SIGGRAPH ASIA 2013
Shaping the Future of
Visual Computing

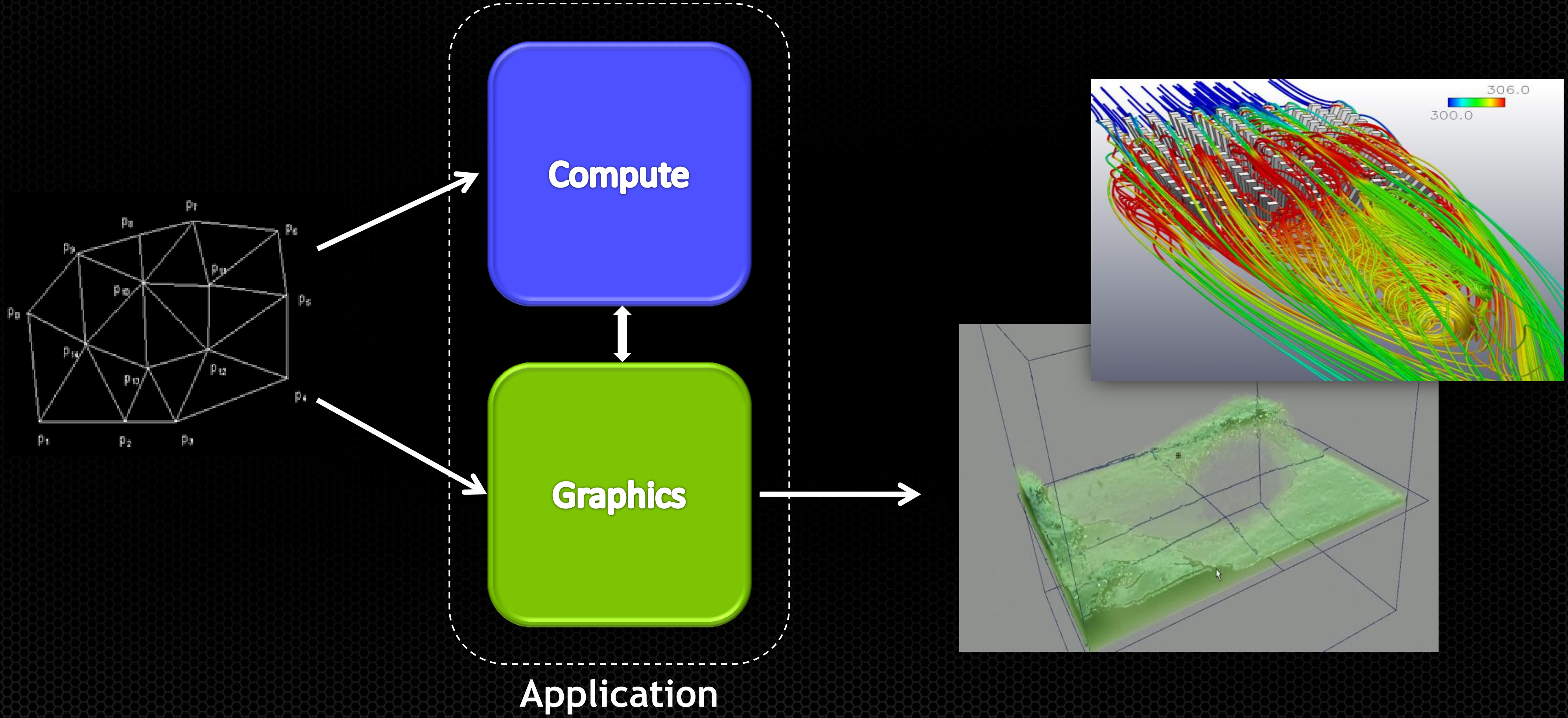
Multi-GPU Programming for Visual Computing

Wil Braithwaite - NVIDIA Applied Engineering

Talk Outline

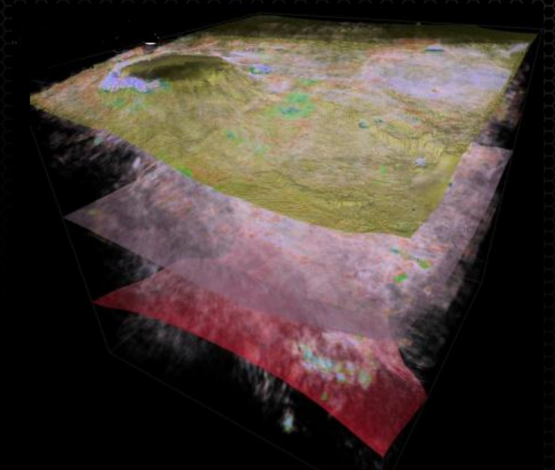
- Part 1 (Wednesday 3pm)
  - Compute and Graphics API Interoperability.
    - The basics.
    - Interoperability Methodologies & driver optimizations.
  - Interoperability at a system level.
    - fine-grained control for managing scaling
  - Demo.
- Part 2 (Thursday 12pm)
  - Peer-to-peer & UVA, NUMA considerations
  - Scaling beyond one-to-one, single-compute : single-render
    - Enumerating Graphics & Compute Resources
    - many-to-many, multiple-compute : multiple-render

Compute & Visualize the same data

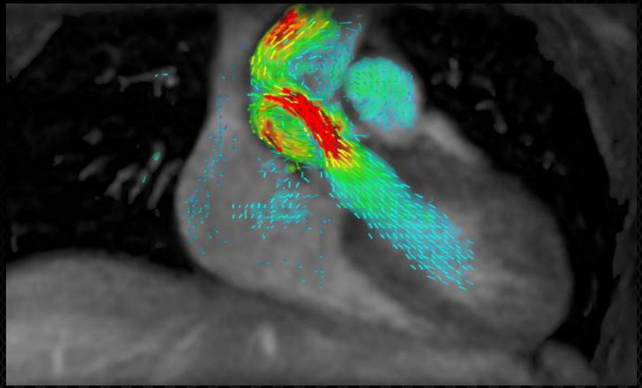


Multi-GPU Compute+Graphics Use Cases

- 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



NVIDIA Index – Seismic Interpretation



Morpheus Medical – Real-time CFD Viz

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$_0$

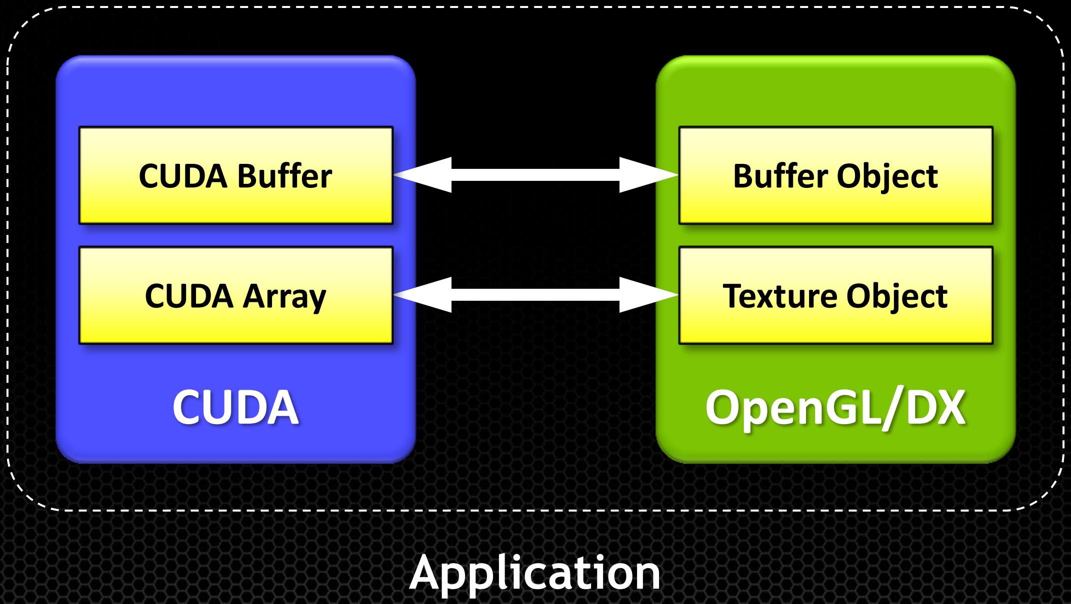
Simulate$_1$ + Visualize$_0$

Simulate$_2$ + Visualize$_1$

Simulate$_3$ + Visualize$_2$

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 image interop

- Setup and Registration of Texture Objects:

```
GLuint texId;
cudaGraphicsResource_t texRes;

// OpenGL buffer creation...
glGenTextures(1, &texId);
glBindTexture(GL_TEXTURE_2D, texId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8UI, texWidth, texHeight, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, 0);
glBindTexture(GL_TEXTURE_2D, 0);

// Registration with CUDA.
cudaGraphicsGLRegisterImage(&texRes, texId, GL_TEXTURE_2D,
                            cudaGraphicsRegisterFlagsNone);
```

Code Sample - Simple image interop

- Mapping between contexts:

```
cudaArray* texArray;

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

    cudaGraphicsSubResourceGetMappedArray(&texArray, texRes, 0, 0);
    runCUDA(texArray, 0);

    cudaGraphicsUnmapResources(1, &texRes, 0);

    runGL(texId);
}
```

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);
```

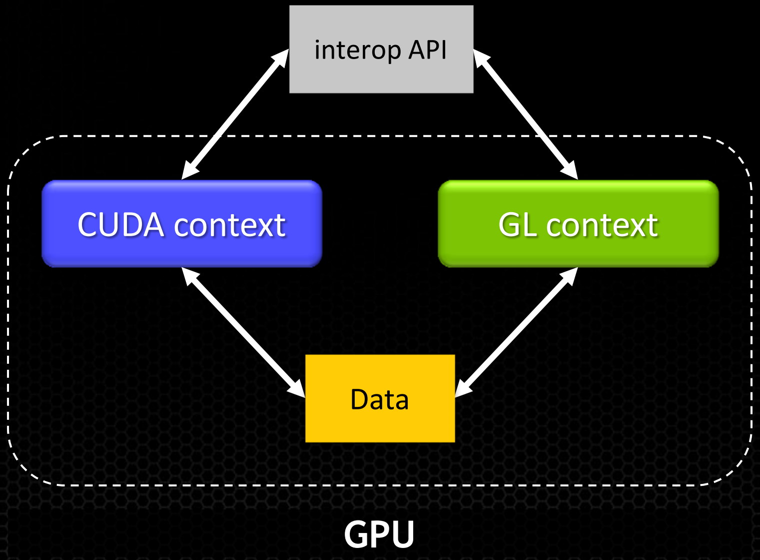
Code Sample - Simple buffer interop

- Mapping between contexts:

```
float* vboPtr;

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

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

      cudaGraphicsUnmapResources(1, &vboRes, 0);

      runGL(vboId);
}
```

Resource Behavior: Single-GPU

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



Code Sample - Simple buffer interop

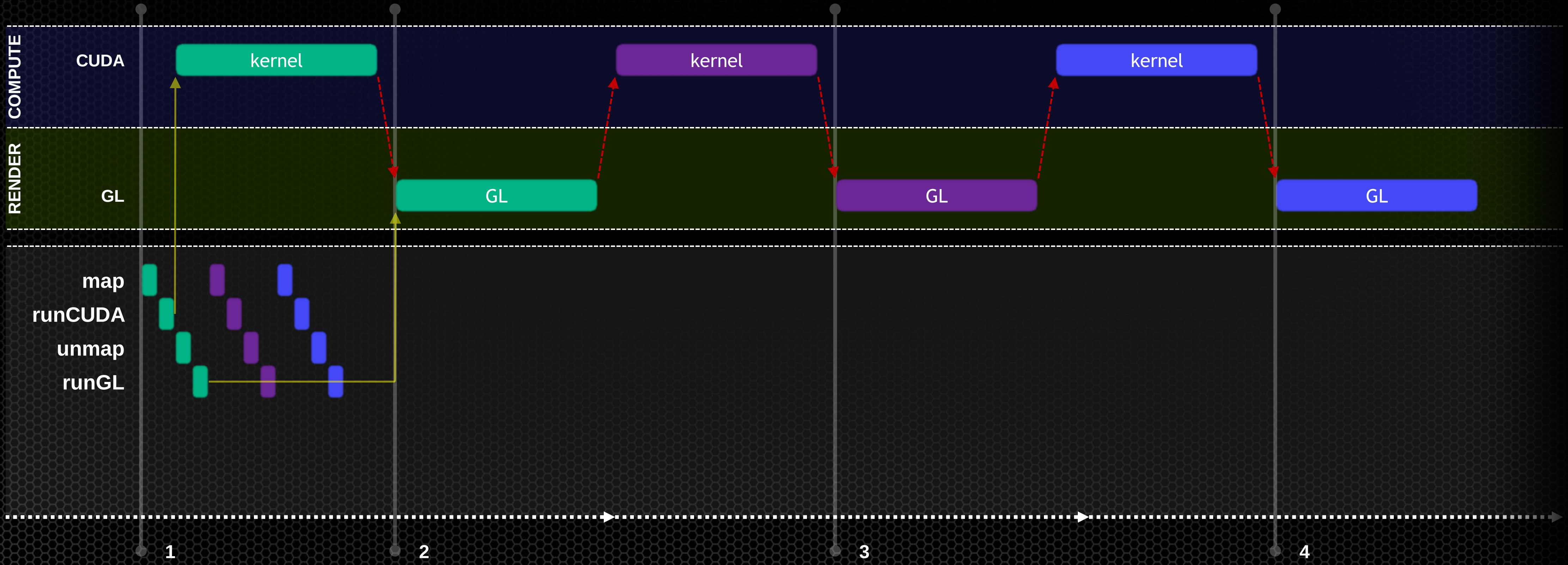
- Mapping between contexts:

```
float* vboPtr;

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

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

    cudaGraphicsUnmapResources(1, &vboRes, 0);

    runGL(vboId);
}
```

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

Timeline: Single-GPU

- Driver-interop



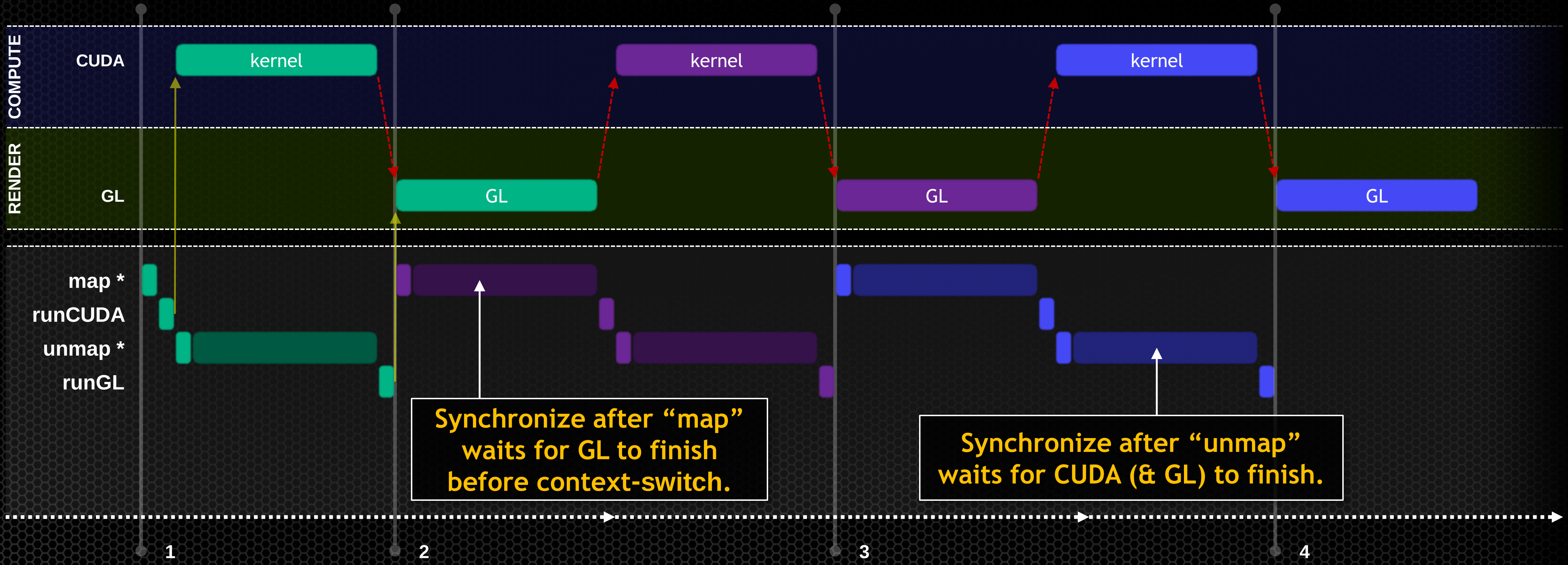
Code Sample - Simple buffer interop

- Adding synchronization for analysis:

```
float* vboPtr;

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

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

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

    runGL(vboId);
}
```

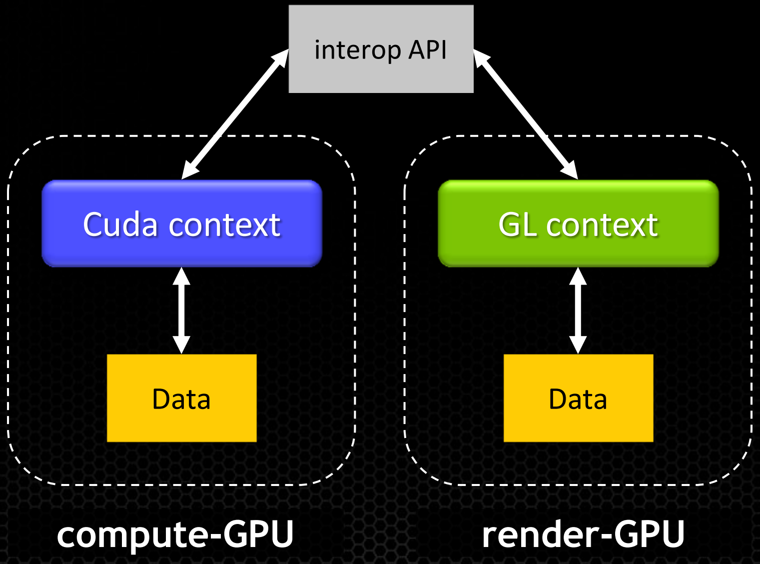
Timeline: Single-GPU

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



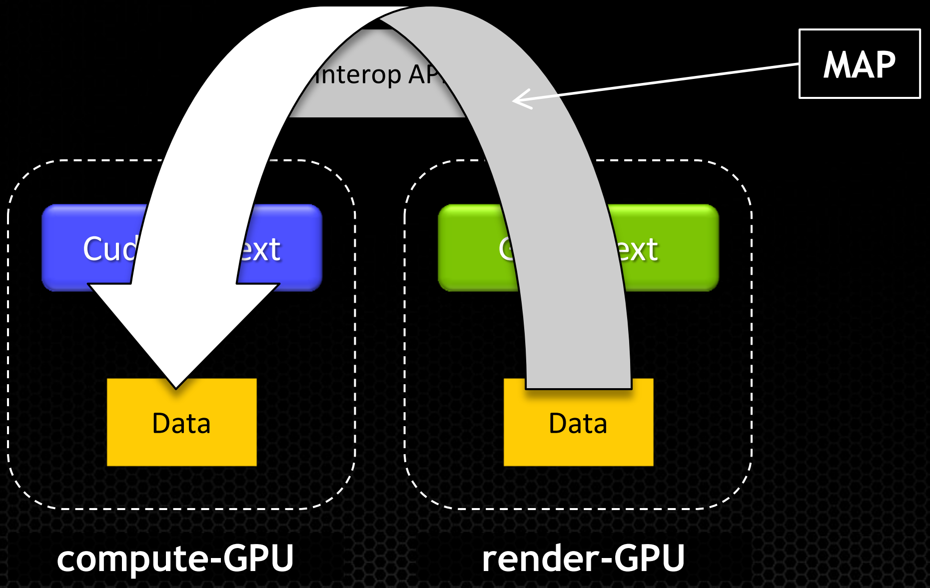
Resource Behavior: Multi-GPU

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



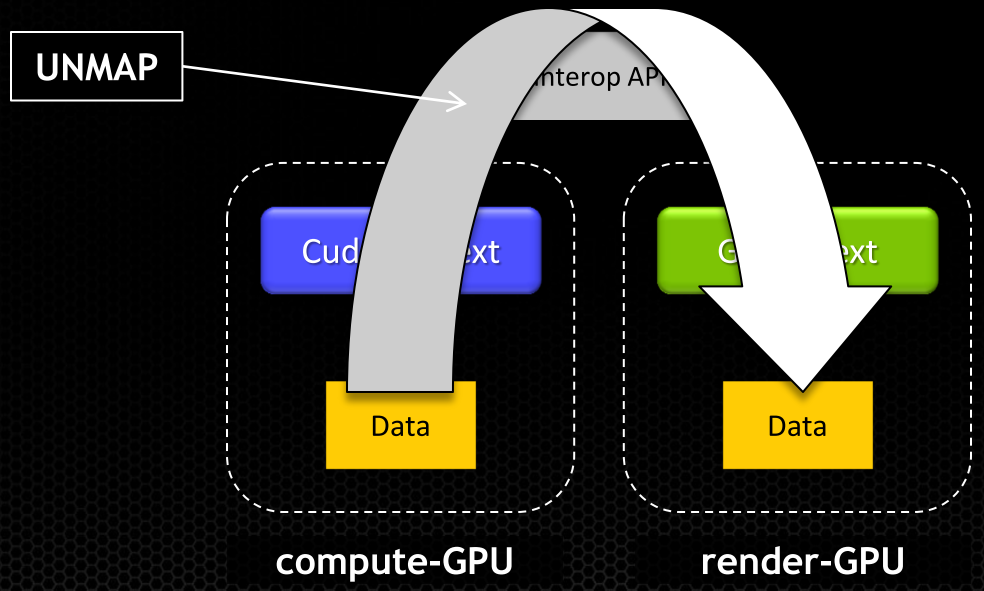
Resource Behavior: Multi-GPU

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



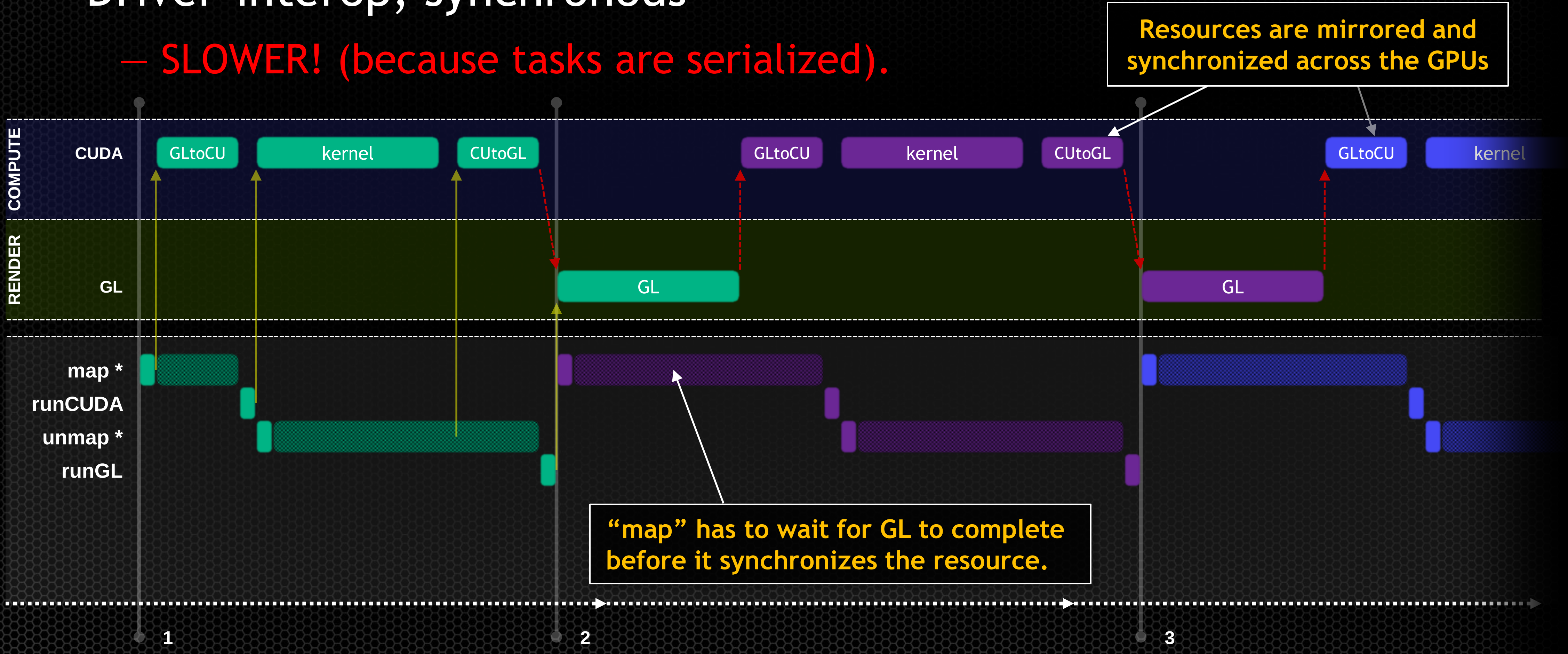
Resource Behavior: Multi-GPU

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



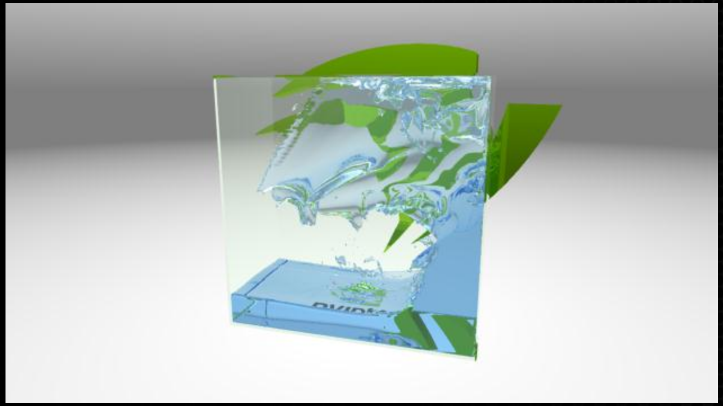
Timeline: Multi-GPU

- Driver-interop, synchronous*
  - SLOWER! (because 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

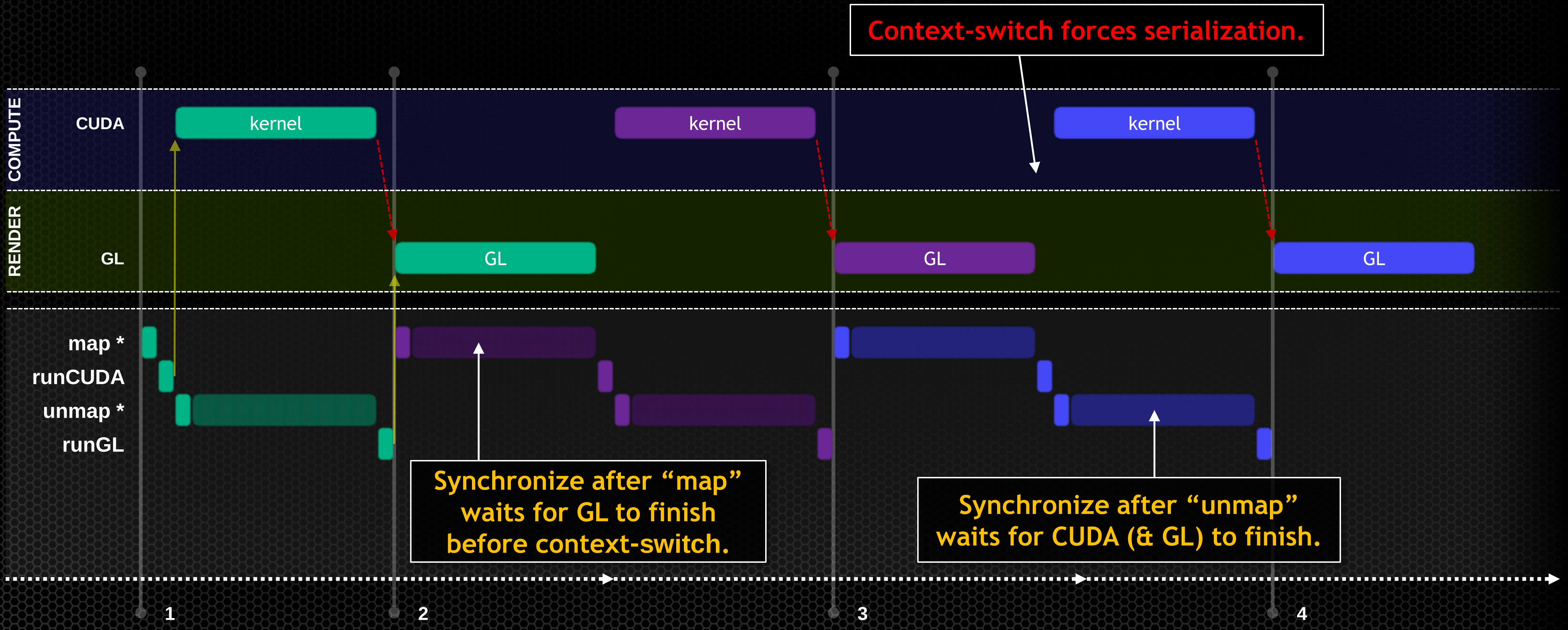
- CUDA produces… and OpenGL consumes:

```
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.

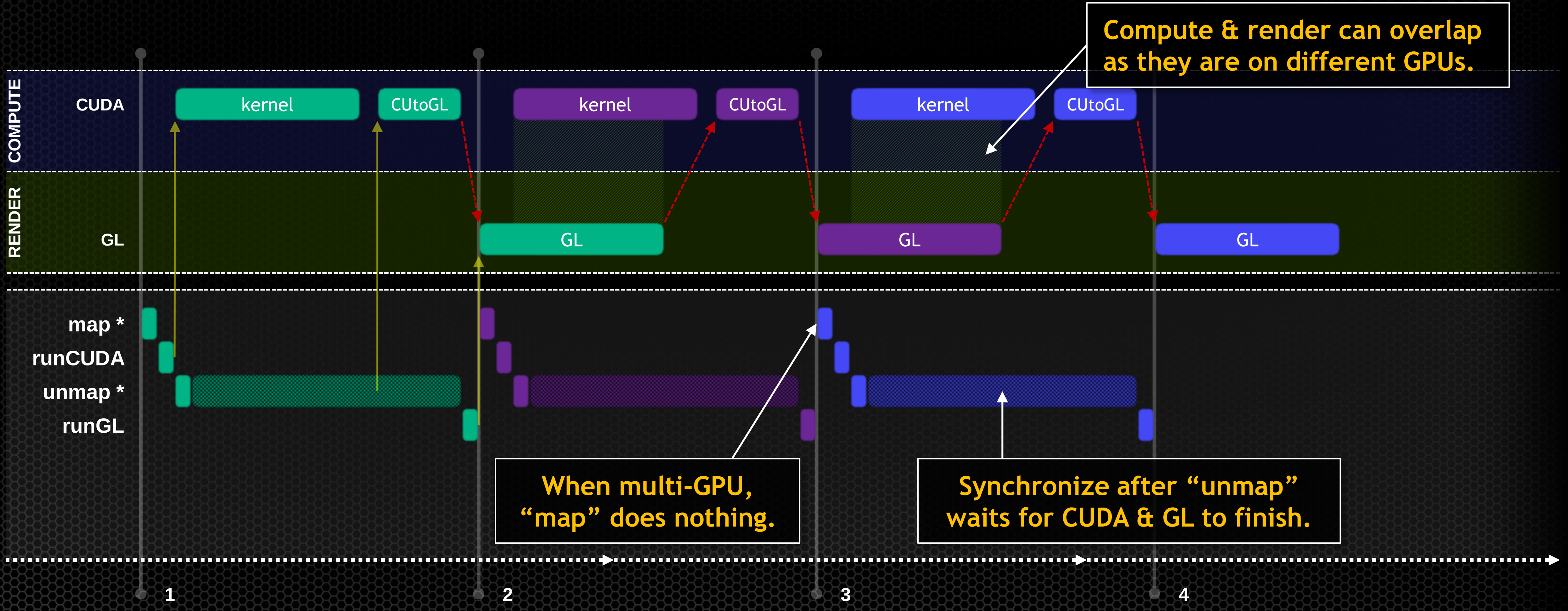
Timeline: Single-GPU

- Driver-interop, synchronous*, WRITE-DISCARD



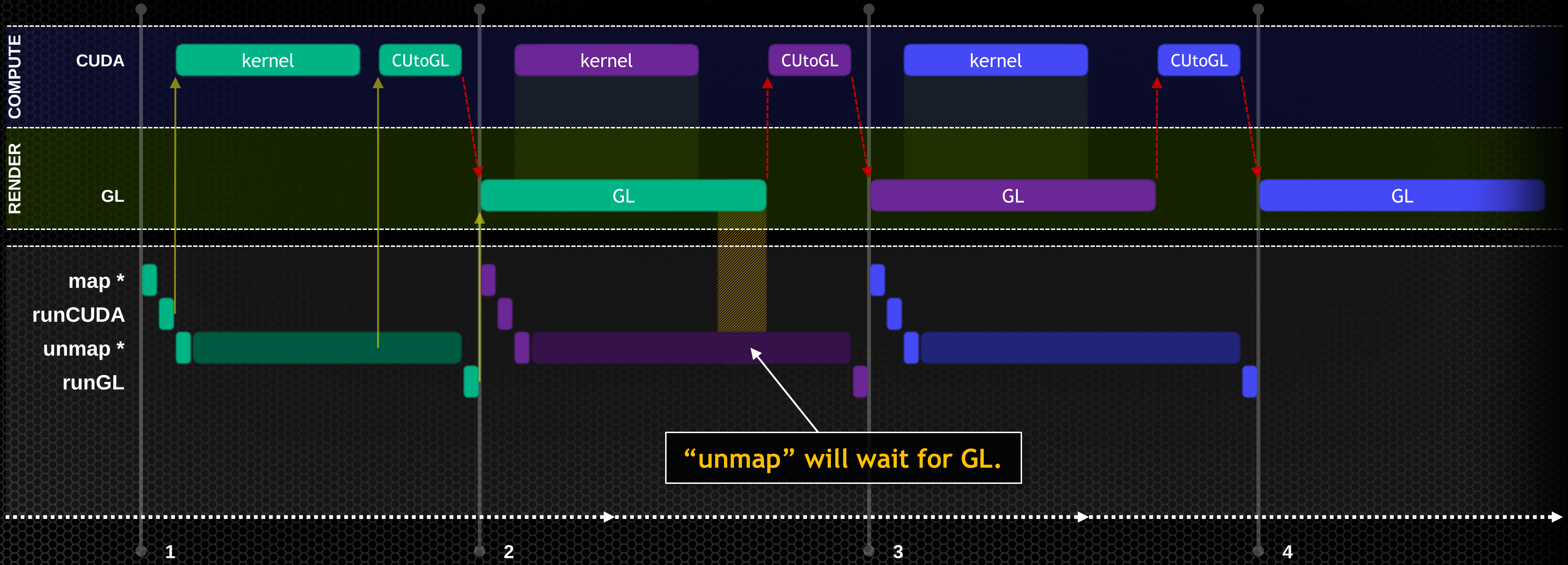
Timeline: Multi-GPU

- Driver-interop, synchronous*, WRITE-DISCARD

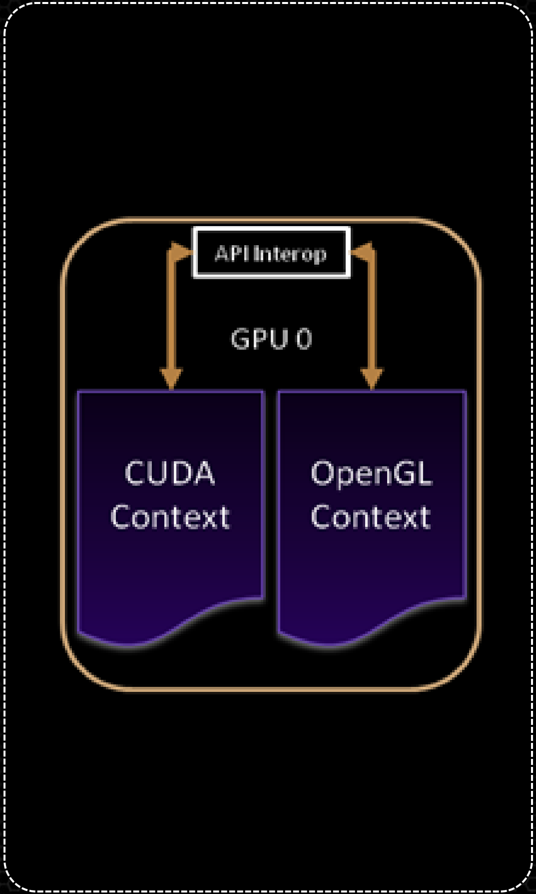


Timeline: Multi-GPU

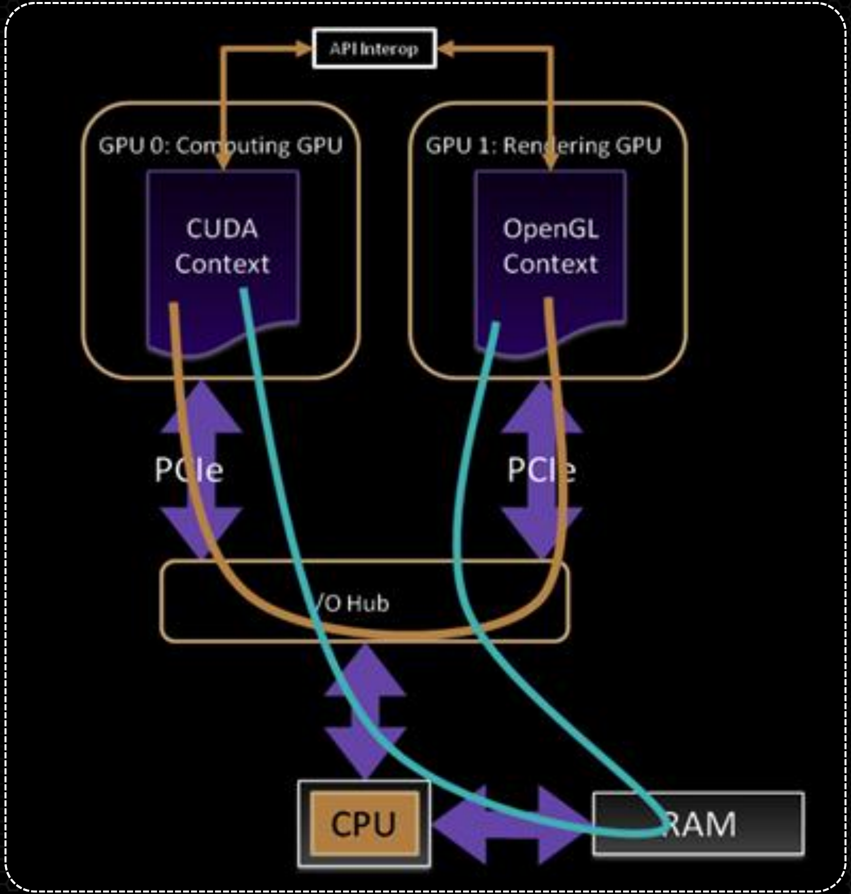
- Driver-interop, synchronous*, WRITE-DISCARD
  - – if render is long…



Driver-Interop: System View



Single-GPU

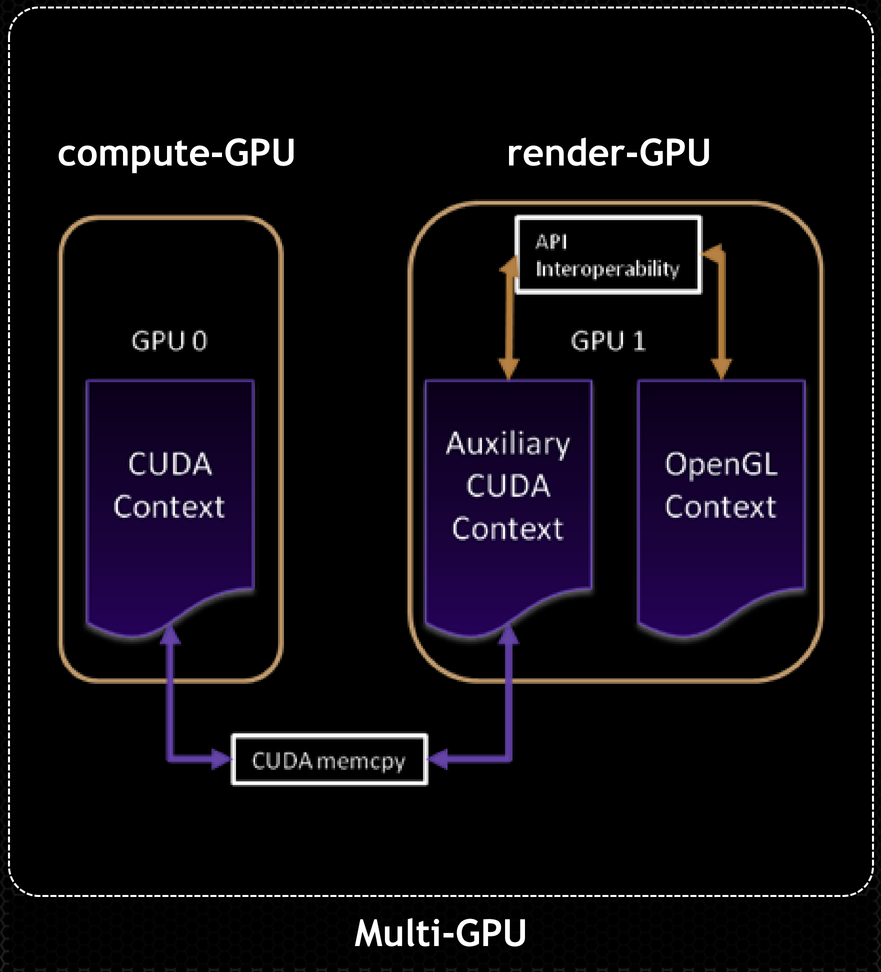


Multi-GPU

Manual-Interop

- Driver-Interop hides all the complexity, but sometimes...
  - We don’t want to transfer all data
    - Kernel may only compute sub-regions.
  - 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.

Manual-Interop: System View



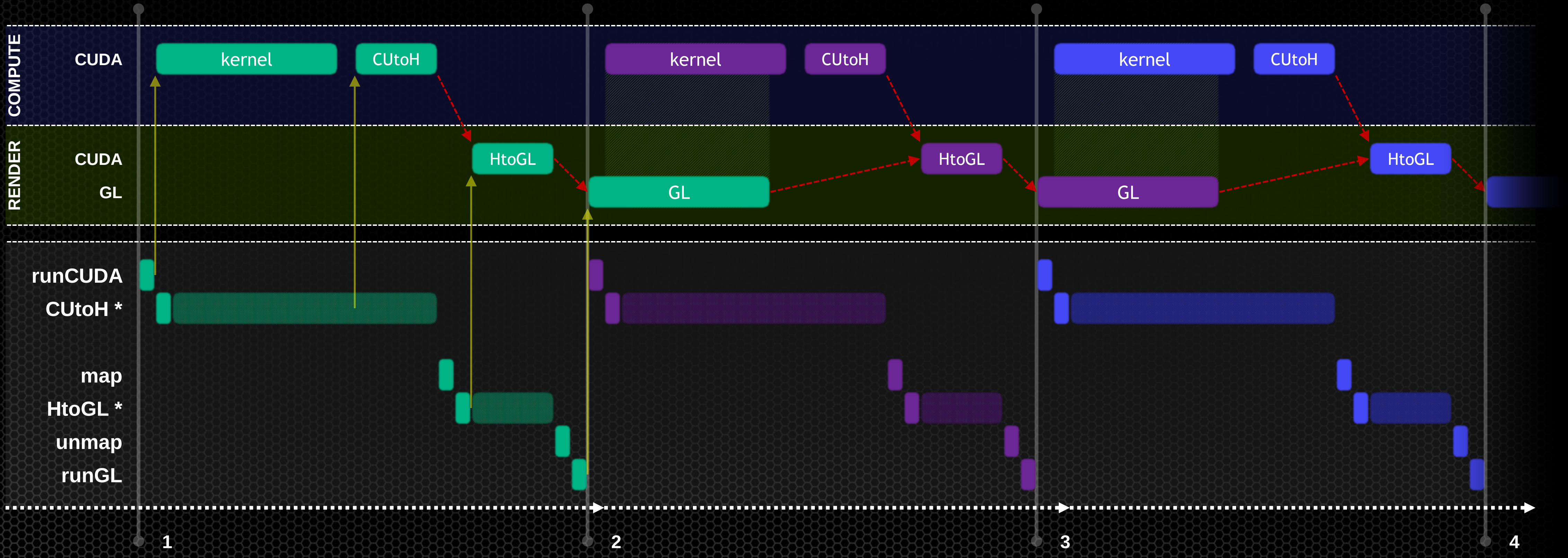
Code Sample - Manual-Interop

```
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

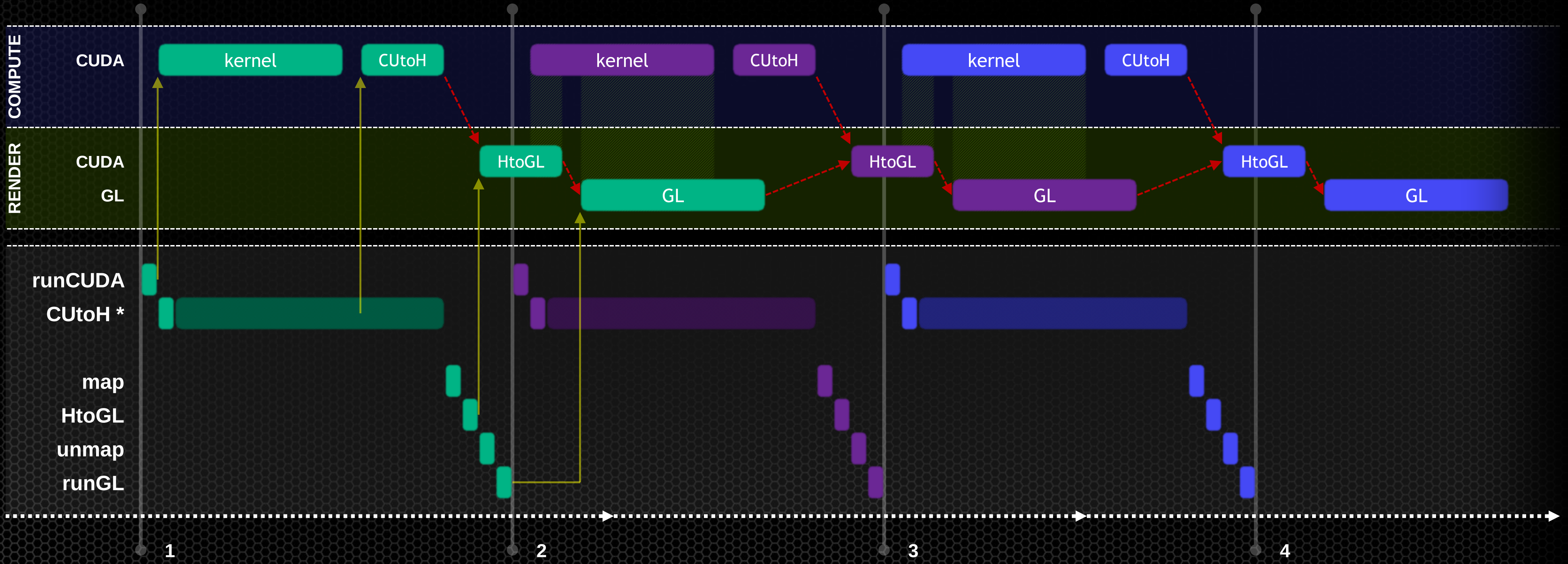


Code Sample - Manual Interop (Async)

```
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);
}
```

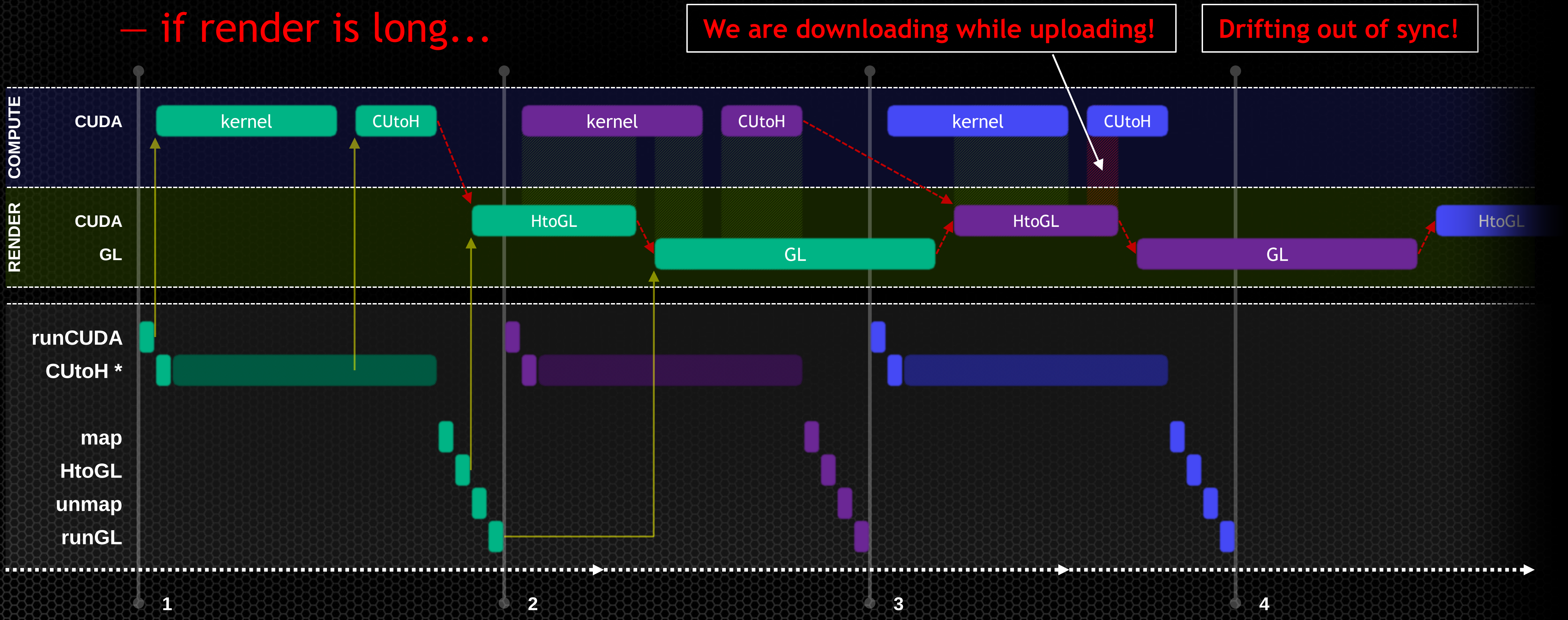
Timeline: Multi-GPU

- Manual-interop, asynchronous, WRITE-DISCARD



Timeline: Multi-GPU

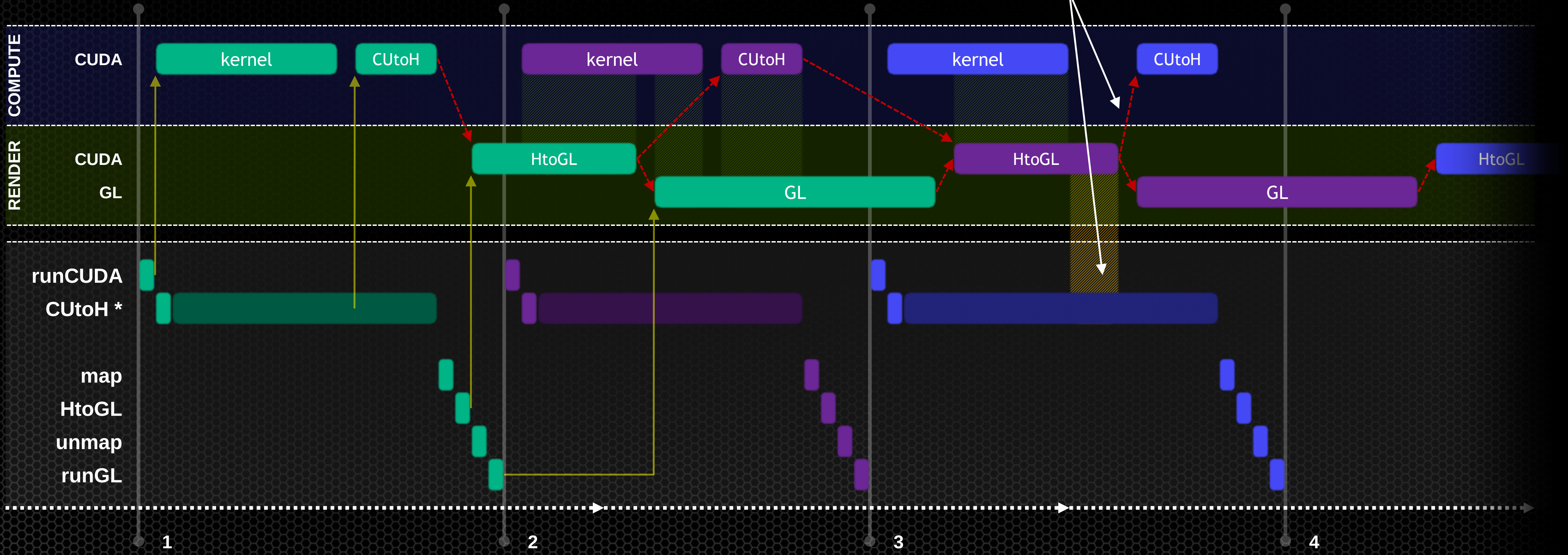
- Manual-interop, asynchronous, WRITE-DISCARD
  – if render is long…



Timeline: Multi-GPU (safe Async)

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

Synchronization must *also* wait for HtoGL to finish



Synchronization across GPUs

- Streams and events are per device
  - Determined by the GPU that is current when created.

- Stream GPU must be set current for
  - Launching kernel to a stream
  - Recording events to a stream

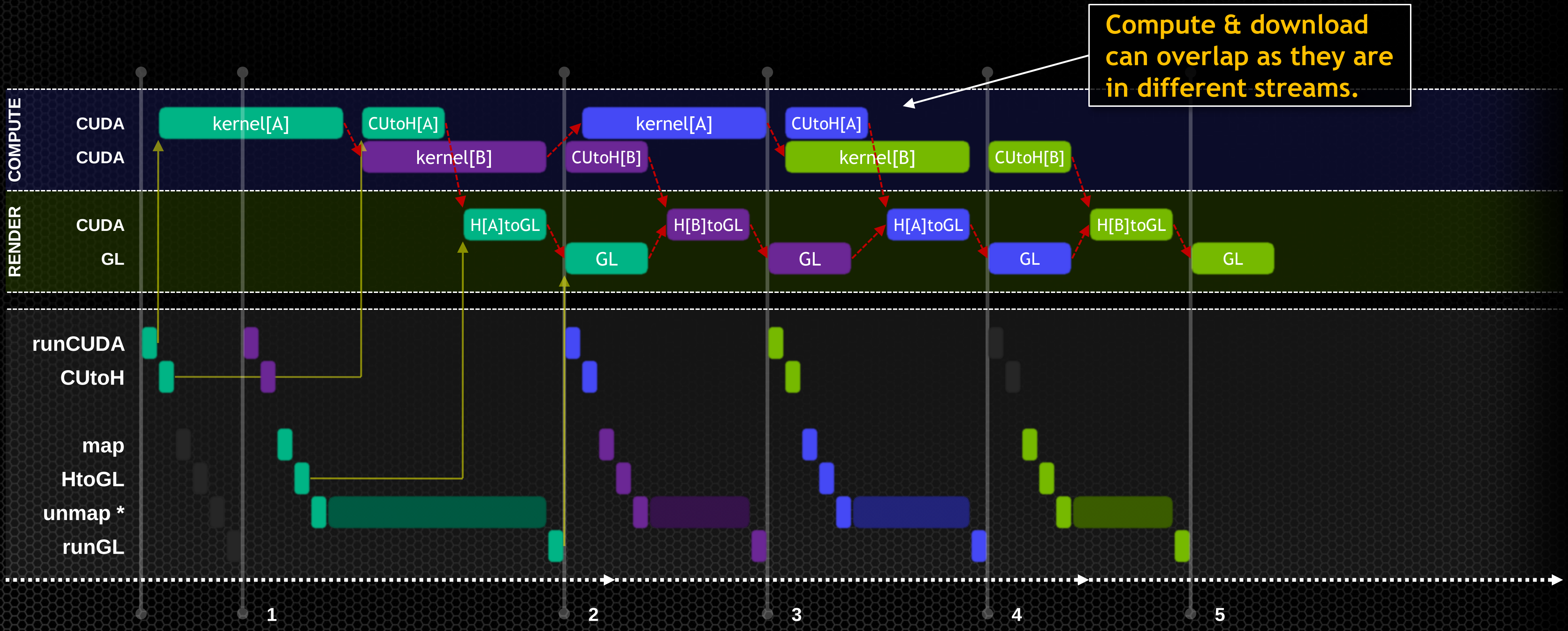
- Agnostic to the current GPU
  - Memcpy can be launched on any stream
  - Synchronize/Query of Events

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

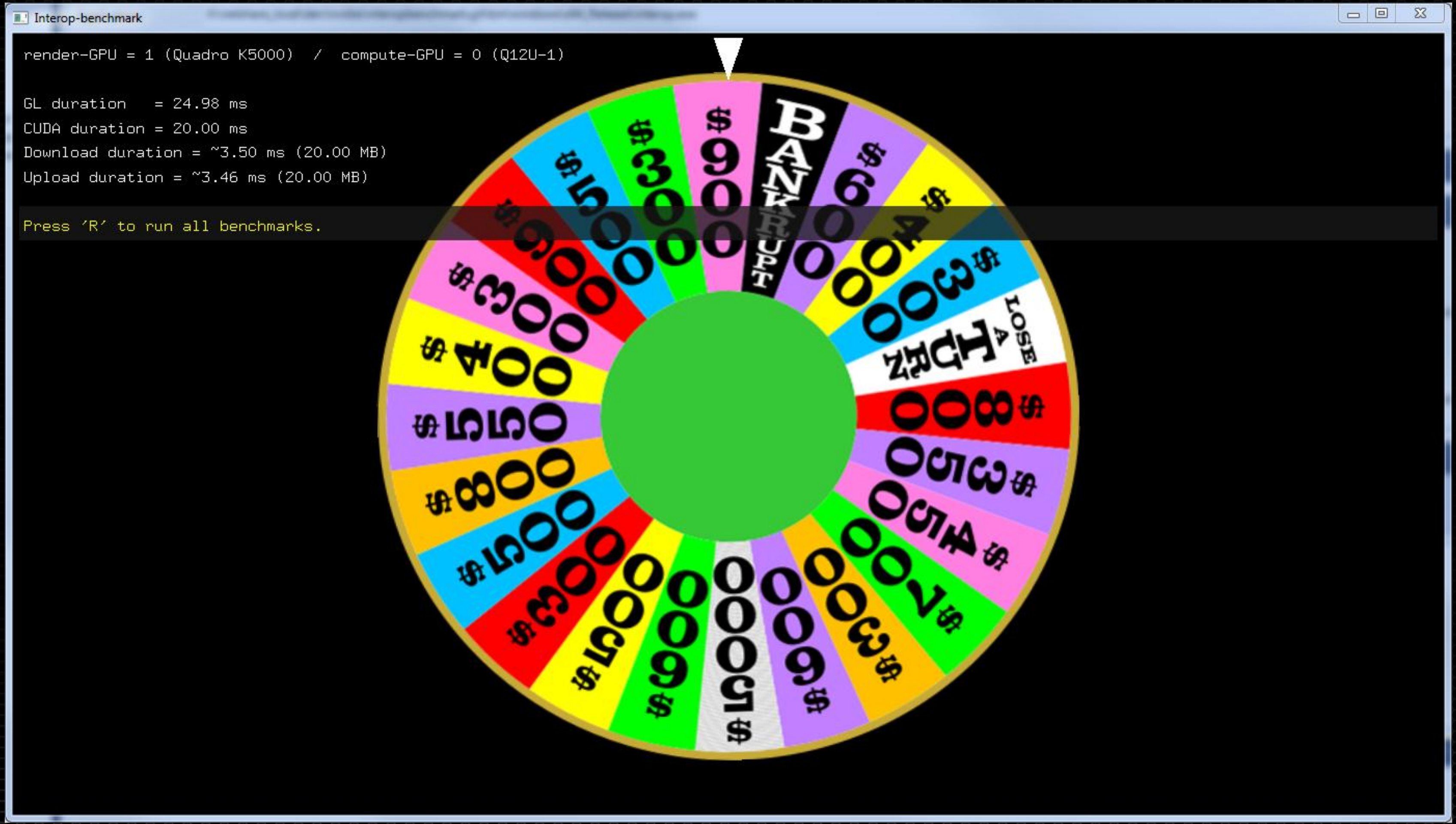
- Manual-interop, WRITE-DISCARD, flipping CUDA



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);
}
```

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 when OpenGL produces and CUDA consumes
  - Use `cudaGraphicsMapFlagsReadOnly`
- Avoid synchronizing GPUs!
  - 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 configuration and OS.

Resources

- CUDA samples/documentation
  - http://developer.nvidia.com/cuda-downloads
- Interop-benchmark demo
  - http://bitbucket.org/wbraithwaite_nvidia/interopbenchmark
- OpenGL Insights, Patrick Cozzi, Christophe Riccio, 2012. ISBN 1439893764.
  - www.openglinsights.com
- Thursday 12pm, Multi-GPU programming for Visual Computing (Part 2)