Examination of the Nvidia RTX

V.V. Sanzharov[1], A.I. Gorbonosov[3], V.A. Frolov[2,3], A. G. Voloboy[2]
vsan@protonmail.com|alexey.gorbonosov@graphics.cs.msu.ru|vova@frolov.pp.ru|voloboy@gin.keldysh.ru
[1]Gubkin Russian State University of Oil and Gas, Moscow, Russia;
[2]Keldysh Institute of Applied Mathematics RAS, Moscow, Russia;
[3]Moscow State University, Moscow, Russia

Hardware acceleration of ray tracing is an active research field, but only with the release of Nvidia Turing architecture GPUs it became widely available. Nvidia RTX is a proprietary hardware ray tracing acceleration technology available in Vulkan and DirectX APIs as well as through Nvidia OptiX. Since the implementation details are unknown to the public, there are a lot of questions about what it actually does under the hood. To find answers to these questions, we implemented classic path tracing algorithm using RTX via both DirectX and Vulkan and conducted several experiments with it to investigate the inner workings of this technology. We tested actual hardware implementation of RTX technology on RTX2070 GPU and the software fallback in the driver on GTX1070 GPU. In this paper we present results of these experiments and speculate on the internal architecture of RTX.

Keywords: *photo-realistic rendering, ray tracing, hardware acceleration, GPU*

1. Introduction

Ray tracing is a cornerstone of photo-realistic image synthesis. Since first papers on ray tracing [19], [5], computer graphics researchers developed a plethora of different techniques to somehow accelerate the computations associated with ray tracing.

The hardware acceleration ray tracing had limited success out of research papers. Until the RTX technology by Nvidia was released in their Turing architecture GPUs. It was stated that Turing hardware contains special so-called «RT cores» which accelerate ray tracing. In the official Turing architecture whitepaper [22] it is stated that RT core contains two units which perform bounding box and ray-triangle intersection tests. But since RTX is closed source, we don't know for sure how exactly it is implemented and if this is all that is to ray tracing acceleration in Turing GPUs. In this paper, we present information on several experiments we did with an RTX GPU. We analyze the experiments' results and speculate on possible techniques used in RTX hardware to accelerate ray tracing. But first of all, let's review the research in ray tracing acceleration hardware to understand what techniques were already tried out in hardware implementations and how well did they perform.

1.1 Related work in ray tracing acceleration hardware

First dedicated hardware solutions closely related to ray tracing were PCI cards for volume data visu-alization which implemented ray casting and Phong shading (such as [9, 12]). Even though these hard-ware traced only primary rays, it already implemented techniques to increase the efficiency of parallel tracing such as grouping rays to make use of memory access coherence [9]. Another notable product was SaarCOR architecture [13] and its updated version in an FPGA chip [14]. The SaarCOR chip implemented the whole ray tracing algorithm - scene and camera data were uploaded from the host and the chip produced the rendered image. Like the ray casting solutions, SaarCOR used packet tracing (in groups of 64 rays). The architecture was fully pipelined to further mitigate memory access latency - simultaneously traversing one group of rays, loading data for the next group and intersec-tion operation performed on another group of rays. An example of ray tracing hardware which was commercially available is ART AR250/350 rendering processor with a custom RISC processor core [4]. The solution was used to accelerate offline rendering and was packaged as x86 PC with 16, 36 or 48 rendering processors as PCI-X cards and gigabit networking system. Software side included RenderMan compliant renderer and network communication interfaces and plugins for 3D applications (CATIA, 3ds Max, Maya). Details about the custom rendering processor to our knowledge were never published.

All works mentioned to this point concern fixed function hardware. One of the first solutions with programmable stages is RPU (ray processing unit) [20]. The traversal and primitive intersection tasks are implemented in fixed function units. RPU supported custom shaders with features such as recursive function calls, trace instruction to initiate tracing of an arbitrary ray, asynchronous load instruction to hide memory latency. RPU also featured geometry shaders, instancing support and shader tables to look up specific shader to execute for a particular geome-try object. As SaarCOR and ray casting solutions, RPU also uses packet ray-tracing which can result in performance drops in the case of incoherent rays. The TRaX architecture [16] implements a different solution - many identical cores consisting of simple thread processors. It can be viewed as general pur-pose architecture and is used in other papers to simu-late their hardware [7]. In the ray-tracing application TRaX accelerates single ray performance and features MIMD execution model as opposed to groups of 4 or

more rays and SIMD model in previously mentioned architectures. The authors in [10] aimed to address problems with incoherent rays by using N-wide SIMD processing architecture with filtering of rays to find coherent groups. The filtering is applied at traver-sal, intersection and shading stages of the ray tracing algorithm.

In [1] authors simulate architecture close to that of Nvidia Fermi GPU. One of the key aspects of it (related to ray tracing) is work compaction. When a warp (group of 32 threads) has more than a half of rays terminated, it terminates and the non-terminated rays are copied to the next warp. This mechanism allows to mitigate the effect of incoherent rays and preserve the parallelism. Another suggestion in this work is related to stack memory layout for threads. Also [1] implements the idea of partitioning BVH into treelets (which approximately matches cache sizes) and group-ing rays according to treelets they intersect. Another architecture - STRaTA [7] is built on top of the TRaX [16] and implements modified treelet technique of [1] and streaming approach to processing rays associated with each treelet. STRaTA adds special small buffers to memory hierarchy to store rays.

In [15] authors focus on improvements related to memory access, in particular, completely avoiding random memory access during ray traversal. Their approach is based around presenting data needed for ray tracing in two streams - stream of geometry data split in segments and stream of rays collected as a queue per geometry segment they intersect. This al-lows for fetching geometry and rays from main mem-ory into caches before they are needed for traversal.

Work [6] in addition to MIMD execution model and treelets proposes using reduced precision BVH traversal which also allows for chip area and power savings. Another specific point of [6] is that authors propose small solution which can be integrated into existing GPU architecture. There are also works focused on developing mobile ray tracing hardware (such as [8, 11]). These solutions usually have such common properties as MIMD execution model, hardware traversal and intersection units. Raycore [11] has distinctive properties that separate it from other architectures - it's fully fixed function Whitted-style ray tracing [19], it uses kD-tree as acceleration structure and includes hardware unit for kD-tree construction.

Summary. Overall, quite a few different architectures and hardware acceleration techniques for ray tracing were proposed over the years. Detailed review and comparison can be found in [2]. Some of the mentioned architectures had been implemented in FPGAs. Production level hardware applications besides Nvidia RTX are represented by [4] and mobile GPUs by Imagination technologies [21]. However, both of those have no published details, [4] is discontinued and [21] is not yet available. Therefore, RTX is the first hardware ray tracing acceleration technology to reach wide public. But since the implementation details are closed (like [4, 21]), it is unclear how exactly does it work and what acceleration techniques it uses. In this paper, we aim to understand the principles be-hind ray tracing acceleration in Nvidia RTX hardware by measuring the performance in several scenarios us-ing Vulkan and DirectX12 API.

2. Experimental analysis of Nvidia RTX

First let's briefly review available information about inner workings of RTX. Access to RTX ray tracing functionality is available through Vulkan API, Microsoft DirectX 12 (DXR) and Nvidia OptiX API libraries [23]. We used both Vulkan and DirectX 12 for our experiments.

2.1 Known details

In summary, for both graphics APIs the corresponding extensions add functionality to create ray tracing pipeline with the corresponding new shader types, commands and objects for acceleration structures, and tools to associate shader groups with acceleration structures (i.e. shader binding table).

Acceleration structure is represented as two-level tree. Bottom level acceleration structure (BLAS) objects contain actual vertices and top level acceleration structure (TLAS) contains BLAS object instances i.e. transformation matrices. The building process is done on the GPU, acceleration structure is some form of BVH [17].

Ray tracing pipeline has five shader types - ray generation, miss, closest hit, any hit and intersection. Shader programs of first three types are mandatory and the last two are optional. All stages of ray tracing algorithm are programmable. There is built-in ray-triangle intersection shader which is used by default. Official whitepaper [22] states that RT core has ray-triangle intersection unit inside. In [18] authors show 2-3.5 times improvement in performance of their algorithm of point location in tetrahedral meshes when us-ing built-in triangle intersection unit on Turing hard-ware while Volta hardware (which has no RTX cores, so software fallback is used for RTX functionality) shows performance loss in the same scenario.

2.2 Experiments

To understand how RTX works under the hood we conducted several experiments. As a base for our investigations we implemented a basic path tracing algorithm [5] and compare it to Open Source implementation of path tracing in Hydra Renderer [24].

Implementation of a minimal path tracer using RTX in Vulkan or DirectX 12 would require developer to:

1. build acceleration structures using ray tracing extension API;

2. create ray tracing pipeline containing at least ray generation, closest hit and miss shader programs;
3. create shader table to bind shader programs to acceleration structures;
4. create and execute command buffers on created pipeline.

There are several design options even in the minimal implementation using RTX which can potentially affect performance. For example, the shading and lighting code can be executed in a ray generation shader, in a single (closest) hit shader or in several hit shaders. We tested two different implementations according to best practices of RTX for Vulkan and DX12:

1. impl_1 (Vulkan): ray generation shader creating ray(s) for each pixel in a cycle until the specified tracing depth is reached;
2. impl_2 (DirectX): ray generation shader spawning primary ray and closest hit shader taking care of generating rays until specified depth is reached. To measure performance in all our experiments we used Nvidia Nsight Graphics software and 2 GPUs—GTX1070 and RTX2070. It is known that while RTX2070 has hardware acceleration for ray tracing, GTX1070 has software implementation of RTX. Using this setup we captured frames from our path tracing application and logged time spent by vkCmdTraceRaysNV (Vulkan) or DispatchRays (DirectX 12) function and «BVH4TraversalInstKernel» kernel in Hy-dra Renderer. In our first set of experiments we ran implemented path tracer on three scenes (Sponza, CrySponza, Hairballs) with different tracing depth. From measured time we calculated frames per second and approximate amount of rays traced per second as:

$$rays = width * height * spp * fps \qquad (1)$$

width, *height* – rendering resolution, *spp* – samples per pixel, *fps* – frames per second.

scene	primary	secondary	tertiary
Sponza, impl_1	807	437	806
Sponza, impl_2	928	777	694
Sponza, Hydra_SW	480	122	130
Crysponza, impl_1	806	419	388
Crysponza, impl_2	754	635	216
Crysponza, Hydra_SW	276	92	80
Hairballs, impl_1	275	223	256
Hairballs, impl_2	567	155	141
Hairballs, Hydra_SW	61	50	56

Table 1. Million rays traced per second (Mrays/s), 1 sample per pixel, 1024 x 1024 resolution, RTX2070

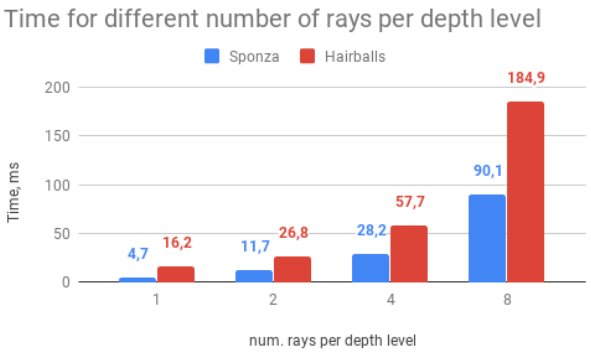

Fig. 1. Time spent by ray tracing "draw call" per frame (1 sample per pixel, 1024 x 1024 resolution) depending on rays traced per depth level. Depth = 3

Next, we modified impl_2 with tracing several rays at each depth level essentially transforming it into an implementation of branched (recursive) path tracing. As can be seen in fig. 1, the time increases consis-tently with the number of rays, even slower in some cases. For example, with 4 rays per depth level the total number of rays is 7 times higher than for 1 ray per depth level (21 against 3). And the performance drop is 6 times for Sponza and 3.6 for Hairballs.

3. Results and discussion

Conclusion #1: *Nvidia RTX is primarily aimed at accelerating random access to memory during ray tracing. More specifically, traversing BVH tree with a sets of random rays.* This conclusion stems from (fig 2, right), where we can see that hardware implemen-tation on the small scene (Sponza) wins only 2 times (477 vs 1140) with «coherent» and «sorted» sets of primary rays. But breaks away 4-5 times for the same Sponza and incoherent rays (122 vs 561). Moreover, large scene (Hair Balls) shows same 4-5 times for both primary (58 vs 283) and secondary (50 vs 210) rays. The fact that acceleration is preserved on the scene where the bottleneck is the memory confirms our conclusion.

Conclusion #2: *Nvidia RTX implements some ray-grouping/ray-sorting.* It's done probably in combination with GPU work creation (see conclusion #4). This assumption is confirmed by the fact that on simple scenes (like Sponza) hardware implementa-tion doesn't have significant performance drop when we move from primary to secondary rays (table 1, fig1). At the same time software implementation sees its performance degrade much faster. However, on the scene where ray grouping could not help (Hair balls), both hardware and software implementation don't have significant performance difference between primary and secondary rays.

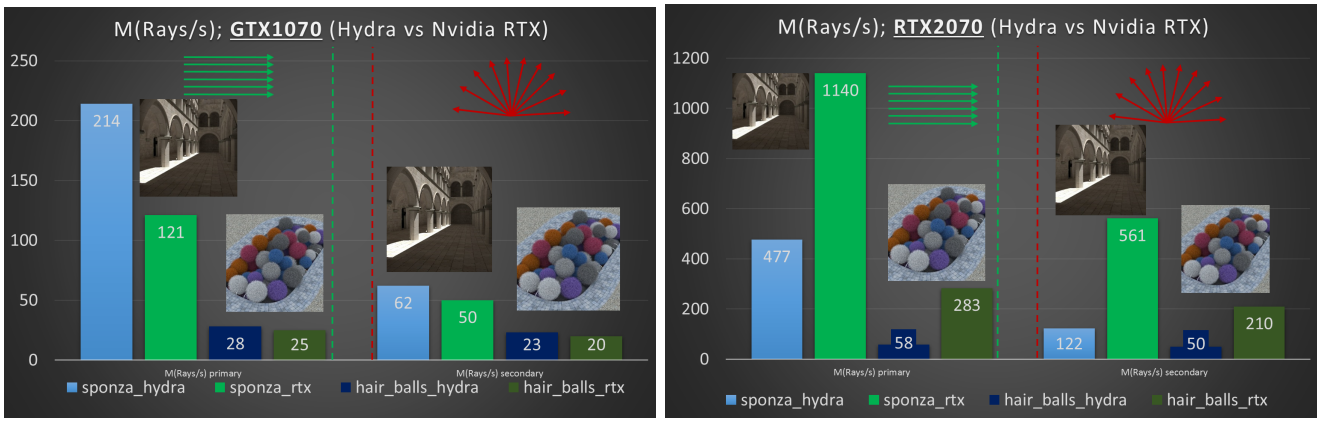

Fig. 2. Comparison on GTX1070 (left) and RTX2070 (right) (Open Source implementation vs Nvidia RTX). The left part of each image (green) shows performance for primary (coherent) rays, and the right part (red) for secondary (random) rays.

Sponza (66K tris) Cry Sponza (262K tris) Hair Balls (224M tris)

Fig. 3. Test Scenes

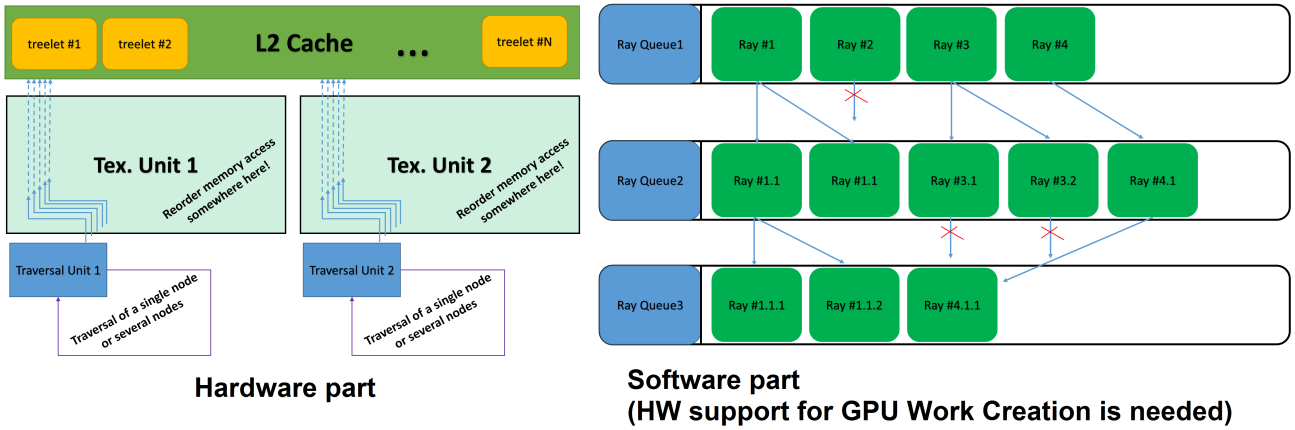

Fig. 4. Supposed internal architecture of Nvidia RTX. According to the results of our experiments, the hardware implementation should be closely connected with the texture units, or it is part of texture unit. We believe the most interesting part is related to reordering of memory access and thus it should work in analogue to well known memory access reordering inside texture units. In this way, **traversal unit itself could be small enough** and probably implements reduced precision BVH traversal [6] (or some analogue) for better cache efficiency and reducing HW cost.

Conclusion #3: *Despite the Nvidia attempt, placing the whole code in a single kernel («CPU style» or «uber kernel») is still inefficient for GPUs.* We make such conclusion because of 2 main reasons. First, open source implementation with separate ker-nel in Hydra Renderer benefits almost 2 times over Nvidia RTX for pure software case (fig. 2, left). Second, when comparing 2 slightly different implementations of RTX in Vulkan and DX12 we have found dramatic changes in performance depending on a slight change in the complexity of shaders in «impl_1» (more complex) vs «impl_2» (simpler), table 1. This can be explained by occupancy drop depending on code complexity and register pressure.

Conclusion #4: *Nvidia RTX uses GPU work cre-ation for rays.* This conclusion is confirmed by simple observation. When we generated random amount of rays (10 to 40), we got 2 times slower in comparison with 10 rays. In contrast to ray tracing, when we calculated Perlin Noise with random noise function calls (10 to 40), we got exactly 4 times of what we should have without GPU work creation. Our experiment with recursive ray tracing (fig.1) also confirms GPU work creation presence since the time is proportional to the number of rays.

4. Final conclusion

Our main conclusion is that Nvidia RTX is some sort of «general» technology, oriented to speeding up random memory access and irregular work distribution on GPUs. In this way we can expect in near future different sets of algorithms (at least some spatial search algorithms) to be hardware accelerated.

We believe Nvidia puts a lot of efforts in their compiler and software support of GPU work creation. On the example of this technology we can see, that «the golden age of software» has ended and the «the golden age of compilers and HW/SW projects» has started.

Despite the overall complexity of Vulkan and DX12, such improvements make GPU implementation of complex rendering engine much simpler for devel-oper. On the other hand, this simplicity is achieved at the cost of tying the project to a fairly heavy tech-nology. We believe that efficient software implemen-tation of RTX will be complex and expensive due to GPU work creation and specific compiler that Nvidia puts inside RTX — even Nvidia's software implemen-tation on GTX1070 essentially loses to simple and straightforward open source ray tracing implementa-tion in Hydra Renderer.

5. Acknowledgments

This work was sponsored by RFBR 18-31-20032 grant.

6. References

[1] Aila T., Karras T. Architecture considerations for tracing incoherent rays //High-performance Graphics. – Eurographics Association, 2010. – p. 113-122.

[2] Deng Y. et al. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture tech-niques // ACM Computing Surveys (CSUR). – 2017. – . 50. – №. 4. – p. 58.

[3] Gribble C. P., Ramani K. Coherent ray tracing via stream filtering //2008 IEEE Symposium on Interac-tive Ray Tracing. – IEEE, 2008. – p. 59-66.

[4] Hall. D. The AR350: Today's ray trace rendering processor. //Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1, 2001

[5] Kajiya J. T. The rendering equation //ACM SIG-GRAPH computer graphics. – ACM, 1986. – . 20. –№. 4. – p. 143-150.

[6] Keely S. Reduced precision hardware for ray tracing.//Proc. HPG. – 2014. – p. 29-40.

[7] Kopta D. et al. An energy and bandwidth efficient ray tracing architecture //High-performance Graphics. –ACM, 2013. – p. 121-128.

[8] Lee W. J. et al. SGRT: A mobile GPU architecture for real-time ray tracing //High-performance graphics conference. – ACM, 2013. – p. 109-119.

[9] Meißner M. et al. VIZARD II: a reconfigurable interactive volume rendering system //ACM Eurographics conf. on Graphics hardware. – Eurographics Associa-tion, 2002. – p. 137-146.

[10] Nah J. H. et al. T&I engine: traversal and intersection engine for hardware accelerated ray tracing //ACM Transactions on Graphics (TOG). – ACM, 2011. – . 30. – №. 6. – p. 160.

[11] Nah J. H. et al. RayCore: A ray-tracing hardware architecture for mobile devices //ACM Transactions on Graphics (TOG). – ACM, 2014. – . 33. – №. 5. – p. 162.

[12] Pfister H. et al. The VolumePro real-time ray-casting system. //Computer graphics and interactive techniques. – N.Y.: Association for Computing Machinery. – 1999. – p. 251-260.

[13] Schmittler J., Wald I., Slusallek P. SaarCOR: a hardware architecture for ray tracing //ACM SIGGRAPH conf. on Graphics hardware. – Eurographics Associa-tion, 2002. – p. 27-36.

[14] Schmittler J. et al. Realtime ray tracing of dynamic scenes on an FPGA chip //ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hard-ware. – ACM, 2004. – p. 95-106.

[15] Shkurko K. et al. Dual streaming for hardware-accelerated ray tracing //High Performance Graphics. – ACM, 2017. – p. 12.

[16] Spjut J. et al. TRaX: A multi-threaded architecture for real-time ray tracing //Symposium on Application Specific Processors. – IEEE, 2008. – p. 108-114.

[17] Stich M. Real-time raytracing with Nvidia RTX, GTC EU 2018

[18] Wald I. et al. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. Authors' Preprint — to be presented at High-Performance Graphics 2019

[19] Whitted T. An improved illumination model for shaded display //ACM SIGGRAPH – ACM, 1979. – . 13. – №. 2. – . 14.

[20] Woop S., Schmittler J., Slusallek P. RPU: a programmable ray processing unit for realtime ray trac-

ing //ACM Transactions on Graphics (TOG). – ACM, 2005. – . 24. – №. 3. – p. 434-444.

[21] Imagination technologies. PowerVR Ray Tracing. 2019. URL = https://www.imgtec.com/graphics-processors/architecture/powervr-ray-tracing/

[22] Nvidia Turing architecture whitepaper. 2019 URL = https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[23] Nvidia RTX Ray tracing developer resources. 2019 URL = https://developer.nvidia.com/rtx/raytracing Ray

[24] Tracing Systems, Keldysh Institute of Applyed Mathematics, Moscow State Uiversity. Hydra Renderer. Open source rendering system. 2019 URL = https://github.com/Ray-Tracing-Systems/HydraAPI Vulkan

[25] specification. 2019 URL = https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html