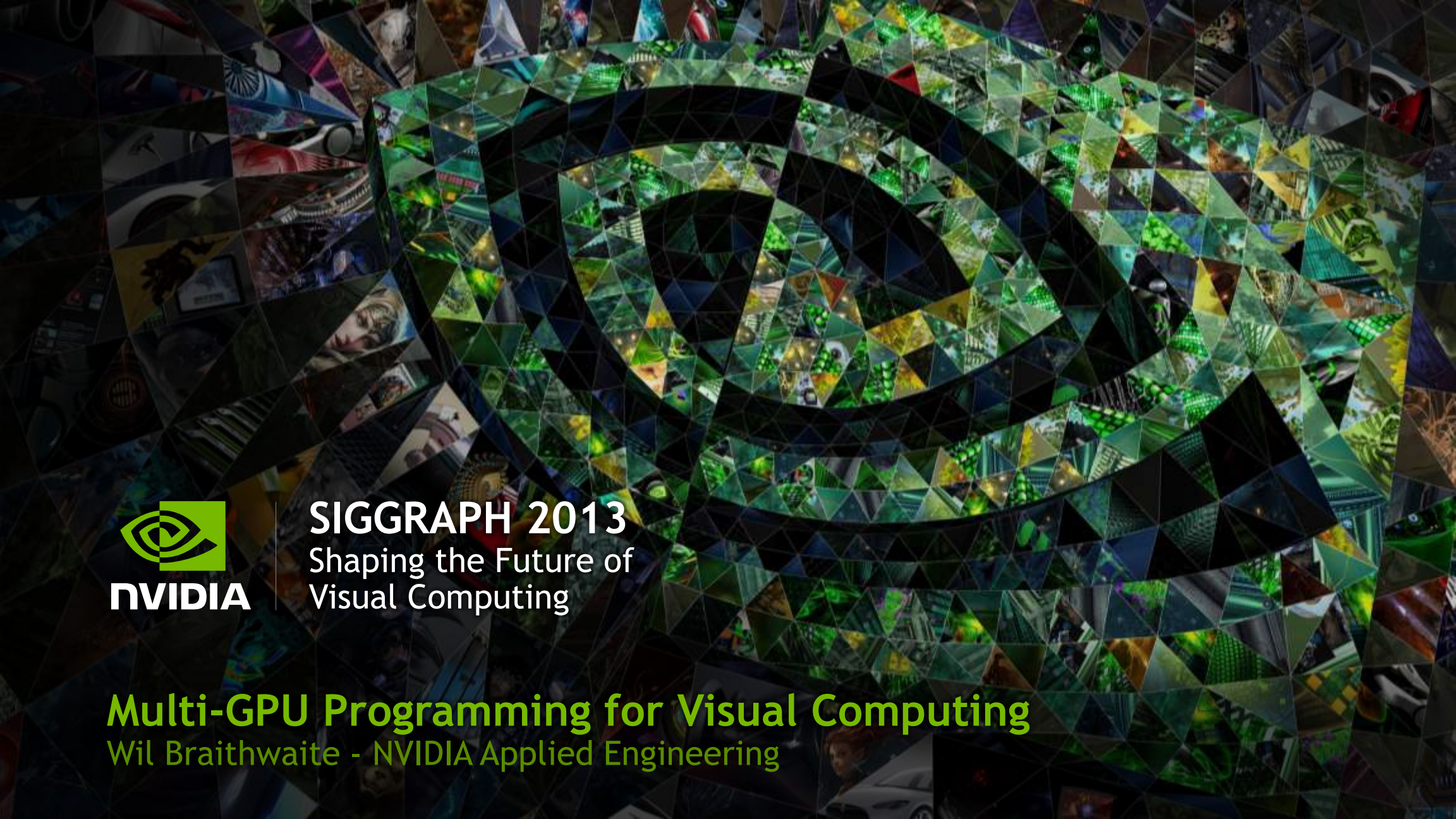


SIGGRAPH 2013
Shaping the Future of
Visual Computing

Multi-GPU Programming for Visual Computing

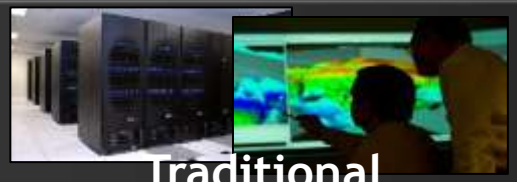
Wil Braithwaite - NVIDIA Applied Engineering

Talk Outline

- Compute and Graphics API Interoperability.
  - The basics.
  - Interoperability Methodologies & driver optimizations.
- Interoperability at a system level.
  - fine-grained control for managing scaling
- Demo.
- Peer-to-peer & UVA, NUMA considerations
- Scaling beyond one-to-one = single-compute : single-render
  - Enumerating Graphics & Compute Resources
  - many-to-many = Multiple-compute : multiple-render

NVIDIA Maximus Initiative

- Mixing Tesla and Quadro GPUs
  - Tight integration with OEMs and System Integrators
  - Optimized driver paths for GPU-GPU communication



Traditional Workstation

Simulate (Cluster) → Visualize

NVIDIA® MAXIMUS™ Workstation

$Simulate_0$

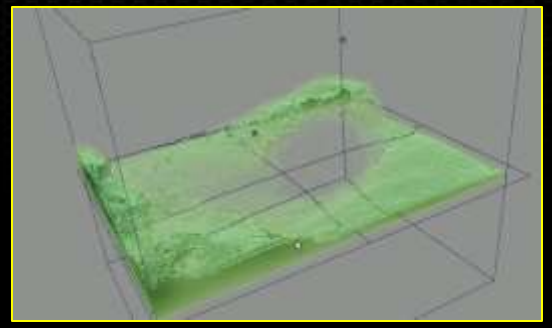
$Simulate_1$ + $Visualize_0$

$Simulate_2$ + $Visualize_1$

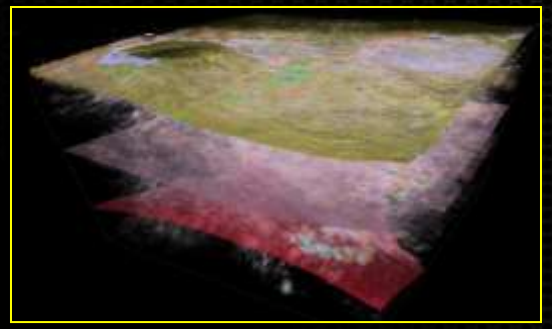
$Simulate_3$ + $Visualize_2$

Multi-GPU Compute+Graphics Use Cases

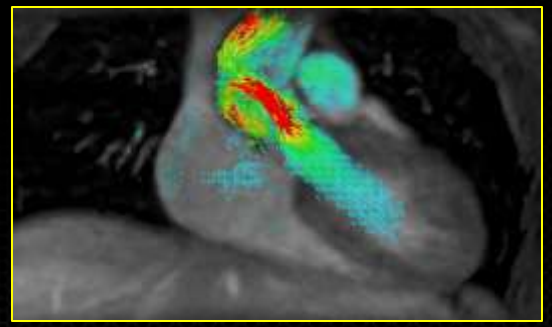
- Image processing
  - Multiple compute GPUs and a low-end display GPU for blitting
- Mixing rasterization with compute
  - Polygonal rendering done in OpenGL and input to Compute for further processing
- Visualization for HPC Simulation
  - Numerical simulation distributed across multiple compute GPUs, possibly on remote supercomputer



Interactive Fluid-dynamics



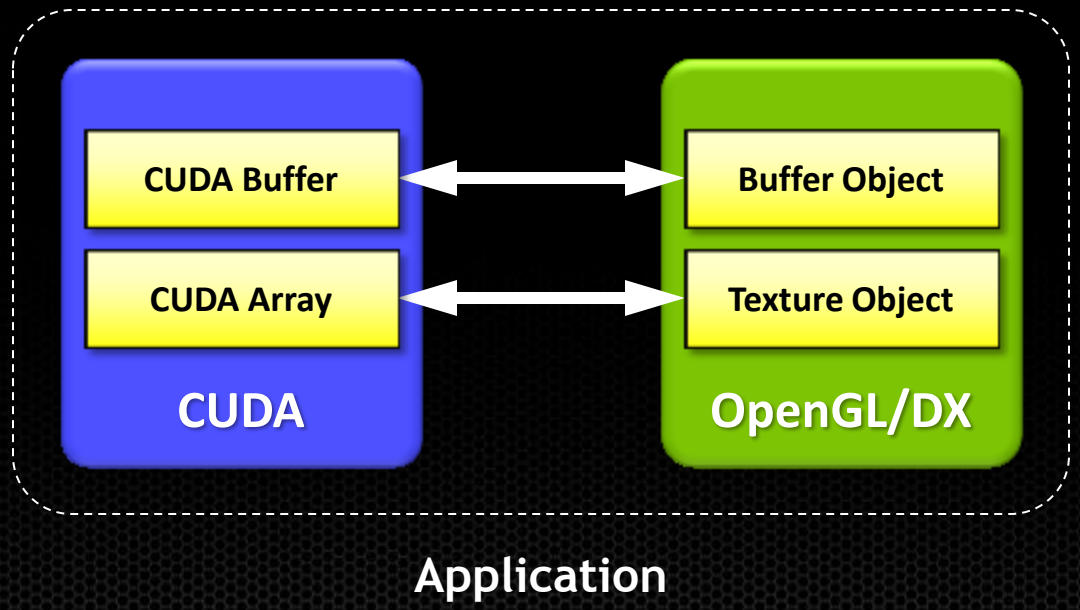
Seismic Interpretation (NVIDIA Index)



Real-time CFD Viz (Morpheus Medical)

Compute/Graphics interoperability

- Setup the objects in the graphics context.
- Register objects with the compute context.
- Map / Unmap the objects from the compute context.



Code Sample - Simple buffer interop

- Setup and Registration of Buffer Objects:

```
GLuint vboId;
cudaGraphicsResource_t vboRes;

// OpenGL buffer creation...
glGenBuffers(1, &vboId);
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, vboSize, 0, GL_STREAM_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Registration with CUDA.
cudaGraphicsGLRegisterBuffer(&vboRes, vboId, cudaGraphicsRegisterFlagsNone);
```

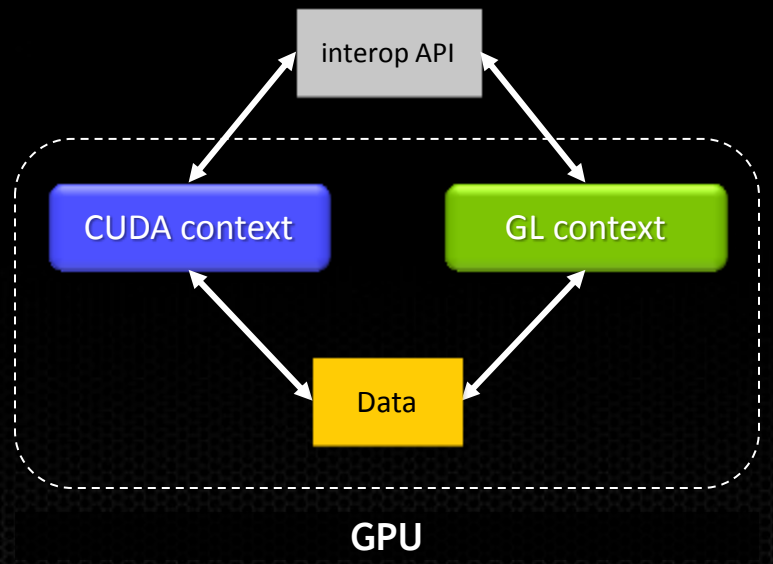
Code Sample - Simple buffer interop

- Mapping between contexts:

```
0
1  float* vboPtr;
2
3  while (!done)
4  {
5      cudaGraphicsMapResources(1, &vboRes, 0);
6
7      cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
8      runCUDA(vboPtr, 0);
9
10     cudaGraphicsUnmapResources(1, &vboRes, 0);
11
12     runGL(vboId);
13 }
14
```

Resource Behavior: Single-GPU

- The resource is shared. ☺
- Context switch is fast and independent on data size.



Code Sample - Simple buffer interop

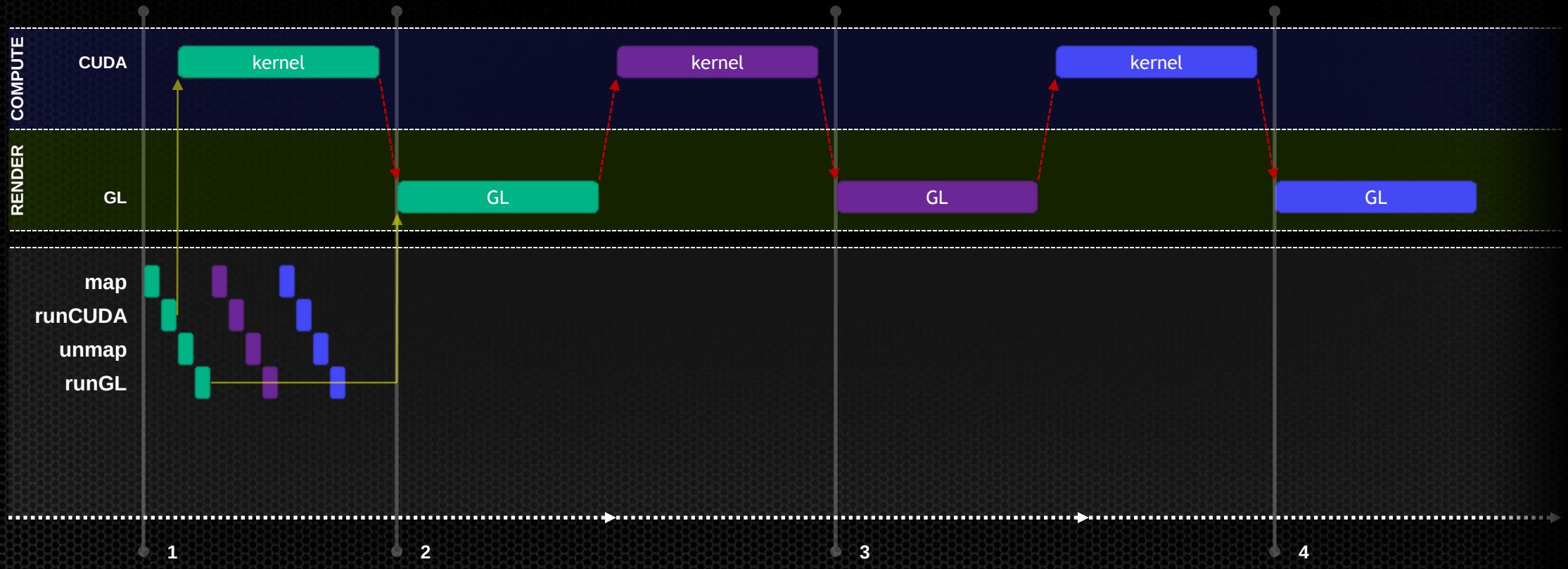
- Mapping between contexts:

```
0
1  float* vboPtr;
2
3  while (!done)
4  {
5      cudaGraphicsMapResources(1, &vboRes, 0);
6
7      cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
8      runCUDA(vboPtr, 0);
9
10     cudaGraphicsUnmapResources(1, &vboRes, 0);
11
12     runGL(vboId);
13 }
14
```

Context-switching happens *when* these commands are processed.

Timeline: Single-GPU

- Driver-interop



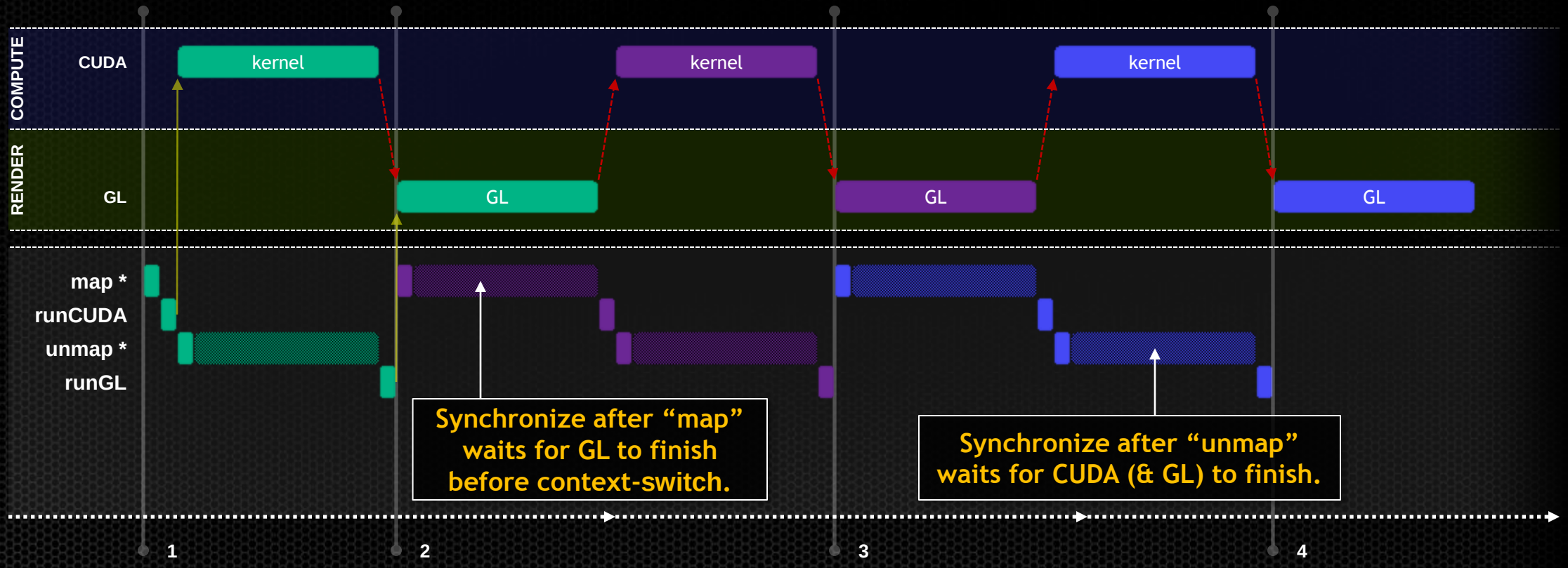
Code Sample - Simple buffer interop

- Adding synchronization for analysis:

```
0
1  float* vboPtr;
2
3  while (!done)
4  {
5      cudaGraphicsMapResources(1, &vboRes, 0);
6      cudaStreamSynchronize(0);
7
8      cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
9      runCUDA(vboPtr, 0);
10
11     cudaGraphicsUnmapResources(1, &vboRes, 0);
12     cudaStreamSynchronize(0);
13
14     runGL(vboId);
15 }
```

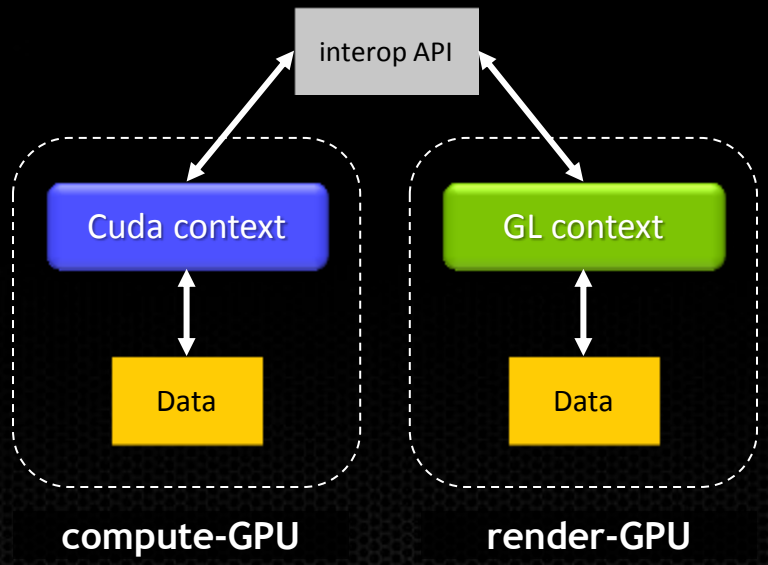
Timeline: Single-GPU

- Driver-interop, synchronous*
  - (we synchronize after map and unmap calls)



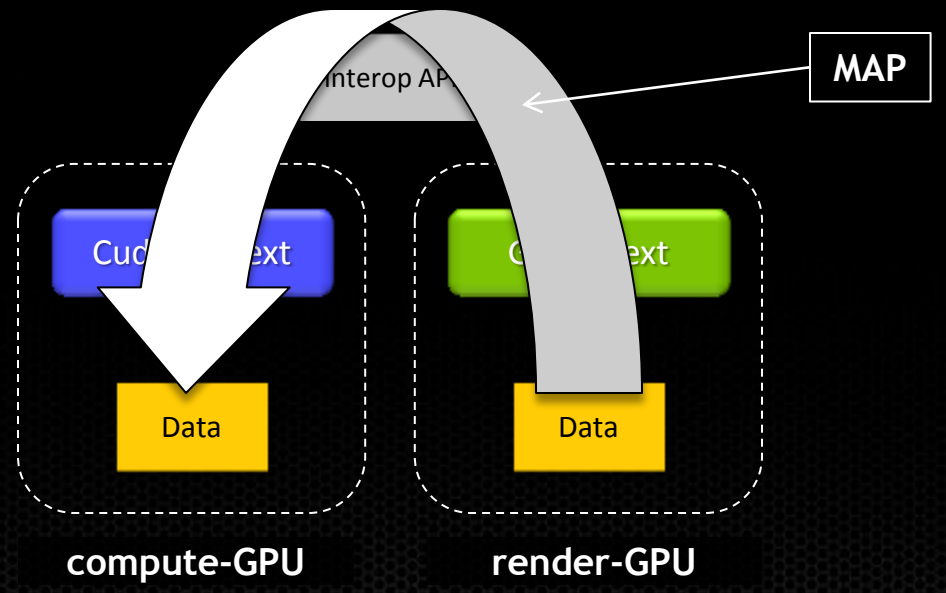
Resource Behavior: Multi-GPU

- Each GPU has a copy of the resource.
- Context-switch is dependent on data size, because driver must copy data. ☹



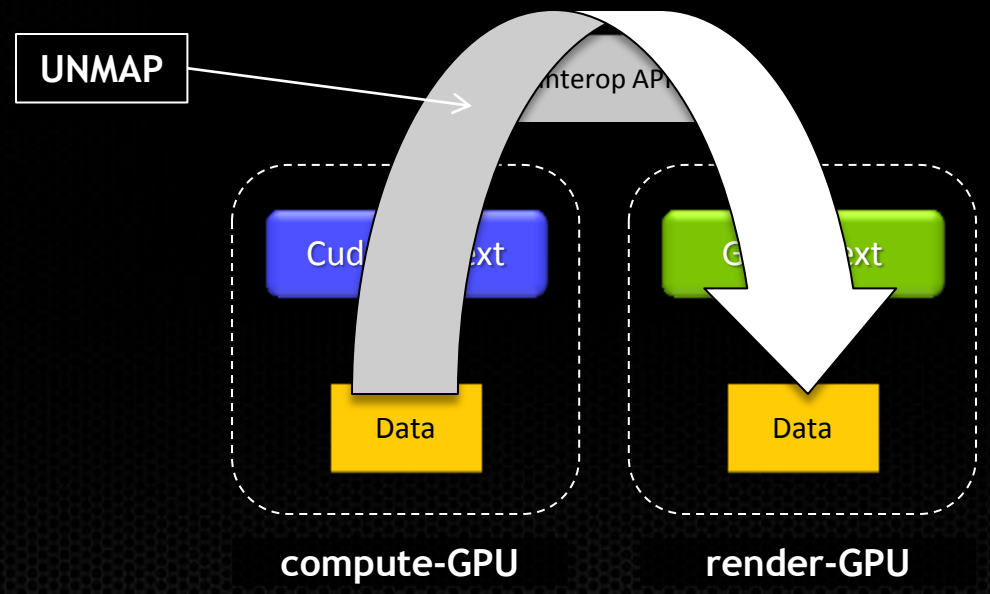
Resource Behavior: Multi-GPU

- Each GPU has a copy of the resource.
- Context-switch is dependent on data size, because driver must copy data. ☹



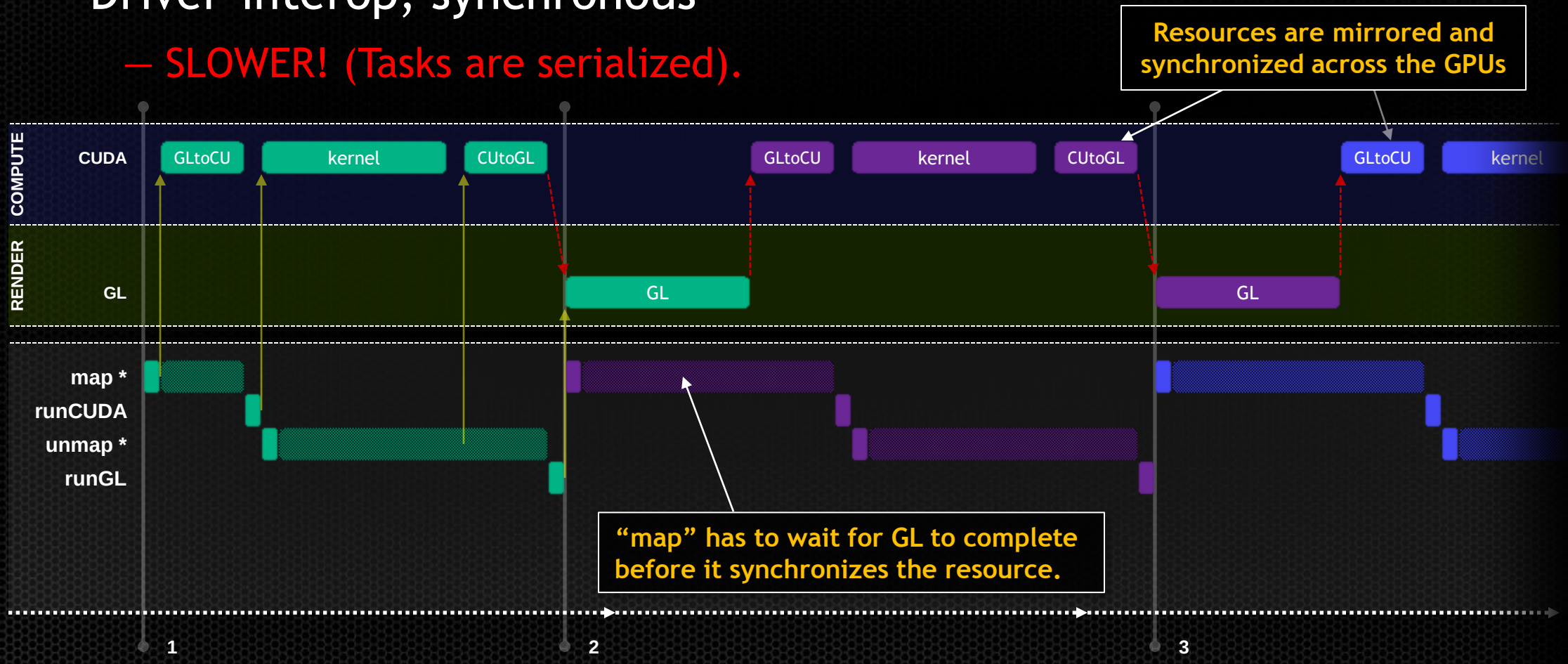
Resource Behavior: Multi-GPU

- Each GPU has a copy of the resource.
- Context-switch is dependent on data size, because driver must copy data. ☹



Timeline: Multi-GPU

- Driver-interop, synchronous*
  - SLOWER! (Tasks are serialized).



Interoperability Methodologies

- READ-ONLY
  - GL produces... and CUDA consumes.
    - e.g. Post-process the GL render in CUDA.
- WRITE-DISCARD
  - CUDA produces... and GL consumes.
    - e.g. CUDA simulates fluid, and GL renders result.

- READ & WRITE
  - Useful if you want to use the rasterization pipeline.
    - e.g. Feedback loop:
      - runGL(texture) → framebuffer
      - runCUDA(framebuffer) → texture

Code Sample - WRITE-DISCARD

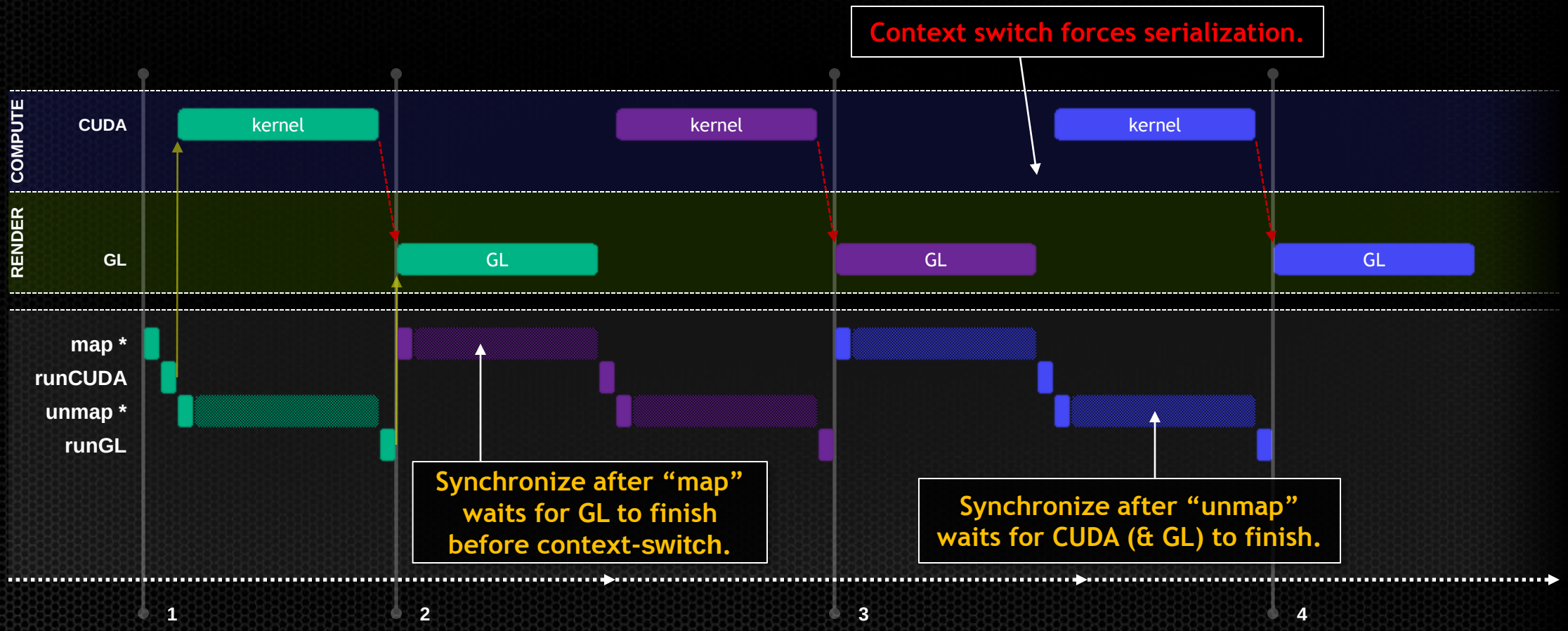
- CUDA produces… and OpenGL consumes:

```
float* vboPtr;

cudaGraphicsResourceSetMapFlags(vboRes, cudaGraphicsMapFlagsWriteDiscard);

while (!done)
{
    cudaGraphicsMapResources(1, &vboRes, 0);
    cudaStreamSynchronize(0);

    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    runCUDA(vboPtr, 0);

    cudaGraphicsUnmapResources(1, &vboRes, 0);
    cudaStreamSynchronize(0);

    runGL(vboId);
}
```

Hint that we do not care about the previous contents of buffer.

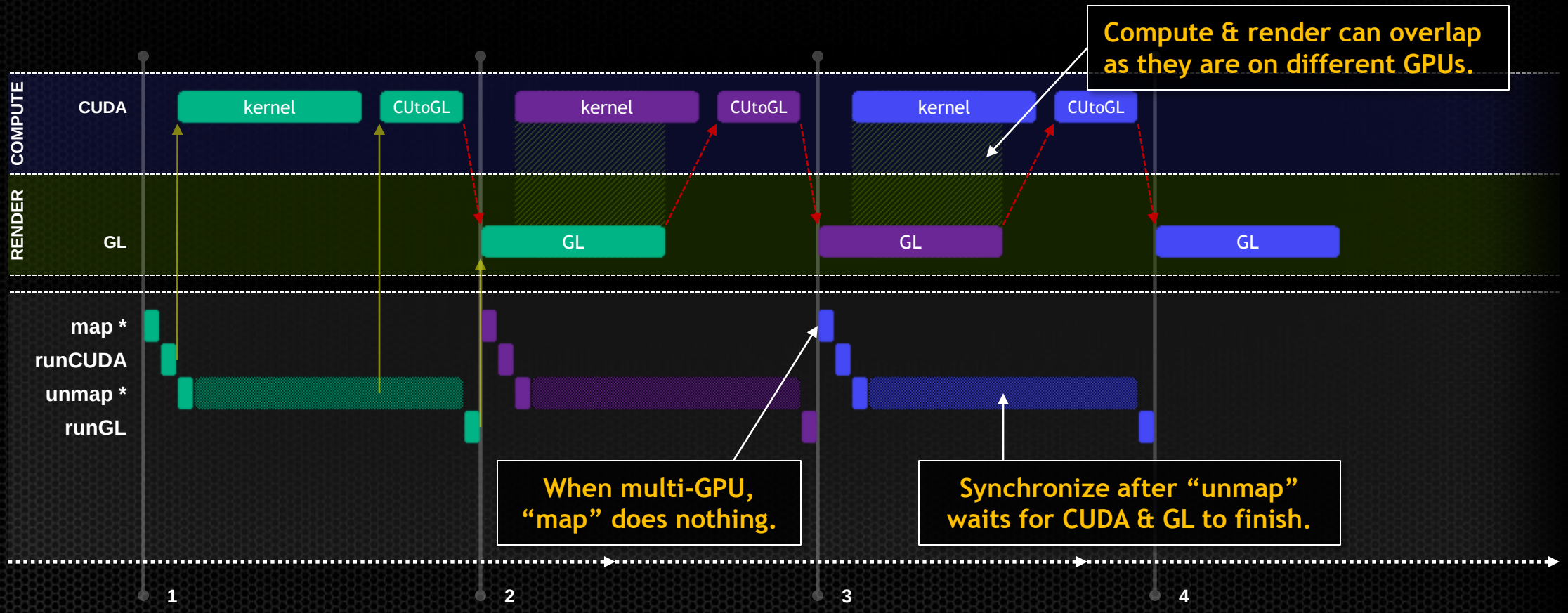
Timeline: Single-GPU

- Driver-interop, synchronous*, WRITE-DISCARD



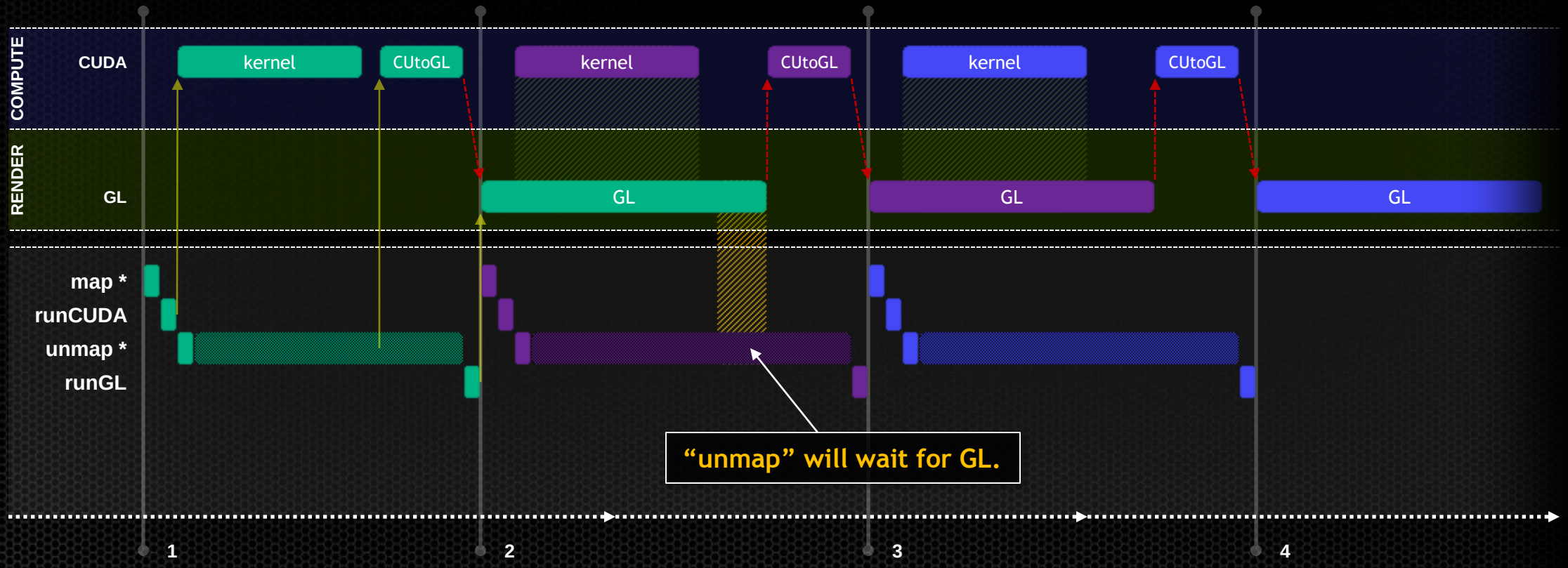
Timeline: Multi-GPU

- Driver-interop, synchronous*, WRITE-DISCARD

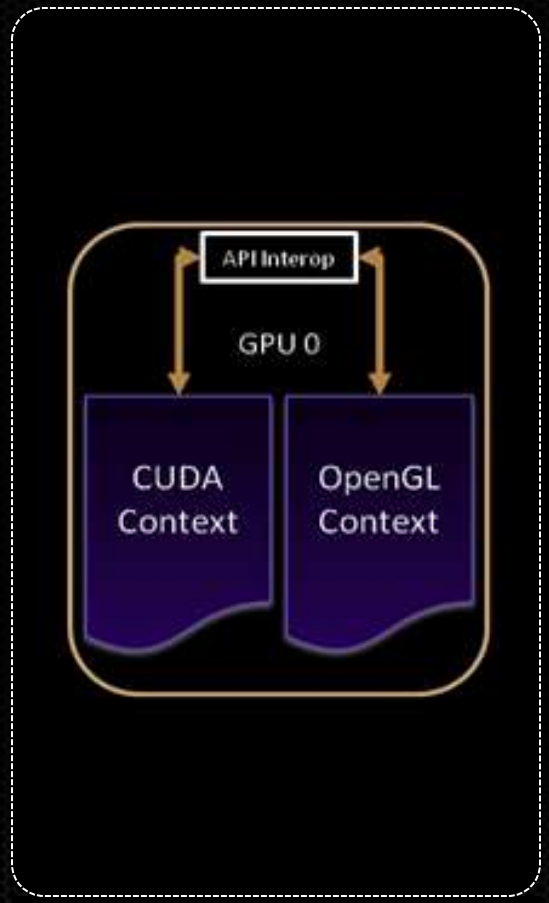


Timeline: Multi-GPU

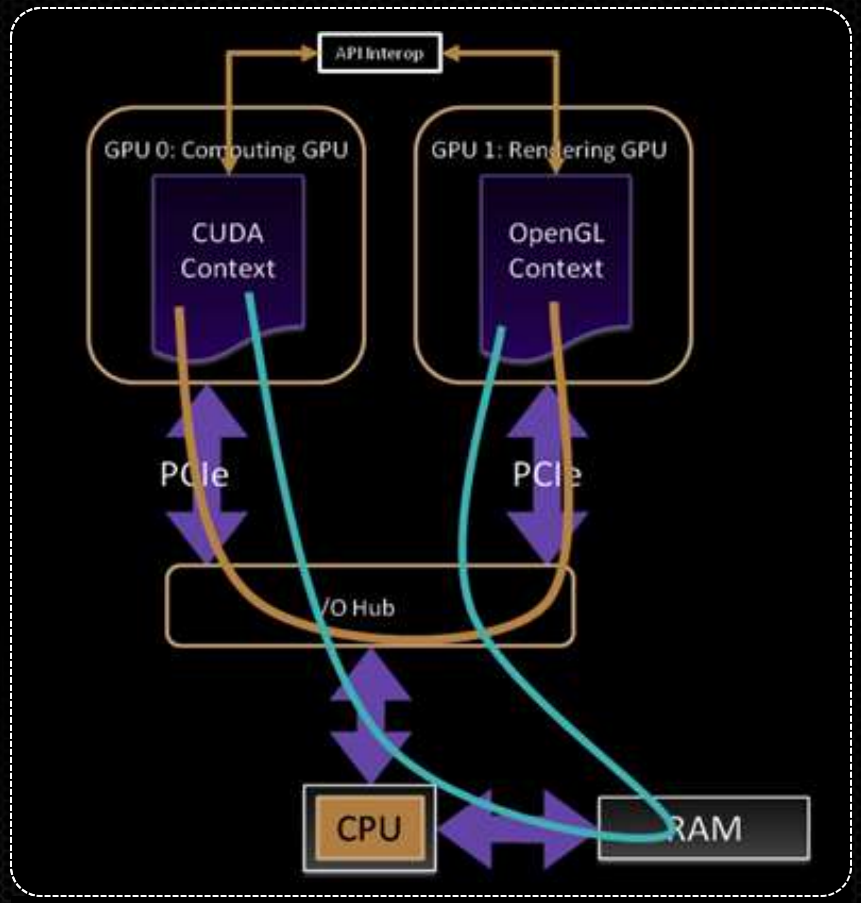
- Driver-interop, synchronous*, WRITE-DISCARD
  - if render is long...



Driver-Interop: System View

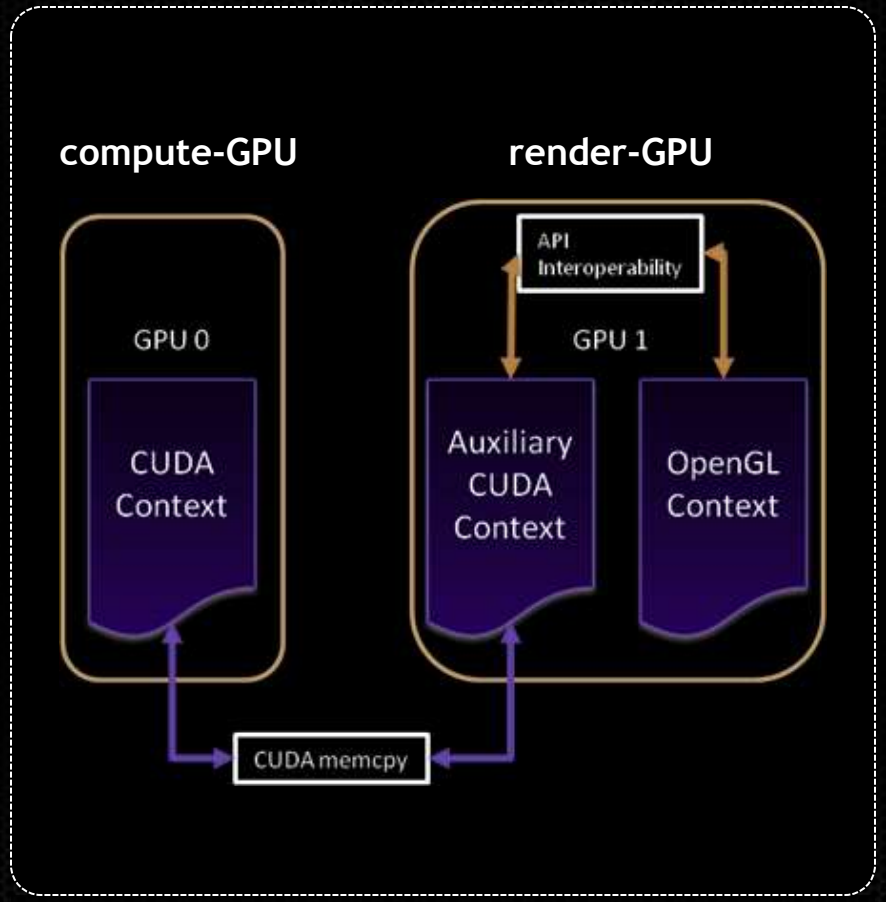


Single-GPU



Multi-GPU

Manual-Interop: System View



Multi-GPU

Manual-Interop

- Driver-Interop hides all the complexity
- but sometimes...
  - We don’t want to transfer all data
    - Kernel may only compute subregions.
  - We may have a complex system with multiple compute-GPUs and/or render-GPUs.
  - We have application specific pipelining and multi-buffering
  - May have some CPU code in your algorithm between compute and graphics.

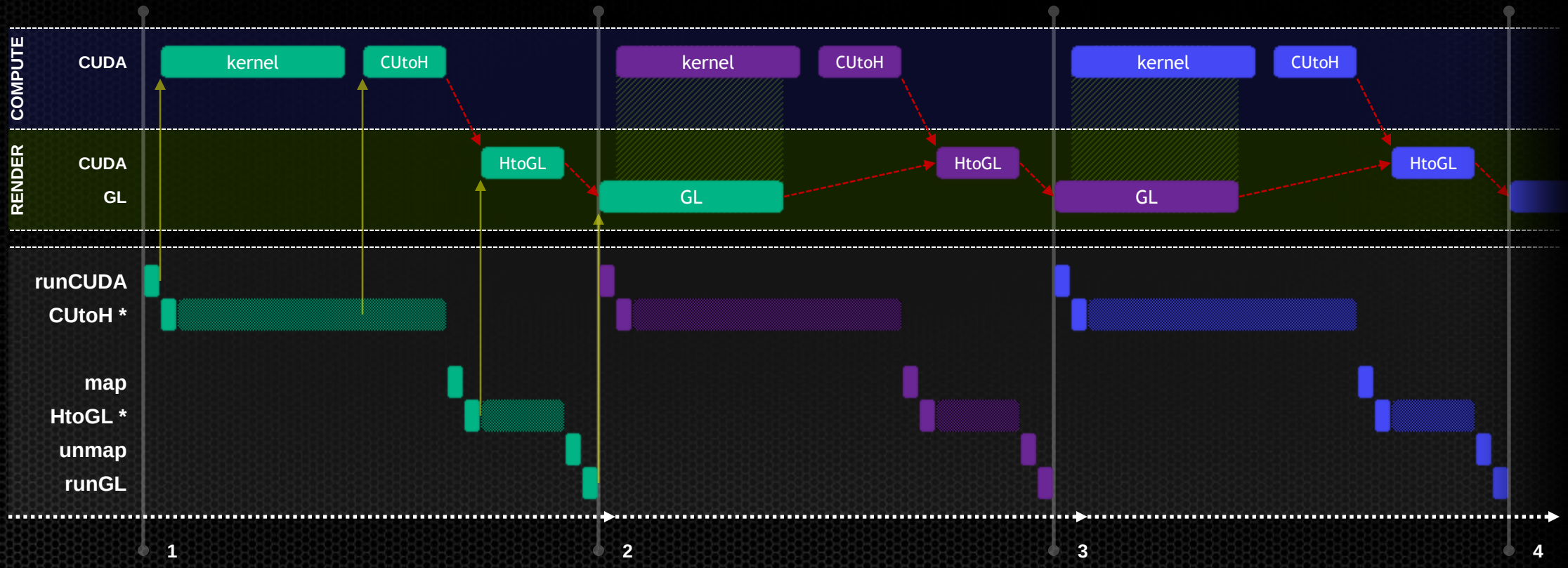
Code Sample - Manual-Interop

```
cudaMalloc((void**)&d_data, vboSize);
cudaHostAlloc((void**)&h_data, vboSize, cudaHostAllocPortable);

while (!done) {
    // Compute data in temp buffer, and copy to host...
    runCUDA(d_data, 0);
    cudaMemcpyAsync(h_data, d_data, vboSize, cudaMemcpyDeviceToHost, 0);
    cudaStreamSynchronize(0);

    // Map the render-GPU's resource and upload the host buffer...
    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, 0);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaMemcpyAsync(vboPtr, h_data, size, cudaMemcpyHostToDevice, 0);
    cudaGraphicsUnmapResources(1, &vboRes, 0);
    cudaStreamSynchronize(0);
    cudaSetDevice(computeGPU);

    runGL(vboId);
}
```

Create a temporary buffer in pinned host-memory.

Timeline: Multi-GPU

- Manual-interop, synchronous*, WRITE-DISCARD

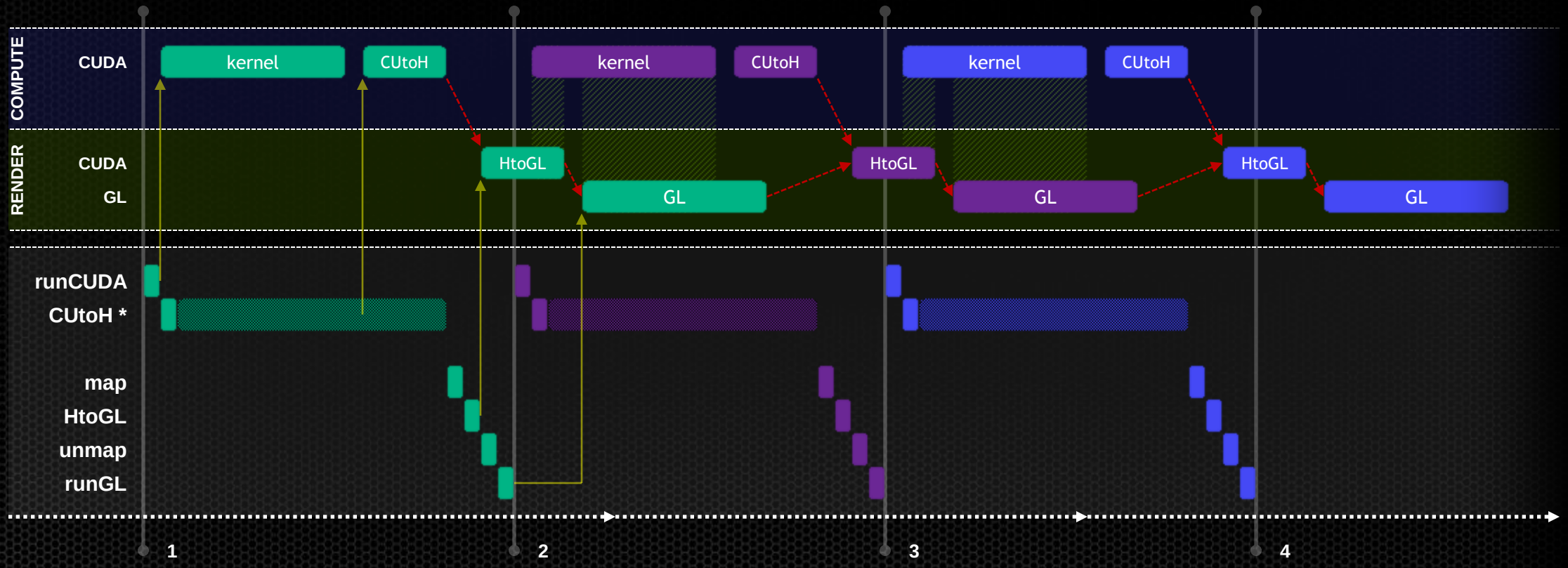


Code Sample - Manual Interop (Async)

```
cudaMalloc((void**)&d_data, vboSize);
cudaHostAlloc((void**)&h_data, vboSize, cudaHostAllocPortable);

while (!done) {
    // Compute data in temp buffer, and copy to host...
    runCUDA(d_data, 0);
    cudaMemcpyAsync(h_data, d_data, vboSize, cudaMemcpyDeviceToHost, 0);
    cudaStreamSynchronize(0);

    // Map the render-GPU's resource and upload the host buffer...
    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, 0);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaMemcpyAsync(vboPtr, h_data, size, cudaMemcpyHostToDevice, 0);
    cudaGraphicsUnmapResources(1, &vboRes, 0);
    cudaSetDevice(computeGPU);
    // cudaStreamSynchronize(0);

    runGL(vboId);
}
```

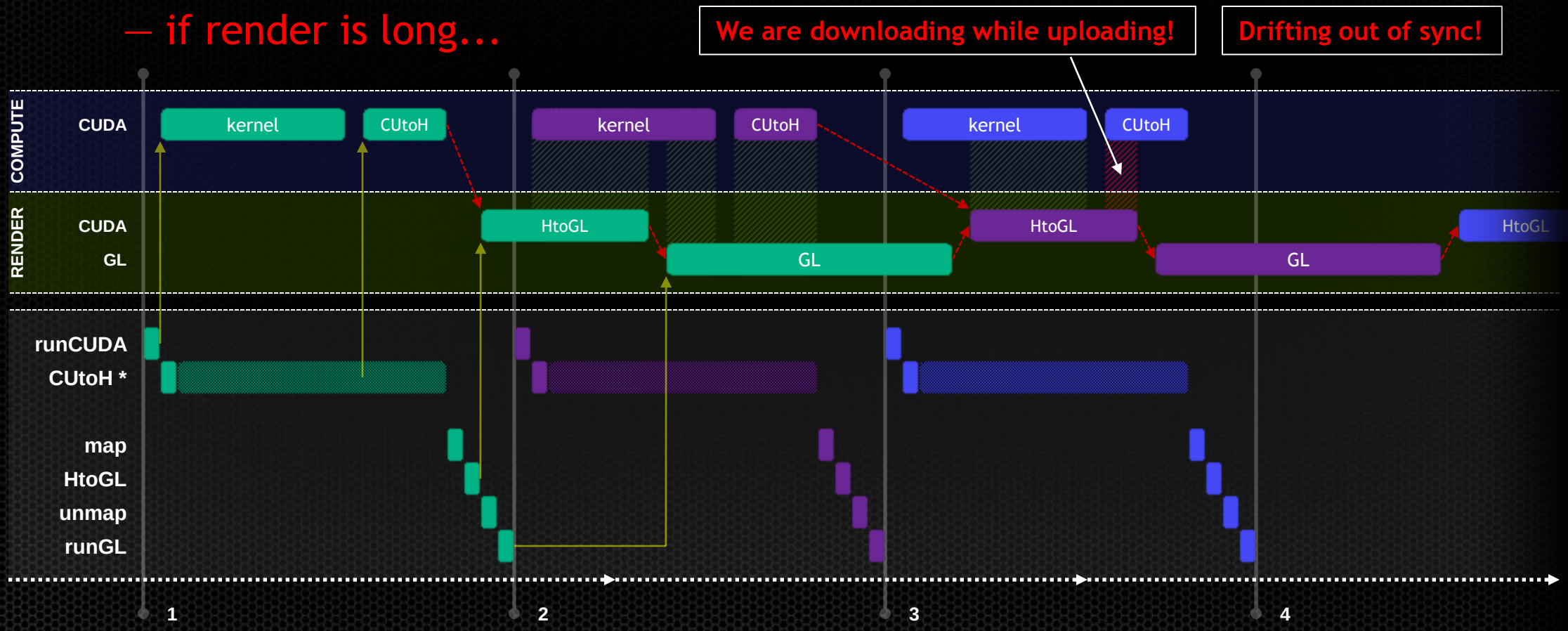
Timeline: Multi-GPU

- Manual-interop, asynchronous, WRITE-DISCARD



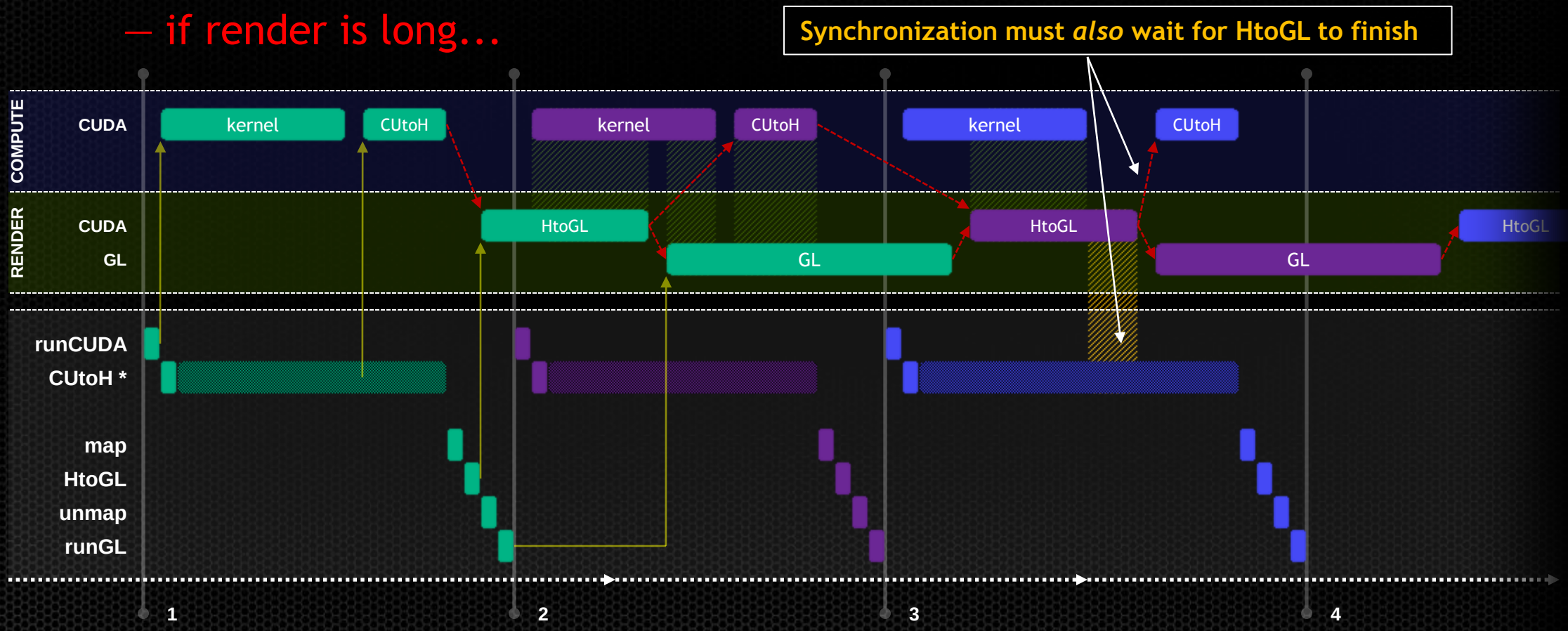
Timeline: Multi-GPU

- Manual-interop, asynchronous, WRITE-DISCARD
  - if render is long...



Timeline: Multi-GPU (safe Async)

- Manual-interop, asynchronous, WRITE-DISCARD
  - if render is long...

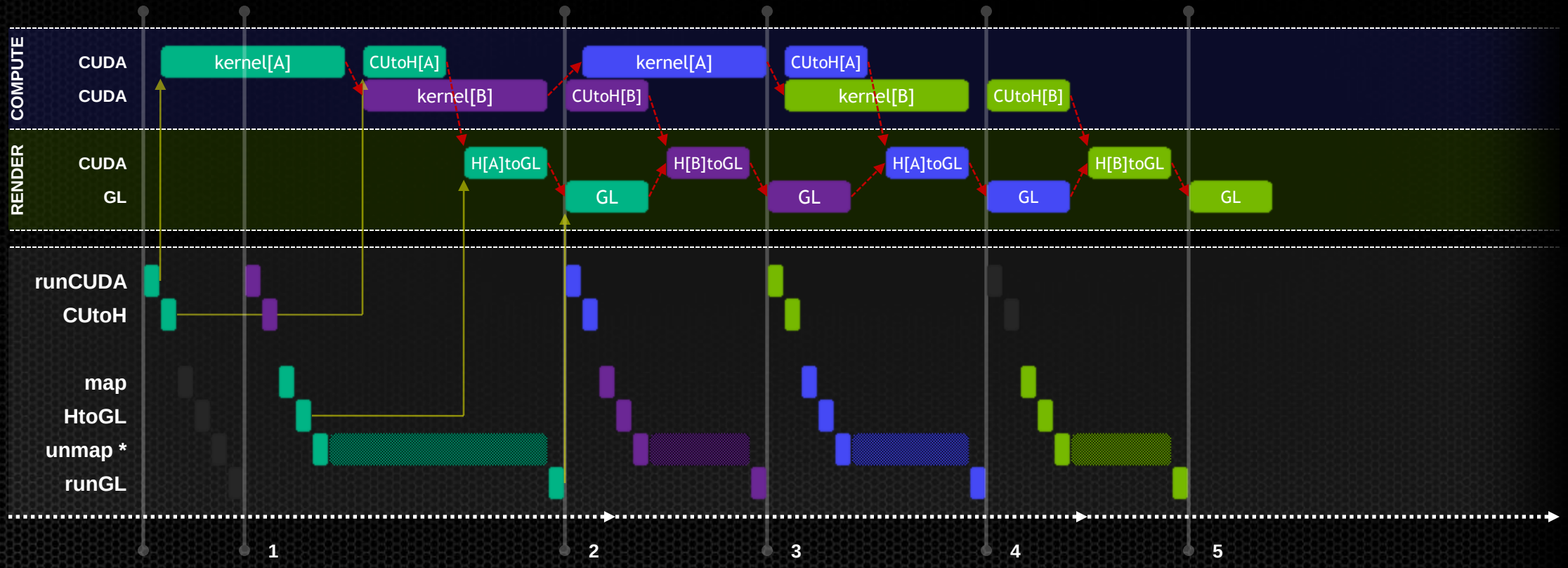


Code Sample - Manual Interop (safe Async)

```
while (!done) {
    // Compute the data in a temp buffer, and copy to a host buffer...
    runCUDA(d_data, 0);
    cudaStreamWaitEvent(0, uploadFinished, 0);
    cudaMemcpyAsync(h_data, d_data, vboSize, cudaMemcpyDeviceToHost, 0);
    cudaStreamSynchronize(0);

    // Map the render-GPU's resource and upload the host buffer...
    // (all commands must be asynchronous.)
    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, 0);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaMemcpyAsync(vboPtr, h_data, size, cudaMemcpyHostToDevice, 0);
    cudaGraphicsUnmapResources(1, &vboRes, 0);
    cudaEventRecord(uploadFinished, 0);
    cudaSetDevice(computeGPU);

    runGL(vboId);
}
```

Timeline: Multi-GPU

- Manual-interop, WRITE-DISCARD, flipping CUDA

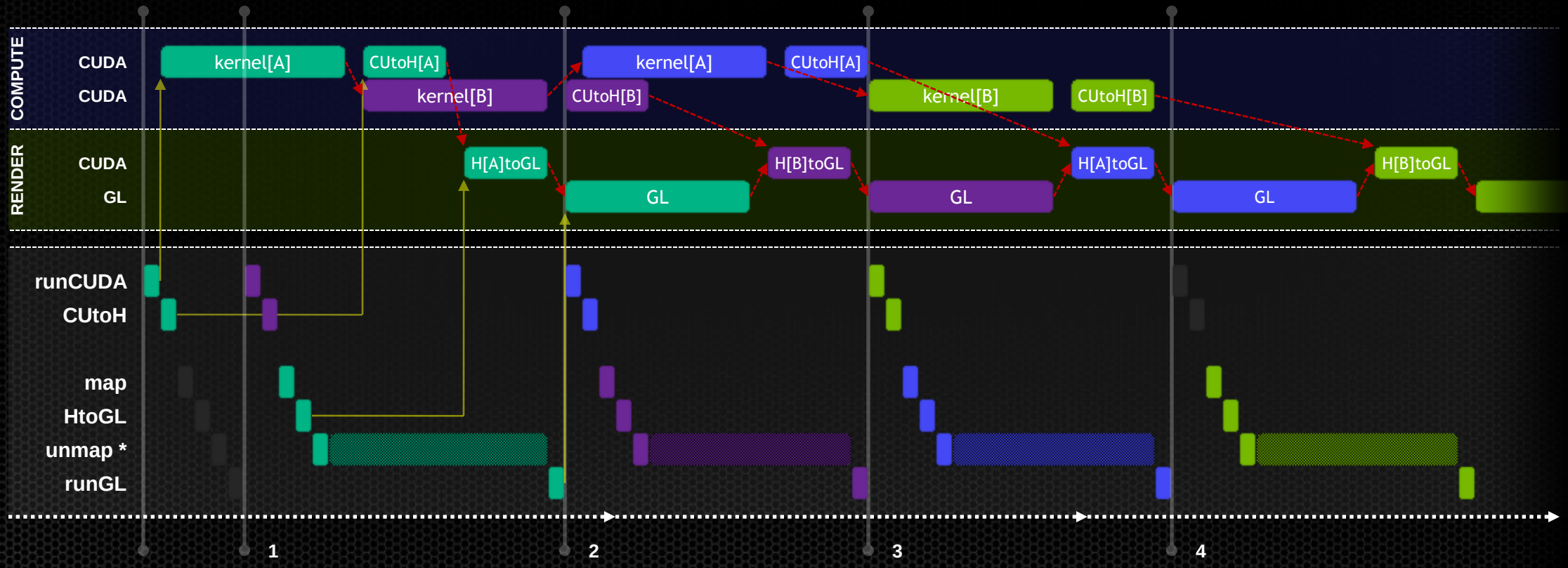


Code Sample - Manual Interop (flipping)

```
int read = 1, write = 0;

while (!done) {
    // Compute the data in a temp buffer, and copy to a host buffer...
    cudaStreamWaitEvent(custream[write], kernelFinished[read]);
    runCUDA(d_data[write], custream[write]);
    cudaEventRecord(kernelFinished[write], custream[write]);
    cudaStreamWaitEvent(custream[write], uploadFinished[read]);
    cudaMemcpyAsync(h_data[write], d_data[write], vboSize, cudaMemcpyDeviceToHost, custream[write]);
    cudaEventRecord(downloadFinished[write], custream[write]);

    // Map the renderGPU's resource and upload the host buffer...
    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, glstream);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaStreamWaitEvent(glstream, downloadFinished[read]);
    cudaMemcpyAsync(vboPtr, h_data[read], size, cudaMemcpyHostToDevice, glstream);
    cudaGraphicsUnmapResources(1, &vboRes, glstream);
    cudaEventRecord(uploadFinished[read], glstream);
    cudaStreamSynchronize(glstream); // Sync for easier analysis!
    cudaSetDevice(computeGPU);

    runGL(vboId);
    swap(&read, &write);
}
```

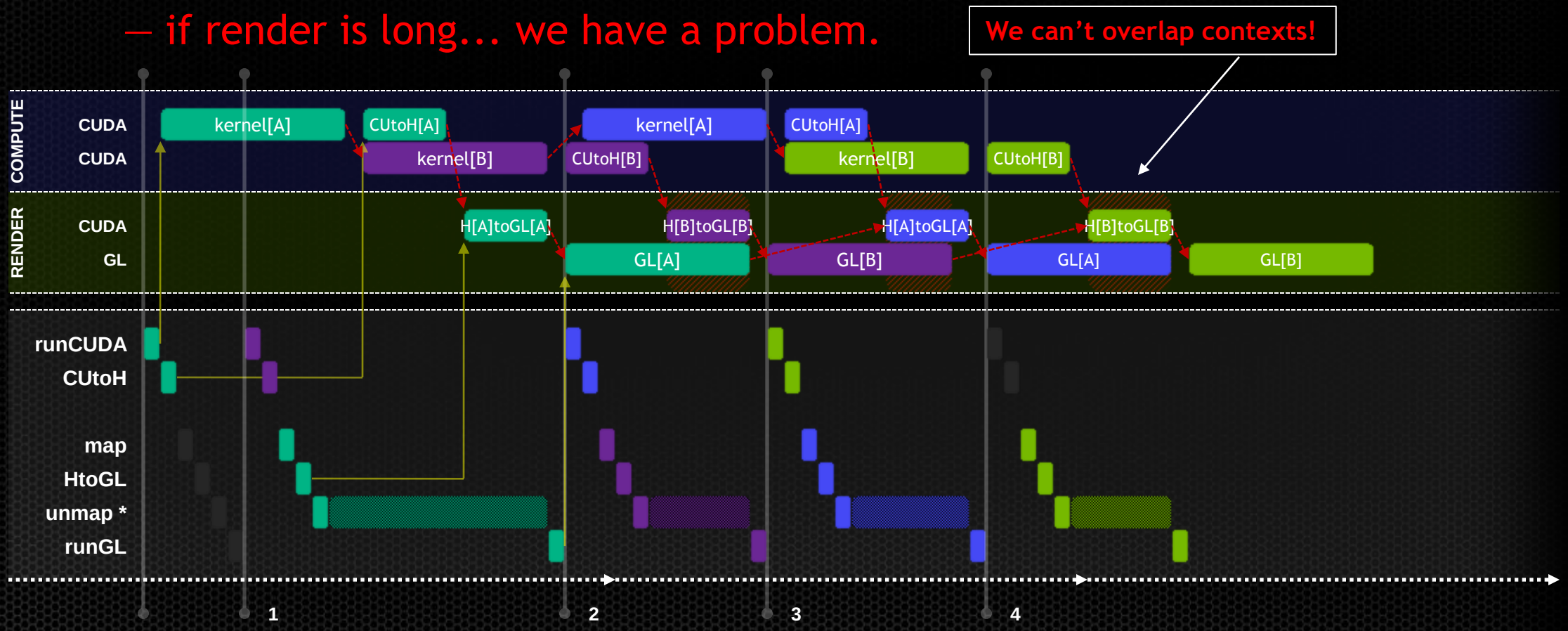
Timeline: Multi-GPU

- Manual-interop, WRITE-DISCARD, flipping CUDA
  - if render is long...



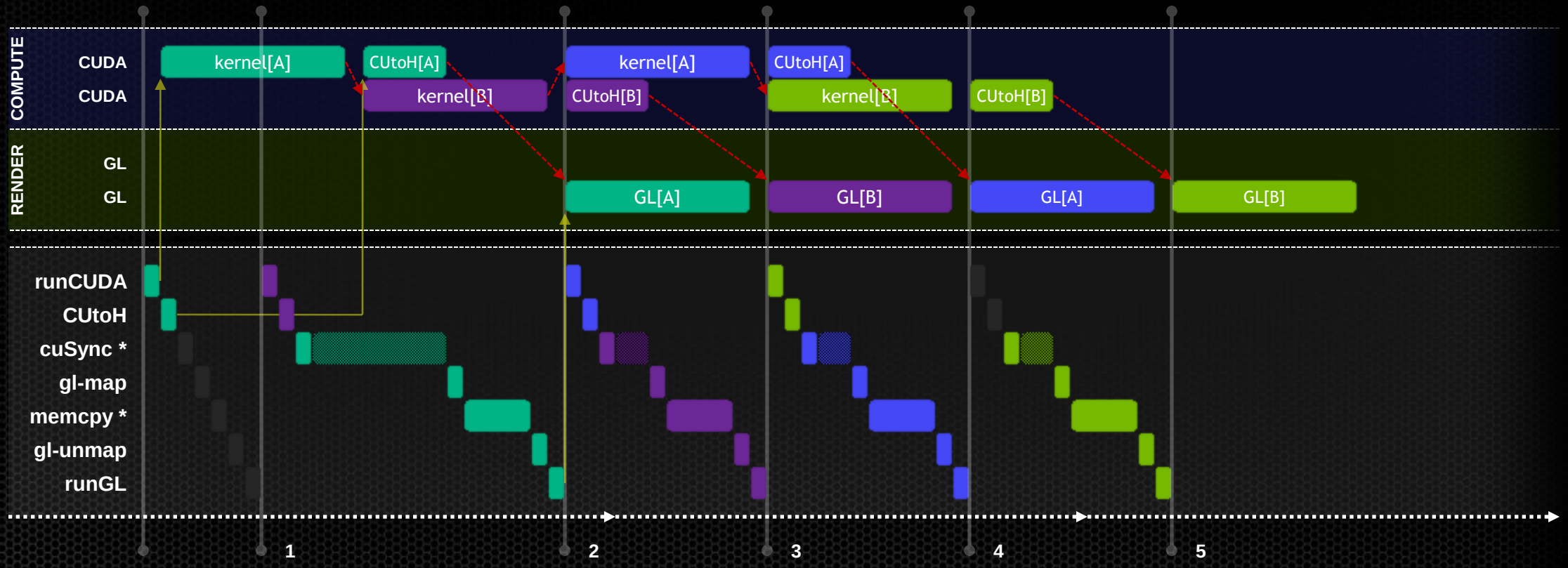
Timeline: Multi-GPU

- Manual-interop, WRITE-DISCARD, flipping CUDA & GL
  - if render is long... we have a problem.

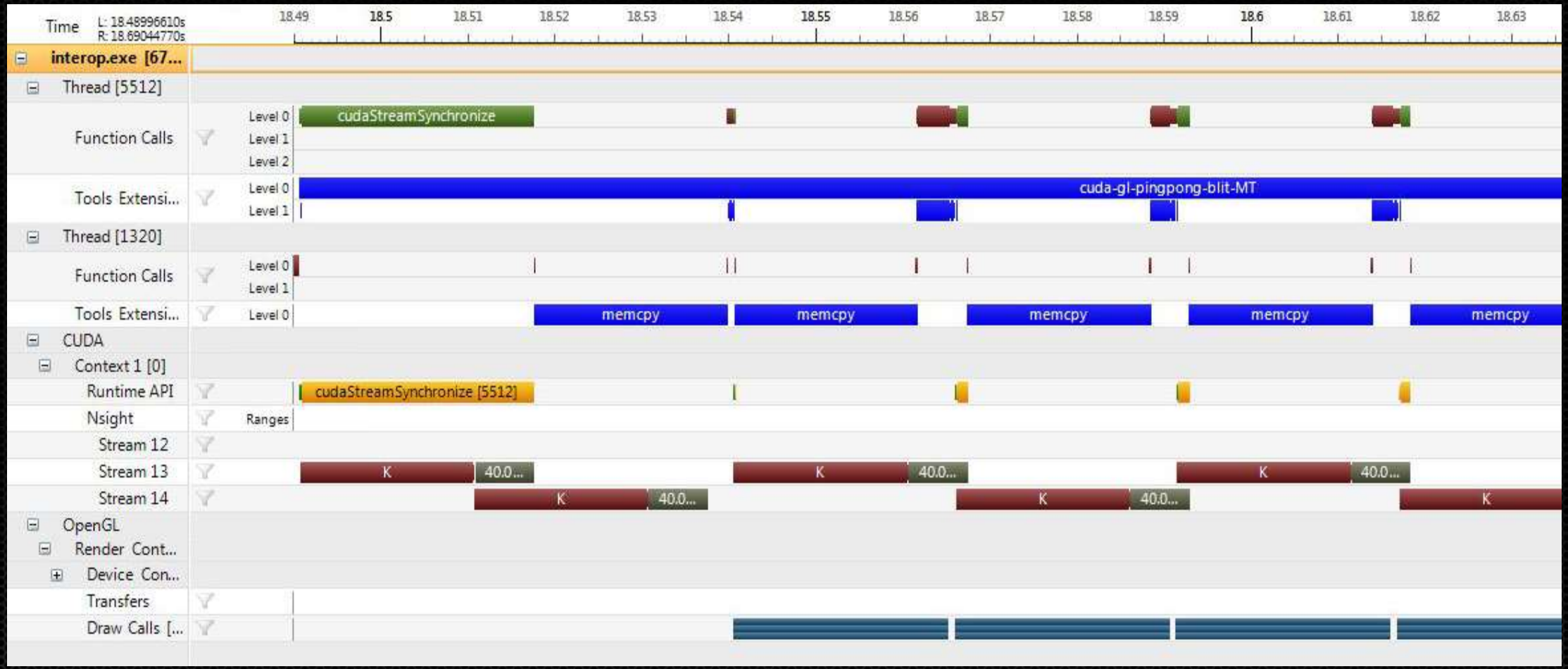


Timeline: Multi-GPU

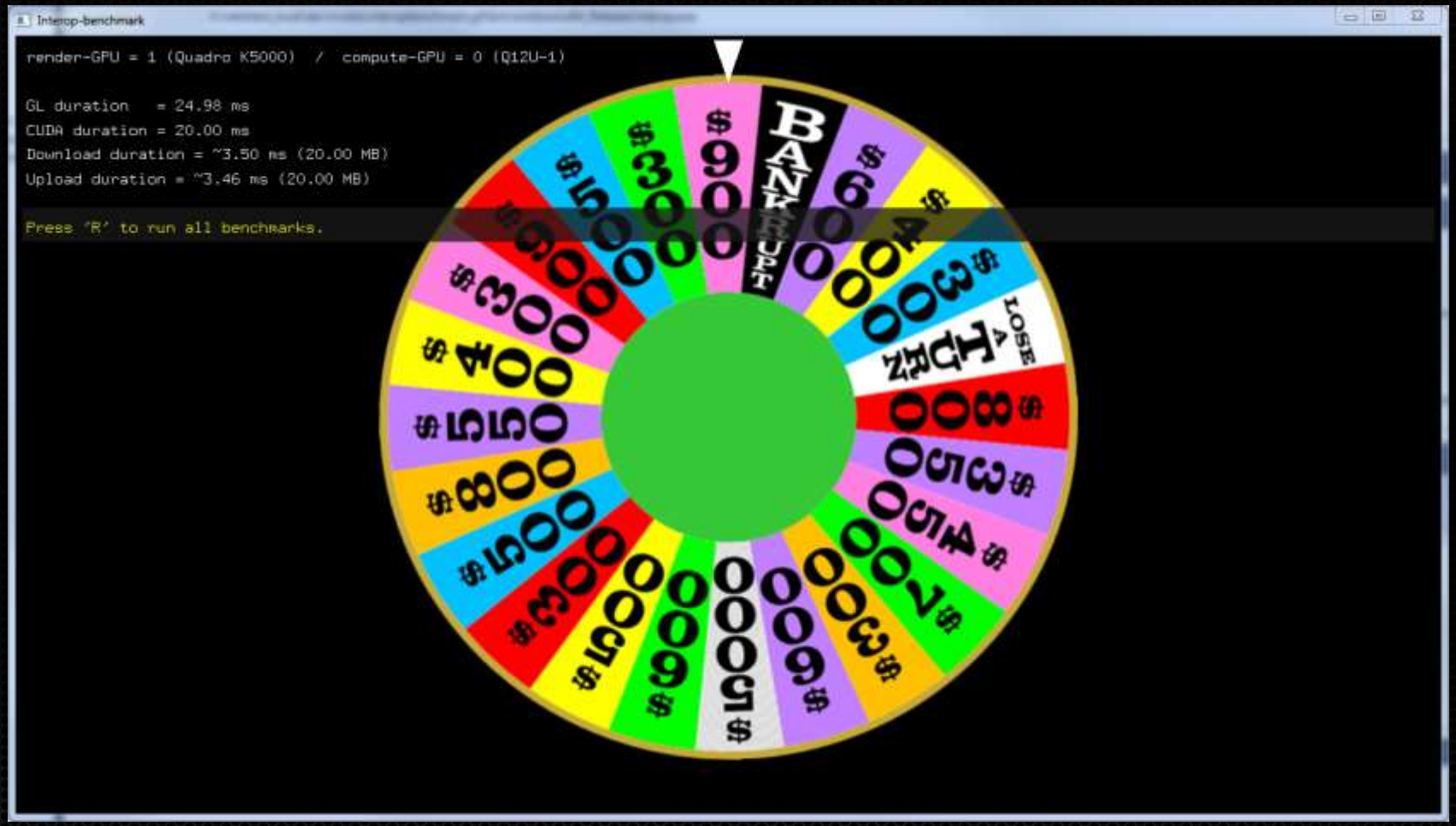
- Manual-interop, WRITE-DISCARD, flipping CUDA & GL
  - Use the GL context for upload.



Visual Studio Nsight



Demo


Interop-benchmark
render-GPU = 1 (Quadro K5000) / compute-GPU = 0 (Q12U-1)
GL duration = 24.98 ms
CUDA duration = 20.00 ms
Download duration = ~3.50 ms (20.00 MB)
Upload duration = ~3.46 ms (20.00 MB)
Press 'R' to run all benchmarks.
BANKRUPT
LOSE A TURN

Demo - results

- runCUDA (20ms)
- runGL (10ms)
- copy (10ms)

- Single-GPU
  - Driver-interop = 30ms
- Multi-GPU
  - Driver-interop = 36ms
  - Async Manual-interop = 32ms
  - Flipped Manual-interop = 22ms

Too large data size makes multi-GPU interop worse.

Overlapping the download helps us break even.

But using streams and flipping is a significant win!

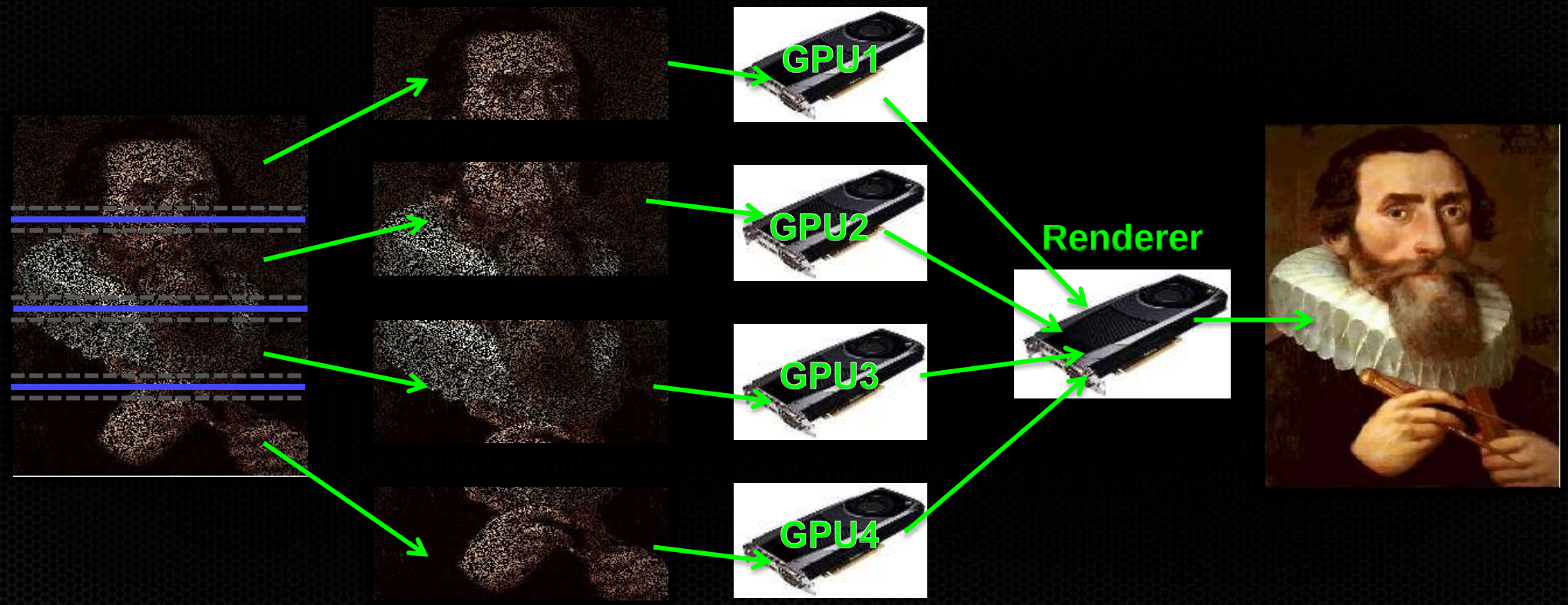
Some final thoughts on Interoperability.

- Similar considerations are applicable when OpenGL is the producer and CUDA is the consumer.
- Use `cudaGraphicsMapFlagsReadOnly`
- Avoid synchronized GPUs for CUDA.
  - Watch out for Windows's WDDM implicit synchronization on unmap!
- CUDA-OpenGL interoperability can perform slower if OpenGL context spans multiple GPUs.
- Context switch performance varies with system config and OS.

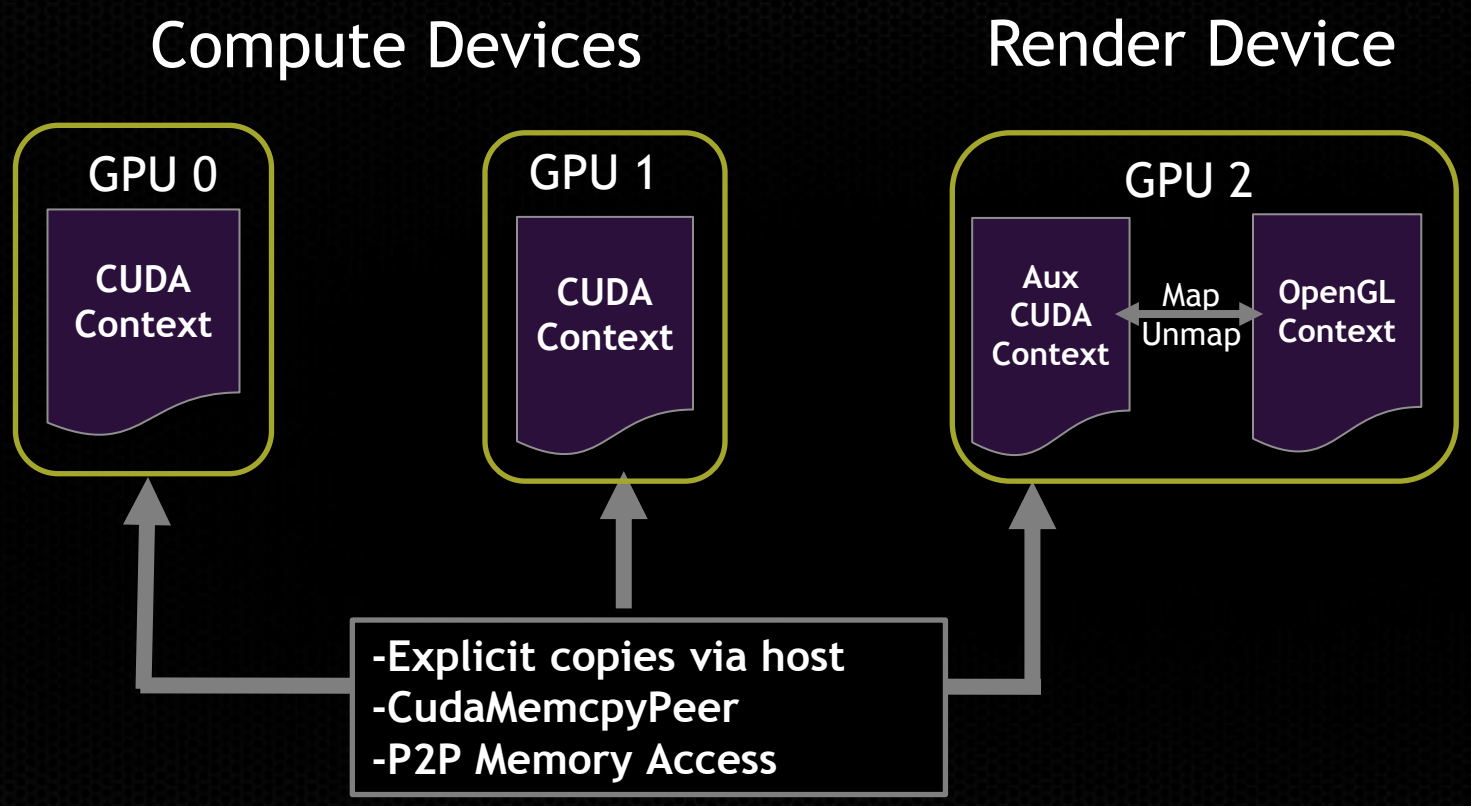
Scaling Beyond Single Compute+Graphics

- Scaling compute
  - Divide tasks across multiple devices
  - When data does not fit into single GPU memory - Distribute data
- Scaling graphics
  - Multi-displays, Stereo
  - More complex rendering e.g. raytracing
- Higher compute density
  - Amortize host or server costs eg CPU, memory, RAID shared with multiple GPUs

Multi-GPU Image Processing - SagivTech

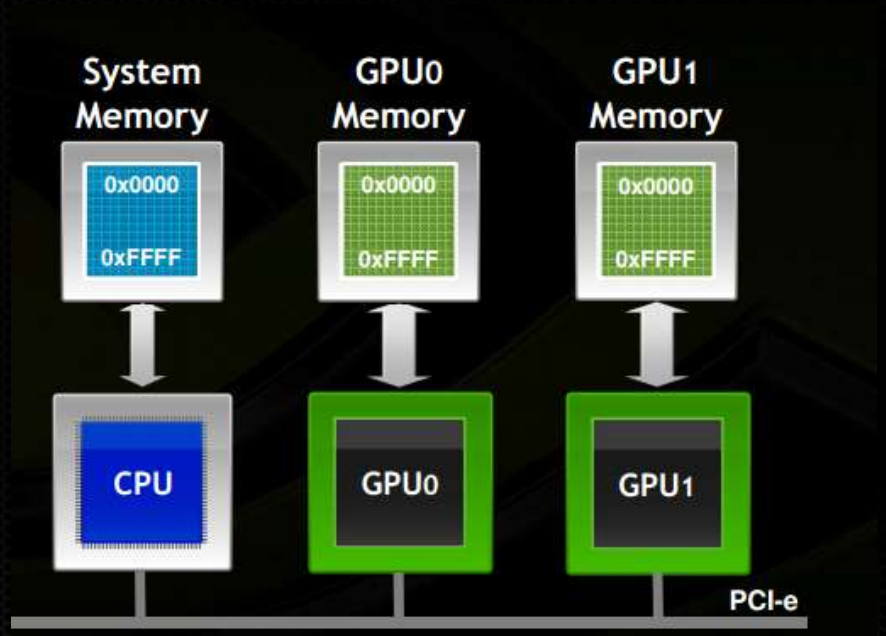


Multiple Compute : Single Render GPU

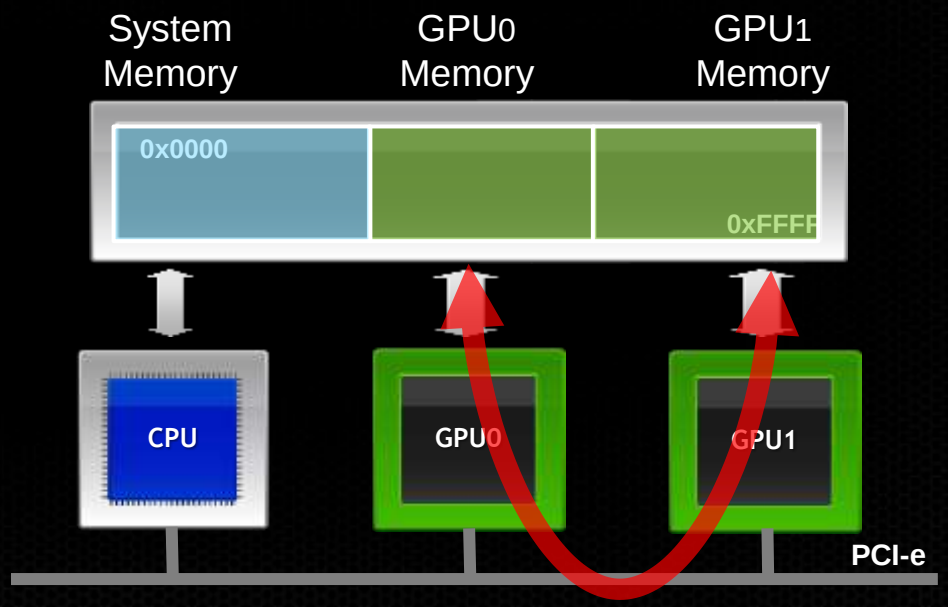


Unified Virtual Addressing (UVA)

- Easier programming with a single address space:

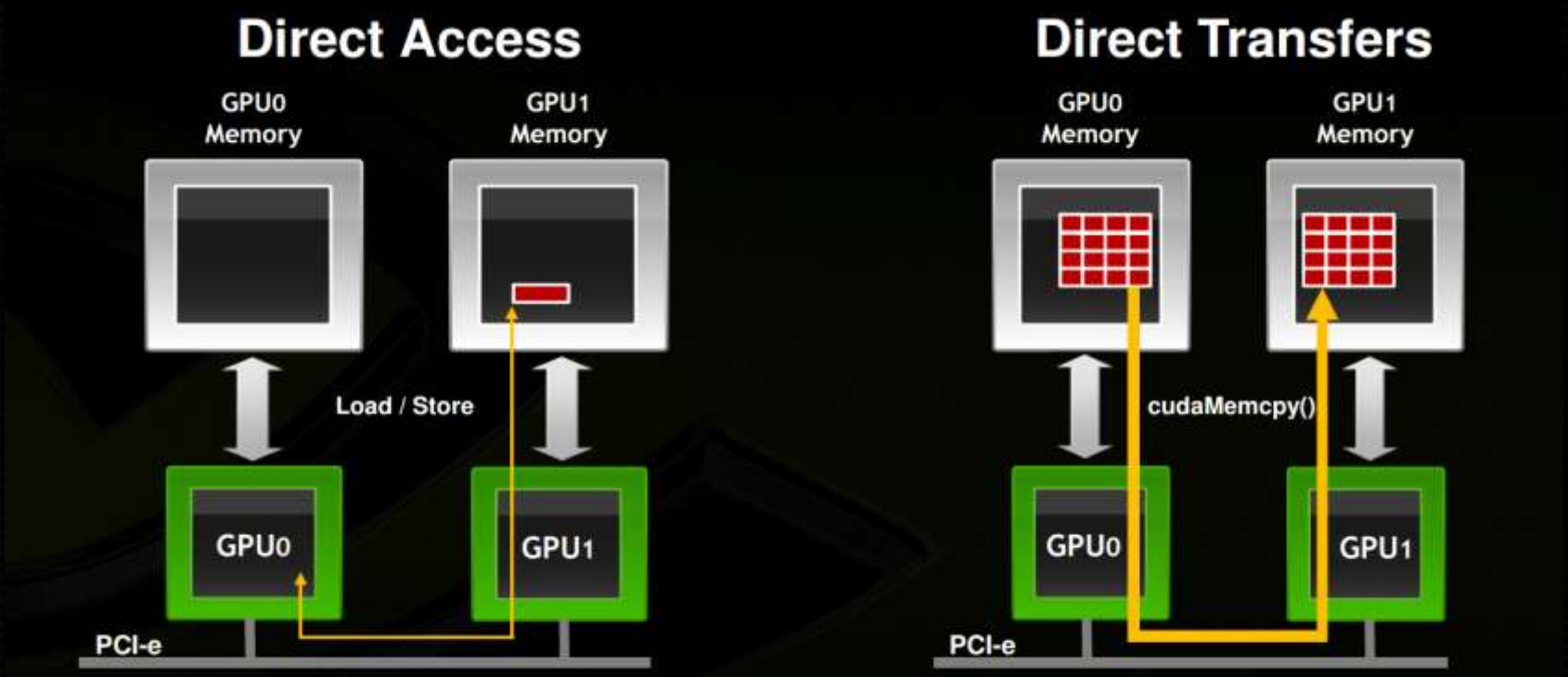


No UVA – Multiple Memory Space



UVA – Single Address Space

Peer-to-Peer (P2P) Communication

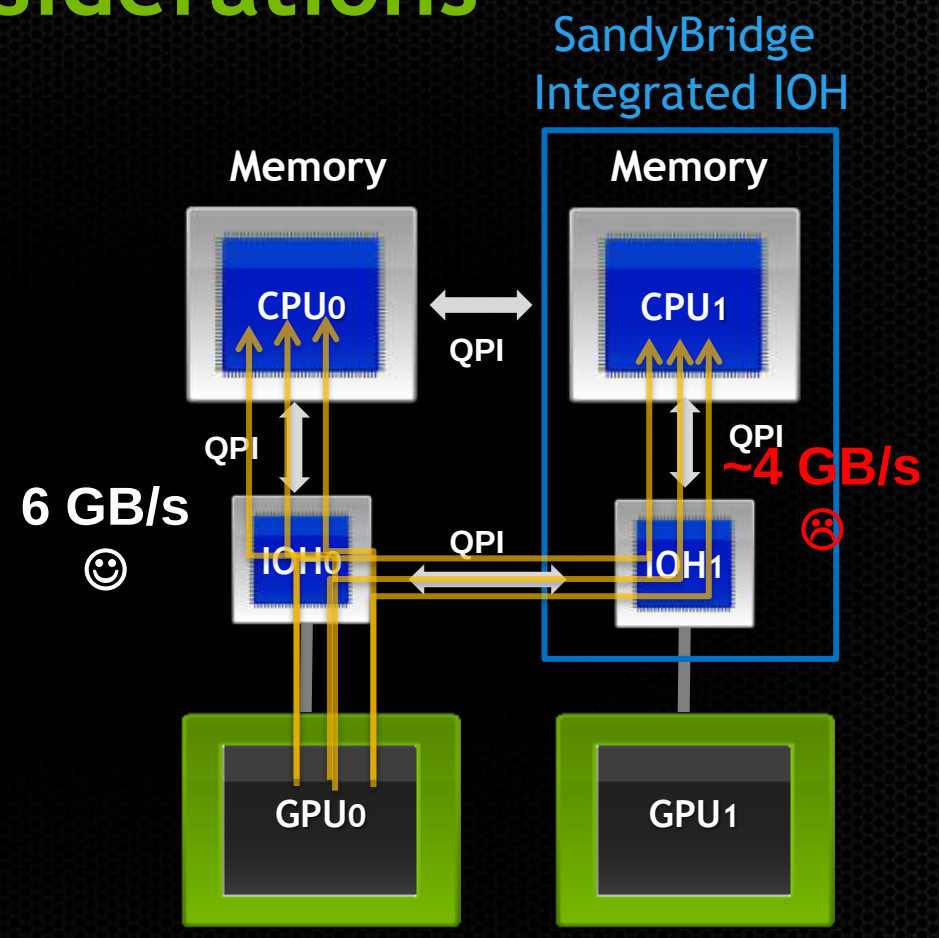


Eliminates system memory allocation & copy overhead

More convenient multi-GPU programming

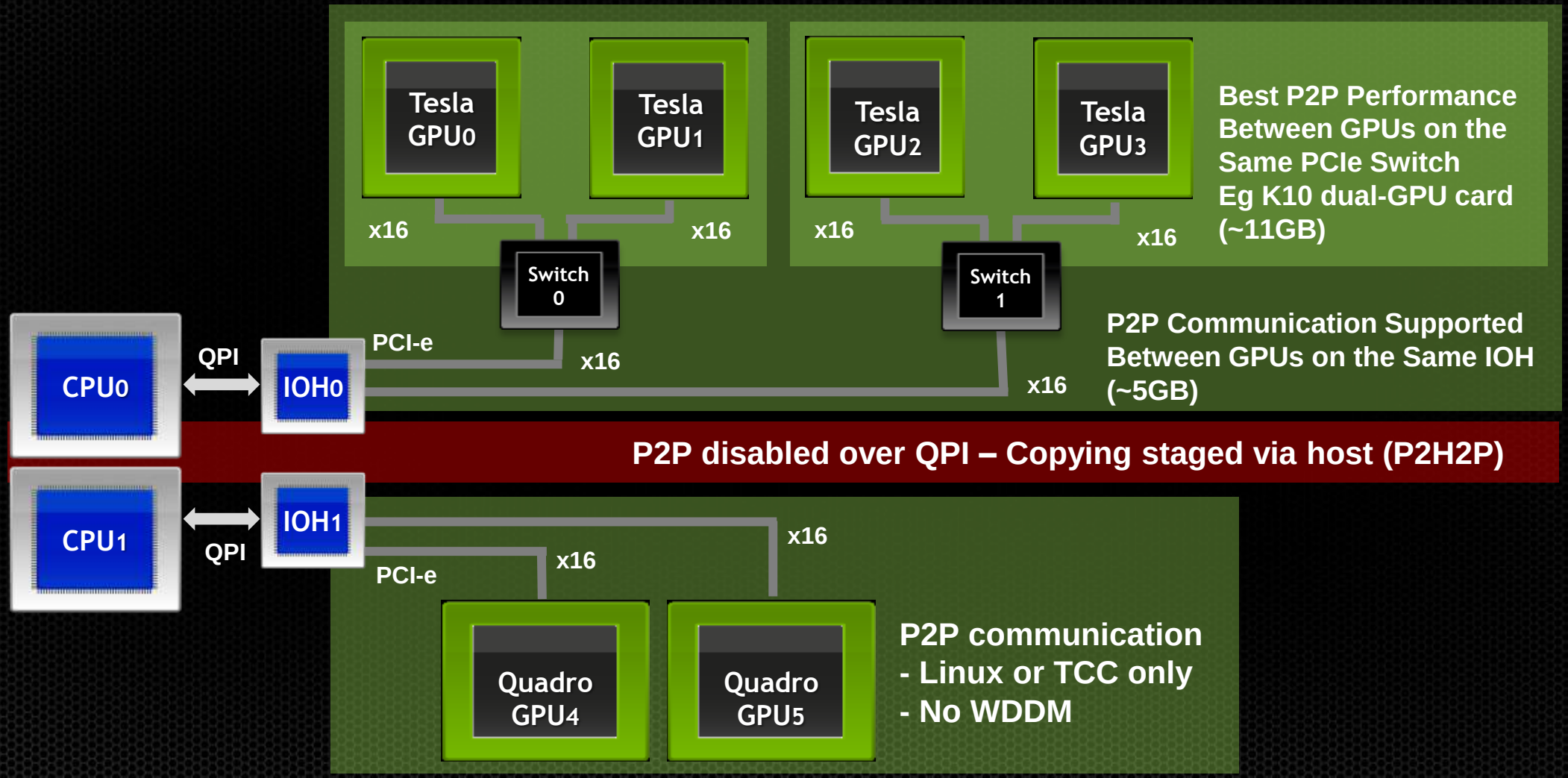
NUMA/Topology Considerations

- Memory access is non-uniform.
  - Local GPU access is faster than remote (extra QPI hop)
  - This affects PCIe transfer throughput.
- NUMA APIs
  - Thread affinity considerations
- Pitfalls of set process affinity
  - Does not work with graphics APIs



Dale Southard. Designing and Managing GPU Clusters. Supercomputing 2011
http://nvidia.fullviewmedia.com/fb/nv-sc11/tabscontent/archive/401-thu-southard.html

Peer-to-Peer Configurations

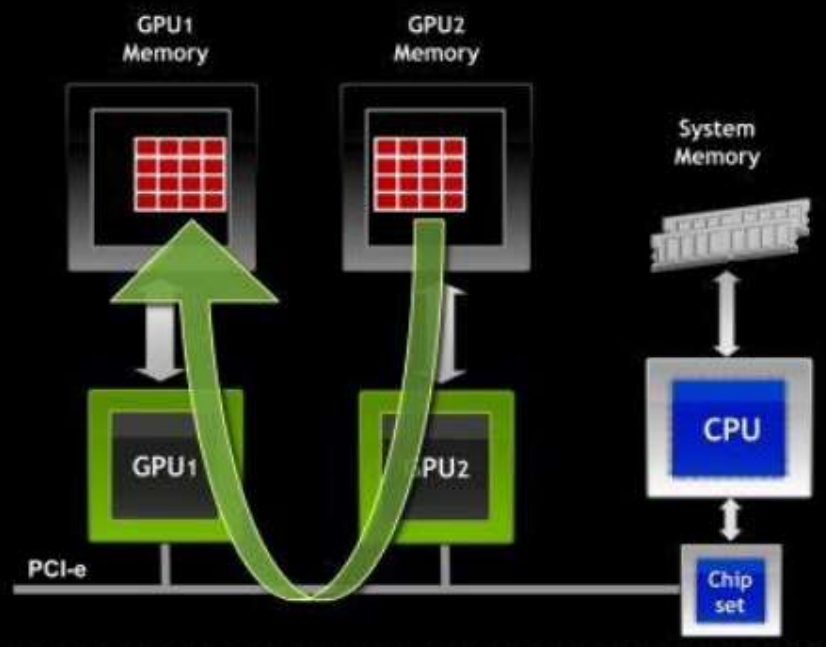


Peer-to-Peer Initialization

- cudaDeviceCanAccessPeer(&isAccessible, srcGPU, dstGPU)
  - Returns in 1st arg if srcGPU can access memory of dstGPU
  - Need to do this bidirectionally
- cudaDeviceEnablePeerAccess(peerDevice, 0)
  - Enables current GPU to access peerDevice
  - Note that this is asymmetric!
- cudaDeviceDisablePeerAccess
  - P2P can be limited to a specific phase in order to reduce overhead and free resources

Peer-to-Peer Copy

- cudaMemcpyPeerAsync
  - Will fall back to staging copies via host for unsupported configurations or if peer-access is disabled.
- cudaMemcpyAsync
  - Using UVA, regular memory copies will also work.



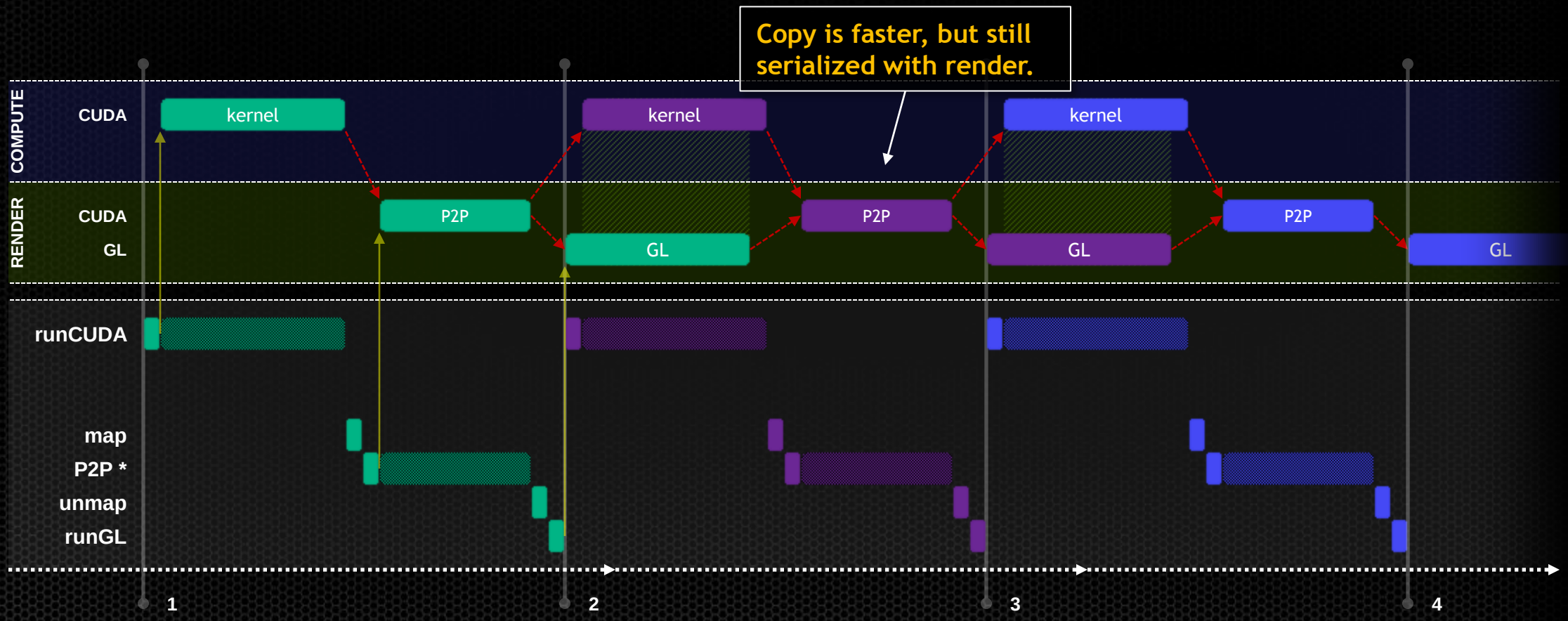
NVIDIA CUDA Webinars - Multi-GPU Programming
http://developer.download.nvidia.com/CUDA/training/cuda_webinars_multi_gpu.pdf

Code Sample - Manual Interop (P2P)

```
cudaMalloc((void**)&d_data, vboSize);

while (!done) {
    // Compute data in temp buffer, and copy to host...
    runCUDA(d_data, 0);
    cudaStreamSynchronize(0);

    // Map the render-GPU's resource and upload the host buffer...

    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, glstream);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaMemcpyPeerAsync(vboPtr, renderGPU, d_data, computeGPU, size, glstream);
    cudaGraphicsUnmapResources(1, &vboRes, glstream);
    cudaStreamSynchronize(glstream);

    cudaSetDevice(computeGPU);

    runGL(vboId);
}
```

Timeline: Multi-GPU

- Manual-interop, synchronous*, P2P, WRITE-DISCARD

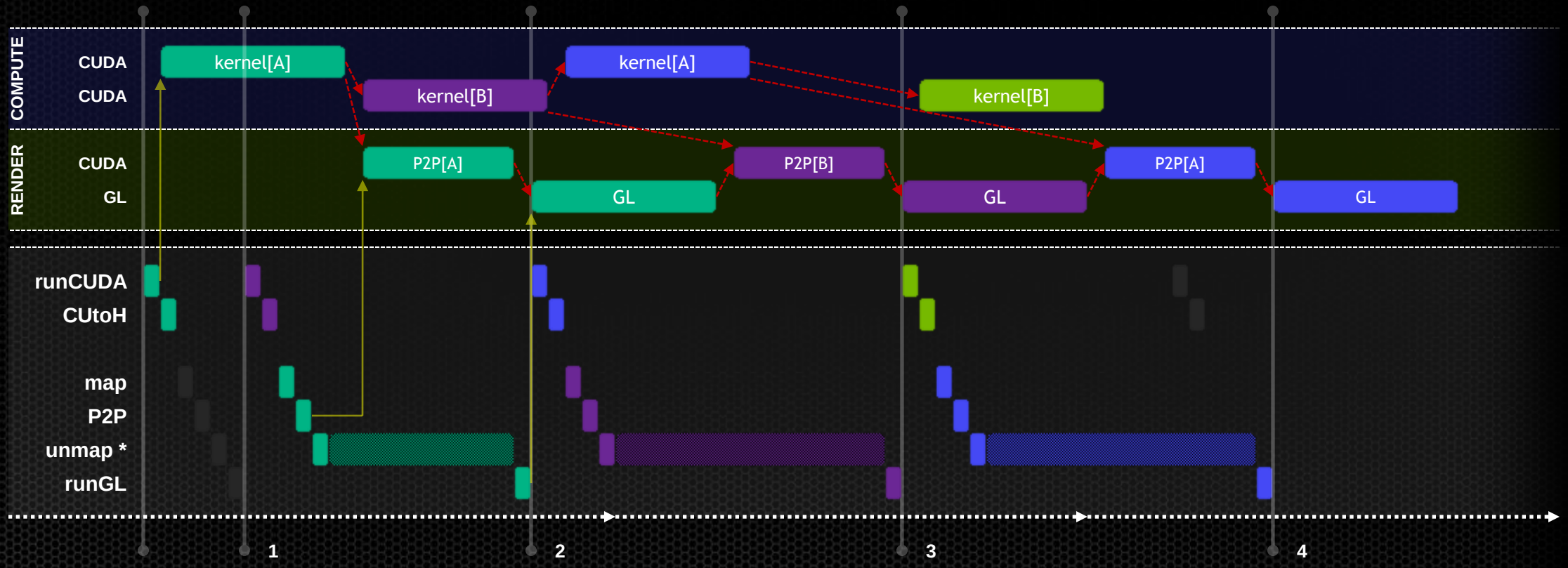


Code Sample - Manual Interop (P2P flipping)

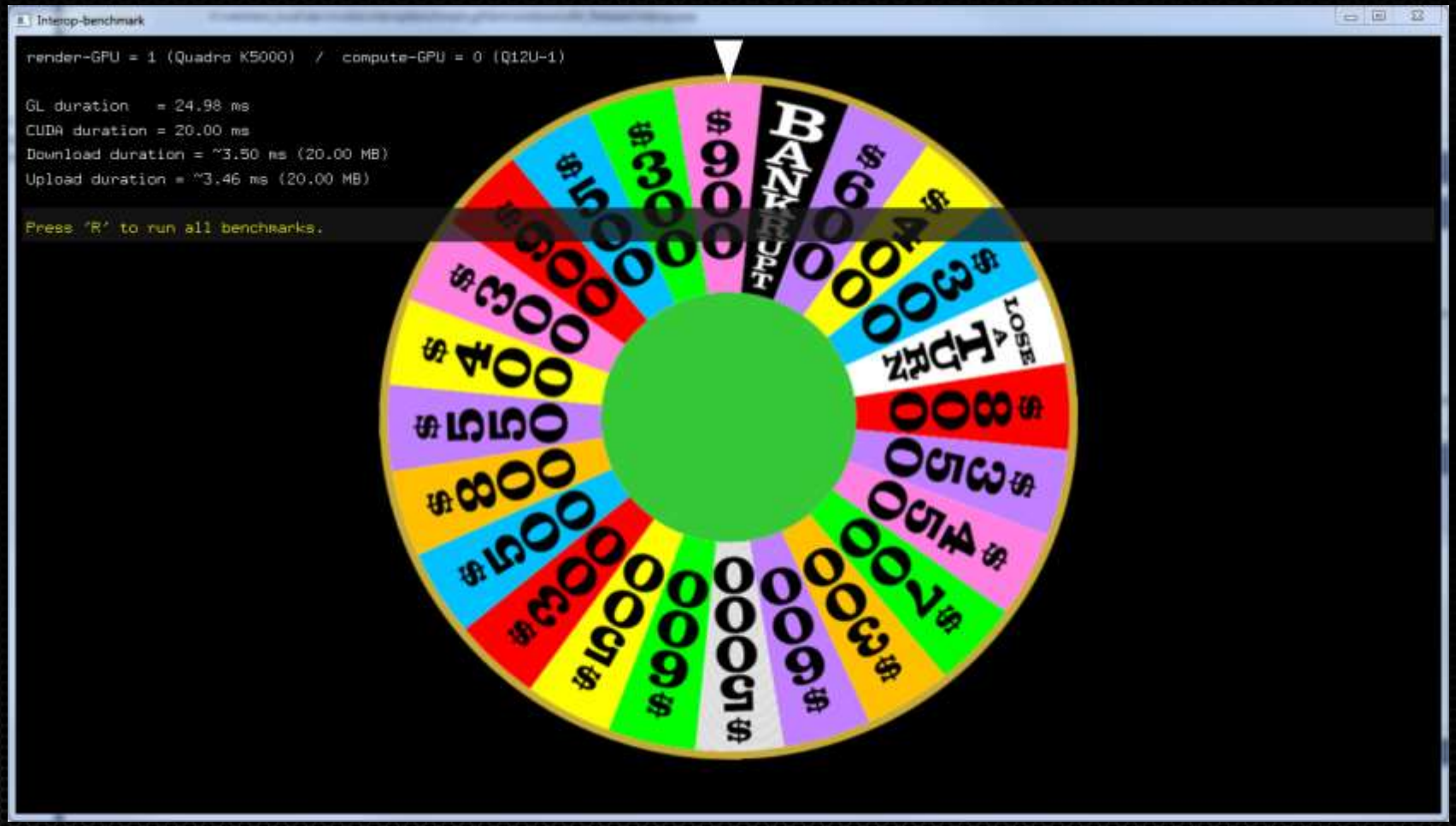
```
int read = 1, write = 0;

while (!done) {
    // Compute the data in a temp buffer, and copy to a host buffer...
    cudaStreamWaitEvent(custream[write], peerFinished[read]);
    runCUDA(d_data[write], custream[write]);
    cudaEventRecord(kernelFinished[write], custream[write]);

    // Map the renderGPU's resource and p2p copy the compute buffer...
    cudaSetDevice(renderGPU);
    cudaGraphicsMapResources(1, &vboRes, glstream);
    cudaGraphicsResourceGetMappedPointer((void**)&vboPtr, &size, vboRes);
    cudaStreamWaitEvent(glstream, kernelFinished[read]);
    cudaMemcpyPeerAsync(vboPtr, renderGPU, d_data[read], computeGPU, size, glstream);
    cudaGraphicsUnmapResources(1, &vboRes, glstream);
    cudaEventRecord(peerFinished[read], glstream);
    cudaStreamSynchronize(glstream); // Sync for easier analysis!
    cudaSetDevice(computeGPU);

    runGL(vboId);
    swap(&read, &write);
}
```

Timeline: Multi-GPU

- Manual-interop, synchronous*, P2P flipping, WRITE-DISCARD



Demo

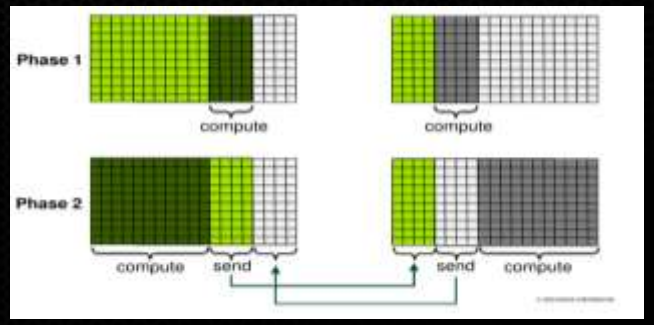

Interop-benchmark
render-GPU = 1 (Quadro K5000) / compute-GPU = 0 (Q12U-1)
GL duration = 24.98 ms
CUDA duration = 20.00 ms
Download duration = ~3.50 ms (20.00 MB)
Upload duration = ~3.46 ms (20.00 MB)
Press 'R' to run all benchmarks.
BANKRUPT
LOSE A TURN

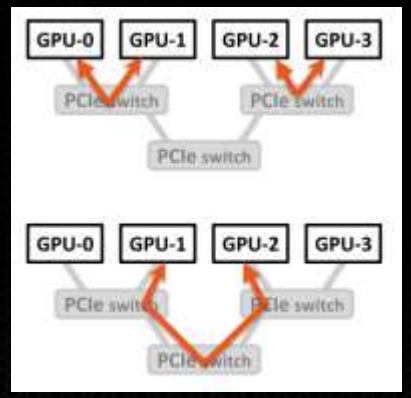
Peer-to-Peer Memory Access

- Sometimes we don't want to explicitly copy but want access to entire space on all GPUs and CPU
- Already possible with linear memory with UVA, recent texture addition useful for graphics.
- Example – large dataset that can't fit into 1 GPU memory
  - Distribute the domain/data across GPUs
  - Each GPU now need to access the other GPU's data for halos/boundary exchange

Mapping Algorithms to Hardware Topology

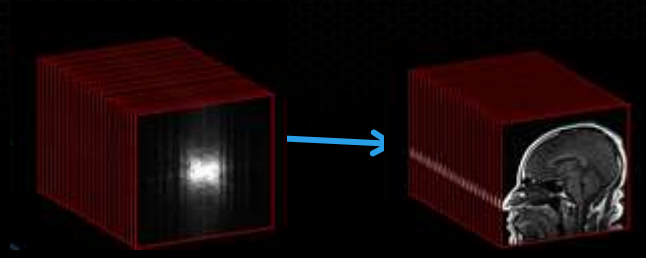
GPU0 GPU1

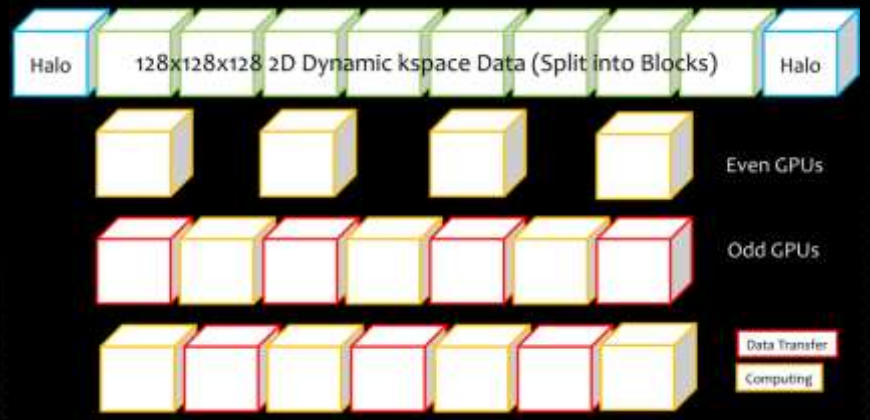




Paulius M. S0515 – Implementing 3D Finite Difference Code on GPUs, GTC 2009.

Data reduction, Multli-phase problems





Won-Ki Jeong. S3308 - Fast Compressive Sensing MRI Reconstruction on a Multi-GPU System, GTC 2013.

Sharing texture across GPUs

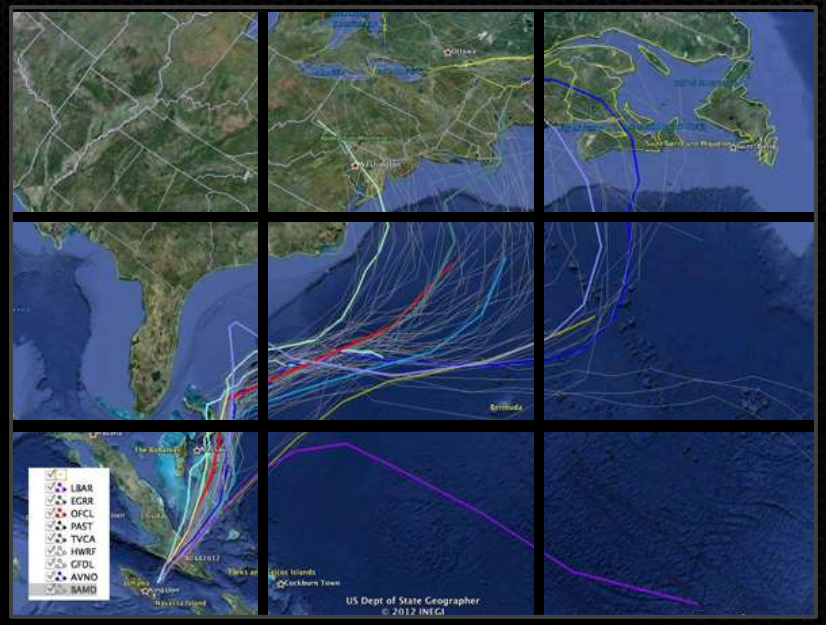
- Kernel runs on device 0 accesses texture from device 1

```
cudaSetDevice(0);
cudaMalloc3DArray(d0_volume, ...);
cudaSetDevice(1);
cudaMalloc3DArray(d1_volume, ...);
while (!done) {
    // Set CUDA device to COMPUTE DEVICE 0
    cudaSetDevice(0);
    cudaBindTextureToArray(tex0, d0_volume);
    cudaBindTextureToArray(tex1, d1_volume);
    myKernel <<<...>>>
}
```

```
__global__ void myKernel(...)
{
    float voxel0 = tex3D(tex0, u, v, w); //accesses gpu0
    float voxel1 = tex3D(tex1, u, v, w); //accesses gpu1 through p2p
}
```

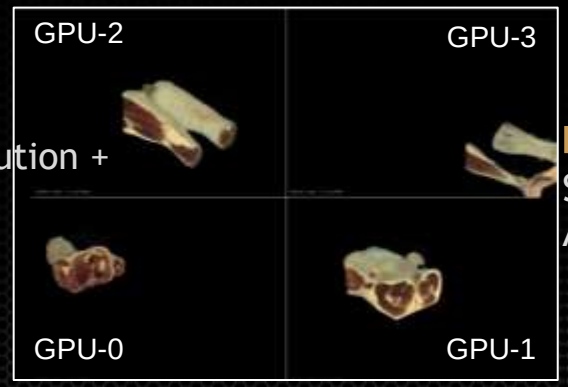
Scaling Graphics - Multiple Quadro GPUs

- Virtualized SLI case
  - Multi-GPU mosaic drive a large wall
- Access each GPU separately
  - Parallel rendering
  - Multi view-frustum eg Stereo, CAVEs



Hurricane Sandy Simulation showing multiple computed tracks for different input params - Image Credits : NOAA

Data Distribution + Render



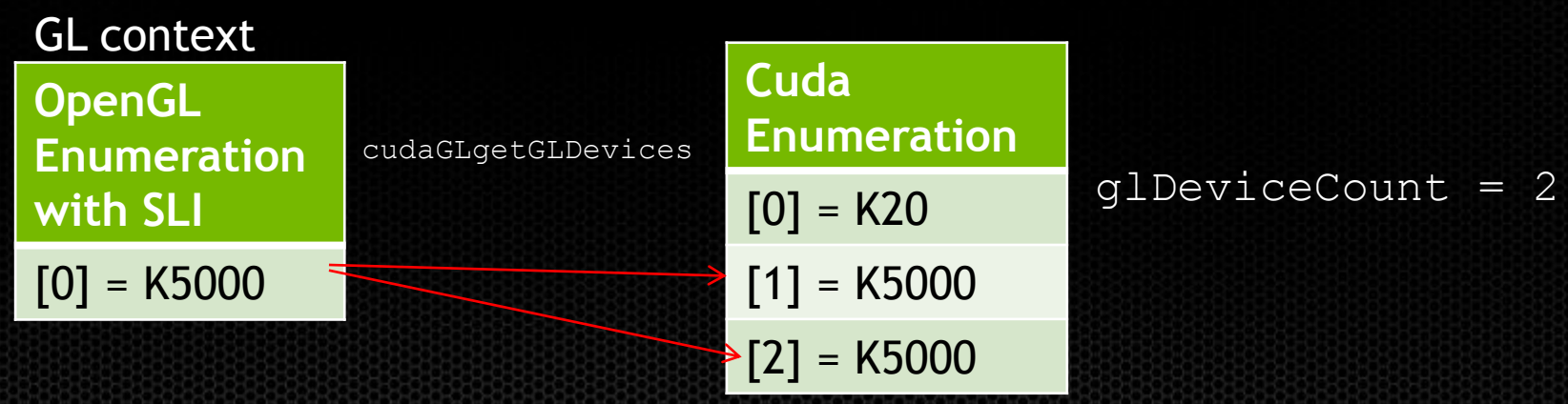
Sort + Alpha Composite

Visible Human 14GB Texture Data

CUDA With SLI Mosaic

- Use driver interop with the current cuda device
  - Driver handles the optimized data transfer (Work in progress)
- The OpenGL context spans 2 cuda devices

```
int glDeviceIndices[MAX_GPU];
int glDeviceCount;
cudaGLGetDevices(&glDeviceCount, glDeviceIndices, glDeviceCount,
cudaGLDeviceListAll));
```



Specifying Render GPU Explicitly

- Linux
  - Specify separate X screens using XOpenDisplay

```
Display* dpy = XOpenDisplay(":0."+gpu)
GLXContext = glxCreateContextAttribs(dpy,…);
```

- Windows
  - Using NV_GPU_AFFINITY extension

```
BOOL wglEnumGpusNV(UINT iGpuIndex, HGPUNV *phGPU)
```

For #GPUs enumerated {

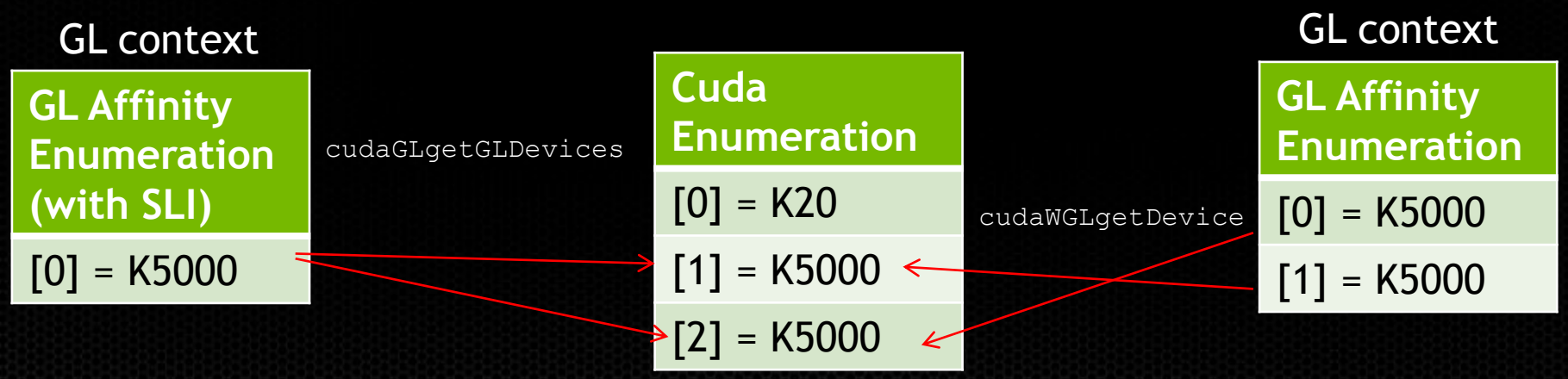
```
        GpuMask[0]=hGPU[0];
        GpuMask[1]=NULL;
        //Get affinity DC based on GPU
        HDC affinityDC = wglCreateAffinityDCNV(GpuMask);
        setPixelFormat(affinityDC);
        HGLRC affinityGLRC = wglCreateContext(affinityDC);
}
```

Mapping Cuda ↔ Affinity GL Devices

- How to map OpenGL device to CUDA

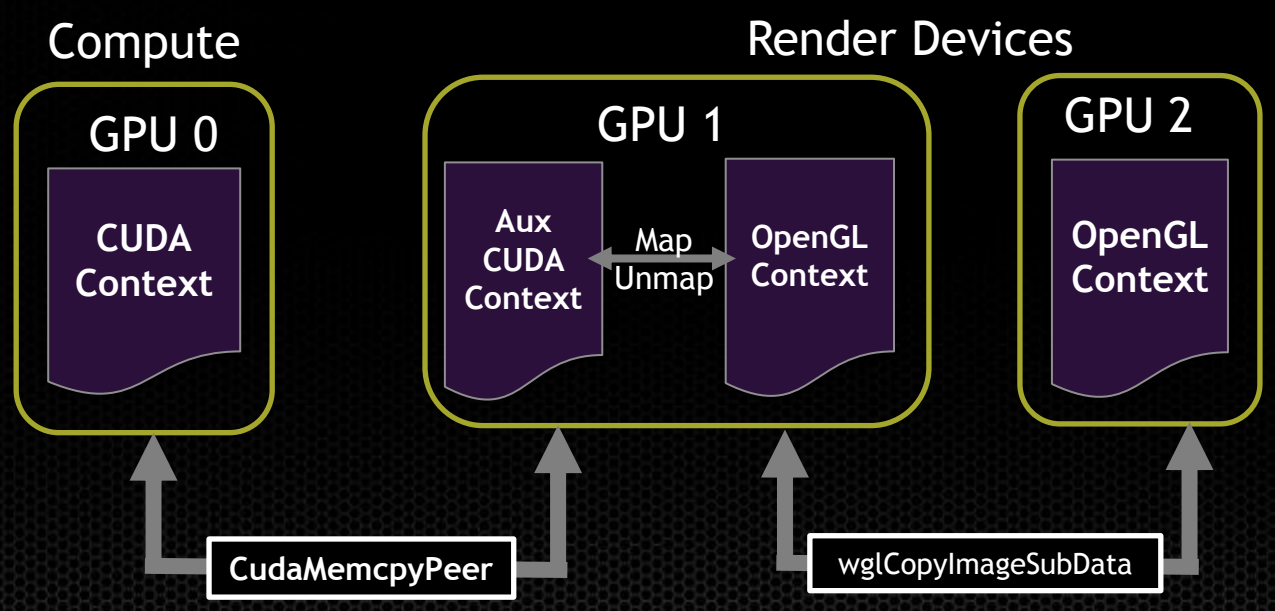
```
cudaWGLGetDevice(int *cudadevice,HGPUNV oglGPUHandle)
```

- Don't expect the same order
  - GL order is windows specific while CUDA is device specific



Compute & Multiple Render

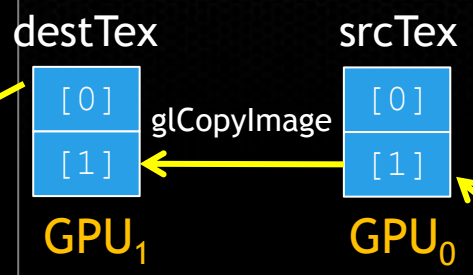
- GL_NV_COPY_IMAGE - Copy between GL devices
  - wglCopyImageSubData and glXCopyImageSubDataNV
  - Texture objects & Renderbuffers only.
  - Can copy Texture subregions.



Multithreading with OpenGL Contexts

Thread - Dest GPU_1

```
// Wait for signal to start consuming
CPUWait(producedFenceValid);
glWaitSync(producedFence[0]);

// Bind texture object
glBindTexture(destTex[0]);

// Use as needed

// Signal we are done wit this texture
consumedFence[0] = glFenceSync(…);
CPUSignal(consumedFenceValid);
```



Thread - Source GPU_0

```
// Wait for
CPUWait(consumedFenceValid);
glWaitSync(consumedFence[1]);

// Bind render target
glFramebufferTexture2D(srcTex[1]);

// Draw here…

// Unbind
glFramebufferTexture2D(0);

// Copy over to consumer GPU
wglCopyImageSubDataNV(srcCtx,srcTex[1],
        ..destCtx,destTex[1]);

// Signal that producer has completed
producedFence[1] = glFenceSync(…);
CPUSignal(producedFenceValid);
```

Multi-level CPU and GPU sync primitives

```
GLsync consumedFence[MAX_BUFFERS];
GLsync producedFence[MAX_BUFFERS];
HANDLE consumedFenceValid, producedFenceValid;
```

Shalini V. S0353 - Programming Multi-GPUs for Scalable Rendering, GTC 2012.

Some final thoughts...

- Hardware architecture influences transfer and system throughput. e.g. NUMA
- Peer-To-Peer copies/access simplify multi-gpu programming.
  - but context-switching between CUDA and GL will serialize.
- When scaling graphics
  - remember the device mapping between CUDA and OpenGL.

Conclusions & Resources

- The driver can do all the heavy-lifting but...
- Scalability and final performance is up to the developer
  - For fine-grained control and optimization, you might want to move the data manually.
- CUDA samples/documentation:
  - http://developer.nvidia.com/cuda-downloads
- OpenGL Insights, Patrick Cozzi, Christophe Riccio, 2012. ISBN 1439893764.

  www.openglinsights.com

Demo code

- win7 & linux Code for the benchmarking demo is available at:
  - http://bitbucket.org/wbraithwaite_nvidia/interopbenchmark