## Title
Distributed Texture Memory in a Multi-GPU Environment

## Authors
Moerschell, Adam
Owens, John D.

**Publication Date**
2006

**DOI**
10.1145/1283900.1283905

Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

# Distributed Texture Memory in a Multi-GPU Environment

Adam Moerschell and John D. Owens

University of California, Davis

**Abstract**
*In this paper we present a consistent, distributed, shared memory system for GPU texture memory. This model enables the virtualization of texture memory and the transparent, scalable sharing of texture data across multiple GPUs. Textures are stored as pages, and as textures are read or written, our system satisfies requests for pages on demand while maintaining memory consistency. Our system implements a directory-based distributed shared memory abstraction and is hidden from the programmer in order to ease programming in a multi-GPU environment. Our primary contributions are the identification of the core mechanisms that enable the abstraction and the future support that will enable them to be efficient.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture; C.1.4 [Parallel Architectures]: Distributed Architectures.

## 1. Introduction

In recent years, both the performance and programmability of graphics processors has dramatically increased, allowing the GPU to be used for both demanding graphics workloads as well as more general-purpose applications. However, the demand for even more compute and graphics power continues to increase, targeting such diverse application scenarios as interactive film preview, large-scale visualization, and physical simulation.

These complex applications would benefit from scalable graphics systems that allow users to add additional hardware to a system and increase its performance. While CPU scalability is common through building CPU clusters, most graphics systems today only support a single GPU. One major reason is the scarcity of well-established programming models and software support for multi-GPU systems. Major GPU vendors now support limited multi-GPU configurations such as ATI's Crossfire and NVIDIA's SLI, but these solutions are both limited in scalability and have feature sets targeted mostly at games. Cluster-based software such as Chromium [HHN*02] is well-suited for many applications that require scalable graphics, yet the programming model of Chromium disallows many forms of data communication that we believe would be useful and necessary in future scalable graphics systems.

The goal of this work is to explore a memory model for multi-GPU systems that both permits generalized communication between GPUs and is easy to use for programmers. We propose to virtualize the memory across GPUs into a single shared address space, with memory distributed across the GPUs, and to manage that memory with a distributed-shared memory (DSM) system hidden from the programmer. While current GPUs have neither the hardware support nor the exposed software support to implement this memory model both completely and efficiently, the system we describe shows promise as a powerful abstraction for multi-GPU graphics, and we hope that our work will influence the design of future graphics hardware and software toward supporting a DSM abstraction.

## 2. Background

Our work has been influenced by two families of previous machines, general-purpose multi-node computer systems and more specialized graphics systems that operate on multiple compute nodes.

### 2.1. General purpose systems

The two most popular mechanisms for sharing memory in multi-node systems are *message passing* and *shared memory*. Our design goal of making the migration from single-node to multi-node systems as easy as possible motivates a shared memory architecture, in which all nodes share a common address space and can transparently access data stored on other nodes. For the programmer, this memory abstraction is the same as for a single GPU: any GPU can access any memory in the system. The underlying memory system, however, faces the challenge of distributing the memory across multiple nodes and managing communication between nodes while maintaining consistency.

One method of maintaining consistency is to use a directory-based shared-memory architecture. The directory is a data structure that stores a copy of each block of memory and a set of state bits along with it. When the directory is distributed among nodes, this architecture is called distributed-shared memory (DSM) and is scalable [LLG*90]; our design is most influenced by that of Simoni [Sim90]. Within the directory, the state tracks which CPU caches hold which blocks and if a cache holds a dirty copy of the block. If a cache has a read miss, it issues a nonexclusive read request to the directory managing the missed block. The directory then locates the block (from main memory or a dirty cache) and sends it back to the requesting cache. In the case of a write hit or a write miss, the cache must request exclusive access from the directory. The directory will then invalidate all other copies of that block in other caches, mark the block as dirty, and allow the cache to proceed with the write.

### 2.2. Graphics systems

Eldridge et al. [EIH00] list five metrics to measure the performance of a graphics system: input rate, triangle rate, pixel rate, texture memory, and display bandwidth. In a scalable system, doubling the number of graphics pipelines should double each of these metrics.

Current vendor support for multi-GPU configurations includes NVIDIA's SLI [NVI05] and ATI's Crossfire [Per05]. These configurations can scale triangle rate and pixel rate, but do not scale texture memory. Both SLI and Crossfire replicate texture memory in the common case that textures are resident on the GPU†. As a result, data rendered to texture must be broadcast to all GPUs in the system, unless the textures are not persistent between frames. Both systems are highly optimized for game applications and have found limited use in more general-purpose applications. Both SLI and Crossfire are limited to 4 GPUs in current systems.

Chromium [HHN*02] allows streams of graphics API commands from precompiled applications to be filtered and forwarded to nodes in a cluster. Though Chromium has good performance scalability, it only allows limited forms of communication, supporting "only architectures that do not require communication between stages in the pipeline that are not normally exposed to an application" [HHN*02]. Our system, in contrast, presents an abstraction for sharing texture memory that is not normally exposed to applications.

Igehy et al.'s parallel API allows for the synchronization of multiple streams of graphics commands to the same drawable image [ISH98]. Instead of using application level barriers and semaphores, they propose a method for creating barriers and semaphores that operate at the graphics context level. Our system is not limited to rendering to a single drawable image, but when operating in a multi-GPU environment, synchronization between GPUs is important. The focus of this paper is more on mechanisms for transferring data and maintaining consistency and less on mechanisms for synchronization, so for simplicity, we use application level synchronization. Igehy et al.'s primitives would be desirable in a final implementation focused on performance.

Voorhies et al. [VKL88] present graphics as a virtual resource. In current systems, graphics hardware is seen as a virtual resource to the system, but the graphics programmer can not see texture memory as a distributed virtual space. We believe our work is consistent with the goals of Voorhies et al. in that a virtualized memory system will allow portable, efficient implementations of both applications atop the abstraction and the underlying mechanisms beneath it.

## 3. Implementation

The goal of our system is to allow programs to run across multiple GPUs with a common, consistent, distributed memory address space. The core of our implementation is a directory-based distributed shared memory system that handles the details of memory management transparently from the programmer. In the terminology of traditional DSM systems, the CPU's memory is main memory and the GPU's memory is treated as a cache. Such a system can be scalable across a large network of CPUs and GPUs because there is no central resource [LLG*90].

At a high level, we implement our system as a parallel program across multiple CPU nodes, each of which can support one or more GPUs. The CPUs and GPUs must cooperate to implement distributed shared memory across them. We have identified two fundamental mechanisms which are necessary to support shared memory: *sharing* and *invalidation*. Sharing allows one node to retrieve a copy of memory that is resident on a different node; invalidation allows a node to notify other nodes that it requires exclusive access to a portion of memory. Together, these two mechanisms allow consistent migration of data between GPUs.

### 3.1. Data Structures

The shared memory abstraction creates a global texture address space for all textures to all GPUs, allowing different textures to be stored on different GPUs. The global address space is made transparent to the programmer so they will only need to work with texture IDs and coordinates local to a given texture. These IDs and coordinates will be globally

[†] If the aggregate texture size exceeds the amount of GPU memory, textures are demand-loaded from the CPU and thus may not be wholly redundant across GPUs. This demand-loading is not currently exposed to the programmer.

accessible to any GPU. However, to render a scene, a GPU must have all necessary texture data resident on that GPU. Thus our system requires the ability to transfer texture data between GPUs, and our first decision is the unit of transfer.

We choose the *page* as the fundamental unit of memory in our system. A page is a contiguous block of a single texture; we can configure the size of a page, but it is typically larger than a single texel but smaller than an entire texture. All transfers in our system are transfers of pages. We chose pages because texels are too small: the number of requests becomes unreasonably large and degrades performance. Entire textures are too large: we often transfer data that we will not use. (We discuss performance implications of page size in Section 5.)

All textures in our system are stored as pages, using the page table primitive of the Glift GPU data structure library [LKS*06]. This primitive has the same interface as a standard texture call (accessed with *s* and *t* coordinates), but is actually stored as individual pages indexed by a page table. A texel lookup, then, requires first looking up the page address in the page table, then using that index to calculate and look up the texel. Changing or updating a texture page requires both updating the contents of the page as well as the entry in the page table that points to the page.

The basic page table simply stores a pointer to the page data. To that pointer we add a flag that indicates if the page data is resident on the GPU or not. The contents of a single texture, then, might be distributed among multiple GPUs. We can also share pages between GPUs by storing the contents of the page on multiple GPUs.

We keep track of the status of each page using a simple directory organization. The directory resides on the CPU, and within it each page has an entry containing a presence bit for each GPU in the system, a single dirty bit, and a pointer to the CPU memory associated with the page.

### 3.2. Read Procedure

Armed with the page-table abstraction for texture data, we next show how a texture read is supported in our system. The challenge in supporting a read is that any texture read may result in many page requests, some of which may be resident on the local GPU, but some of which may only be resident on a remote GPU.

Because we do not know a priori which texture data is needed for a given texture call, and because the GPU exposes no capability to page in texture data in the middle of a pass, we divide a texture read call into two passes, requesting the necessary nonresident data from the directory between the passes.

Consider a fragment program that contains a texture read. We divide this fragment program into two separate partial fragment programs and run them as two passes on the GPU. On the first pass, we compute all calculations up to the texture access. Instead of requesting the texture data at this point, we instead retrieve the resident/nonresident flag. Any texel request that is not resident must then retrieve its texture page from the directory or a remote GPU, so we also calculate the page's global address. The result of this pass is a buffer of required texture pages; these requests are all nonexclusive, because these pages are read-only. We read that buffer back to the CPU.

When the CPU receives the list of global addresses, it looks up the location of those pages from the directory and sends a request to the remote GPUs for each page that is dirty on the remote GPU. The remote GPU on that node renders the desired pages into a texture and supplies the resulting texture to the local node. Then the local CPU both renders those pages back onto the local GPU and also updates its page table.

At this point all necessary texture data is resident on the local GPU, and we can begin the second pass. The partial fragment program in this pass begins where the last one left off, by requesting texel data, which is now wholly resident on the GPU. Internally, texture references become indirect lookups of texture data via the page table.

We discuss the necessary code changes to support this operation below (Section 3.5).

### 3.3. Write Procedure

Writes to texture memory, as in a render-to-texture call, are more complex but use a similar factorization. Writes require three passes, and one of the primary difficulties is that a write operation might write to some texels in a page but not change other texels.

The write procedure is similar to the read procedure; we retrieve all requested texture pages from remote nodes. Unlike the read procedure, however, we plan to write to some of these pages. Thus we mark these requests as exclusive and invalidate them on their remote nodes. We accomplish this invalidation by setting the dirty bits, clearing the presence bits, and updating the page tables on the remote nodes to indicate their data is no longer valid. On the local node, we also create a write mask texture that indicates which texels will be updated by the write.

At the end of the first pass, all relevant texture pages to be read are local to our GPU. On the second pass, we write the computed write data into a buffer the size of the computation domain, generate the write mask, and request exclusive access to the pages that have been touched. On the third pass, we write back into the texture pages addressed by the page table, using the value of the write mask at each texel to select between the old value and the new value.

### 3.4. Memory Consistency Model

Our system observes the three requirements Culler et al. list to guarantee sequential consistency [CGS97]. Every GPU will issue texture memory operations in program order.

Writes complete in the order they are issued, and reads complete only after previous writes to the same address have completed. The programmer must use proper application synchronization primitives to avoid race conditions when multiple GPUs are updating the same texel.

### 3.5. Programmer's View

The ultimate goal of our system is to make scalable multi-GPU support transparent to the programmer. We could expose the necessary support in two ways. First, graphics calls could be intercepted and modified to allow the program to run on multiple GPUs. This could be implemented in a low level driver or graphics library without any change to existing applications. A second implementation would be to create a new set of multi-GPU API calls to replace current pixel and texture operations. This would essentially provide a DSM-supported multi-GPU programming environment. Programs would have to be written using this new API to take advantage of multiple GPUs. Though these methods provide different interfaces to the programmer, they both depend on the same low-level mechanisms.

Our system is most similar to the first implementation. We currently take a stream of OpenGL calls from an unmodified application and transform it into a multi-GPU program. The programmer must currently make two specific manual changes to the input stream of OpenGL calls. We believe both of these changes are straightforward to automate.

**Fragment Program Factorization** The major change to GPU code necessary for read and write operations is to factor the fragment program, splitting at texture accesses. (We discuss alternatives in Section 6.) Each level of indirect texture lookup incurs an additional pass, though all texture accesses at any level of indirection can be satisfied on the same pass up to the output limit of the GPU fragment program. Currently, we perform this transformation manually, but it could easily be automated. Intermediate data in a partitioned program is stored in local textures; systems that solve the multipass partition problem (MPP) [CNS*02] all perform a similar operation.

**Image Space Partitioning** Splitting the output image between multiple GPUs requires changing the viewport and perspective calls to support rendering only part of the final image.

This architecture is orthogonal to a cluster- and stream-based system such as Chromium, which primarily manages the "top-to-bottom" application-to-display flow. Instead, we provide "side-to-side" communication, allowing one global address space for textures across all GPUs.

### 3.6. Threading System

One of the most challenging aspects of implementing our system was a design that avoids deadlock (Figure 1). A simple example will illustrate the problem: consider two GPUs, each of which is performing a texture read for which some of the texture is on the other GPU. If implemented incorrectly, the first GPU could be stalled waiting for data from the second GPU at the same time the second GPU is waiting for data from the first.

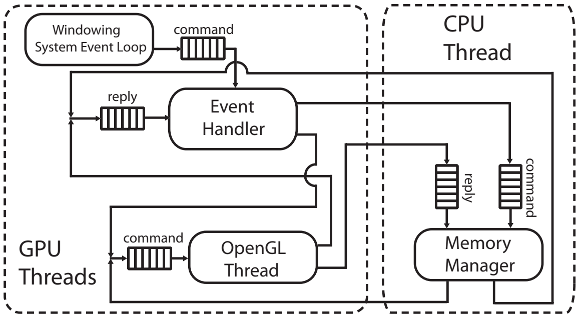

**Figure 1:** *Our multi-GPU threading system, showing the communication paths between each of the threads (Section 3.6).*

Our solution has three threads per GPU and one thread per CPU. For each GPU, we have a thread that reacts to events (the "event loop"), a thread that handles these events (the "event handler"), and a thread that interacts with the graphics system (the "OpenGL thread"). Each CPU contains a piece of the distributed memory directory, which is controlled by a "memory manager". Each thread operates on commands placed in its command queue, and will wait for replies to commands it issues to other threads.

**Windowing System Event Loop** We begin with the event loop, which reacts to mouse or keyboard events from the application. (This event loop is similar in structure to the GLUT event loop.) We require one event loop per GPU, though some events could have the same callbacks within the event handler and appear identical. Events detected by this thread are placed in the event handler's command queue.

**Event Handler** The event handler processes commands from its command queue, which come from either the event loop or from the event handler itself. It is the source of all commands sent to the graphics hardware, initiating GPU passes by requesting the OpenGL thread to read or write local data or process a set of OpenGL commands. It can also request remote data from the memory manager.

**OpenGL Thread** The OpenGL thread communicates with the graphics hardware and only has a single input queue of commands. It also is the only thread with access to the OpenGL context. The event handler and memory manager cooperate to ensure that any command sent to the GPU through the OpenGL thread is able to be fulfilled, so it never needs to block. It receives local read, write, and graphics commands from the event handler and remote read and invalidate memory operations from the memory manager. It replies to read and write commands from the event handler with a list of remote pages.

**Memory Manager** The memory manager provides the interface between the GPU and the rest of the system. It implements one node of the distributed shared memory directory and communicates with remote nodes to send and receive data. It receives requests from the event handler and sends remote read and invalidate commands to the OpenGL thread as well as replies to the event handler.

**Deadlock Avoidance** In a multithreaded system, a cycle of thread dependencies results in deadlock. Our system must stay deadlock-free and at the same time ensure memory consistency. In our system, the event handler can block waiting for replies from either the OpenGL thread and the memory manager; the memory manager will only block waiting for replies from the OpenGL thread; and the OpenGL thread never blocks. The lack of a possible cycle in these dependencies ensures that deadlock will not occur.

## 4. Applications

Our evaluation system for applications has dual 2.0 GHz AMD Opterons with 4 GB of RAM and dual NVIDIA GeForce 7800 GTXs over PCI-Express busses running driver version 1.0-8756 and Cg version 1.4 (26 Sept. 2005). Our operating system is Red Hat Linux Fedora Core 4.

Though this system is a small one, it successfully exposes the interesting and relevant issues for graphics hardware, and we have designed our system with scalable systems in mind. Larger multi-node systems would primarily exercise CPU scalability, which is already well-understood [LLG*90, Sim90]. The mechanisms we study here are applicable to any size of multi-node system.

Our goal is supporting any GPU-based application; to test it, we study two representative applications in detail. The first is a GPGPU boiling simulation; the second is a trace from a first-person shooter video game.

**GPGPU—Boiling Simulation** Our multi-GPU implementation of Harris et al.'s boiling simulation [HCSL02] applies a set of simple equations to a 2D input array containing heat data. At a resolution of 512×512 the simulation runs at 2.65 frames per second with our 2-GPU system. It requires four render-to-texture passes per frame; in each of them, each fragment accesses up to 5 texels in two textures, with the largest step in any direction being a single texel.

Dividing this application across multiple GPUs requires partitioning the computational space and allowing each GPU to operate on a subsection of the heat data. Since each fragment needs to access the texture values of its neighbors, on every frame, each GPU will need remote data that is outside of its computational domain. Any data along a partition will need to be shared between GPUs. Since we have chosen the page as our fundamental unit of memory, GPUs sharing a partition boundary must have shared copies of the pages along the boundary.

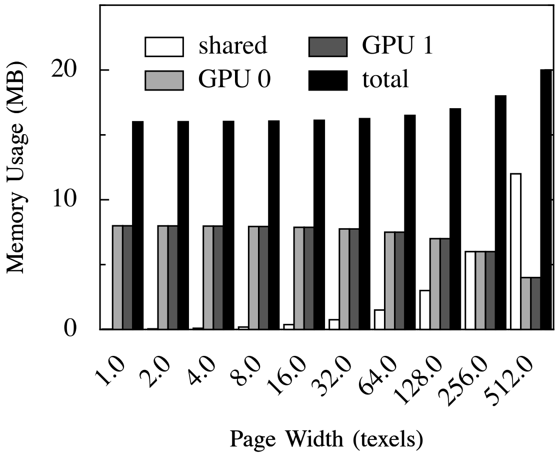

**Figure 2:** *As pages become larger, the amount of texture data that is unique to only one GPU (GPU 0 or GPU 1) decreases, while the amount of memory on both GPUs (shared) increases. "total" indicates the total amount of texture data loaded (the amount of data stored in the CPU directory.) Data from the boiling simulation at 1024×1024.*

**Standard Graphics—Game Trace** For a standard graphics application we choose a 500-frame trace of the "GLQuake" demo from id Software, captured with GLIntercept. This trace uses 135 RGBA8 textures, ranging in size from 8×8 texels to 512×256 texels. At 1024×768 resolution, it runs at 20 frames per second on our 2-GPU system.

To divide this application across multiple GPUs, we partition the output image. One half of the scene will be rendered on each GPU. The projection and viewport commands are modified to display the proper image. Textures must be shared on demand as each frame in the trace will have different texture requirements.

## 5. Analysis of Results

We now analyze three results in more detail: page size, memory usage, and performance.

Choosing the proper page size is important for performance and efficiency. Figure 2 shows the memory footprint of the boiling simulation for different page sizes. Ideally, both GPUs would have the same amount of data and no shared data at all. Small page sizes result in very little shared data, but also result in an increase in page requests and hence overall memory traffic.

Figure 3 shows our system's performance when using different page sizes on the boiling simulation. As predicted, smaller pages incur performance hits due to increased communications overhead. The overhead of transferring redundant data causes large page sizes to incur performance hits as

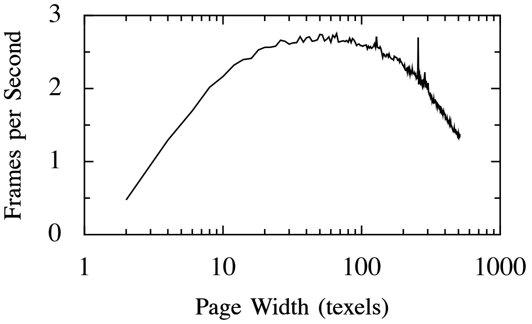

**Figure 3:** *Frames per second for our dual-GPU boiling simulation at 512×512 as a function of page size.*

| Write Pass | Stage | % |
|---|---|---|
| 1 | Read Request Pass (**R**) | 45.48 |
| 1 | Read Request Uniquify | 0.05 |
| 1 | Non-Exclusive Read Requests | 13.18 |
| 2 | Write Request Pass (**R**) | 21.81 |
| 2 | Write Request Uniquify | 9.61 |
| 2 | Exclusive Write Requests | 9.11 |
| 3 | Page Writes | 0.77 |

**Table 1:** *Boiling simulation at 512×512 with 64×64-pixel pages in dual-GPU configuration, showing the aggregate percentage of time spent in each pass of the write procedure for the four render-to-texture passes. Stages with (**R**) require one or more stream-compacted screen readbacks.*

well. The spikes in the graph occur when pages line up such that page boundaries correspond to the partition boundary. If a page spans the partition, both GPUs will be writing to texels in the same page, and thrashing will occur, causing a decrease in performance.

Table 2 shows performance for the boiling application running under different configurations. Our dual-GPU implementation has lower performance than our single-GPU implementation due to the increased communications overhead of swapping modified pages between the GPUs. Both the single and dual GPU configurations using our system are two orders of magnitude slower than the original single-GPU implementation. The majority of the display time in the simulation occurs in the four render-to-texture passes. Table 1 shows what fraction of time is spent in each pass of the write procedure. The main performance problem occurs when reading data back from the GPU. The readback process itself is expensive, and the GPU sits idle while requests are being processed.

| Our System | | Original |
|---|---|---|
| Single-GPU | Dual-GPU | Single-GPU |
| 3.65 | 2.63 | 362.74 |

**Table 2:** *Measured frames per second for the 512×512 boiling simulation with 64×64-pixel pages.*

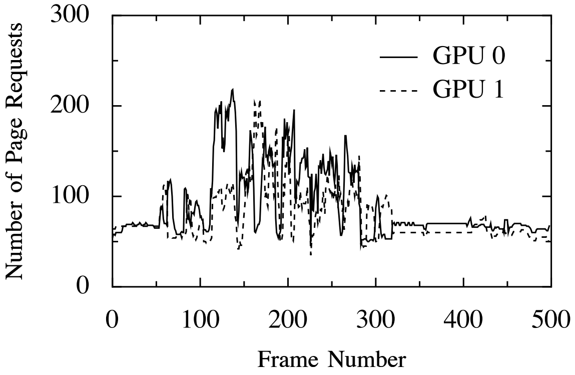

**Figure 4:** *Number of texture page requests for 8×8 pages over 500 frames of the GLQuake trace at 1024×768, measured on both GPUs. This data was generated by clearing the contents of GPU memory each frame.*

As memory traffic increases, the memory manager quickly becomes a large bottleneck. For maximal efficiency, a high frame-to-frame coherence of textures minimizes memory traffic and increases performance. Fortunately, real applications typically exhibit such a coherence, as we show in Plate 1. Similarly, the boiling simulation has high frame coherence because any pages that are not on a partition border will never be needed on another GPU. We also note that the GLQuake trace has a large variation in the amount of texture pages needed each frame (Figure 4). Together, these results imply that the contents of individual texture memories in a multi-GPU system are likely to be substantially different, and that we will be able to scalably leverage the aggregate texture memory in such a system.

## 6. Discussion

The system we present here meets our functionality goals but does not yet deliver efficient parallel performance for several reasons, including vendor hardware and system software limitations and future work in our system. In this section we identify these limitations.

**Performance** One of the main bottlenecks in our system is processing read requests from fragment programs with more than one texture access. Current hardware only supports up to four render targets, so each program must be separated into multiple passes, incurring a full screen readback per pass per render target. More render targets would help to reduce this cost. A second bottleneck in the system is the processing of read and write requests. When reading back requests from the boiling application, every pixel writes to texture and thus makes an exclusive request. The proper primitive here is not stream compaction, because there are no pixels not requesting data, but instead uniquify, because of the substantial redundancy in the request stream. Today, however, implementing either compact or uniquify on the CPU or the GPU are expensive operations; better support on the GPU would significantly improve our performance.

**Toward full GPU utilization** One of the major sources of inefficiency in our system is that the GPU is often idle. For instance, when the GPU requires remote texture data, it sends its requests to the memory manager on its host CPU and idles until that response is fulfilled. If data is remote, that request is inevitable, but the idle time is not.

One possible solution is to overlap GPU computation and remote requests. Currently those two phases are serialized, because the GPU waits for the remote request. We can accomplish this by dividing work into multiple batches per frame and overlapping the first phase of batch $n$ with remote texture requests for batch $n+1$; other partitions, such as a screen-space subdivision with aggressive frustum culling, may be possible. Essentially, this approach threads the input application's command stream to the graphics hardware in the same way that multiple threads can effectively cover high memory latencies in CPU systems.

Another possible solution that is applicable to read-only remote texture memory in applications that can tolerate a temporary lack of image quality is to store all textures as mipmaps (subject to the discussion on mipmaps below) and to replicate the coarsest levels of the mipmaps across all GPUs. When the local GPU receives a request for a remote texture page, it immediately satisfies it from the coarse mipmap and at the same time requests it from the remote GPU for use in future frames.

We do not currently optimize for the case when all textures are local. When this occurs, we can dynamically eliminate the need for intermediate CPU communication because no requests will be generated. This optimization could be implemented using occlusion queries. When rendering to a buffer to generate requests, all fragments not requesting a page can be killed and an occlusion query can be used to see if any fragments make it to the buffer.

### 6.1. Limitations in our System

**One fragment per pixel** One limitation of our system is that texel requests can only be generated by one fragment that contributes to a given pixel. An example of this is blending; consider a rendering pass that blends two fragments, each requiring a remote texel read, at a single pixel location. In our system, this requires two passes (Section 3.2). The first pass produces a remote texture page address for each of those two fragments, but those two addresses must be stored in a single pixel location. If blending is enabled, the two requests will be blended together, producing a single, erroneous request. If we disable blending, the rear fragment will be killed by the depth test. The fundamental problem is that an arbitrary number of fragments may contribute to any pixel, but we have no mechanism to store all fragments in a render target, only all pixels.

We currently disable blending and only store the nearest fragment. We could implement depth peeling [Eve01] at the cost of one pass per layer. The proper solution would be hardware support of an F-buffer [HPS05, MP01] to store the intermediate fragments, coupled with the ability to read those fragments back to the CPU.

**Mipmapping** Modern graphics hardware filters textures through the mipmap [Wil83], a precomputed pyramid of prefiltered textures; each mipmapped texture read fetches and blends eight texels. The mipmapping hardware is not exposed to the GPU programmer, however: the fragment program's interface to the texturing system is limited to sending a single address and returning a single (filtered) texture value. Mipmapping texture page addresses is not meaningful.

We do not currently support mipmapping in our system. It would be straightforward to manually decompose all mipmapped texture requests into eight separate accesses, then perform the filtering in the kernel; such a strategy would incur a significant performance penalty over using hardware mipmapping, however. The best long-term solution would be exposing filtering to the programmer within the fragment program.

**Flow control in fragment programs** Because we partition fragment programs into multiple passes, we also do not properly support flow control (conditionals and complex loops) in fragment programs. Partitioning in the face of flow control is a known hard problem; most solutions to the multipass partitioning problem assume a directed acyclic graph (DAG) as the input and are not suited to partition fragment programs with flow control. An instruction scheduling approach to the MPP [RLV*04] offers a starting point for a possible solution.

### 6.2. Implications for Graphics Hardware

Though the Glift page tables we use are reasonably efficient, it is highly likely that the GPU features hardware-supported page tables (that map texture IDs to texture addresses, for instance, or supporting demand paging) that would deliver higher performance. Exposing these hardware page tables to the programmer, and generalizing their functionality, would lead to better performance in our system, permit better memory management in graphics and GPGPU applications, and allow data structures with superior performance.

One step in the right direction is DirectX 10 and the Windows Display Driver Model (WDDM). The first version of WDDM will support the virtualization of textures and render targets on per surface basis, with the future intent of moving to virtualization at the granularity of pages. This will eventually allow faults to occur within shaders and suspend the execution of a context until faults are serviced. Currently, this is only planned to work on a single GPU or in a restricted multi-GPU environment similar to SLI or Crossfire [Bly06]. We believe that the paging infrastructure created by WDDM could be leveraged to distribute texture memory over multiple GPUs.

Above, we discussed the benefit of hardware support for the F-buffer, for more render targets, and for exposing filtering in fragment programs. Stream compaction would benefit from either hardware support of the parallel scan primitive or hardware support for efficient reductions; uniquify requires an efficient sort. Another primitive that would substantially impact our implementation is block scatter. Supporting generalized scatter, in which any fragment can write to any destination, is problematic for many reasons (including coherency, hardware efficiency, and proper write semantics). However, a block scatter restricts the fragment destination to the same offset within an arbitrary block, mitigating the difficulties with arbitrary scatter. Block scatter with blocks the size of pages would (among other benefits) reduce our write procedure (Section 3.3) from three to two passes.

On the software side, making Cg thread-safe would eliminate the need for protecting each Cg call with a lock and thus increase overall software performance.

## 7. Conclusion

Our system supports a distributed-shared memory abstraction for scalable, consistent, distributed texture memory over multiple GPUs. While our implementation currently requires manual, but simple, code transformations, it can be made wholly transparent to programmers.

The core contribution of this paper is our identification of the mechanisms for supporting this abstraction together with the limitations that constrain its performance. Today, all signs point to increased virtualization of texture memory in future single-CPU-single-GPU graphics systems; we hope the mechanisms and exposed limitations of this work help point the way to flexible, powerful, high-performance virtualization techniques for the parallel, multi-GPU systems of the future.

**Acknowledgements** Many thanks to Eric Demers, Bob Drebin, Mike Houston, Aaron Lefohn, Pat McCormick, Henry Moreton, Shubhabrata Sengupta, and the anonymous reviewers for their helpful comments and contributions to this work.

## References

[Bly06] Blythe D.: Private communication. Microsoft, 7 June 2006.

[CGS97] Culler D. E., Gupta A., Singh J. P.: *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[CNS*02] Chan E., Ng R., Sen P., Proudfoot K., Hanrahan P.: Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Graphics Hardware 2002* (Sept. 2002), pp. 69–78.

[EIH00] Eldridge M., Igehy H., Hanrahan P.: Pomegranate: A fully scalable graphics architecture. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, pp. 443–454.

[Eve01] Everitt C.: *Interactive Order-Independent Transparency.* Tech. rep., NVIDIA Corporation, May 2001. `http://developer.nvidia.com/object/Interactive_Order_Transparency.html`.

[HCSL02] Harris M. J., Coombe G., Scheuermann T., Lastra A.: Physically-based visual simulation on graphics hardware. In *Graphics Hardware* (Sept. 2002), pp. 109–118.

[HHN*02] Humphreys G., Houston M., Ng R., Frank R., Ahern S., Kirchner P., Klosowski J.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (July 2002), 693–702.

[HPS05] Houston M., Preetham A. J., Segal M.: *A Hardware F-Buffer Implementation.* Tech. Rep. CSTR 2005-05, Stanford University Department of Computer Science, 2005.

[ISH98] Igehy H., Stoll G., Hanrahan P.: The design of a parallel graphics interface. In *Proceedings of SIGGRAPH 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 141–150.

[LKS*06] Lefohn A. E., Kniss J., Strzodka R., Sengupta S., Owens J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics 26*, 1 (2006), 60–99.

[LLG*90] Lenoski D., Laudon J., Gharachorloo K., Gupta A., Hennessy J.: The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (1990), pp. 148–159.

[MP01] Mark W. R., Proudfoot K.: The F-Buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (2001), pp. 57–64.

[NVI05] NVIDIA Developer Relations: *NVIDIA GPU Programming Guide*, 2.4.0 ed., 8 July 2005. `http://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf`.

[Per05] Persson E.: *Programming for CrossFire$^{TM}$*, 2005. `http://www.ati.com/developer/`.

[RLV*04] Riffel A. T., Lefohn A. E., Vidimce K., Leone M., Owens J. D.: Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware 2004* (Aug. 2004), pp. 35–44.

[Sim90] Simoni R.: *Implementing a Directory-Based Cache Consistency Protocol.* Tech. Rep. CSL-TR-90-423, Stanford University Computer Systems Laboratory, Mar. 1990.

[VKL88] Voorhies D., Kirk D., Lathrop O.: Virtual graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (Aug. 1988), vol. 22, pp. 247–253.

[Wil83] Williams L.: Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)* (Detroit, Michigan, July 1983), vol. 17, pp. 1–11.

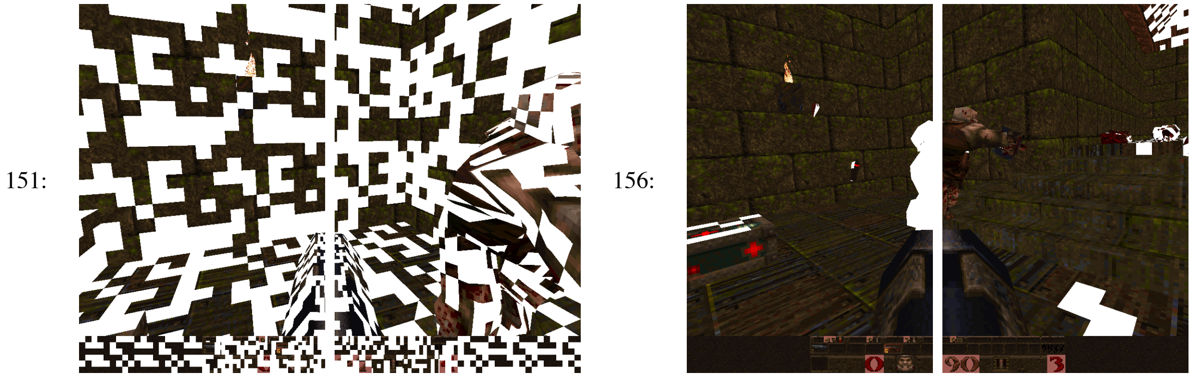

Plate 1: Frames 151 and 156 from the GLQuake trace, split left/right across 2 GPUs. Before Frame 151, all texture pages (size 8×8) were randomly assigned to one of the two GPUs; white textures are remote. For this experiment, we have artificially limited all pages to reside on only one GPU unless the page is used on both GPUs. After 5 frames, most pages have successfully migrated to their local GPU, minimizing the number of required remote reads.