Modern GPU Architecture

Graffathon, 12 June 2016

Konsta Hölttä (sooda)
System Software Engineer

Introduction

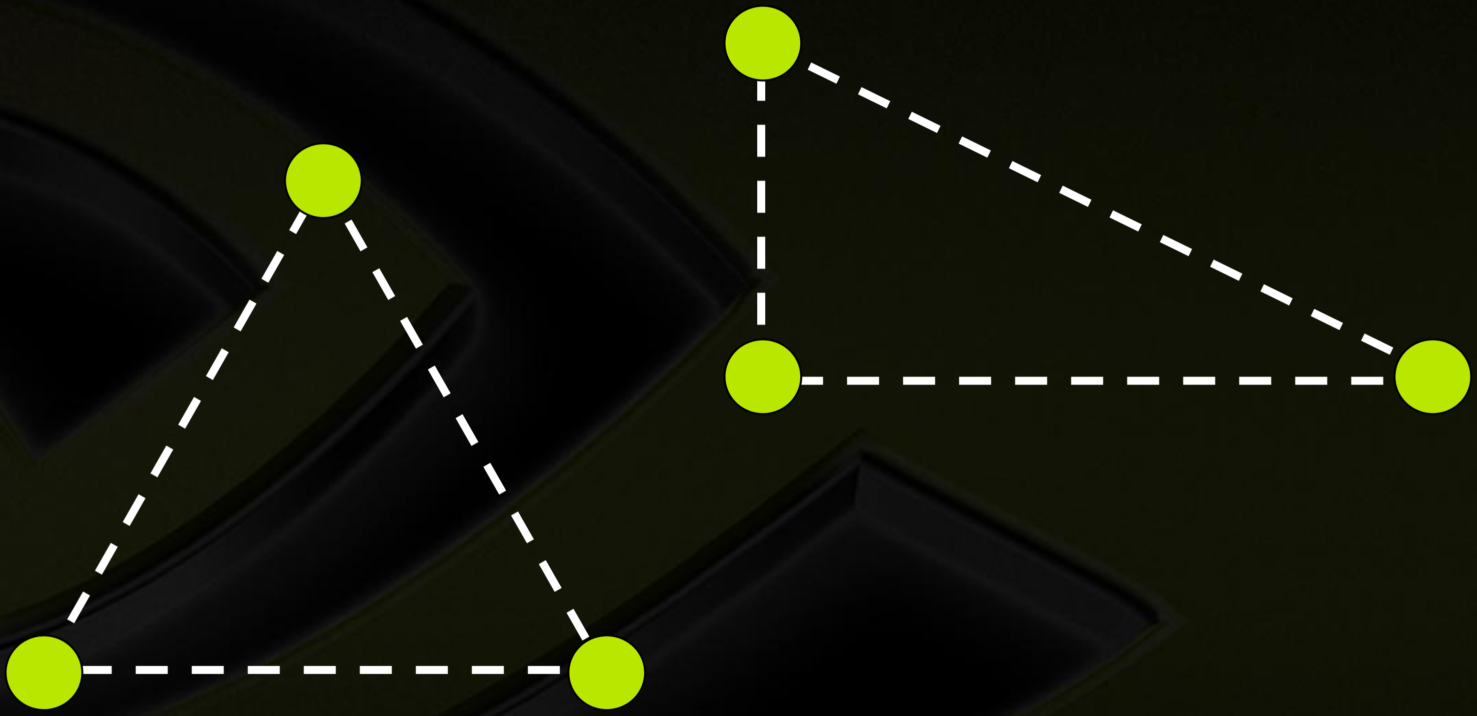
- Why so fast, how to use
- NVIDIA approach
- High-level basics with some details
- … and I’m just a kernel hacker working with Tegras, with a hobby in computer graphics
- https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

Recap: triangle pipeline

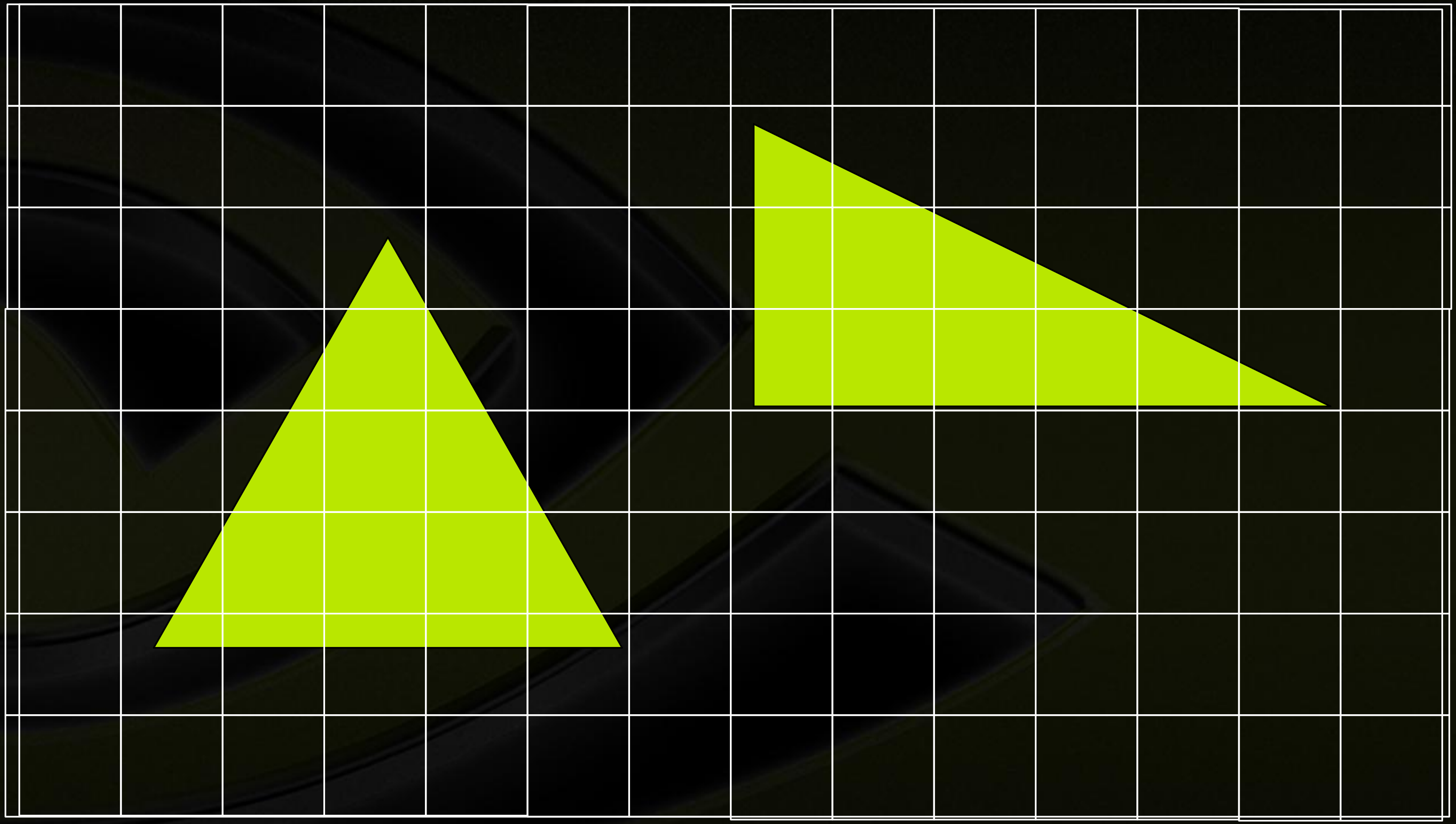
```
// (glDrawArrays / glDrawElements / ...)
draw_triangles({
    x00, y00, z00,
    x01, y01, z01,
    x02, y02, z02,

    x10, y10, z10,
    x11, y11, z11,
    x12, y12, z12});
```

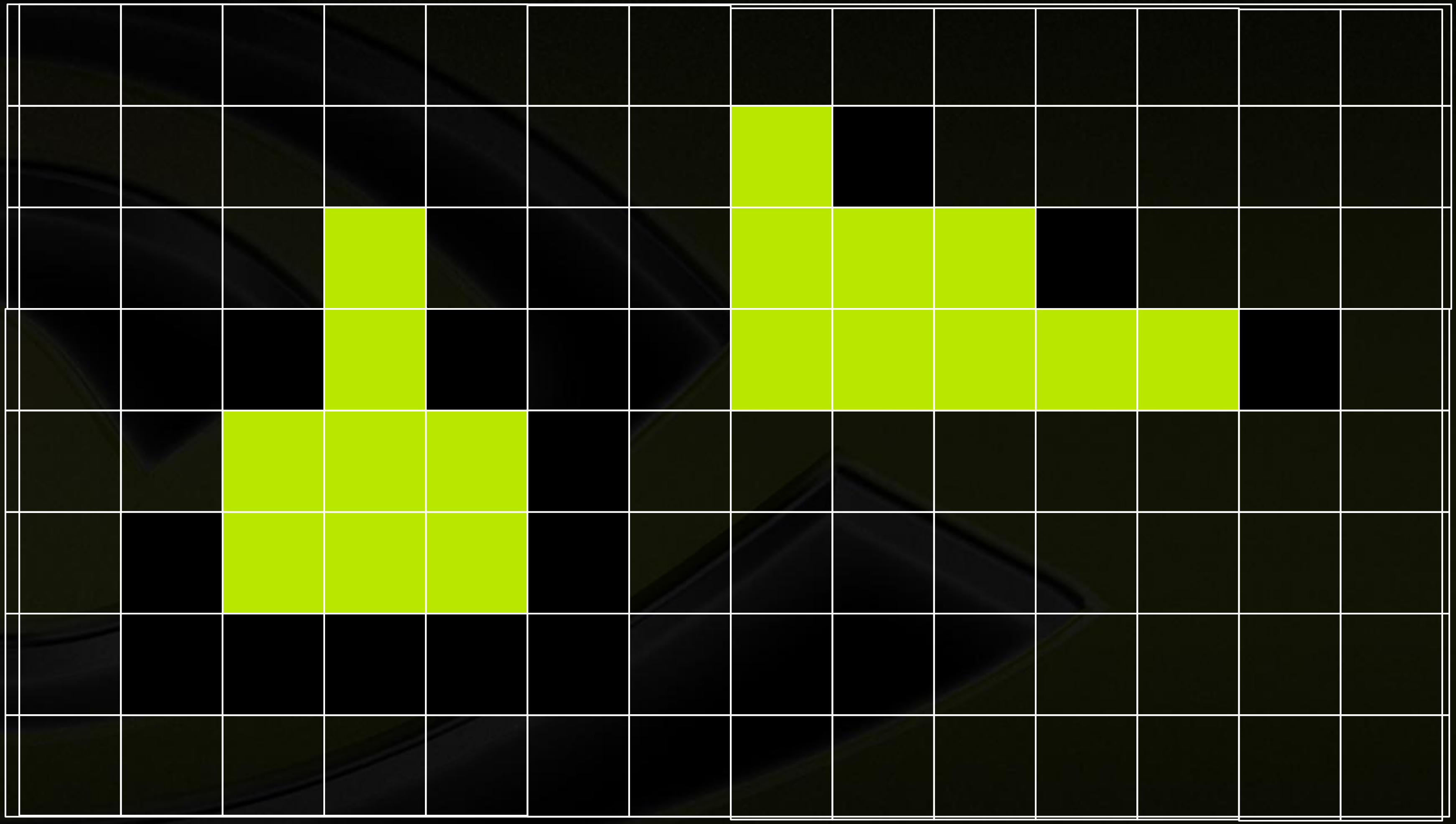
Recap: triangle pipeline



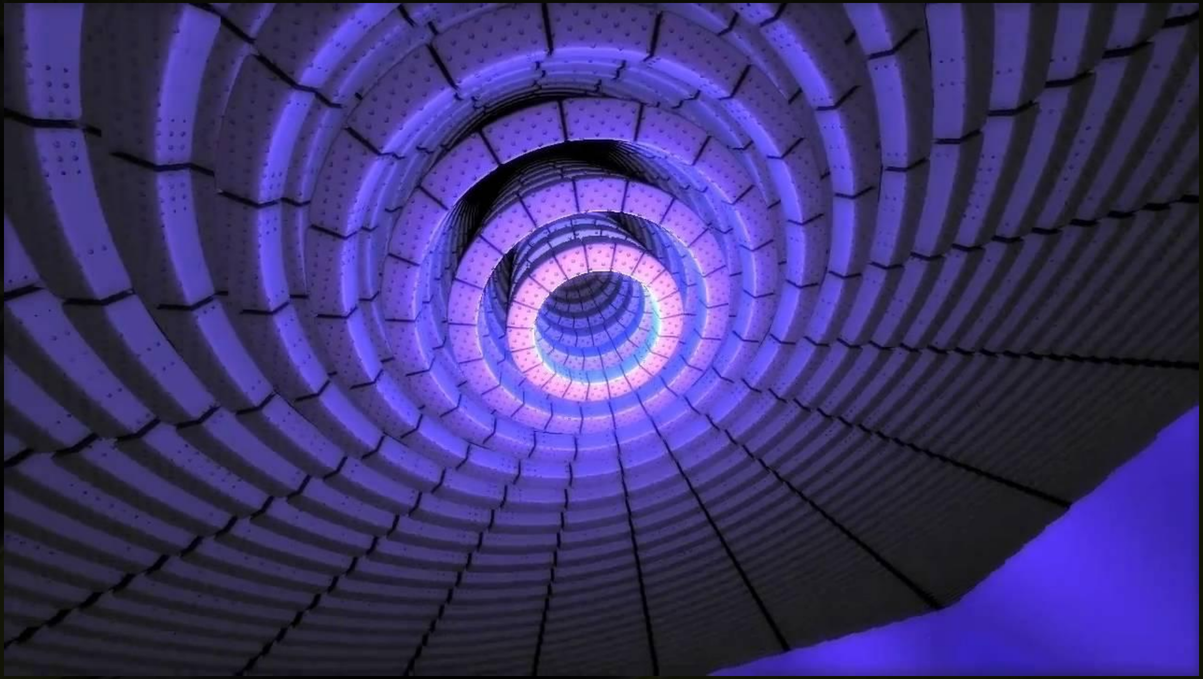
Recap: triangle pipeline



Recap: triangle pipeline



Two approaches to shading



Brief NV history

- Aug 1999: GeForce 256 (DX7, GL 1)
- …
- Nov 2006: Tesla (8800 GTX, DX10, GL 3)
- Mar 2010: Fermi (GTX 480, DX11, GL 4)
- Mar 2012: Kepler (GTX 680, DX12, GL 4.5)
- Sep 2014: Maxwell (GTX 980)
- May 2016: Pascal (GTX 1080) (GP100 Apr 2016)

GTX 1080

- 8 GB DDR5X main memory
  - 320 GB/s, 256b wide
- 1733 MHz
- 2560 CUDA Cores
- 9 TFLOPS
- 180 W

vs Intel Core i7-6700K: 0.1-0.2 TFLOPS, 91 W

- Completely different designs, purposes

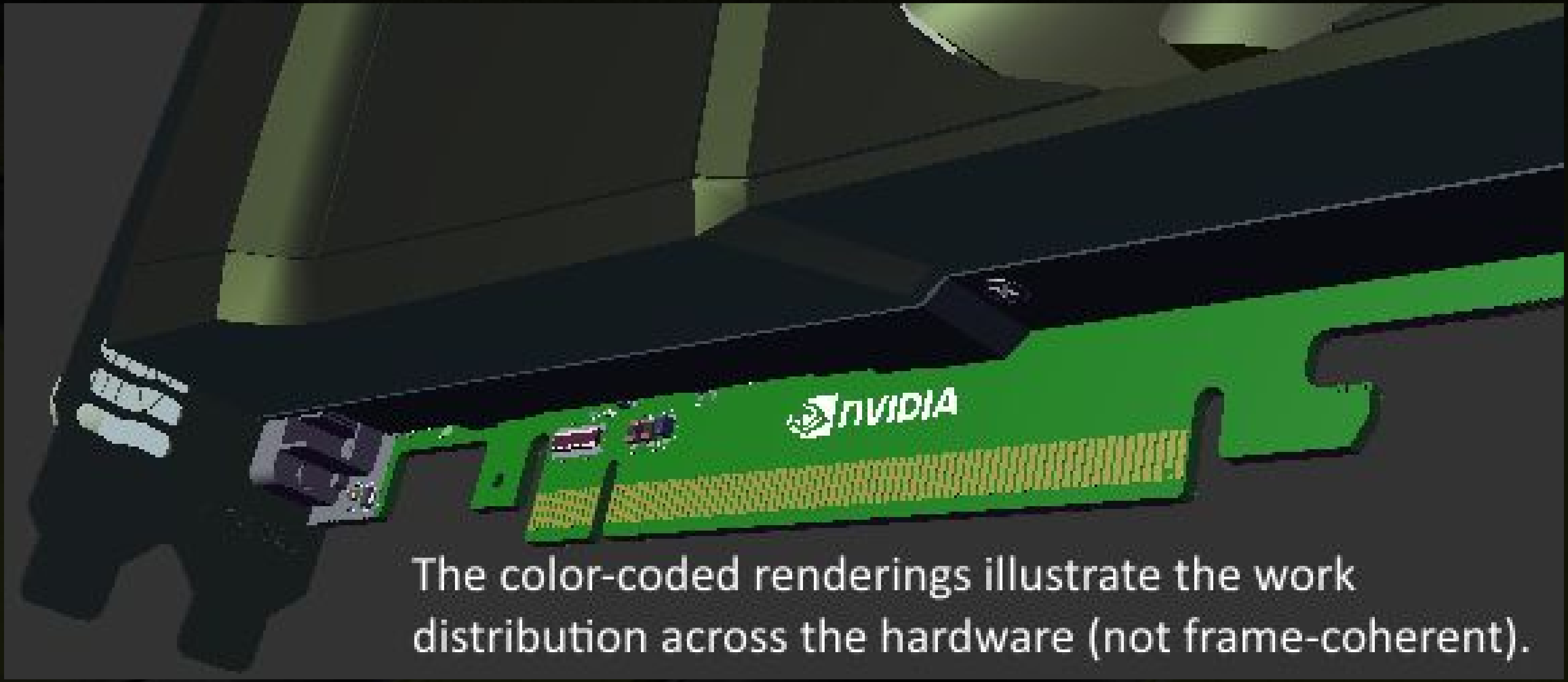
Super parallel work distributor

- GPUs are not magically faster CPUs
- 2560 super dumb cores
  - Compare to modern CPUs good at decision-making with branch prediction, instruction reordering and stuff
- Hiding latency with >9000 threads
  - Preferably something like 9 000 000
  - Don’t do that on a CPU!
- Can’t run 2560 operating systems
- CPUs optimized for low latency, GPUs for high throughput

Thread/Warp Scheduling

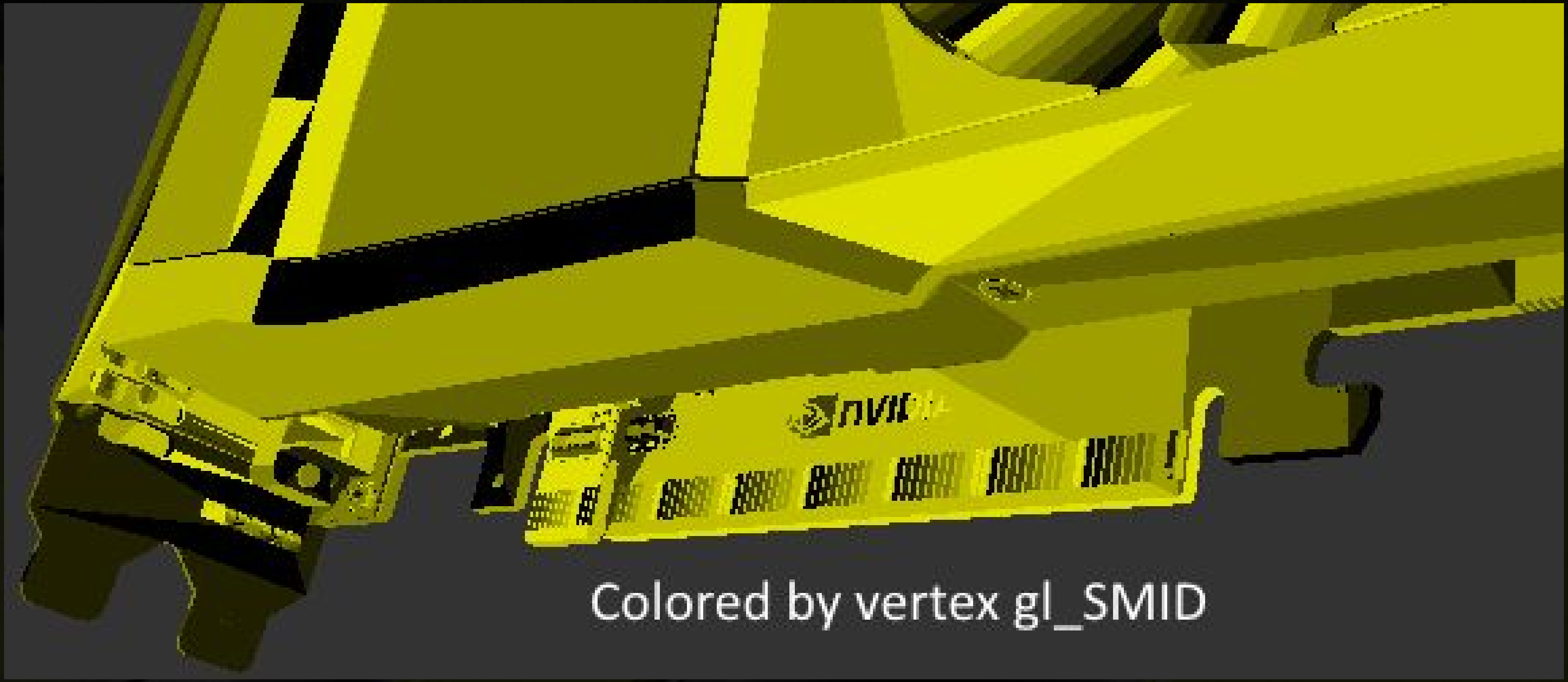
- NVIDIA warp = 32 threads
- Single Instruction Multiple Threads
  - i.e., every 32 threads run in lockstep
- N warps in SM, N SMs in GPC, N GPCs in GPU
- Fast HW warp schedulers
- Threads just masked out in branches
  - Avoid strongly divergent code

Scheduling 1/4



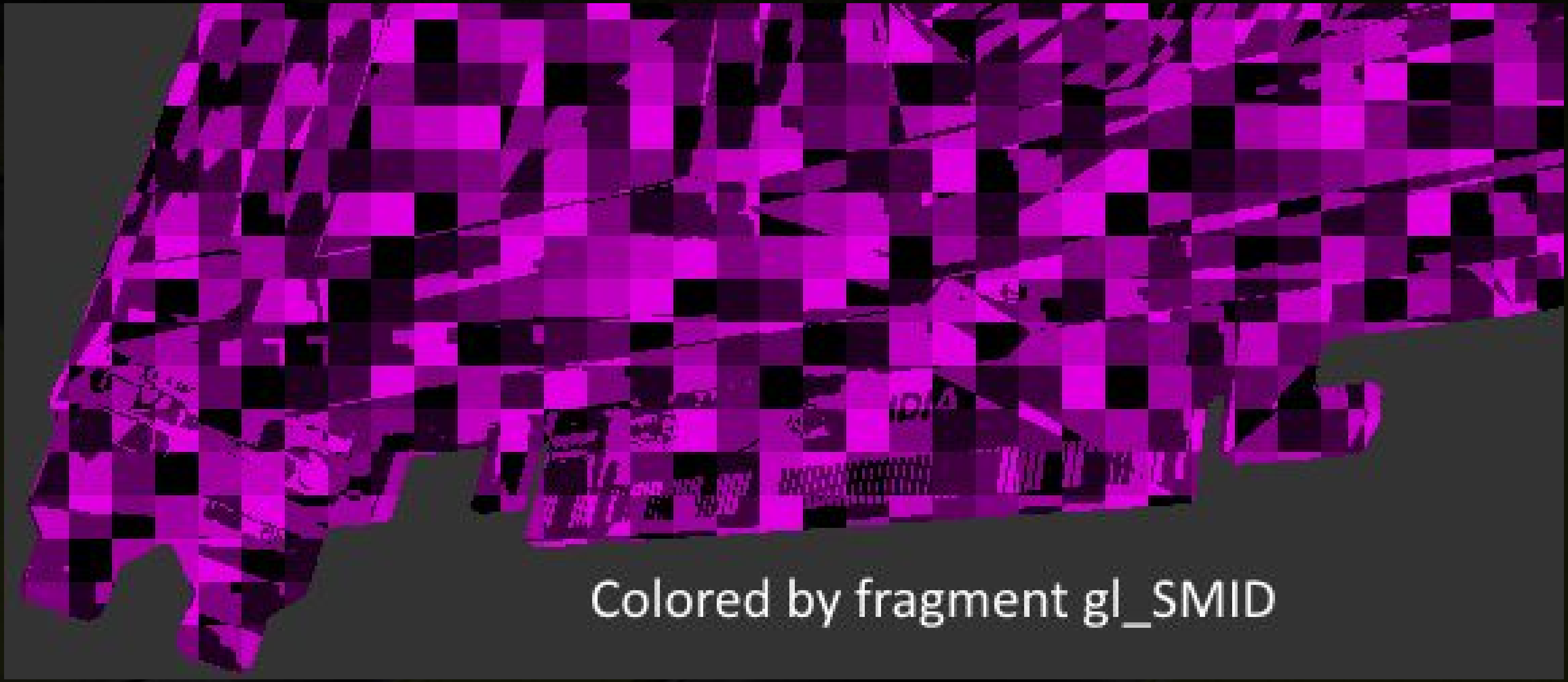
The color-coded renderings illustrate the work distribution across the hardware (not frame-coherent).

Scheduling 2/4

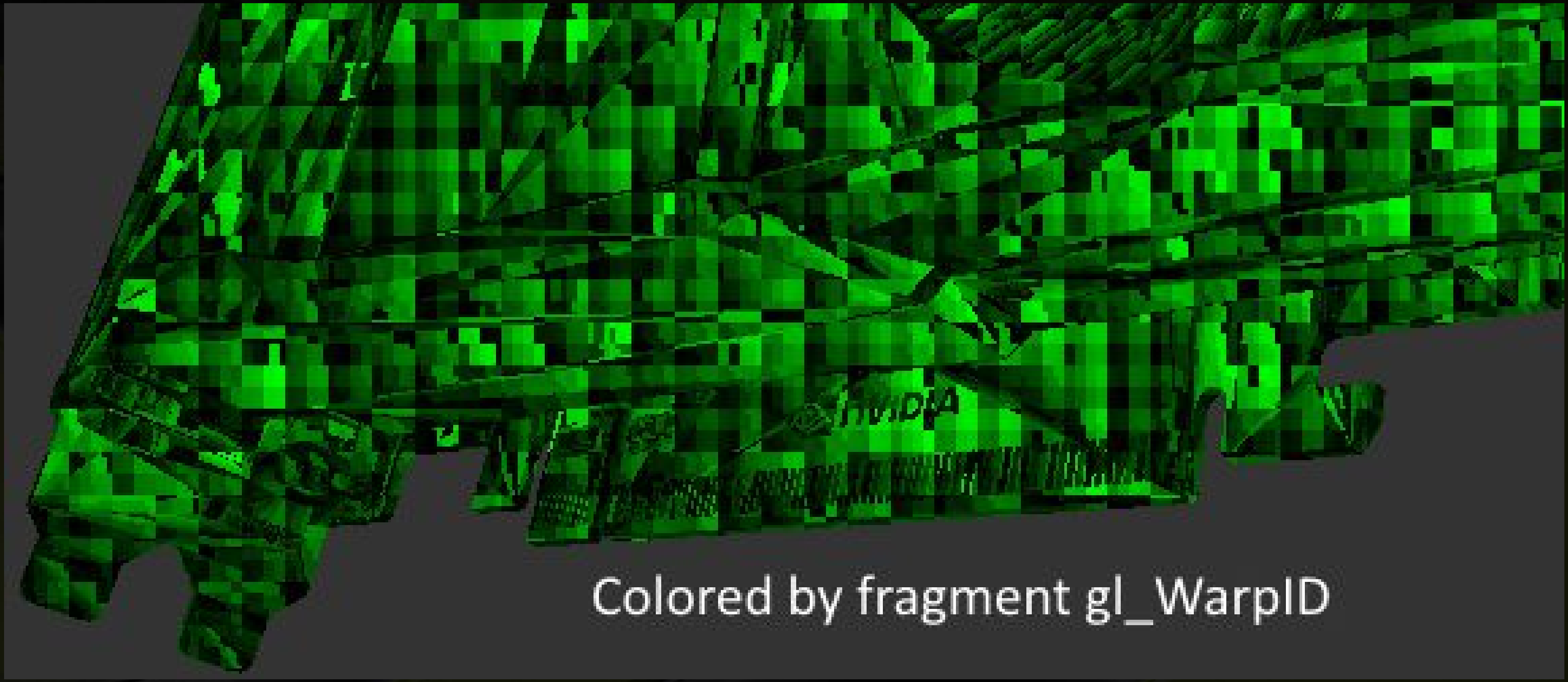


Colored by vertex gl_SMID

Scheduling 3/4


Colored by fragment gl_SMID

Scheduling 4/4



Bad frag shader

```
float r, g;
if (mod(int(x) + int(y), 2) != 0)
    r = shiny_checkerboard(x, y); // big func
else
    g = shiny_checkerboard(x, y); // ran twice
```

- All threads “run” both, masked out ones are discarded

(Bad example?)

```
float r, g, tmp;
tmp = shiny_checkerboard(x, y);
if (mod(int(x) + int(y), 2) != 0)
    r = tmp;
else
    g = tmp;
```

Good frag shader

```
vec2 rg = shiny_checkerboard(
               mod(int(x) + int(y), 2), xy);
...
vec2 shiny_checkerboard(int flag, vec2 xy) {
    return colors(xy) * vec2(flag, 1 - flag); }
```

- Prefer vectors
- Prefer arithmetic over branches
- “Constant” global branches are OK
  - Think a demo timeline

Sum array items (CPU vs CUDA-ish)

```
for (i = 0; i < N; i++)
    sum += arr[i]; // depends on previous

for (t = 0; t < N/M; t++) { // parallel
    for (i = t * M; i < (t + 1) * M; i++)
        sums[t] += arr[i];
}
for (j = 0; j < N/M; j++) sum += sums[t];
```

Also, graphics

- The cores work with any GPGPU problem
  - CUDA, {comp,geom,vert,frag} shaders
  - “Registers” part of local shared memory
- Vert/frag scheduling pipe, clipping, rasterizer, zcull,

  AA, several HW optimizations
- Precalc in vert shader, interpolate/prefetch for frag!

Memory is slow

- Problem even with N GB/s
  - 1920*1080*60*4 = ~500 MB/s for just one buf's pixels
  - plus all intermediate postprocessing stages
- Reading vertices, textures, etc
- Latency is your enemy (100s of cycles)
- RAM? "Random" Access Memory, LOL

Prefer sequential access

- Memory accessed in batches, by warp
- Fetched in cache lines
- Also, try to keep the cache hot (>10x faster)
- Same with CPU
- AoS vs SoA

Draw call

- Internal housekeeping in the driver
- Often expensive state change in HW
- Can run in parallel
- Vulkan

From API to HW

- GFX API → Pushbuffer → Host interface → Frontend → Primitive distributor → Cores
- I’m working with the host btw (in Tegra)
- Single command buffer in flight per engine
  - Or, how the shadertoy frontpage is so heavy
  - Roughly a single GL/DX context at a time (I guess)

Summary

- Think parallel & independent & concurrent
- There’s more: tessellation, display heads, color, FPS, ctxsw, power mgmt, SLI, …
- Demo coding is more art than science
  - Also, no need to always push HW limits for art
- Shader optimization is more magic than science (unless you really think about it and know the per-chip hardware details)