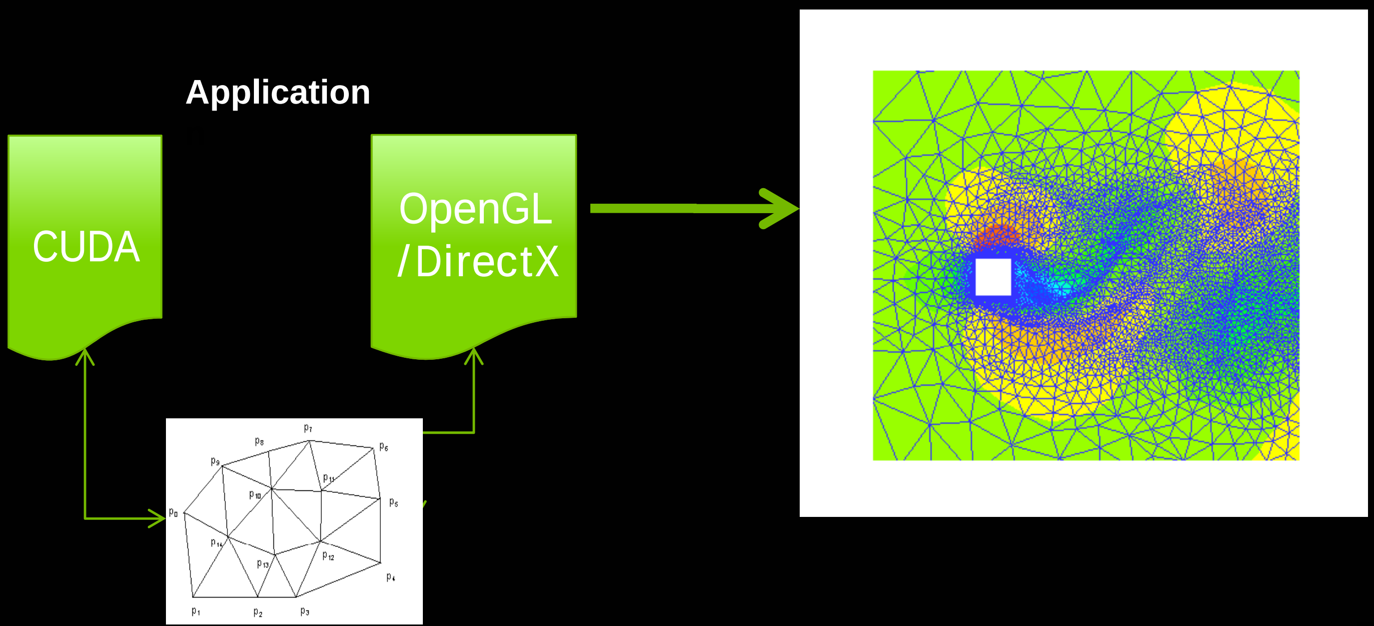
Mixing graphics and compute with multiple GPUs

Agenda

- Compute and Graphics Interoperability
- Interoperability at a system level
- Application design considerations

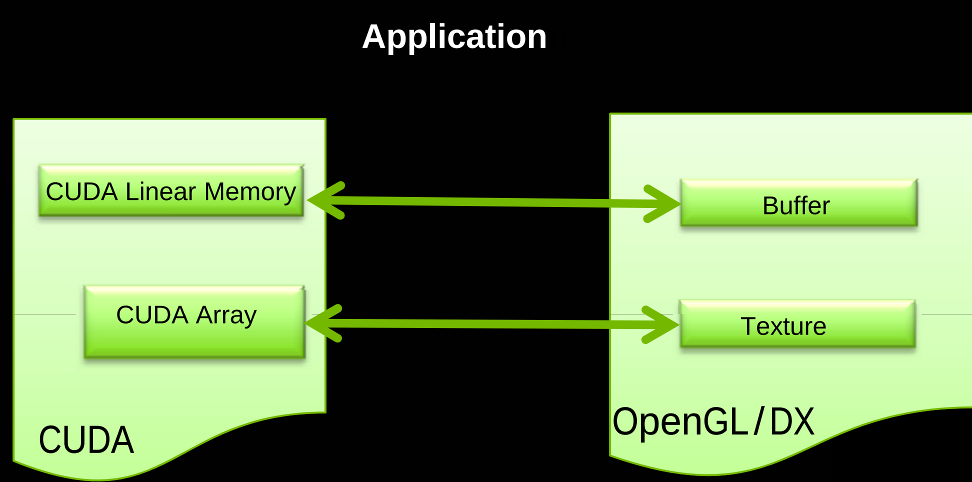
Putting Graphics & Compute together

- Compute and Visualize the same data



Compute/Graphics interoperability

- Set of compute API functions
  - Graphics sets up the objects
  - Register/Unregister the objects with compute context
  - Mapping/Unmapping of the objects to/from the compute context every frame



Simple OpenGL-CUDA interop sample: Setup and Register of Buffer Objects

```
GLuint imagePBO;
cudaGraphicsResource_t    cudaResourceBuf;
//OpenGL buffer creation
glGenBuffers(1, &imagePBO);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, imagePBO);
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, size, NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB,0);
//Registration with CUDA
cudaGraphicsGLRegisterBuffer(&cudaResourceBuf, imagePBO,
   cudaGraphicsRegisterFlagsNone);
```

Simple OpenGL-CUDA interop sample: Setup and Register of Texture Objects

```
GLuint imageTex;
cudaGraphicsResource_t    cudaResourceTex;
//OpenGL texture creation
glGenTextures(1, &imageTex);
glBindTexture(GL_TEXTURE_2D, imageTex);
//set texture parameters here
glTexImage2D(GL_TEXTURE_2D,0, GL_RGBA8UI_EXT, width, height, 0,
   GL_RGBA_INTEGER_EXT, GL_UNSIGNED_BYTE, NULL);
glBindTexture(GL_TEXTURE_2D, 0);
//Registration with CUDA
cudaGraphicsGLRegisterImage (&cudaResourceTex, imageTex, GL_TEXTURE_2D,
   cudaGraphicsRegisterFlagsNone);
```

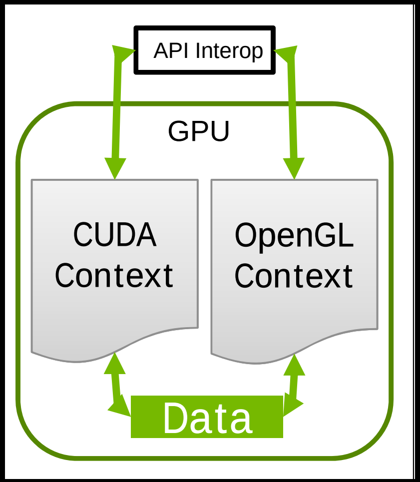
GPU TECHNOLOGY CONFERENCE

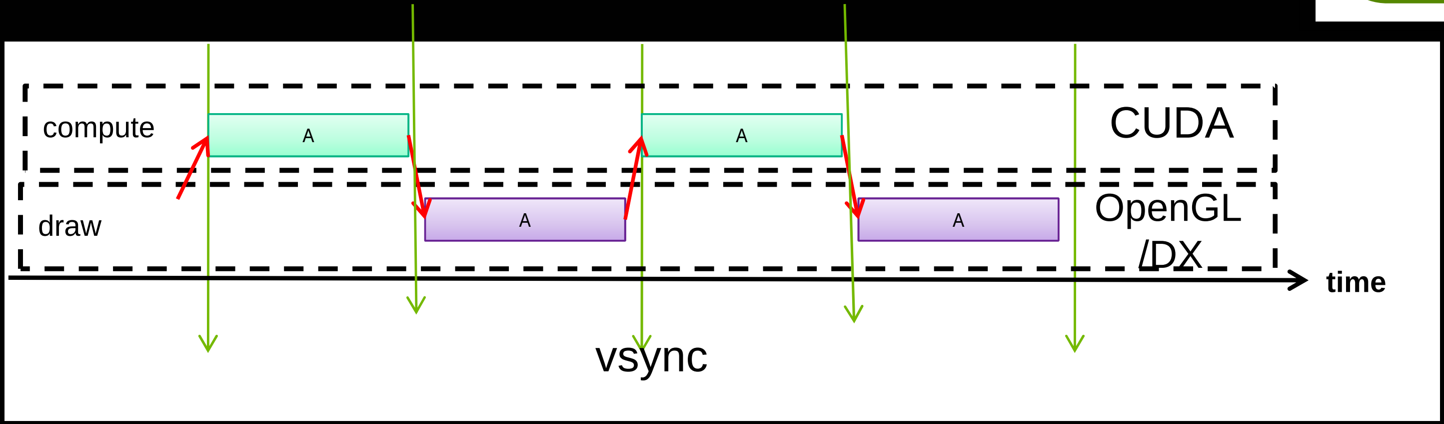
Simple OpenGL-CUDA interop sample

```
unsigned char *memPtr;
cudaArray *arrayPtr;
while (!done) {
   cudaGraphicsMapResources(1, &cudaResourceTex, cudaStream);
   cudaGraphicsMapResources(1, &cudaResourceBuf, cudaStream);
   cudaGraphicsSubResourceGetMappedArray(&cudaArray, cudaResourceTex, 0, 0);
   cudaGraphicsResourceGetMappedPointer((void **)&memPtr, &size, cudaResourceBuf);
   doWorkInCUDA(cudaArray, memPtr, cudaStream);//asynchronous
   cudaGraphicsUnmapResources(1, & cudaResourceTex, cudaStream);
   cudaGraphicsUnmapResources(1, & cudaResourceBuf, cudaStream);
   doWorkInGL(imagePBO, imageTex); //asynchronous
}
```

Interoperability behavior:single GPU

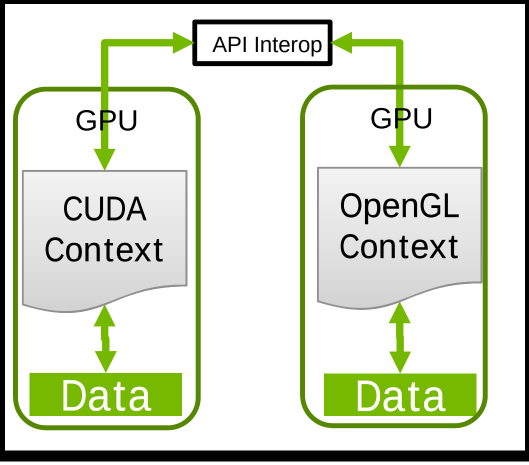
- The resource is shared
- Tasks are serialized

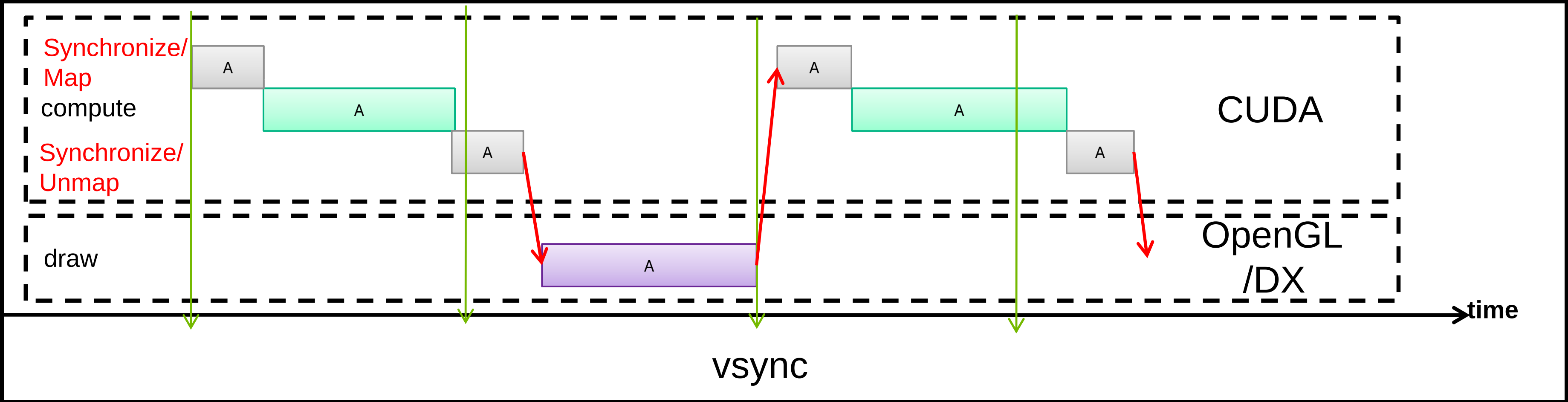




Interoperability behavior: multiple GPUs

- Each context owns a copy of the resource
- Tasks are serialized
- map/unmap might do a host side synchronization

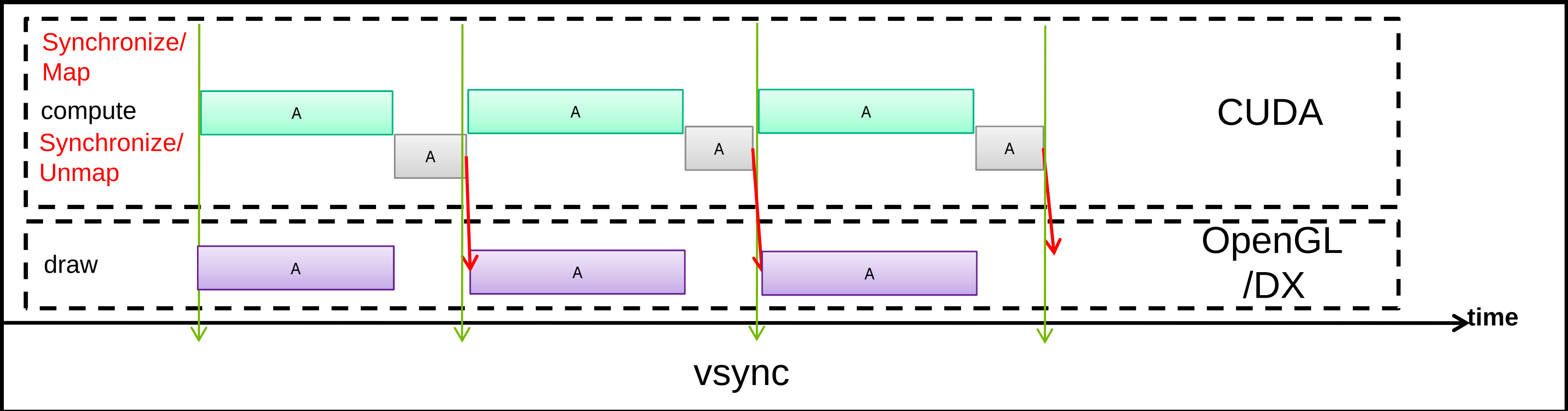




Interoperability behavior: multiple GPUs Improvements

- If one of the APIs is a producer and another is a consumer then the tasks can overlap.

CUDA as a producer example:



Simple OpenGL-CUDA interop sample

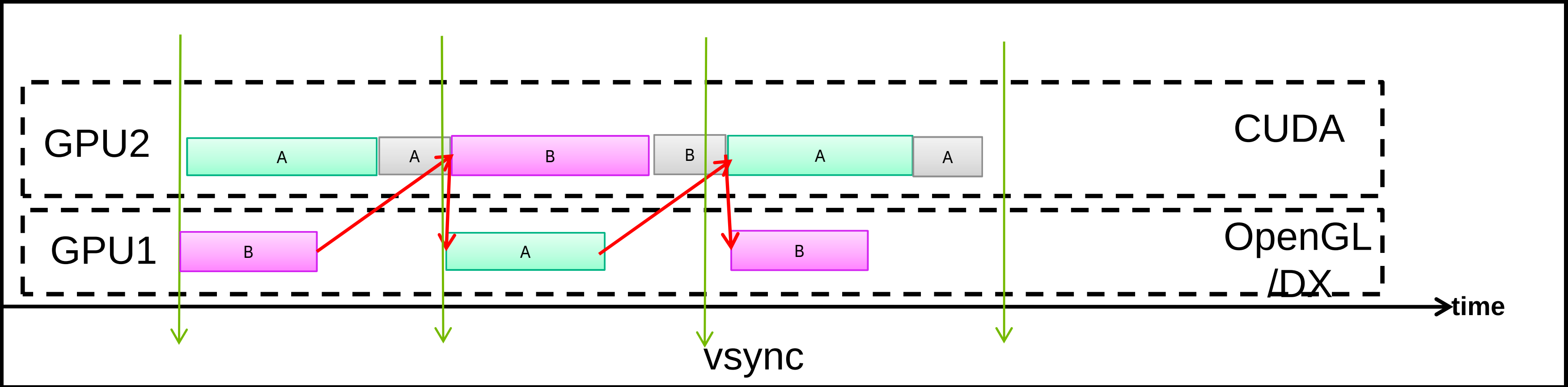
- Use mapping hint with cudaGraphicsResourceSetMapFlags()

cudaGraphicsMapFlagsReadOnly/cudaGraphicsMapFlagsWriteDiscard:

```
unsigned char *memPtr;
cudaGraphicsResourceSetMapFlags(cudaResourceBuf, cudaGraphicsMapFlagsWriteDiscard)
while (!done) {
   cudaGraphicsMapResources(1, &cudaResourceBuf, cudaStream);
   cudaGraphicsResourceGetMappedPointer((void **)&memPtr, &size, cudaResourceBuf);
   doWorkInCUDA(memPtr, cudaStream);//asynchronous
   cudaGraphicsUnmapResources(1, &cudaResourceBuf, cudaStream);
   doWorkInGL(imagePBO); //asynchronous
}
```

Interoperability behavior: multiple GPUs Improvements

- use ping-pong buffers to ensure the graphics and compute are not stepping on each other's toes.



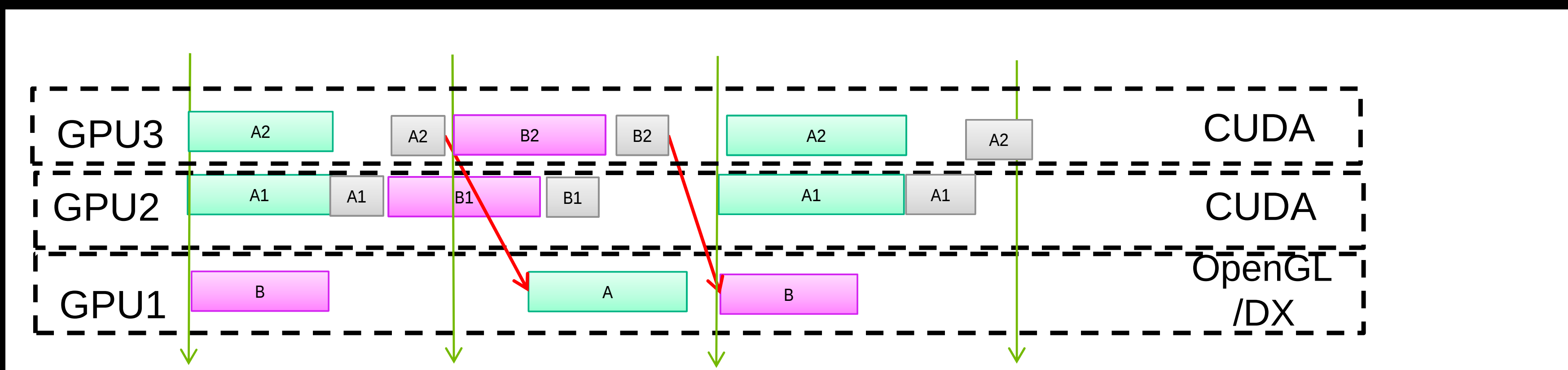
Simple OpenGL-CUDA interop sample

- ping-pong buffers:

```
unsigned char *memPtr;
int count = 0;
cudaGraphicsResourceSetMapFlags(cudaResourceBuf, cudaGraphicsMapFlagsWriteDiscard)
while (!done) {
    cudaResourceBuf = (count%2) ? cudaResourceBuf1 : cudaResourceBuf2;
    imagePBO = (count%2) ? imagePBO2 : imagePBO1;
    cudaGraphicsMapResources(1, &cudaResourceBuf, cudaStream);
    cudaGraphicsResourceGetMappedPointer((void **)&memPtr, &size, cudaResourceBuf);
    doWorkInCUDA(memPtr, cudaStream);//asynchronous
    cudaGraphicsUnmapResources(1, & cudaResourceBuf, cudaStream);
    doWorkInGL(imagePBO); //asynchronous
    count++;
}
```

Simple OpenGL-CUDA interop sample: What now?

- You can continue using a single threaded application if map/unmap and other calls are CPU asynchronous. If they are CPU synchronous, this won't be possible:



Application Example:pseudocode

- Multithreaded CUDA centric application: Adobe Premiere Pro with an OpenGL plugin

Main CUDA thread

```
int count = 1;
while (!done) {
    SignalWait(oglDone[count]);
    doWorkInCUDA(memPtr, NULL);
    EventSignal(cudaDone[count]);
    count = (count+1)%2;
}
```

OpenGL Worker thread

```
mainCtx = wglCreateContext(hDC);
wglMakeCurrent(hDC, mainCtx);
//Register OpenGL objects with CUDA
int count = 0;
while (!done) {
    SignalWait(cudaDone[count]);
    cudaGraphicsUnmapResources(1, &cudaResourceBuf[count], NULL);
    doWorkInGL(imagePBO[count]);
    cudaGraphicsMapResources(1, &cudaResourceBuf[count], NULL);
    cudaGraphicsResourceGetMappedPointer((void **)&memPtr, &size,
    cudaResourceBuf[count]);
    EventSignal(oglDone[count]);
    count = (count+1)%2;
}
```

GPU TECHNOLOGY CONFERENCE

Application Example:pseudocode

- Multithreaded OpenGL centric application: Autodesk Maya with a CUDA plug-in
  - **S0364 - Interacting with Huge Particle Simulations in Maya with the GPU, Wil B., GTC 2012 Proceedings**

CUDA worker thread N

```
SignalWait(setupCompleted);
wglMakeCurrent(hDC,workerCtx);
//Register OpenGL objects with CUDA
int count = 1;
while (!done) {
    SignalWait(oglDone[count]);
    glWaitSync(endGLSync[count]);
    cudaGraphicsMapResources(1, &cudaResourceBuf[count], cudaStream[N]);
    cudaGraphicsResourceGetMappedPointer((void **)&memPtr, &size, cudaResourceBuf[count]);
    doWorkInCUDA(memPtr, cudaStream[N]);
    cudaGraphicsUnmapResources(1, &cudaResourceBuf[count], cudaStream[N]);
    cudaStreamSynchronize(cudaStreamN);
    EventSignal(cudaDone[count]);
    count = (count+1)%2;
}
```

Main OpenGL thread

```
mainCtx = wglCreateContext(hDC);
workerCtx = wglCreateContextAttrib
                (hDC,mainCtx…);
wglMakeCurrent(hDC, mainCtx);
//Create OpenGL objects
EventSignal(setupCompleted);
int count = 0;
while (!done) {
    SignalWait(cudaDone[count]);
    doWorkInGL(imagePBO[count]);
    endGLSync[count] = glFenceSync(…);
    EventSignal(oglDone[count]);
    count = (count+1)%2;
}
```

Application design considerations

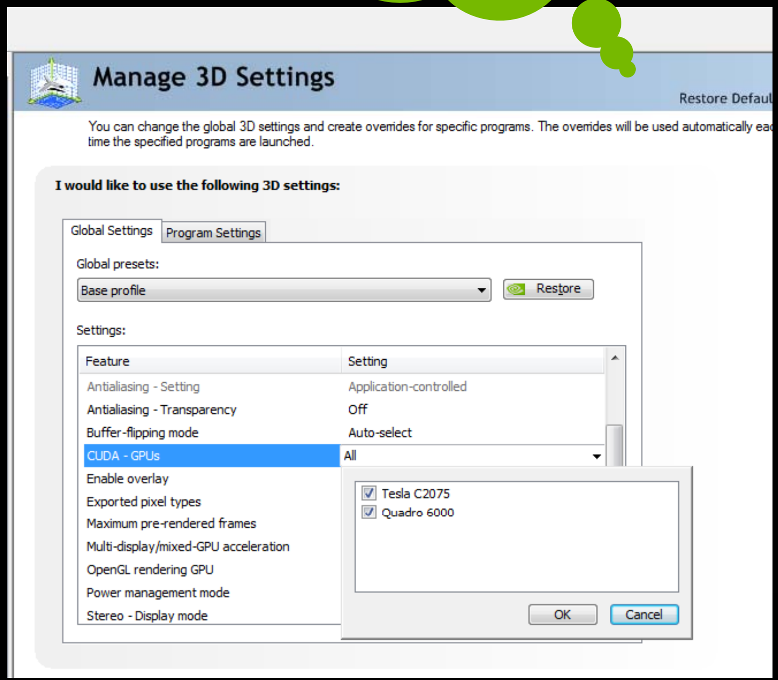
- Use cudaD3D[9|10|11]GetDevices/cudaGLGetDevices to chose the right device to provision for multi-GPU environments.
- Avoid synchronized GPUs for CUDA!
- CUDA-OpenGL interop can perform slower if OpenGL context spans multiple GPU!

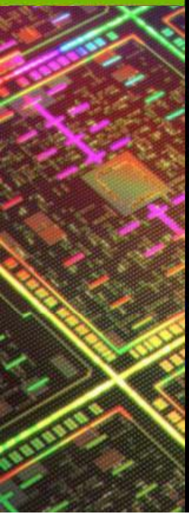
GPU TECHNOLOGY CONFERENCE

Application design considerations

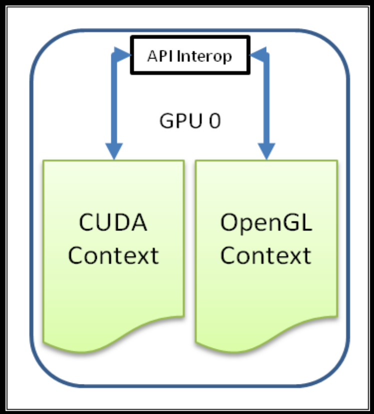
- Allow users to specify the GPUs!
  - Typical heuristics:
    - TCC mode
    - GPU #
    - available memory
    - # of processing units
  - Affecting factors: OS, ECC, TCC mode

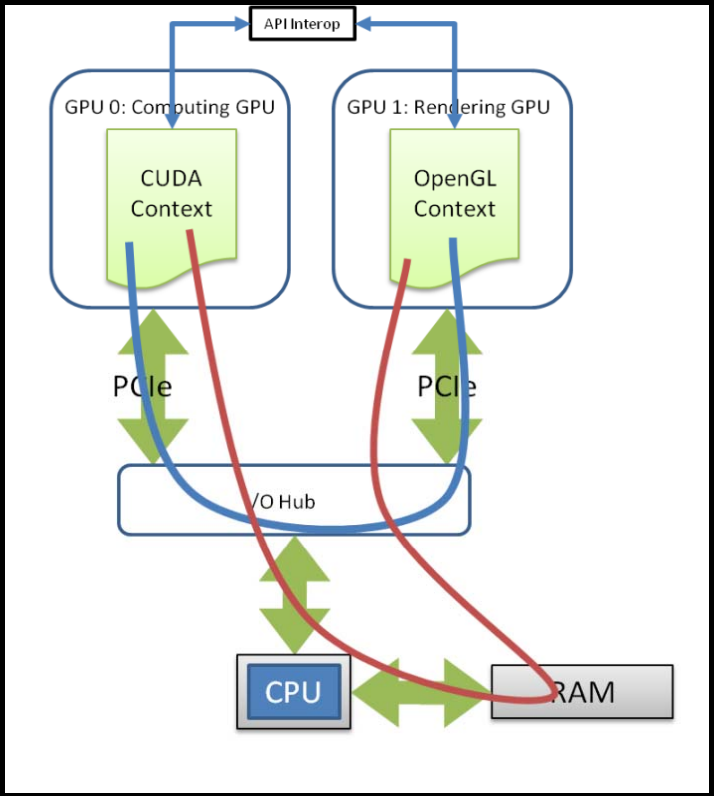
Don't make your users go here:





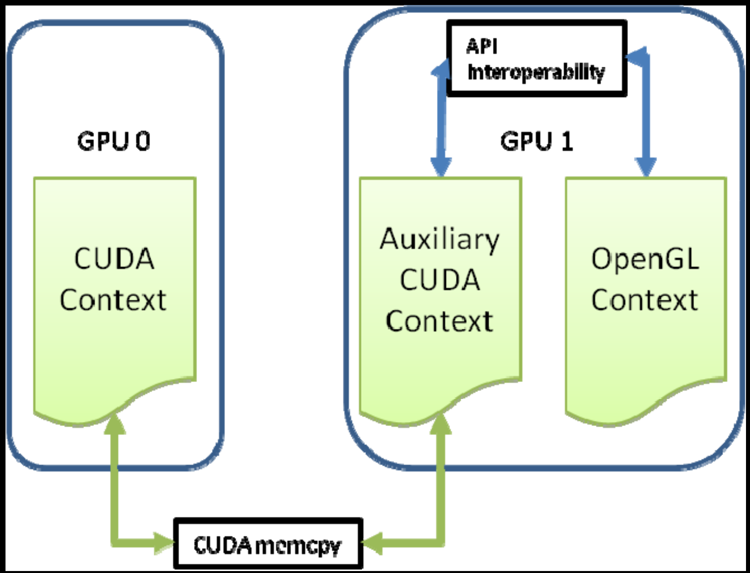
API Interop hides all the complexity



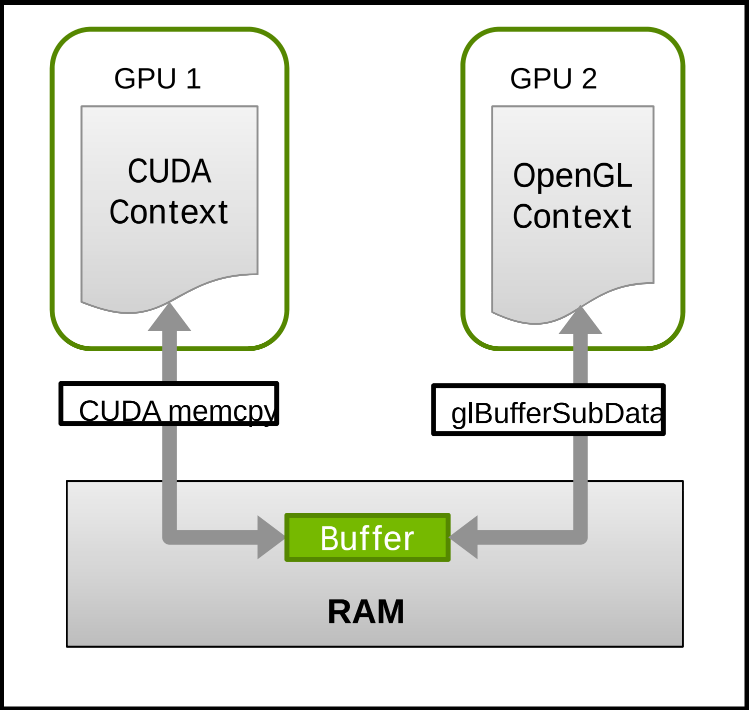


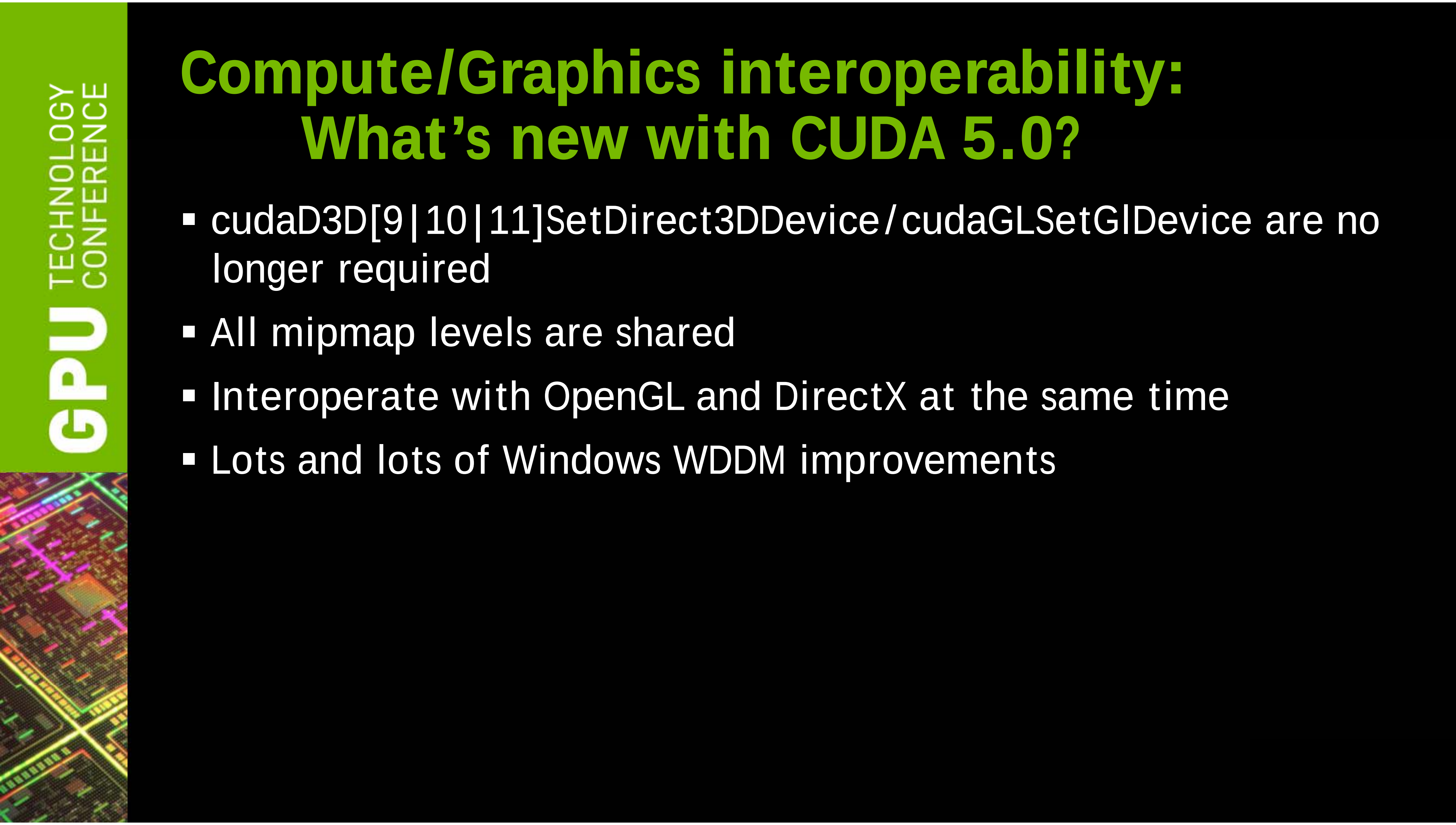
If not cross-GPU API interop then what?

A)



B)




GPU TECHNOLOGY CONFERENCE
Compute/Graphics interoperability: What's new with CUDA 5.0?
cudaD3D[9|10|11]SetDirect3DDevice/cudaGLSetGlDevice are no longer required
All mipmap levels are shared
Interoperate with OpenGL and DirectX at the same time
Lots and lots of Windows WDDM improvements

Conclusions/Resources

- The driver will do all the heavy lifting but..
- Scalability and final performance is up to the developer and..
- For fine grained control you might want to move data yourself.
- CUDA samples/documentation:

http://developer.nvidia.com/cuda-downloads