work very hard. For the Hairball model, there are potentially many more surface hits before the bounding cube is reached, meaning the ray tracer has to work harder to calculate the full path. This is in contrast to the graphics benchmark, which stops once the first surface intersection occurs. RT core acceleration is actually 20.1x faster in this case, allowing its superior ray tracing speed to show over the software implementation as ray tracing completely dominates the workload.

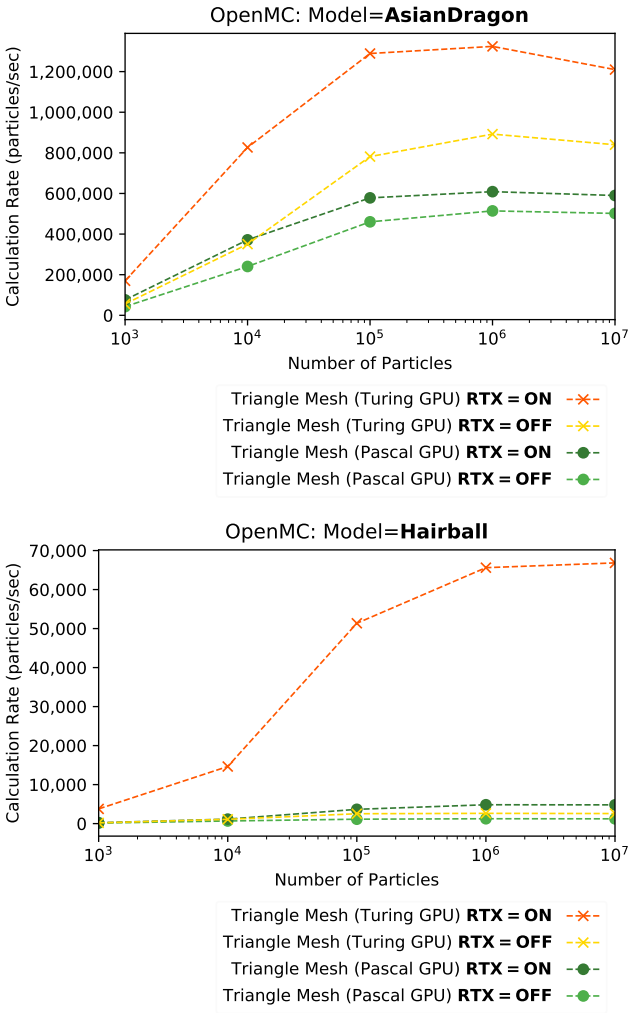


Fig. 9: Effects of RTX mode when the particle count is scaled up for the Asian Dragon and Hairball models.

B. Launch Dimensions

Figure 9 shows how the calculation rate varies as the number of simulated particles is scaled up for the Asian Dragon and Hairball models respectively. These two models represent the most realistic examples in terms of scale. The same performance peak at 10^6 particles (as seen in Section V-B) is visible with RT core acceleration enabled. This again is likely to be an artifact of the way OptiX internally manages kernel launch dimensions to balance load, and that there is an optimal input problem size past which performance begins to deteriorate.

VII. Profiling

As mentioned in Section III-D, the available profiling tools do not yet support the Turing architecture, nor do they support profiling of RTX mode kernels with OptiX 6. We attempt here to use the available profiling data to speculate on how the main transport kernel might be behaving on the Turing GPU.

A. Occupancy

The kernel uses 80 registers per thread by default, which leads to an acceptable (albeit not spectacular) occupancy of 37.2% regardless of model. This is expectedly lower than the RT core benchmark, due to the greatly increased compute workload. An attempt was made to reduce the number of registers at compile time in order to improve occupancy, however this did not result in any significant performance improvement, suggesting that the OptiX scheduling mechanism is performing well in this scenario.

B. Memory Bandwidth

This code is memory bandwidth/memory latency bound in general, which is expected for this type of code. There is a noticeable drop in memory bandwidth throughput and utilisation as the models get larger, as shown in Figure 10. This is likely due to a number of possible reasons:

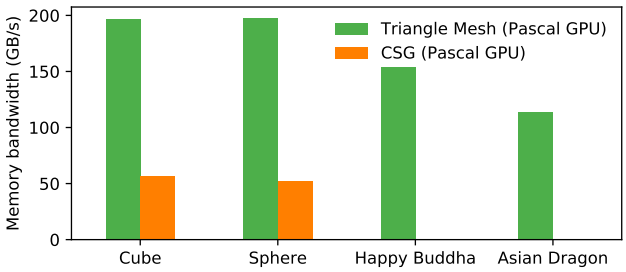


Fig. 10: Variation in memory bandwidth throughput for different models. Note that bandwidth for the Hairball model could not be obtained, as it caused the profiler to crash due to its complexity.

1) Vertex/normal buffer lookup: As the memory footprint of the kernel increases with model size, cache efficiency is reduced for the triangle mesh models as more vertex and normal data must be fetched from main memory during ray traversal. The Cube and Sphere models require 144 Bytes and 984 KBytes respectively, so can easily fit in L1/L2 cache, while the Buddha and Dragon models must undergo some amount of cache rotation at 13.2 MBytes and 82.4 MBytes respectively. This extra data movement puts further pressure on the memory subsystem. For CSG models, there is no vertex buffer lookup, as geometry traversal is done purely based on the boolean operator tree and primitive shape parameters.

2) BVH tree traversal: As the model size increases, the ray tracing algorithm has more intricate BVH hierarchies to build and traverse. This results in further memory lookups and pressure on the memory subsystem.

C. Additional Factors

1) Tally writes: Each simulated particle maintains its own set of tallies, which are written into global tally buffers indexed by particle number. While this approach means that atomic writes are not necessary, the incoherent and stochastic movement of particles leads to a random write pattern on those buffers. This effect becomes more detrimental to performance as the launch dimensions are increased.

2) Particle banks: Particles are initialised on the host and loaded onto the device at the beginning of the simulation. Each thread reads a particle to simulate based on its launch index. Subsequently, as particles collide with other particles in the material, new particles are created and stored in the fission and secondary particle banks. Reads introduce an additional source of memory pressure which scales with launch dimensionality.

3) Workload imbalance and thread divergence: Due to the random nature of particle histories, some particles may survive longer than others, introducing thread imbalance. This has been observed in other codes, such as [35] and [36]. Fortunately for this implementation, the OptiX scheduling mechanism does a reasonably good job of minimising the impact of the imbalance, being able to ensure that short-lived threads are efficiently replaced, thereby maintaining a good percentage of theoretical occupancy.

VIII. Conclusions and Future Work

In terms of on-node performance, the speedups of between ~10x on CSG models and ~33x on RT core accelerated triangle meshes presented here show that the Turing architecture is a formidable platform for Monte Carlo particle transport. Our results compare competitively with other GPU-based ports of Monte Carlo particle transport codes such as [35] and [36] which observed ~3x and ~2.7x speedups respectively using similar methods on older GPU architectures.

A. Wider Applicability of RT Cores

A secondary goal in this paper was to gain an understanding, through implementation, of the kinds of scientific workloads which might be suitable candidates for RT core acceleration. It is clear from the OpenMC results that Monte Carlo particle transport is indeed a natural candidate to benefit from the accelerated BVH traversal and triangle intersection testing brought by RT cores, as hypothesised.

Any algorithm which spends a significant amount of its runtime tracing large numbers of straight lines through large 3D triangle mesh geometries should be able to exploit RT cores. If that is not the case, or the code cannot be reformulated to do so, then RT cores are not going to be useful. Despite this restriction, several potential candidates have been identified. In cancer radiotherapy for example, patient dose simulations are required before treatment, which use 3D biological models. Improving performance in that sector could potentially mean radiologists could provide diagnosis and treatment to patients on a much shorter timescale [37]. Another example can be found in the geoscience field, where seismic ray tracing is used to simulate the movement of seismic waves through the earth in order to gain a representation of the Earth's interior. This technique is useful for analysing earthquakes, and also for oil and gas exploration.

B. Performance Portability

It may be non-trivial for other codes to reap the benefits described above, due to the restrictions placed on how RT cores can be accessed. There are no PTX instructions available for the developer to issue instructions to RT cores (that have been made publicly known), necessitating the use of the OptiX library and all of its ramifications in terms of architectural design. As a result, existing codes would likely have to undergo significant development rework, which may or may not be feasible depending on the amount of investment in the original code and the hardware it has been designed to run on. The upcoming OptiX 7 release is much less opinionated than previous versions and is much closer in terms of API design to vanilla CUDA, which could potentially ease the performance portability burden. Ideally, performance portable enabling languages such as OpenCL would support RT core acceleration, but that is currently not the case.

Furthermore, OpenMC is written in modern C++, and makes extensive use of the C++ STL, polymorphism, and other C++ features. Since many of these language features are not fully supported by CUDA code, a large amount of code duplication and reworking was required to allow OpenMC to run on the GPU. This results in two distinct codebases, which is a major drawback in terms of maintainability. Performance portability is becoming increasingly important in HPC as the range of available compute platforms continues to grow [38], which means that these drawbacks are likely to heavily influence any decisions to port large-scale existing codebases.

There is a definite trend in HPC towards many-GPU designs, such as in the latest Summit [6] supercomputer which contains tens of thousands of Volta-class NVIDIA GPUs. It is conceivable that future supercomputer designs may feature GPUs containing RT cores, thus providing additional motivation to exploit them. It is also conceivable that RT cores may be opened up by NVIDIA in future iterations, and may eventually find support in more performance portable libraries.

C. Future Work

A number of potentially valuable extensions to this work have been identified. Firstly, it would be useful from an experimental perspective to compare the OptiX-based ports of OpenMC with a vanilla CUDA port. This would provide an additional point of comparison, and would allow us to understand to what extent the OptiX library is contributing to the achieved performance results. Secondly, the excellent work surrounding the DAGMC toolkit could be incorporated into the OpenMC GPU port. If models could be loaded from DAGMC files rather than OBJ files, the existing CAD workflows could be reused, removing the need for difficult model conversion stages. Thirdly, the built-in support for multi-GPU execution in the OptiX API could be utilised to further increase on-node performance.

It is useful to note that NVIDIA is not alone in putting dedicated ray tracing hardware in its GPUs - in fact, both Intel and AMD are planning similar fixed-function hardware units in their upcoming GPU architectures. This provides further suggestion that there may be increased motivation to implement support for RT core acceleration in the near future.

D. Summary

A viable, novel and highly performant GPU port of OpenMC has been presented using the OptiX library, which supports both native CSG models and triangle mesh models. Triangle meshes provide the most significant performance improvements compared to CPU-based versions, and the presence of RT cores in the Turing architecture provides impressive additional speedup. We believe ours is the first work to use ray tracing hardware to accelerate Monte Carlo particle transport.

A major factor contributing to performance is the geometric complexity of the model. Specifically, highly topologically complex models with many layers are the most computationally demanding. The speedup from RT core acceleration becomes even more significant at this scale, due to its ability to relieve the SM of large numbers of computation and its internal caching of triangle vertex and BVH tree data.

While further work is required to support the entire OpenMC feature set, the developed code is already being evaluated by the UKAEA for ITER neutronics analysis, and is being considered for inclusion into the official OpenMC repository.

REFERENCES

[1] E. D. Cashwell and C. J. Everett, "A Practical Manual on the Monte Carlo Method for Random Walk Problems," *Mathematics of Computation*, 2006.

[2] N. A. Gentile, R. J. Procassini, and H. A. Scott, "Monte Carlo Particle Transport: Algorithm and Performance Overview," 2005.

[3] X. Jia, P. Ziegenhein, and S. B. Jiang, "GPU-based high-performance computing for radiation therapy," 2014.

[4] R. M. Bergmann and J. L. Vujić, "Monte Carlo Neutron Transport on GPUs," 2014, p. V004T11A002.

[5] A. Heimlich, A. C. Mol, and C. M. Pereira, "GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation," *Progress in Nuclear Energy*, vol. 53, no. 2, pp. 229–239, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.pnucene.2010.09.011

[6] J. Hines, "Stepping up to summit," *Computing in Science and Engineering*, 2018.

[7] P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Annals of Nuclear Energy*, vol. 51, pp. 274–281, 2013.

[8] A. Turner, "Investigations into alternative radiation transport codes for ITER neutronics analysis," in *Transactions of the American Nuclear Society*, 2017.

[9] A. R. Siegel, K. Smith, P. K. Romano, B. Forget, and K. G. Felker, "Multi-core performance studies of a Monte Carlo neutron transport code," *International Journal of High Performance Computing Applications*, 2014.

[10] M. Martineau, P. Atkinson, and S. McIntosh-Smith, "Benchmarking the NVIDIA V100 GPU and tensor cores," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11339 LNCS. Springer Verlag, 2019, pp. 444–455.

[11] N. Ren, J. Liang, X. Qu, J. Li, B. Lu, and J. Tian, "GPU-based Monte Carlo simulation for light propagation in complex heterogeneous tissues," *Optics Express*, 2010.

[12] R. Procassini, M. O'Brien, and J. Taylor, "Load balancing of parallel Monte Carlo transport calculations," *International topical meeting on mathematics and computation, supercomputing, reactor physics and nuclear and biological applications*, 2005. [Online]. Available: https://inis.iaea.org/search/search.aspx?orig_q=RN:40054813

[13] S. P. Hamilton, S. R. Slattery, and T. M. Evans, "Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms," *Annals of Nuclear Energy*, vol. 113, pp. 506–518, 2018. [Online]. Available: https://doi.org/10.1016/j.anucene.2017.11.032

[14] K. Xiao, D. Z. Chen, X. S. Hu, and B. Zhou, "Monte Carlo Based Ray Tracing in CPU-GPU Heterogeneous Systems and Applications in Radiation Therapy," no. June 2015, pp. 247–258, 2015.

[15] M. Martineau and S. McIntosh-Smith, "Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2017-Septe, pp. 498–508, 2017.

[16] R. M. Bergmann, K. L. Rowland, N. Radnović, R. N. Slaybaugh, and J. L. Vujić, "Performance and accuracy of criticality calculations performed using WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," *Annals of Nuclear Energy*, vol. 103, pp. 334–349, 2017.

[17] P. C. Shriwise, A. Davis, L. J. Jacobson, and P. P. Wilson, "Particle tracking acceleration via signed distance fields in direct-accelerated geometry Monte Carlo," *Nuclear Engineering and Technology*, vol. 49, no. 6, pp. 1189–1198, 2017. [Online]. Available: https://doi.org/10.1016/j.net.2017.08.008

[18] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," p. 89, 2013.

[19] T. Karras and Tero, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, pp. 33–37, 2012. [Online]. Available: https://dl.acm.org/citation.cfm?id=2383801

[20] M. Hapala and V. Havran, "Review: Kd-tree traversal algorithms for ray tracing," *Computer Graphics Forum*, vol. 30, no. 1, pp. 199–213, 2011.

[21] M. Nyers, "GPU rendering RTX ray tracing benchmarks - RTX 2080 Ti," 2019. [Online]. Available: http://boostclock.com/show/000250/gpu-rendering-nv-rtxon-gtx1080ti-rtx2080ti-titanv.html

[22] NVIDIA, "NVIDIA Turing GPU," *White Paper*, 2018.

[23] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking," 2019. [Online]. Available: http://arxiv.org/abs/1903.07486

[24] Microsoft Inc, "Announcing Microsoft DirectX Raytracing," 2019. [Online]. Available: https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/

[25] K. Group, "The Vulkan API Specification and related tools," 2019. [Online]. Available: https://github.com/KhronosGroup/Vulkan-Docs

[26] S. G. Parker, A. Robison, M. Stich, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, and K. Morley, "OptiX: A General Purpose Ray Tracing Engine," *ACM Transactions on Graphics*, vol. 29, no. 4, p. 1, 2010.

[27] S. Thomson, "Ray-traced Radiative Transfer on Massively Threaded Architectures," 2018. [Online]. Available: https://www.era.lib.ed.ac.uk/bitstream/handle/1842/31277/Thomson2018.pdf

[28] P. P. Wilson, T. J. Tautges, J. A. Kraftcheck, B. M. Smith, and D. L. Henderson, "Acceleration techniques for the direct use of CAD-based geometry in fusion neutronics analysis," *Fusion Engineering and Design*, 2010.

[29] Monte Carlo Team, "MCNP - A General Monte Carlo N-Particle Transport Code, Version 5," *Los Alamos Nuclear Laboratory*, 2005.

[30] P. K. Romano, "Parallel Algorithms for Monte Carlo Particle Transport Simulation on Exascale Computing Architectures," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[31] A. Turner, A. Burns, B. Colling, and J. Leppänen, "Applications of Serpent 2 Monte Carlo Code to ITER Neutronics Analysis," *Fusion Science and Technology*, vol. 74, no. 4, pp. 315–320, 2018.

[32] T. J. Tautges, "MOAB-SD: Integrated structured and unstructured mesh representation," *Engineering with Computers*, 2004.

[33] P. C. Shriwise, "Geometry Query Optimizations in CAD-based Tessellations for Monte Carlo Radiation Transport," Ph.D. dissertation, University of Wisconsin - Madison, 2018.

[34] A. Badal, I. Kyprianou, D. P. Banh, A. Badano, and J. Sempau, "PenMesh-Monte Carlo radiation transport simulation in a triangle Mesh Geometry," *IEEE Transactions on Medical Imaging*, 2009.

[35] D. Karlsson and Z. Yuer, "Monte-Carlo neutron transport simulations on the GPU," 2018.

[36] S. P. Hamilton and T. M. Evans, "Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code," *Annals of Nuclear Energy*, vol. 128, pp. 236–247, 2019. [Online]. Available: https://doi.org/10.1016/j.anucene.2019.01.012

[37] X. Jia, X. Gu, Y. J. Graves, M. Folkerts, and S. B. Jiang, "GPU-based fast Monte Carlo simulation for radiotherapy dose calculation," p. 18, 2011. [Online]. Available: http://arxiv.org/abs/1107.3355

[38] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," pp. 1–7, 2016. [Online]. Available: http://arxiv.org/abs/1611.07409