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.

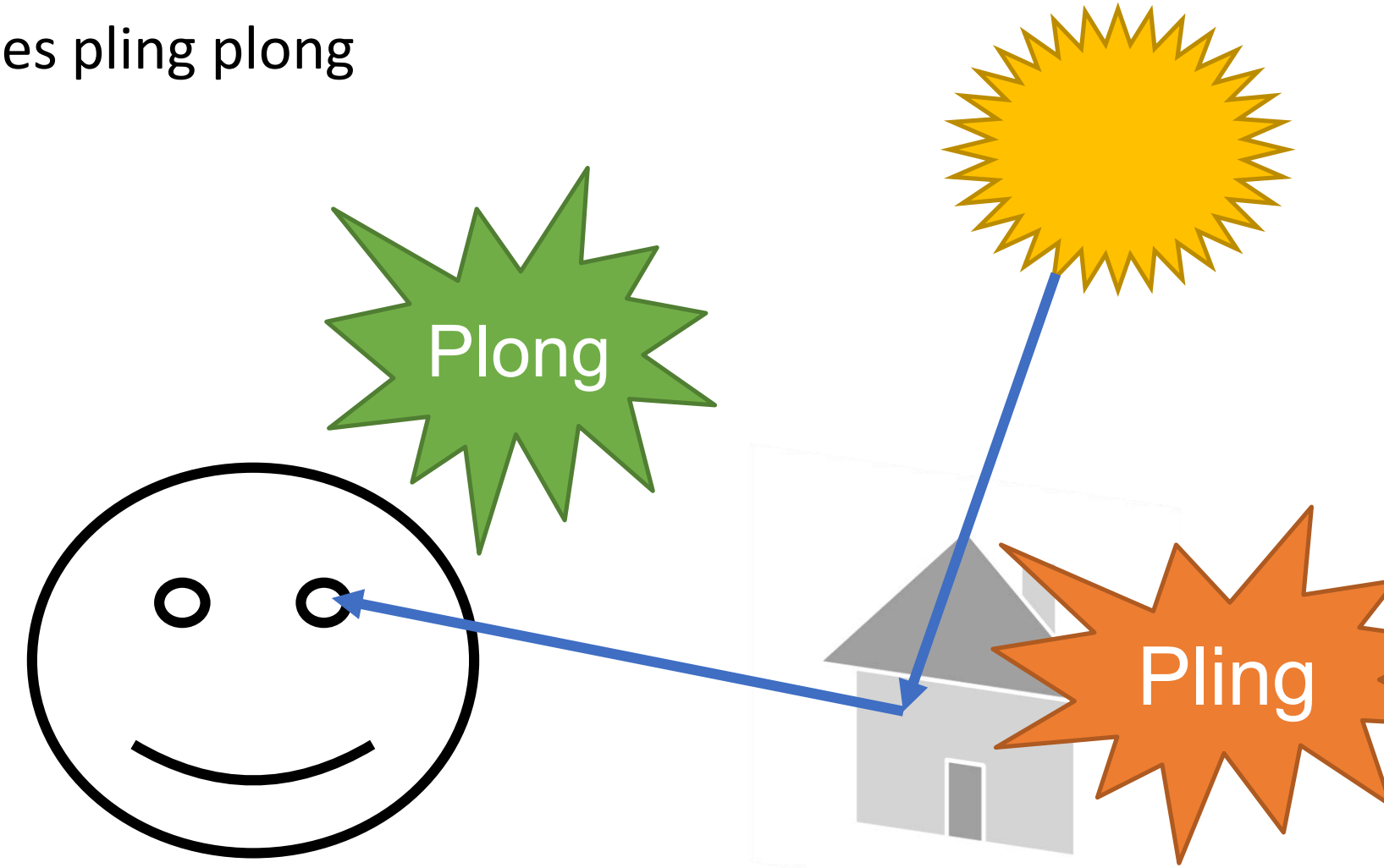
The hardware acceleration ray tracing had limited success out of research papers. Until the RTX technology by Nvidia was released in their Turing architecture GPUs. It was stated that Turing hardware ray tracing algorithm - scene and camera data were uploaded from the host and the chip produced the rendered 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

Abstract

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.

Raytracing

- Simulated light-ray goes pling plong
- Sloooow
- Looks good



History

"Those who cannot remember the past are condemned to **repeat** it."

-George Santyana

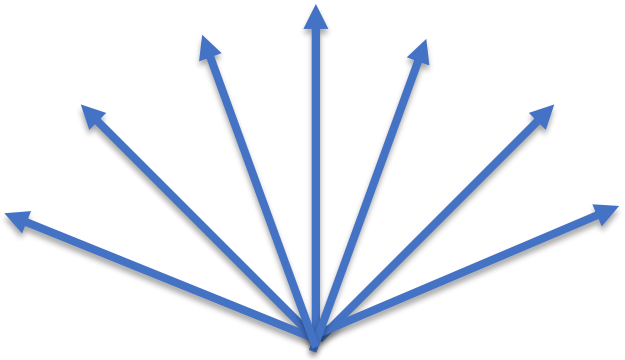
Early days (around 2002) (SaarCOR)

- FPGA
- Packet tracing of groups of 64
- Pipelined to reduce memory
  - Traversing, loading and intersection of groups
- Fixed function hardware.
  - Can’t change the algorithm nor data structure
- SIMD

Ray Processing Unit (RPU) (2005) (Based on SaarCOR)

- Accelerated ray traversal
- Traversal and intersection in fixed hardware
- Allowed custom shaders
- Also packet, as it is based on SaarCOR
- Performance drops on incoherent rays

Coherent



Incoherent

TRaX

- Just add more threads lol
- Accelerated single ray performance
- MIMD

More improvements

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

- Then they are like: What if we partition the BVH tree?!?

- BVH: Bounding volume hierarchy

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 is always an issue

- What if we improve the memory access?
- Removing random memory access during ray traversal:
  - Presenting data needed in streams
    - One for geometry
    - One for rays which is collected as a queue per geometry they intersect
  - This fetches the memory into caches before the traversal
- But it is still hard for incoherent rays, which needs filtering to find coherent groups

TL;DR

- They all flopped consumerwise
- ”RTX is the first hardware ray tracing acceleration technology to reach wide public” - Paper

- But again RTX is closed-source proprietary tech
  - So that’s what this paper is about

What we know

- From the documentation, we know a few things:

- Data structure is represented as a two-level tree
  - One for object vertices
    (Bottom level accelerated structure)
  - One for object instances
    (Top level accelerated structure)

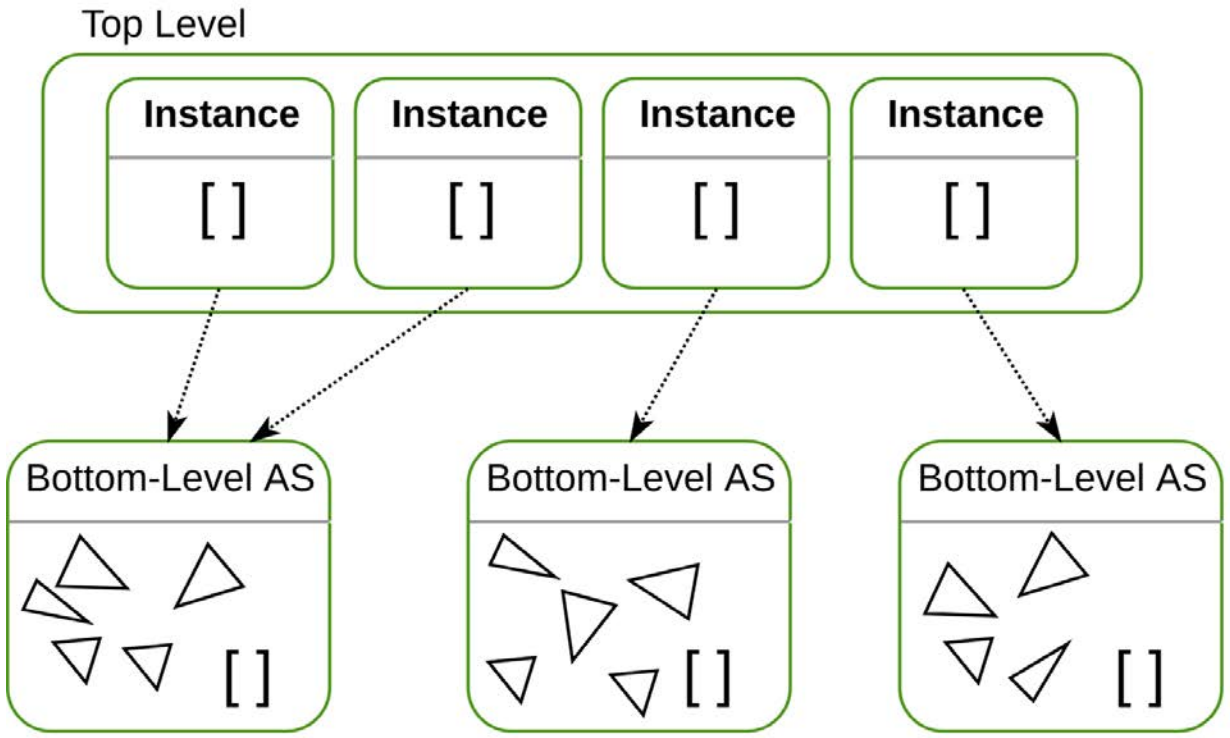


Image: https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1 figure 4

What we know

- That the acceleration structure is a BVH (bounding volume hierarchy)
- Pipeline have 5 shader types, where all are programmable
  - Generation
  - Miss
  - Closest hit
  - Any hit (Optional)
  - Intersection (Optional)
- An RT-core has ray-triangle intersection unit inside

Experiment

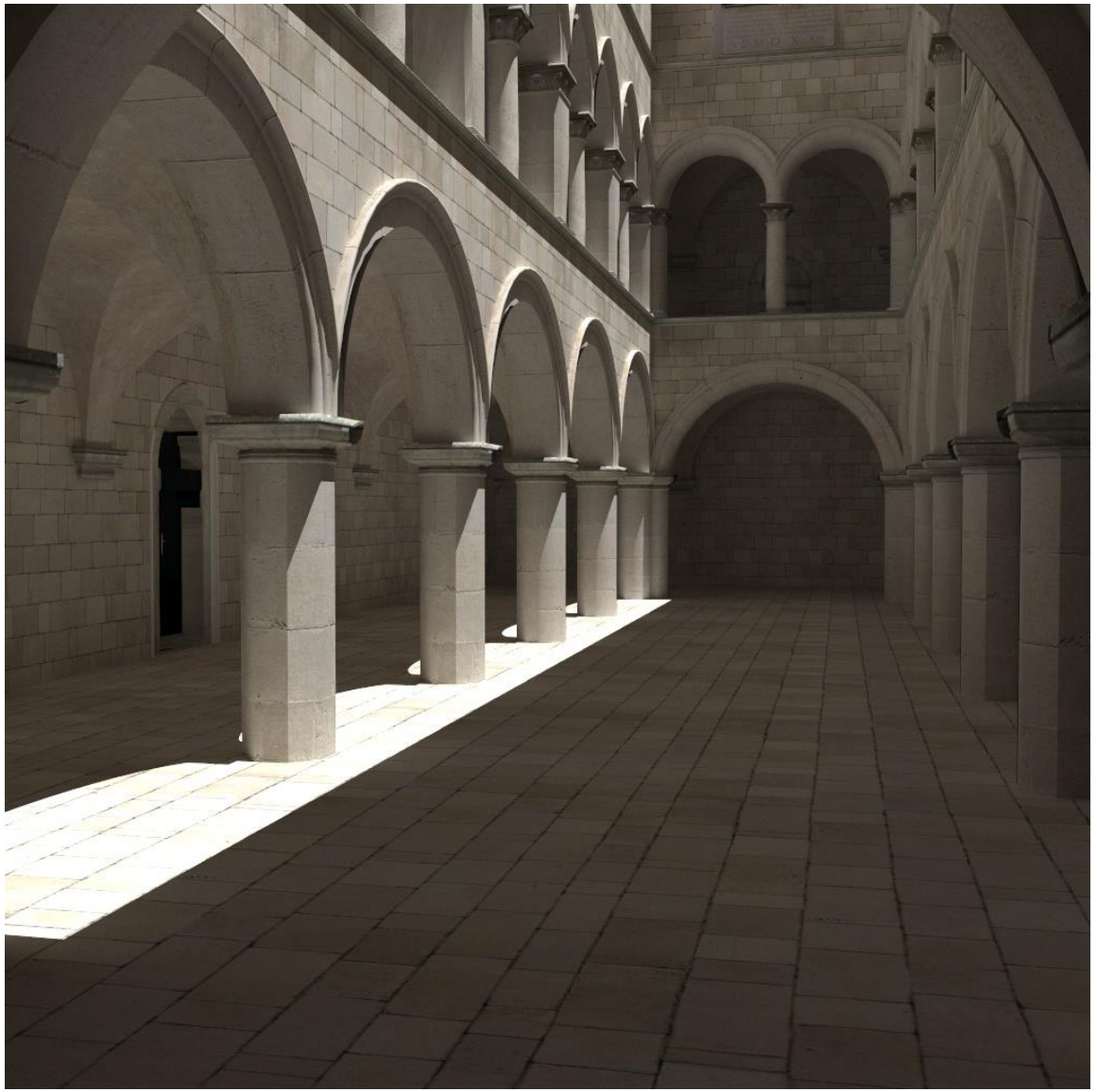
- RTX ray tracing can be used via Vulkan API, Microsoft DirectX12 or Nvidia Optix Api.
- Paper used both Vulkan and DirectX 12 for testing

Two types of minimal RTX implementation
(best practice)

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.

Scene 1 - Sponza

- Simple geometry
- 66K triangles

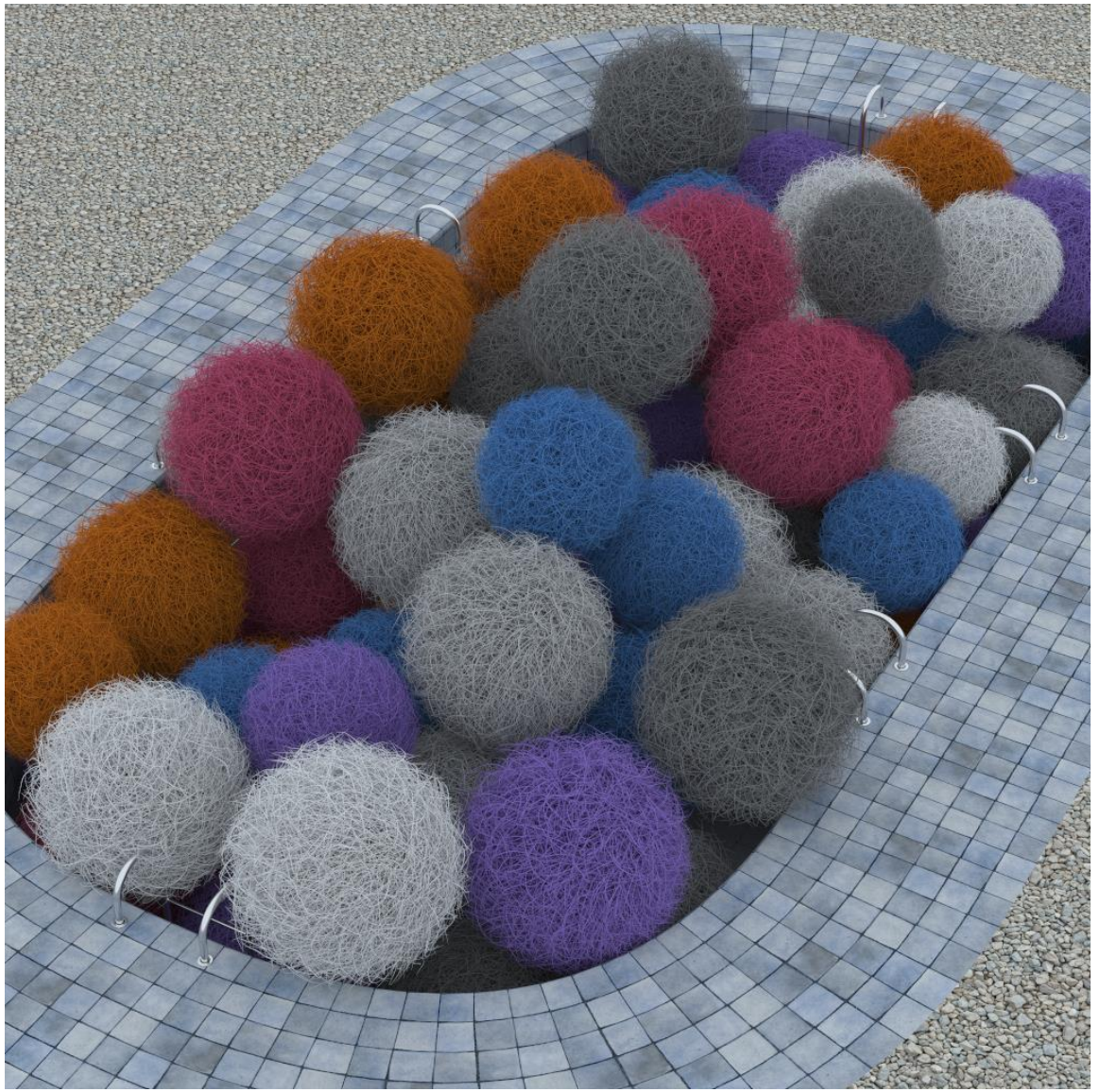


Scene 2 - Cry Sponza

- Sponza with extras
- 252K triangles

Scene 3 - Hair Balls

- Hard
- 224 Mill triangles



Experiment

- They compared:
  - RTX impl_1
  - RTX impl_2
  - GTX1070, with its software implementation of RTX
  - Open source implementation: Hydra Renderer

- Looked rays traced per second
  - Rays = width * height *samples per pixel * fps

RESULT 1

scene	primary
Sponza, impl_1	807
Sponza, impl_2	928
Sponza, Hydra_SW	480
Crysponza, impl_1	806
Crysponza, impl_2	754
Crysponza, Hydra_SW	276
Hairballs, impl_1	275
Hairballs, impl_2	567
Hairballs, Hydra_SW	61

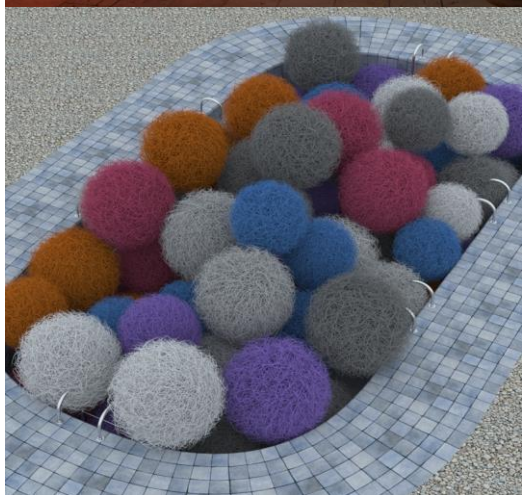
Table 1. Million rays traced per second (Mrays/s), 1 sample per pixel, 1024 x 1024 resolution, RTX2070

hidden

Sponza

Crysponza

Hairballs



RESULT 1

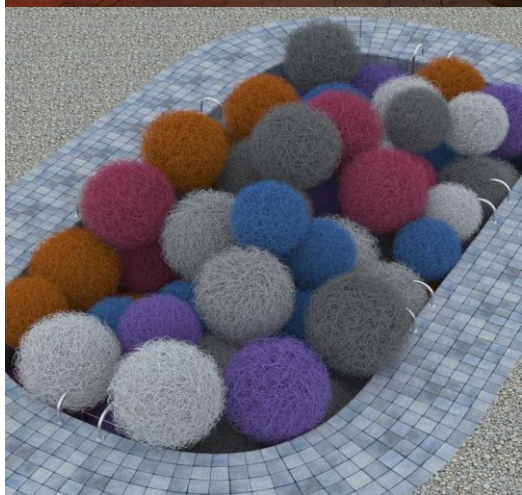
scene	primary	secondary	tertiary
Sponza, impl_1	807	437	806
Sponza, impl_2	928	777	694
Sponza, Hydra_SW	480	122	130
Crysponza, impl_1	806	419	388
Crysponza, impl_2	754	635	216
Crysponza, Hydra_SW	276	92	80
Hairballs, impl_1	275	223	256
Hairballs, impl_2	567	155	141
Hairballs, Hydra_SW	61	50	56

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

Sponza

Crysponza

Hairballs



- Also tested by spawning multiple rays per hit
- Total rays needed to be rendered*:
  - 1 per level => 3
  - 2 per level => 7
  - 4 per level => 21
  - 8 per level => 73

Math: 1 ray + 4 rays + 4*4 rays = 21

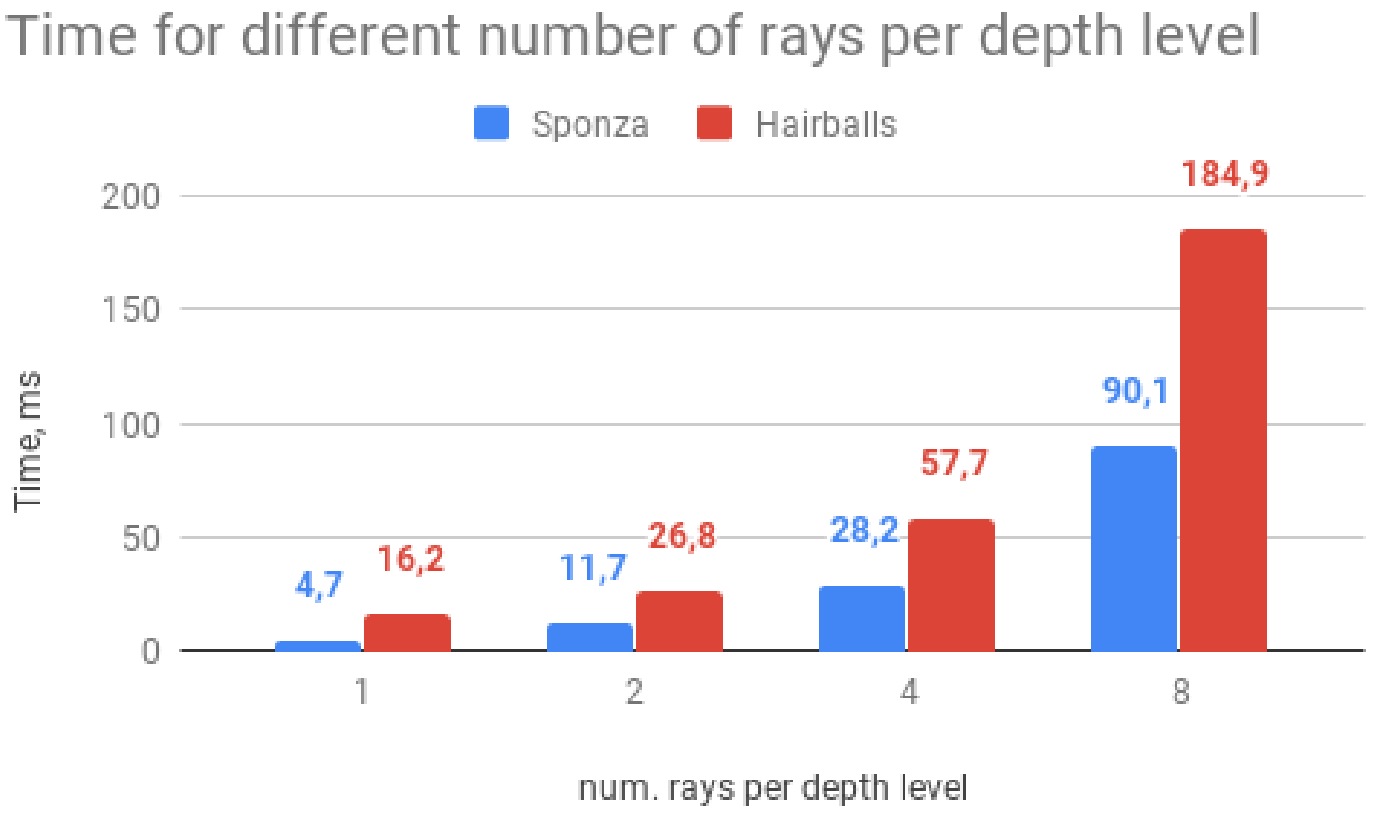
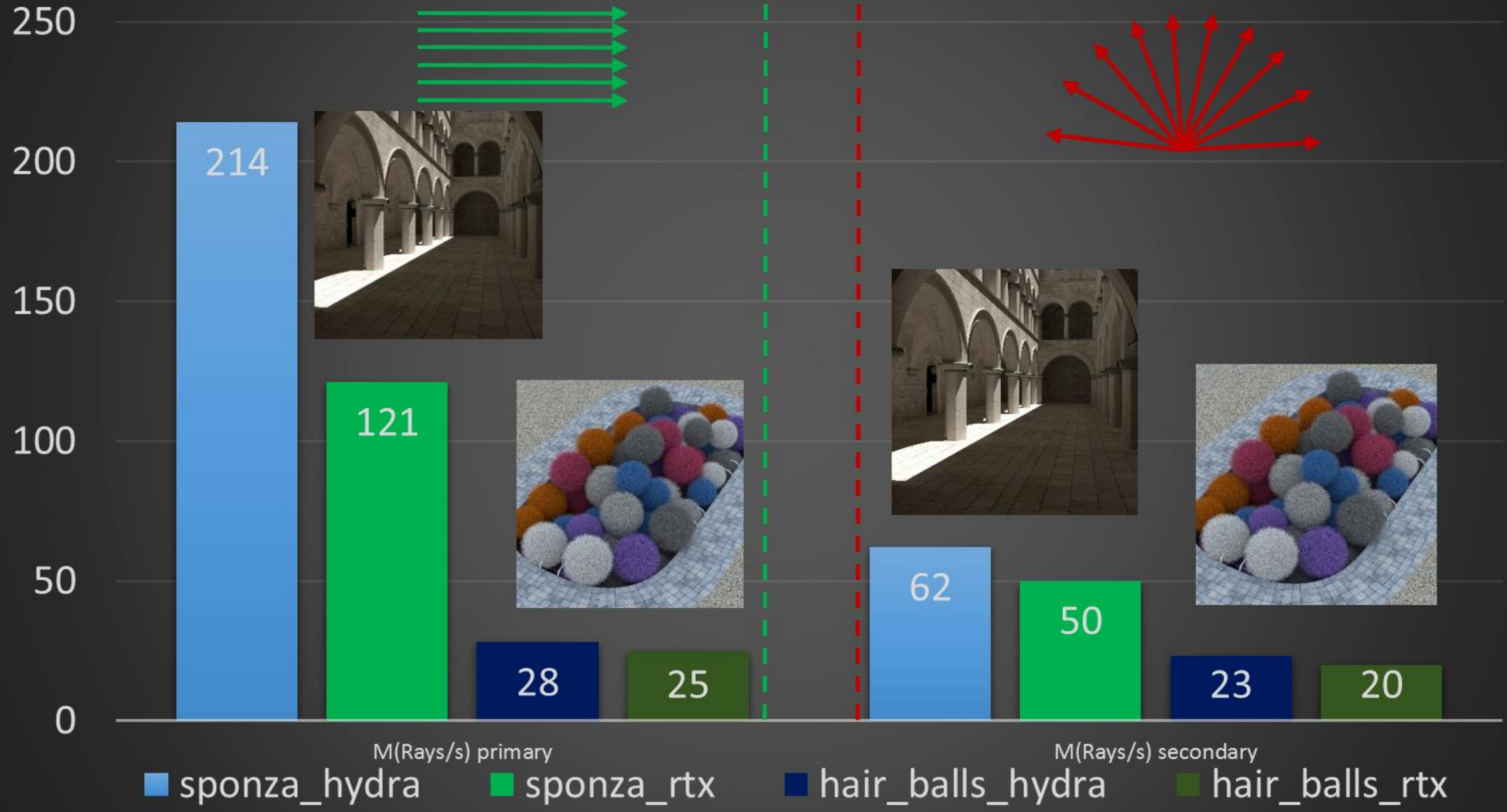
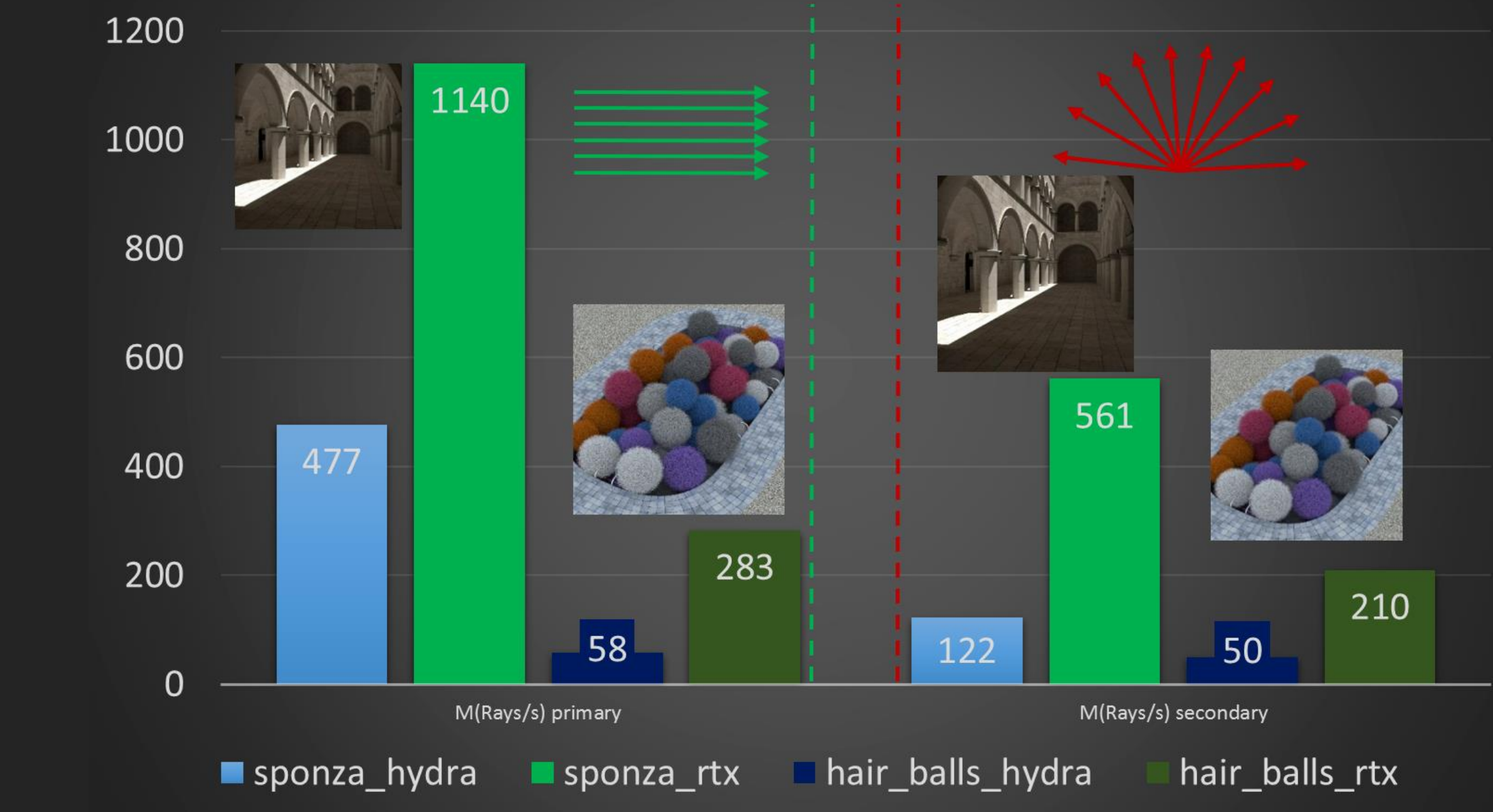


Fig. 1. Time spent by ray tracing "draw call" per frame (1 sample per pixel, 1024 x 1024 resolution) depending on rays traced per depth level. Depth = 3

M(Rays/s); GTX1070 (Hydra vs Nvidia RTX)



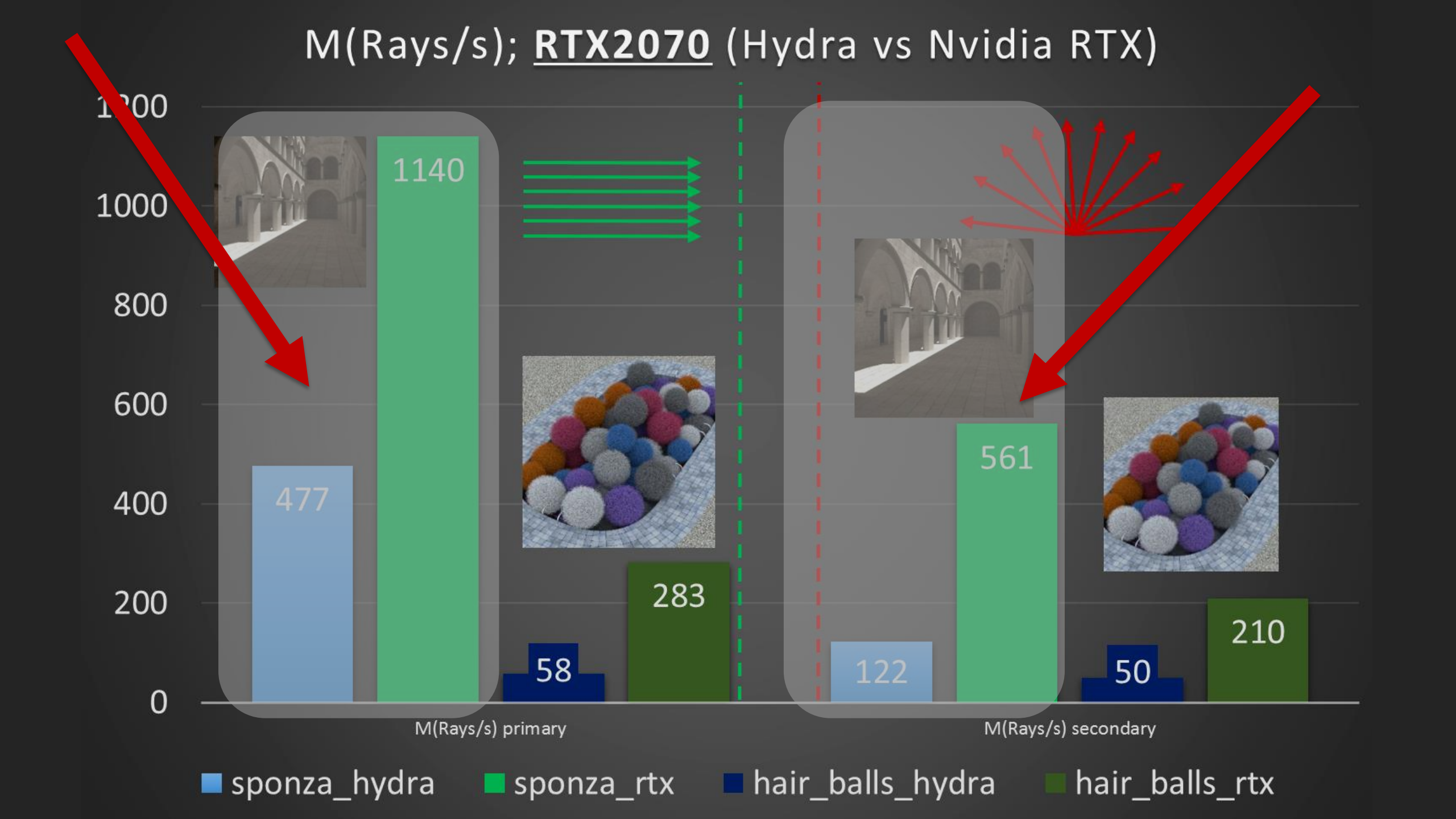
M(Rays/s); RTX2070 (Hydra vs Nvidia RTX)

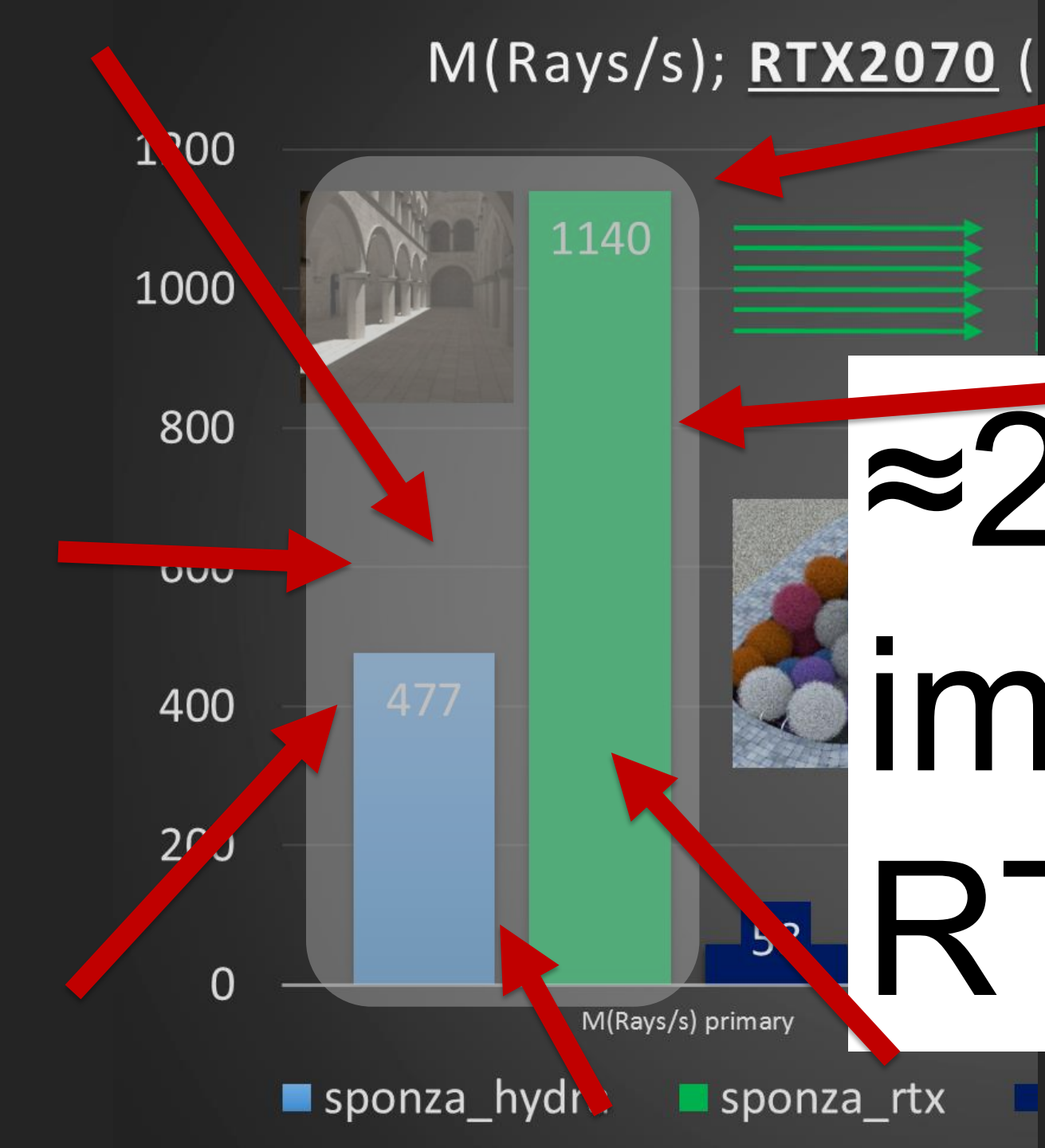


Conclusion 1

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.*

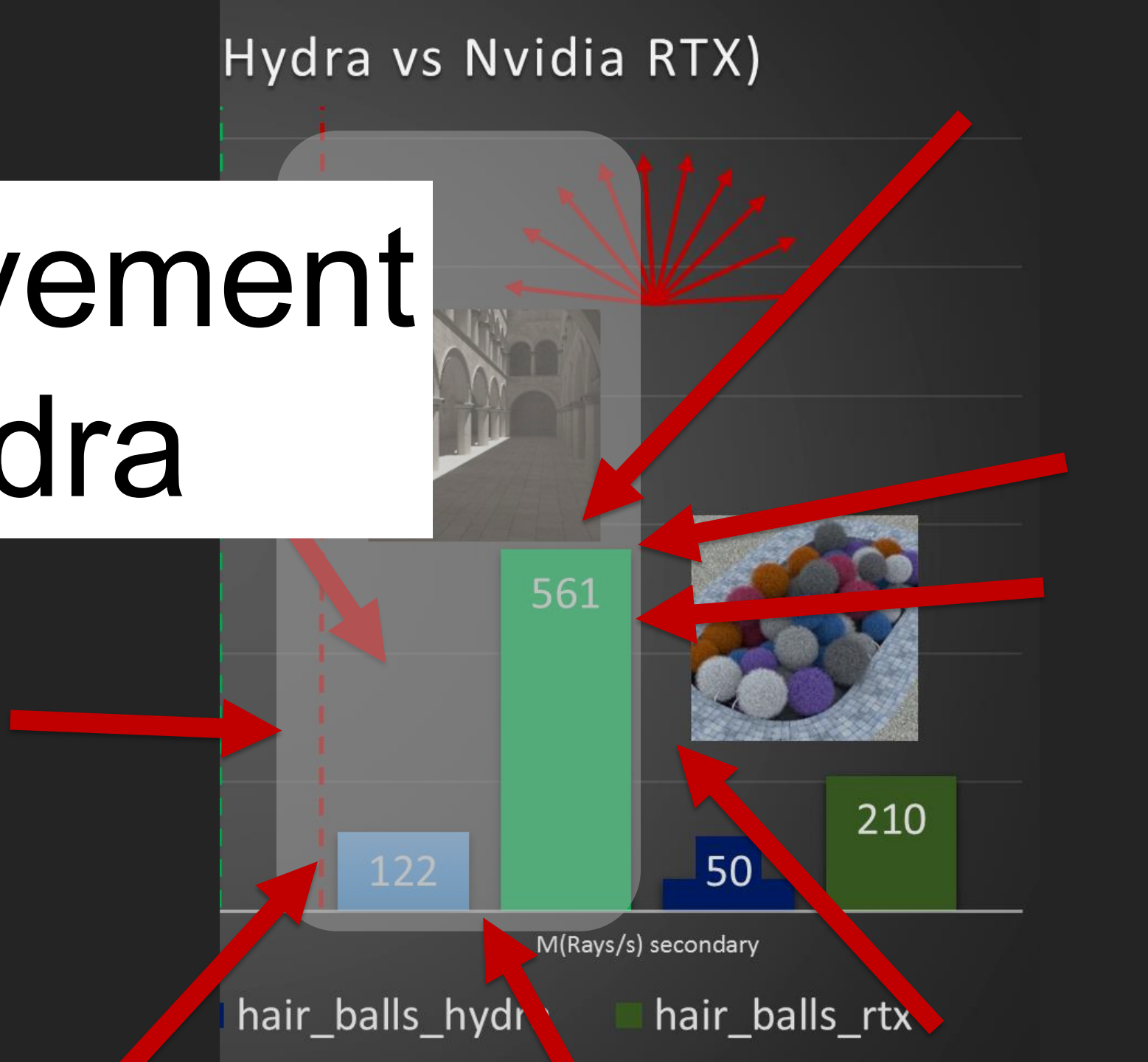
M(Rays/s); **RTX2070** (Hydra vs Nvidia RTX)



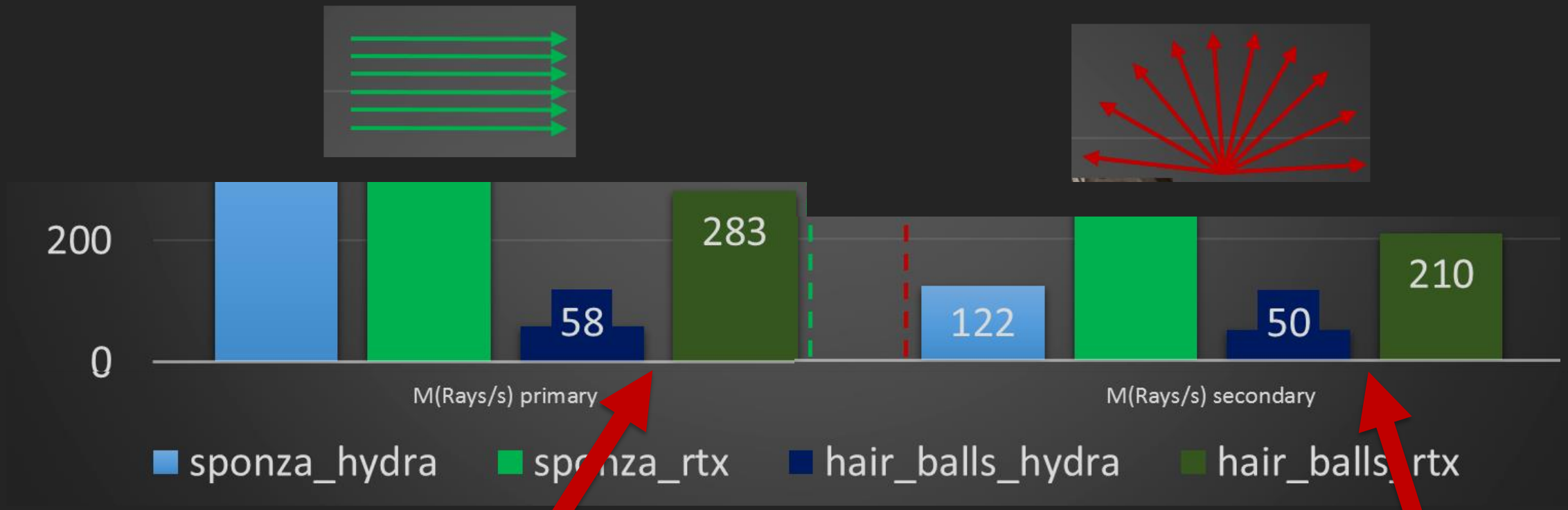


≈2x improvement RTX vs hydra

≈5x improvement RTX vs hydra



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).


200
0
283
58
122
50
210
M(Rays/s) primary
M(Rays/s) secondary
sponza_hydra
sponza_rtx
hair_balls_hydra
hair_balls_rtx

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 con-clusion.

Conclusion II

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.

Remember

- 1 primary +
  1 secondary +
  1 tertiary => 4.7ms

- 1 primary +
  4 secondary +
  8 tertiary = 21 => 28.2ms

6x time difference,
for 7x the rays

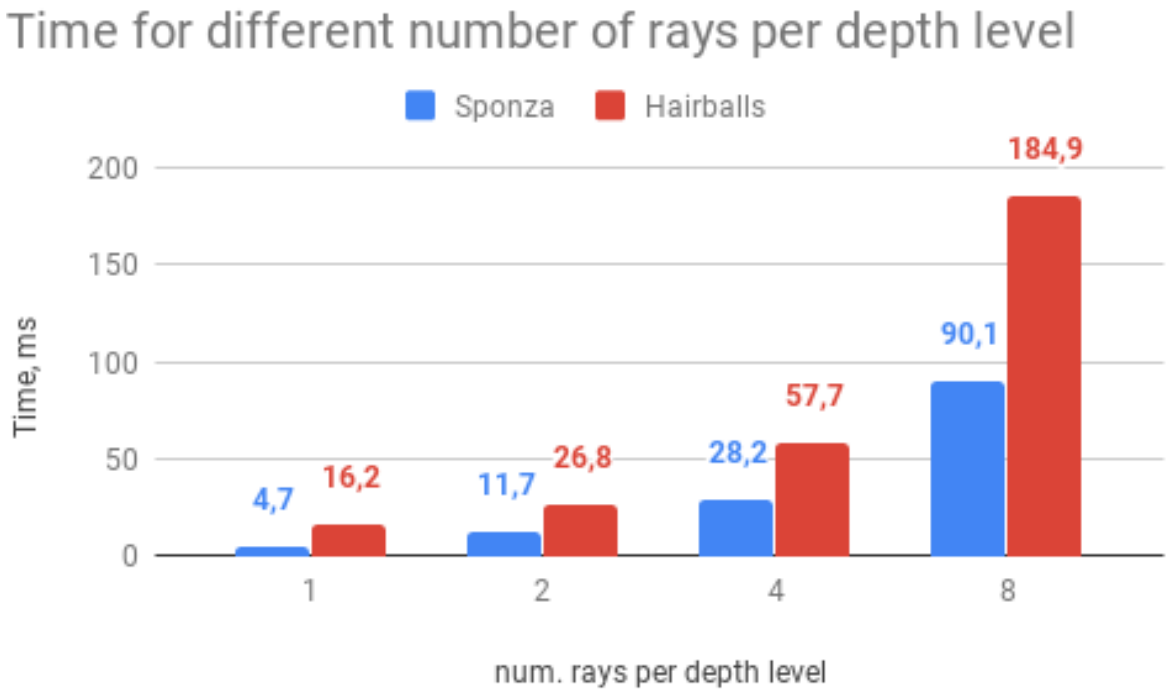


Fig. 1. Time spent by ray tracing "draw call" per frame (1 sample per pixel, 1024 x 1024 resolution) depending on rays traced per depth level. Depth = 3

"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"

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

What?

"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"

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

"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."

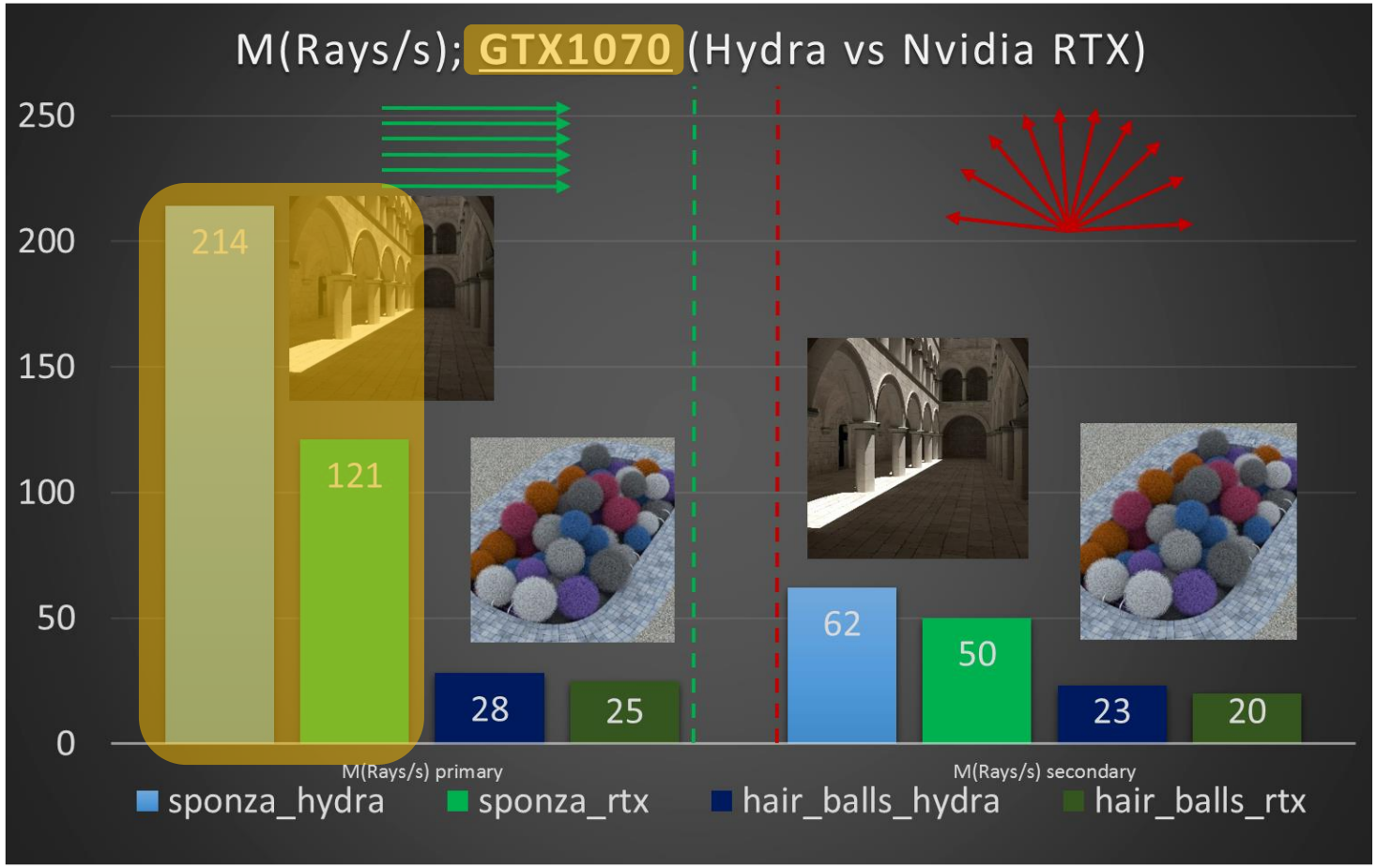
Must be the implementation difference and usage of Vulcan/DirectX

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 3

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). Sec-ond, when comparing 2 slightly different implementa-tions of RTX in Vulkan and DX12 we have found dra-matic 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.

- Hydra uses separate smaller kernels
- RTX software fallback uses “uber kernel”
- Not that efficient on regular GPU’s => hydra beats it



Conclusion 4

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.

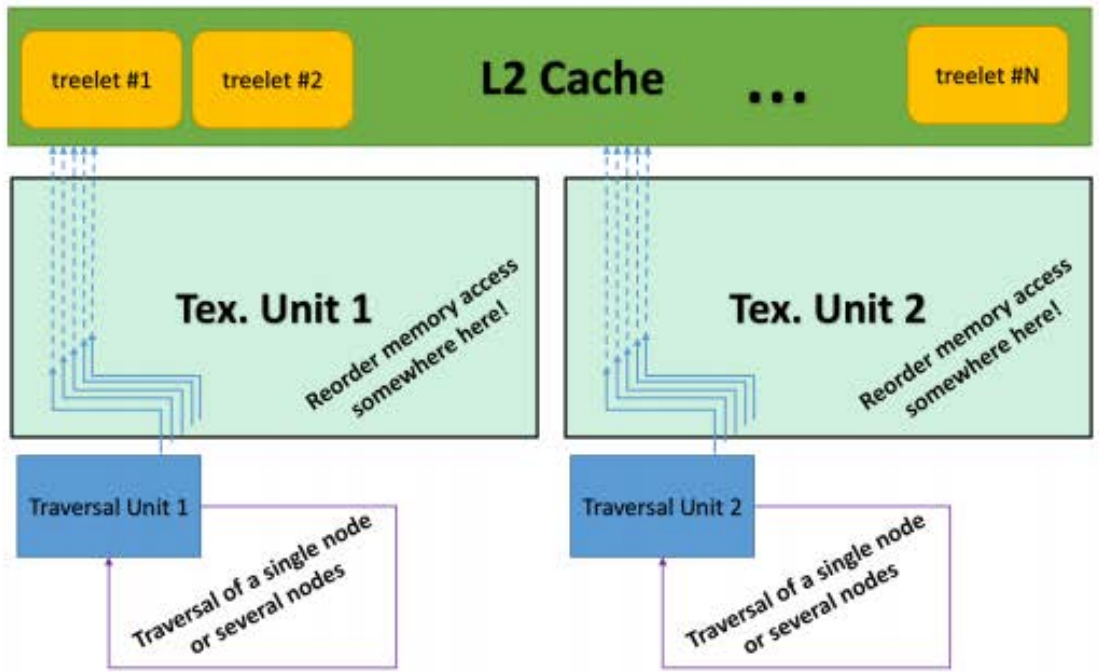
- **ExecuteIndirect**: GPU work creation now allows ray tracing. This enables shaders on the GPU to invoke and control ray tracing without an intervening round-trip back to the CPU. This is useful for adaptive ray tracing scenarios like shader-based culling, sorting, classification, and refinement.

Source: https://news.developer.nvidia.com/directx-12-ultimate-preview/

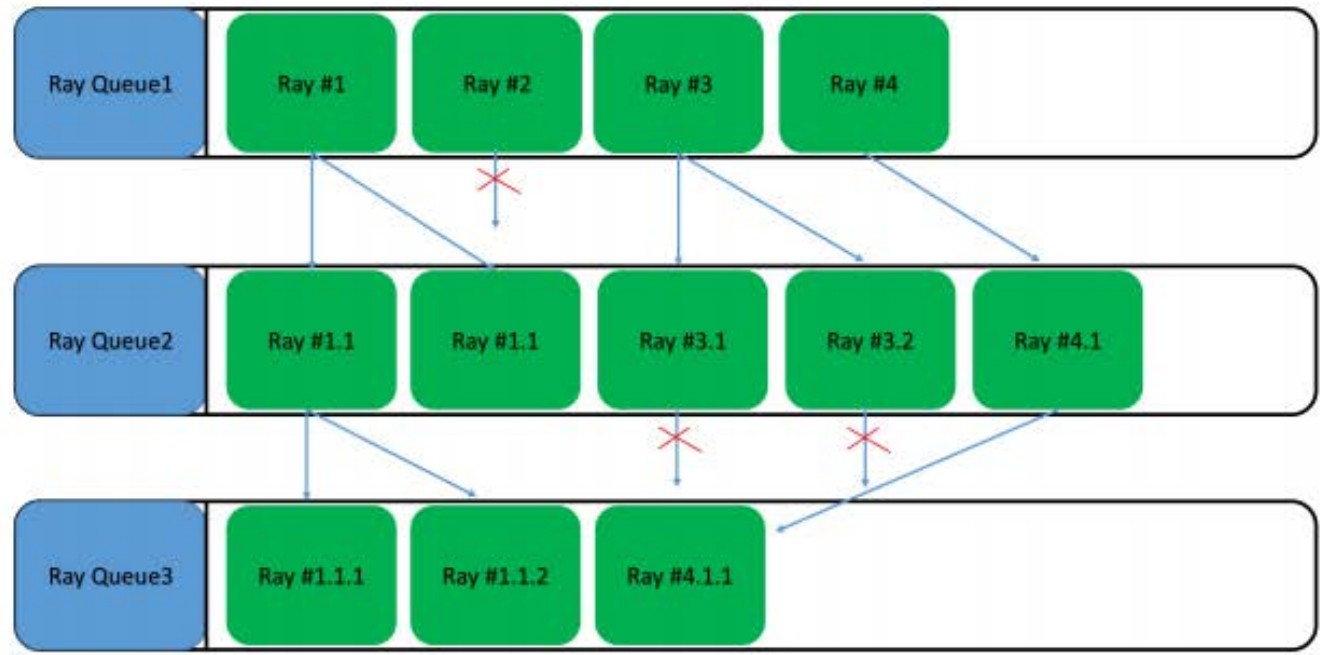
Final Conclusion

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.



Hardware part



Software part
(HW support for GPU Work Creation is needed)

TL;DR

- Nvidia optimizes RAM access during ray tracing.
  - To improve performance while traversing the Bounding volume hierarchy tree
- Nvidia RTX implements some ray grouping/ray-sorting.
- Uses GPU work creating of rays
- Work really well