29 Fermi Asynchronous Texture Transfers

Shalini Venkataraman

29.1 Introduction

Many real-world graphics applications need to transfer textures efficiently in and out of the GPU memory in the form of 2D images, 2.5D terrains, or 3D volumes and their time-varying counterparts. In the pre-Fermi generation of NVIDIA hardware, any data transfer would stall the GPU from rendering because the GPU had a single hardware execution thread that could either execute transfers or rendering. The OpenGL pixel buffer object (PBO) [ARB 04] provides a mechanism to optimize transfers, but it is CPU asynchronous in that it allows for concurrent CPU processing while the GPU performs uploads and downloads. However, the GPU is still blocked from rendering OpenGL commands while the actual data transfers occur. As discussed in Chapter 28, many applications go a step further to use multiple threads for resource preparation such that the GPU is always kept busy. At the hardware-execution level, however, the GPU will end up serializing the transfer and the draw command queues.

This chapter explains how this limitation is overcome by the *copy engine hardware* found in the NVIDIA Fermi architecture [NVIDIA 10] generation and later GPUs. A copy engine is a dedicated controller on the GPU that performs DMA transfers of data between CPU memory and GPU memory independent of the graphics engine (Figure 29.1). Each copy engine allows one-way-at-a-time bidirectional transfer. The NVIDIA Fermi GeForce and the low-end Quadro cards[1] have one copy engine such

[1] Quadro 2000 and below.

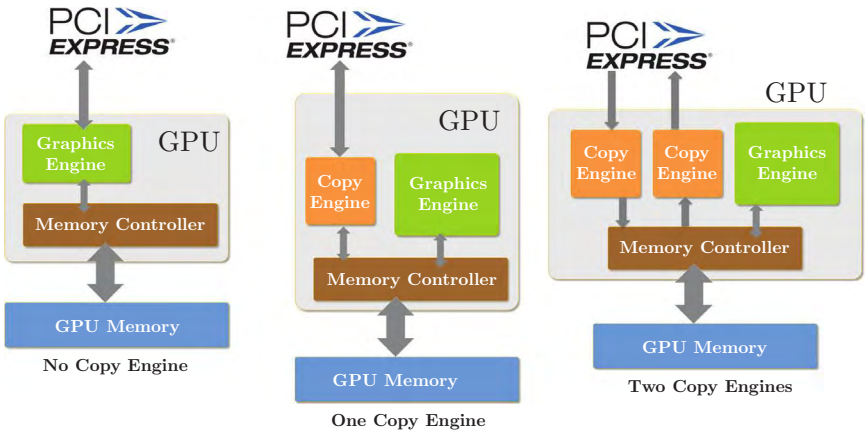

Figure 29.1. Copy engine and graphics engine layout for various GPUs.

that unidirectional transfers can be concurrently performed with rendering, allowing for two-way overlap. The Quadro mid-higher level cards[2] have two copy engines so that bidirectional transfers can be done in parallel with rendering. This three-way overlap means that the current set of data can be processed while the previous set is downloaded from the GPU and the next set is uploaded.

Figure 29.1 shows the block diagrams comparing GPUs with (left) no copy engine, where the graphics engine handles transfers and drawing; (center) a single copy engine that handles transfers in both directions; and finally, (right) two copy engines, each dedicated to transfers in a single direction.

Some examples for overlapped transfers include the following:

- **Video or time-varying geometry and volumes.** This includes transcoding, visualizing time-varying numerical simulations, and scanned medical data such as 4D ultrasound.

- **Remoting graphics.** Powerful GPU servers are used for offscreen rendering, and the results are downloaded to the server's main memory, which is sent over the network to thin clients such as phones and tablets.

- **Parallel rendering.** When a scene is divided and rendered across multiple GPUs and the color and depth are read back for composition, parallelizing readback will speed up the pipeline. This is likewise the case for a sort-first implementation, where at every frame, the data have to be streamed to the GPU based on the viewpoint.

[2] Quadro 4000, 5000, 6000, and Quadro Plex 7000.

- **Data bricking for large images, terrains, and volumes.** Bricks or LODs are paged in and out as needed in another thread without disrupting the rendering thread.
- **OS cache.** Operating systems can page in and out textures as needed, eliminating shadow copies in RAM.

This chapter starts by covering existing methods for texture transfers, such as synchronous and CPU-asynchronous methods like PBOs, and explains their limitations. Then, GPU-asynchronous methods using the Fermi copy engines are introduced where transfers can occur concurrently with GPU rendering. Achieving this parallelism on the GPU hardware requires application restructuring into multiple threads with a context per thread and use of OpenGL fences to manage the synchronization. Finally, it concludes with results showing the speedup achieved for various data sizes, application characteristics, and GPUs with different overlap capabilities. The complete source code that is used to generate the results is available on the OpenGL Insights website, www.openglinsights.com.

29.2 OpenGL Command Buffer Execution

Before diving into transfers, I will lay the groundwork for understanding OpenGL command buffers, specifically the interplay between the drivers and the OS and how it is all finally executed by the graphics hardware. I use GPUView [Fisher and Pronovost 11], a tool developed by Microsoft and available as part of the Windows 7 SDK. GPUView will allow us to see, as a function of time, the state of all the context-specific CPU queues as well as the queue for the graphics card.

Figure 29.2 shows the trace for an OpenGL application with multiple contexts. The application thread continuously submits work to the CPU command queue from where the OS uploads to the GPU hardware queue. The CPU command queue exists per OpenGL context. OpenGL calls for a context are batched in a list of commands for that context, and when enough commands are built up, they are flushed to the CPU command queue. Each CPU command queue has its own color so that it is easy to see which queue the graphics hardware is currently working on. Periodically, when there is room, the graphics scheduler will add a task from the CPU context command queue onto the GPU hardware queue. The GPU hardware queue shows the work that is currently being processed by the graphics card and queue of tasks it is working on. There are two GPU hardware queues in this example showing some of the packets that are processed in parallel and others in serial. The arrows follow a series of command packets as they are flushed to the CPU command queue, and then wait in the queue and are executed on the GPU. When the GPU has finished executing the final packets, the CPU command queue can return and the next frame begins. Throughout this chapter, we use GPUView traces to understand what happens under the hood for the various transfer approaches.

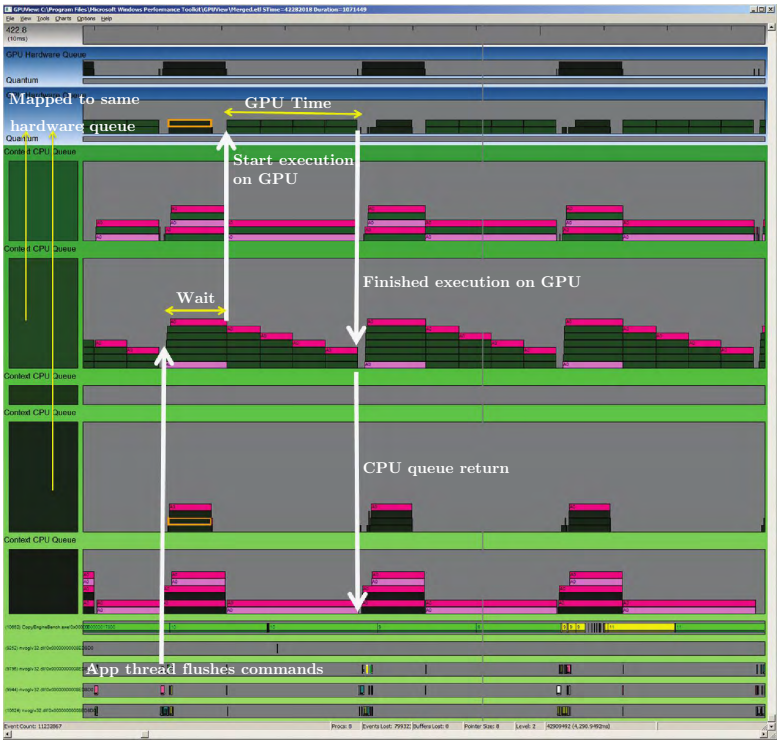

Figure 29.2. Screenshot of GPUView.

29.3 Current Texture Transfer Approaches

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

- Copy. The CPU cycles spent for data conversion, if any, to native GPU formats; use `memcpy` from application space to driver space for upload and vice versa for download.
- Upload. The time for the actual data transfer on the PCI-e bus from host to GPU.
- Process. The GPU cycles for rendering or processing.
- Download. The time for the data transfer from the GPU back to host.

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

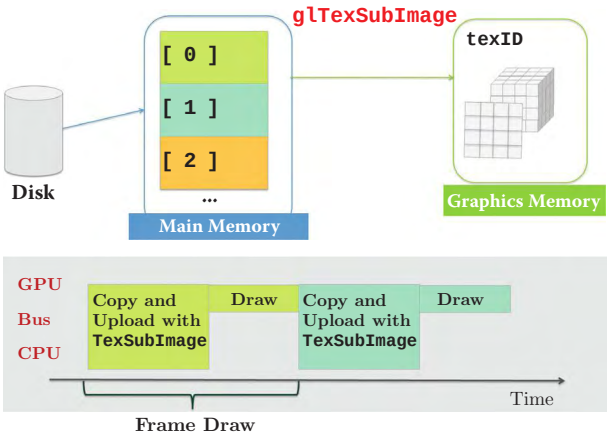

Figure 29.3. Synchronous texture uploads with no overlap.

29.3.1 Synchronous Texture Transfers

For simplicity, we start by analyzing an upload-render pipeline. The straightforward upload method for textures is to call `glTexSubImage`, which uses the CPU for copying data from user space to driver pinned memory and blocks it during the subsequent data transfer on the bus to the GPU. Figure 29.3 illustrates the inefficiency of this method as the GPU is blocked during the CPU copy. The corresponding

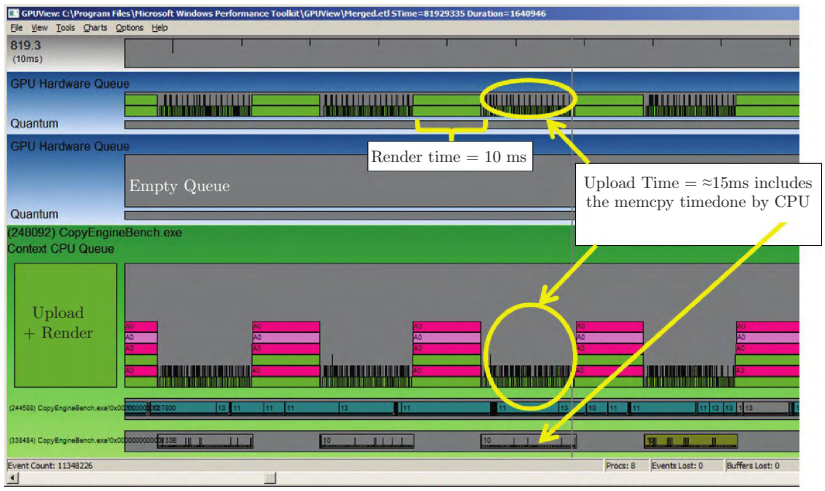

Figure 29.4. GPUView timeline showing synchronous texture uploads.

GPUView trace is shown in Figure 29.4, and shows that upload and render are handled sequentially and the additional GPU hardware queue is unused. This graph also shows that the `memcpy`s are in fact interspersed with the transfer, causing the spikes and gaps in the CPU and GPU command queues. Ideally, we would like the execution timeline of a packet to be solid until completion to show that the GPU is kept fully busy, as is shown by the render.

29.3.2 CPU Asynchronous Texture Transfers

The OpenGL PBO [ARB 04] 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. In this case, `glTexSubImage`, `glReadPixels`, and `glGetTexImage` operate with little CPU intervention. PBOs allow direct access into GPU driver-pinned memory, eliminating the need for additional copies. After the copy operation, the CPU does not stall while the transfer takes place and continues on to process the next frame. Ping-pong PBOs can further increase parallelism where one PBO is mapped for `memcpy` while the other feeds to the texture.

Figure 29.5 shows the workflow along with the timeline for the same upload-render workflow, and Listing 29.1 shows the code snippets to map the PBOs and populate or "unpack" them. Multiple threads can be used to feed the data for transfer (see Chapter 28); however, at the hardware-execution level, there is only one thread of execution causing transfers to be serialized with the drawing as shown in the GPUView trace in Figure 29.6. The trace also shows solid lines for upload and render, signifying 100% GPU utilization without any CPU intervention.

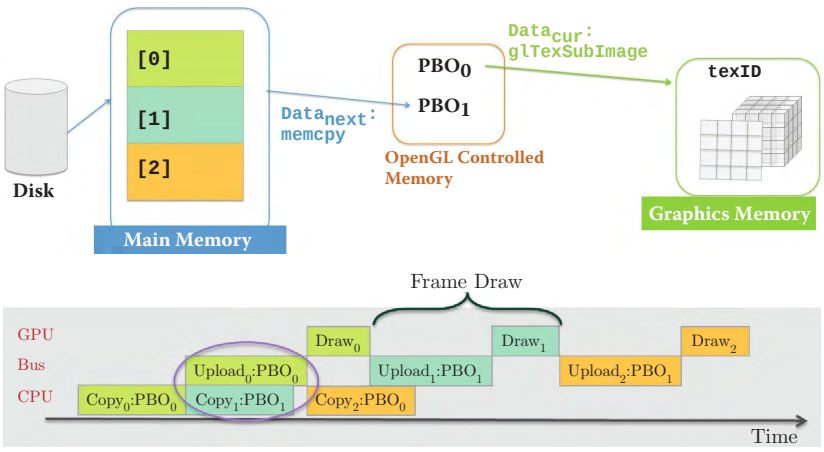

Figure 29.5. CPU asynchronous uploads with ping-pong PBOs.

```
GLuint pbo[2]; // The ping-pong pbo's
unsigned int curPBO = 0;
// Bind current pbo for app->pbo transfer
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo[curPBO]);
GLubyte *ptr;
ptr = (GLubyte *)glMapBufferRange(GL_PIXEL_UNPACK_BUFFER_ARB, 0, size, ↩
    GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);
memcpy(ptr, pData, width * height * sizeof(GLubyte) * nComponents);
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
glBindTexture(GL_TEXTURE_2D, texId);
// Bind next pbo for upload from pbo to texture object
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo[1 - curPBO]);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGBA, GL_UNSIGNED_BYTE ,↩
    0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
glBindTexture(GL_TEXTURE_2D, 0);
curPBO = 1 - curPBO;
```

Listing 29.1. CPU asynchronous upload using ping-pong PBOs.

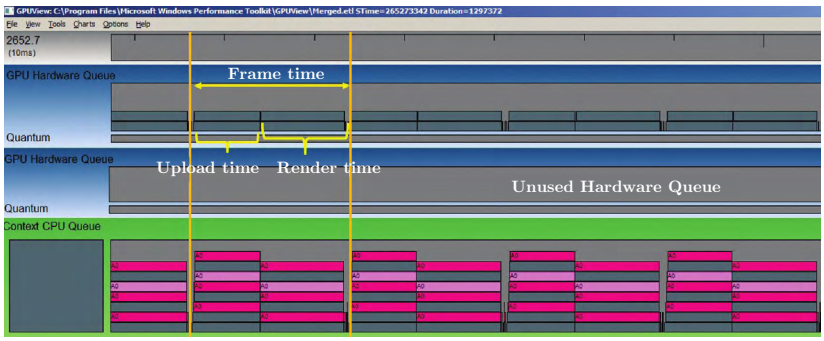

Figure 29.6. CPU asynchronous transfers using PBOs in the same thread as render.

29.4 GPU Asynchronous Texture Transfers

The GPUView diagrams show that only one GPU hardware queue was used and the additional GPU hardware queues signifying tasks for the copy engines are empty. The copy engines are not used by default, as there is some additional overhead in initialization and synchronization, which is not warranted for small transfers as the results later show (Section 29.6). In order to trigger the copy engine, the application has to provide a heuristic to the drivers, and it does this by separating the transfers in a separate thread. When the transfers are partitioned this way, the GPU scheduler ensures that the OpenGL commands issued in the render thread will map and run on the graphics engine and the commands in the transfer threads on the copy engines in parallel and completely asynchronously. This is what I refer to as GPU asynchronous transfers.

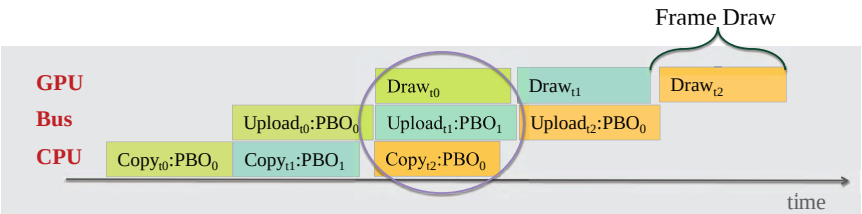

Figure 29.7. GPU asynchronous transfer showing overlap of upload and draw.

Figure 29.7 shows the end-to-end frame time amortized over three frames for an upload-render case. The current frame upload ($t1$) is overlapped with the render of the previous frame ($t0$) and CPU `memcpy` of the next frame ($t2$). Figure 29.8 shows the GPUView trace on a Quadro 6000 card where three separate GPU command queues are exposed, although the download queue is currently unused. The upload here is hidden by the render time.

So far in this chapter, I have touched mostly on the upload-render case for the sake of simplicity in illustration. However, the same principles apply for a render-download and a upload-render-download overlap case.

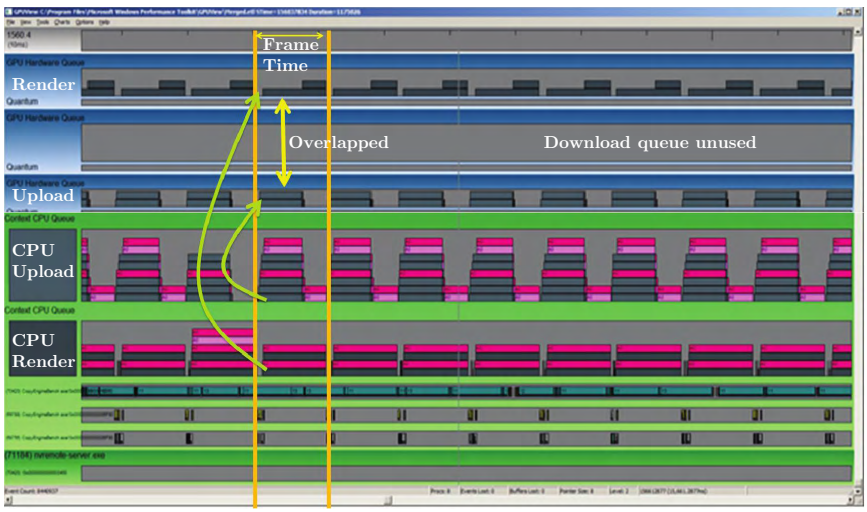

Figure 29.8. GPUView timing diagram showing the overlap of upload and draw at the GPU hardware queues.

29.5 Implementation Details

29.5.1 Multiple OpenGL Contexts

A separate thread with its associated OpenGL context is created for each stage applicable in the pipeline: upload, render, and download. Figure 29.9 shows the schematic for a upload-render pipeline. The upload thread is responsible for streaming source data from main memory into a shared texture that the render thread subsequently accesses for drawing.

Likewise, as shown in Figure 29.10, the render thread renders to a framebuffer-object attachment that the download thread is waiting on to transfer back to main memory. The offscreen rendering is done via FBOs [ARB 08]. All the transfers are still done using PBOs, as was explained in Section 29.3.2. Multiple textures for both source and destinations are used to ensure sufficient overlap such that uploads and downloads are kept busy while the GPU is rendering to or with a current texture.

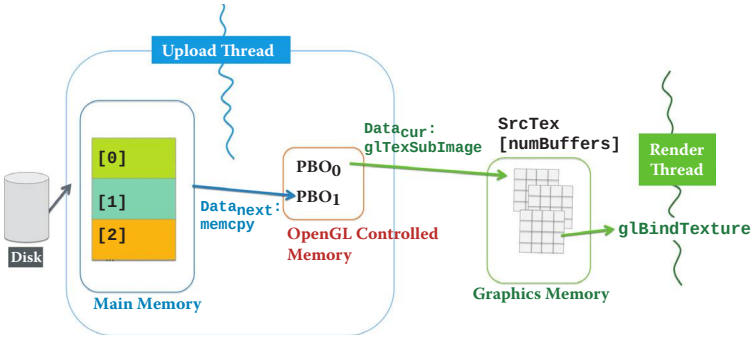

Figure 29.9. Schematic showing upload and render threads with shared textures.

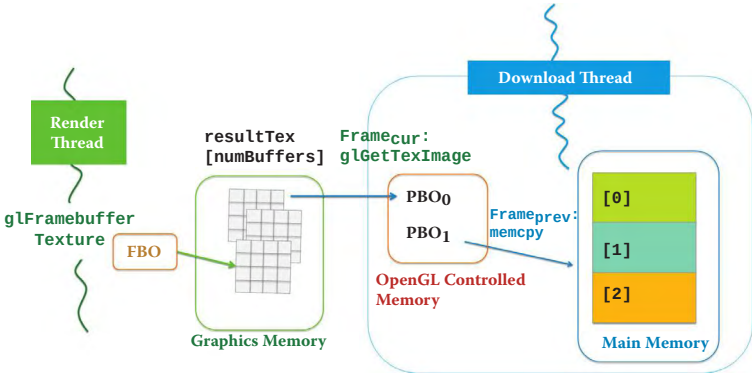

Figure 29.10. Render and download threads accessing shared offscreen textures.

The textures are shared between multiple contexts using `WGL_ARB_create_context` [ARB 09c] on Windows and `GLX_ARB_create_context` on Linux [ARB 09b]. Threads are then spawned to handle the upload, render, and download processes. To manage concurrent access from threads to shared textures, synchronization primitives like CPU events and GPU fences are created per texture.

29.5.2 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. For example, the render thread needs to know when the texture upload has completed in order to use the texture. This handshaking is managed by synchronization objects as part of the `GL_ARB_Sync` mechanism [ARB 09a]. Synchronization objects can be shared between different OpenGL contexts, and an object created in a context can be waited on by another context. Specifically, we use a fence, which is a type of synchronization object that is created and inserted in the command stream (in a nonsignaled state) and when executed, changes its state to signaled. Due to the in-order execution of OpenGL, if the fence is signaled, then all commands issued before the fence in the current context have also been completed. In an upload-render scheme, as shown in Figure 29.11, the render waits on the fence, `endUpload`, inserted after texture upload to start the drawing while the upload waits on the `startUpload` fence, which the render queues after the drawing. These fences are created per texture. Corresponding CPU events are used to signal the GPU fence creation to avoid busy waiting. For example, the `endUploadValid` event is set by the upload thread to signal to the render thread to start the `glWaitSync` for the `endUpload` fence before rendering to that texture.

Likewise, in a render-download scheme, as shown in Figure 29.12, the download waits on the fence `startDownload` inserted after render to start reading from the texture, and the render waits for download to complete before using the `endDownload` fence.

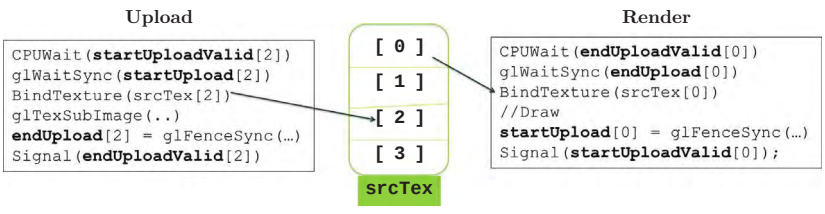

Figure 29.11. Upload thread produces to `srcTex[2]` while render thread consumes from `srcTex[0]`.

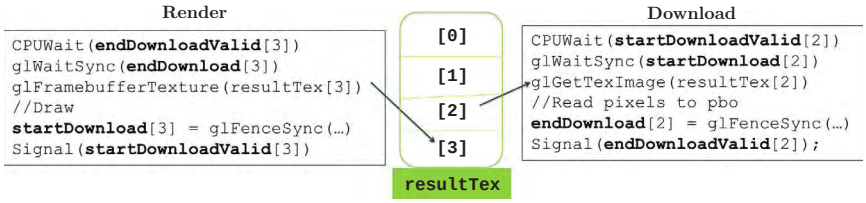

Figure 29.12. Render thread produces resulting image to `resultTex[3]` while download thread consumes from `resultTex[2]`.

29.5.3 Copy Engine Considerations

An OpenGL context attached to a copy engine is a fully functional GL context so that non-DMA commands can be issued in the transfer threads. However, some of these calls may time-slice with the rendering thread, causing us to lose parallelism. In the event that the driver has to serialize calls between the transfer and render context, it generates a debug message, "Pixel transfer is synchronized with 3D rendering," which the application can query using the `GL_ARB_debug_output` extension (Chapter 33). Another limitation of Fermi's copy engine is that it is only limited to pixel transfers and not vertex transfers.

When FBOs are used in conjunction with copy engines, there is some overhead in doing the FBO validation during texture and renderbuffer attachments. For this reason, `glGetTexImage` is the prefered path for download rather than using `glReadPixels` to read from a renderbuffer or texture. Lastly, the optimal number of shared textures should be set based on the ratio of render time to transfer time; this requires some experimentation. When both the times are balanced, a double-buffered texture is sufficient.

29.6 Results and Analysis

The following tests were done on a Dell T7500 Workstation with Intel Xeon Quad Core E5620 at 2.4GHz and 16GB RAM. The test boards used were NVIDIA Quadro 6000 and NVIDIA GeForce GTX 570 attached to the PCI-e x16 slot with Quadro and Forceware 300.01 drivers. As the PCI transfer rates are highly sensitive to the chipset, a workstation-class motherboard is used to achieve the best results.

The results compare the performance gain achieved on the GeForce and Quadro with various texture sizes for applications that range from transfer-heavy to balanced to render-heavy. The horizontal axis shows the ratio of render to transfer time on a logarithmic scale. The baseline for each series is the time taken for CPU-asynchronous transfers (Section 29.3.2) on that card.

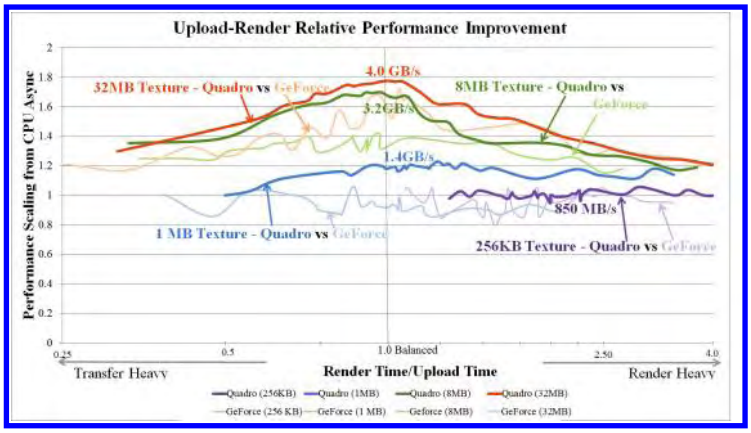

Figure 29.13. Comparing upload-render performance improvement.

For the upload-render overlap in Figure 29.13, the performance improvement on Quadro is higher than GeForce (lighter colored lines) for all data sizes. The overlap performance on the Quadro is also more deterministic as compared to the GeForce, which shows a lot of jitter between runs. It is also seen that for texture

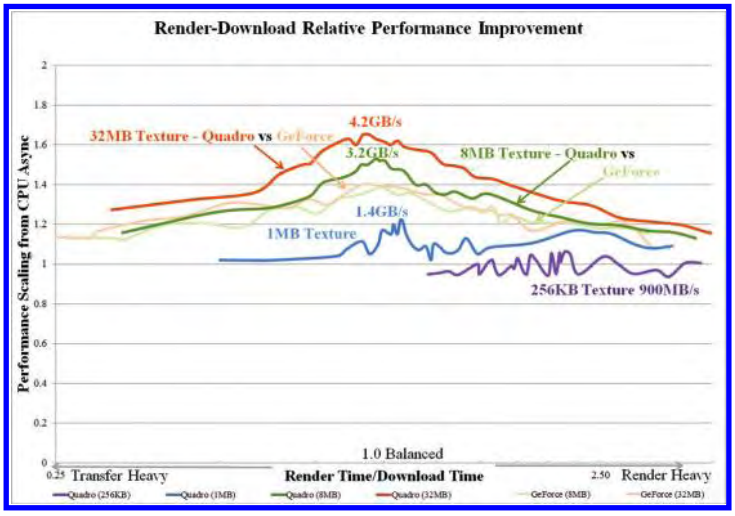

Figure 29.14. Comparing render-download performance improvement.

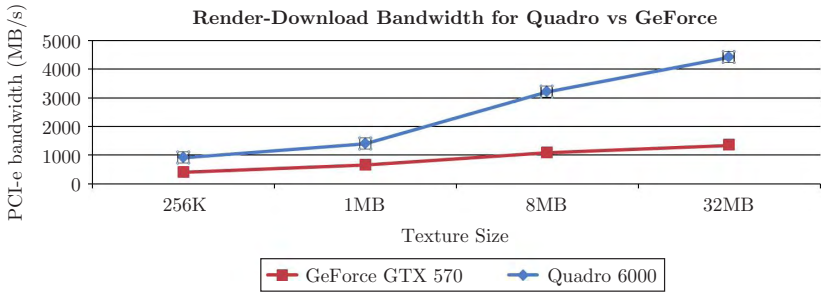

Figure 29.15. Quadro download performance is 3X higher than GeForce.

sizes less than 1MB invoking the copy engine shows very little gain. There are only significant gains, 50% or more, which warrant the extra programming effort, from using the copy engine for textures bigger than 1MB. Using the copy engine also favors applications that are balanced between transfer and rendering time. For example, we see close to linear scaling at around 1.8X for the 32MB texture in the balanced case, which sustains a bandwidth of 4GB/s.

Figure 29.14 shows the performance improvement in the render-download overlap case. The bandwidth for the peaks are comparable to the upload case. For

Figure 29.16. GeForce GTX 570 trace for upload, download, and bidirectional overlap.

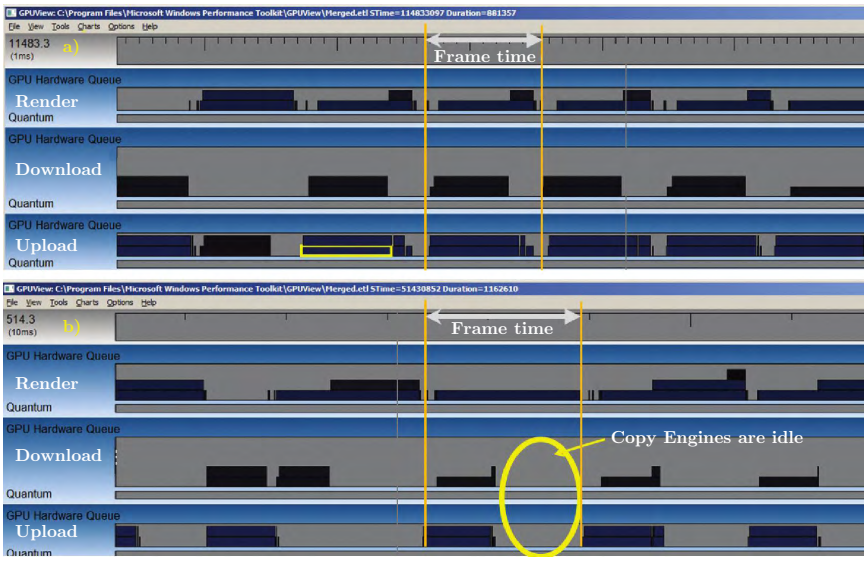

Figure 29.17. Ideal case of bidirectional transfers where upload, render, and download take the same time (top). Render-bottlenecked applications where the copy engines are idle 50% of the time (bottom).

the GeForce, although using the copy engine gives a 40% improvement in render-download overlap performance for the 8MB and 32MB texture, the absolute download performance is significantly better on the Quadro, peaking at more than 3X, as shown in Figure 29.15.

The behavior on GeForce cards is verified by the GPUView trace in Figure 29.16 (top), which shows the upload-render overlap working as expected, and Figure 29.16 (middle), which shows the render-download overlap but with much larger download times for the same texture size. The GeForce boards are only optimized for the upload path, as that is the use case for many consumer applications and games that do bricking and paging. Figure 29.16 (bottom) shows how doing bidirectional transfers on the GeForce serializes the upload and download, as they now end up sharing one copy engine.

Figure 29.17 shows the bidirectional transfer overlap with rendering that is available on mid- to high-end Quadros. Figure 29.17 (top) shows the ideal case where there is maximum overlap and very little idle time in any of the three queues. Figure 29.17 (bottom), on the other hand, shows a render-heavy case where the copy engines are idle half the time.

29.6.1 Previous Generation Architecture

Figure 29.18 shows the same application running on a Quadro 5800 based on the GT 200 chip and the NVIDIA Tesla architecture. The number of threads and the CPU command queues remain the same, but at the hardware level, the GPU scheduler serializes the different command queues, and therefore, no overlap is seen. The length of the CPU command queue for the download shows the latency from the time when the download commands were queued by the application to when the GPU has completed execution.

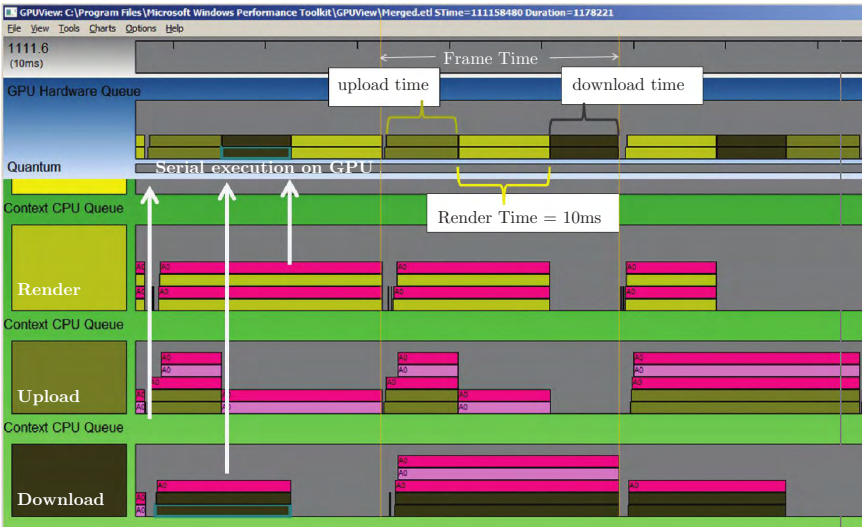

Figure 29.18. The same multithreaded application on a Tesla card (Quadro 5800) is serialized in the GPU hardware queue.

29.7 Conclusion

The GPU asynchronous texture-transfer mechanism enabled by the copy engines in NVIDIA's Fermi and above generation provides for the best transfer performance for applications that (1) are balanced in terms of render and transfers and (2) have significantly large data (more than 1MB) to transfer. I also illustrated how to use multiple threads and synchronization primitives to invoke the copy engines using a 2D texture streaming example. Results with the aid of GPUView show the two-way overlap that takes place on the GeForce and low-end Quadros and three-way overlap on the higher-end Quadros. This example can be easily applied to a terrain-paging system where the different LODs are uploaded concurrently with the rendering

depending on view parameters. Other extensions could include large volumetric rendering by bricking 3D textures for applications in medical imaging, oil and gas, and numerical simulations. The render-download usage case maps directly to server-based OpenGL rendering that is rampant with the proliferation of thin clients like phones and tablets.

Bibliography

[ARB 04] OpenGL ARB. "OpenGL PBO Specification." http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt, 2004.

[ARB 08] OpenGL ARB. "OpenGL FBO Specification." http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt, 2008.

[ARB 09a] OpenGL ARB. "OpenGL ARB_Sync Specification." http://www.opengl.org/registry/specs/ARB/sync.txt, 2009.

[ARB 09b] OpenGL ARB. "OpenGL GLX_ARB_create_context." http://www.opengl.org/registry/specs/ARB/glx_create_context.txt, 2009.

[ARB 09c] OpenGL ARB. "OpenGL WGL_ARB_create_context." http://www.opengl.org/registry/specs/ARB/wgl_create_context.txt, 2009.

[Fisher and Pronovost 11] Mathew Fisher and Steve Pronovost. "GPUView." http://graphics.stanford.edu/~mdfisher/GPUView.html, 2011.

[NVIDIA 10] NVIDIA. "Fermi White Paper." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.