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



**Author**
Moerschell, Adam

**Publication Date**
2007

Peer reviewed

**Distributed Texture Memory in a Multi-GPU Environment**

By

ADAM THOMAS MOERSCHELL
B.S. (University of California at Davis) 2004

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in charge

2007

**Distributed Texture Memory in a Multi-GPU Environment**

To my family and friends who always had more faith in my abilities than I did...

## Acknowledgments

In my time at UC Davis, I have had the privilege to meet and work with many great people. Their influence on my work was essential.

First, I would like to thank my advisor John Owens. Without his superb guidance and excitement about my work I would have had a difficult time completing it. John pretty much changed the course of my professional life when he approached me about working with GPUs. I had no idea how much I would enjoy working in this field, and I owe it all to him for bringing me into it.

I would like to thank the other reviewers of my thesis, Chen-Nee Chuah and Ken Joy, for their feedback and comments. I additionally want to thank Chen-Nee for helping me early on in my graduate career by advising me and introducing me to research.

Next, I would like to thank all of my fellow graduate students who have helped me along the way. Aaron Lefohn for being a great teacher and mentor in the ways of computer graphics. Shubho Sengupta for being such a knowledgeable resource. The rest of the students in the graphics lab for their support, and all of the other students I worked with in classes and on projects.

During the summer of 2005 I had the privilege to have an internship at Los Alamos National Laboratory. Working on the Scout project there with with Patrick McCormick, Jeff Inman, and Jim Ahrens was a great experience. I would like to thank them all, especially Pat for his interest in my work and the time he spent reviewing my paper submission.

Many thanks go to Eric Demers and Bob Drebin of ATI, who took time out of their busy schedules to review my paper and give me some useful feedback.

I would also like to thank Mike Houston, Henry Moreton, Nat Duca, and Nick Triantos for visiting the graphics lab and talking to me about my work.

Special thanks are due to my many amazing friends. I cannot begin to list all your names, but I want to thank you all for giving me your support when I most needed it.

Finally, my biggest thanks goes out to my family, because without them I would not be where I am today. My family has been a pillar of support throughout my schooling and my life. Their faith in me has been beyond measure and I love them all dearly.

# Contents

# List of Figures

# List of Tables

## Abstract

In this work, we demonstrate a system that allows texture memory on multiple GPUs to be virtualized in a manner that is both scalable and transparent to the programmer. Our system is built using a directory-based shared memory abstraction to allow texture memory to be distributed while staying consistent. We use texture pages as our basic memory block and discuss the data structures, threading model, and consistency mechanisms necessary to implement a paging system in a multi-GPU environment. The system is demand-driven, and pages will only be loaded into the texture memory of a GPU that makes a request. The main contribution of this work is the identification of the mechanisms required to implement our abstraction, as well as the discussion of its limitations in order to make it more efficient.

# Chapter 1

# Introduction

Recently the graphics processing unit (GPU) has become a powerful computational tool used for both graphics and complex computational tasks. With the potential for computational power that exceeds the best CPUs, the GPU now targets both increasingly demanding graphics workloads as well as non-graphical, general-purpose computation. This move has allowed the GPU to assist in many diverse applications, including physical simulation, mathematical processing, film rendering, large-scale visualization, and interactive graphics.

These complex applications would benefit from scalable graphics systems that allow users to add additional hardware to a system and increase its performance. Recently, CPUs have been moving to multi-CPU and multi-core systems in order to improve performance. While CPU scalability is common through these technologies or 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. These systems make multiple GPUs appear as one logical entity to the programmer, and do not allow for independent execution on each GPU. Cluster-based software such as Chromium [16] 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. Custom multi-GPU configurations can also be implemented by hand. The problem is that hand implementations are time consuming; They require the programmer to design threading systems to operate multiple GPUs and to develop the inter-GPU communication systems required to transfer data. The new Windows Display Driver Model (WDDM), to be debuted with the Windows Vista operating system, will introduce memory virtualization support to GPUs, but it is unclear if multi-GPU support will be any different from SLI or Crossfire.

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. Besides being transparent to the programmer, this model is scalable to multiple GPUs within a single node, and can be distributed across a clustered environment as well. The DSM system ensures the consistency of the memory system using texture pages as the basic memory block. Since a DSM system is demand driven, only texture data used by the GPU is loaded into its texture memory. This makes memory usage more efficient. 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 high-performance DSM abstraction.

This abstraction will be beneficial to many applications wishing to leverage the parallel processing capacity of multiple GPUs. One obvious usage model is to use screen space subdivision. If each GPU is in charge of rendering a certain subsection of the final framebuffer, only textures that appear in the screen space of a given GPU will be loaded into that GPU's texture memory. This method could be used for a single display driven by multiple GPUs, or when many displays are linked together and each node renders a different frustum of the final image. A second usage of this system would be for general purpose GPU applications (GPGPU). Many GPGPU applications do not focus on producing a

visual image, but rather operate on data sets and take advantage of the parallel processing capabilities of GPUs. In this type of application, each GPU can operate on its own subset of the data and have the data it needs paged to it on demand.

In both screen space and data space subdivision, the same operations are happening across all GPUs. Distributed applications do not need to function this way. Each compute node could operate independently and run different graphics code. One example of this would be a distributed ray tracer that runs on the GPU. In a clustered environment, GPUs would be given batches of rays, and when a GPU finishes a batch, it will request more rays to process. All of the rays will be tracing a common data set or scene, but each ray will have different data dependencies depending on the path it takes. As the rays travel, the data they need can be loaded on demand to the GPU tracing that ray.

We begin describing this memory system with related work in Chapter 2, then describe our system design and implementation in Chapter 3. We describe the multi-GPU applications running under our system and use them to draw conclusions about our system in Chapter 4. We identify both performance bottlenecks and potential hardware and software additions and changes in Chapter 5 and conclude in Chapter 6.

# Chapter 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. Most general-purpose systems do not directly map to a GPU environment because they were initially designed for use with CPU computing. Current GPU models are limited and do not allow full utilization of the memory resources of the GPU.

## 2.1 General purpose systems

The two most popular mechanisms for sharing system memory in multi-node and multi-processor systems are *message passing* and *shared-memory*.

### 2.1.1 Message passing

In a message passing system, inter-node communication is facilitated by sending messages between nodes. Usually this is through an API, such as the Message Passing Interface (MPI) [5, 29], that allows point to point, broadcast, and barrier mechanisms. This means that each node is operating independently and has its own, individual memory space. The only way that applications can access data generated by other nodes is if the programmer implements request and reply mechanisms specific to an application. Applications that use message passing normally have a predefined communication structure that must be im-

plemented by the programmer. Exposing message passing as an inter-GPU communication tool is a bad fit for our system because it does not fulfill our goal of a unified memory space that is invisible to the programmer.

### 2.1.2 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.

In a simple implementation, a shared-memory architecture uses a snooping protocol. Snooping protocols get their name because caches "snoop" every memory transaction by every processor. This allows the caches to be coherent and consistent because they are able to maintain a global view of the status of each memory block. In order to support snooping, broadcast mechanisms are necessary. Broadcasting creates a lot of unnecessary traffic and has limited scalability. We choose to not implement a snooping shared-memory system due to its limited scalability.

Another method to maintain consistency is to use a directory-based shared-memory architecture. The directory is a data structure, residing in system memory, that stores a copy of each block of memory and a set of state bits along with it. Within the directory, the state information tracks which CPU caches hold which blocks and if a cache holds a dirty copy of the block. This gives the directory a global view of all the memory blocks in the system. If a cache has a read miss, it issues a non-exclusive 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. When the directory is distributed over multiple nodes,

this architecture is called distributed shared-memory (DSM) and is scalable [19]; our design is most influenced by that of Simoni [28]. Simoni goes to lengths to describe the details of implementing a directory based shared-memory system, including many subtle deadlock cases that a naive implementation may neglect to address.

## 2.2 Graphics systems

Eldridge et al. [10] 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 approaches succeed in scaling some of these metrics, but fail at others.

### 2.2.1 Hardware approaches

Current vendor support for multi-GPU configurations includes NVIDIA's SLI [23] and ATI's Crossfire [25]. These configurations can scale 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[1]. As a result, data rendered to texture must be broadcast to all GPUs in the system, unless the generated textures are not persistent between frames. Both systems are highly optimized for game applications and have found limited use in more general-purpose applications. This is because GPGPU applications rely heavily on render-to-texture operations where generated textures are used in future frames, and thus must be broadcast to all GPUs.

In addition, SLI systems use special-purpose connectors to share data (resulting in a dedicated high-bandwidth path with correspondingly high performance) rather than the more general-purpose system busses we choose. Currently, SLI is limited to four GPUs within a single machine. NVIDIA has deployed special purpose multi-GPU boxes called Quadro Plex VCSs [32]. Each Quadro Plex VCS can hold up to four GPUs, but it is built on top of SLI technology, and suffers the same limitations.

[1] 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.

Crossfire, unlike SLI, does not require special purpose connectors, because it is able to transfer all needed data over the system bus. Crossfire is also limited to two GPUs (three if one is used for physics), but the technology has been designed in a way that it could be scaled to more than two GPUs. The main limitation is the form factor of a PC and number of card slots on the motherboard [9].

Both of these technologies make multiple GPUs look like a single entity to the programmer. Though this eases the programming model, if a programmer wishes to use multiple GPUs independently in a single application, with each running a different command stream, there is not a common method or straightforward implementation to achieve this. ATI has begun to address this issue with Asymmetric Physics Processing for Crossfire [1]. The fact that this system allows programmers to use multiple GPUs independently in one application could allow us to manage multiple GPUs in a way that Crossfire does not. If we chose to build our setup on top of this system, we would be limited to applications that used physics and graphics, and could not look at the more general problem of distributing texture memory.

ATI has also created a new data parallel virtual machine for GPUs called Close to the Metal (CTM) [24]. CTM allows programmers to use GPUs as a highly parallel processor without having to understand graphics programming. It also allows multiple GPUs to be accessed independently within a single application. This abstraction is very powerful and would allow us to create a distributed memory system for GPUs, but it would limit our work to non-graphical applications.

A custom hardware approach, proposed by Manzke et al., uses GPUs, FPGAs, and Scalable Coherent Interface (SCI) connections to create a distributed shared memory system [21]. Though SCI can create a distributed shared memory system in hardware, their setup prevents them from using SCI to implement the default SCI cache coherency protocol, and it is unclear as to what coherency and consistency mechanisms they use. This novel hardware-based approach could become a viable multi-GPU solution.

### 2.2.2 Software approaches

Chromium [16] allows streams of graphics API commands from precompiled applications to be intercepted, 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" [16]. Our system, in contrast, presents an abstraction for sharing texture memory that is not normally exposed to applications. We believe our system and Chromium are complementary. Our current implementation could be implemented using Chromium, but a proper low level implementation would not benefit from Chromium due to the lack of lower level support.

Igehy et al.'s parallel API allows for the synchronization of multiple streams of graphics commands to the same drawable image [17]. 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 work 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. [31] 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. Though textures are an abstraction of texture memory, this abstraction is not a global virtual space of all graphics memory in the system. 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.

Current commodity graphics cards map textures to physical addresses. In early 2007, with Windows Display Driver Model (WDDM), texture memory will be a virtual space, and all texture memory references will be to virtual addresses. In WDDM, faults to

texture memory can occur as fine grained as the pixel level. The miss penalties for page faults are mitigated through context switching to other running graphics applications. Page faults may be serviced at the same time as a graphics context is executing [26]. WDDM plans to support multiple GPUs treated as a single logical entity, but it is not known if it will support configurations where applications can communicate with each card individually. In future versions of WDDM the virtual space can be shared across multiple GPUs [3]. It is known that multiple GPUs will operate in either a mirrored or instanced memory mode, but at this point it is not clear what the conditions for each mode of operation are. Exposing the details of page fault handling to the programmer could allow for the creation of a distributed texture memory system across multiple nodes.

One virtual texture representation is SGI's clipmaps [30]. A clipmap is essentially an extension to mipmapping that allows very large textures to partially reside on the GPU. The main observation is that with finite display resolution a user can only visualize a limited amount of texture data at one time. When most of the texture is visible, a lower quality mip level can be used. When the user is zoomed in, only a portion of the texture will be visible, and therefore only a subsection of the highest quality mip level will need to be in texture memory. Clipmaps could be implemented in our system, but due to mipmapping limitations discussed in Chapter 5 other changes must be made before it can be implemented.

ClawHMMER is an example of a clustered GPGPU application [14]. The nature of the algorithm allows for a static distribution of texture data that is never shared or migrated to other nodes in the cluster. Due to this independence, the application scales well on its own, and would not benefit from a global texture memory system. Though ClawHMMER does not need a global texture memory system, Chapter 4.1 shows that there are other GPGPU applications which will take advantage of a global texture memory system when migrating to a multi-GPU environment.

Clustered GPGPU computing is an increasingly popular research area, and ZippyGPU [12] has implemented a distributed memory architecture for clustered GPU usage. It creates a virtual address space and ZippyGPU manages the memory underneath. ZippyGPU attempts to maximize locality by distributing texture data appropriately for the user. It is a library built upon known graphics APIs and the message passing interface

(MPI). It is unclear what memory consistency model they use or if texture writes are allowed.

# Chapter 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. The system allows GPUs to run identical or independent command streams, each with access to the global memory space. In the terminology of traditional DSM systems, the CPU's system memory is main memory and the GPU's texture memory is treated as a cache. In a multi-node system, the directory is split and distributed over multiple nodes. Such a system is scalable across a large network of CPUs and GPUs because there is no central resource [19]. Though there is no central resource, data must be distributed properly to avoid bottleneck nodes with highly volatile memory blocks.

At a high level, our system is implemented as a parallel program across multiple CPU nodes, each of which supports 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.

Currently, the system is implemented as a single-CPU, multi-GPU system. In this case there is only one directory and multiple physical texture memories. While designing the system, a move to a multi-node environment has been kept in mind, and will be discussed in further detail in Chapter 5. Due to the fact that drivers for commodity graphics cards are large, complex pieces of software and that are largely closed source, it is difficult for us to implement our system at a level that is as close to the hardware as it should be. Obtaining and modifying the source of a current graphics driver would be just as weighty of a task as implementing our own driver. Instead, we choose to implement our system using a current graphics API and to deal with the constraints imposed by the graphics pipeline. This has obvious performance penalties, but we feel that despite these, our system shows the mechanisms necessary to create a globally addressable, distributed texture memory that is transparent to the programmer. We discuss the limitations to this system, and future software and hardware support necessary to better support our system in Chapter 5.

## 3.1 Memory Consistency Model

Graphics applications rely heavily on texture read and render-to-texture (write) operations. In order to understand how memory operations work in a global texture memory, a definable consistency model is necessary. A memory consistency model specifies how memory transactions will complete in a multi-processor or concurrent programming environment. This eases the programmability of the system, and allows the programmer to have clear expectations of how memory operations will occur. Our system observes the three requirements Culler et al. list to guarantee sequential consistency [8]. Every GPU issues 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.

Though our memory model is designed to be transparent to the user, there are still hazards that the programmer must be aware of. This is the case in any multi-processing system. Race conditions may arise when multiple GPUs try to modify the value of the same texel. In applications where this occurs, the programmer must use proper application synchronization primitives to avoid race conditions. It is conceivable that the parallel API

proposed by Igehy et al. [17] could be used for synchronization at the graphics context level.

## 3.2 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, and effectively making the size of texture memory the sum of all texture memories in the system. The global address space is made transparent to the programmer so he will only need to work with texture IDs and coordinates local to a given texture without having to worry about implementing an inter-GPU communication mechanism. These IDs and coordinates describe a texel address globally accessible to any GPU. However, to render a scene, our system must guarantee that a GPU will have all necessary texture data resident in memory. Thus our system requires the ability to transfer texture data between GPUs, and our first decision is the unit of transfer.

### 3.2.1 Basic Memory Block

Texture data can be made local to a GPU at many granularities.

**Graphics Context** When a specific graphics context is made current and begins execution on a GPU, all textures associated with the context are made local to the GPU. If many context switches occur or if the texture data used in an application does not fit into texture memory, this is not a viable solution.

**Driver Call Stack** All textures that will be bound by commands in the current driver call stack are guaranteed to be local to the GPU. At this point it is not known which parts of the texture will be visible, if at all, and the texture data may not need to be downloaded to the GPU.

**Polygon/Shader** When a texture is bound to be mapped to a polygon or as an input to a fragment shader, it is loaded to the GPU. In the case that the polygon is not fully visible on the screen, it may not be necessary for the full texture to be resident on the GPU.

Depending on eviction policies, there may be a lot of unnecessary swapping, especially when texture memory is close to full.

**Texture Data on Demand** When a texture is accessed in a fragment processor, if the requested data is not local to texture memory, it is loaded on demand from system memory. This can help by not loading unused data to the GPU. The best implementation of this strategy would involve graphics hardware that has faulting capabilities.

We have implemented a system that allows data to be loaded to GPU texture memory on demand. In order to efficiently manage texture data, we choose the *page* as the fundamental unit of memory in our system. A page is a contiguous, rectangular 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. The size of the page is important to the performance of the system, and we discuss performance implications of page size in Chapter 4.2. Though pages could be any shape, we only examine square pages.

### 3.2.2 GPU Structures

All textures in our system are stored as pages, using a page table primitive to access them on the GPU. This primitive has the same interface as a standard texture call (accessed using a texture ID and $s$ and $t$ coordinates), but data from multiple textures is actually stored as individual pages and indexed by the 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.

A basic page table is simply a texture storing pointers to page data. Page data is packed into a separate physical memory texture. A page table pointer is just an $(s,t)$ coordinate within the physical memory where the lower left corner of a page resides. Due to the limitations of graphics hardware, physical memories have limited capacities (maximum

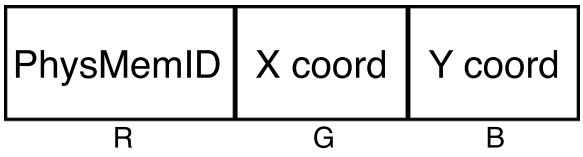


Figure 3.1: GPU page table entry.

of $4096 \times 4096$ texels in current hardware and $8192 \times 8192$ in DirectX 10 hardware) and are not easily resizable. Therefore, as physical memories become full, additional physical memories must be created. In order to index the proper physical memory, the page table entry must be expanded to store the ID of the physical memory the page resides in. Since our system is demand driven, and pages are loaded only when needed, we use a negative physical memory ID to indicate that the page is not resident in any physical memory on the GPU. Each page table entry needs to store three values, therefore it can be represented as one texel in a RGB formatted texture. Figure 3.1 shows an example of a page table entry. The contents of a single loaded texture might be distributed among multiple GPUs. Our page table structure allows pages to be shared between GPUs by storing the contents of the page in physical memories on multiple GPUs.

Figure 3.2 shows a page table lookup. First, the page table entry is located. Next, the ID field is checked and if the page is valid, the $x$ and $y$ coordinates will be used to index the correct physical memory texture containing the page.

Our original implementation of the page table used a Glift page table object [18], but we found this primitive to be too limited for our system. A Glift page table object can only have a single physical memory texture associated with the page table. This physical memory is also limited by the same size constraints as our physical memories. Since we want each GPU to be able to address more than $4096 \times 4096$ texels, our system requires a page table to be able to point to more than one physical memory. We could make due with only one physical memory texture, but we would be forced to manage multiple page tables. It would be possible to implement a page table with multiple physical memories in Glift, but it would only take advantage of Glift's page table address translator and the page table entry structure would have to be modified. Also, this implementation would require us to

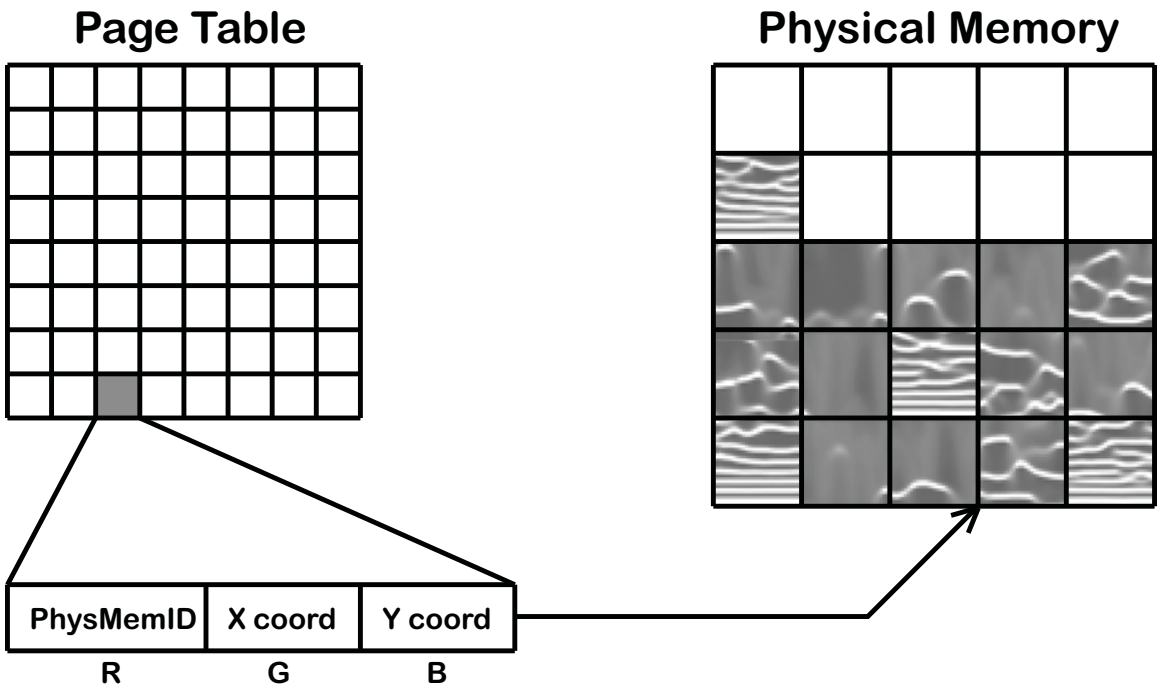


Figure 3.2: Page table texture look up.

manage physical memories by hand since they would not be easily wrapped into one Glift object. We felt that, even though it was possible to use Glift to implement our page table, it was cleaner to implement our own page table rather than only use part of the functionality of Glift.

### 3.2.3 CPU Structures

We keep track of the status of each page using a simple directory organization. The directory resides in system memory on the CPU, and within it each page has an entry containing a presence/valid bit for each GPU in the system, a single dirty bit, and a pointer to the CPU memory associated with the page. The directory can be distributed over multiple CPUs, each managing a separate section of the global memory space. The contents of the directory give a global view of the status of every page in the whole system.

Figure 3.4 depicts the entire system and shows where the directory fits in. Using our read and write procedures, described below, within our memory consistency model, the CPU is able to use the directory to manage GPU texture memory.

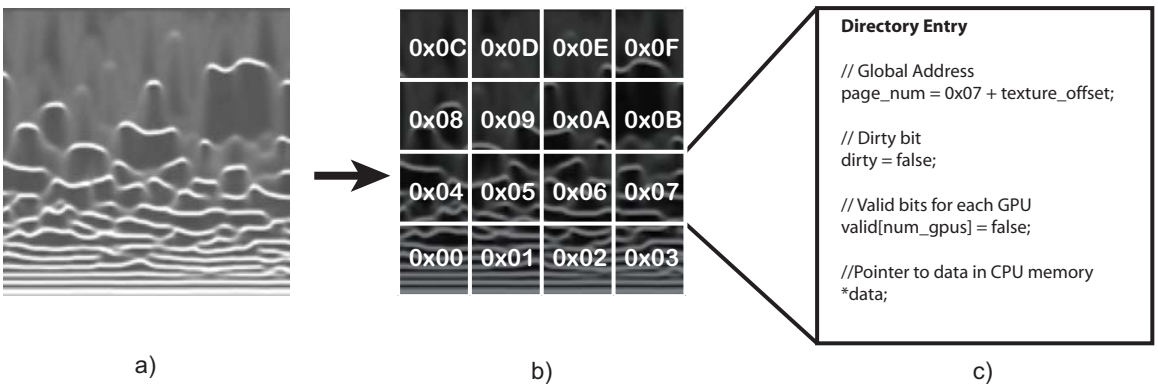


Figure 3.3: Texture load procedure. a) Original texture b) Texture broken into pages c) Directory entry for a given page.

When a texture is loaded into the system, it must be broken into pages (seen in Figure 3.3). Pages are assigned in row major order starting at the lower left hand of the texture. Since every texture is broken into pages the same way, it is easy to use texture coordinates to determine which page a texel resides in. Pages on the right and top edges of the texture may have unused data because because pages are statically sized. Once a texture is broken into pages, each page is given an entry in the directory. Initially the data is only located at the data pointer, and dirty bit and valid bits are false because the system is demand driven and no pages have been requested by any GPUs.

## 3.3 Address Spaces

The global memory system is transparent to the programmer, because texture data is accessed by binding a texture ID and providing texture coordinates. There is one subtle difference from a single GPU system: texture IDs are globally visible and can be bound in any GPU command stream. In order to create this system, there are multiple address spaces describing the location of texture data.

- Local Texture Space - Addresses texture data using $s$ and $t$ coordinates. This is the only address space that the programmer can see.
- Local to Texture Page Address - The local page address starts at zero (as in Figure 3.3b), and is used as an intermediate value used to obtain the global page address.

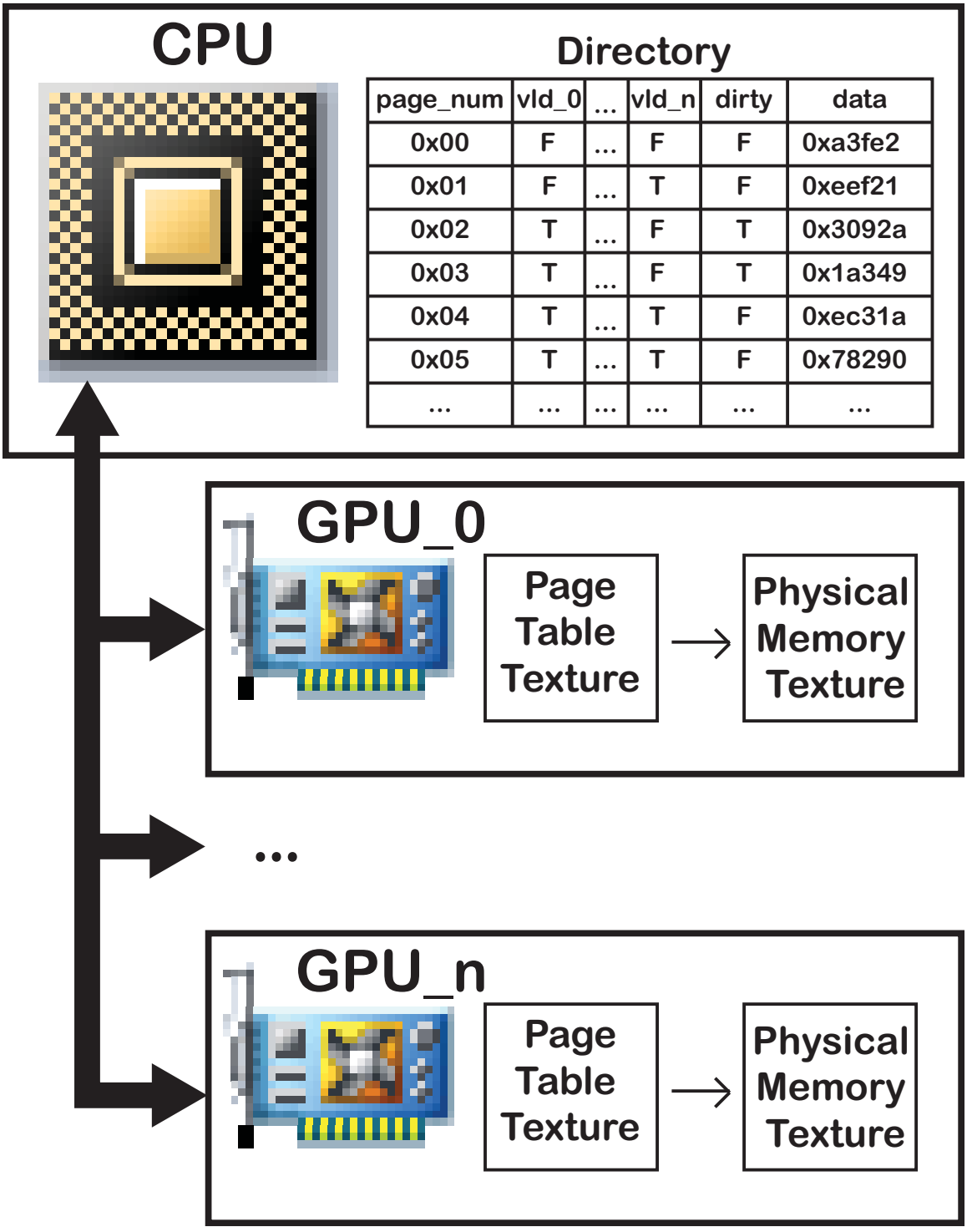


Figure 3.4: The global view of memory in the system.

This address is constructed using local texture space coordinates and the width of the texture.

- Global Page Address - Every page has a globally unique page address. Each address has an entry in the directory.
- Physical Page Address - This is the location of the page in a physical memory texture. Since a page can reside on multiple GPUs simultaneously, a given page could map to multiple physical addresses.
- Texel Address - This is the location of a texel in the physical memory. It is calculated by adding the texel's offset to the physical address of the page it resides in.

In order to retrieve the final address of a texel, we must first determine which page in the texture it resides in (local to texture page address). This requires the texture coordinates, page size, and width of the original texture to be passed to the fragment program. The equation below shows how the page address is calculated.

$$LocalToTexturePageAddress = \lfloor \frac{S}{PageSize} \rfloor + \lfloor \frac{T}{PageSize} \rfloor \cdot \lceil \frac{TextureWidth}{PageSize} \rceil \tag{3.1}$$

While determining which page the texel resides in, we must also determine the offset of that texel in the page. Since pages are two-dimensional, we must record both the $x$ and $y$ coordinates.

$$TexelOffset_x = S \bmod PageSize \tag{3.2}$$

$$TexelOffset_y = T \bmod PageSize \tag{3.3}$$

In order to do a page table look up, we need the global address of the page. Since each texture occupies a certain range of the global address space, each texture has a page offset in global memory. Each texture ID maps to a page offset that must be passed into the fragment program. This offset, plus the local to texture page address, is the global address of the page. This address then allows the GPU to look up the page table entry for the page.

$$GlobalPageAddress = LocalToTexturePageAddress + TexturePageOffset \tag{3.4}$$

Now, assuming that the given page is local to the GPU, we are able to obtain the physical address of the page by querying the page table. The physical address is the $x$,$y$ coordinate of the lower left hand corner of the page in physical memory.

$$PhysicalAddress = PageTable[GlobalPageAddress] \tag{3.5}$$

Finally, we take the retrieved physical address and add the texel offset to it. The generated texel address is used to do a texture look up in the physical memory texture.

$$TexelAddress = PhysicalAddress + TexelOffset \tag{3.6}$$

$$FinalTexelValue = PhysicalMemoryTexture[TexelAddress] \tag{3.7}$$

The complete texel lookup procedure will require both the physical memory and page table textures to be bound to the fragment program, as well as the texture offset, page size, and texture width constants.

## 3.4 Read Procedure

Armed with the page-table abstraction for texture data, we now 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 in the directory or 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 calculate the address of the global page

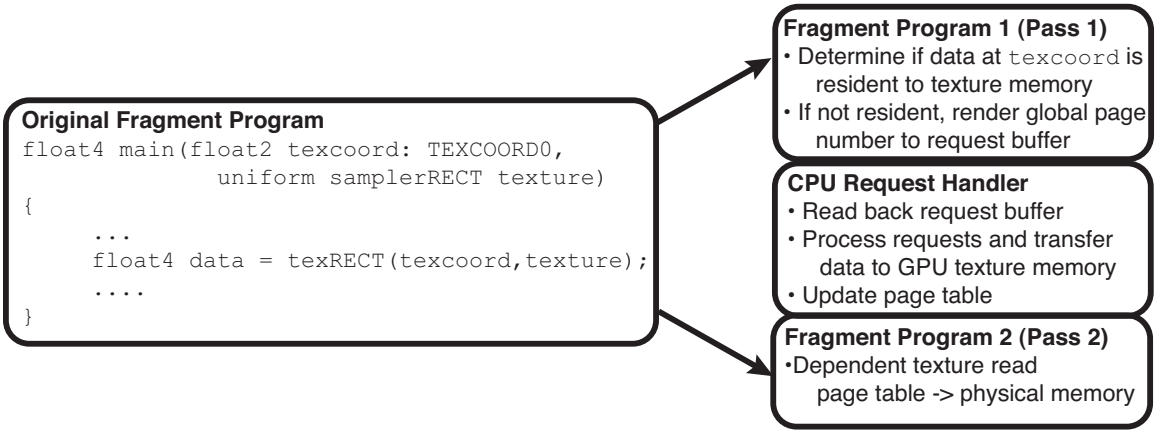


Figure 3.5: Fragment program factorization.

the data resides in and retrieve the physical memory ID from the page table to determine if the texel is resident. Any texel request that is not resident must then retrieve its texture page from the directory or a remote GPU, so we must create a request for the nonresident page. The result of this pass is a buffer of required texture pages; these requests are all non-exclusive, 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 a remote GPU. The remote GPU then renders the desired pages into a texture and supplies the resulting texture to the local node. Then the local CPU copies this data into the directory, renders the pages into the local GPU's texture memory, and updates the GPU's 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. The factorization of a fragment program into these two passes is outlined in Figure 3.5.

When rendering an image, it is highly likely that pixels that are near to each other in image space will map to texels from the same page. This means that the intermediate buffer that is read back to the CPU will be highly redundant with many requests for the

same texture page. CPU readback is one of the slowest operations for a graphics application, and it is important to limit how much data is read back from the GPU. Currently, we implement a compaction algorithm that removes pixels that do not make a page request. Though this reduces the amount of data read back, the proper primitive is *uniquify*. A uniquify operation would reduce the temporary buffer to a set of unique pixel values to be read back. We have determined that current GPU uniquify implementations do not always increase the performance of readbacks, so we do not use uniquify when measuring results in Chapter 4.2.

## 3.5 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 the directory and remote GPUs. Unlike the read procedure, however, we plan to write to some of these pages. Thus we mark any write requests as exclusive. In servicing an exclusive request, the most up to date copy of a page is loaded on to the requesting GPU and any copies on remote GPUs are invalidated. The most up to date copy must be loaded to the GPU, because the write may not modify all texels in the page. Page invalidation is accomplished by setting the dirty bits and clearing the presence bits in the directory, while updating the page table entries on the remote GPUs to indicate their data is no longer valid. On the local GPU, 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.

## 3.6 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 alternative. 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. This process is outlined in Figure 3.5. 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. Currently, we perform this transformation manually, but it could easily be automated. Intermediate data in a partitioned program could be stored in local textures; systems that solve the multipass partition problem (MPP) [7] all perform a similar operation.

**Partitioning** Some applications may choose a work division that uses 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. There are well known methods to accomplish this without creating visual artifacts or distortions [4]. It is up to the programmer to design the work partitioning across GPUs.

**Independent GPU Operation** In the case where the programmer wants to use each GPU independently with separate command streams, each GPU must be exposed to the programmer. This is achieved in our system by allowing each GPU to have a different display callback bound to it. Synchronization is left up to the programmer via application synchronization primitives. This functionality is more similar to the second method of exposing multi-GPU functionality. Though this exposes some of the multi-GPU environment to the programmer, the memory system is still transparent. We feel that the exposure of this functionality makes our system more versatile.

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.

## 3.7 Threading System

One of the most challenging aspects of implementing our system was a design that avoids deadlock (Figure 3.6). 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.

Our solution has three threads per GPU and one thread per CPU. For each GPU, we have a thread that captures events (the "event loop"), a thread that handles these events (the "event handler"), and a thread that manages 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. These queues allow commands from multiple sources to be issued to the thread. Replies to commands will contain data, or, to ensure proper synchronization, an acknowledgment that a procedure has finished. To ensure that these replies are not stalled behind other commands to the thread (a situation causing deadlock), a reply queue is added to threads that issue

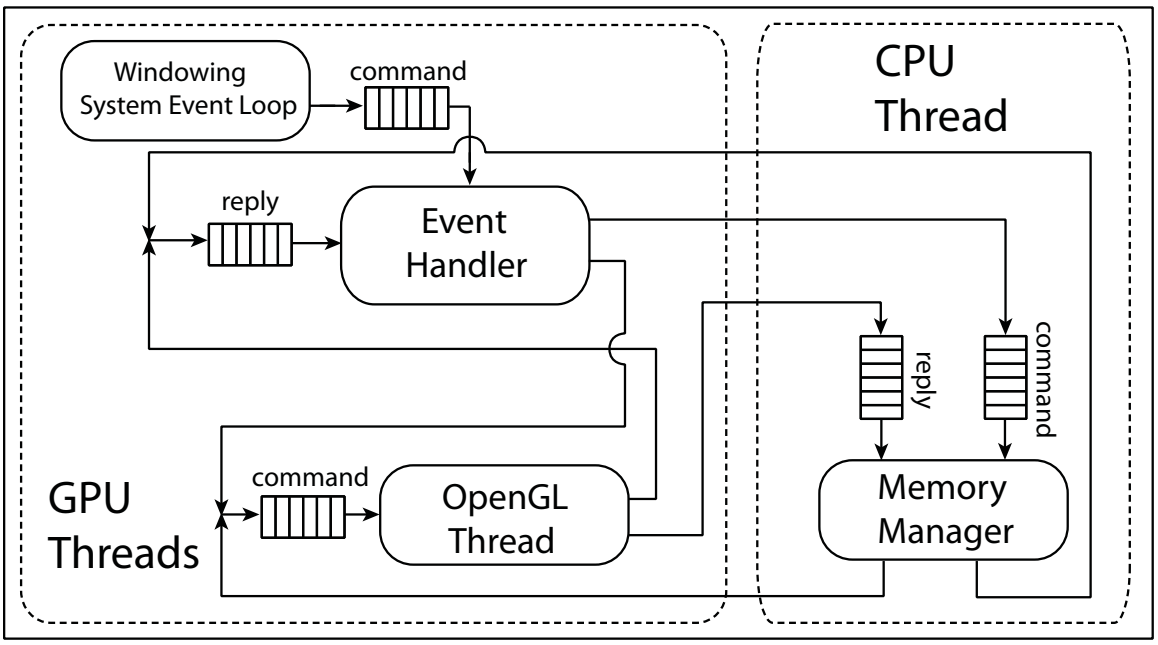


Figure 3.6: Multi-GPU threading system, showing the communication paths between each of the threads.

commands that require replies.

A threading system is necessary because we wish to support nodes with multiple GPUs installed. To take advantage of such a system, there must be multiple graphics contexts operating simultaneously. When creating a graphics context it must be made current (bound) to a specific execution thread in order to operate. If we did not use multiple threads, there would be significant overhead in making the context of each GPU current to the execution thread. Also, in the single threaded case, we do not take advantage of the operating system's scheduler and thread switching to multiplex the threads.

We now describe the role of each of the threads in the overall system.

**Windowing System Event Loop** We begin with the event loop, which receives all events generated by the windowing system. These events range from refreshing and resizing the window to user input such as mouse clicks and keystrokes. We require one event loop per GPU, and 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. All user defined callbacks (display, mouse, motion, keyboard, etc.) are registered with the event handler. It is also the source of all rendering commands sent to the graphics hardware, initiating each GPU pass by packaging the proper callback, fragment program, and ancillary GL calls into a command to send to the OpenGL thread. Each command constitutes one "pass", and the event handler is in charge of managing the ordering of passes. These passes are parts of read or write procedures (see Chapter 3.4 – 3.5), or executing pure OpenGL commands. The event handler also must submit requests for remote data to 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 OpenGL thread receives commands from both the memory manager and the event handler. These commands were designed in such a way that they never cause the OpenGL thread to block: each command can always be executed to completion once it arrives in the OpenGL thread's command queue. It receives local read, write, and graphics commands from the event handler and remote read and invalidate memory operations from the memory manager.

**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.

## 3.8 Deadlock Avoidance

As described by Simoni, the cache controller cannot stall and wait to process a memory request from another cache until after it has received a reply to a memory request of its own [28]. In the context of our system, GPU texture memory is our cache, and the graphics context is our interface to access that cache. This means that the graphics context

cannot block waiting for a reply, so that is why we design the OpenGL thread to accept commands that never block and can always execute to completion.

In a multithreaded system, a cycle of thread dependencies may result 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 or 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.

One subtle case where consistency and deadlock issues arise is when multiple GPUs wish to update the same page. Consider a case where the memory manager receives two exclusive requests: one from GPU0 and one from GPU1. Assuming the request from GPU0 arrives first, a valid copy of the page is loaded onto GPU0 and GPU1's copy will be invalidated. At this point the memory manager tells the event handler in charge of GPU0 that it is okay to proceed. Next, the second exclusive request is processed and the memory manager attempts to send the most up to date copy of the page to GPU1. A problem could arise if the first event handler and GPU0 have not fully completed the first write procedure before this data request arrives at the head GPU0's OpenGL thread command queue. In this case, the previously valid, but not yet written to, page is returned. In order to avoid this situation, when replying to any memory request (exclusive or non-exclusive), the memory manager sets a memory lock on the requesting event handler. If any requests to invalidate or read memory on this GPU are generated by subsequent commands to the memory manager, they stall until the lock is released. Once the event handler has finished the read or write procedure, it releases the memory lock. There is no worry of deadlock in this case because the event handler always runs to completion to release the lock and never generates a request to the memory manager while its memory is locked. When this subtle case does arise, performance is degraded due to the extra stalls incurred by the locks.

# Chapter 4

# Evaluation Studies

Our evaluation system has dual 2.0 GHz AMD Opteron processors with 4 GB of RAM and dual NVIDIA GeForce 7800 GTXs over PCI-Express busses running driver version 1.0-9623 and Cg version 1.5 (23 May 2006). 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 [19, 28]. The mechanisms we study here are applicable to any size of a multi-node system.

We are interested in evaluating our system for both memory usage and performance. Applications can have significant variation in their memory usage patterns, and the system configuration can have a large impact on how the memory is distributed. The system configuration and memory distribution also affect performance. Since 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.

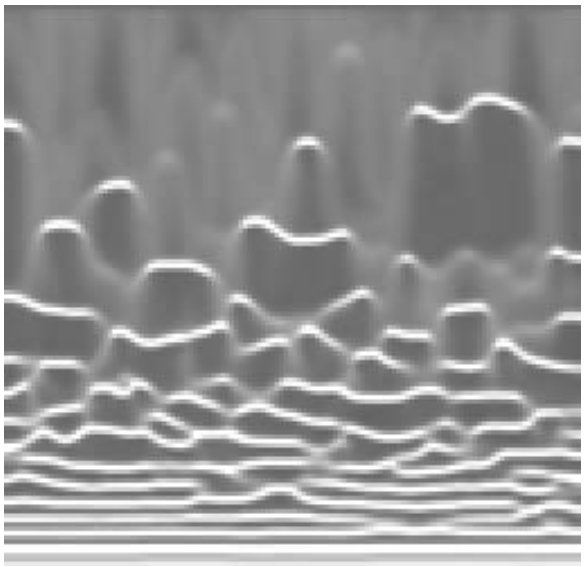

Figure 4.1: Screen shot of the GPGPU boiling simulation.

## 4.1 Applications

### 4.1.1 GPGPU—Boiling Simulation

Our multi-GPU implementation of Harris et al.'s boiling simulation [13] applies a set of simple equations to a 2D input array containing heat data. At a resolution of 512×512, with each GPU rendering 512×256, the simulation runs at 2.54 frames per second with our 2-GPU system. A screen shot of a typical frame can be seen in Figure 4.1. To produce a single frame, the application requires four render-to-texture passes followed by one read pass. Each render-to-texture pass reads a maximum of 9 texels to generate a final value. Our hardware only allows four outputs from a fragment program. Therefore, in order to check what texel data is needed for the render-to-texture, multiple passes are required (this is first pass of the write procedure, see Chapter 3.5 and 3.6). Table 4.1 outlines the four write procedures and their texel read and pass requirements.

The boiling application has memory access patterns describable as Glift single and neighbor iterators [18]. Figure 4.2 depicts each of these texture read patterns. The texels in gray are read and used to calculate the value of the pixel rendered to texture. The first pass, *border*, simply writes a border to the top and bottom of the input texture. This

| Pass | Texels Read | Passes Required |
|---|---|---|
| Border | 1 | 1 |
| Diffusion | 5 | 2 |
| Buoyancy | 3 | 1 |
| Latent Heat | 9 | 3 |

Table 4.1: Number of texels read and pass requirements for each render-to-texture pass of the boiling simulation.

is simply a single iterator and is access pattern 1. Next, the *diffusion* pass calculates a diffusion coefficient using the value of the input texture at the current location and its four neighbors using access pattern 3. Next, the *buoyancy* pass is only interested in computing vertical movement and reads the output of the diffusion pass using access pattern 2. Finally, the *latent heat* pass reads both the input texture and the buoyancy texture. It uses access pattern 4 on the input texture and 3 on the buoyancy texture. The final step is a read pass that displays the output texture from the latent heat pass on the screen. This texture will also become the input texture for the next frame.

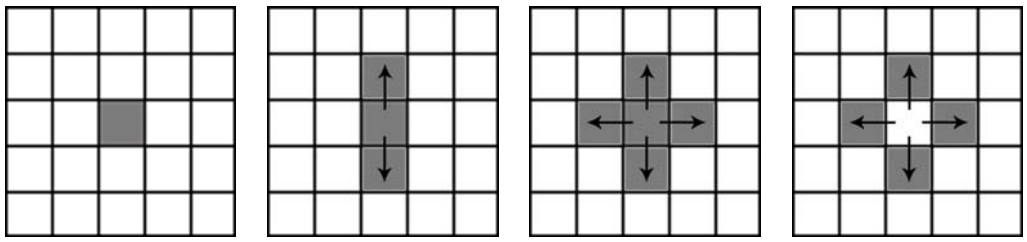


Figure 4.2: Texture read patterns for boiling simulation.

Dividing this application across multiple GPUs requires partitioning the computational space and allowing each GPU to operate on a subsection of the heat data. Each GPU will operate on one half of each generated texture (input, diffusion, buoyancy, and latent heat). Since each generated pixel 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 the 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.

We believe this application is representative of GPGPU applications. It has heavy

reliance on render-to-texture operations and uses iterator access patterns. The fact that the work division we choose will require a lot of inter-GPU communication means that our system will be stressed by running this application.

### 4.1.2 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, with each GPU rendering 512×768, 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 [4]. Textures must be shared on demand as each frame in the trace will have different texture requirements.

Though GLQuake is an old game, we feel that it is representative of standard interactive graphics applications. It does not use advanced rendering techniques that newer games have, but the texture access patterns should be similar. Our interest in this application is to investigate frame to frame coherency and memory traffic patterns created by this genre of application.

## 4.2 Analysis of Results

Due to the fact that our memory system is implemented at the software API level and not at the driver or hardware level, we do not see significant performance improvements in the applications we implemented, and in fact see performance degradation compared to the single-GPU case. Our main performance limitation is slow data data transfers: to and from single GPUs and between multiple GPUs. This does not mean we can not draw some interesting conclusions. Our applications run at a reasonable speed that allows for some performance measurements, and we can also observe memory usage patterns. We now analyze three results in more detail: memory usage, page size, and performance.

The memory footprint required by an application can have a large impact on the

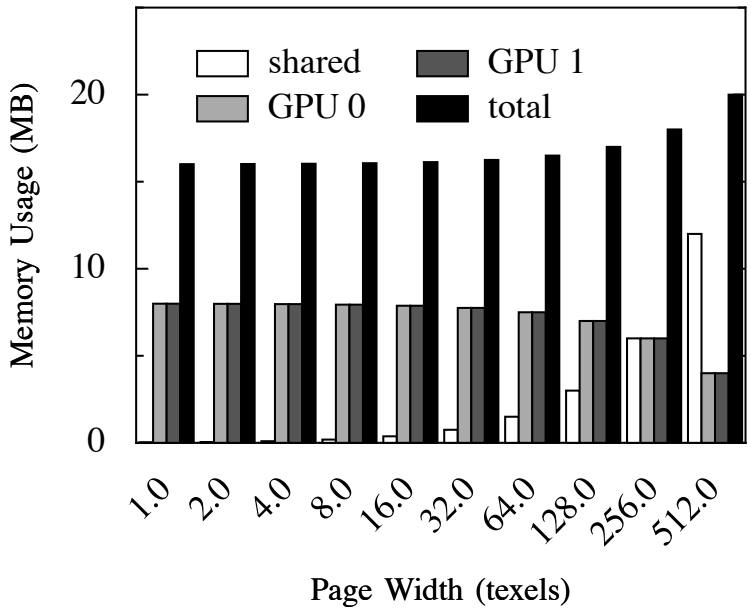


Figure 4.3: As pages become larger, the amount of texture data that is unique to only one GPU (GPU 0 or GPU 1) decreases, while the amount of memory on both GPUs (shared) increases. "total" indicates the total amount of texture data loaded (the amount of data stored in the CPU directory.) Data from the boiling simulation at 1024×1024.

efficiency of the memory system. From a memory system perspective, the only parameters we can manipulate to change the memory footprint are the size of texture memory and the size of the pages stored in texture memory. Since texture memory is normally static on a given GPU, and only limits the memory footprint in terms of capacity, we ignore this factor.

Choosing the proper page size is important to minimize the amount of unneeded data while maximizing data transfer efficiency. In a multi-GPU system, each page will either be unused, stored on one GPU, or stored on multiple GPUs. If the memory system uses a directory that limits how many shared copies of a page are allowed or if the application writes to texture memory, shared pages become a contended resource. Ideally, all GPUs would have their own working data set and a minimum amount of shared data. Figure 4.3 shows the memory footprint of the boiling simulation for different page sizes. Small page sizes result in very little shared data, but, as seen in Figure 4.4, also result in a larger amount of page requests and hence a lot of communications overhead to service the requests. When moving to larger sized pages, the communications overhead goes down, but the amount of

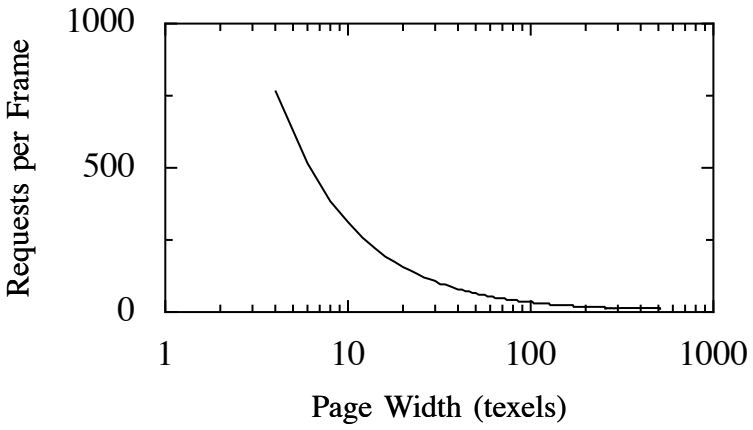


Figure 4.4: Read requests per frame as a function of page size for the boiling simulation at 512×512 using two GPUs. Read requests include standard read requests, and exclusive requests for non-resident pages.

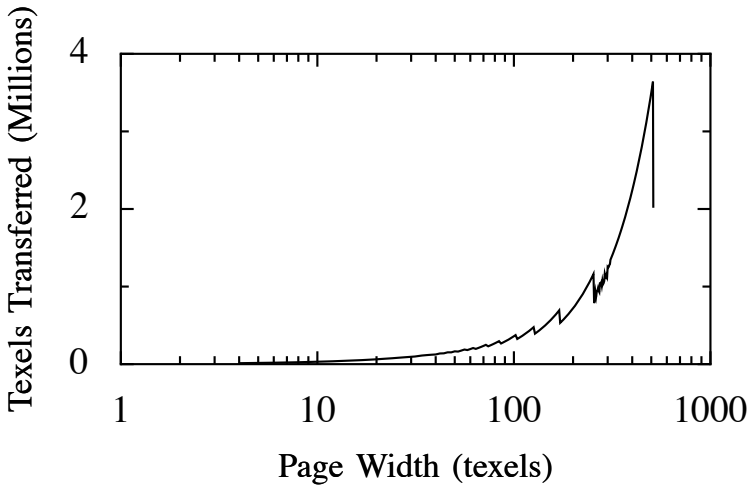


Figure 4.5: Total data transferred per frame to both GPUs per frame as a function of page size for the boiling simulation at 512×512 using two GPUs. Data is transferred during a standard read request or during an exclusive request for a non-resident page.

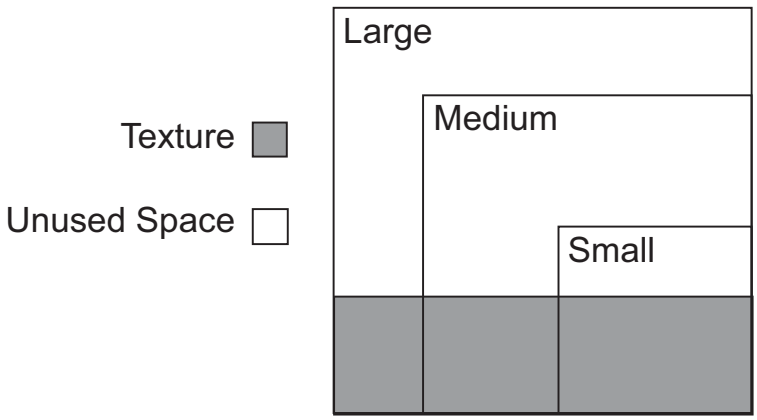


Figure 4.6: If a texture has a different aspect ratio than the pages larger pages will have more wasted space.

data transferred increases, as seen in Figure 4.5. Also, all the data residing in the page may not be needed.

The total data in the system increases with larger pages because some of the textures used in the boiling application are long and skinny. Figure 4.6 shows that textures with aspect ratios different than the page aspect ratios will have some amount of wasted space, and as the page size becomes larger this wasted space becomes larger. The only solution to this would be to pack textures into pages, but this will have addressing problems. The final texel address would not simply be the page's physical address plus the texel offset (see Chapter 3.3), but rather a complicated equation that would require more parameters to be sent to the shader program and more calculations to occur to translate the local texture space address.

Figure 4.3 shows us that page size has a large impact on the memory footprint our application will have. The measurements show that large page sizes are definitely a bad choice, but it is unclear as to how small we should make our pages. With smaller pages incurring large communications overhead and larger pages having large data transfer costs, there must be an optimal page size. The problem is that every application has its own memory usage pattern, and, therefore, a page size tuned for a specific application may not be the correct size for all applications.

Figure 4.7 shows our system's performance when using different page sizes on the

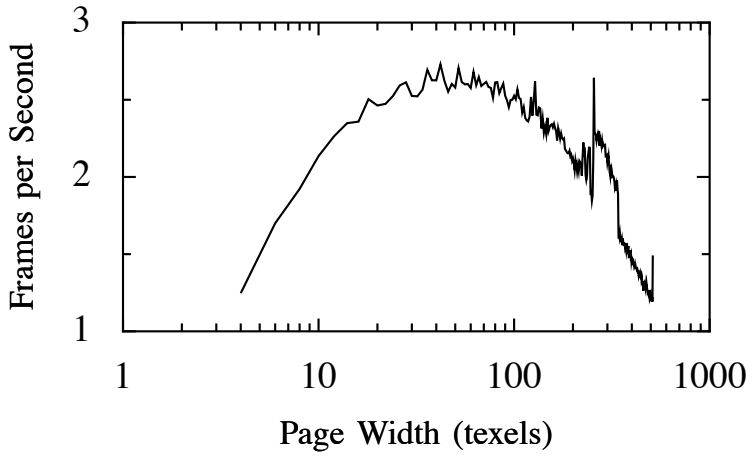


Figure 4.7: Frames per second for the boiling simulation at 512×512 using two GPUs as a function of page size. We see that the maximum performance corresponds to a page width of an intermediate size; small pages incur per-page overheads, while large pages incur more transfer costs.

| Our System | | Original |
|---|---|---|
| Single-GPU | Dual-GPU | Single-GPU |
| 3.48 | 2.54 | 362.74 |

Table 4.2: Measured frames per second for the 512×512 boiling simulation with 64×64-pixel pages.

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 well. The spikes around 128 and 256 pixels 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. For the boiling application, the optimal point of operation is pages with widths somewhere between 32 and 64. In this range, the effects of thrashing cause minimal performance degradation.

Even though we expect our performance to be poor due to our software implementation, it is still important to analyze why our performance is so poor. Table 4.2 compares the original application to two different configurations of our system. In all cases our implementation is two orders of magnitude slower than the original implementation. We even see a degradation in performance by using two GPUs. The disparity between the single and

| Stage | % |
|---|---|
| Page Read Request Pass (**R**) | 41.82 |
| Read Request Uniquify | 0.01 |
| Non-Exclusive Page Request Handling | 19.62 |
| Page Write Request Pass (**R**) | 17.92 |
| Write Request Uniquify | 9.10 |
| Exclusive Page Request Handling | 10.75 |
| Page Writes | 0.74 |
| Display (**R**) | 28.91 |

Table 4.3: 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.

dual GPU configurations can be attributed to the fact that, in the single GPU case, each page fault will only occur once. Once each page fault has happened, all required data is local to the GPU and no more transfers will occur.

Though our single-GPU implementation outperforms the dual-GPU implementation, the difference is not that large (¡30%). There must be more than just the transfer of data between the GPUs that is causing our performance problems. The boiling application consists of four write passes and one read pass. Table 4.3 breaks down these passes into their major components and shows what percent of frame time is spent in each stage. The major contributors to frame time are the page read and write request passes. These passes determine which texels will be read and written to, and require a readback of a temporary buffer to the CPU. Though the display pass simply displays our results to the screen, the readback in the Read Procedure causes this pass to have significant costs as well. Reading data back to the CPU is a very expensive process, and it shows that this also hinders the performance of our application. The handling of exclusive and non-exclusive requests are a large factor in frame time, but due to the problems with readback they are not as significant of a factor.

Though currently not the most significant factor in frame time, the handling of exclusive and non-exclusive requests can be a large bottleneck in our system. As memory traffic increases, the memory manager will quickly become a large bottleneck. For maximal efficiency, a high frame-to-frame coherence of textures minimizes memory traffic and

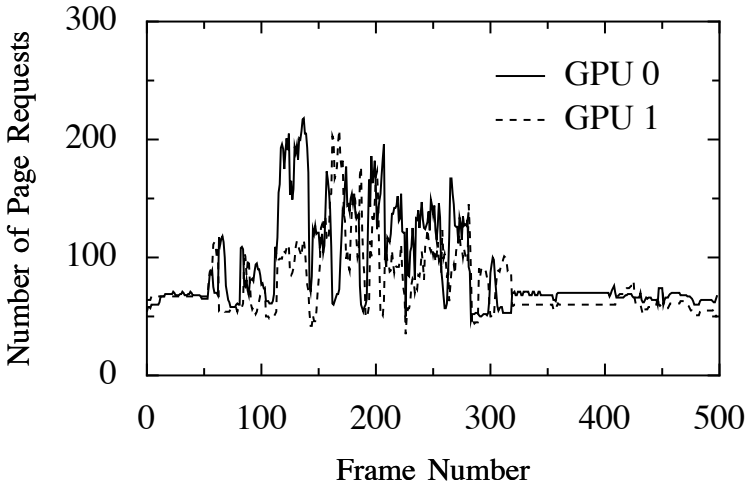


Figure 4.8: Number of texture page requests for 8×8 pages over 500 frames of the GLQuake trace at 1024×768, measured on both GPUs. This data was generated by clearing the contents of GPU memory each frame.

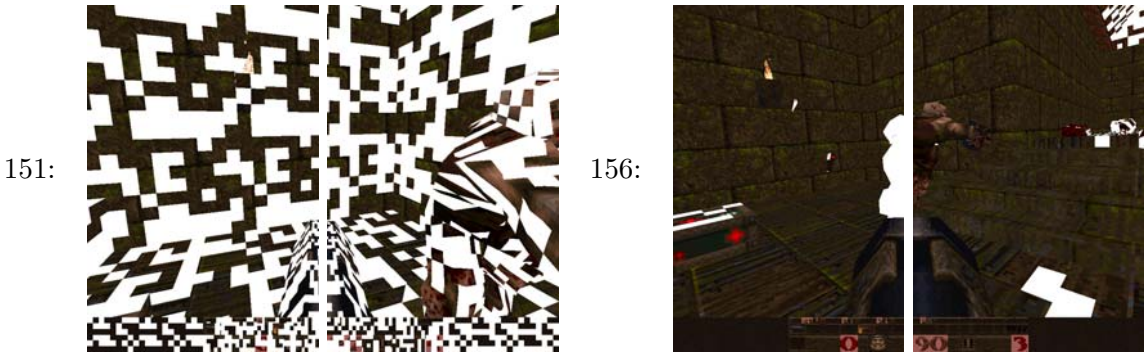


Figure 4.9: Frames 151 and 156 from the GLQuake trace, split left/right across 2 GPUs. Before Frame 151, all texture pages (size 8×8) were randomly assigned to one of the two GPUs; white textures are requested pages. For this experiment, we have artificially limited all pages to reside on a GPU for only a single frame unless the page is used for consecutive frames. After 5 frames, most pages have successfully migrated to their local GPU, minimizing the number of required remote reads.

increases performance. Fortunately, real applications typically exhibit such a coherence, as we show in Figure 4.9. 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 4.8). 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.

# Chapter 5

# Discussion

The system we present here meets our functionality goals, but does not yet deliver efficient performance for several reasons, including vendor hardware support and software limitations in our system. These performance problems and system limitations lead us to discuss future hardware and software support that would be necessary for a performance oriented implementation. We also discuss the implications of some upcoming hardware and software to our work. Also, through future work, we believe we can increase the robustness of the system, and the scale of its implementation.

## 5.1 Performance

### 5.1.1 CPU/GPU Communication

One of the main bottlenecks in our system is processing read requests from fragment programs with more than one texture access. DirectX 9 hardware only supports up to four render targets, so each program must be separated into multiple passes, incurring the overhead of rerendering the scene for each pass, and incurring a full screen readback per pass per render target. More render targets would help to reduce the cost of rerendering the scene. The need for multiple render targets could be completely alleviated if faulting within shaders was allowed (this is discussed in Chapter 5.2).

A second major bottleneck in the system is the readback of read and write requests. When reading back requests from the boiling application, every pixel writes to texture and

thus makes an exclusive request. Currently we try to mitigate this cost by stream compacting the results. 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.

### 5.1.2 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.

## 5.2 Limitations in our System

### 5.2.1 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 that shows an operation that we cannot current handle 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 (Chapter 3.4). The first pass must produce a remote texture page address for each of those two fragments, but those two addresses must be stored in a single pixel location as a single request. 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 [11] at the cost of one pass per layer. The proper solution would be hardware support of an F-buffer [15, 22] to store the intermediate fragments, coupled with the ability to read those fragments back to the CPU. DirectX 12 is targeting order-independent transparency with possible hardware data structures, like the A-Buffer [6], that store multiple fragments per pixel; this sort of support also applies to the one-fragment-per-pixel problem.

### 5.2.2 Mipmapping

Modern graphics hardware filters textures through the mipmap [33], 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 ad-

dress and returning a single (filtered) texture value. Mipmapping texture page addresses or across pages in a physical memory texture 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. A second approach would be to add a border to each page, and to have a mipmapped page table where each page and all lower levels of the mipmap are stored in the same location. The border would allow bilinear filtering to occur on each mip level, but it would complicate addressing. Also, it would complicate consistency assurance because the texels in the border would be located in multiple pages. The best long-term solution would be exposing filtering to the programmer within the fragment program. This way, the eight lookups could come from multiple pages and still take advantage of hardware acceleration.

### 5.2.3 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 [27] offers a starting point for a possible solution.

### 5.2.4 Texture binding limit

Our current implementation creates extra physical memory textures as more texture data needs to be stored on the GPU. Current shader models require all needed textures to be bound prior to rendering a polygon. This means that in the final render pass, the page table plus all physical memories must be bound to the fragment shader. In a system with a lot of texture data, the number of textures bound to a fragment program can hit hardware limits quickly. Current hardware only supports up to 16 textures to be bound to

a fragment shader. In order to select the proper texture, a set of conditionals is required to pick the correct physical memory texture to read from.

DirectX 10 will support an arrayed texture construct [2]. This construct allows up to 512 textures to be combined together in one object. A texture look up will consist of a texture coordinate and an array index. This is a significant increase over 16 and will decrease the amount of conditionals, but it still is a hard limit. A better solution to this problem would be to allow textures to be bound on the fly from within a fragment program. This would create almost infinite space for physical memories and alleviate the need for a set of conditionals within the fragment program to determine which physical memory the page resides in.

## 5.3 Implications for Graphics Hardware

Though the 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 support 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 WDDM [26]. 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. WDDM will allow for multi-GPU configurations, but the exact memory model that will be present is not clear at this point. It has been stated that multi-GPU memory will run in an instanced mode or a mirrored mode. The conditions for a multi-GPU configuration to run in either of these modes have not been released. We assume that the instanced memory mode would allow each GPU to fault independently and have a unique set of memory blocks, whereas the mirrored mode would act similar to Crossfire or SLI by duplicating all memory blocks.

We believe that the paging infrastructure created by WDDM could be leveraged to distribute texture memory over multiple GPUs. The system as it stands could be used in place of the page table model presented in Chapter 3. By allowing the programmer access to texture pages and the ability to override the page fault handlers, consistency mechanisms and a directory for multiple GPUs could be implemented using WDDM. Since details for multi-GPU configurations are not available, implications for the threading systems are not clear.

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.

## 5.4 Future Work

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 could substantially impact our implementation is scatter. Current ATI and NVIDIA hardware support generalized scatter, but our system was implemented before this primitive was available. If we migrated the system to use scatter it would allow us to reduce the amount of temporary data required for the write procedure (Chapter 3.5). It would allow us to write directly to pages without a extra render pass to copy data from a temporary buffer and would also alleviate the need for a write mask texture. As discussed in Chapter 4, our system does not have great performance, so it is unknown how much performance benefit it would see from scatter.

There are several other performance and functional improvements that could be added to the system, and we describe them in the remainder of this chapter.

### 5.4.1 Eviction strategies

Currently, there are no eviction policies for pages loaded into GPU physical memory. In a standard cache, blocks can be evicted due to capacity and conflict misses. Our

implementation does not have conflict misses because the physical memory textures are essentially fully associative caches. There are also no capacity misses because new physical memories are created on the fly as new pages need to be loaded to the GPU. This means that eventually we will run out of GPU texture memory. Though it would be nice to implement a standard cache structure, there are factors imposed by our implementation and the GPU pipeline that disallow this.

If our physical memory acted like a n-way associative cache and evicted pages due to conflict misses and used a scheme for capacity misses, such as least recently used, problems would arise. Currently, our system only requests pages that are not valid in physical memory and are deemed needed. This means that there may be pages that are already resident in texture memory that will be necessary to generate the frame. If, when we load the requested pages, we evict pages that were not requested, we could be evicting pages that were needed by the frame. This means that we must have some knowledge of all data used in every frame.

A simple solution would be to request every page required by the frame regardless of whether or not it is valid in GPU memory. This will increase the amount of data needed to be read back to the CPU. Once all the pages have been received by the memory manager, it can proceed to upload only the pages that are not valid. If memory limitations are hit, it can then evict pages that do not exist in the list of needed pages. There is no guarantee that the amount of texture data needed for a frame will fit into the given physical memories, so additional physical memory textures will still need to be able to be generated.

The primary problem is that we are working at the granularity of frames. We can only determine data dependencies for a single frame, which is a batch of requests from all fragments that are generated. An ideal implementation would be at the hardware level. If we could work at a fragment granularity, then evicting data would not cause inconsistencies, because if another pixel needed evicted data it could request it. We discuss this in Chapter 5.3.

### 5.4.2 Moving to a Clustered Environment

One goal of this work was to allow scalability to multiple nodes, each having one or more GPUs installed. Though multi-node configurations are not currently supported, the system was designed to make the move to multi-node systems as easy as possible. The main changes to the system are splitting the directory, slightly modifying the threading model, and adding some extra communication pathways.

In order to distribute the system over many nodes, the directory must be split. This can be done in a similar fashion to the DASH machine [20]. In terms of our system, each node will have its own memory manager which is in charge of a certain subset of the global address space. The memory managed by each node's directory will be accessible to any GPU on any node in the system. This has the side effect that certain data distributions will incur bad performance. If many GPUs wish to access memory from the same node in rapid succession, the directory on that node will quickly become a bottleneck.

When moving to a multi-node environment, where data accesses on remote nodes can incur high latency, some aspects of the memory system may need to be exposed to the programmer. In particular, the memory location to which the data is loaded to can be a large factor in the performance of a system, and should be configurable by the programmer. If this is not a viable solution, tuning mechanisms should be available that will test memory access patterns of an application and determine a favorable data distribution.

Currently, the event handlers in our threading model will batch all texture requests for a single frame into one command that will be submitted to a single memory manager. When moving to a cluster, the event handlers must be able to submit these requests to any memory manager in the system. In order to support this functionality, two major changes to the event handler are required. First, the event handler must be able to receive a batch of requests from the GPU and determine which memory manager holds the directory entry for each page. This will allow the event handler to make several batches of requests, each intended for a different memory manager. Secondly, the event handler must be able to issue multiple memory requests and service their replies in any order. Once all replies have been received, it may proceed with the next pass. This will allow the event handler to

simultaneously request all required pages from each memory manager in the system.

Similar to the event handler, each memory manager in the system must be able to issue a command to any OpenGL thread in the system. Based upon the requests it receives from the event handler, the memory manager will always know which OpenGL thread requires data.

Thus far we have described the necessary communication pathways needed for migration to a multi-node environment, but not how they will be implemented. Our system is command driven, therefore it makes sense to use a message passing system for an underlying implementation. Earlier we stated that message passing was not a favorable solution to our system, but this was because it would leave too much in hands of the programmer. In a clustered implementation, the message passing components of our system would be transparent to the user. Using this transparent message passing system, memory managers and event handlers on each node will be able to package commands into a message and forward them to the proper threads on other nodes.

### 5.4.3 Towards Efficient Memory Usage

In the current implementation of our system, GPU texture memory is treated as a cache to main memory. This obviously creates redundant data storage, as CPU memory has a copy of every page. A more efficient allocation scheme would allow GPU texture memory to be an extension of main memory, rather than a set of cached copies of pages.

Technically, our system does not require each page to be mirrored in CPU memory. Currently, our system operates similar to a non-uniform memory architecture (NUMA), but it could be modified to operate like a cache-only memory architecture (COMA). A COMA is a multiprocessor memory system composed of only caches. When applying a COMA to our system, the only time when pages would need to be resident in CPU memory would be if GPU memory is full. Otherwise they would be cached on at least one GPU. By keeping a copy of the pages in main memory, it reduces the access time when a GPU requests a page. If a page is not dirty, it currently takes two transactions across the PCI-E bus: 1) GPU to CPU request and 2) CPU to GPU response. In the case where pages only reside in GPU memory, a page fault could take up to four transactions because GPUs cannot communicate

directly with each other and require the CPU to service the requests and responses.

One fundamental limitation to giving GPUs explicit control over their memory is that they cannot operate fully independently of the CPU. The CPU controls the GPU by sending it a stream of commands, so, even at the low level of the driver, the CPU is still managing memory for the GPU. In the future, as required by the WDDM, GPUs will support virtual memory. This will allow GPUs to fault when memory they need is not local. This gives the GPU a little more control over the global view of memory, but in general a fault just turns into a request that the CPU services.

Though it would seem that giving GPUs explicit control of their memory would be beneficial, in a multi-GPU environment there still must be a memory consistency mechanism that is common to all GPUs. Since current GPUs are managed by CPU-driven command streams, it only makes sense that the CPU should also manage memory consistency. Therefore, we should look to minimize the mirroring of data in the system.

One way to minimize the amount of redundantly mirrored data would be to view the CPU as another consumer of pages. The CPU would still manage the directory, but instead of storing a copy of each page in the directory, the CPU would have a separate page table and corresponding physical memory to store some subset of the pages loaded into the system. The memory consistency model would still work with the directory to ensure consistency, but it would not always have a guaranteed copy of a page in CPU memory. Storage of pages in CPU memory cannot be totally eliminated because it is necessary to cover the case when a texture is loaded and will not fit into GPU texture memory. In this case, the texture can be loaded into the CPU physical memory to save it for future access by GPUs. Currently, the GPU is seen as a cache of texture memory, and if system memory is an overflow for GPU memory it would be acting like another level of caching hierarchy. This setup would move our system from a NUMA to a COMA architecture. A second usage of the CPU memory structure would be for instances when texture data is operated on by the CPU, such as a read-pixels or a read-texture command. This way the pages can be located in CPU space for access as well. Lastly, when implementing eviction strategies for GPU texture memory, the CPU memory would be the destination for evicted pages.

The proposed system would increase the time for inter-GPU requests to be ser-

viced. To alleviate this, the CPU memory could hold a copy of pages that are in high contention between multiple GPUs to allow for quicker access. Currently, the system has no notion of a high contention page, and support for identifying these pages must be added to the system. This is easily accomplished through counters and timers associated with thresholds. Any page marked as in high contention could be kept in system memory like a CPU L2 cache that holds copies of data in the L1 cache. A write-through cache system would be necessary to keep the data in system memory up to date.

# Chapter 6

# Conclusion

The system we have created supports a distributed-shared memory abstraction for scalable, distributed texture memory over multiple GPUs. While our implementation has limitations in performance and generality, we feel that it makes significant contributions to multi-GPU computing.

**Mechanisms** We have identified many mechanisms necessary to create a successful distributed texture memory system. Most importantly, a distributed memory system must define its consistency model. If a memory system does not have a definable consistency model, the programmer will have no knowledge or guarantees of how the system will operate. With memory consistency in mind, we investigated the necessary data structures for both the GPU and CPU to support our memory system. Finally, by using these data structures within the constraints of the graphics pipeline, we were able to develop read and write procedures that allowed GPUs to access memory in a consistent and coherent fashion.

**Limitations** Though our system successfully implements a distributed texture memory system, it is not without limitations. By approaching the problem at the software level we ran into many limitations that hindered both performance and functionality. Any memory hierarchy will need to efficiently handle faulting when accessing nonresident data. The fact that our system is crippled by reading this fault data back to the CPU shows how important it is to have an efficient faulting and handling scheme. Some improvements

could be seen with an efficient implementation of the uniquify primitive, but it is also conceivable that the faulting functionality could be implemented at a lower level to achieve higher performance as well. There are also several functional limitations such as the flow control, one fragment per pixel, and texture binding limit problems that could easily be alleviated though advancements in graphics hardware.

**Threading System** Any system wishing to leverage multiple GPUs connected to a single CPU will be required to implement some type of threading system to maximize the computing capabilities of the GPUs. Without threading, a system is limited to serially switching GPU contexts or broadcasting commands to all GPUs simultaneously. Both of these options have performance and scalability limitations. We have presented a threading system to avoid such limitations, and have gone to lengths to ensure that it that works within our consistency model while avoiding deadlock.

The core contribution of this work 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.

# Bibliography

[1] ATI. *Asymmetric Physics Processing with CrossFire$^{TM}$*, 2006. `http://www.ati.com/technology/crossfire/physics/`.

[2] David Blythe. The Direct3D 10 System. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[3] David Blythe. Private communication. Microsoft, 7 June 2006.

[4] David Blythe and Tom McReynolds. Advanced graphics programming techniques using OpenGL. *ACM SIGGRAPH Course Notes*, July 2000.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[6] Loren Carpenter. The A-Buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 103–108, July 1984.

[7] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Graphics Hardware 2002*, pages 69–78, September 2002.

[8] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[9] Eric Demers. Private communication. ATI, 16 June 2006.

[10] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A fully scalable graphics architecture. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 443–454, July 2000.

[11] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. `http://developer.nvidia.com/object/Interactive_Order_Transparency.html`.

[12] Zhe Fan, Feng Qiu, and Arie Kaufman. ZippyGPU: Programming toolkit for general-purpose computation on GPU clusters. In *Supercomputing 2006 Workshop on General-Purpose GPU Computing: Practice and Experience*, November 2006.

[13] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Graphics Hardware 2002*, pages 109–118, September 2002.

[14] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: A streaming HMMer-search implementation. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Mike Houston, Arcot J. Preetham, and Mark Segal. A hardware F-buffer implementation. Technical Report CSTR 2005-05, Stanford University Department of Computer Science, 2005.

[16] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and Jim Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, July 2002.

[17] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 141–150, July 1998.

[18] Aaron E. Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 26(1):60–99, 2006.

[19] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[20] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[21] Michael Manzke, Ross Brennan, Keith O'Conor, and John Dingliana. A scalable and reconfigurable shared-memory graphics architecture. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications*, August 2006.

[22] William R. Mark and Kekoa Proudfoot. The F-Buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 57–64, 2001.

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

[24] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications*, August 2006.

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

[26] Steve Pronovost, Henry Moreton, and Tim Kelley. *Windows Display Driver Model (WDDM) v2 and Beyond*, 22 May 2006. `http://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/PRI103_WH06.ppt`.

[27] Andrew T. Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone, and John D. Owens. Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware 2004*, pages 35–44, August 2004.

[28] Richard Simoni. Implementing a directory-based cache consistency protocol. Technical Report CSL-TR-90-423, Stanford University Computer Systems Laboratory, March 1990.

[29] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[30] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 151–158, July 1998.

[31] Douglas Voorhies, David Kirk, and Olin Lathrop. Virtual graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 247–253, August 1988.

[32] Ian Williams and Mark Harris. NVIDIA Quadro Plex VCS. 4 September 2006.

[33] Lance Williams. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, volume 17, pages 1–11, Detroit, Michigan, July 1983.