

SLICING THE WORKLOAD

MULTI-GPU OPENGL RENDERING APPROACHES

INGO ESSER - NVIDIA DEVTECH PROVIZ

OVERVIEW

- ▶ **Motivation**
- ▶ Tools of the trade
 - ▶ Multi-GPU driver functions
 - ▶ Multi-GPU programming functions
- ▶ Multi threaded multi GPU renderer
 - ▶ General workflow
 - ▶ Different applications

MOTIVATION

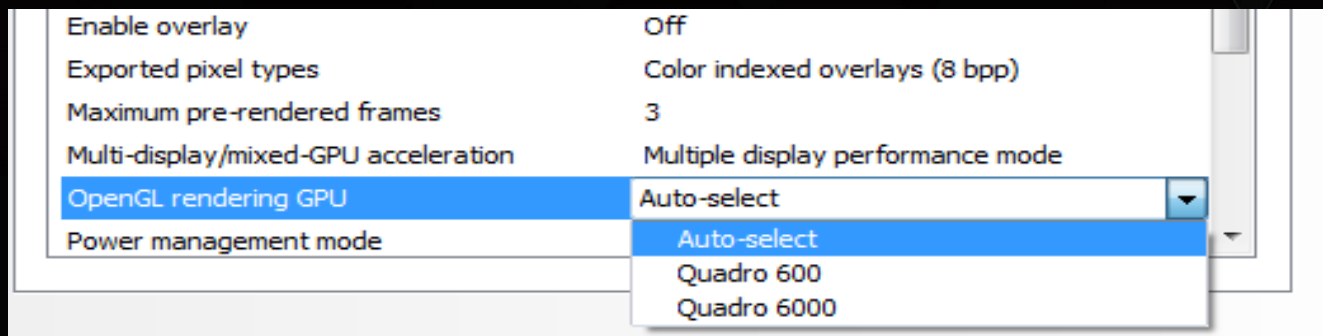
- ▶ Apps are becoming less CPU-bound, more GPU-bound
 - ▶ S5135 - GPU-Driven Large Scene Rendering in OpenGL
 - ▶ S5148 - Nvpro-Pipeline: A Research Rendering Pipeline
- ▶ Fragment Load (complex fragment shaders, higher resolutions)
 - ▶ Slice image space
- ▶ Data / Geometry Load (large datasets)
 - ▶ Slice data / geometry
- ▶ Processing (complex compute jobs)
 - ▶ Offload complex calculations to other GPUs
- ▶ Stereo Rendering / VR is a natural fit

OVERVIEW

- ▶ Motivation
- ▶ Tools of the trade
 - ▶ Multi-GPU driver functions
 - ▶ Multi-GPU programming functions
- ▶ Multi threaded multi GPU renderer
 - ▶ General workflow
 - ▶ Different applications

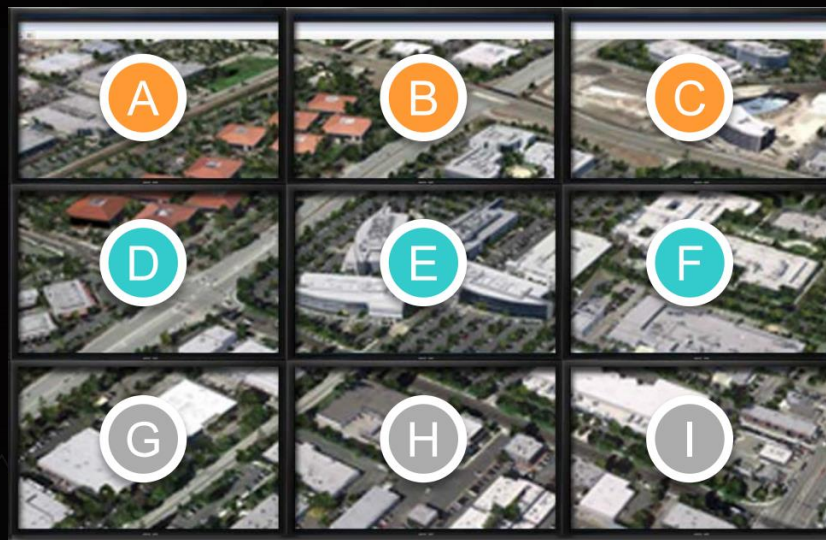
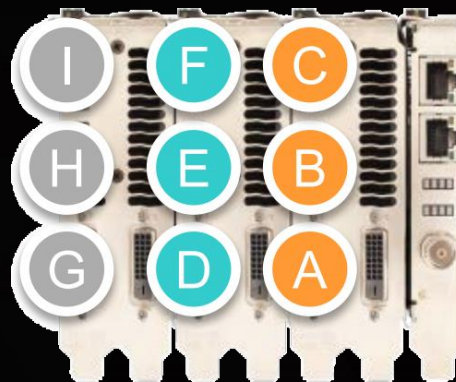
DIRECTED GPU RENDERING

- ▶ Quadro only
- ▶ Allows picking rendering GPU
- ▶ Fast blit path to display GPU
- ▶ Dedicate GPUs
 - ▶ OpenGL
 - ▶ Compute
- ▶ Choose via
 - ▶ NVIDIA Control Panel
 - ▶ NVAPI: developer.nvidia.com/nvapi



QUADRO MOSAIC

- ▶ Via SLI bridge or Quadro Sync board
- ▶ Advantages:
 - ▶ Transparent behavior
 - ▶ One unified desktop
 - ▶ No tearing
 - ▶ Fragment clipping possible
- ▶ Disadvantages:
 - ▶ Single view frustum
 - ▶ Whole scene rendered



QUADRO SLI FSAA

- ▶ Use two Quadro boards with SLI connector
- ▶ Transparently scale image quality
- ▶ Up to 128x FSAA

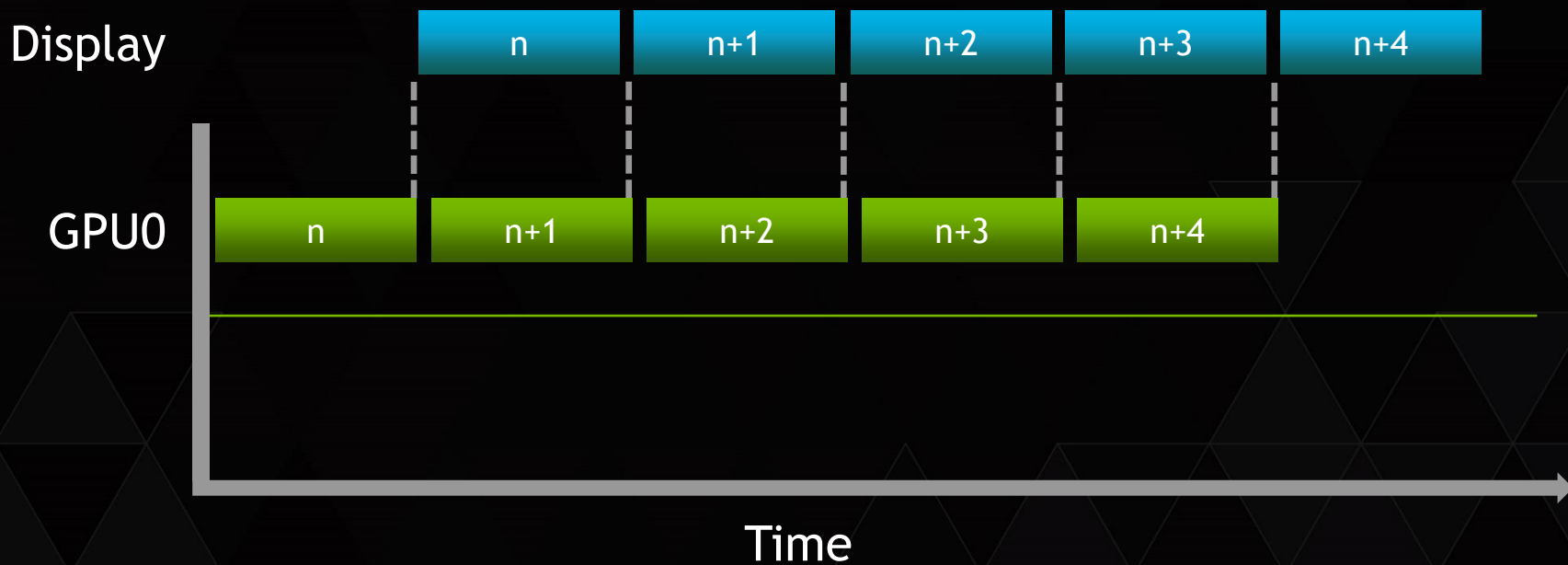


QUADRO SLI AFR

- ▶ Semi-automagic multi-GPU support for alternate frame rendering (AFR)
- ▶ SLI AFR abstracts GPUs away
 - ▶ Application sees one GPU
- ▶ Driver mirrors static resources between GPUs
 - ▶ No transfer between GPUs for unchanged data
 - ▶ E.g. static textures, geometry data
 - ▶ Dynamic data might need to be transferred

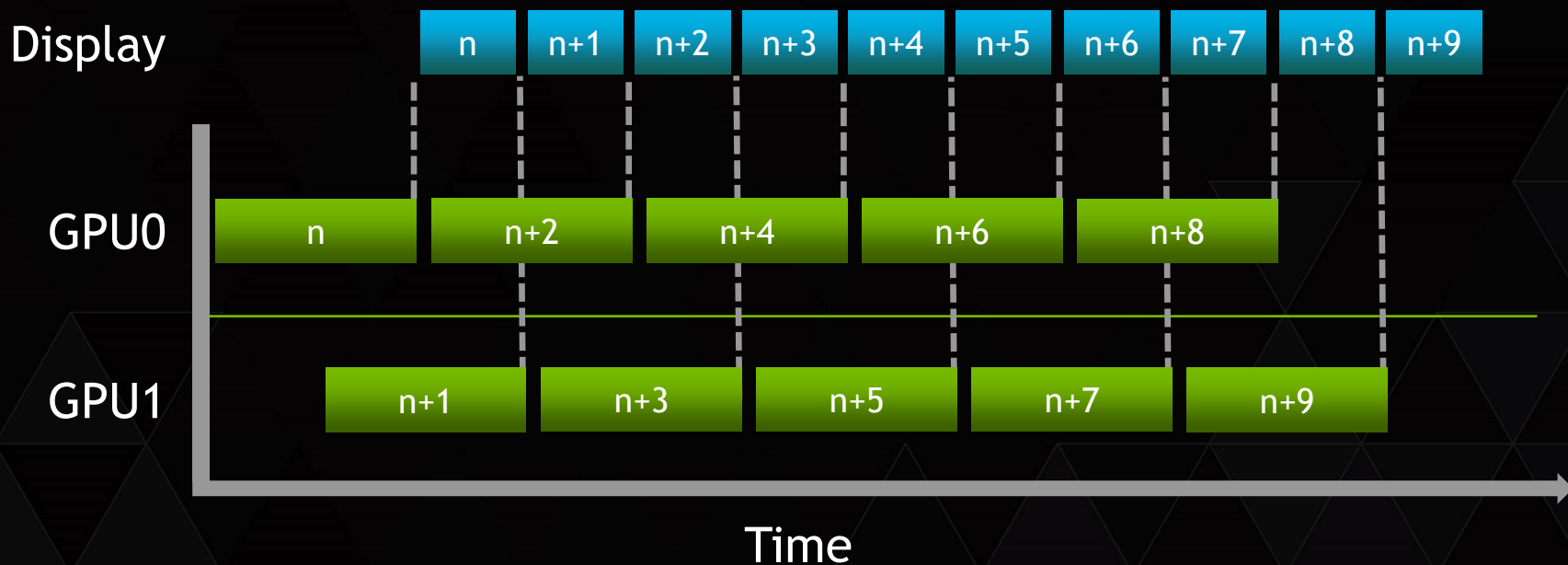
QUADRO SLI AFR

- Single GPU frame rendering



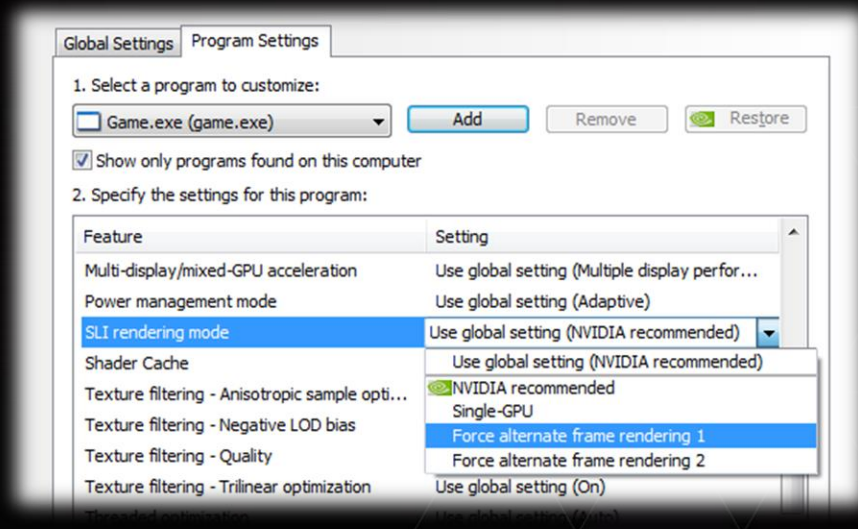
QUADRO SLI AFR

- ▶ SLI AFR rendering on two GPUs
- ▶ Same frame time, same latency
- ▶ Frames rendered in parallel, twice the frame rate



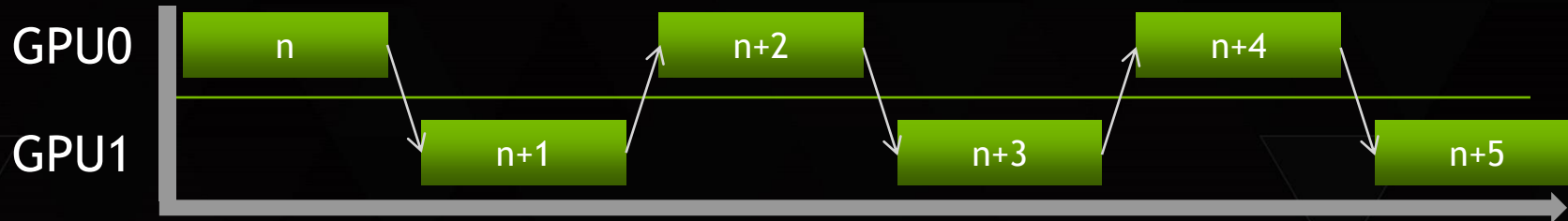
QUADRO SLI AFR

- ▶ Switch on SLI
 - ▶ Application needs a profile
 - ▶ Force AFR1 / AFR2 in NV control panel
 - ▶ For testing: Use profile “SLI Aware Application”



QUADRO SLI AFR

- ▶ Prerequisites for AFR (driver is conservative)
 - ▶ Unbind dynamic resources before calling swap
 - ▶ GPU Queue must be full - no flushing GL queries
 - ▶ Clear full surface

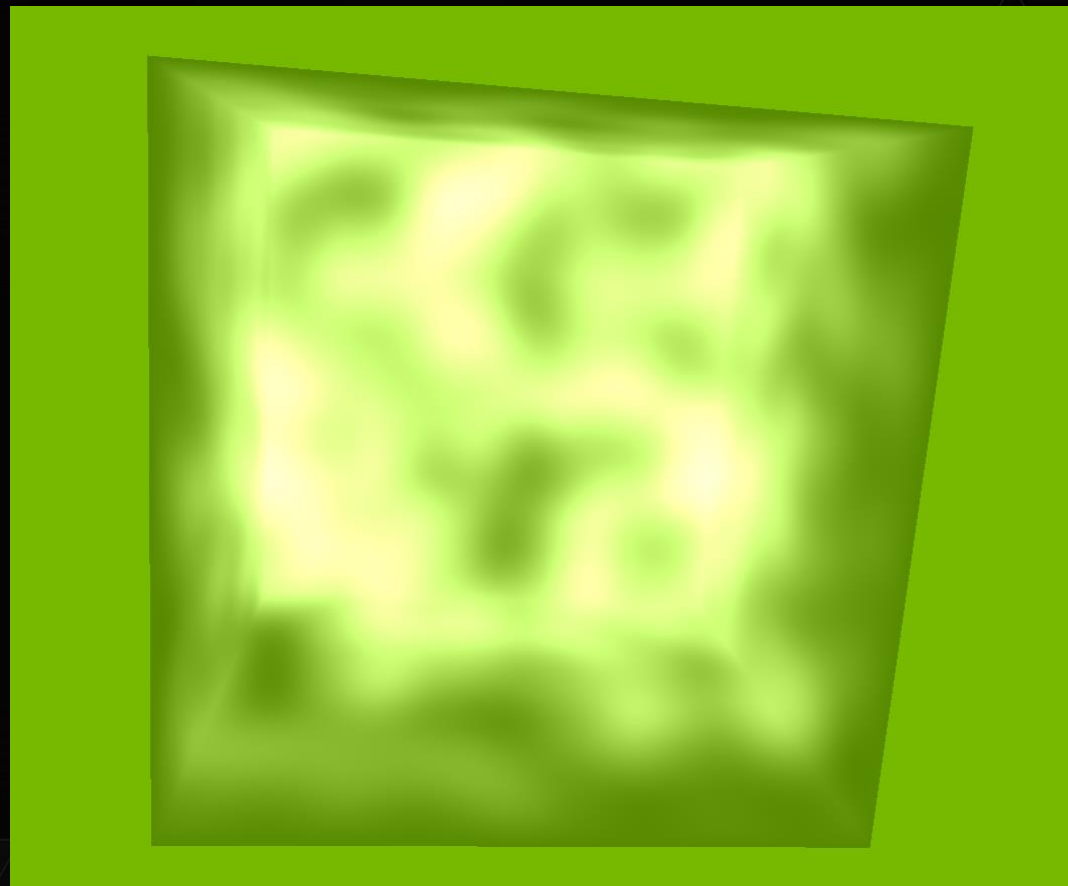


- ▶ If SLI AFR doesn't scale: Use GL debug callback
 - ▶ `glEnable(GL_DEBUG_OUTPUT);`
 - ▶ `glDebugMessageCallback(...);`
 - ▶ Working on improving debug messages, feedback from developers welcome!

OVERVIEW

- ▶ Motivation
- ▶ Tools of the trade
 - ▶ Multi-GPU driver functions
 - ▶ Multi-GPU programming functions
- ▶ Multi threaded multi GPU renderer
 - ▶ General workflow
 - ▶ Different applications

MULTI-GPU RENDERING



DISTRIBUTING WORKLOAD

- ▶ Use NV_gpu_affinity extension
- ▶ Enumerate GPUs
 - ▶ `wglEnumGpusNV(UINT iGPUIndex, HGPUNV* phGPU)`
- ▶ Enumerate displays per GPU
 - ▶ Needed to determine final display for image present
 - ▶ `wglEnumGpuDevicesNV(HGPUNV hGPU, UINT iDeviceIndex, PGPU_DEVICE lpGpuDevice);`
- ▶ Create an OpenGL context for a specific GPU

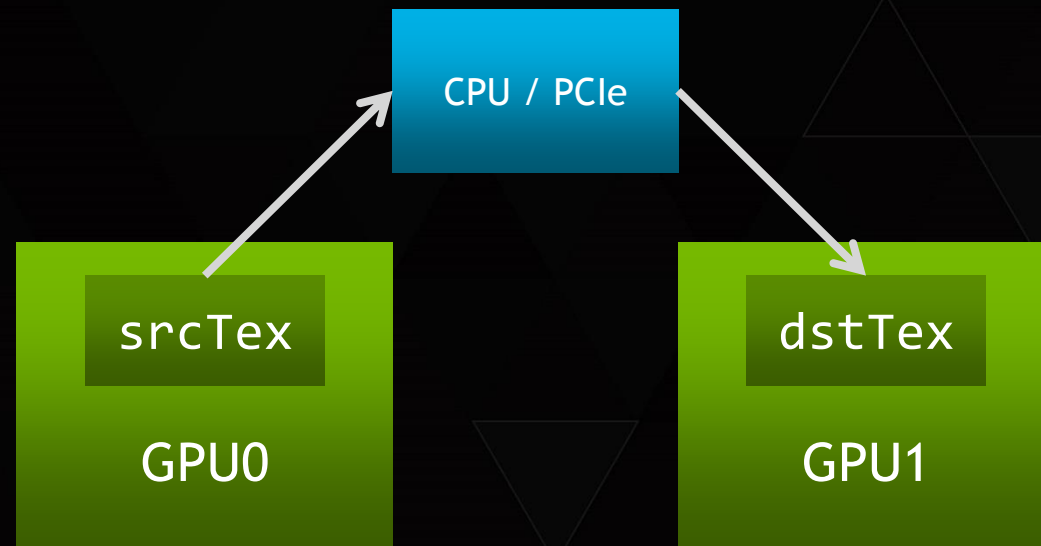
```
HGPUNV GpuMask[2]= {phGPU, nullptr};
//Get affinity DC based on GPU
HDC affinityDC = wglCreateAffinityDCNV( GpuMask );
SetPixelFormat( affinityDC, ... );
HGLRC affinityGLRC = wglCreateContext( affinityDC );
```


SHARING DATA BETWEEN GPUS

- ▶ For multiple contexts on same GPU
 - ▶ ShareLists & GL_ARB_Create_Context
- ▶ For multiple contexts across multiple GPUs
 - ▶ Readback (GPU₁-Host) → Copy on host → Upload (Host-GPU₀)
- ▶ NV_copy_image extension for OGL 3.x
 - ▶ Windows - `wglCopyImageSubDataNV`
 - ▶ Linux - `glXCopyImageSubdataNV`
 - ▶ Avoids extra copies, same pinned host memory is accessed by both GPUs

NV_COPY_IMAGE EXTENSION

- ▶ Transfer in single call
 - ▶ No binding of objects
 - ▶ No state changes
 - ▶ Supports 2D, 3D textures & cube maps
- ▶ Async for Fermi & above



```
wglCopyImageSubDataNV( srcCtx, srcTex, GL_TEXTURE_2D, 0, 0, 0, 0,  
                        tgtCtx, tgtTex, GL_TEXTURE_2D, 0, 0, 0, 0,  
                        width, height, 1 );
```

OPENGL SYNCHRONIZATION

- ▶ OpenGL commands are asynchronous
 - ▶ `glDraw*(...)` can return before rendering has finished
- ▶ Use Sync object (GL 3.2+) for apps that need to sync on GPU completion
 - ▶ Much more flexible than using `glFinish()`
- ▶ Fence is inserted in consumer GL stream; blocks execution until producer signals fence object

GPU0

`glDraw`

`wglCopy...`

`glFenceSync`

GPU1

`glWaitSync`

`glBind`

`glDraw`



OVERVIEW

- ▶ Motivation
- ▶ Tools of the trade
 - ▶ Multi-GPU driver functions
 - ▶ Multi-GPU programming functions
- ▶ Multi threaded multi GPU renderer
 - ▶ General workflow
 - ▶ Different applications

SETTING THE STAGE

- ▶ App with rendering function

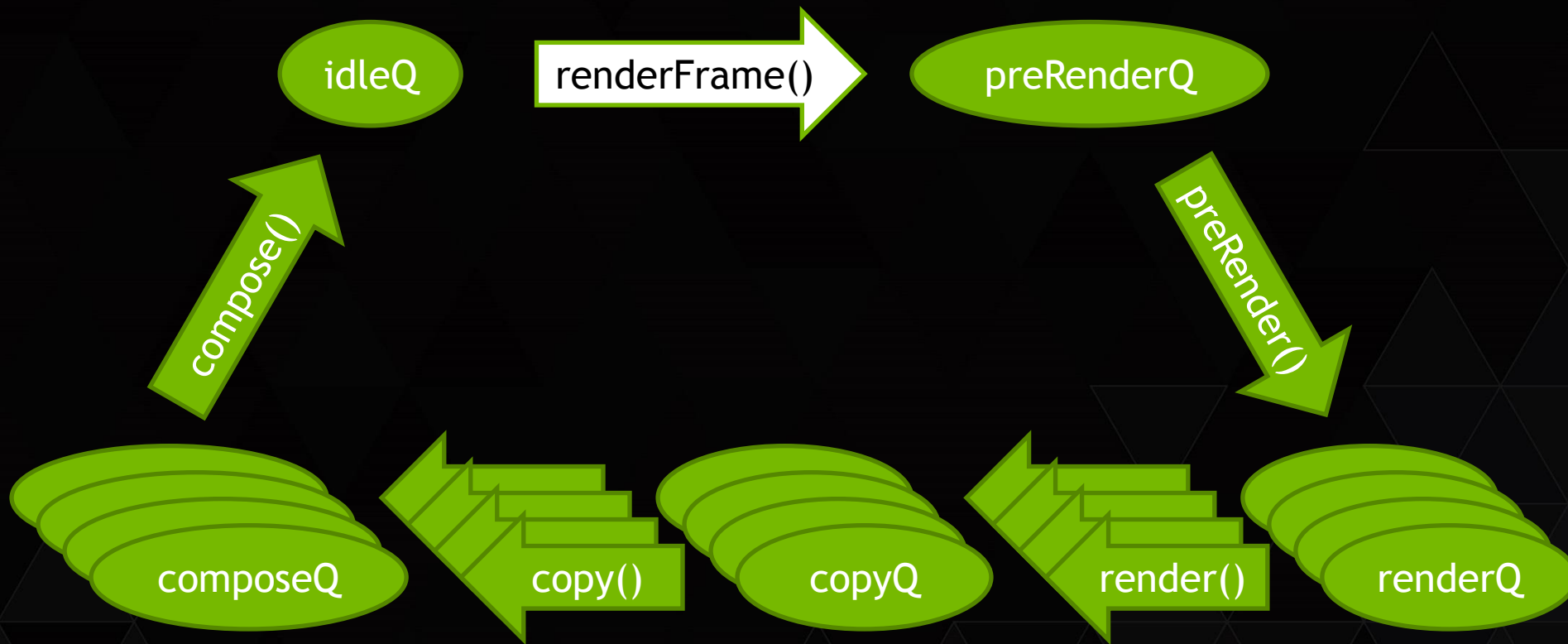


renderFrame()

- ▶ Fragment bound
- ▶ Improvements
 - ▶ Split image to distribute rendering load (sort-first)
 - ▶ Use multiple GPUs (4 in the example)
 - ▶ Do parallel rendering
 - ▶ Hide transfer overhead

RENDER PIPELINE

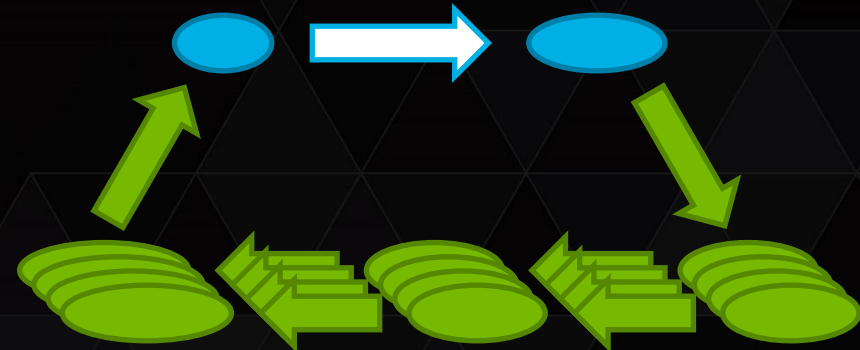
GTC 2014 - ID S4455



APP::RENDERFRAME CALL

- ▶ Take an event token from the idle queue
- ▶ Add data for this frame (e.g. frame number, view matrix)
- ▶ Put token into the first queue of pipeline

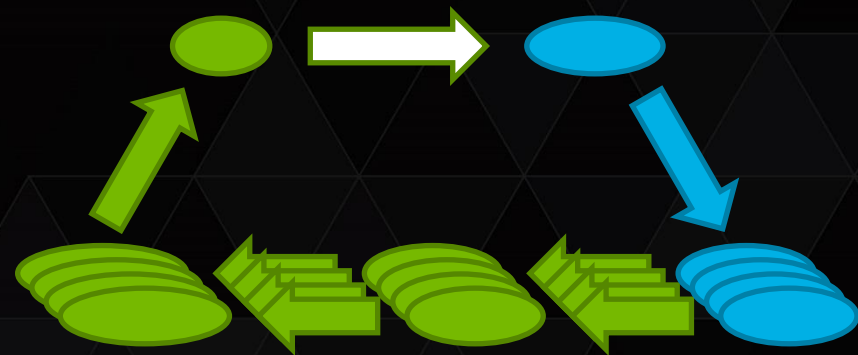
```
auto event = m_idleQueue->pop();  
event->setType( Event::RENDER );  
/* update payload */  
m_preRenderQueue->push(event);
```



PRERENDER STEP

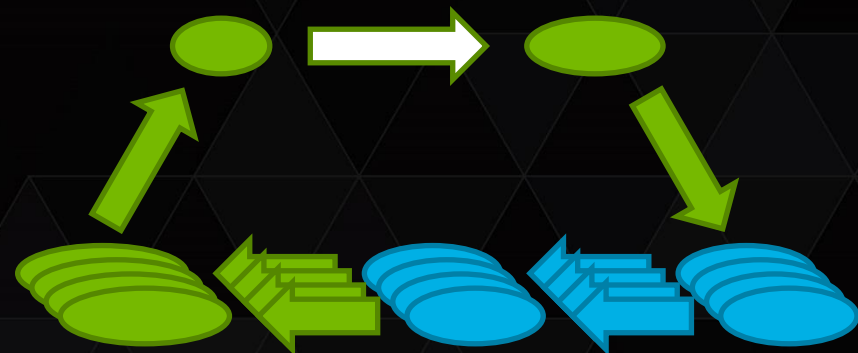
- ▶ Optional pre-computation (e.g. load balancing information)
- ▶ Put event token into N render queues
- ▶ Parallel execution begins here

```
auto event = inputQueue->pop();  
/* pre-computation code */  
for( auto& i : outputQueues ) {  
    i->push( event );  
}
```



RENDER STEP

- ▶ N affinity contexts, optimally rendering 1/Nth of GPU load
- ▶ “Manually” multiplex scene resources to all threads
- ▶ E.g. scissor / depth / stencil buffer to confine rendering area
- ▶ Use texture from the event token as render target
- ▶ Insert fence at the end to signal render step has finished



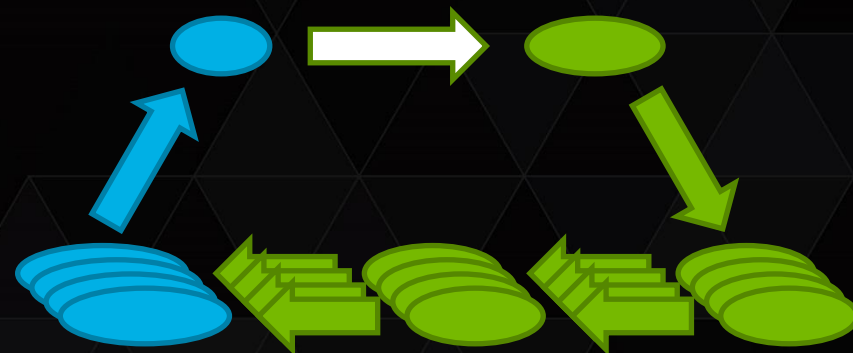
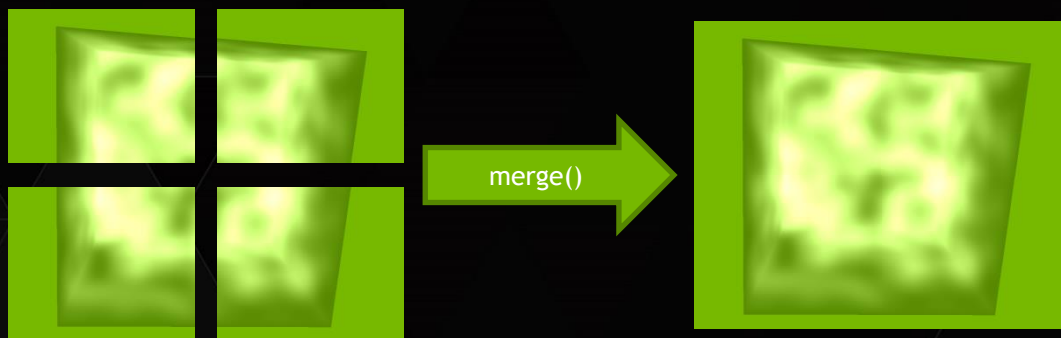
COPY STEP

- ▶ N copy threads copying N textures
- ▶ Wait for fence from preceding render thread
- ▶ Copy data from render GPU to display GPU
- ▶ Use textures from event token as source & target
- ▶ Insert fence at the end to signal copy has finished



COMPOSE STEP

- ▶ Pop from N event queues (CPU synchronization)
- ▶ Perform N `glWaitSync` (GPU synchronization)
- ▶ Take N textures and combine image data to output image
- ▶ Optional post-processing (overlays etc.)
- ▶ Call `SwapBuffers` to present frame

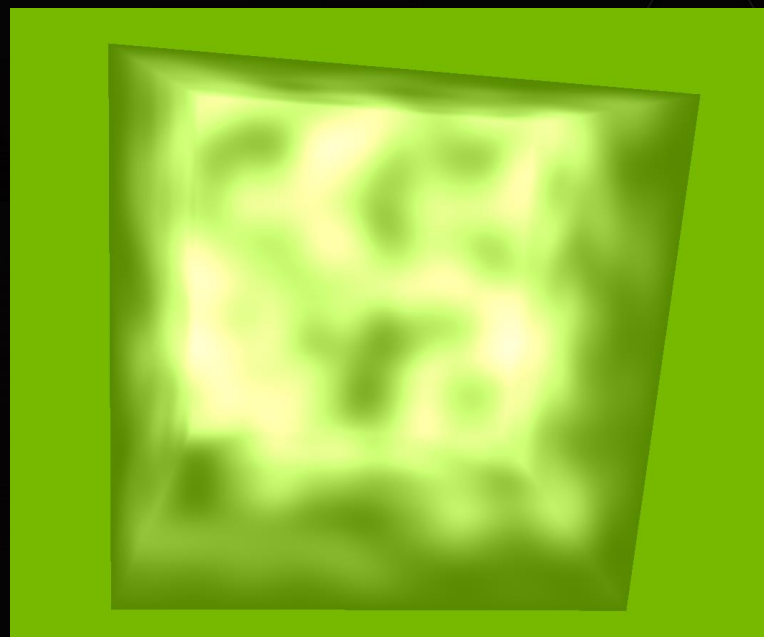


OVERVIEW

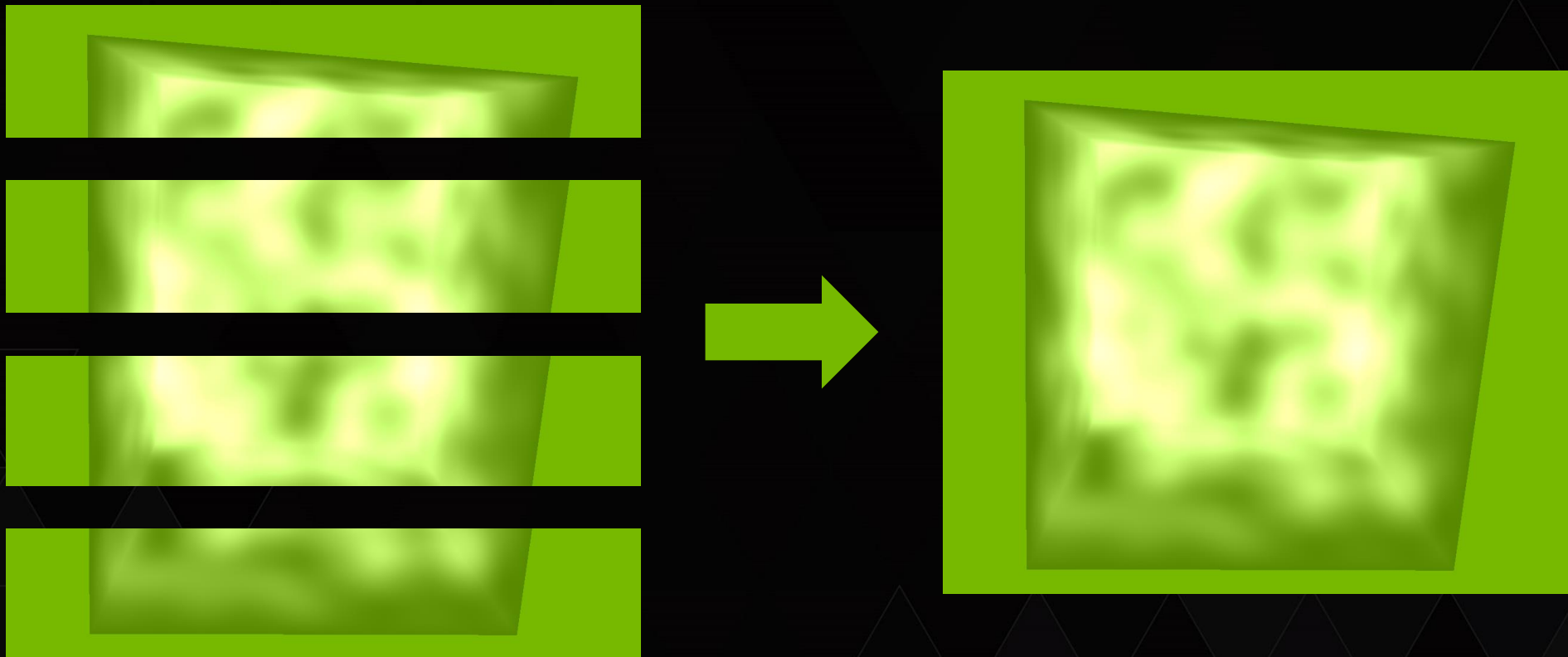
- ▶ Motivation
- ▶ Tools of the trade
 - ▶ Multi-GPU driver functions
 - ▶ Multi-GPU programming functions
- ▶ Multi threaded multi GPU renderer
 - ▶ General workflow
 - ▶ Different applications

SLICING IMAGE SPACE

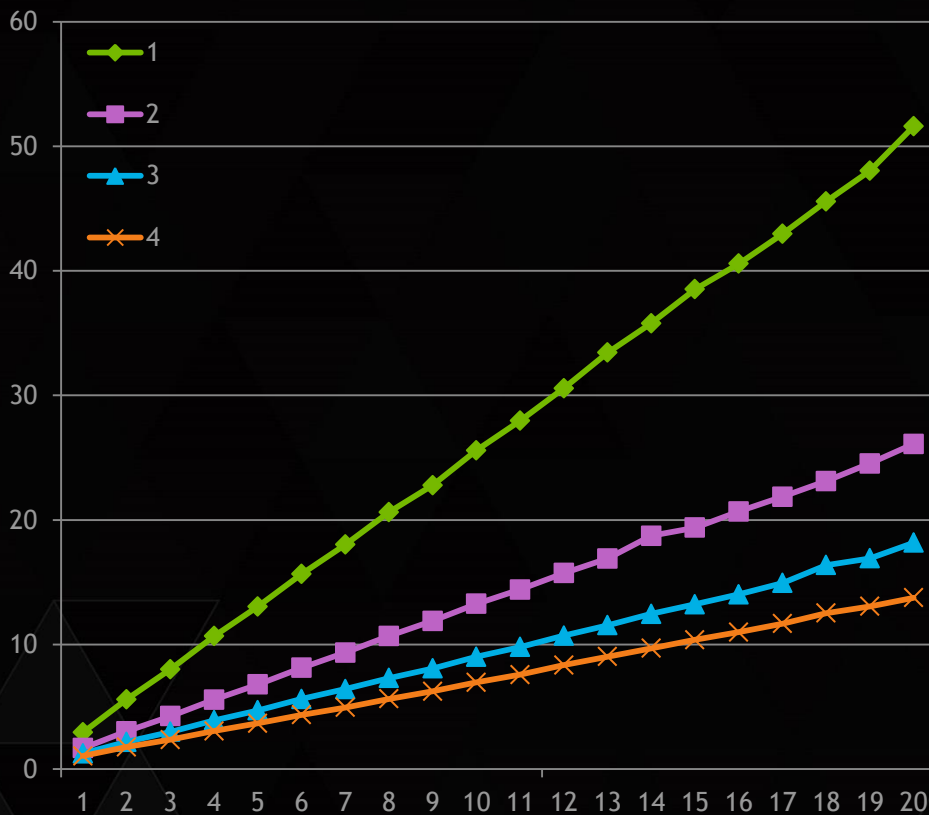
- ▶ Fragment bound scenario
- ▶ Split image up into N sub-images
- ▶ Every GPU renders the same scene, just different image regions
- ▶ Compose by reassembling output image from sub-images
- ▶ Scales when fragment load is distributed well



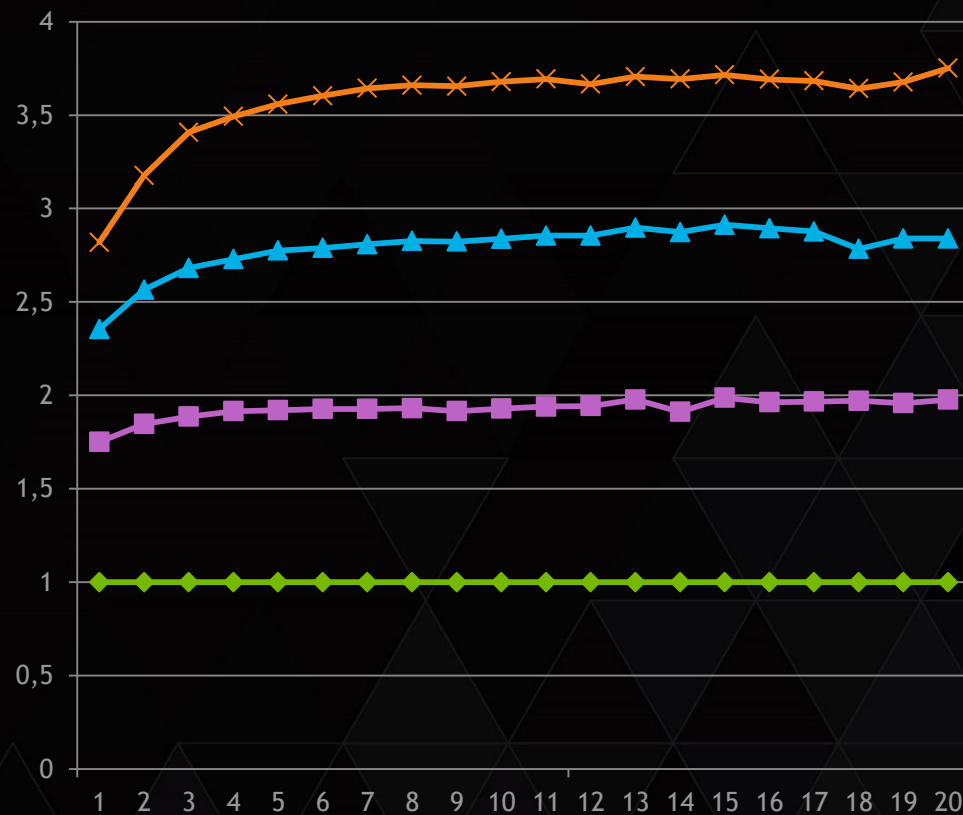
SLICING & COMPOSITION



RESULTS - SLICING IMAGE SPACE



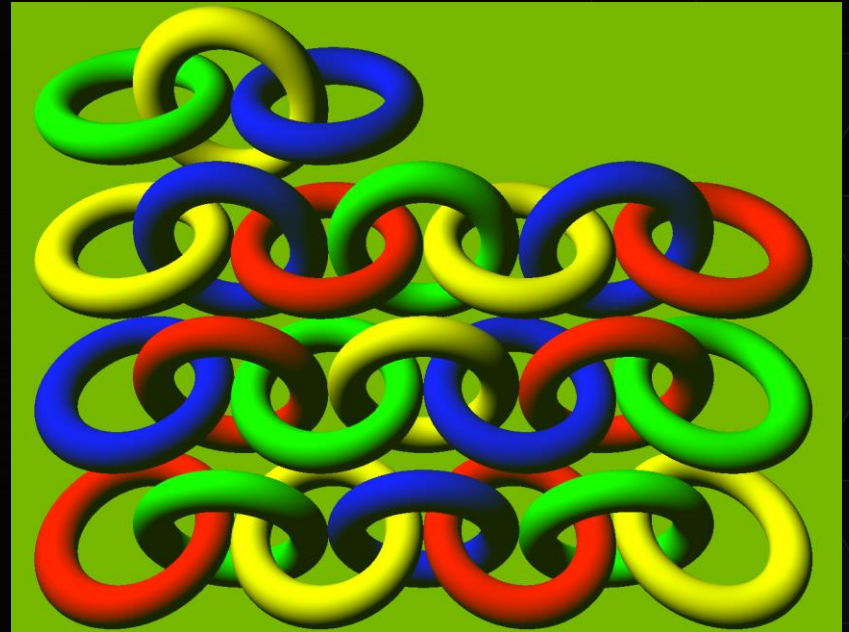
Frame time vs. workload



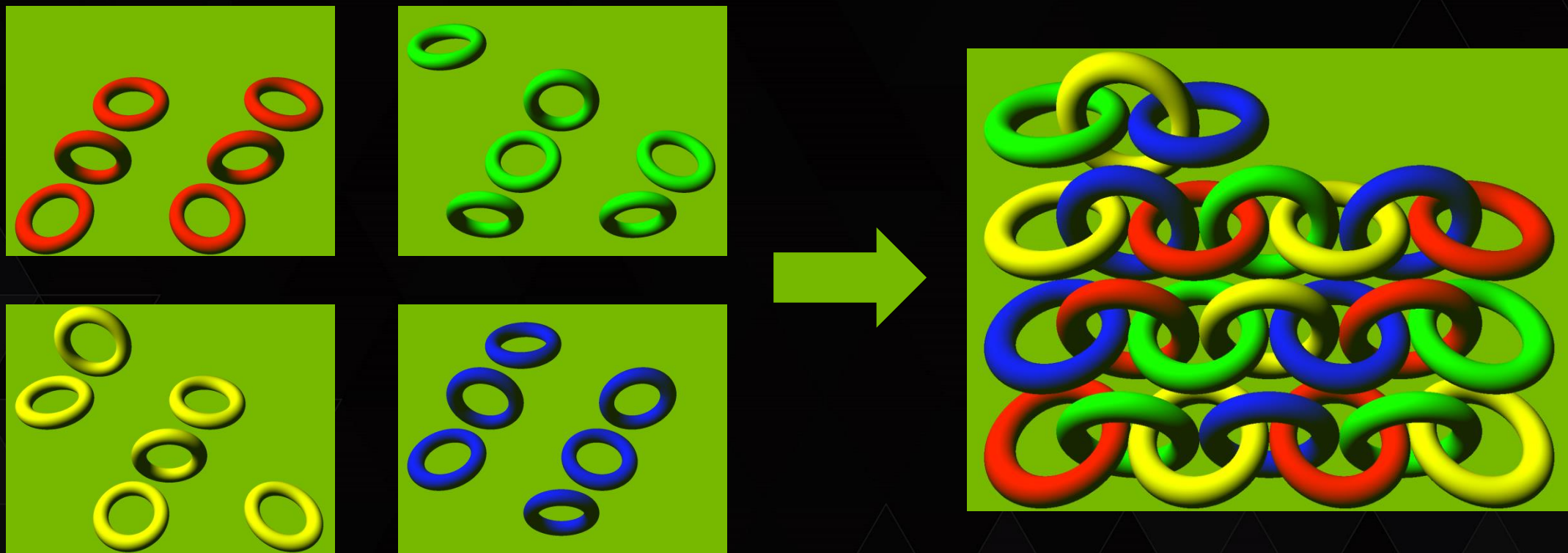
Scaling vs. Workload

SLICING VERTEX SPACE

- ▶ Geometry bound scenario
- ▶ Split scene up into N parts
- ▶ Every GPU renders the same frustum, but with a different sub-scene
- ▶ Compose output image by depth comparison
- ▶ Scales when geometry is distributed well
- ▶ Transfer full color and depth images

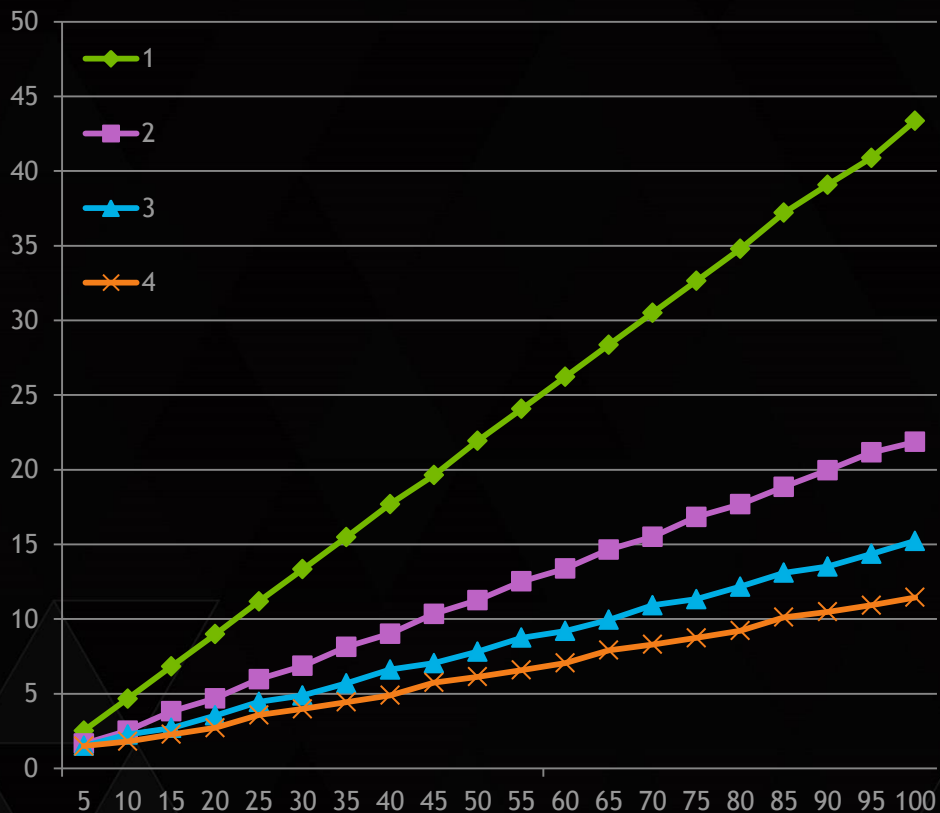


SLICING & COMPOSITION

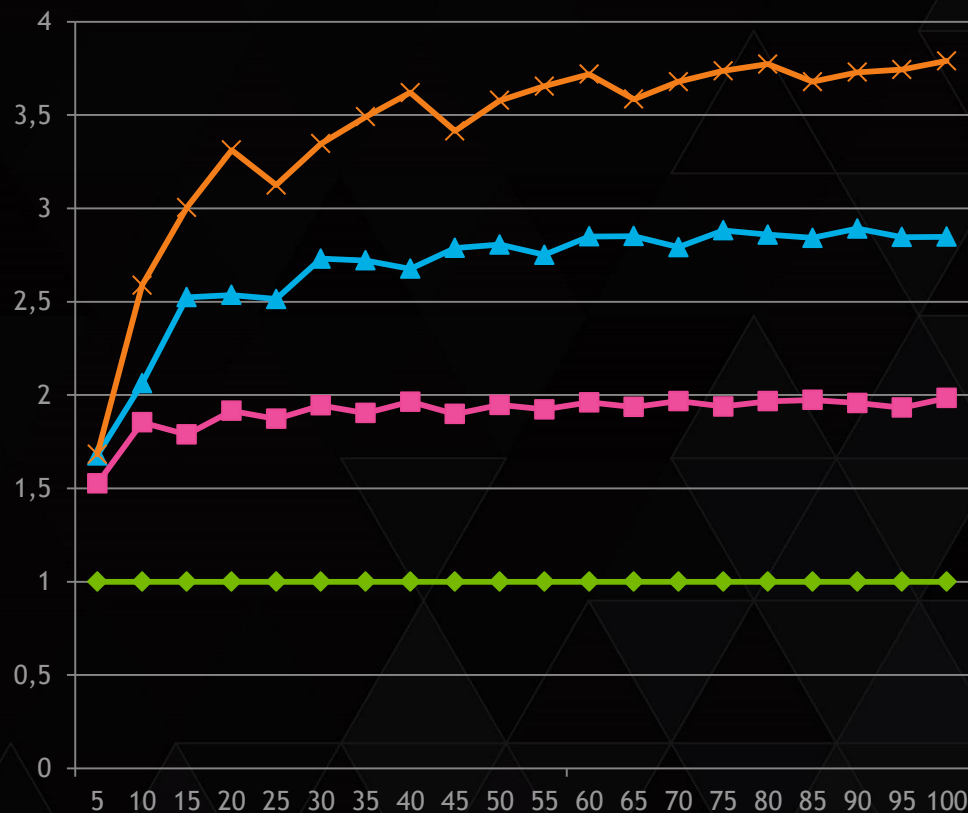


Every Torus: 724201 vertices / 722500 faces

RESULTS - SLICING VERTEX SPACE (LO RES)

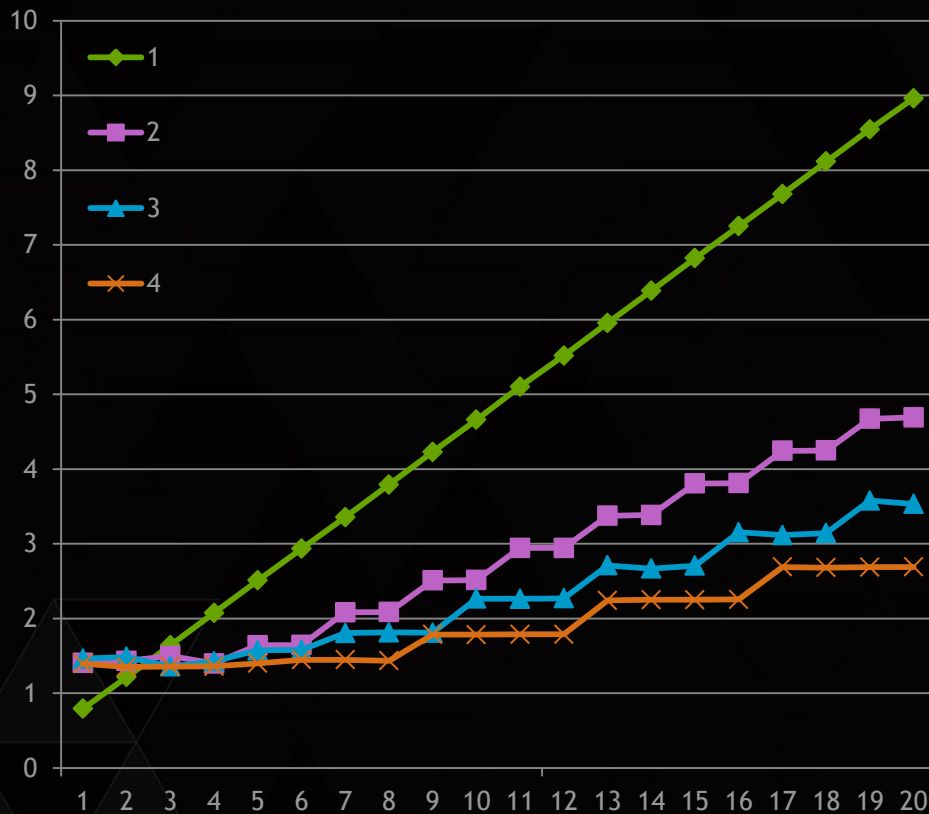


Frame time vs. #objects

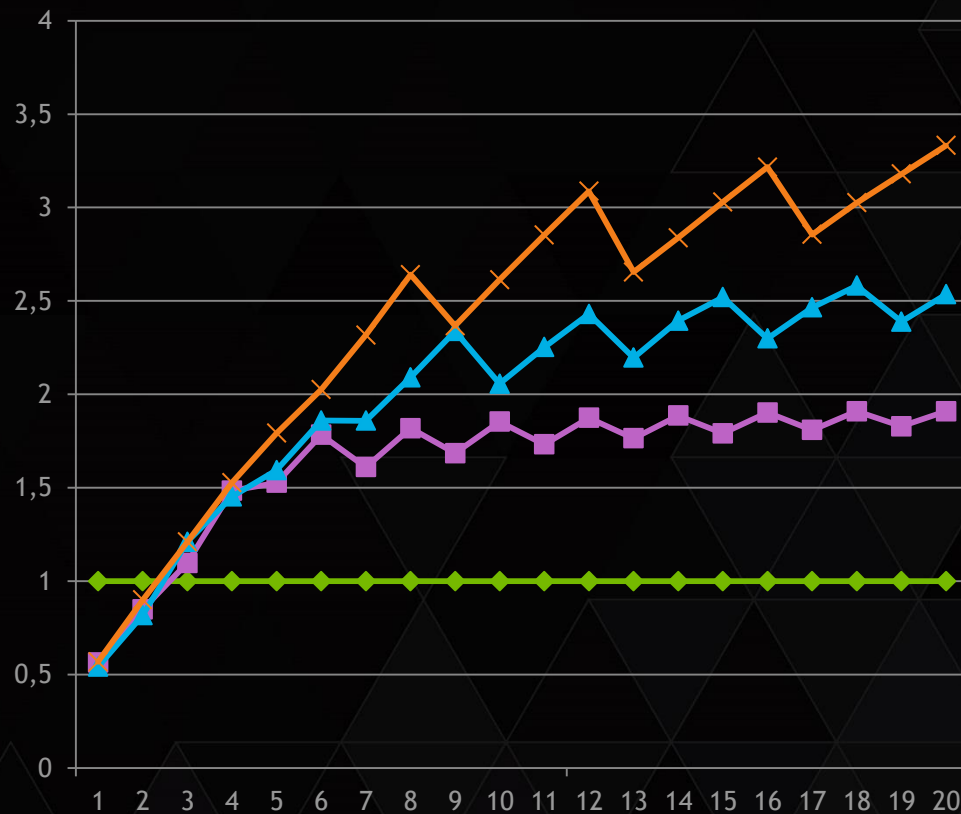


Scaling vs. #objects

RESULTS - SLICING VERTEX SPACE (LO RES)

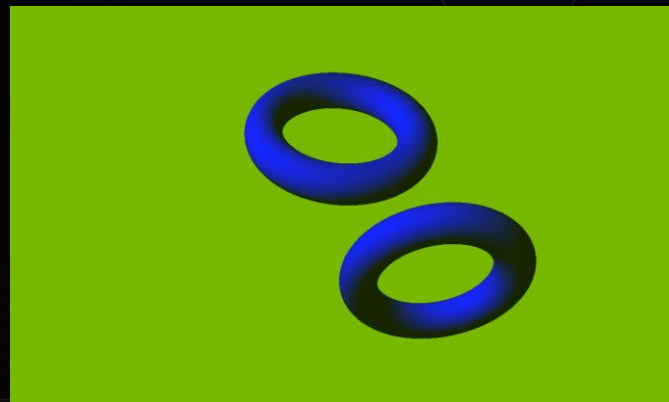
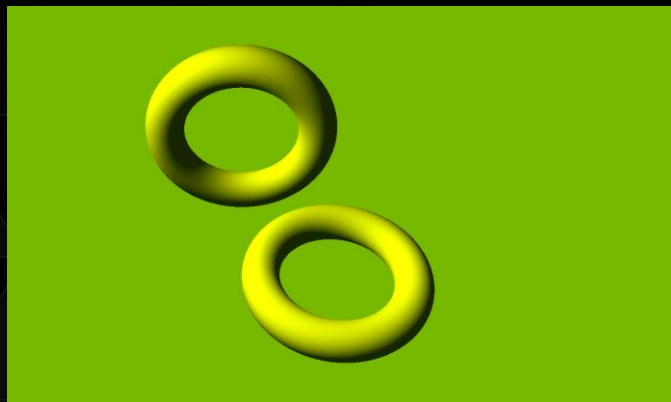
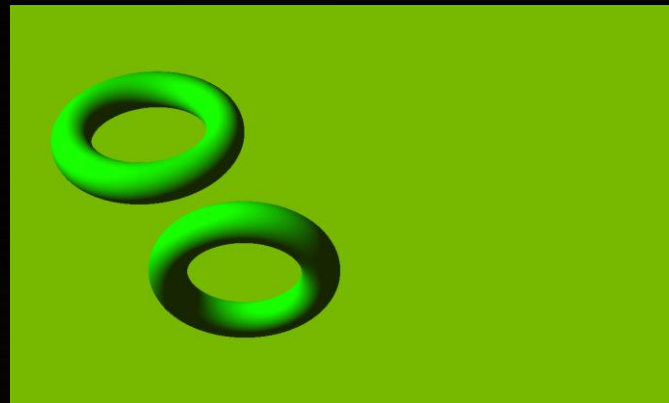
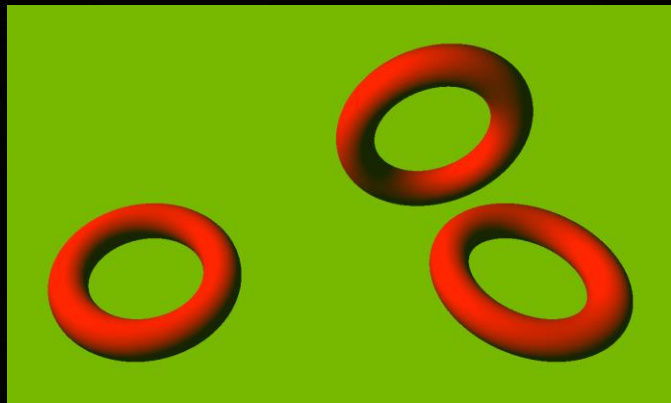


Frame time vs. #objects

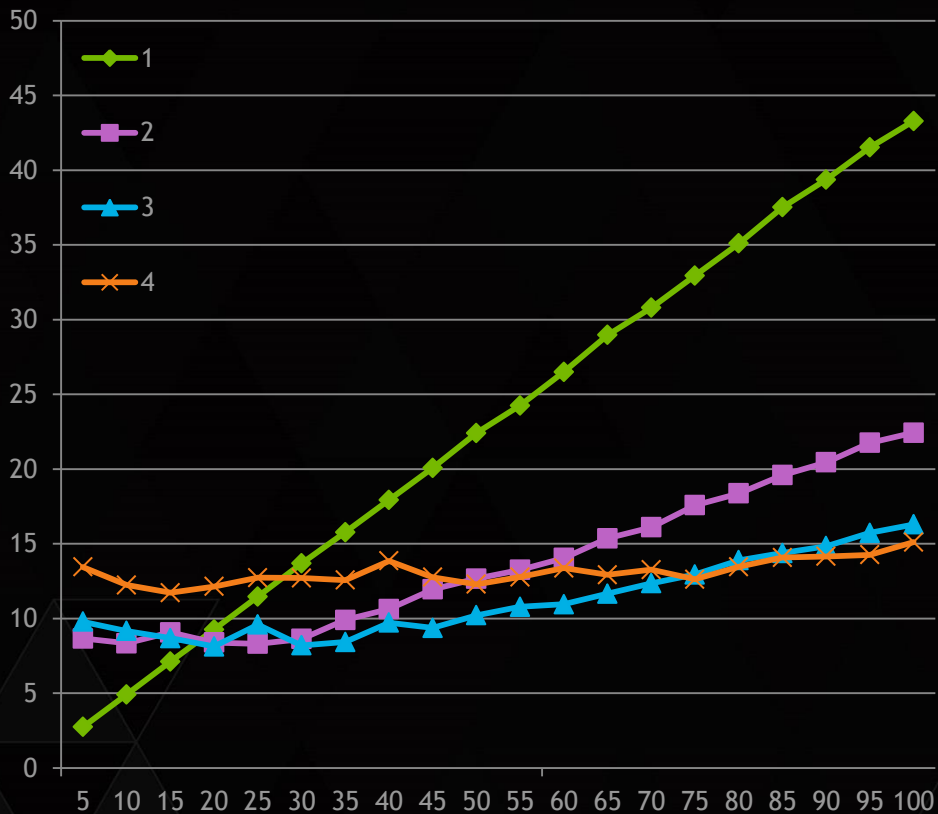


Scaling vs. #objects

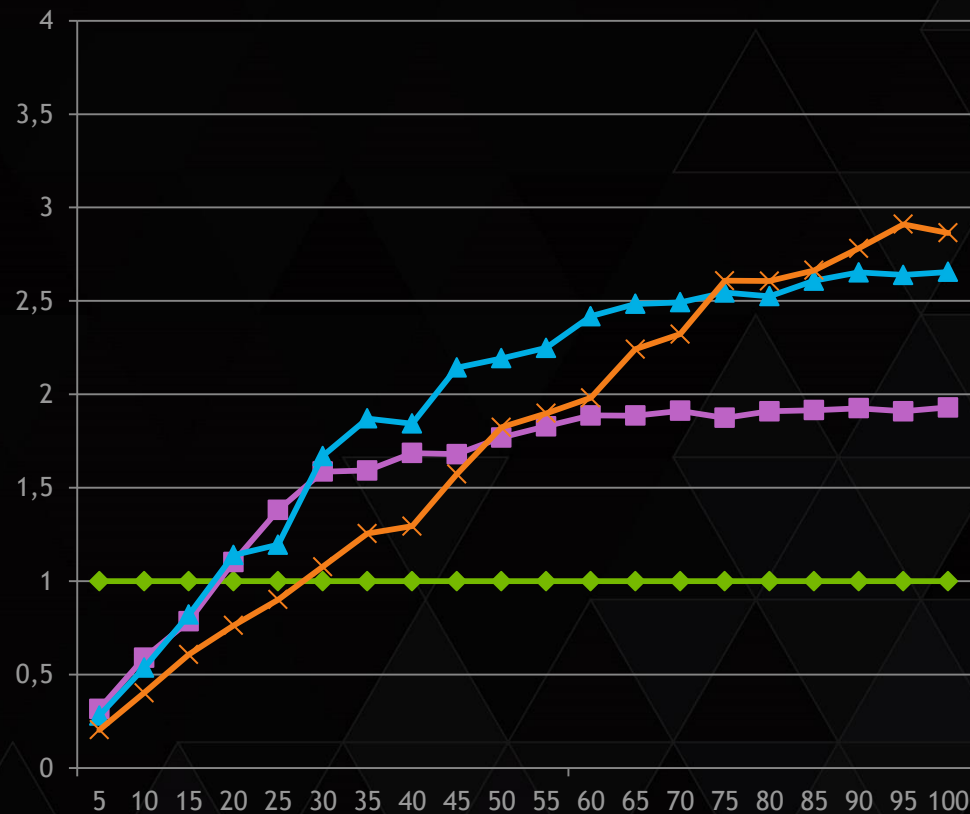
RESULTS - SLICING VERTEX SPACE



RESULTS - SLICING VERTEX SPACE (HI RES)



Frame time vs. #objects



Scaling vs. #objects

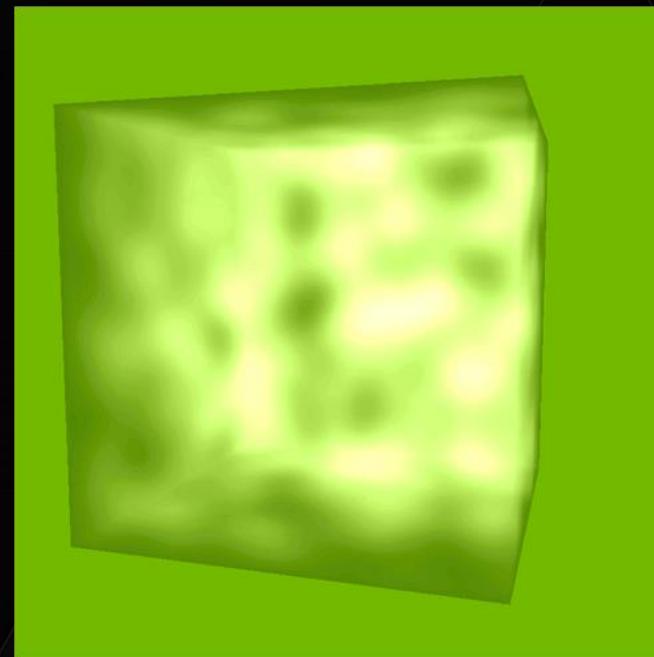
RESULTS - SLICING VERTEX SPACE (HI RES)

- ▶ PCIe 2.0 x16 can transport ~700 Full HD images per second
- ▶ Per displayed frame:
 - ▶ 4 Full HD color images
 - ▶ 4 Full HD depth images
- ▶ $700 / 8 = 87.5$ max fps, 11.4 min ms per frame
- ▶ 800x600 image: 2.6 min ms per frame
- ▶ 4k image: 45.6 min ms per frame
- ▶ Improvements: Compression / PCIe 3.0

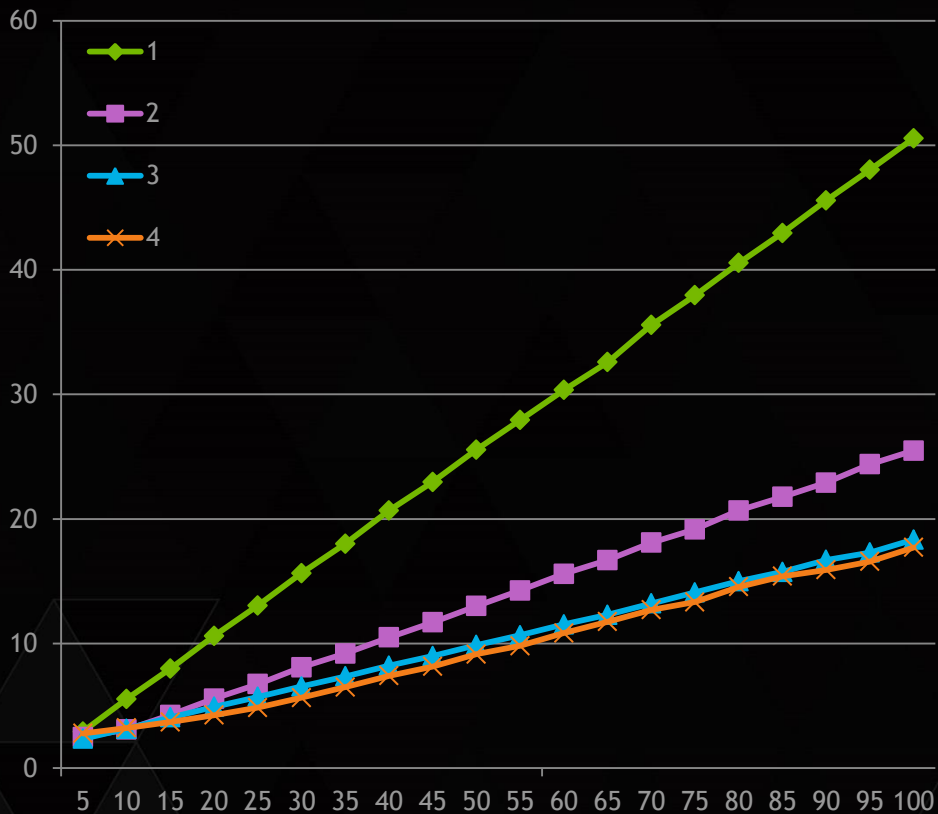
SLICING TIME

- ▶ General GPU bound scenario
- ▶ Implement „SLI AFR“, distribute whole frames across GPUs
- ▶ Every GPU renders a whole frame
- ▶ No composition, just display output image on display GPU
- ▶ Only scales without inter-frame dependencies

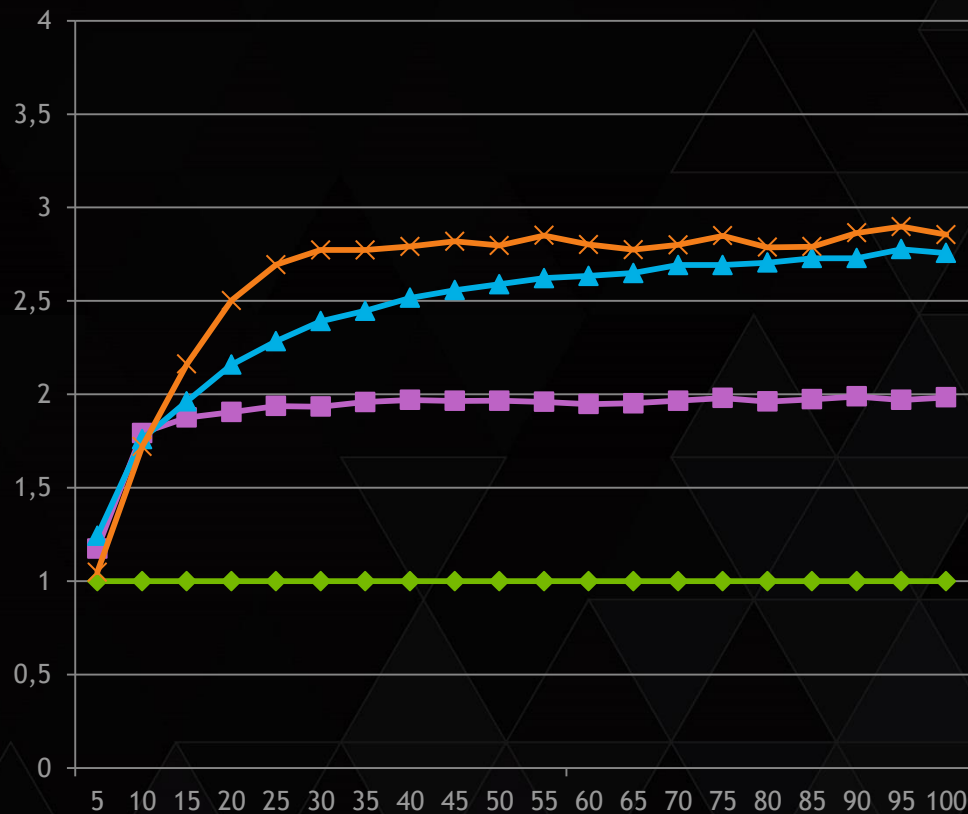
SLICING & COMPOSITION



RESULTS - SLICING TIME

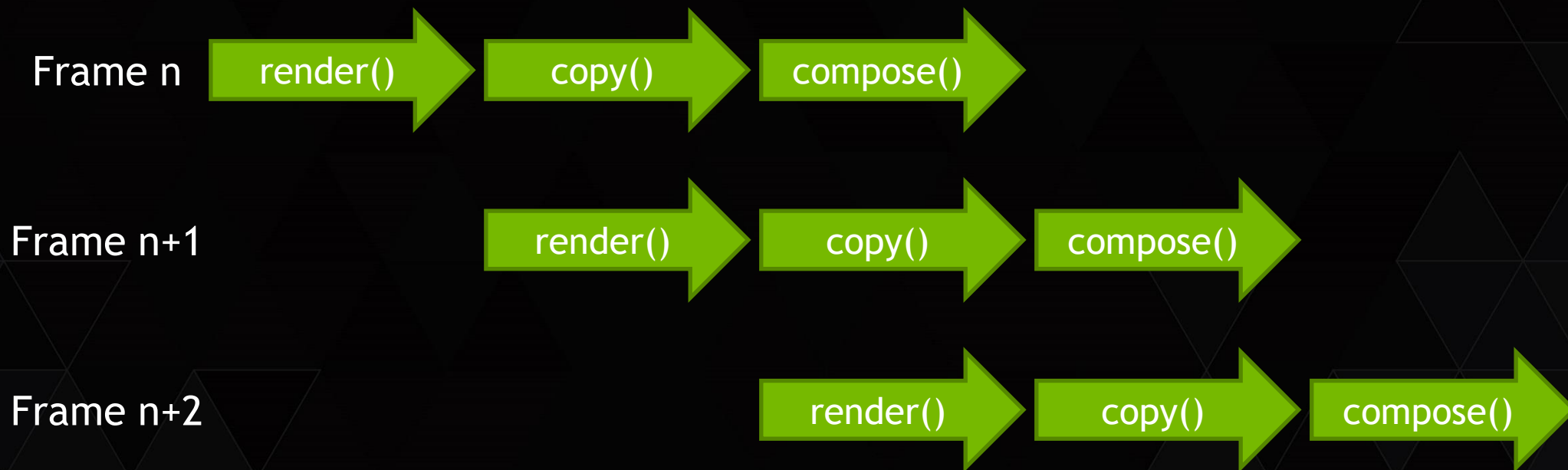


Frame time vs. workload



Scaling vs. Workload

RESULTS - SLICING TIME



IN CLOSING

- ▶ Other applications possible, e.g.
 - ▶ Stereo rendering
 - ▶ Volume rendering
 - ▶ Shadow passes
- ▶ Further questions?
 - ▶ iesser@nvidia.com
- ▶ Source code soon available at
<https://github.com/nvpro-samples>

GPU TECHNOLOGY
CONFERENCE

THANK YOU

JOIN THE CONVERSATION

#GTC15

