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.

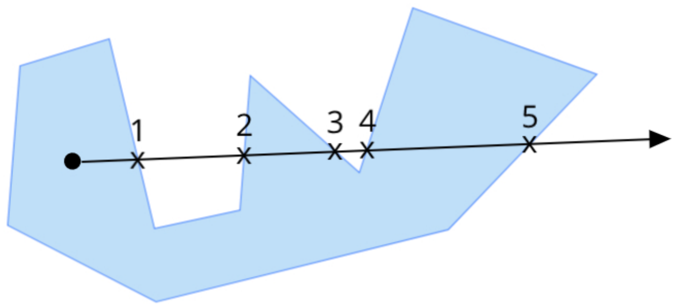


Fig. 1: Using ray tracing to solve the point-in-polygon problem.

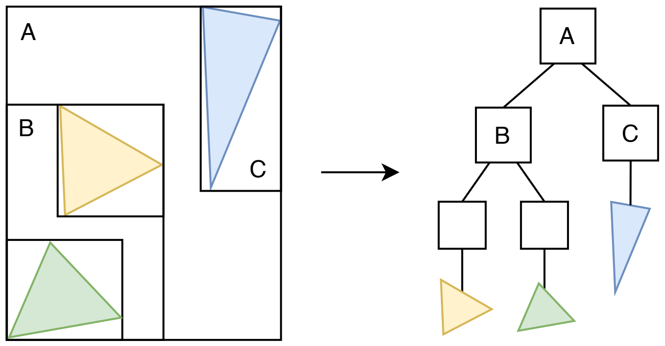


Fig. 2: Construction of a BVH tree by recursively organising bounding boxes of decreasing size.

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.

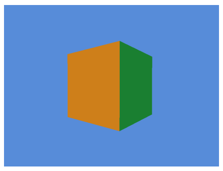
III. RT Core Benchmarking

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.

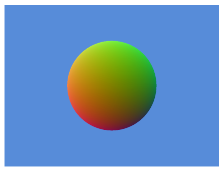
A simple benchmarking tool was developed to evaluate the raw ray tracing performance of RT cores, based on a sample from the OptiX library. The benchmark renders a fixed number of frames of a 3D triangle mesh scene as fast as possible. Each frame is rendered by launching a fixed number of primary rays 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

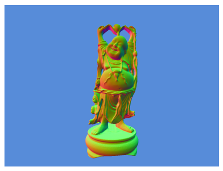
A number of standard 3D graphics models were selected to use as rendering targets, spanning a wide range in terms of triangle count and topological complexity, with the intention of identifying different performance characteristics. The simplest model contains a trivial 12 triangles, while the largest 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



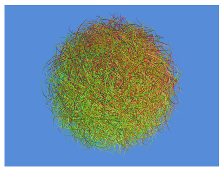
(a) Cube, 12



(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 1080 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.

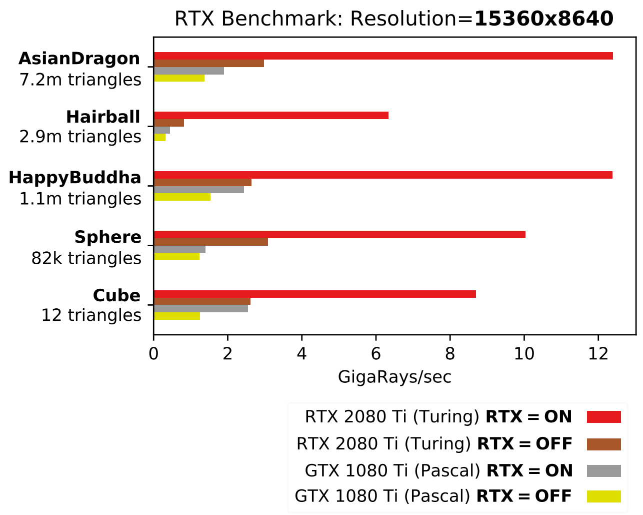


Fig. 4: Results of the RTX-enabled graphics benchmark on each of the tested models and GPUs for a fixed launch resolution of 15360x8640. Figures in GigaRays/s, higher is better.

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.

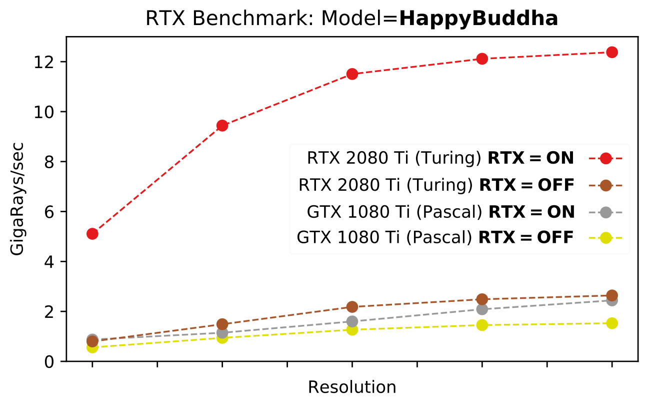


Fig. 5: Results of the RTX-enabled graphics benchmark for the Happy Buddha model as the launch dimensions are increased.

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.

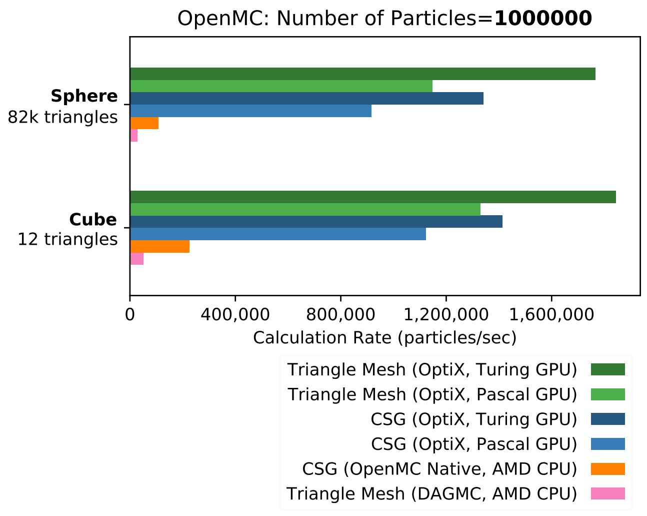


Fig. 6: Range in particle calculation rate between CPU and GPU versions for the Sphere and Cube models with a particle count of 10^6. Higher is better.

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.

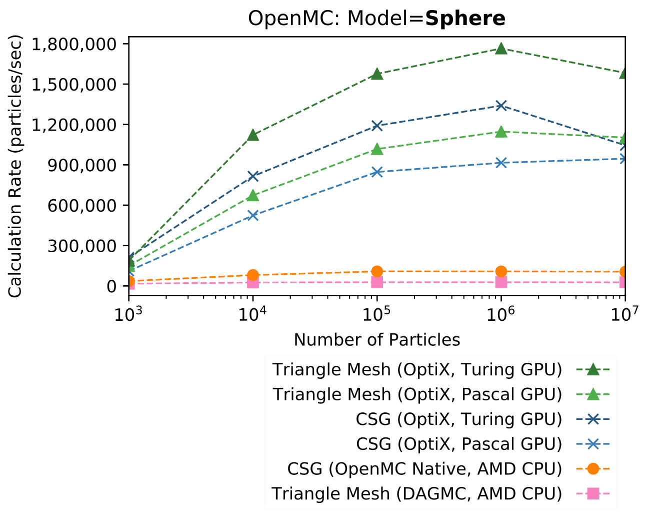


Fig. 7: Comparison between CPU and GPU versions when scaling up the particle count for the Sphere model.

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.

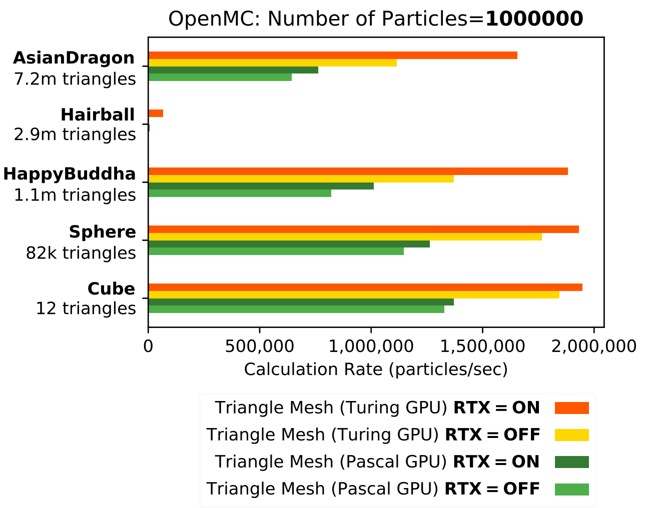
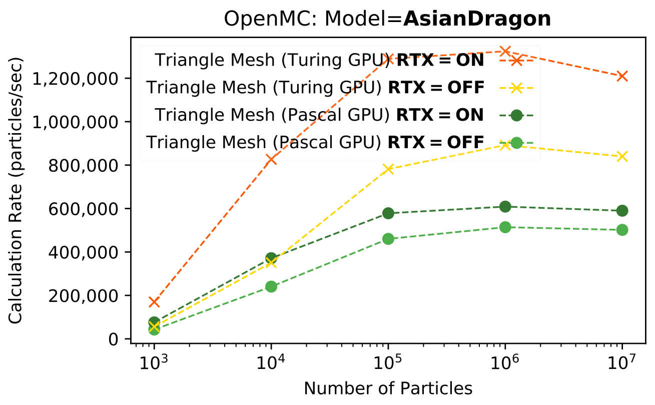


Fig. 8: Effects of RTX mode (which uses RT cores on the Turing GPU) on all models for a particle count of 10^6.

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.

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.



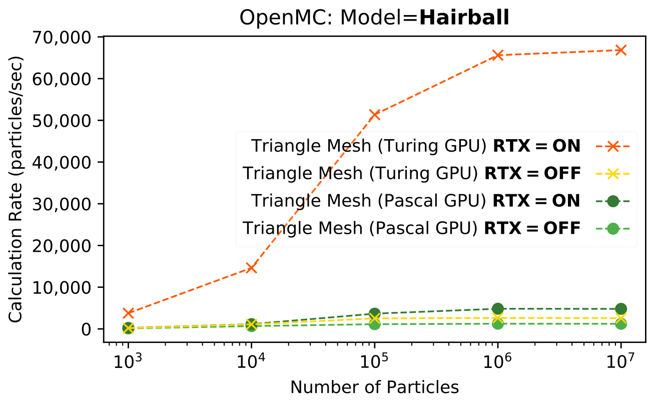


Fig. 9: Effects of RTX mode when the particle count is scaled up for the Asian Dragon and Hairball models.

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:

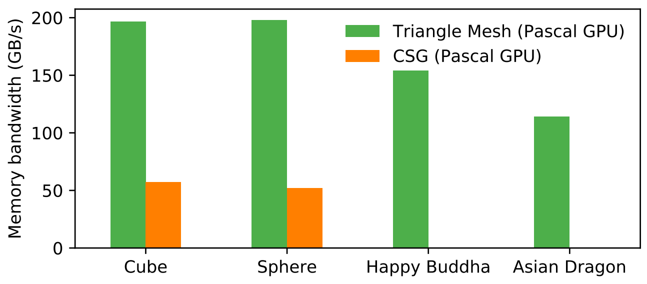


Fig. 10: Variation in memory bandwidth throughput for different models. Note that bandwidth for the Hairball model could not be obtained, as it caused the profiler to crash due to its complexity.

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 aim of this project 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

However, it will most likely be non-trivial for other codes to reap the same benefits, 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 libraries 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. However, it is 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.

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