Exploration of capabilities and applications of recent specialized hardware components in modern GPUs

Andrei-Viorel Grigorescu
Department of Computer Science
Babeș-Bolyai University
Cluj-Napoca, Romania

***Abstract*—Graphics processing units (GPUs) have been historically used for general purpose computing even when their purpose had been specifically for graphics rendering, obtaining popularity in various applications due to their high degree of parallelism and performance improvements associated with it. Ever since, GPU vendors have introduced novel hardware components to optimize other specific tasks, such as ray tracing and tensor operations. This paper focuses on the NVIDIA Turing architecture, namely on its tensor cores and real-time ray tracing (RT) cores. The intended purposes and functionalities of these components are explored, showing how a software developer can use them. The related works reviewed here present usage of RT cores in tasks more general than graphics through creative reframing of their problems. Starting from Bullet physics engine, the original implementation of a ray cast functionality using RT cores is written in Vulkan, with the intention to provide a vendor-agnostic solution. Implementation details, weaknesses and potential solutions are provided. Despite the overhead that can be improved upon, the short ray cast times encourage further work.**

Keywords—GPU programming, ray tracing, tensor cores, rigid body physics simulation

I. Introduction

Motivated by the impressive single instruction/multiple data (SIMD) parallelism capabilities of GPUs' streaming multiprocessors (SMs), interest and effort was directed into framing various problems in terms of steps in the graphics rasterization process so that this hardware capability may be used to speed up other tasks. With the advent of general purpose GPU programming APIs (GPGPU), previous graphics primitives were abstracted away and replaced with high performance computing concepts that a usual programmer may be more familiar with. This development led to numerous and successful applications that leverage the speed offered by streaming processors (SPs), ranging from video/audio processing and image manipulation to cryptography and discrete simulations.

With NVIDIA's release of a new GPU architecture [1] in 2018 and subsequent series of devices based on it, named Turing, two new specialized units have made their appearance on the streaming multiprocessors of the device: tensor cores and real-time ray tracing (RT) cores. This generation marks the first time that such components for solving highly specific problems were made commercially available for a broad segment of consumers, which draws attention to their functions and performance.

Tensor cores serve to accelerate tensor operations, tensors being an umbrella term for multi-dimensional numerical structures that contain vectors and matrices. By increasing the speed and frequency of tensor operations such as multiplication and addition, the performance of a broad range of applications is affected, because while the operation set is limited, they are very commonly found in practice so that it justifies this use.

However, in the case of RT cores, it is a subject of question on whether they possess the same degree of generality in applications they can serve, as opposed to previously mentioned SPs and tensor cores. After all, their marketed use is that of graphics rendering using the more recently popularized method of ray tracing, and using such cores makes little to no sense outside the context of a virtual world.

This paper intends to explore these recent specialized components in graphics coprocessors. As such, the background in Chapter 2 will cover the functions and capabilities of these components, define the problems that these components are often used in, as well as the options available to a programmer to work with them. Then, focusing on the RT cores which do not have as much coverage as the otherwise simple tensor cores, a review of the literature in Chapter 3 will show examples of their creative usage in problems other than those related to rendering graphics. Having identified reasonable critique in the literature review from the previous chapter, Chapter 4 will present the original approach of this work, as well as implementation details. Chapter 5 will then present the evaluation of the demonstrative program, including the benchmark performed, the results obtained, but also discussion around the qualitative aspects of the outcomes. Finally, conclusions and further work directions will be traced out in Chapter 6, such as observations from the design process, as well as points that can be improved and developed upon, both in terms of the presented program, but also on the architecture as well.

II. Background

A. Tensor cores

Tensor cores, originally appearing on Volta architecture, mainly target operations on up to two-dimensional matrices, which are otherwise a very common method of describing various concepts and models, from tridimensional affine transformations to control models for drivetrains. The operations available for tensor cores form an instruction set called MMA, short for matrix multiplication and addition.

These are used to implement operations of the form $D = A * B + C$, which might be familiar to GEMM formulation (general matrix multiplication). In [2] it is described how a naïve matrix multiplication algorithm with three nested iterative structures can be optimized into a form similar to those in BLAS (Basic Linear Algebra Subprograms), as in the depiction from Fig. 1. The inner loop that iterates over the individual values in each row and column can be extracted to be the outermost structure, preventing repeated memory accesses as the results are now accumulated into the result matrix. Then, in order to fit the working memory, the input

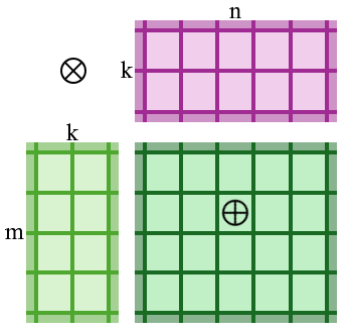

Fig. 1. An example of GEMM showing fragments from input matrices being multiplied then accumulated over output matrix in $D = A * B + C$ manner *(adapted from [2])*.

matrices are tiled to increments of same size. Finally, the article describes how these optimizations are implemented into a hierarchical block decomposition that matches GPU memory layout, with the smallest unit being computed not iteratively as previously described, but this time through tensor cores.

The MMA instruction set is similar to intra-warp communication primitives [3] including the necessity that all SPs call the instruction and with similar parameters, whatever conditional execution and branching may exist within the warp. Instructions in the set include loading a fragment from shared memory to registers, filling it with a constant value and storing it back in the memory. Once the fragments are loaded, the operation that requires the cooperation of the entire warp and performs the actual matrix multiplication and addition (MMA sync), may take place. This operation takes in the input fragments, which include information about their size, number type used and matrix layout. Finally, at a higher level such as in the kernel libraries provided by NVIDIA for GEMM, the fragments from the input matrices are multiplied and added in such a way, row by row and column by column, as to recreate the result matrix.

The matrix sizes supported are subjected to limitation to a few specific sizes, such as $16 \times 16 \times 16$ or $32 \times 8 \times 16$. Using MMA for matrices smaller than this would likely require padding matrices of these predetermined sizes with zeroes in order to be supported. In terms of the floating-point number standards that are supported, there are standard-precision floats, half-precision floats, other quantized formats such as INT8 or INT4 and even sub-byte operation support. The argument for using floating point numbers with less precision or more aggressive quantization is that the throughput of the tensor cores is increased in such scenarios, even by several times. Byte and sub-byte operation support might be surprising, but there is also a bitwise MMA operation available that allows the user to perform not multiplication and addition operations, but bit counting and logical operations such as XOR or AND.

Finally, another feature of the tensor cores that was later introduced in 2020 with the Ampere architecture is that of fine-grained structured sparsity. While less presented in benchmarks and literature, this advancement is presented in [4] to offer double the compute throughput by leveraging a 2:4 sparsity pattern for the purpose of compressing the matrix. Developing on this feature, a paper [5] also exists that presents how artificial neural networks can be pruned and retrained as to benefit from this sparsity, with comparable accuracy but better inference speed.

B. *Real-time ray tracing (RT) cores*

RT cores were introduced with Turing architecture with the goal of making ray tracing, a technique for rendering that produces photorealistic graphics but which comes with a cost in complexity, more tractable for use in real-time usage scenarios. This is made possible with contribution of AI models, the training and inference of which may be accelerated with tensor cores.

Ray tracing refers to emitting rays with a given origin position and direction into a virtual world, then observing what geometry it hits. Another ray can then be recursively traced from the hit position based on the properties of the object or on the data payload carried by the ray, as to create phenomena such as diffraction, reflection, and so on. The particular case in which there is no recursion is called ray casting. Ray tracing rendering method refers to determining the color of a pixel on the screen by emitting a ray from the camera position in a direction corresponding to that pixel, as determined by the screen space and camera model. Due to the capacity to simulate light transport phenomena through the methods previously presented, it is able to render photorealistic graphics.

So far, verifying for intersections between a ray and a triangle would imply checking against all the geometry in the scene. However, acceleration structures (AS) are spatial data structures that facilitate the process of finding intersections. An example of such structure is that of a bounding volume hierarchy (BVH). A bounding volume is usually the smallest volume cube that encompasses the entirety of an object, thus bounding it. For simplicity in operations, an additional constraint is placed that this box is also aligned to the axes of the world coordinate system used by the scene graph, known as axis-aligned bounding boxes (AABBs). What a BVH does then is to recursively divide the AABB of a 3D model into a tree structure of progressively smaller AABBs until the individual triangle can be determined directly, as seen in Fig. 2.

RT cores are thus capable of performing two important functions: that of autonomously traversing such structures and then performing ray-triangle intersections. With generous hardware specifications, having a great size, short latency and broad bandwidth to processor cores, these advances represent the core effort towards real-time ray tracing.

At a logical level, RT cores consider two types of AS. There is the bottom-level, which represents 3D models and their geometry in the manner described above.

Then there is the top-level AS, which represents the scene graph and all the objects in it. Individual objects are

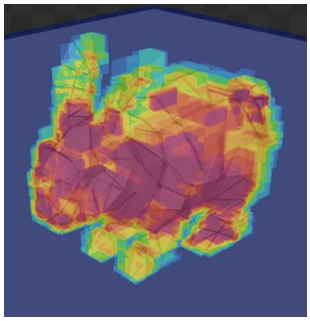

Fig. 2. Screen capture from NVIDIA Nsight Graphics during AABB Overlap Heatmap visualization, showing a bunny model with bounding boxes from the underlying BLAS representation and triangles of the tetrahedra.

represented by TLAS entries referencing any BLAS any number of times, but with each entry having its own affine transformation matrix or transform, which supports translation, rotation and scale.

In this way, ray tracing operations, which were originally expensive and done on the CPU and later in a GPGPU manner, can be offloaded to the RT cores so that the compute cores may be used for other tasks. Thus, client code can emit ray probes, wait for a result from the RT core or do other work in the meanwhile, then obtain data about the object hit, as well as the hit position or normal. These results may be useful to more than just rendering of photorealistic graphics, such as in collision detection, where it is useful to learn whether two bodies collide, what the collision points are and, possibly, how much do the bodies overlap.

C. APIs for interacting with tensor cores and RT cores

1) CUDA and OptiX

NVIDIA's API for GPGPU programming, CUDA, includes a new set of instructions for use with tensor cores, namely warp-matrix multiplication and addition (WMMA). The set contains "load_matrix_sync", "store_matrix_sync", "fill_fragment" and "mma_sync" (respectively "bmma_sync" for bitwise) instructions. It also offers the definition of a template-parametrized class named "fragment", which contains information about the size, floating-point number type and layout of the matrix in either row-major or column-major form.

As for the RT cores, these cannot be accessed from CUDA directly, requiring interoperability with OptiX, NVIDIA's ray tracing engine. One of the advantages of using NVIDIA's OptiX is that it allows the user optimized access to vendor-specific features. For example, more geometry primitives are supported for BLAS construction, such as linear swept spheres.

2) Vulkan

Vulkan offers the possibility of building pipelines such as graphics or compute pipelines, and even allows supplying of user code for the programmable shader stages in the form of pre-compiled shaders using SPIR-V. It also uses concepts such as extensions to enable further functionality starting from a core Vulkan version, which helps keep both a cross-platform standard but also allowing specific features where compatible. Another concept is that of layers, which wrap API calls for various purposes such as compatibility or validation layers, which help the developer identify misuse of the API and provide suggestions.

After introduction of RT cores in 2018, Khronos Group has released a specification for Vulkan Ray Tracing [6] in 2020. This standard, a result of the cooperation with the major GPU vendors, has led to development of a cross-platform option for ray tracing that is supported by NVIDIA, AMD and Intel, as well as Samsung, Qualcomm and other GPU vendors, including mobile devices.

The standard mentions two ways to use hardware-accelerated ray tracing: ray tracing pipelines and ray queries. The ray tracing pipeline is formed from 5 stages, each with programmable shaders, as seen in Fig. 3.

- The "ray generation" shader launches a grid (term related to 3D-like logical decomposition of streaming multiprocessors) of rays into a TLAS – the entry point for ray tracing into a scene.
- Each ray may then invoke the "intersection" shader, which describes how the geometry should be interpreted, or use the default ray-triangle intersection. This allows for use of different types of data without requiring tessellation, or the conversion of otherwise arbitrary data into geometry.
- Then, whenever a potential intersection happens, such as with an AABB, the "any hit" shader is invoked which describes whether any potential intersection is confirmed or not. There can be multiple potential intersections and there is no guarantee on the order.
- Finally, if ray-geometry intersection is confirmed, the "closest hit" shader is called.
- Otherwise, invoke the "miss" shader.

The closest hit shader then allows for tracing more rays in a recursive manner, supplying the current payload as input to the next one. In regard to recursion, there is a small upper limit to this recursion, such as 32, and the fewer rays are traced, the faster they can be resolved. Payload, while it can be customized, is also recommended to be small in order to improve memory usage and SM occupancy, otherwise it could impact the parallelism.

Rasterization in graphics pipelines can usually work strictly with the result of the previous step, so that vertex buffers are processed by vertex shaders, rasterized into fragments, then the fragments only processed by fragment shaders. Meanwhile, ray tracing casts rays that may bounce and hit any piece of geometry at any time and invoke different shaders or texture samplers - meaning that all the scene, textures and shaders must be loaded in memory at all times so that all this out-of-order execution can be supported. Shader binding tables (SBTs) are analogous to C++ virtual tables (vTables) that serve in polymorphism of object-oriented programming, but as opposed to those, SBTs must be built manually.

The other option, ray query, allows for emitting of only one ray per instruction. The advantage is that these ray queries may be performed from other pipelines such as graphics or compute pipelines. The TLAS is still required as an entry point, but otherwise the ray query may be defined, progressed, and checked for results, such as the hit object, triangle and barycentric coordinates for determining an object-space hit position. This style of using "single rays" can allow for development of more flexible or otherwise more simple algorithms than allowed by the ray tracing pipeline, such as wavefront propagation.

In Vulkan, tensor cores could be used through the cooperative matrix multiply operations in compute shaders. These operations, now present in GLSL extensions ("coopmatLoad", "coopmatStore", "coopmatMulAdd"), are similar to the WMMA instruction set offered by CUDA, with the concept of "coopmat" representing that of fragments.

3) OpenCL

Also proposed by the Khronos Group is OpenCL (compute language), an open standard specifically aimed for GPGPU programming, and which benefits from large popularity. Presently, it does not officially offer options for using either the tensor cores or the RT cores. However, it is

possible [1] to use in-line PTX (parallel thread execution) instructions, akin to an assembly language for the GPU, to call the MMA functions, though with a rather uncomfortable syntax.

Other APIs such as DirectX12 support ray tracing (DXR), though because this one in particular is more so commonly found in game graphics development, it is not treated in this work. Instead, other noteworthy mentions are the "position fetch" extension for Vulkan Ray Tracing and NVIDIA's proprietary ray tracing extension for Vulkan that served as a foundation for the standard and which includes some additional but vendor-specific features. In the case of the position fetch extension, it enables a BLAS construction flag that includes the vertex data in the BLAS, instead of having to transfer it separately, and this can represent an opportunity to cut down on unnecessary transfers.

III. Literature Review

The current work is very similar in explorative nature to that in [7]. It seeks to explore the capabilities of the new specialized components and to seek more general applications for the RT cores. However, the method presented here makes use of software advances that have appeared since the publication of this paper, namely in Vulkan's ray tracing pipeline, and the focus is on the potential for software written in this way to be used on different hardware platforms that have also implemented similar components..

While NVIDIA suggests in its whitepaper [1] surrounding the Turing GPU architecture that there are many non-rendering related operations that RT cores can assist with, such as in physics and collision detection, particle simulation, visibility queries and advanced audio simulation (related to the previous application), it is difficult to find applications that indeed leverage RT cores for tasks other than related to rendering even after several years since the introduction of these components to a large number of users.

In one such instance [8], DBSCAN, a data clustering algorithm, is implemented with RT cores by transforming the problem, with noticeable performance in larger datasets. Albeit the limitation to 2 or 3 dimensions, the authors describe that there are still numerous datasets such as geospatial ones that can make us of the method, as well as dimensionality reduction techniques. Finally, similar proposals for extension of the software functionalities available for the developer are made as in this presented work.

However, a handful of examples have indeed been found in the research domain of collision detection that have implemented hardware-accelerated ray tracing to speed up existing techniques. The techniques that RT cores could have been applied to are ones that use ray casting as a base element for answering questions related to the problem. But before approaching the papers that describe hardware-accelerated implementation of ray tracing, it may be of interest to present the implementation of past methods, described before the release of RT cores, that have inspired the newer adaptations.

One of the foundational methods for collision detection based on ray tracing primitives is given in [9]. This CPU implementation from 2008 introduces the concept of detecting collisions, contact points and adequate reactions between two objects, with focus on deformable bodies, by emitting rays from each vertex in the direction opposite to the normal associated with it. That is to say, the rays are sent within the object and, instead of checking for intersections across an entire 3D volume, it suffices that intersections with the inner faces of foreign bodies are found along these rays. The paper describes the method as pertaining to a narrow phase step in collision detection, relating to minute details about intersections between two potentially colliding bodies, as opposed to broad phase, which uses necessary but not sufficient checks for culling the list of potential intersections between those that may indeed collide, such as based on the intersection between their bounding boxes.

A similar method based on rays has been applied in 2015 using GPGPU [10], which offers robust collision detection for cloth simulations. This method extends the previous one by casting secondary rays outside the model for offering predictions into future collisions, a development previously published by the same author. Cloths are given a support surface based on a distance, such as half the cloth's thickness, converting its representation from a surface to a volume. It details how checks are separated based on the specifics of each involved object, such as having different implementations for checks between rigid bodies and cloths compared to those between strictly cloths. Despite the high vertex count, the OpenCL implementation has obtained real-time frame rates.

In 2023, one of the first hardware-accelerated ray tracing implementations for collision detection is presented in [11]. Among other contributions, the algorithm is similar to [9], while describing the same acceleration structure classification in TLAS and BLAS levels. The implementation offered combines CUDA for GPGPU programming and OptiX for ray tracing. Another interesting application is in [12], which proposes the use of hardware-accelerated ray tracing for the collision detection of a robot arm approximated by linear swept spheres. This marks an example where such collision detection has more immediate real-world application.

Another remark, and the one that serves as a motivation for the proposed approach, is that present hardware-accelerated ray tracing methods have received critique from literature [13] for being implemented in NVIDIA OptiX, which is regarded by the same literature as requiring specific hardware prerequisites in the form of RTX series GPUs with such RT cores. While true in case of the robot collision detection at [12] which uses linear swept spheres specific to the NVIDIA implementation, the other approach described in the particle simulation at [11] uses a subset of the features which should be available in the Vulkan Ray Tracing specification.

IV. Approach

A. Original approach

To address the limitation identified in literature, namely critique of existing collision detection methods based on hardware-accelerated ray tracing that use NVIDIA OptiX and are thus regarded as having a specific hardware platform

[1] According to a forum reply on NVIDIA Developer forum https://forums.developer.nvidia.com/t/opencl-can-call-tensor-core/246596/2

requirement, an alternative is provided in the form of an implementation using Vulkan Ray Tracing specification.

More exactly, the original contribution here is going to be a drop-in replacement for functions in a pre-existing physics engine. For this purpose, the Bullet physics engine [14] has been picked as the starting point for this demonstration. This choice is motivated by open-source access to the code, decomposition of the rigid body physics pipeline in broad phase and narrow phase checks familiar to those seen in the ray tracing pipeline specification, and the pre-existing support for OpenCL acceleration.

In order to simplify the code and resource management, the implementation proposed here also makes use of libraries provided by NVIDIA in its "nvpro" samples [15], namely "nvh" and "nvvk". These are used in their ray tracing tutorials to gradually introduce concepts and abstract the management of resources not directly related to ray tracing, while explaining what the used methods and classes do behind the scenes. It is important to note that the library is mainly used for educational purposes and as such simplifies many important processes, such as acceleration structure building and synchronizations, and that a more practical application would benefit from a more specialized and nuanced implementation.

The literature review shows several examples for implementing the narrow phase checks. Instead, one of the first efforts attempted here to apply hardware-accelerated ray tracing is to try and replace the AABBs provided by Bullet in the broad phase check with those specific to ray tracing acceleration structures. There is, however, a problem with this approach. Vulkan's ray tracing specification describes the acceleration structures, both top and bottom level, as "opaque". That is to say, the implementation of these acceleration structures is non-standard and specific to each vendor. This is problematic because despite having bounding boxes be created as part of the acceleration structures, they cannot be accessed nor compared outside the supported operations of BVH traversal and ray-triangle intersections, operations that do not aid in this use case.

The acceleration structures that Bullet offers in the form of AABBs, while seeming to be mostly redundant, cannot be replaced. Their software implementation allows for more questions to be answered in the interest of the client, such as intersection between two AABBs. Unlike the point-to-ray reformulation used in [7], here it is considered, though without experimentation, that expressing a bounding box through a series of rays is not worth the imposed overhead. As replacing these structures is not feasible, any implementation that desires to keep such functionalities must maintain two types of acceleration structures: the ones for hardware-accelerated ray tracing, and the software ones for other types of queries in the simulation.

However, more discussion can be held around this aspect. While current hardware cannot entirely replace the software acceleration structures, they nearly offer all the necessary functionality. Knowing that later extensions such as "position fetch", which adds vertex data to bottom-level acceleration structures, were released in 2023, discussion can be held with currently supported vendors in order to offer options for transparently reading and using the created bounding boxes, reinforcing ideas emitted in [8].

The next component to be investigated is that of user ray cast queries. The physics engine allows its user to perform ray casts and obtain as results the hit object, if any, then also the hit position, normal, covered distance and so on. These queries can be helpful for various applications, such as simulating visibility checks, measuring distances or even simulating LIDAR sensors, often used in robotics and autonomous driving. The functionality that will be tested in this situation is a common occurrence in the physics simulation demos and namely being able to drag and move objects in the scene. Once the user performs a click, the screen coordinates are converted through the camera model into a direction, and a ray is sent from the camera in that direction to identify which object the user has tried to click on. Upon confirming the hit, a pivot point is created at hit position and attached to the hit object, then the pivot is dragged around by following the user's cursor.

In the present, such functionality is implemented on the GPU using OpenCL kernels written for ray casting. To accelerate the process, a parallel linearized BVH is built on-demand using AABBs known from Bullet's internal data when casting rays is requested into the scene. Rays may be cast in batch, thus not requiring reconstruction of the PLBVH with each individual ray.

B. Implementation process

Implementing the proposed solution requires, besides the usual Vulkan setup involving various resources, the following steps:

1) Storing vertex and index arrays of object meshes registered in the scene

For convex bodies, Bullet offers the "b3ConvexUtil" class, which holds the vertices and faces of the body. Faces may be converted into indices by decomposing them using a triangle fan approach, where one point is fixed (the first) and the other two are iterated through the entries in the face's index array. This decomposition is not necessarily ideal, especially as the edge count of the face increases, and as it is elongated along one of the diagonals, but for simple quadrilaterals it should suffice. Normally, the "b3ConvexUtil" class is used only temporarily, until the vertex and index data are added to a global pool pertaining to Bullet's representation of the narrow phase data, but in this case, they are withheld from deletion until the world is removed so that they may supply their data for the next step.

2) Converting the geometry data into BLAS objects

This requires that the vertex and input data that are supplied to a BLAS to be available on memory buffers that are visible from the GPU. The BLAS objects are built in bulk, but adding more collision shapes does not require building the previously built structures. Given the task is related to physics and not rendering, the BLAS may be declared opaque to simplify ray tracing recursion.

3) Populating TLAS structure with instance entries

The TLAS is populated with entries containing the corresponding BLAS shape and the individual affine transformation. One of the differences between the transformations on Bullet and those in Vulkan is that Bullet transforms do not support scale. Then, that TLAS is built, in this case construction being carried after the original scene and the objects in it were successfully loaded.

4) Updating the TLAS

On demand, the TLAS is updated with the new transformations of individual objects. Given that physics integration takes place on the GPU, this requires that those transformations are transferred from device memory to host memory. This transfer may be costly if performed with each physics step, and performing the update of the TLAS by issuing the command from the GPU would be preferable, but because the two tasks are done on different compute APIs, this data is not actually accessible where it is needed.

5) Perform hardware-accelerated ray tracing

Once the TLAS is updated, a compute shader that operates the ray casting is launched and given the TLAS so that it has the required entry point. This kernel then, using the hardware-accelerated functions, creates a ray query for each input ray and iterates it until it is either a confirmed hit or a confirmed miss. The ray query, if it collided with an object, will be able to offer its identifier, the index of the triangle hit (which multiplied by 3 gives the first vertex involved), and the barycentric coordinates to the shader. With these and the "position fetch" extension, which allows the shader to access the vertex positions from the BLAS, an object-space ray hit position and normal can be computed according to Eq. 1, which are then returned to CPU.

$$hitPos_{obj} = (1 - u - v) * vertex_0 + / \\ u * vertex_1 + v * vertex_2 \quad (1)$$

6) Finalize results with world-space transforms

These results are partial because they still require an object-to-world-space transformation, which may be completed only with information available on the CPU (at least in this situation and not the one where transforms are available on the GPU). By applying the transformation matrix to the partial results like in Eq. 2, the world-space hit position and normal may be obtained and returned further for the client code to use.

$$hitPos_{world} = objPos + objRot * hitPos_{obj} \quad (2)$$

C. Testing

Implementation errors and misunderstanding of ray tracing specification led to several bugs in the application .The debugging of those was made simpler using tools such as NVIDIA Nsight suite, especially Nsight Graphics, which offers options for labelling memory objects, as well as inspecting these. Especially, it offers the developer the opportunity to visualize the acceleration structures, as well as insight into optimizing them. Initially rays had no hits because of improper handling of GPU buffers containing vertex and index data, leading to empty acceleration structures.

Another aid was visualizing the rays emitted with their source and hit position. This has helped with the detection of a bug in object-to-world-space transformation of hit points, the cause being narrowed down to improper operand order in non-commutative affine transformations, namely rotation.

Other disparities with the pre-existing method were found, such as treatment of objects that contain the source of the ray. To match the previous behavior, the "gl_RayFlagsCullBack FacingTrianglesEXT" flag was used to ignore inner faces, though this remains adjustable in any combination, such as for checking strictly the back faces in case of methods described in literature. This led to another inconsistency: it became apparent that the rays were, in some cases, ignoring the front face instead of the back face. This was thought to be because of a concept called winding, where the order of vertices in a triangle, clockwise or counterclockwise, can be used to define whether a face is oriented towards the front or the back. However, upon closer inspection of the ray query parameters used in the compute shader, it was found that "gl_RayFlags TerminateOnFirstHitEXT" does not terminate on closest hit, but on any hit, which as discussed in the background chapter, can happen with a bounding box as well.

V. Evaluation

A. Benchmark

There are two metrics that can be measured by the benchmark: the average *time* taken for ray casting, with and without acceleration structure building to distinguish between dynamic and static scenarios respectively; then, the average *error* of the resulting hit positions between the two implementations, as accuracy is also an important aspect when offering a new GPU-based implementation. For the latter, the two algorithms can be executed separately per each batch of rays cast, then the error and its maximum can be calculated.

As for the simulation used, there are at least two cases to consider: one is a baseline test with fewer objects, such as a $10 \times 10 \times 10$ grid of 1000 objects, so that a reference speed can be established, and a stress test featuring a larger grid of $45 \times 55 \times 45$, for a total of 111,375 objects. Another variable that may be considered is how the simulation scales with a varied number of rays. The stress test will place the ray cast bar higher (50 units as opposed to 15, considering cubes are 1-2 units long).

For the performance benchmarks, these are performed on an NVIDIA RTX 3070 Mobile (Laptop) GPU with background applications closed. The program is compiled in Release configuration, with compiler optimizations that favor speed ("/Ox" as it is referred to by Visual Studio 2022). Additionally, because of overhead imposed at beginning of the simulation, first 250 frames are skipped (5 data points) for the purpose of offering clearer results. Stress test cannot make use of the tetrahedron shape because convergence leads to bunching of objects in a small space, which in turn leads to many broad phase pairs being generated and overloading a technical limit in Bullet's current rigid body physics pipeline.

B. Results

Table 1 contains the average time required for strictly the ray tracing part per 50 frames, namely ray casting and acceleration structure building where appliable, as well as the average error. The error represents the maximum error

Table 1 Summary of performance and error results obtained during the benchmarks. Contains values aggregated over the duration of a testing episode from the complete data.

Object type	Object grid size	Rays cast/frame	Average time 50 frames (milliseconds)				Average result error
			with updates (dynamic)		without updates (static)		
			Bullet	Proposed method	Bullet	Proposed method	
Cubes	10x10x10	500	51.0857	**0.1142**	0.2857	**0**	0.000006
		2000	51.0285	**51**	**0.6571**	51	0.122343
	45x55x45	500	**94.1714**	475.2285	25.5142	**0**	1.33301
		2000	**101.9428**	506.9428	**45.1142**	51	1.454564
Tetrahedra	10x10x10	500	51	**0.08571**	0.4285	**0**	0.422094
		2000	**51**	**51**	**1.1428**	51	0.634864

between the results of a ray (using the old and the new method), collected every 50 frames. The number of frames over which the test is performed is limited to first 2000 frames (40 data points, with the reminder that the first 5 are skipped). This length suffices for the convergence of the smaller grid sizes, but not for the convergence of the larger grids.

C. Analysis and discussion

The major differences observed in results between the two methods (error) may be attributed to a delay in the update of the old method's AABBs used for ray casting. As Fig. 4 shows, the results offered by the proposed method may be preferable to those generated by the GPGPU method. Another possible explanation could be that the old method accounts for a collision margin in the simulation.

In terms of time, it appears at first glance that the proposed method obtains most performance in simulations with low object counts. To better explain the observed phenomenon, discussion branches based on the causes identified for the additional overhead present in the rest of the tests.

The major reason for non-competitive times are memory transfers of rigid bodies' transform data from the GPU to CPU and back. The reason for this is that the physics integration takes places on the OpenCL side of the GPGPU implementation, and these results are only communicated with the CPU upon interactions. Because the ray cast kernel is implemented in Vulkan, it does not have access to this data which is crucial in two steps: building and updating the TLAS, as well as in converting results from object-space to world-space. While these are done on CPU, they are recommended to be done on the GPU.

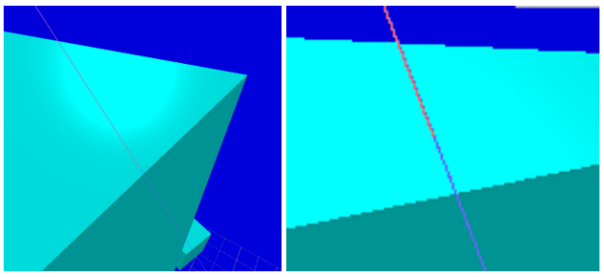

Fig. 4. A case of major difference between the results of the two methods: old method (red) and new method (blue). Right image is same instance but zoomed in. Lines are not drawn over the geometry, they can be occluded, so the continuity of the blue line suggests no intersection.

The reason for building the acceleration structures on the CPU come both from use of "nvpro" library which reduces implementation complexity instead of issuing the command for structure building from the GPU, and from the lack of transform data on Vulkan side. Meanwhile, the partial kernel results are only completed on CPU and not on GPU because of the same lack of transform data. Unless Vulkan can directly access OpenCL buffers, a complete solution would be represented by a complete Vulkan implementation of the physics engine, which is beyond the scope of this research.

There are various other suggestions and good practices that may be implemented in order to obtain a better performance either in general or for particular use cases. In the case that the scene does not update or if the geometry of moving objects is not of interest, acceleration structure updating may be entirely forgone. Alternatively, it could be updated on demand, rather than continuously, greatly improving performance on cases where only punctual checks are performed.

. Although the GPU driver may offer hints for moments where a complete rebuilding may be preferable to an update, the current implementation does not ever completely rebuild the TLAS. Rectifying this could offer considerable advantage when there are dramatic displacements in the objects of a scene, where outdated acceleration structures may yield sub-par performance. Another good practice is that of using partitioned TLAS (PTLAS), which divide the TLAS on a spatial basis, so that only the affected regions are indeed updated.

While not provided in the current solution, concave shapes could be treated uniformly with convex bodies. Bullet currently treats the two with different implementations and vertex/index buffers passing, which explains their omission in the current method. However, the implementation offered can serve as a reference for extending the support to concave bodies. Deformable bodies could also be supported in theory, but that would require update of BLAS instances and would prevent their reuse, as each instance may have its own deformation.

VI. Conclusions

This work has presented the novel specialized hardware components in recent GPU architectures, as well as their purpose, functions and options for programming them. The mainly discussed graphics API, Vulkan, was used to demonstrate the use of the hardware-accelerated ray tracing in an application unrelated to rendering, and namely in collision detection for physics simulation. The merit of this API is that its ray tracing specification represents a standard that was agreed upon by the major desktop GPU vendors (NVIDIA, AMD, Intel), as well as of other partners (Samsung, Qualcomm), offering the developers and the end users of their applications the freedom to choose any of the compatible hardware platforms as they see fit.

A. Further work

Regarding the demonstrative implementation, a Vulkan ray tracing kernel was successfully implemented on top of the Bullet physics engine, which offers a physics pipeline implemented in OpenCL. However, in terms of performance, because of this combination of computing APIs, the results are unsatisfactory and marked by several bottlenecks. However, this paper still serves to indicate the implementation steps and helpful utilities that could eventually shape a physics engine written entirely in Vulkan. For such an implementation, as it was shown in the literature review, there are numerous methods of collision detection based on hardware-accelerated ray tracing that could be applied. Such future works could help address the concerns in literature that these solutions are locked to hardware platforms offered by particular GPU vendors, and fuel more in-depth technical discussions about the performance obtained.

Discussion of results presents several directions for improving the performance. The main cause of overhead, memory transfers, can be removed by sharing data of the bodies' affine transformations between the two employed APIs, Vulkan and OpenCL. This would enable the results to be computed fully on the GPU, as well as permitting acceleration structure updates to be commanded from kernels. Other good practices are mentioned in said chapter as well.

Additionally, it has also been discussed that current optimizations used in hardware-accelerated ray tracing, such as BVH structures, are also present in physics simulations. However, these capabilities may not be shared because, as of yet, the software alternatives implemented in these simulations offer a broader control and instruction set over them. Meanwhile, the AABBs in the acceleration structures are opaque, and cannot aid the developer in accelerating checks such as the broad phase. However, it is found in this work that discussion about extending the transparency of these structures may represent the small step needed to enable such applications, and a discussion between GPU vendors, API standard writers and developers may be fruitful.

References

[1] NVIDIA, *NVIDIA Turing GPU Architecture [Whitepaper],* WP-09183-001_v01.

[2] A. Kerr, D. Merrill, J. Demouth and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C++," NVIDIA, 05 December 2017. [Online]. Available: https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/. [Accessed 12 June 2025].

[3] G. Barlas, "Multicore and GPU Programming: An Integrated Approach," Morgan Kaufmann, 2022, p. 493.

[4] NVIDIA, "NVIDIA Ampere GA102 GPU Architecture [Whitepaper]," V2.0, 2020.

[5] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu and P. Micikevicius, Accelerating Sparse Deep Neural Networks, arXiv preprint arXiv:2104.08378, 2021.

[6] D. Koch, "Vulkan Ray Tracing Final Specification Release," Khronos Group, 23 November 2020. [Online]. Available: https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release. [Accessed 10 June 2025].

[7] I. Wald, W. Usher, N. Morrical, L. Lediaev and V. Pascucci, " RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location," in *High-Performance Graphics 2019 - Short Papers*, Strasbourg, 2019.

[8] V. Nagarajan and M. Kulkarni, "Accelerating DBSCAN using Ray Tracing Hardware," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, St. Petersburg, Florida, 2023.

[9] E. Hermann, F. Faure and B. Raffin, "Ray-traced collision detection for deformable bodies," in *GRAPP 2008 - 3rd International Conference on Computer Graphics Theory and Applications*, Madeira, Portugal, Jan 2008.

[10] F. Lehericey, V. Gouranton and B. Arnaldi, "GPU ray-traced collision detection for cloth simulation," in *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology (VRST '15)*, pp. 47-50, Association for Computing Machinery, New York, NY, USA, 2015.

[11] S. Zhao and J. Zhao, "Revolutionizing granular matter simulations by high-performance ray tracing discrete element method for arbitrarily-shaped particles," *Computer Methods in Applied Mechanics and Engineering,* vol. 416, no. 116370, 2023.

[12] S. Sui, L. Sentis and A. Bylard, Hardware-Accelerated Ray Tracing for Discrete and Continuous Collision Detection on GPUs, arXiv preprint arXiv:2409.09918, 2024.

[13] Y. Lei, Q. Liu and H. Liu, "Developing a memory-efficient GPGPU-parallelized contact detection algorithm for 3D engineering-scale FDEM simulations," *Computers and Geotechnics,* vol. 179, no. 107031, 2025.

[14] E. Coumans, "GPU rigid body simulation using OpenCL," https://pybullet.org/wordpress/, 2013.

[15] M.-K. Lefrançois, P. Gautron, N. Bickford and D. Akeley, "NVIDIA Vulkan Ray Tracing Tutorial," [Online]. Available: https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/. [Accessed 13 June 2025].