NVIDIA QUADRO DUAL COPY ENGINES

WP-05462-001_v01 | October 2010

White Paper

DOCUMENT CHANGE HISTORY

WP-05462-001_v01

Version	Date	Authors	Description of Change
01	October 14, 2010	SV, SM	Initial Release

TABLE OF CONTENTS

LIST OF FIGURES

NVIDIA QUADRO DUAL COPY ENGINES

INTRODUCTION

The evolution of high-performance and fully programmable graphics processing units (GPUs) has led to tremendous advancements in graphics and parallel computing. With the introduction of the new NVIDIA Quadro® professional graphics solutions, based on the innovative NVIDIA® Fermi architecture [1], application developers greatly optimize data throughput for maximized application performance.

In the past, data transfers would stall due to architectural limitations in synchronizing the data with the GPU processing. For example, during texture uploads or frame buffer readbacks, the GPU is blocked from processing and incurs a heavy context switch. This synchronization requirement of traditional GPUs limits the overall processing throughput capabilities and creates bottlenecks with high performance applications.

With the introduction of the Fermi architecture, the new Quadro® solutions feature NVIDIA Dual Copy Engines that enable asynchronous data transfers with concurrent 3-way overlap. The current set of data can be processed while the previous set can be readback from the GPU, and the next set is uploaded. Figure 1 shows a typical system architecture block diagram. It is seen that even with high performance PCI Express ×16 bandwidth, the Quadro GPU memory bandwidth is many orders faster than the bus and the CPU RAM bandwidth. By overlapping transfers and compute, the PCI Express memory latency can be hidden so that by the time the GPU is ready to process a piece of data, it is already fetched into the high bandwidth area.

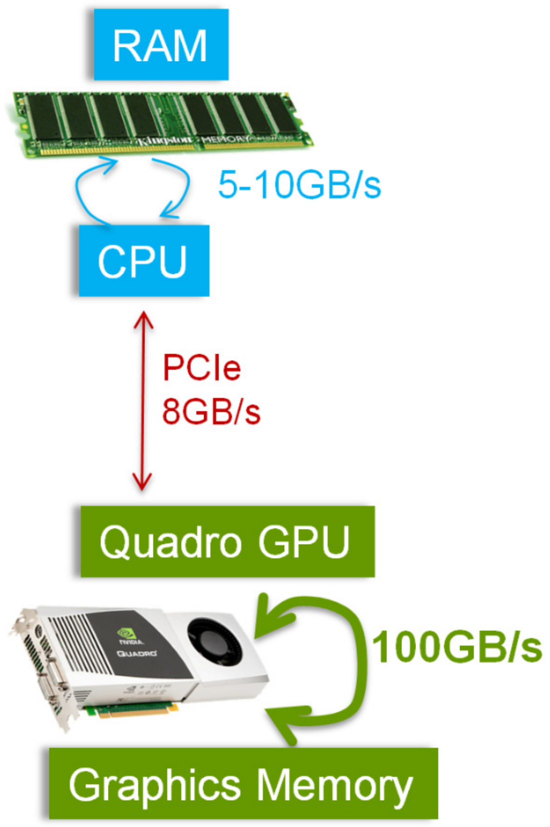

Figure 1. Typical System Architecture Block Diagram

Some examples for overlapped transfers with Quadro dual copy engines are:

- **Video processing or time-varying geometry/volumes** including post processing, video upload to maintain a frame rate and readback to save to disk.
- **Parallel numerical simulation** that uses domain decomposition techniques such as Finite Element/Volume. The Quadro GPU can be used as a co-processor that is able to download, process and readback the various subdomains with CPU scheduling.
- **Parallel rendering** - When a scene is divided and rendered across multiple Quadro GPUs with the color and depth readback for composition, parallelizing readback will speed up the pipeline. Likewise for sort-first implementation where at every frame the data has to be streamed to the GPU based on the viewpoint.
- **Data bricking** for large image, terrains and volumes. Bricks or LODs are paged in and out as needed in another thread without disruption to the rendering thread.
- **Cache for OS** – OS can page in and out textures as needed eliminating shadow copies in RAM.

CURRENT STREAMING APPROACHES

A typical download-process-readback pipeline can be broken down into the following:

- **Copy** – involves CPU cycles in data conversions if any to native GPU formats and `memcpy` from the application memory space to the driver space.
- **Download** – the time for the actual data transfer on PCI Express from host to device.
- **Process** – GPU cycles for rendering and compute.
- **Readback** – time for the data transfer from device back to host.

To achieve maximum end-to-end throughput on the GPU, maximum overlap is required between these various components in the pipeline.

Synchronous Downloads

The straightforward download method for textures is to call `glTexSubImage` which involves and blocks the CPU while copying data from user space to the driver space and subsequent data transfer on the bus to the GPU. Figure 2 illustrates the inefficiency of this method as the GPU is idle while the CPU is busy with the `memcpy`.

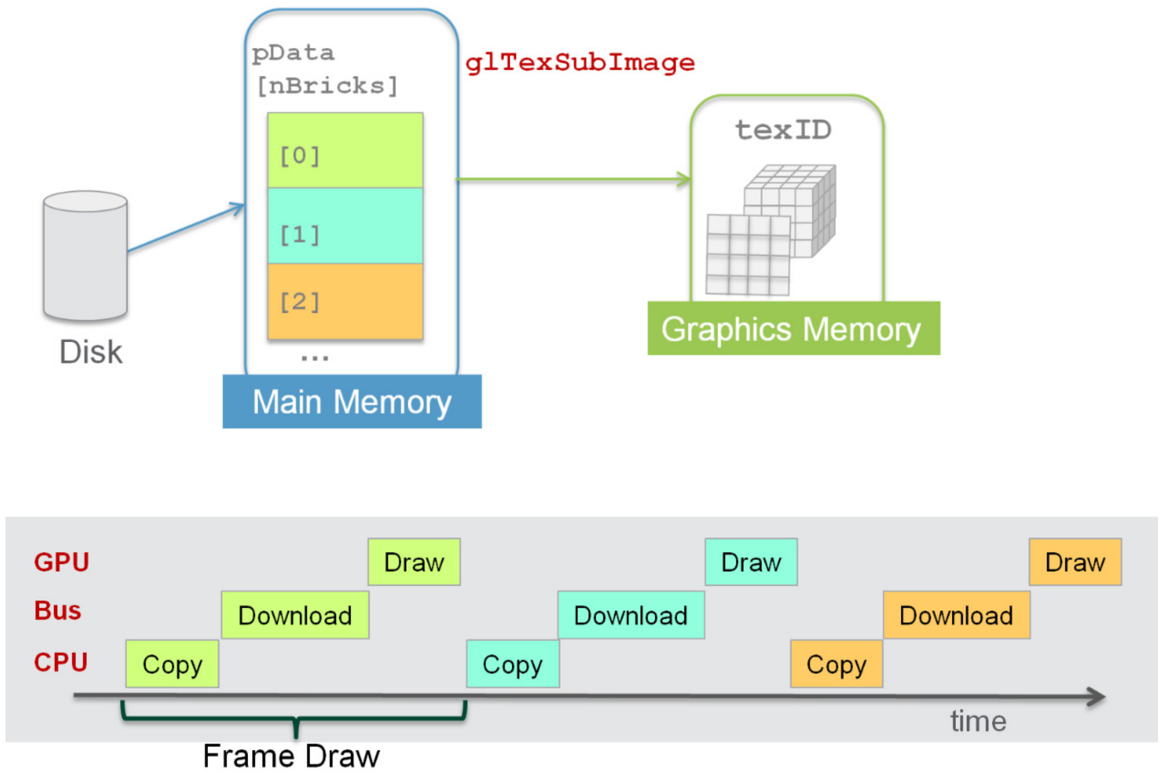

Figure 2. Synchronous Downloads With No Overlap

Initialization

```
glGenTextures(1,&texID);
//TODO - read from file to pData
glBindTexture(GL_TEXTURE_2D, texID);
//TODO – Set Texture Params like wrap, filter using glTexParameteri
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,width,height,0,GL_RGBA,GL_UNSIGNE
D_BYTE,pData[0]);
```

Draw

```
glBindTexture(GL_TEXTURE_2D, texID);
glTexSubImage2D(GL_TEXTURE_2D,0, 0,0,width,height,
            GL_RGBA, GL_UNSIGNED_BYTE, m_pData[m_curBrick]);
//TODO - Call drawing code here
```

CPU Asynchronous Downloads with PBOs

The OpenGL PBO [2] mechanism provides for transfers that are asynchronous on the CPU. If an application can schedule enough work between initiating the transfer and actually using the data, CPU asynchronous transfers are possible. In this case, the `glTexSubImage` call operates with little CPU intervention. PBOs allow direct read/write into GPU driver memory eliminating need for additional `memcpys`. The CPU after the copy operation does not stall while the transfer takes place and can continue on to process the next frame. However, downloads and uploads still involve GPU context switch and cannot be done in parallel with the GPU processing or drawing. Multiple PBOs can potentially speed up the transfers. A ping pong version is shown in Figure 3.

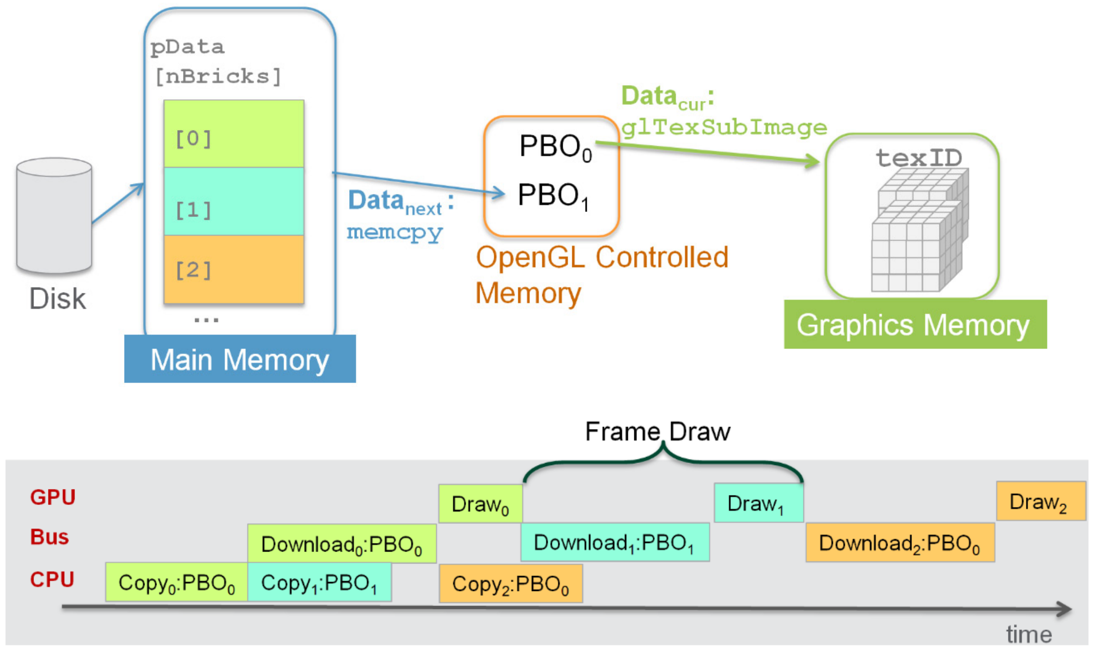

Figure 3. CPU Asynchronous Downloads with Ping Pong PBOs

Initialization

```
GLuint pboIds[2];
glGenBuffersARB(2, pboIds); //Allocate 2 PBO’s
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pboIds[0]);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB,width*height*sizeof(GLubyte*
nComponents),0,GL_STREAM_DRAW_ARB);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pboIds[1]);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB,width*height*sizeof(GLubyte)
*nComponents,0,GL_STREAM_DRAW_ARB);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
//TODO – Same texture initialization from “texture streaming” section
```

Draw

```
static unsigned int curPBO = 0;
glBindTexture(GL_TEXTURE_2D,texId);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pboIds[curPBO]); //bind pbo
//Copy pixels from pbo to texture object
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,xdim,ydim,GL_RGBA,
GL_UNSIGNED_BYTE,0);

//bind next pbo for app->pbo transfer
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, m_ pboIds[1-curPBO]);
//bind pbo
//to prevent sync issue in case GPU is still working with the data
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB,
xdim*ydim*sizeof(GLubyte)*nComponents, 0, GL_STREAM_DRAW_ARB);
GLubyte* ptr = (GLubyte*)glMapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB,
GL_WRITE_ONLY_ARB);
assert(ptr);
memcpy(ptr,pData[m_curBrick],width*height);
glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB,0);
curPBO = 1-curPBO;
//TODO – Call drawing code here
```

GPU ASYNCHRONOUS TRANSFERS WITH QUADRO DUAL COPY ENGINES

The copy engine featured in Quadro solutions provides real GPU-asynchronous texture downloads. Texture data can be downloaded or uploaded in parallel with 3D rendering. As shown in Figure 4, supported Quadro solutions[1] add an additional DMA engine making it now possible to overlap download, processing, and readback. To take advantage of this, one thread (channel) is used for rendering, one is used for download and the third is used for upload, and all transfers are done via PBOs. When partitioned this way, the render thread will run on the graphics engine and the transfer threads on the copy engines in parallel and completely asynchronous. These are fully functional GL contexts so that non-DMA commands can be issued in the transfer threads but will time slice with the rendering thread. Copy engines can also handle format conversions and swizzling for same data types without CPU intervention, in contrast to previous hardware constraints where the input data formats had to be GPU native. Figure 5 shows the end-to-end frame time amortized over 3 frames for a time sequence. It is seen how the current frame download (t_1) is overlapped with render of previous frame (t_0) and CPU `memcpy` of next frame (t_2).

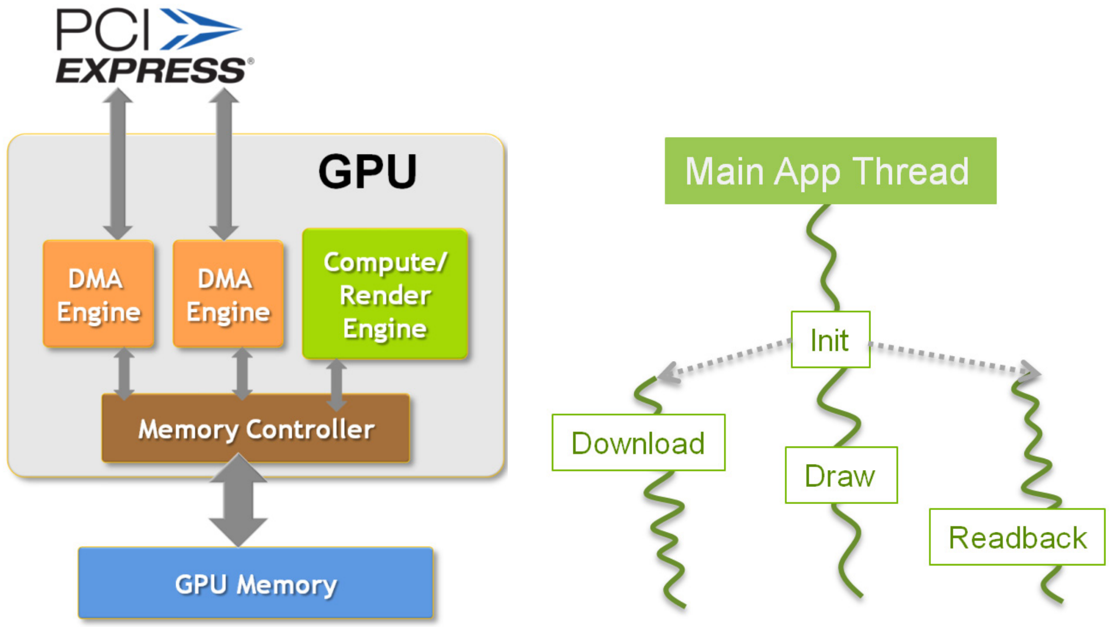

Figure 4. Quadro Dual Copy Engine Block Diagram and Application Layout

[1] Quadro 4000, Quadro 5000, and Quadro 6000 only

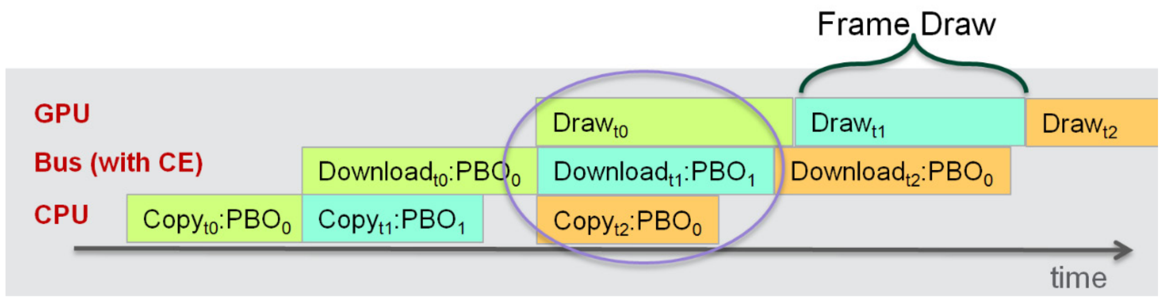

Figure 5. GPU Asynchronous Transfers with Dual Copy Engines

Synchronization

OpenGL rendering commands are assumed to be asynchronous. When a `glDraw*` call is issued, it is not guaranteed that the rendering is done by the time the call returns. When sharing data between OpenGL contexts bound to multiple CPU threads, it is useful to know that a specific point in the command stream was fully executed. This is managed by sync objects as part of the ARB_sync [3] mechanism in OpenGL 3.2. Sync objects can be shared between different OpenGL contexts and so a sync object created in a context or thread can be waited by another context.

A type of sync object – **fence** is a token created and inserted in the command stream (in a non signaled state) and when executed changes its state to signaled. Due to the in-order nature of OpenGL, if the fence is signaled, then every command issued before the fence was also completed. Cooperating threads can wait for the fence to become signaled and resume operation similar to using mutexes in CPU threads. In a download-process-readback scheme, the processing waits on the fence inserted after texture download. Similarly, the readback waits on the fence inserted by the main thread after render.

Multi-Threaded Downloads

Multiple textures can be used to ensure sufficient overlap such that downloads and readbacks are kept busy while the GPU is rendering with a current texture. Since the textures are shared between multiple contexts, synchronization primitives like events and fences are created per texture. The following snippets illustrate the steps for streaming 3D textures.

Shared Objects

```
GLsync fence[numBufers]; //multiple textures to ensure overlap
GLuint tex[numBufers];
HANDLE continue[numBufers], done[numBufers]; //events
HDC hDC;
HGLRC downloadRC, drawRC;
```

Main Draw

```
//Get 2 OpenGL contexts from same DC
downloadRC = wglCreateContext(hDC);
drawRC = wglCreateContext(hDC);
//Before any loading, share textures between contexts
wglShareLists(downloadRC, drawRC);
glGenTextures(numBuffers, tex);
for (i=0;i<numBuffers;i++) {
   continue[i] = CreateEvent(NULL, FALSE, FALSE, NULL);
   done[i] = CreateEvent(NULL, FALSE, FALSE, NULL);
}
//Create download thread from the main render thread
HANDLE downloadThread = CreateThread(NULL, NULL, downloadFunc,
downloadData, NULL, NULL);
int curRender = 0;
while (!done) {
   WaitForSingleObject(done[curRender]); //Wait for fence creation
   glWaitSync(fence[curRender], 0, 0);
   //At this point, the texture we want to use for render is ready
   Render();//Draw function calls glBindTexture(tex[curRender])
  glDeleteSync(fence[curRender]);
  SetEvent(continue[curRender]); //Download can start filling this tex
   curRender = (curRender+1)%numBuffers;
}
//Cleanup
WaitForAndDestroyThread(downloadThread);
glDeleteTextures(numBuffers, tex); //delete textures
for (i=0;i<numBuffers;i++) {
   CloseHandle(continue[i]); continue[i] = NULL; //Destroy the 2 events
   CloseHandle(done[i]); done[i] = NULL;
}
wglDeleteContext(downloadRC);
wglDeleteContext(drawRC);
```

Download Thread

In the download thread, a fence is inserted after the textures are updated using the `TexSubImage` call and the main thread is notified to wait for this fence completion before using that texture for the drawing. The mapping, CPU `memcpy`, and unmapping proceed in parallel with the render thread.

```
DWORD WINAPI downloadFunc (LPVOID param) {
  ThreadData *threadData = (ThreadData*) param;
  wglMakeCurrent(hDC, downloadRC);
  // ALLOCATE AND INIT PBO'S (CODE FROM PREVIOUS SECTIONS)
  while (1) {
     static unsigned int curPBO = 0, curDownload =0;
      WaitForSingleObject(continue[curDownload]);
      // Renderer has signaled that is has finished using this texture
      glBindTexture(GL_TEXTURE_3D,texId[curDownload]);
      glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo[curPBO]);
     //Copy pixels from pbo to texture object
```

```
      glTexSubImage3D(GL_TEXTURE_3D,0,0,0,0,xdim,ydim,zdim,GL_LUMINANCE,
GL_UNSIGNED_BYTE,0);
       fence[curDownload] = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE,0)
       //Tell main render fence is now valid to use.
       SetEvent(done[curDownload]);
      curDownload = (curDownload+1)%numBuffers;
      //APP->PBO transfer
      glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo[1-curPBO]);
      //prevent sync issue in case GPU is still working with the data
      glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB,xdim*ydim*zdim*sizeof(G
Lubyte), 0, GL_STREAM_DRAW_ARB);
      GLubyte* ptr = (GLubyte*)
glMapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, GL_WRITE_ONLY_ARB);
      assert(ptr);
      memcpy(ptr,m_pVolume[m_curTimeStep],m_w*m_h*m_d);
      glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB);
      glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB,0);
      curPBO = 1-curPBO;
   } // while
   // DELETE PBO'S (CODE FROM PREVIOUS SECTIONS)
   wglMakeCurrent(NULL, NULL);
return TRUE;
}
```

Readback with Quadro Dual Copy Engines

An additional readback thread is created and a fence is inserted in the main thread after the rendering and asynchronous `ReadPixels`. The readback thread waits on this fence before it starts mapping the buffers. Multiple PBOs can be used to alternate between `ReadPixels` and copy into system memory in the readback thread.

Shared Objects

```
GLsync doneReadFence[numReadbackBuffers]; //multiple sync for overlap
//event to signal end of render+readback and to start the render
HANDLE doneRead[numReadbackBuffers], startRead[numReadbackBuffers];
HGLRC readbackRC;
GLuint readbackPBO[numReadbackBuffers]; //for readpixels
```

Main Render

```
int curRender = 0, curRead =0; //the buffer for async readpixels
while (!done) {
   WaitForSingleObject(done[curRender]); //Wait for fence creation
   glWaitSync(fence[curRender], 0, 0);
   Render();//Draw function, glBindTexture(tex) is called inside
   glDeleteSync(fence);
   WaitForSingleObject(startRead[curRead]); //Wait for readback
   //Bind readbackPBO[curRead] and do async glReadPixels here
   doneReadFence[curRead]=glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE,0);
   SetEvent(doneRead[curRead]); // fence is ready for readback to wait
}
```

Readback Thread

```
DWORD WINAPI readbackFunc (LPVOID param) {
   ThreadData *threadData = (ThreadData*) param;
   wglMakeCurrent(hDC, readbackRC);
   << ALLOCATE AND INIT PBO'S (CODE FROM PREVIOUS SECTIONS) >>
   static unsigned int curMap = 0;
   while (1) {
        WaitForSingleObject(doneRead[curMap]); //Wait for render fence
        glWaitSync(doneReadFence[curMap],0, GL_TIMEOUT_IGNORED);
       //At this point, main thread has finished doing readpixels
       glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, readbackPBO[curMap]);
       GLubyte* ptr = (GLubyte*) glMapBufferARB(GL_PIXEL_PACK_BUFFER_ARB,
GL_READ_ONLY);
       assert(ptr);
       << process Pixels eg memcpy here using ptr >>
       glUnmapBufferARB(GL_PIXEL_PACK_BUFFER_ARB);
       glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB,0);
       glDeleteSync(doneReadFence[curMap]);
       SetEvent(startRead[curMap]); //main thread can start readback now
       curMap = (curMap+1)%numReadbackBuffers;
    } // while
   << DELETE PBO'S (CODE FROM PREVIOUS SECTIONS) >>
   wglMakeCurrent(NULL, NULL);
   return TRUE;
}
```

Note: Having two separate threads running on a Quadro graphics card with the consumer NVIDIA® Fermi architecture or running on older generations of graphics cards the data transfers will be serialized resulting in a drop in performance.

RESULTS

The following results (Figure 6) show a download-processing-readback pipeline streaming HD (8 MB per frame) and 4K (32 MB per frame) images with varying processing times (10 ms, 20 ms, and 30 ms) comparing the four methods listed.

- Synchronous
- CPU asynchronous with PBO's
- GPU asynchronous using the copy engine for download
- Static or cached case where no streaming is involved

It is seen that the performance measured by fps is almost the same between HD and 4K video streaming for all the processing times despite the 4× data size that is downloaded for the 4K images. This shows that download and processing is happening truly asynchronously on the GPU using Quadro copy engines.

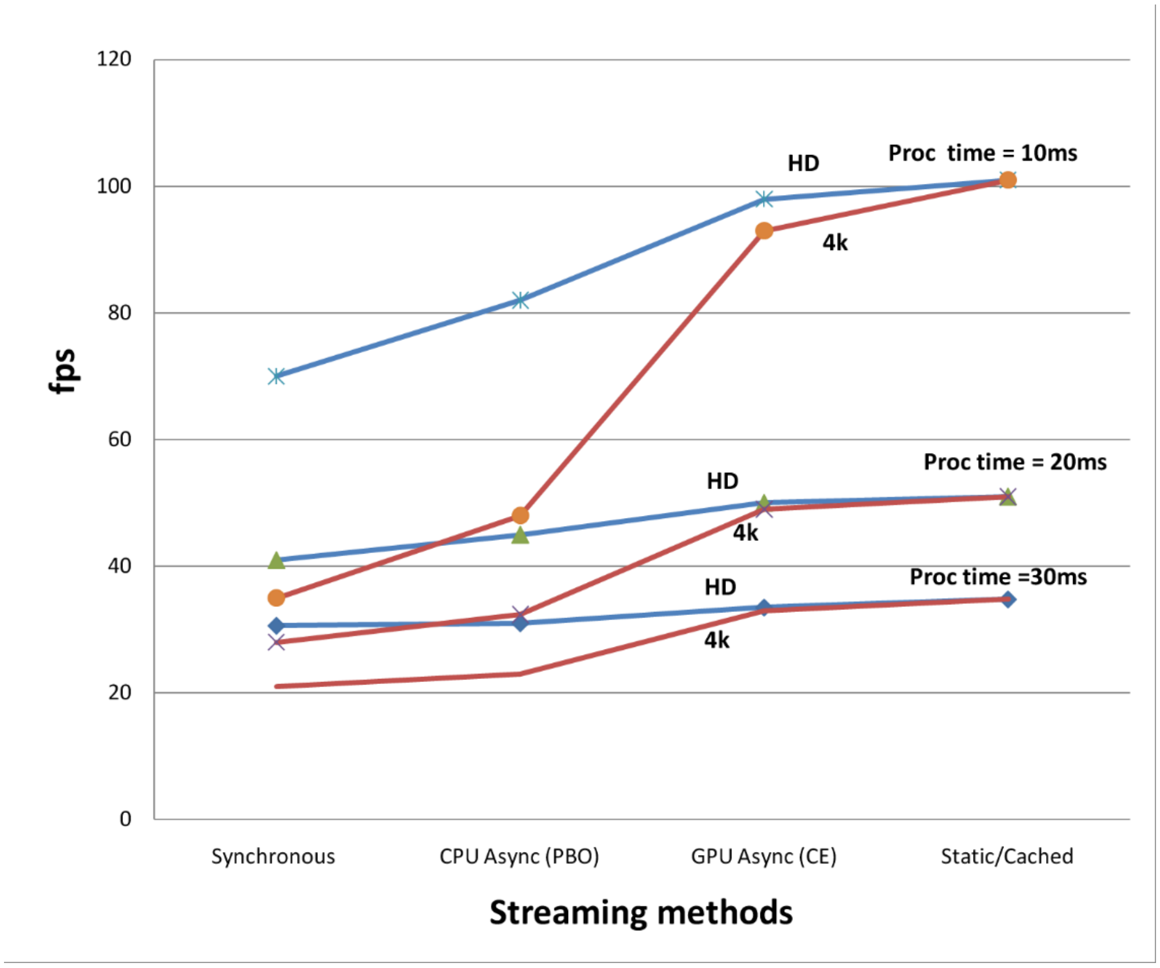

Figure 6. Download Process Performance Comparison

REFERENCES

[1] Fermi White paper -
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[2] OpenGL PBO Specification
http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt

[3] OpenGL ARB_Sync Specification
http://www.opengl.org/registry/specs/ARB/sync.txt