Master Of Science in Computer Engineering

- Computer architecture -

Fine tuning
of a
ray tracing application

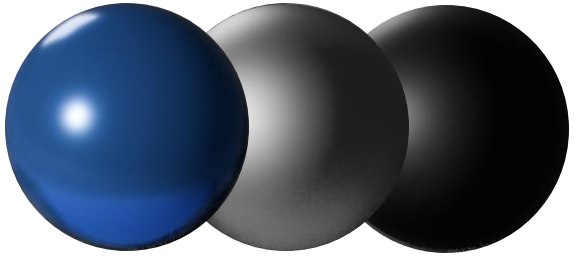
Francesco De Lucchini
Salvatore La Porta

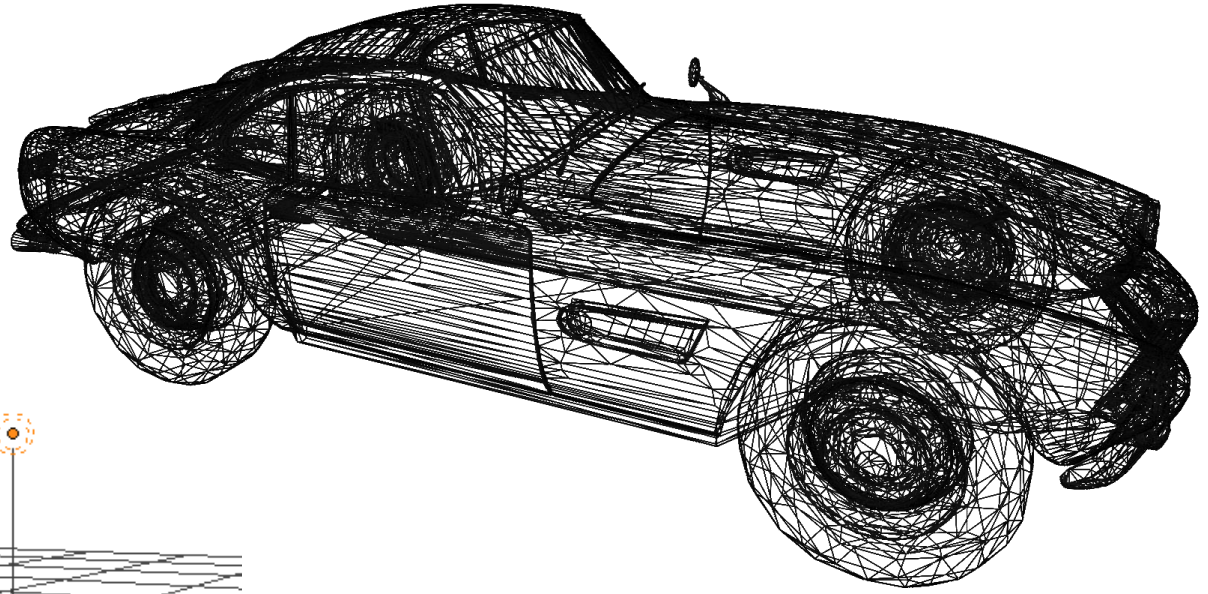
A.Y. 2024 / 2025

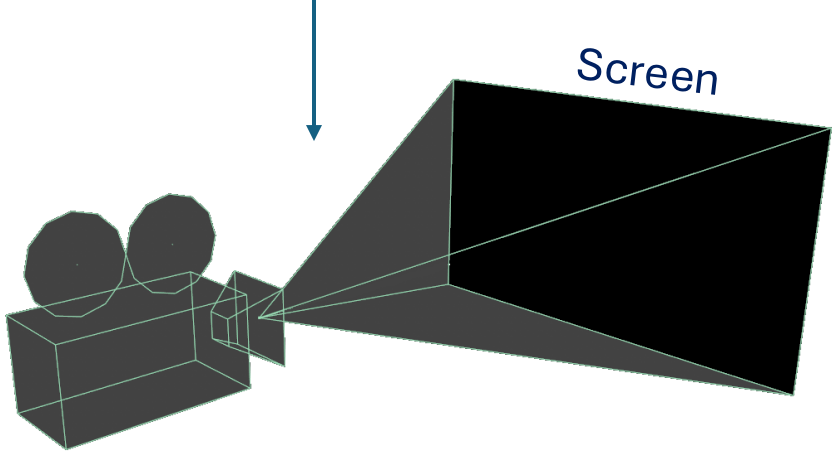
Ray tracing in a nutshell

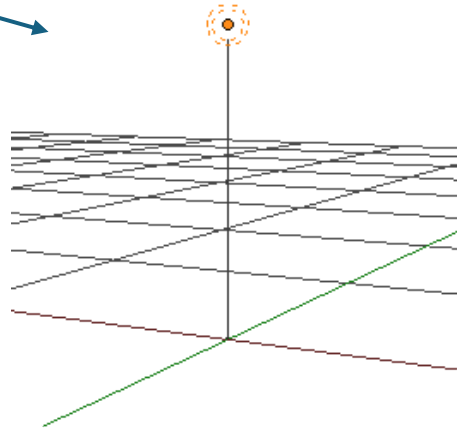
Inputs:

- Materials
- Triangles
- Lights
- Camera



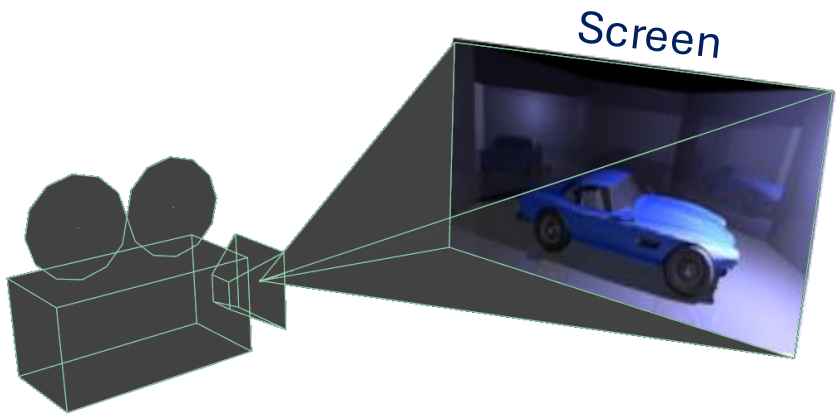




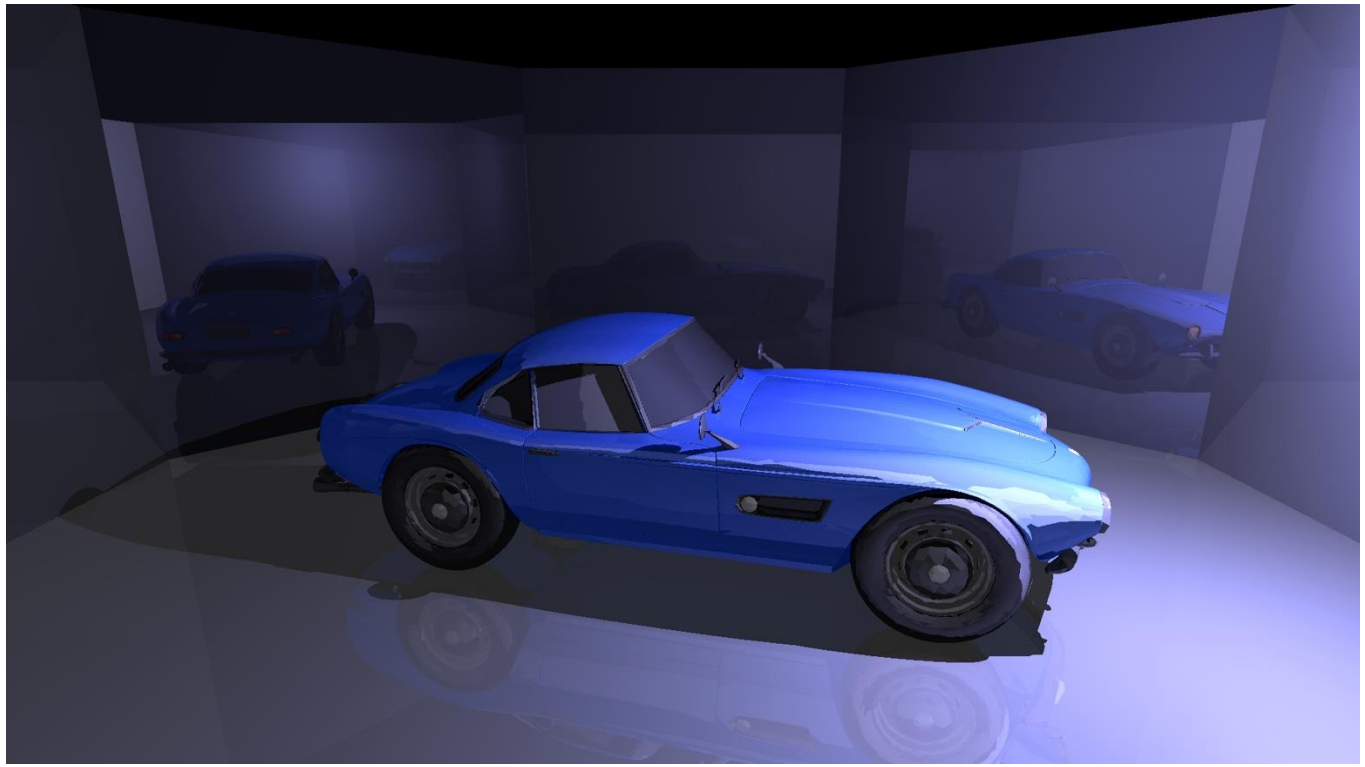


Ray tracing in a nutshell

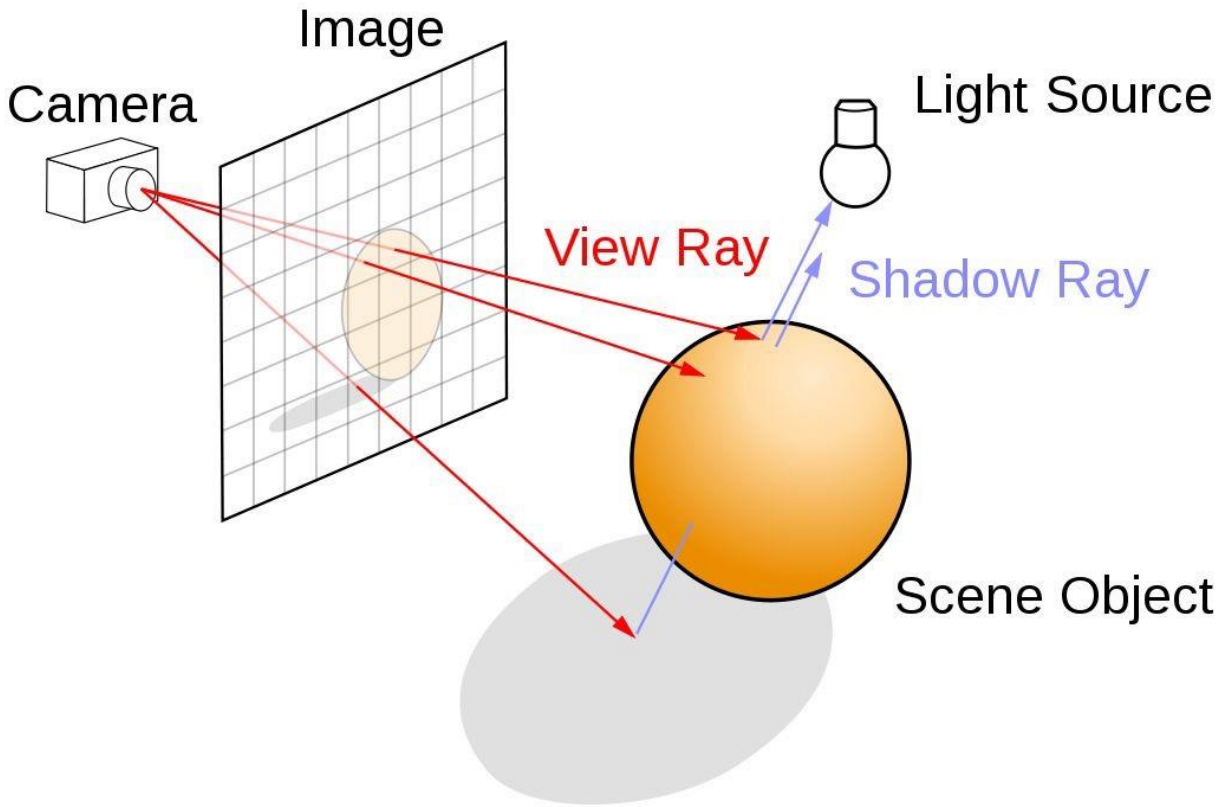
Output: The rendered image



This is the actual output of the application

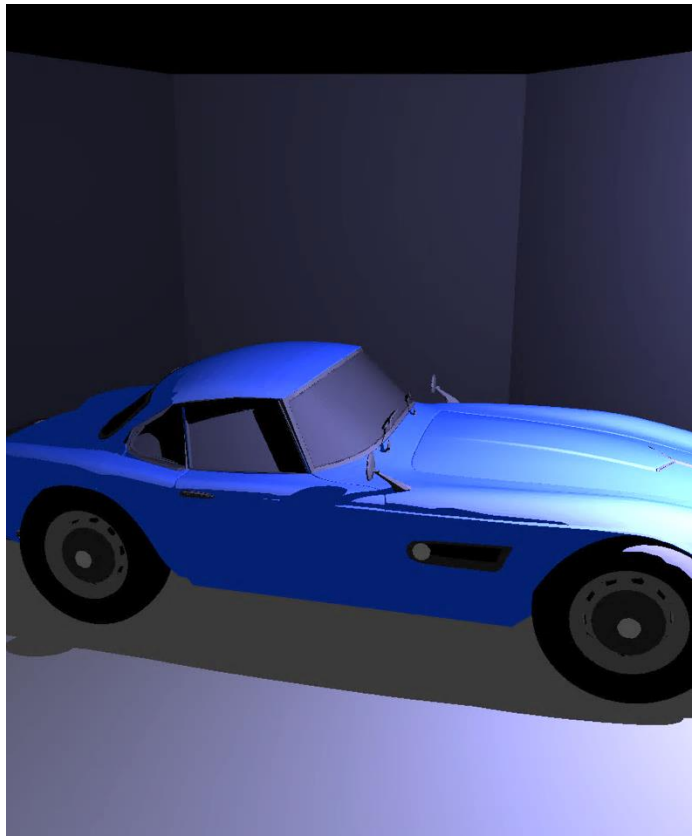


Ray tracing - Intuition



Ray Tracing | NVIDIA Developer

Ray tracing - Intuition



No bounces

1 bounce per ray

2 bounces per ray

Ray tracing - Computational complexity

Naïve ray tracing's asymptotic complexity is roughly

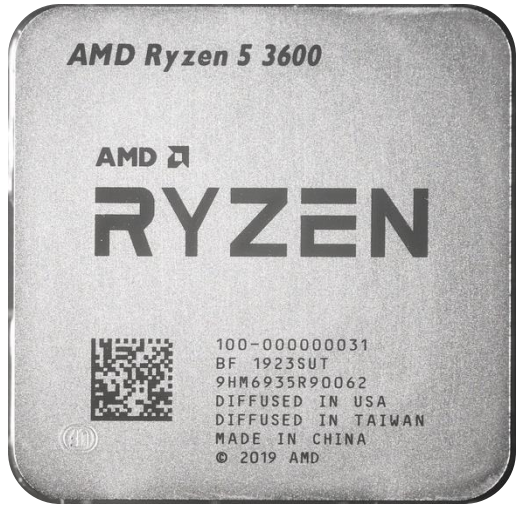
$$O(Pixels \cdot Triangles)$$

For example, consider the "BMW 507" scene rendered in Full HD:

- $Pixels = 1920 \cdot 1080 = 2'073'600$
- $Triangles = 45'999$

$\Longrightarrow$ **Billions of COMPLEX operations** (geometry in $\mathbb{R}^3$)

Chapter I
CPU ray tracing



Test system specifications

Hardware (CPU)	
Name	**AMD Ryzen 5 3600**
Launch	Q3 2019
Architecture	**Zen 2 “Matisse” (x86-64)**
Process size	7nm @ TSMC
Clock frequency	3.6 GHz up to 4.2 GHz
Cores	**6**
Threads	**12 (AMD SMT)**
L1I cache	32 KB (per core)
L1D cache	32 KB (per core)
L2 cache	512 KB (per core)
L3 cache	32 MB (shared)

Software	
OS	Zorin OS 17.3
Kernel	Linux 6.8.0-57
Compiler	GCC 11.4.0
Profiler	AMD µProf 5.0

Testing methodology

Parameters of the experiments:

- Scene: “BMW 507”
- Bounces per ray: 4
- **Resolution: varies according to the desired workload**

For each experiment, **the mean of 30 independent runs** is taken

(*When not vanishingly little, 99% confidence intervals are shown*)

Timer starts after the scene is loaded in memory

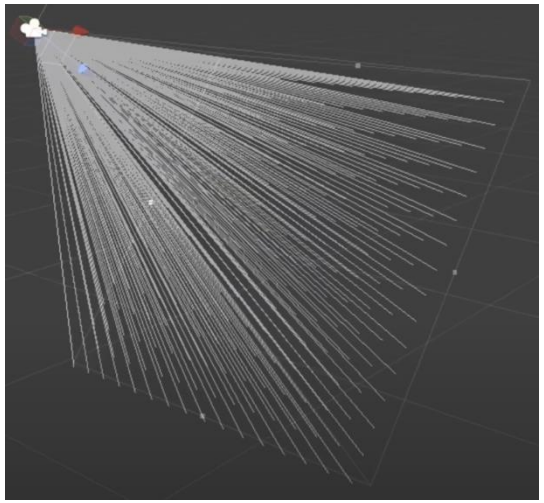
Goal: 1 FPS @ 1080p

Thread organization

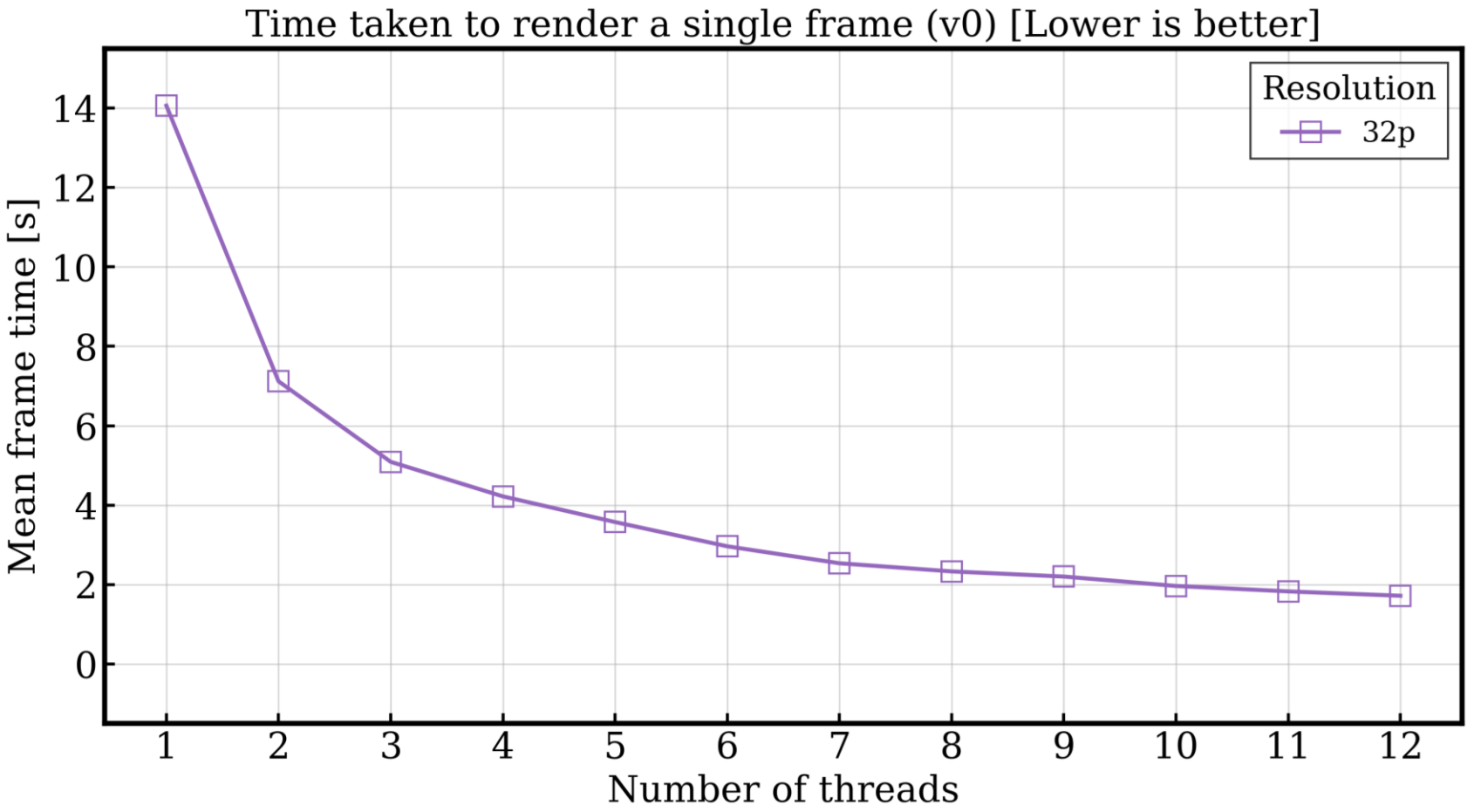
Ray tracing is **embarrassingly parallel**:

- Each ray (pixel) is independent from the others
- The shared memory (the scene) is read only

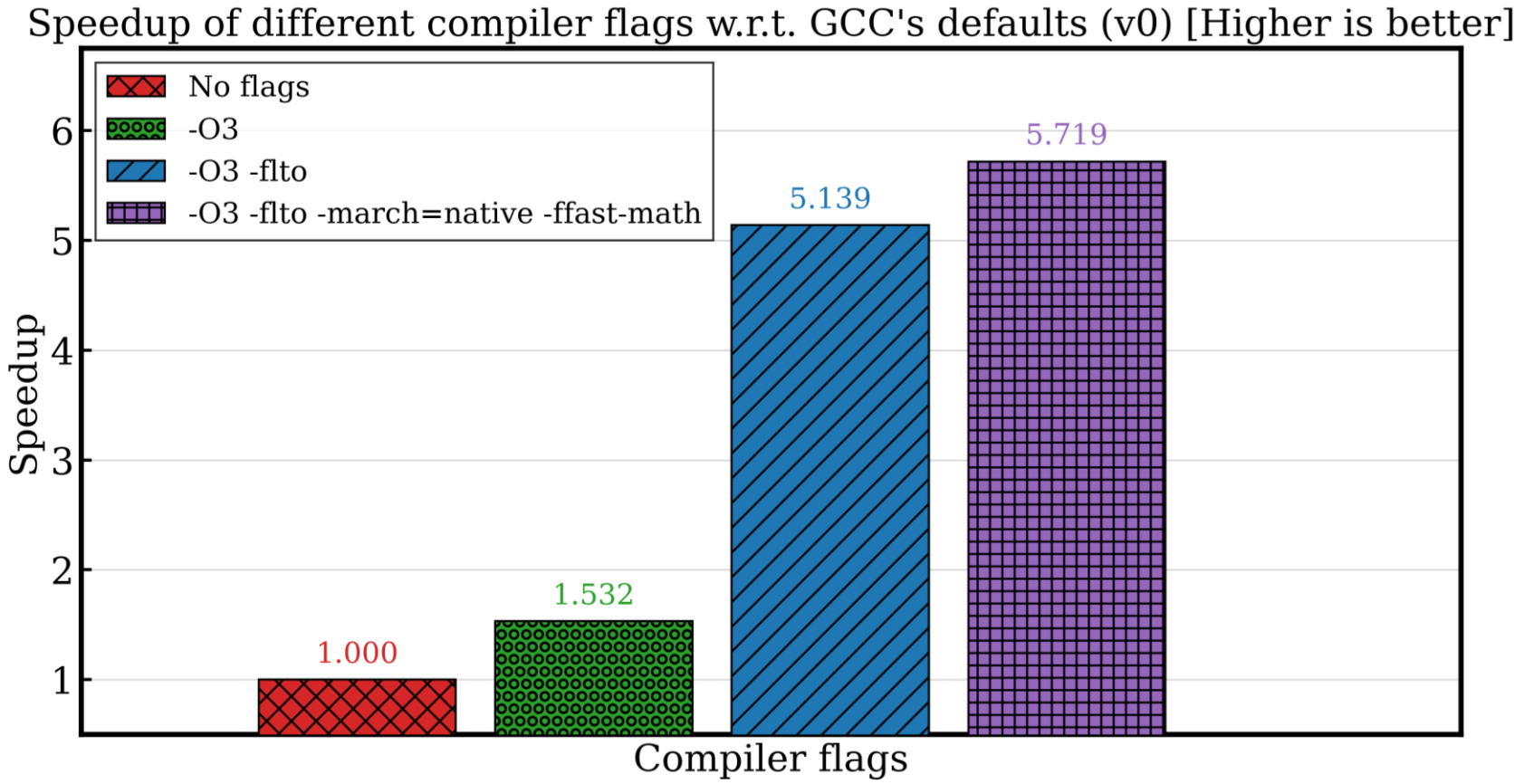
⟹ Barely need synchronization!



Version 0 “Naïve” - Frame time



Side note - Compiler flags matter!



Ray tracing is ***HEAVY***

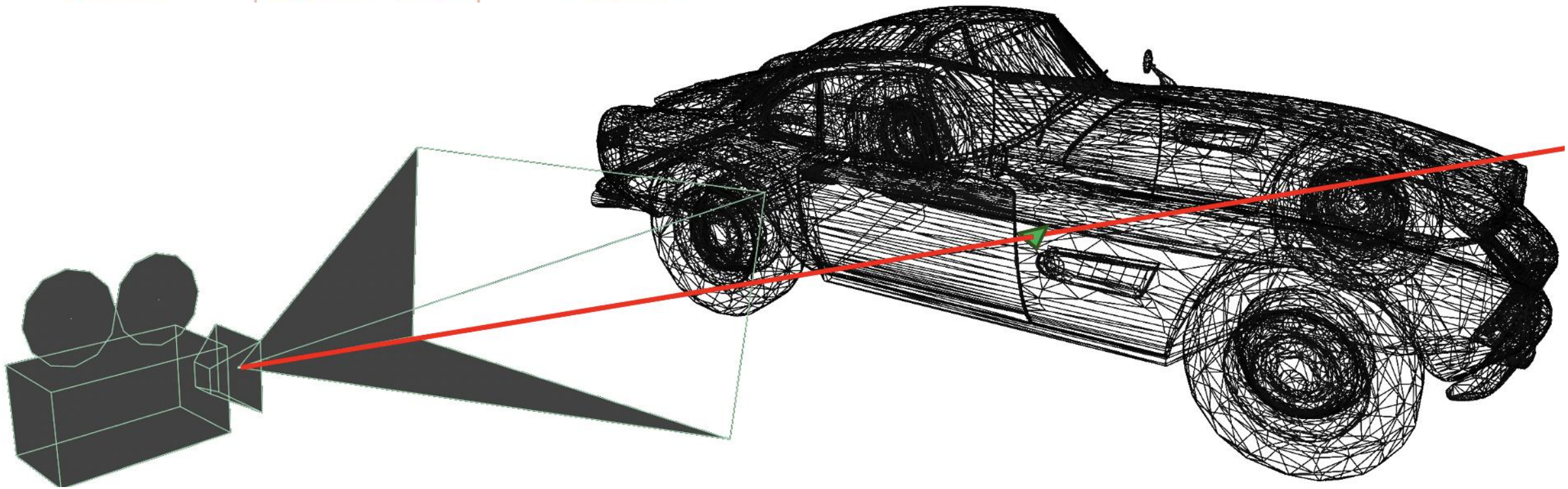
- Even with 12 threads, the naïve version of the algorithm takes about 1.7 seconds (≈0.59 FPS) to render in 64x32
- That is the same resolution of the RCA Studio II (1977 video game console)

RCA Studio II - Wikipedia

Where to improve?

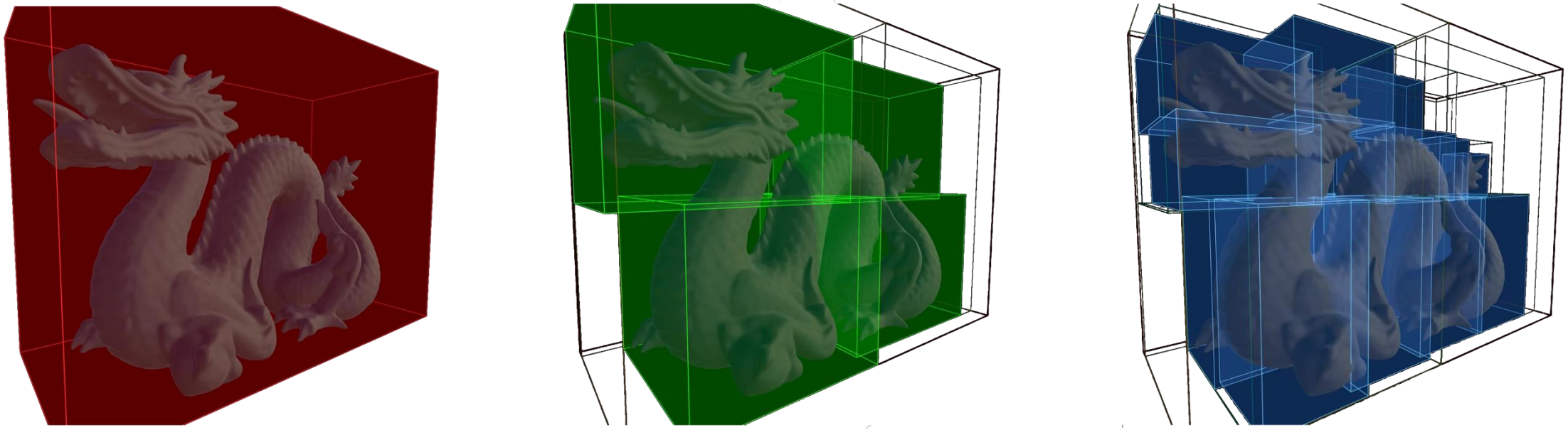
Functions	Modules	CPU_TIME(s) ▼
hit_triangle	raytracer.exe	84.8487%
raytrace	raytracer.exe	14.9085%

The *hit_triangle* function checks if a line (ray) intersects a triangle in $\mathbb{R}^3$



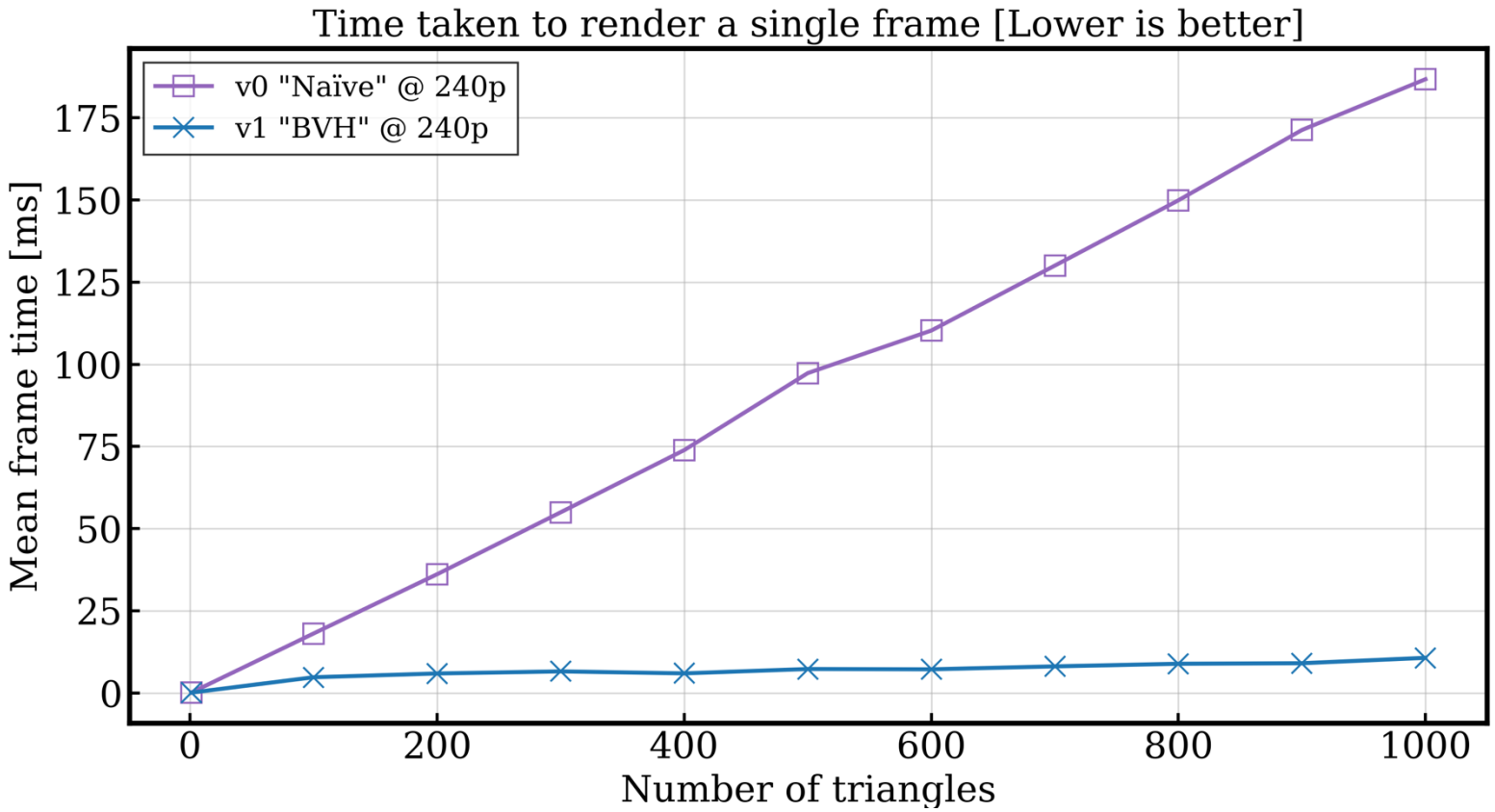
Bounding **V**olume **H**ierarchy

The *hit_triangle* function can't be optimized (already state of the art)
⟹ The best solution is **to reduce the number of times it is called**



Sebastian Lague - Coding Adventure: Optimizing a Ray Tracer (by building a BVH)

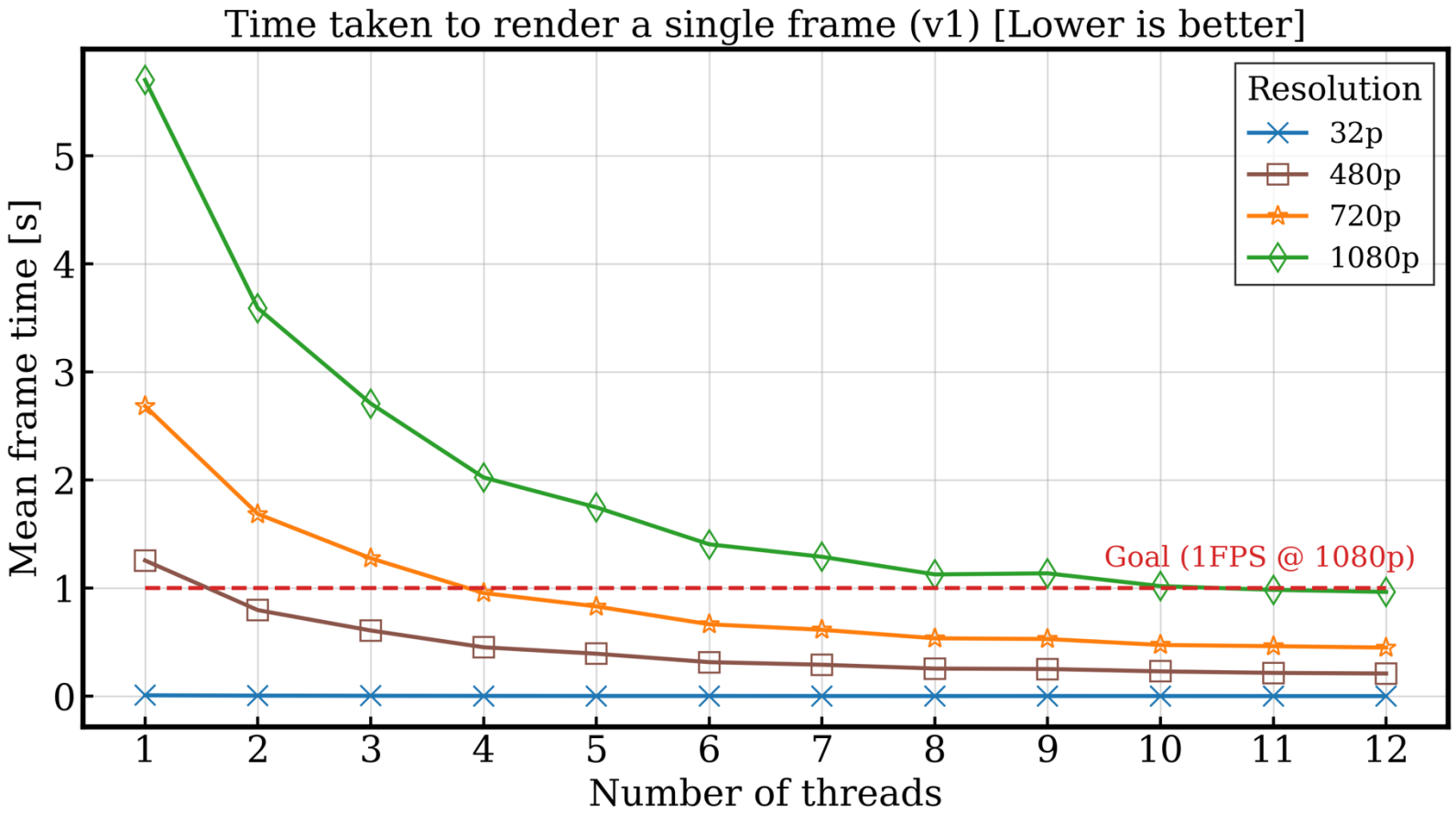
v1 “BVH” vs v0 “Naïve” - Complexity



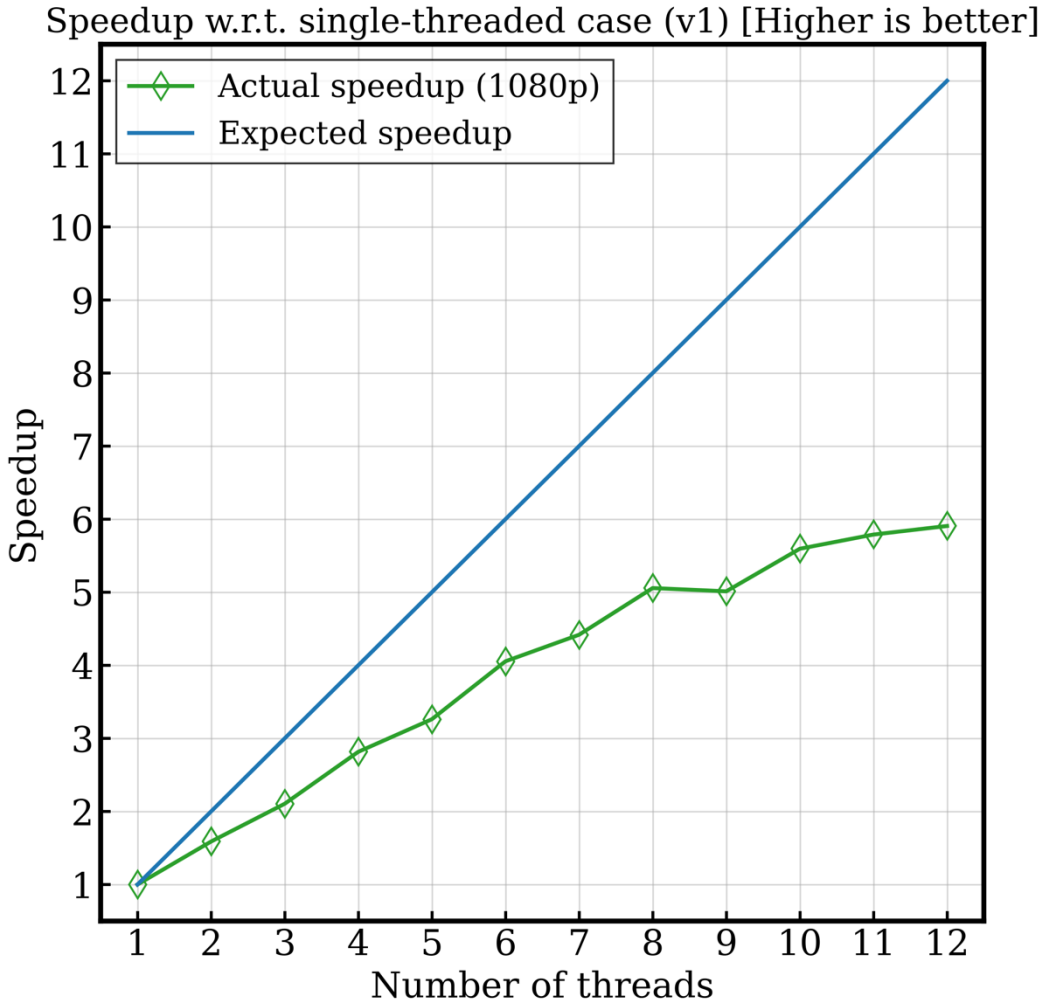
BVH scales complexity down to $O(Pixels \cdot \mathbf{log_2}(Triangles))$

This leads to a **speedup of around 1000x** with $\approx 46k$ triangles

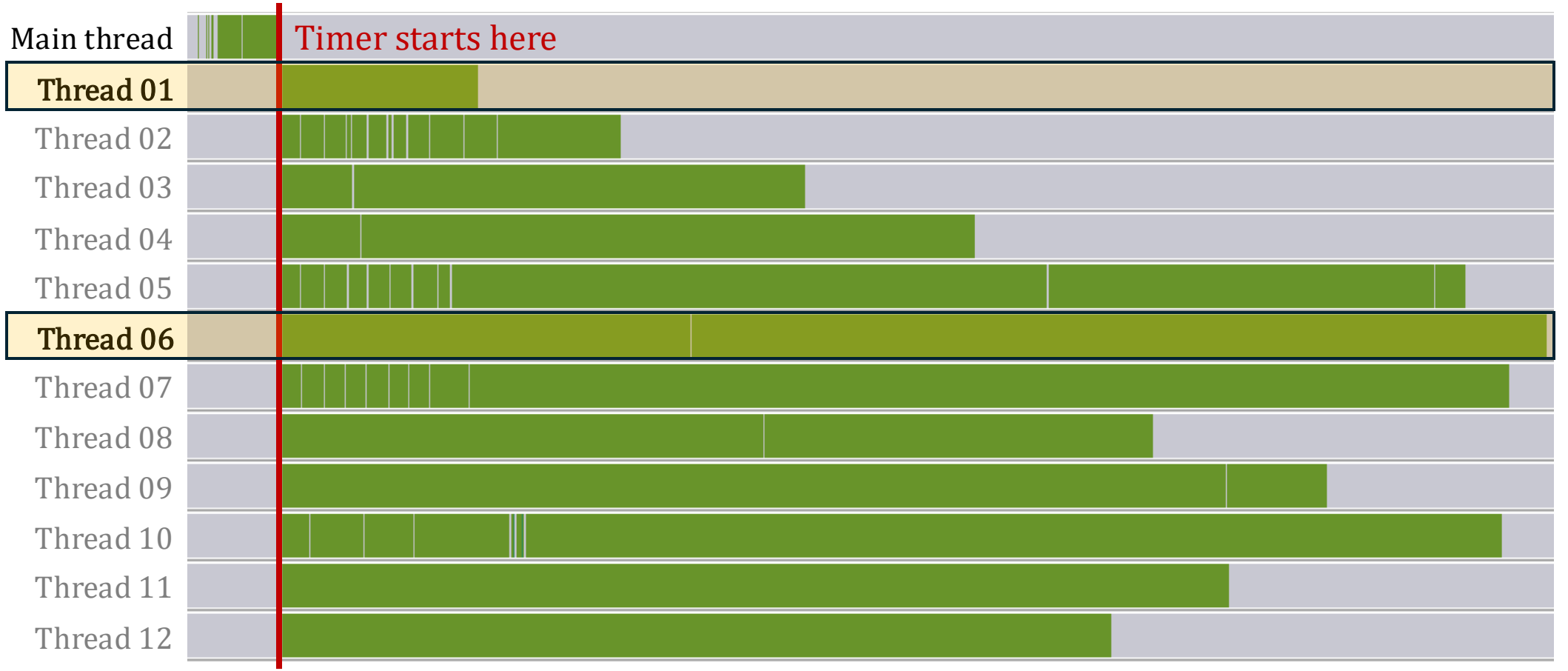
Version 1 “BVH” – Comparison with the goal



Version 1 “BVH” – Speedup



Version 1 “BVH” – Concurrency



Full concurrency is achieved only 15% of the time!

Version 1 “BVH” – Concurrency

Threads render parts of the image with very different degrees of difficulty

Version 2 “BVH + FTWM”

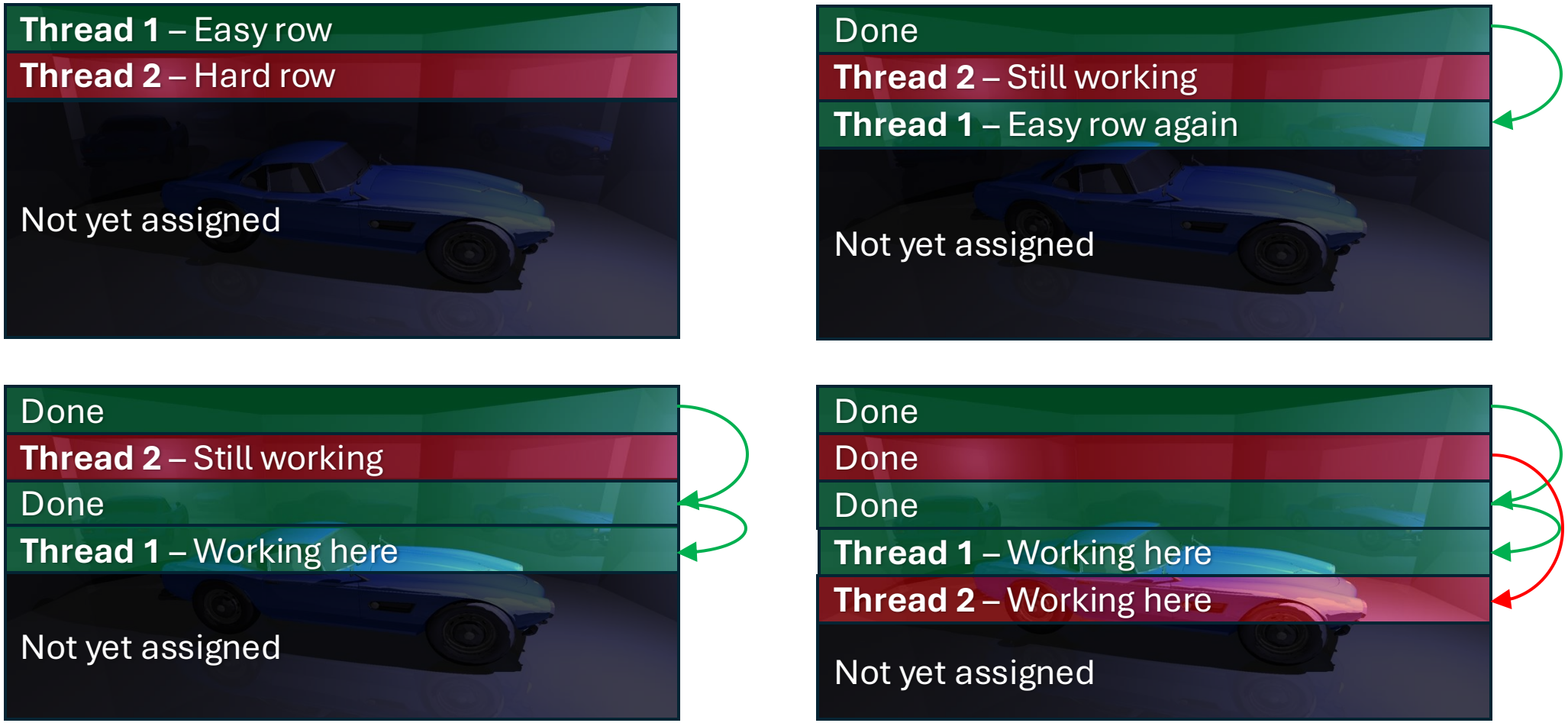
New scheduling policy - **F**aster **T**hreads **W**ork **M**ore:

- Each thread renders the first available row of pixels
- This can be easily achieved by using an **atomic integer**

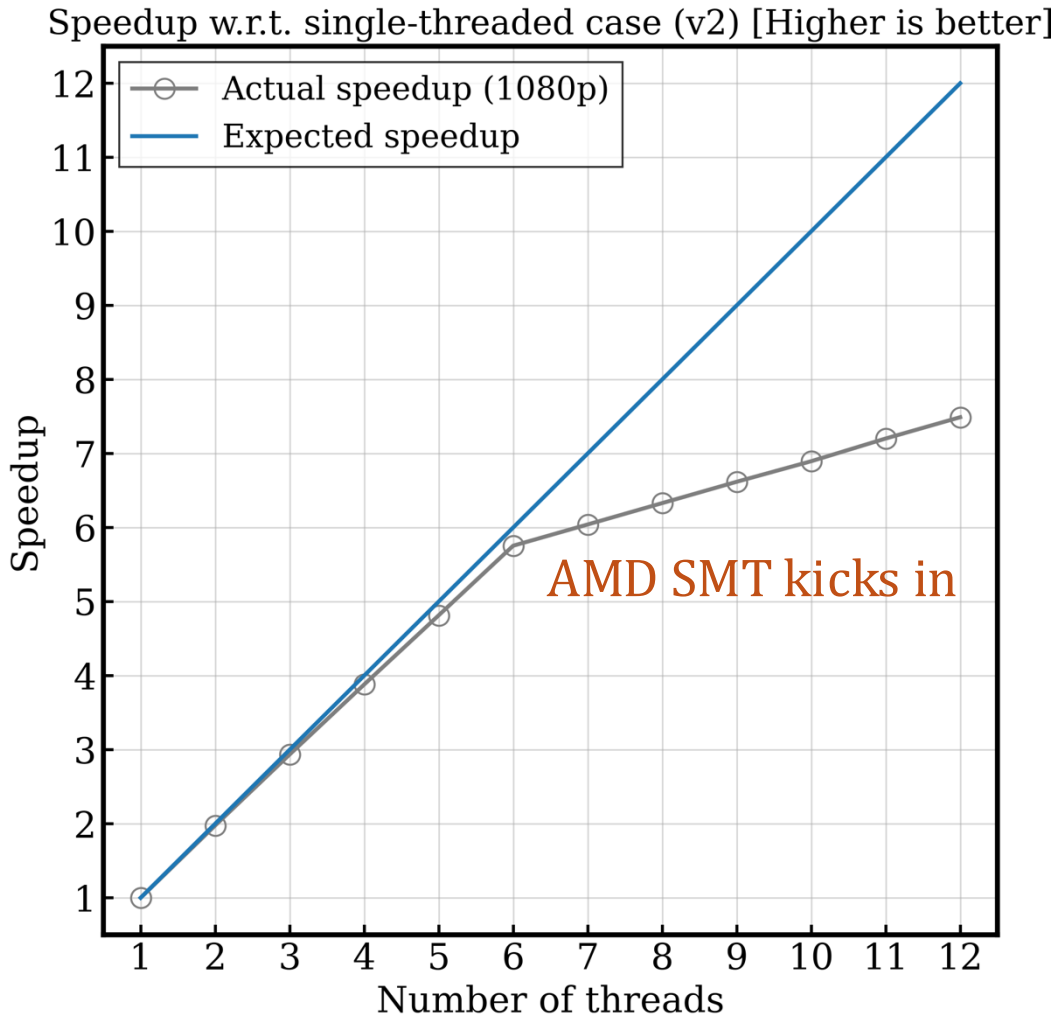
```
// Global code:
atomic_int row_idx = 0;

// Thread code:
while (1) {
    int row_to_render = atomic_fetch_add(&row_idx, 1);
    // ...
}
```

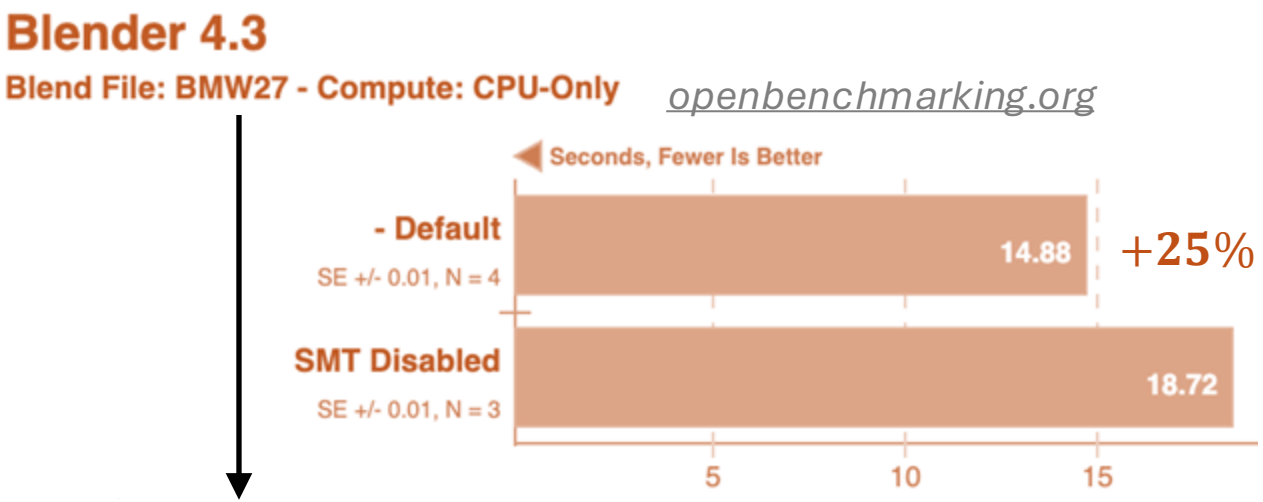
$\Longrightarrow$ atomic_fetch_add(&row_idx, 1) is translated to **LOCK** INCL row_idx

Faster **T**hreads **W**ork **M**ore - Example



Version 2 “BVH + FTWM” - Speedup



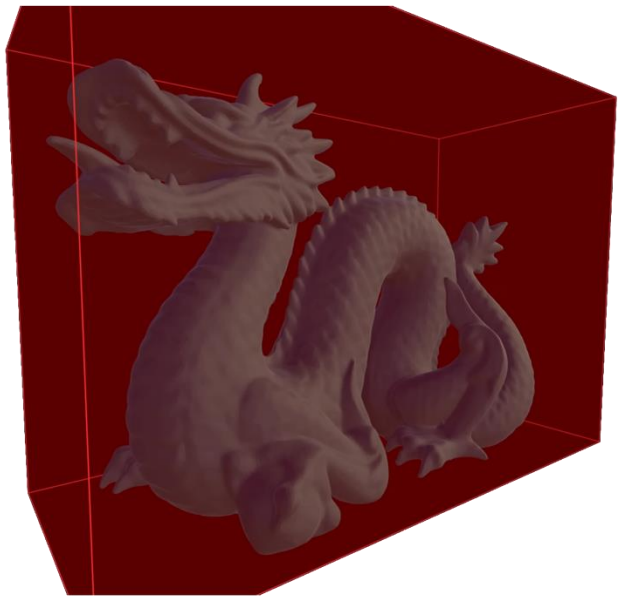


Blender BMW27 - TechPowerUp

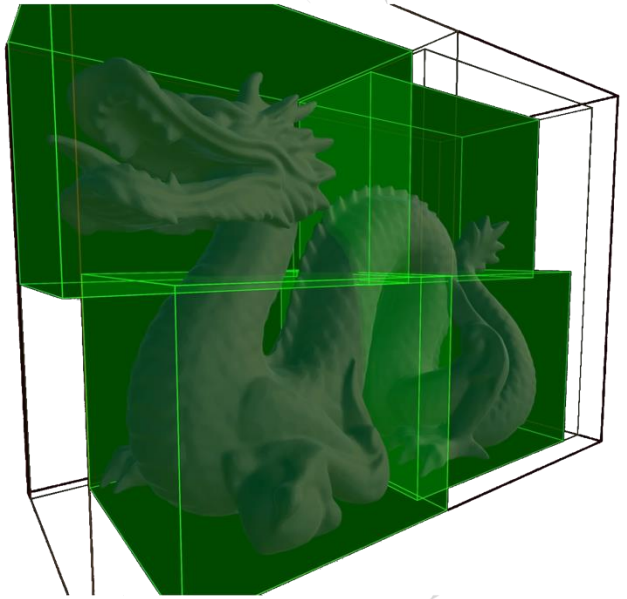
Version 2 “BVH + FTWM” - Concurrency

Full concurrency is achieved most of the time

Side note – Version 3 “Optimal BVH”



How to split the boxes?

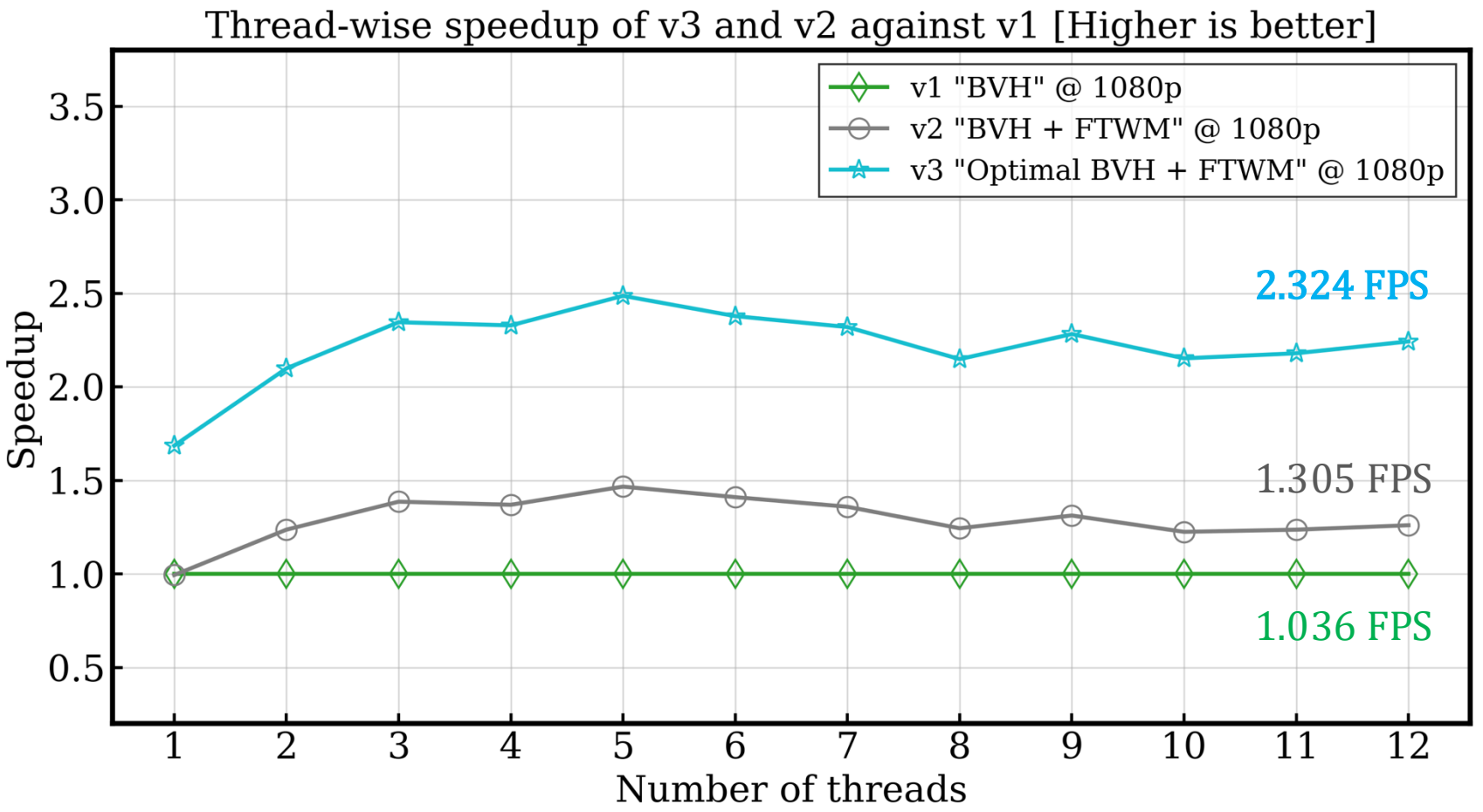


Intuition:

- v1 and v2 splitted the boxes in a random way
- v3 splits the boxes taking into account their **surface area**

The resulting binary tree (BVH) of v3 is, on average, less deep ⟹ v3 is better

Final results



Chapter II
GPU ray tracing

Test system specifications

Hardware (GPU)	
Name	**RTX 2060 Super**
Launch	Q2 2019
Architecture	**Turing**
Process size	12nm @ TSMC
VRAM	8GB GDDR6 (256-bit bus)
CUDA cores	**2176**
Streaming Multiprocessors	**34** (64 CUDA cores each)
Warp size	**32**
L1 cache	64 KB (per SM)
L2 cache	4 MB (shared)

Software	
OS	Zorin OS 17.3
Kernel	Linux 6.8.0-59
CUDA toolkit	12.9
Profiler	Nsight Compute 2025.2

Testing methodology

Parameters of the experiments:

- Scene: “BMW 507”
- Bounces per ray: 4
- **Resolution: 1080p (fixed)**

For each experiment, **the mean of 30 independent runs** is taken

(*When not vanishingly little, 99% confidence intervals are shown*)

Timer starts after the scene is loaded in memory

Goal: 60 FPS @ 1080p

Porting to CUDA - Recursive functions

Although CUDA supports recursion,
the stack size is (very) limited (by default, just 1KB)

```
size_t stack_size;
cudaDeviceGetLimit(&stack_size, cudaLimit::cudaLimitStackSize);
printf("Default stack size: %zu bytes\n", stack_size);
/* Default stack size: 1024 bytes*/
```

Moreover, it's best practice to spawn threads whose
memory footprint can be calculated at compile time

Therefore, **all recursive functions were converted into iterative ones**

Porting to CUDA - Thread organization

In CUDA, **threads are executed in groups of 32** (warp size)

⟹ The number of threads per block should be a multiple of 32

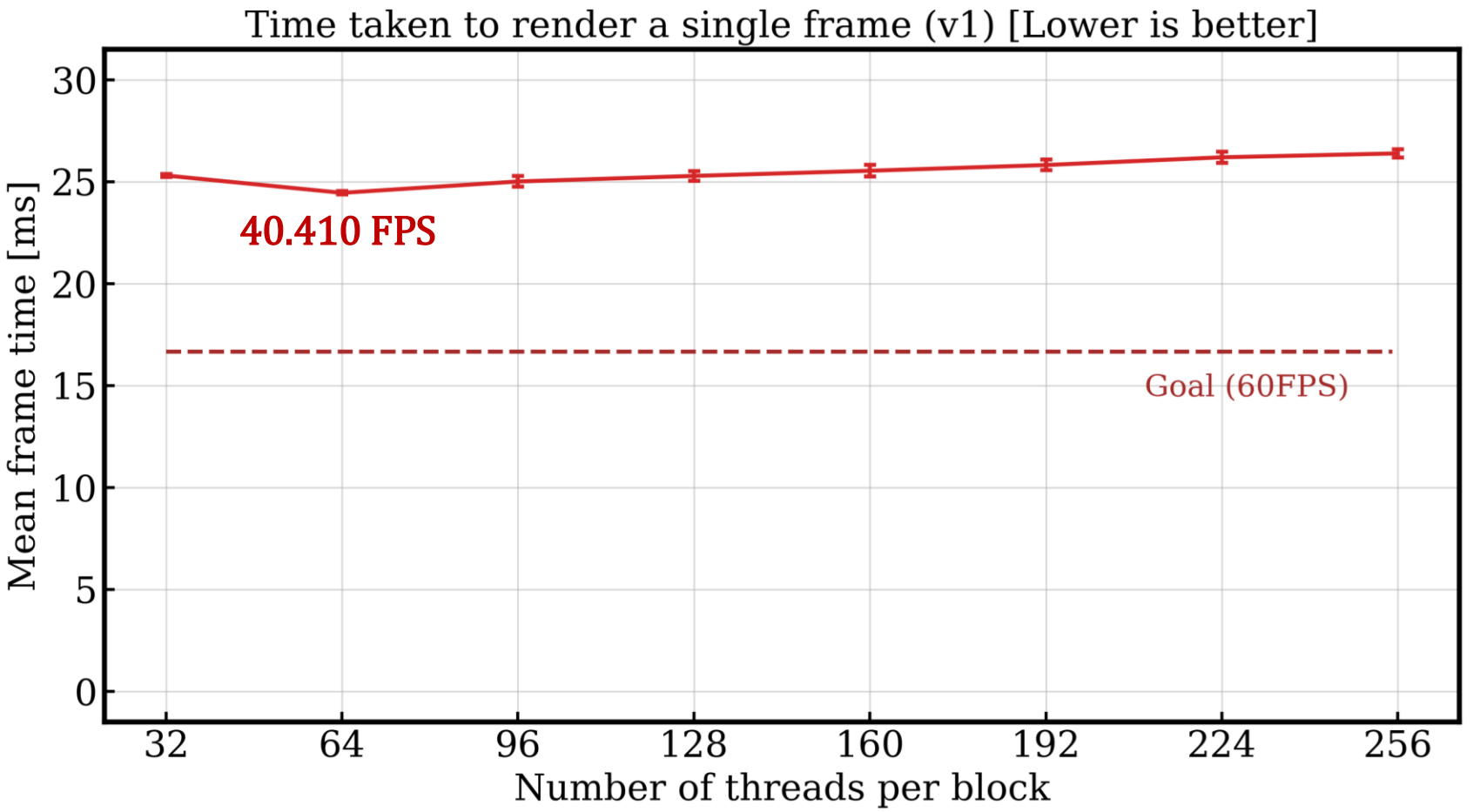
“GPUs are designed to handle a large number of concurrent, lightweight threads”
CUDA C++ Best Practices - Section 2.1

⟹ For example, in a ray tracing application, **each thread can render a single pixel**

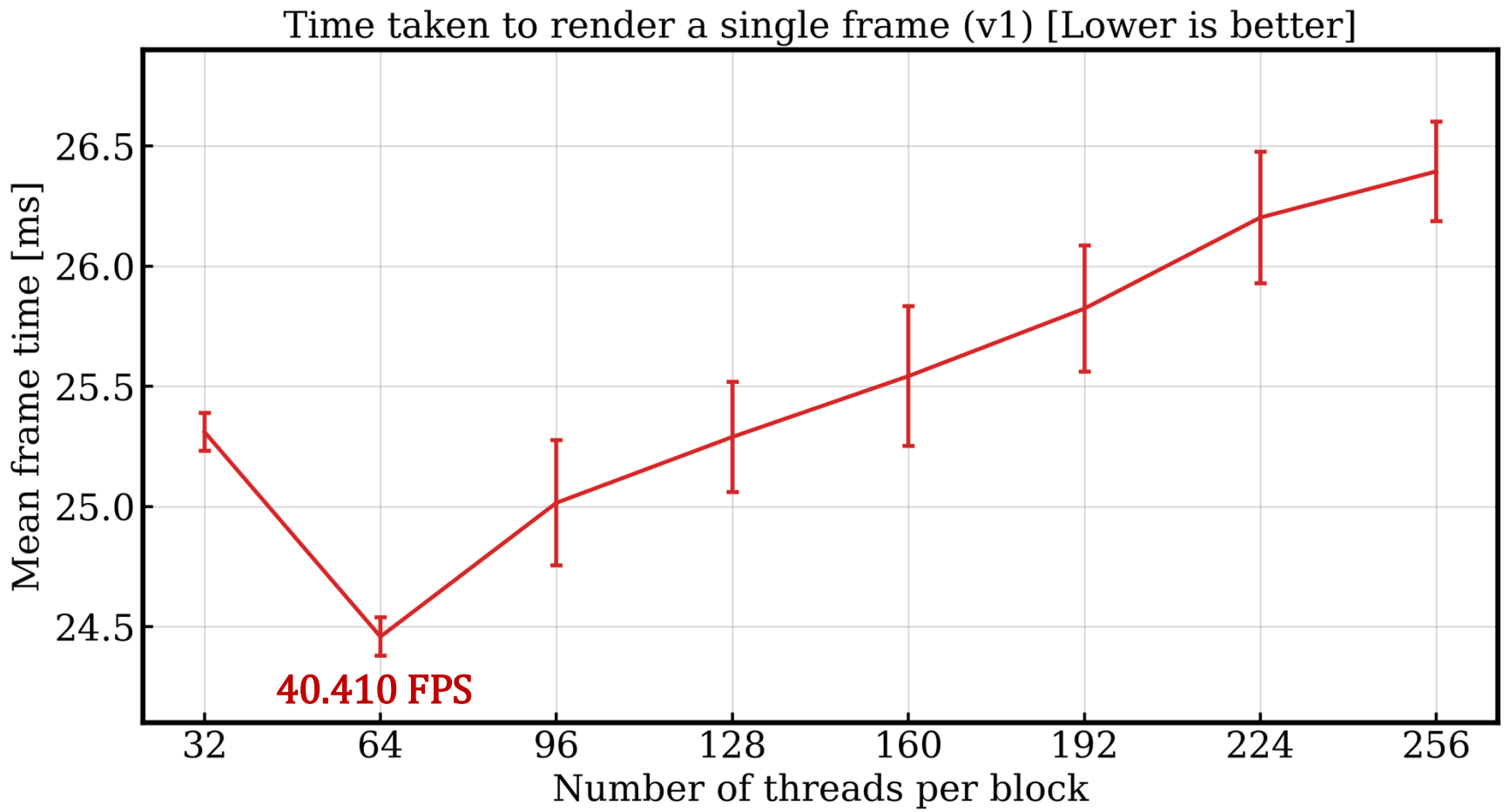
Since there are $2'073'600$ pixels (Full HD), different thread configurations are possible:

- $64'800$ blocks of 32 threads each
- $32'400$ blocks of 64 threads each
- $21'600$ blocks of 96 threads each
- …

Version 1 “CUDA” - Frame time

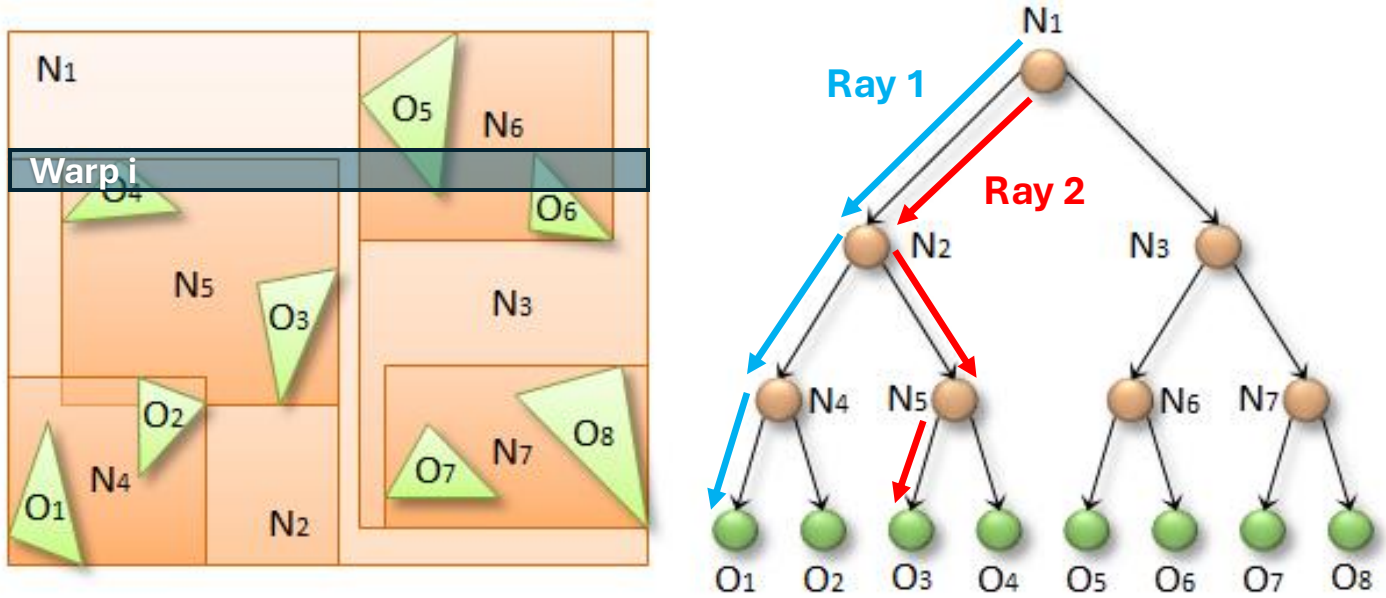


Version 1 “CUDA” - Frame time (zoomed)



Where to improve?

- Nsight Compute reports **18.73 out of 32 average active threads per warp**,
- This is because **threads in the same warp potentially** hit different triangles and, therefore, **take different paths in the BVH**

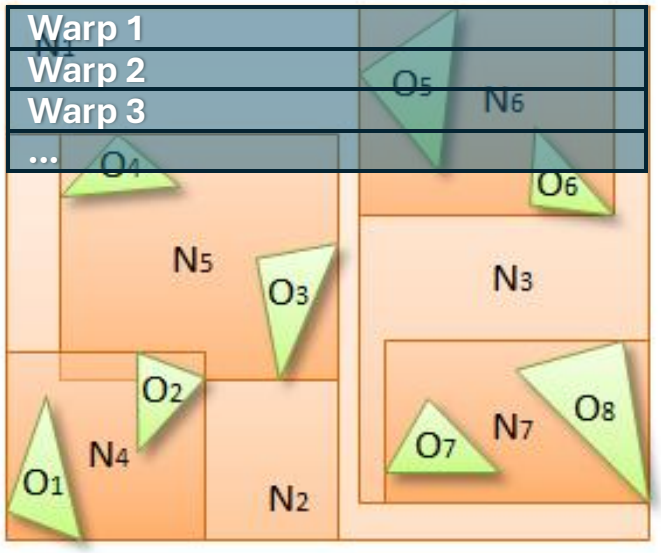


Thinking parallel, Part II | NVIDIA Technical Blog

Solution? Schedule tiles, not lines

- To minimize divergence, all the threads in a warp should render pixels close to each other

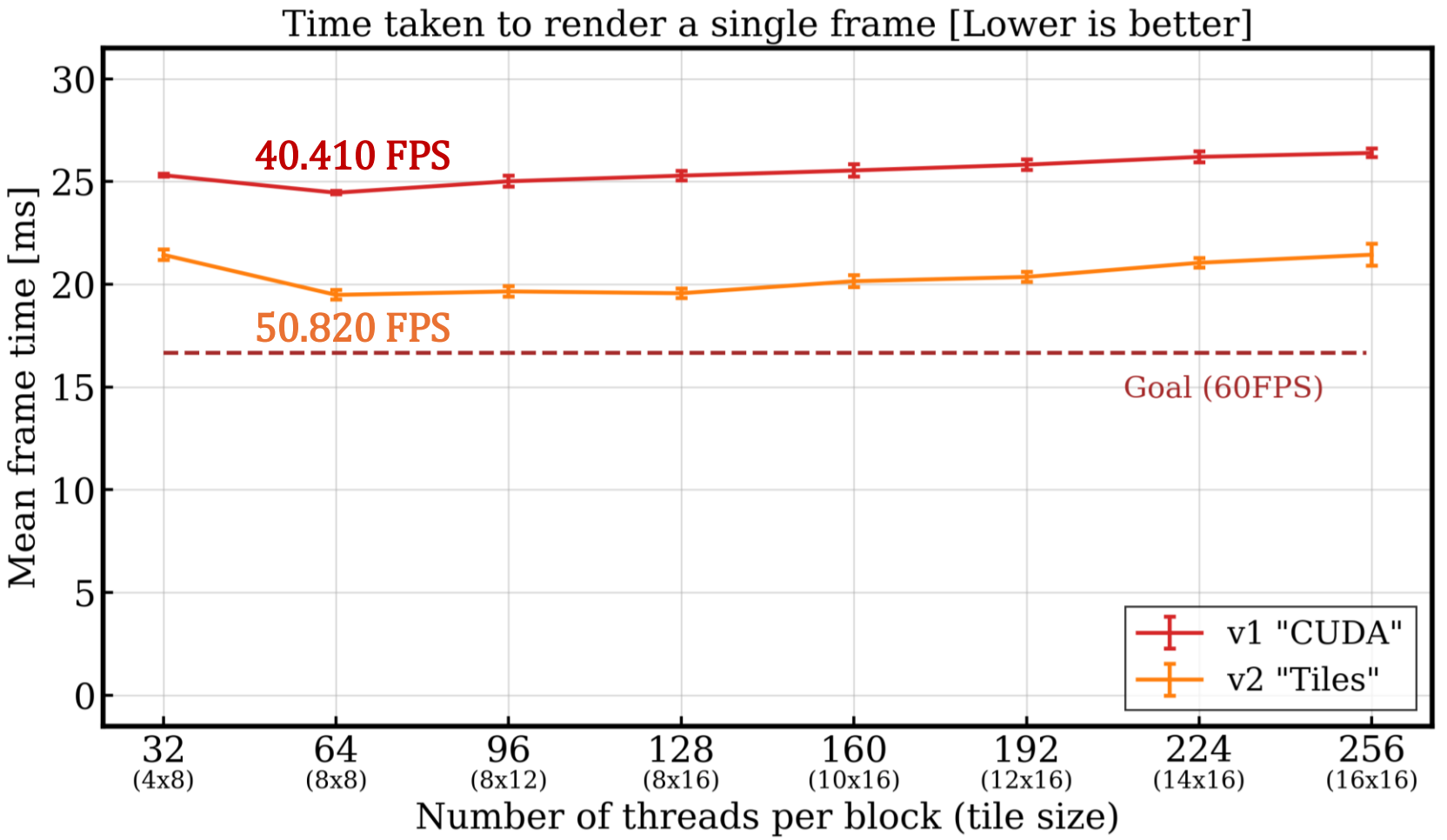
v1: **Line**-based scheduling



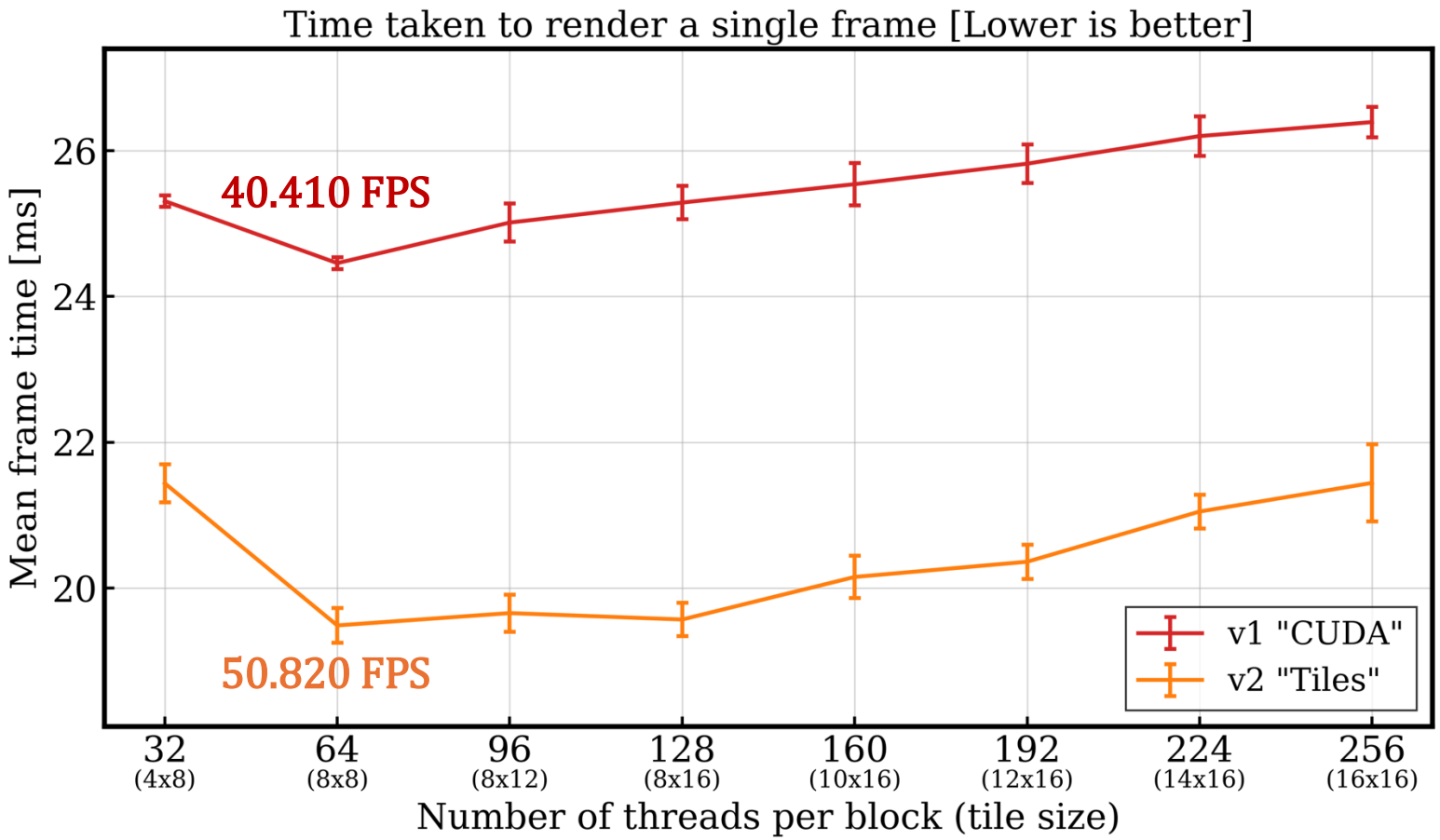
v2: **Tile**-based scheduling

Inspiration: Cinebench®

v2 “Tiles” vs v1 “CUDA” - Frame time



v2 “Tiles” vs v1 “CUDA” - Frame time (zoomed)



With 64 threads per block, the speedup of v2 against v1 is $\approx 25.7\%$

Where to improve?

Nsight Compute reports **uncoalesced global memory accesses**, i.e., threads in the same warp are accessing non-contiguous or misaligned memory addresses

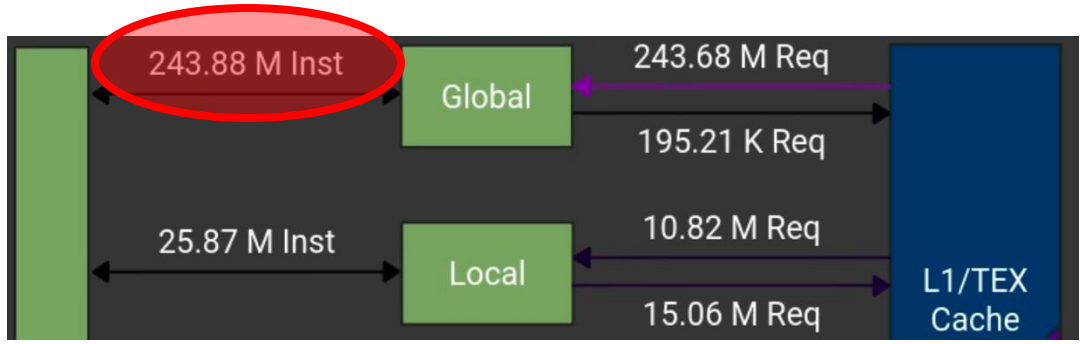
“Any access to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned”

CUDA Programming guide - Section 5.3.2

Solution? Alignment

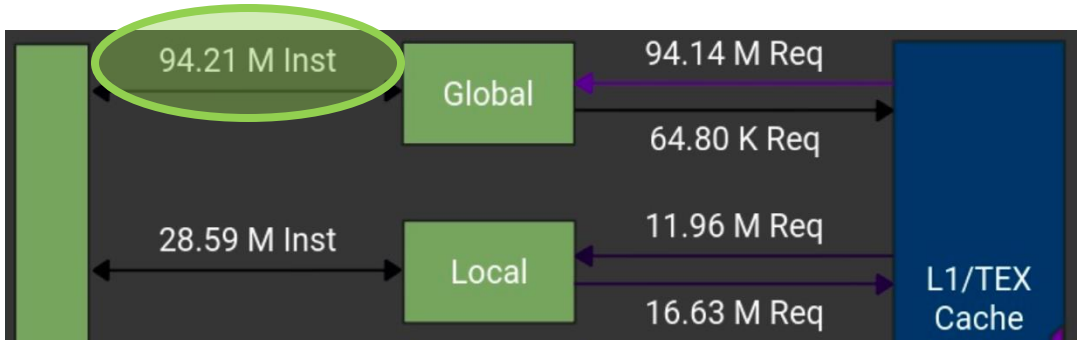
Before

```
struct __device_builtin__ float3
{
    float x, y, z;
};
```



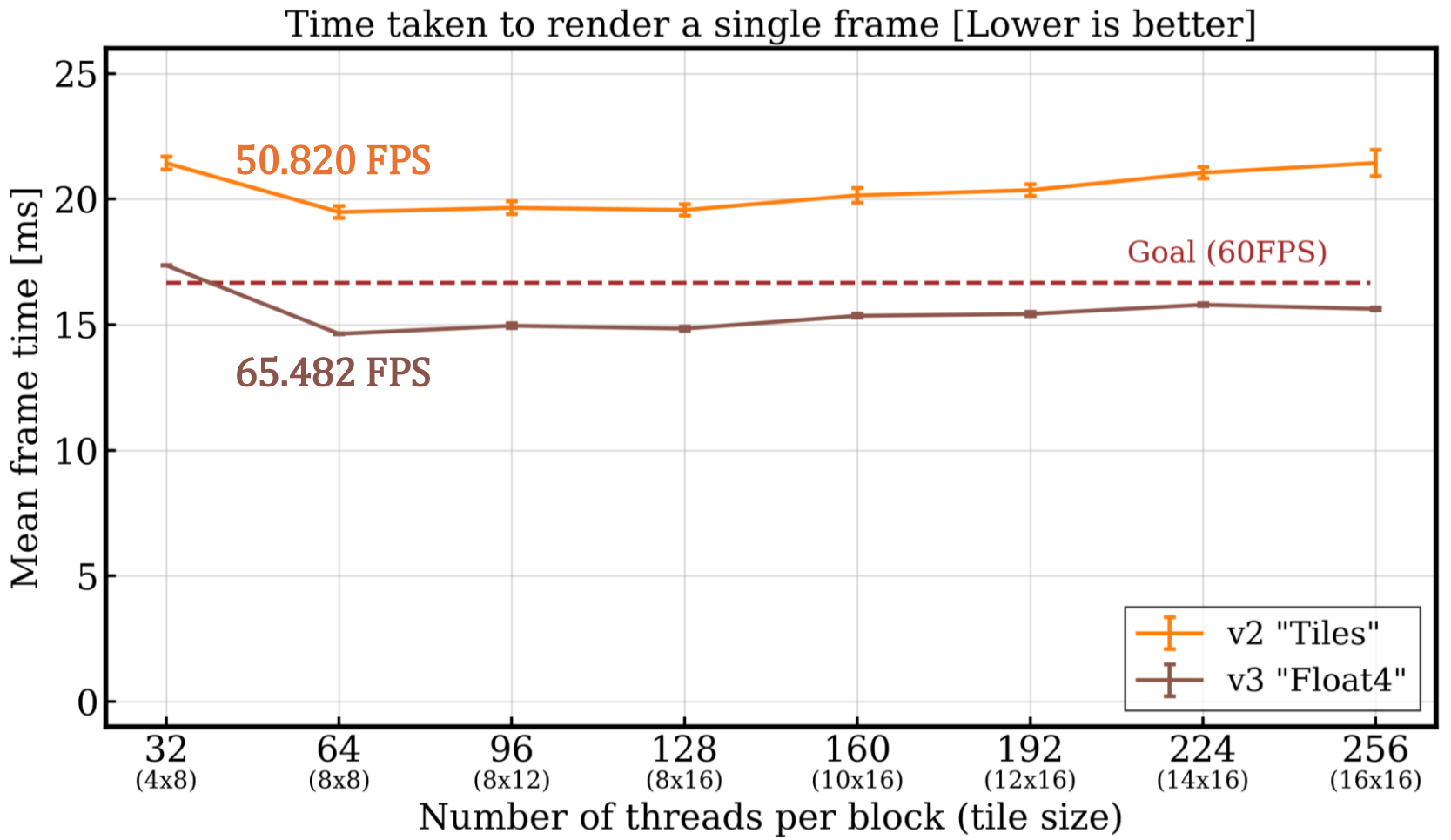
After

```
struct __device_builtin__ __builtin_align__(16) float4
{
    float x, y, z, w;
};
```

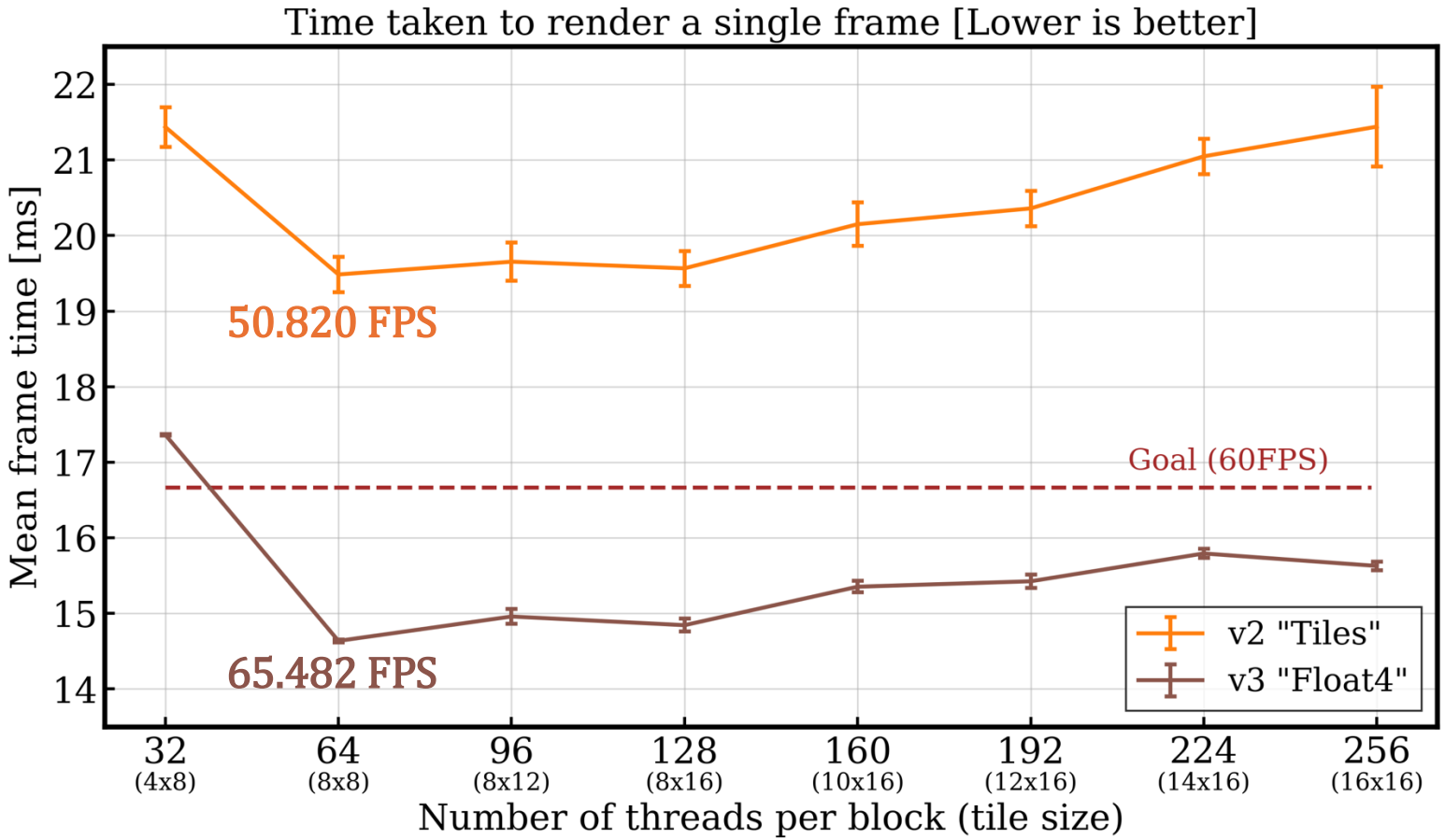


Requests to Global memory decreased by 61.37%

v3 “Float4” vs v2 “Tiles” - Frame time

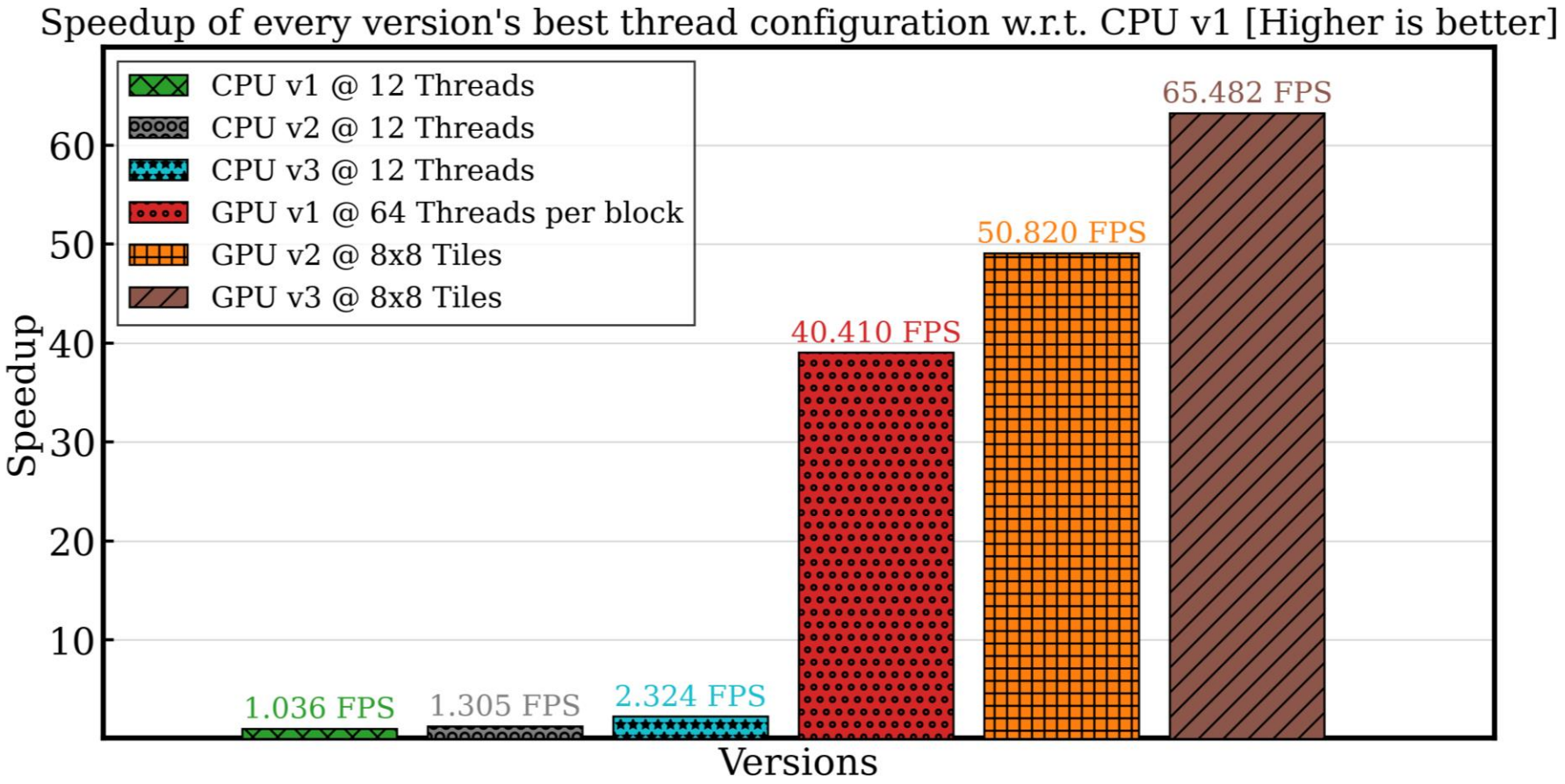


v3 “Float4” vs v2 “Tiles” - Frame time (zoomed)



With 64 threads per block, the speedup of v3 against v2 is $\approx 28.8\%$

Final results



GPU's v3 is ≈ 28 times faster than CPU's v3

Appendix - Failed FP16 Optimization attempt

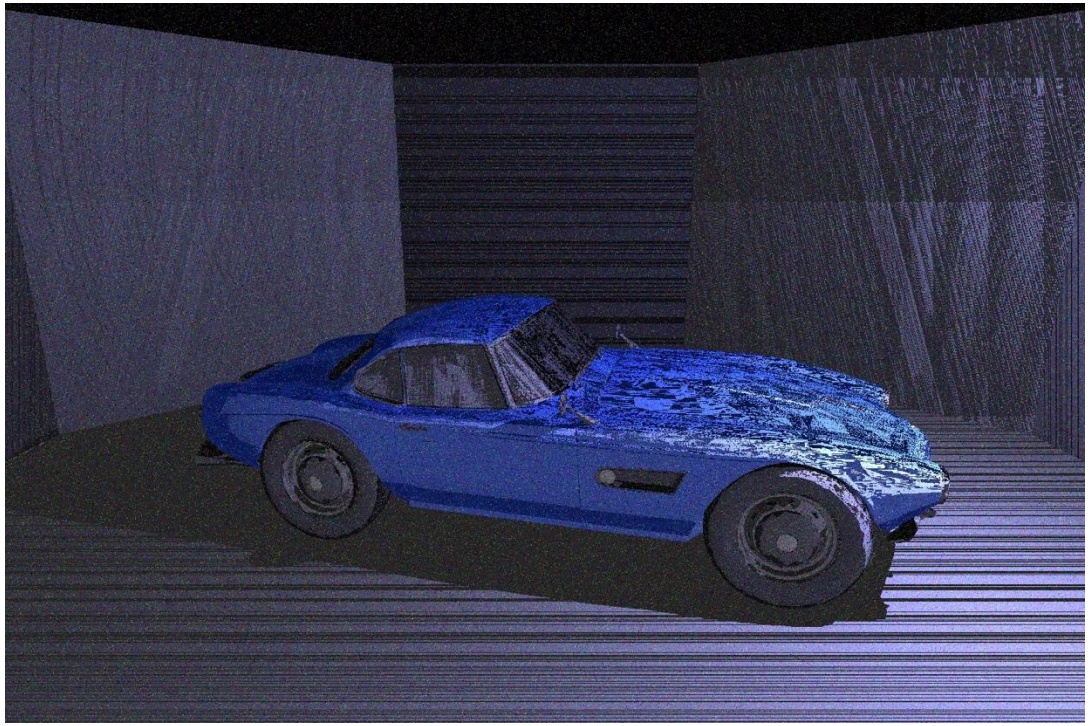
- FP16 arithmetic offers a **theoretical 200% speedup** over FP32 arithmetic. In addition, it also benefits both storage and memory bandwidth

Theoretical Performance	
Pixel Rate:	105.6 GPixel/s
Texture Rate:	224.4 GTexel/s
FP16 (half):	14.36 TFLOPS (2:1)
FP32 (float):	7.181 TFLOPS
FP64 (double):	224.4 GFLOPS (1:32)

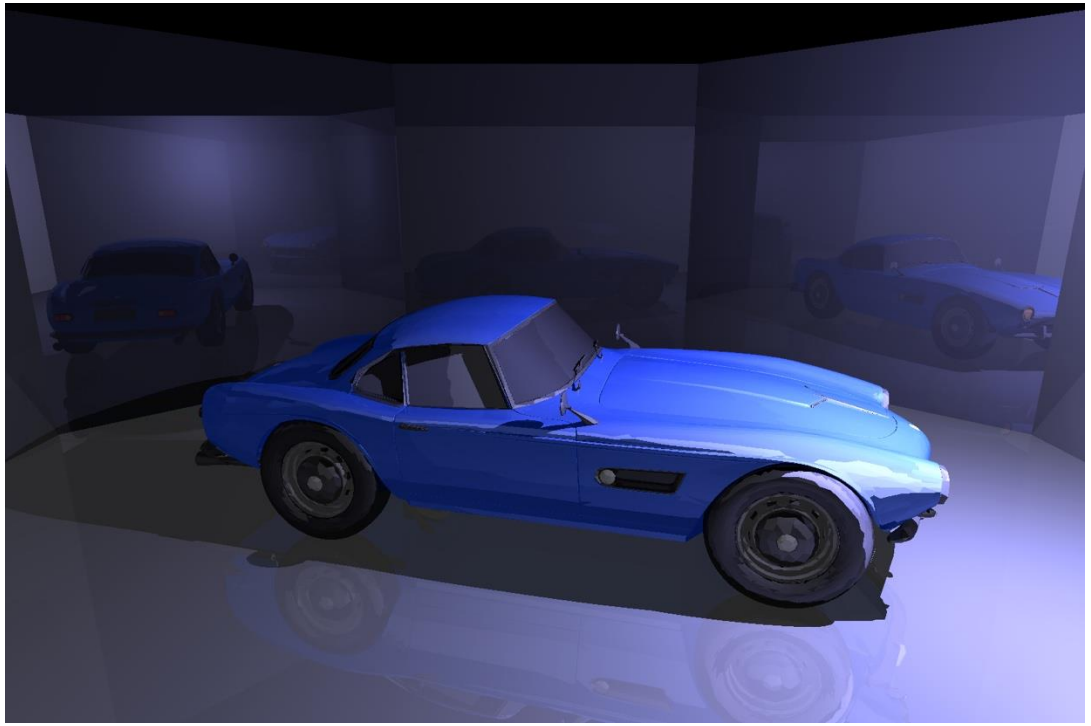
RTX 2060 Super - TechPowerUp

Appendix - Failed FP16 Optimization attempt

FP16



FP32



Unfortunately, in this case, FP16 math introduces too much noise