Thesis

On

Present status and evolution of graphics processing unit

A Thesis Report submitted in partial fulfillment of the requirement for the degree of Bachelor of Science in Electrical and Electronic Engineering.

SUBMITTED BY

Selim khan Id NO: 091800036

Debashis Paul Id NO: 091800041

Ananda kumer Dhar Id NO: 092800042

SUPERVISED BY

Abu Shafin Mohammad Mahdee Jameel

Lecturer, Department of Electrical & Electronic Engineering

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EASTERN UNIVERSITY, DHAKA, BANGLADESH

JUNE-2012

EASTERN UNIVERSITY

Declaration

We, thereby, declare that the work presented in this thesis is the outcome of the investigation performed by us under the supervision of **Abu Shafin Md. Mahdee Jameel.** Lecturer, Department of Electrical & Electronic Engineering, Eastern University. We also declare that no part of this thesis has submitted elsewhere for the award of any degree.

Signature of the Candidates:

External

SELIM KHAN

ID # 091800036

DEBASHIS PAUL

ID # 091800041

ANANDA KUMER DHUR

ID # 092800042

Supervisor

Abu Shafin Md. Mahdee Jameel
Lecturer, Department of Electrical & Electronic Engineering
Eastern University, Bangladesh

Ashif Iqbal Sikder

Lecturer,

Department of Electrical & Electronic Engineering

Eastern University, Bangladesh

Approval

This thesis report implementation of present status and evolution of graphics processing unit. Has submitted to the following respected members of the board of examiners of the faculty of engineering in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical & Electronics Engineering on June of 2012 by the following student and has accepted as satisfactory.

1. SELIM KHAN	**ID # 091800036**
2. DEBASHIS PAUL	**ID # 091800041**
3. ANANDA KUMER DHAR	**ID #092800042**

Supervisor	**Chairperson**
---	---
Abu Shafin Mohammed Mahdee Jameel	**Prof.Dr.Mirza Golam Rabbani.**
Lecturer,	**Dean in charge, Faculty of E&T**
Department of Electrical & Electronic	**Chairperson, Department of EEE**
Engineering, Eastern University.	**Eastern University, Bangladesh.**

Acknowledgement

First, we would like to thank and express our gratefulness to Almighty Allah for giving us the strength and energy to complete this thesis successfully.

We wish to acknowledge our gratefulness to our thesis supervisor **Abu Shafin Md. Mahdee Jameel**. Lecturer, Department of Electrical & Electronic Engineering. For his valuables advice, endless patience, important suggestions, and energetic supervision and above all scholarly guidance from beginning to the end of the thesis work.

We would like to express our heartiest gratefulness to all of our teachers here at the Department of Electrical & Electronic Engineering, Eastern University of Bangladesh. Discussions with many of them have enriched our conception and knowledge about this thesis work .we would like to thank the department of Electrical & Electronic Engineering, Eastern University for giving us the opportunity to work in "Computer lab" and for providing the necessary information.

Abstract

The raw compute performance of today's graphics processor is truly amazing. Today's commodity CPU at apricot only a few hundred dollars. As the programmability and performance of modern graphics hardware continues to increase, many researchers are looking to graphics hardware to solve computationally intensive problems previously performed on general purpose CPUs.

The challenge, however, is how to re-target these processors from game rendering to general computation, such as numerical modeling, sciatica computing, or sign al processing. Traditional graphics API s abstract the GPU as a rendering device, involving textures, triangles, and pixels. Mapping an algorithm to use these primitives is not a straightforward operation, even for the most advanced graphics developers. There salts were cult and often unmanageable programming approaches, hindering the overall adoption of GPUs as a mainstream computing device.

Contents

CHAPTER-6

CHAPTER-7

CHAPTER-8

List of Figures

Chapter: 1

1.1: Introduction

In the early 1990s ubiquitous interactive 3D graphics was still the stuff of science fiction. By the end of the decade, nearly every new unit (GPU) dedicated to providing a high performance, visually rich, interactive 3D experience. This dramatic shift was the inevitable consequence of consumer demand for videogames, advances in manufacturing technology, and the exploitation of the inherent parallelism in the feed-forward graphics pipeline. Today, the raw computational power of a GPU dwarfs that of the most powerful CPU, and the gap is steadily widening. Furthermore, GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine. Today, GPUs can implement many parallel algorithms directly using graphics hardware. Well-suited algorithms that leverage all the underlying computational horsepower often achieve tremendous Speedups. Truly, the GPU is the first widely deployed commodity desktop parallel computer.

1.2: Overview

Computer viruses, bot clients, root kits, and other types of malicious software, collectively referred to as *malware*, abuse infected hosts to carry out their malicious activities. From the first viruses written directly in assembly language, to application-specific malicious code written in high-level languages like javascript, any action of the malware results in the execution of machine code on the compromised system's processor.

Besides the central processing unit, modern personal computers are equipped with another powerful computational device: the graphics processing unit (GPU). Historically, the GPU has been used for handling 2D and 3D graphics rendering, effectively offloading the CPU from these computationally-intensive operations. Driven to a large extent by the ever-growing video game industry, graphics processors have been constantly evolving, increasing both in computational power and in the range of supported operations and functionality. The most recent development in the evolution chain is generalpurpose computing on GPUs (GPGPU), which allows programmers to exploit the massive number of transistors in modern GPUs to perform computations that up till now were traditionally handled by the CPU. In fact, leading vendors like AMD and NVIDIA have released software development kits that allow

programmers to use a C-like programming language to write general-purpose code for execution on the GPU. GPGPU has been used in a wide range of applications, while the increasing programmability and functionality of the latest GPU generations allows the code running on the GPU to fully cooperate with the host's CPU and memory.

Given the great potential of general-purpose computing on graphics processors, it is only natural to expect that malware authors will attempt to tap the powerful features of modern GPUs to their benefit. Two key factors that affect the lifetime and potency of sophisticated malware are its ability to evade existing anti-malware defenses and the effort required by a malware analyst to analyze and uncover its functionality often a prerequisite for implementing the corresponding detection and containment mechanisms. Packing and polymorphism are among the most widely used techniques for evading malware scanners. Code obfuscation and anti-debugging tricks are commonly used to hinder reverse engineering and analysis of (malicious) code. So far, these evasion and anti-debugging techniques take advantage of the intricacies of the most common code execution environments. Consequently, malware defense and analysis mechanisms, as well as security researchers' expertise, focus on IA-32, the most prevalent instruction set architecture (ISA). The ability to execute general purpose code on the GPU opens a whole new window of opportunity for malware authors to significantly raise the bar against existing defenses. The reason for this is that existing malicious code analysis systems primarily support IA-32 code, while the majority of security researchers are not familiar with the execution environment and ISA of graphics processors.

Furthermore, we discuss potential attacks and future threats that can be facilitated by next-generation GPGPU architectures. We believe that a better understanding of the offensive capabilities of attackers can lead researchers to create more effective and resilient defenses.

Chapter: 2

2.1: Definition of GPU

A GPU (Graphics Processing Unit) is essentially a dedicated hardware device that is responsible for translating data into a 2D image formed by pixels. In this paper, we will focus on the 3D graphics.

- It is a processor optimized for 2D/3D graphics, video, visual computing, and display.
- It is highly parallel, highly multithreaded multiprocessor optimized for visual computing.
- It provides real - time visual interaction with computed objects via graphics images, and video.
- It serves as both a programmable graphics processor and a scalable parallel computing platform.
- Heterogeneous Systems: combine a GPU with a CPU

2.2: The History of the GPU

It's one thing to recognize the future potential of a new processing architecture. It's another to build a market before that potential can be achieved. There were attempts to build chip-scale parallel processors in the 1990s, but the limited transistor budgets in those days favored more sophisticated single-core designs. The real path toward GPU computing began, not with GPUs, but with nonprogrammable 3G-graphics accelerators. Multi-chip 3D rendering engines were developed by multiple companies starting in the 1980s, but by the mid-1990s it became possible to integrate all the essential elements onto a single chip. From 1994 to 2001, these chips progressed from the simplest pixel-drawing functions to implementing the full 3D pipeline: transforms lighting, rasterization, texturing, depth testing, and display.

NVIDIA's GeForce 3 in 2001 introduced programmable pixel shading to the consumer market. The programmability of this chip was very limited, but later GeForce products became more flexible and faster, adding separate programmable engines for vertex and geometry shading. This evolution culminated in the GeForce 7800. So-called general-purpose GPU (GPGP) programming evolved as a way to perform non-graphics processing on these graphics-optimized architectures, typically by running carefully crafted shader code against data presented as vertex or texture information and retrieving the results from a later stage in the pipeline. Though sometimes awkward, GPGPU programming showed great promise.

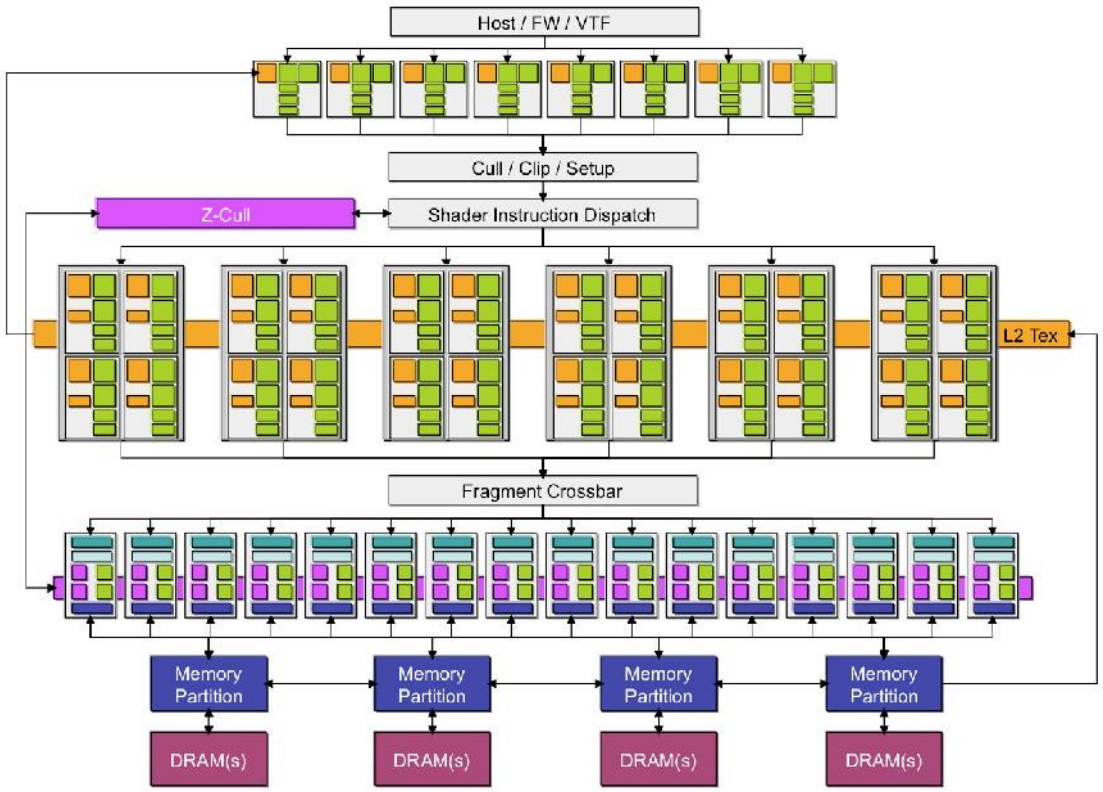


Figure1: The GeForce 7800 had three kinds of programmable engines for different stages of the 3D pipeline plus several additional stages of configurable and fixed-function logic.

Managing three different programmable engines in a single 3D pipeline led to unpredictable bottlenecks; too much effort went into balancing the throughput of each stage. In 2006, NVIDIA introduced the GeForce 8800, as Figure shows. This design featured“unified shader architecture” with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance. The 8800 also introduced CUDA, the industry’s first C-based development environment for GPUs. (CUDA originally stood for “Compute Unified Device Architecture,” but the longer name is no longer spelled out.) CUDA delivered an easier and more effective programming model than earlier GPGPU approaches.

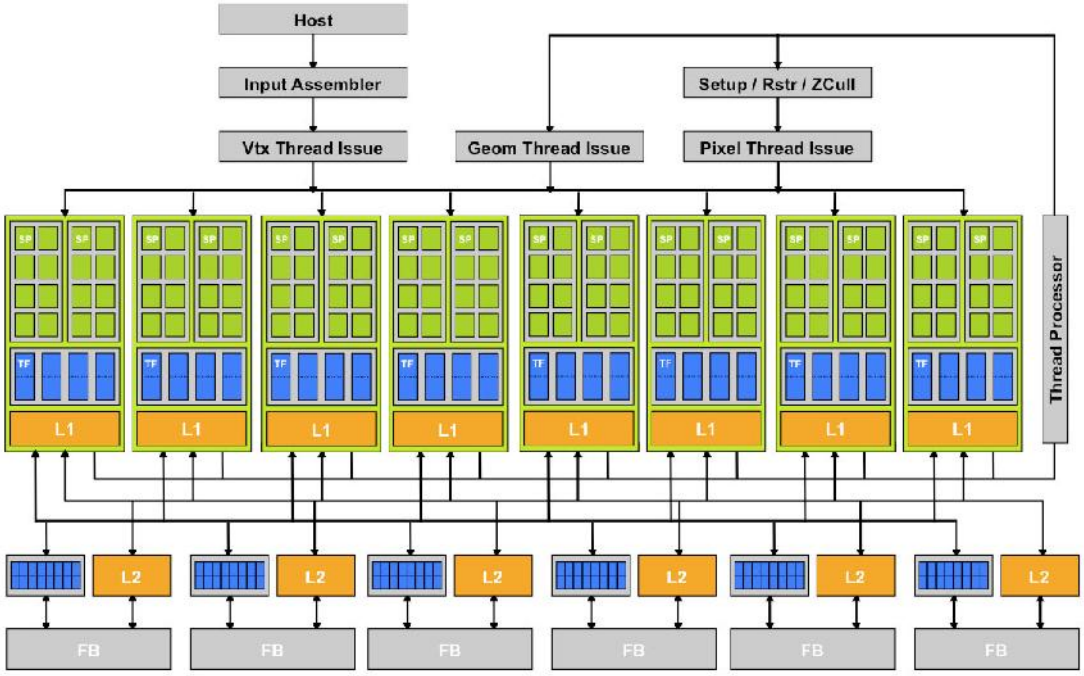


Figure2: The GeForce 8800 introduced unified shader architecture with just one kind of programmable processing element that could be used for multiple purposes. Some simple graphics operations still used special-purpose logic.

At the time of this writing, the price for the entry-level Tesla C1060 add-in board is under $1,500 from some Internet mail-order vendors. That's lower than the price of a single Intel Xeon W5590 processor—and the Tesla card has a peak GFLOPS rating more than eight times higher than the Xeon processor.
The Tesla line also includes the S1070, a 1U-height rack mount server that includes four GT200-series GPUs running at a higher speed than that in the C1060 (up to 1.5 GHz core clock vs. 1.3 GHz), so the S1070's peak performance is over 4.6 times higher than a single C1060 card. The S1070 connects to a separate hostcomputer via a PCI Express add-in card.

Although GPU computing is only a few years old now, it's likely there are already more programmers with direct GPU computing experience than have ever used a Cray. Academic support for GPU computing is also growing quickly. NVIDIA says over 200 colleges and universities are teaching classes in CUDA programming; the availability of OpenCL (such as in the new "Snow Leopard" version of Apple's Mac OS X) will drive that number even higher.

Chapter: 3

3.1: GPU Architecture

The GPU has always been a processor with ample computational resources. The most important recent trend, however, has been exposing that computation to theprogrammer. Over the past few years, the GPU has evolved from a fixed-function special-purpose processor into a full-fledged parallel programmable processor with additional fixed-function special-purpose functionality. More than ever, the programmable aspects of the processor have taken center stage. We begin by chronicling this evolution, starting from the structure of the graphics pipeline and how the GPU has become a general-purpose architecture, then taking a closer look at the architecture of the modern GPU.

3.2: Evolution of GPU Architecture

The fixed-function pipeline lacked the generality to efficiently express more complicated shading and lighting operations that are essential for complex effects. The key step was replacing the fixed-function per-vertex and per-fragment operations with user-specified programs run on each vertex and fragment. Over the past six years, these vertex program and fragment programs have become increasingly more capable, with larger limits on their size and resource consumption, with more fully featured instruction sets, and with more flexible control-flow operations. After many years of separate instruction sets for vertex and fragment operations, current GPUs support the unified Shader Model 4.0 on both vertex and fragment shader.

• The hardware must support shader programs of at least 65 k static instructions and unlimited dynamic instructions.

• The instruction set, for the first time, supports both 32-bit integers and 32-bit floating-point numbers.

• The hardware must allow an arbitrary number of both direct and indirect reads from global memory (texture).

• Finally, dynamic flow control in the form of loops and branches must be supported.

As the shader model has evolved and become more powerful, and GPU applications of all types have increased vertex and fragment program complexity, GPU architectures have increasingly focused on the programmable parts of the graphics pipeline. Indeed, while previous generations of GPUs could best be described as additions of programmability to a fixed-function pipeline, today's GPUs are better characterized as a programmable engine surrounded by supporting fixed-function units.

Some historical key points in the development of the GPU:

_ Efforts for real time graphics have been made as early as 1944

_ In the 1980s, hardware similar to modern GPUs began to show up in the research community (\Pixel-Planes", a parallel system for rasterizing and texture-mapping 3D geometry

_ Graphic chips in the early 1980s were very limited in their functionality

_ In the late 1980s and early 1990s, high-speed, general-purpose microprocessors became popular for implementing high-end GPUs

_ 1985 the first mass-market graphics accelerator was included in the Commodore Amiga

_ 1991 S3 introduced the first single chip 2D-accelerator, the S3 86C911

_ 1995 NVIDIA releases one of the first 3D accelerators, the NV1

_ 1999 NVIDIA's Geforce 256 is the first GPU to implement Transform and Lighting in Hardware

_ 2001 NVIDIA implements the first programmable shader units with the Geforce 3

_ 2005 ATI develops the first GPU with united shader architecture with the ATI Xenon for the XBox360

_ 2006 NVIDIA launches the first united shader GPU for the PC with the Geforce 8800.

3.3: Architecture of a Modern GPU

In Section I, we noted that the GPU is built for different application demands than the CPU: large, parallel computation requirements with an emphasis on throughput rather than latency. Consequently, the architecture of the GPU has progressed in a different direction than that of the CPU.

Consider a pipeline of tasks, such as we see in most graphics APIs (and many other applications), that must process a large number of input elements. In such a pipeline, the output of each successive task is fed into the input of the next task. The pipeline exposes the task parallelism of the application, as data in multiple pipeline stages can be computed at the same time; within each stage, computing more than one element at the same time is data parallelism. To execute such a pipeline, a CPU would take a single element (or group of elements) and process the first stage in the pipeline, then the next stage, and so on. The CPU divides the pipeline in time, applying all resources in the processor to each stage in turn.GPUs have historically taken a different approach. The GPU divides the resources of the processor among the different stages, such that the pipeline is divided in space, not time. The part of the processor working on one stage feeds its output directly into a different part that works on the next stage. This machine organization was highly successful in fixed-function GPUs for two reasons. First, the hardware in any given stage could exploit data parallelism within that stage, processing multiple elements at the same time. Because many task-parallel stages were running at any time, the GPU could meet the large compute needs of the graphics pipeline. Secondly, each stage's hardware could be customized with special-purpose hardware for its given task, allowing substantially greater compute and area efficiency over a general-purpose solution.

For instance, the rasterization stage, which computes pixel coverage information for each input triangle, is more efficient when implemented in special-purpose hardware's programmable stages (such as the vertex and fragment programs) replaced fixed-function stages, the special-purpose fixed function components were simply replaced by programmable components, but the task-parallel organization did not change. The result was a lengthy, feed-forward GPU pipeline with many stages, each typically accelerated by special purpose parallel hardware. In a CPU, any given operation may take on the order of 20 cycles between entering and leaving the CPU pipeline. On a GPU, a graphics operation may take thousands of cycles from start to finish. The latency of any given operation is long. However, the task and data parallelism across and between stages delivers high throughput. The major disadvantage of the GPU task-parallel pipeline is load balancing. Like any pipeline, the performance of the GPU pipeline is dependent on its slowest stage. If the vertex program is complex and the fragment program is simple, overall throughput is dependent on the performance of the vertex program. In the early days of programmable stages, the instruction set of the vertex and fragment programs were quite different, so these stages were separate.

However, as both the vertex and fragment programs became more fully featured, and as the instruction sets converged, GPU architects reconsidered a strict task-parallel pipeline in favor of a unified shader architecture, in which all programmable units in the pipeline share a single programmable hardware unit. While much of the pipeline is still task-parallel, the programmable units now divide their time among vertex work, fragment work, and geometry work (with the new DirectX 10 geometrysharers).These units can exploit both task and data parallelism.

As the programmable parts of the pipeline are responsible for more and more computation within the graphics pipeline, the architecture of the GPU is migrating from a strict pipelined task-parallel architecture to one that is increasingly built around a single unified data-parallel programmable unit. AMD introduced the first unified shader architecture for modern GPUs in its Xenos GPU in the XBox 360 (2005). Today, both AMD's and NVIDIA's flagship GPUs feature unified sharers (Fig. 3).

The benefit for GPU users is better load-balancing at the cost of more complex hardware. The benefit for GPGPU users is clear: with all the programmable power in a single hardware unit, GPGPU programmers can now target that programmable unit directly, rather than the previous approach of dividing work across multiple hardware units.

Fig3. Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores. The NVIDIA Geforce 8800 GTX (top) features 16 streaming multiprocessors of 8 thread (stream) processors each.

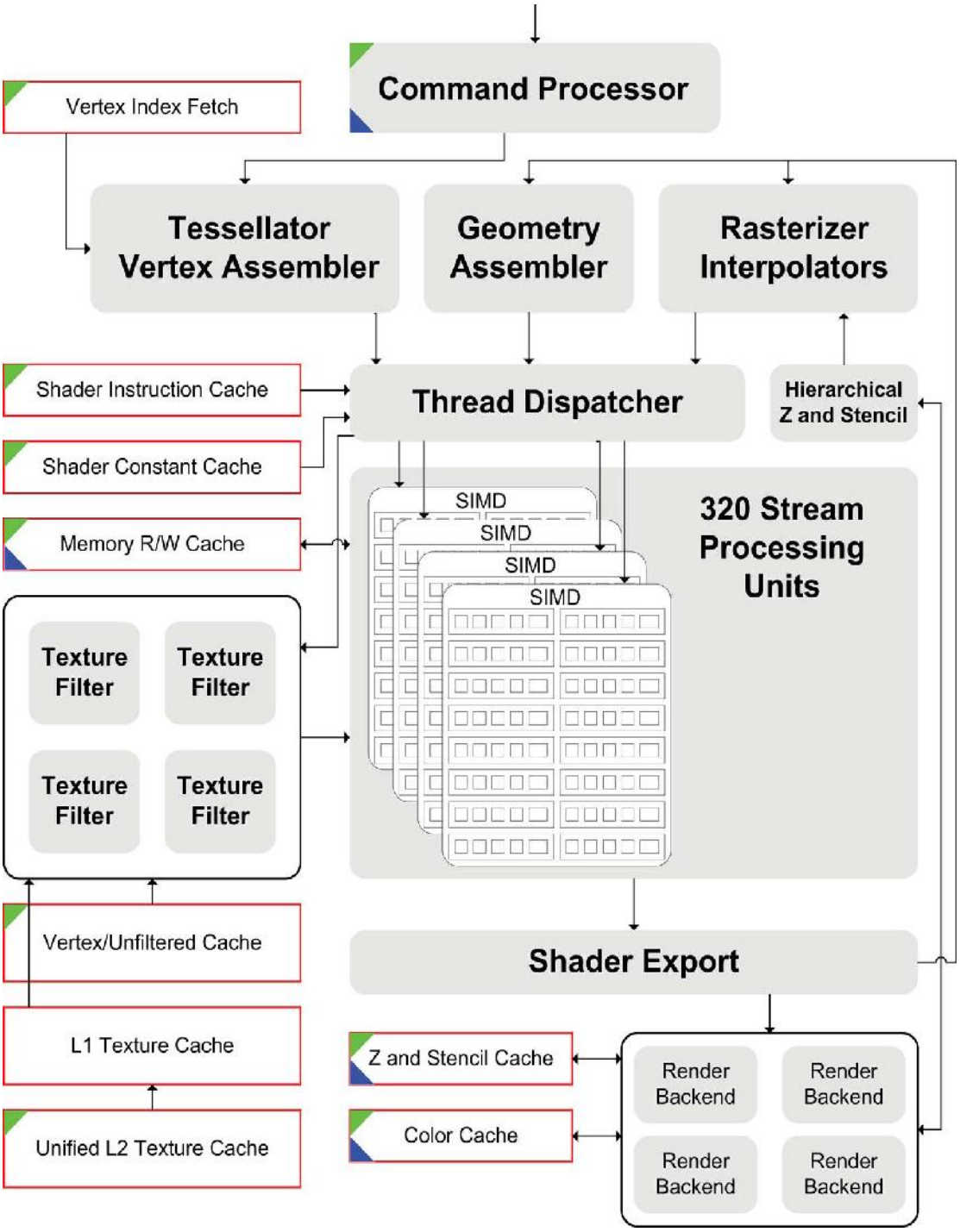


Fig4. (continued) Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores.(b) AMD's Radeon HD 2900XT contains 320 stream processing units arranged into four SIMD arrays of 80 units each.

Chapter: 4

4.1: GPU System Architectures

• CPU-GPU system architecture
– The Historical PC
– Contemporary PC with Intel and AMD CPUs
• Graphics Logical Pipeline
• Basic Unified GPU Architecture
– Processor Array

Historical PC

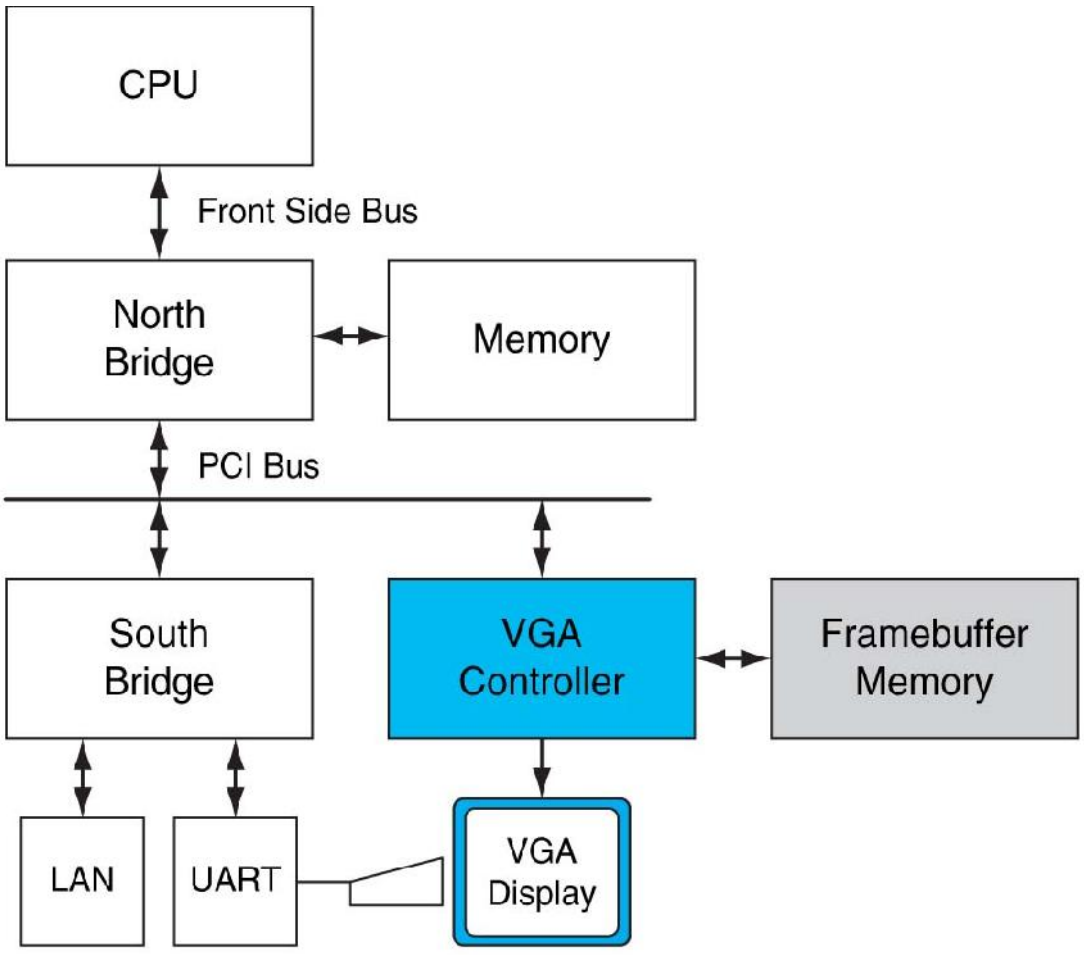


Figure5: Historical PC.

Intel and AMD CPU

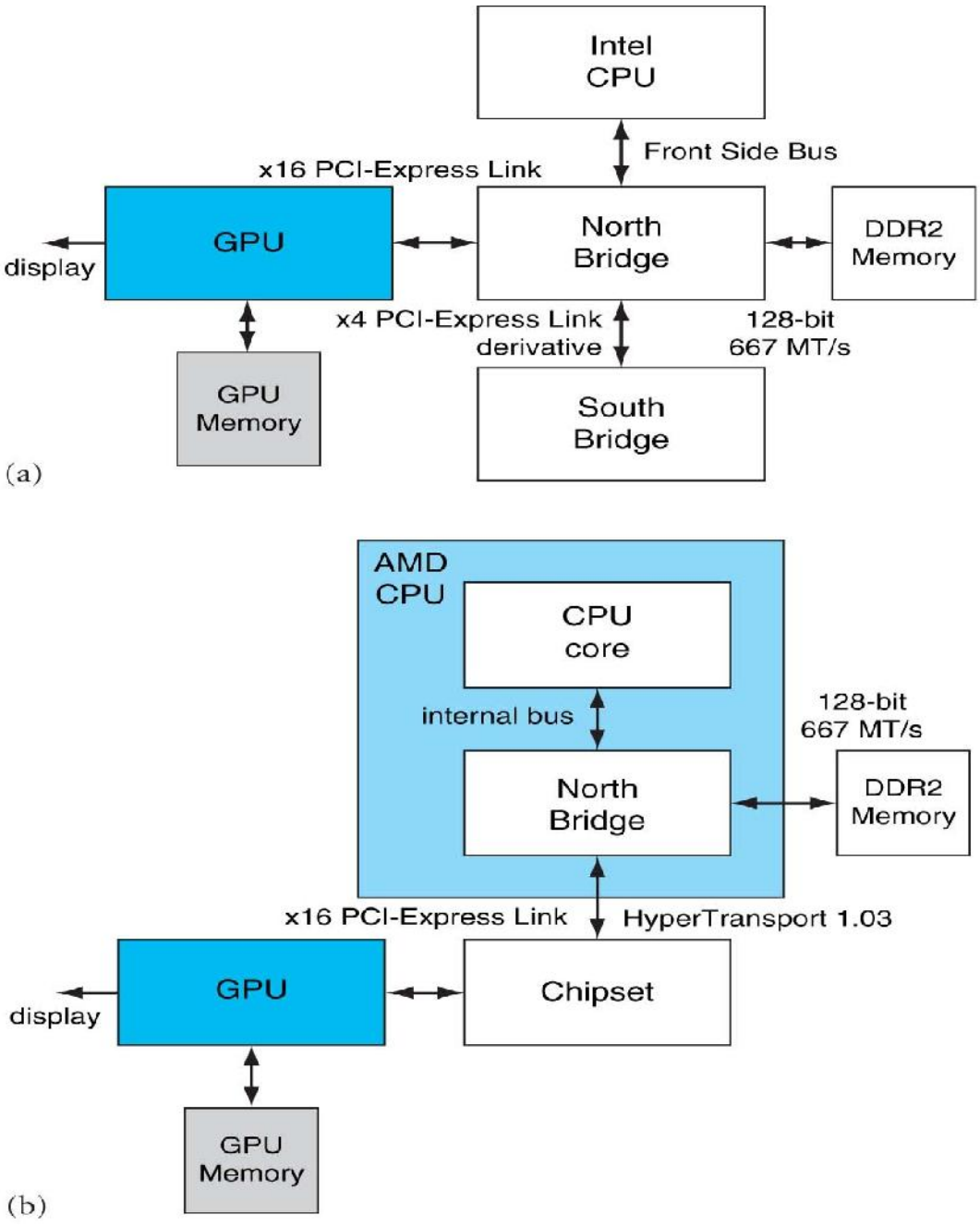


Figure 6: Contemporary PCs with Intel and AMD CPUs.

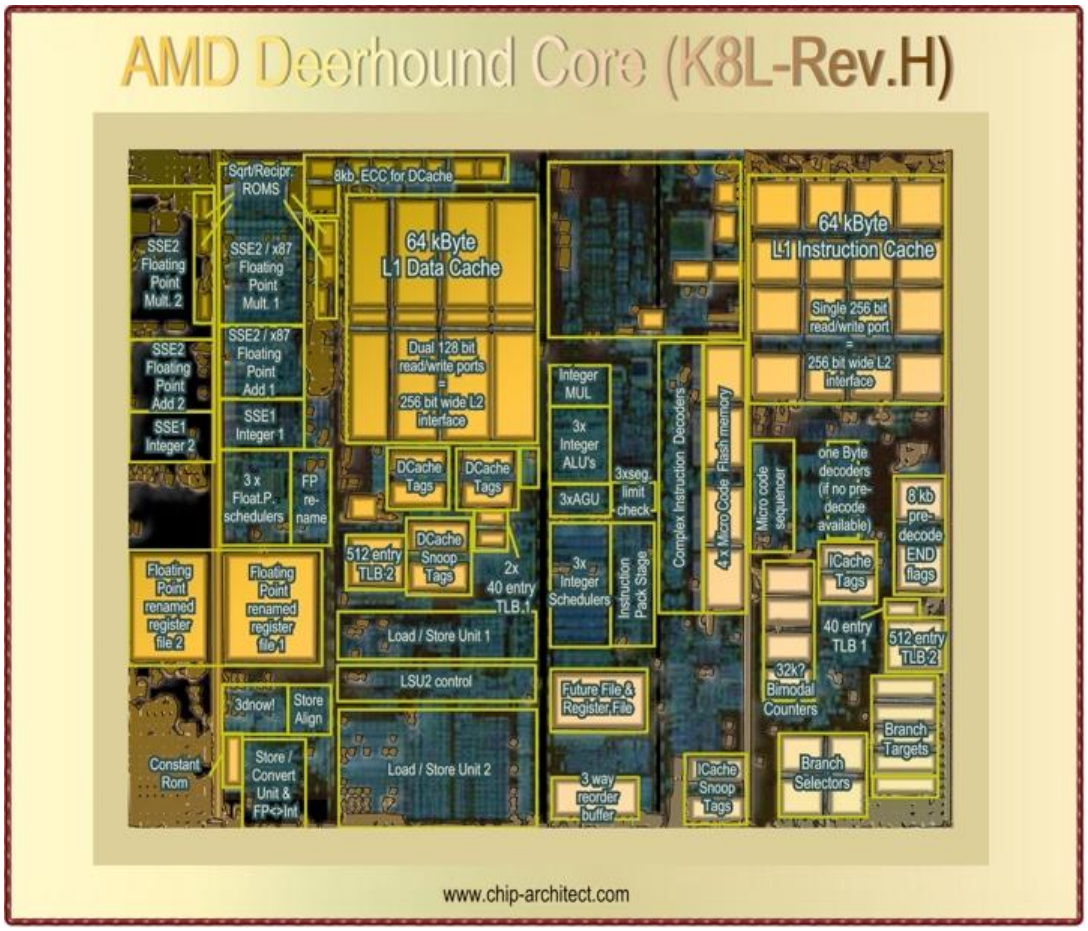
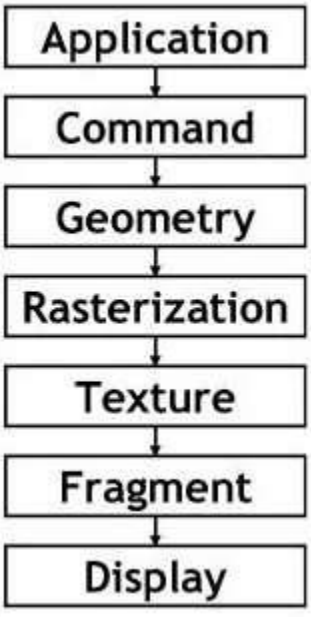


Figure7: AMD Deerhound core

The Graphics Pipeline



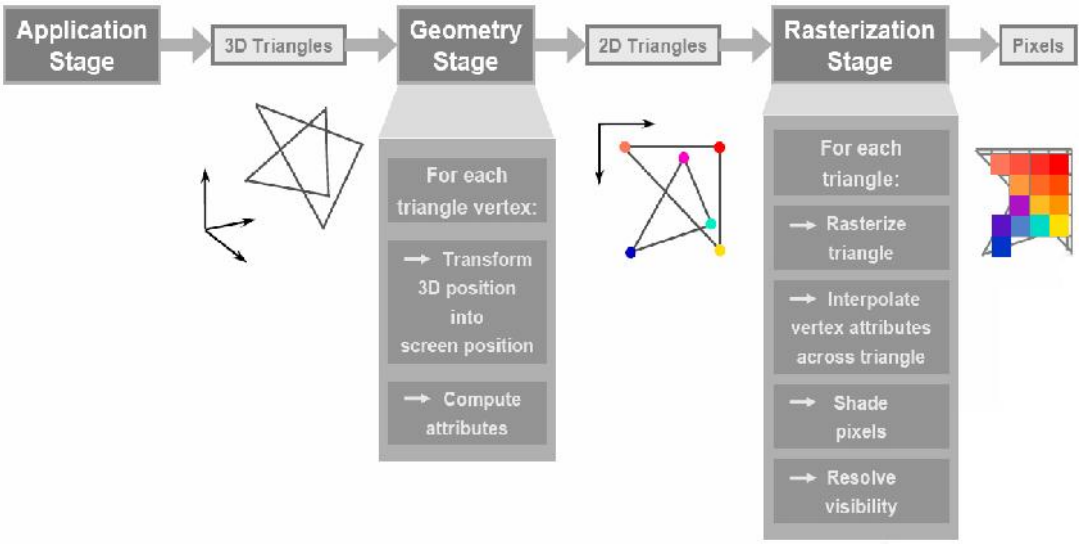


Figure8: The 3D Graphics Pipeline

First, among some other operations, we have to translate the data that is provided by the application from 3D to 2D.

Basic Unified GPU Architecture

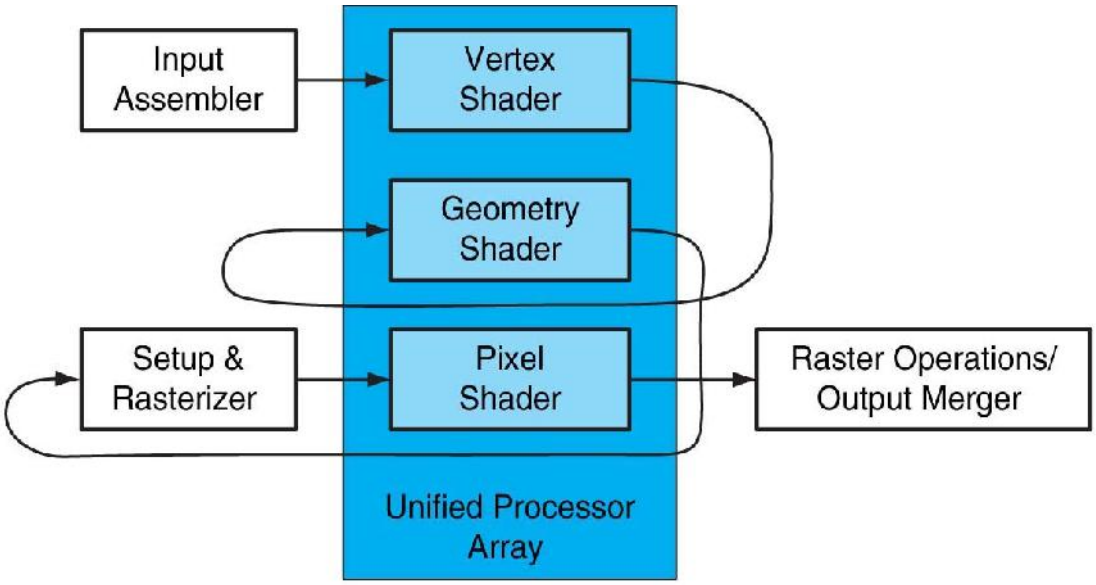


Figure9: Logical pipeline mapped to physical processors.

Processor Array

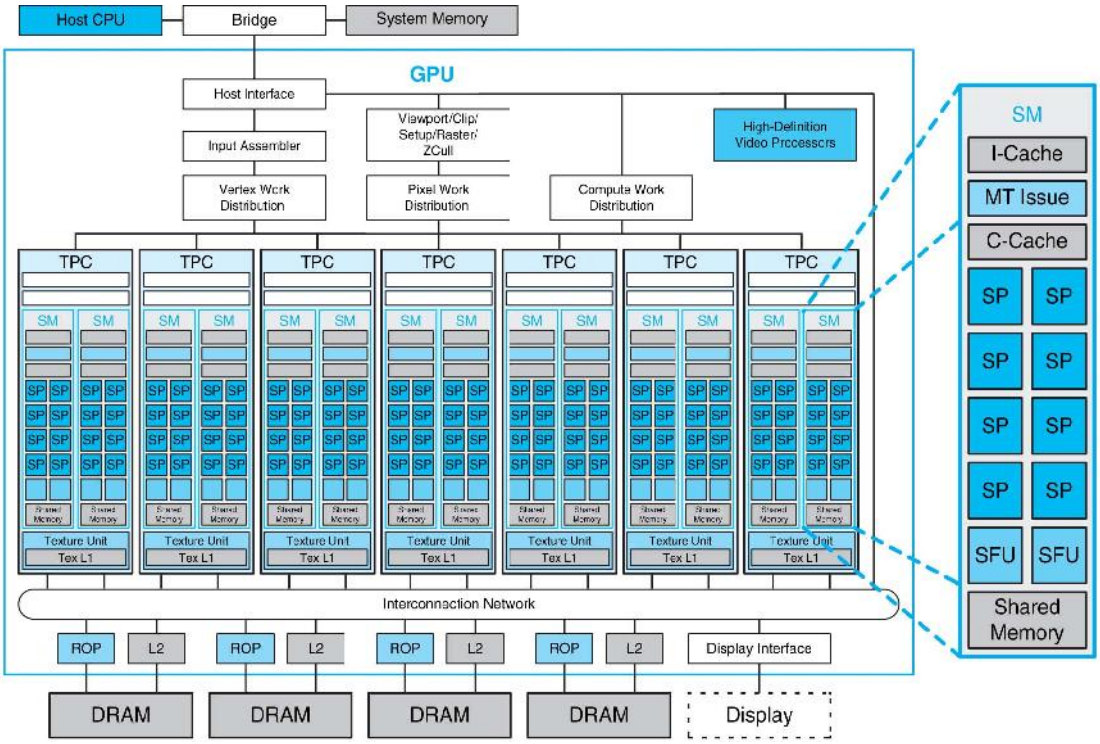


Figure10: Basic unified GPU architecture.

The input to the GPU is a list of geometric primitives, typically triangles, in a 3-D world coordinate system. Through many steps, those primitives are shaded and mapped onto the screen, where they are assembled to create a final picture. It is instructive to first explain the specific steps in the canonical pipeline before showing how the pipeline has become programmable. Vertex Operations: The input primitives are formed from individual vertices. Each vertex must be transformed into screen space and shaded, typically through computing their interaction with the lights in the scene.

Because typical scenes have tens to hundreds of thousands of vertices, and each vertex can be computed independently, this stage is well suited for parallel hardware. Primitive Assembly: The vertices are assembled into triangles, the fundamental hardware-supported primitive in today's GPUs.

Rasterization: Rasterization is the process of determining which screen-space pixel locations are covered by each triangle. Each trianglegenerates a primitive called a fragment at each screen-space pixel location that it covers. Because many triangles may overlap at any pixel location, each pixel's color value may be computed from several fragments.

Fragment Operations: Using color information from the vertices and possibly fetching additional data from global memory in the form of textures (images that are mapped onto surfaces), each fragment is shaded to determine its final color. Just as in the vertex stage, each fragment can be computed in parallel. This stage is typically the most computationally demanding stage in the graphics pipeline. Composition: Fragments are assembled into a final image with one color per pixel, usually by keeping the closest fragment to the camera for each pixel location. Historically, the operations available at the vertex and fragment stages were configurable but not programmable.

For instance, one of the key computations at the vertex stage is computing the color at each vertex as a function of the vertex properties and the lights in the scene. In the fixed-function pipeline, the programmer could control the position and color of the vertex and the lights, but not the lighting model that determined their interaction.

4.2: Characteristics of Graphics

• Large computational requirements
• Massive parallelism
– Graphics pipeline designed for independent operations
• Long latencies tolerable
• Deep, feed-forward pipelines
• Hacks are OK—can tolerate lack of accuracy
• GPUs are good at parallel, arithmetically intense, streaming-memory problems.

Chapter: 5

NVIDIA Geforce 6800

5.1: General information

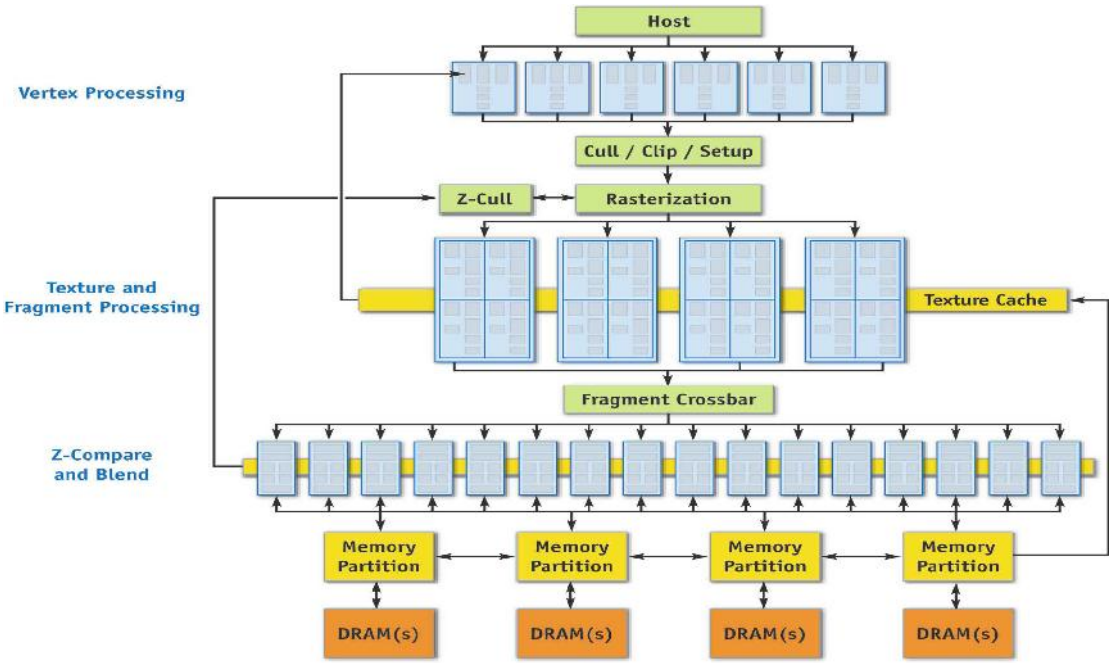
• Impressive performance stats

- 600 Million vertices/s
- 6.4 billion texels/s
- 12.8 billion pixels/s rendering z/stencil only
- 64 pixels per clock cycle early z-cull (reject rate)

• Riva series (1st DirectX compatible)
– Riva 128, Riva TNT, Riva TNT2

• GeForce Series
– GeForce 256, GeForce 3 (DirectX 8), GeForce FX, GeForce 6 series

NVIDIA Geforce 6800
Block Diagram



In Detail

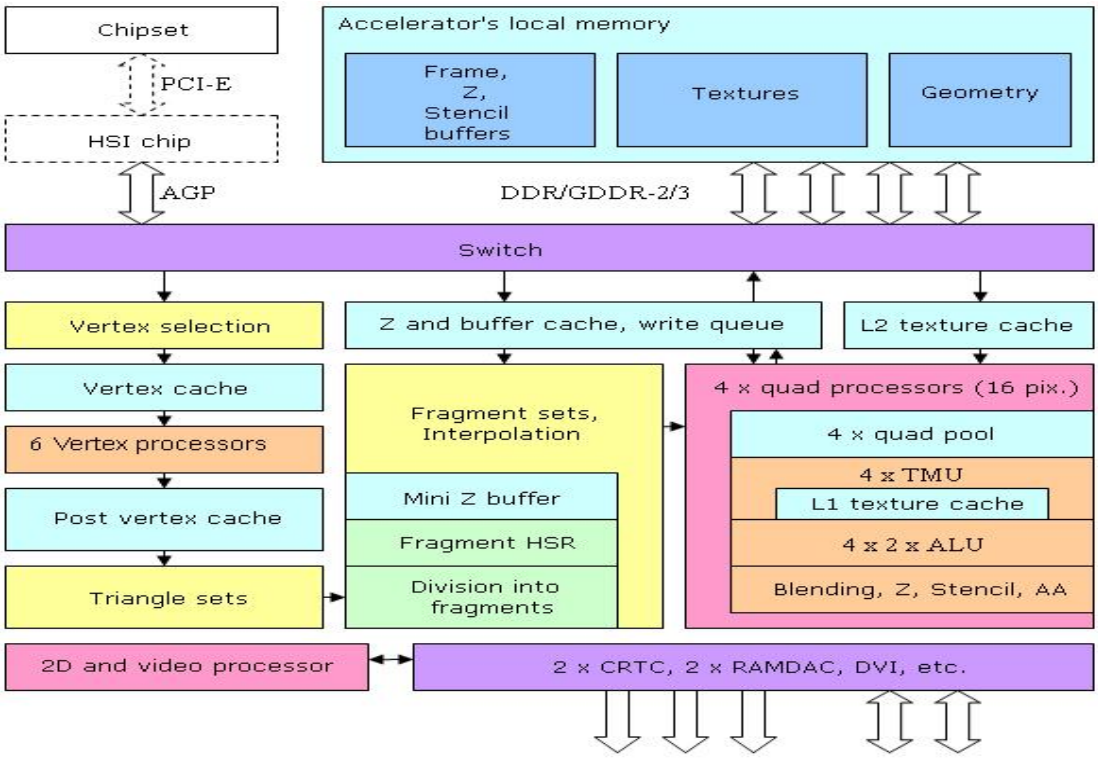


Figure11: A more detailed view of the Geforce 6800

5.2: Vertex Processor (or vertex shader)

• Allow shader to be applied to each vertex

• Transformation and other per vertex ops

• Allow vertex shader to fetch texture data (6 series only)

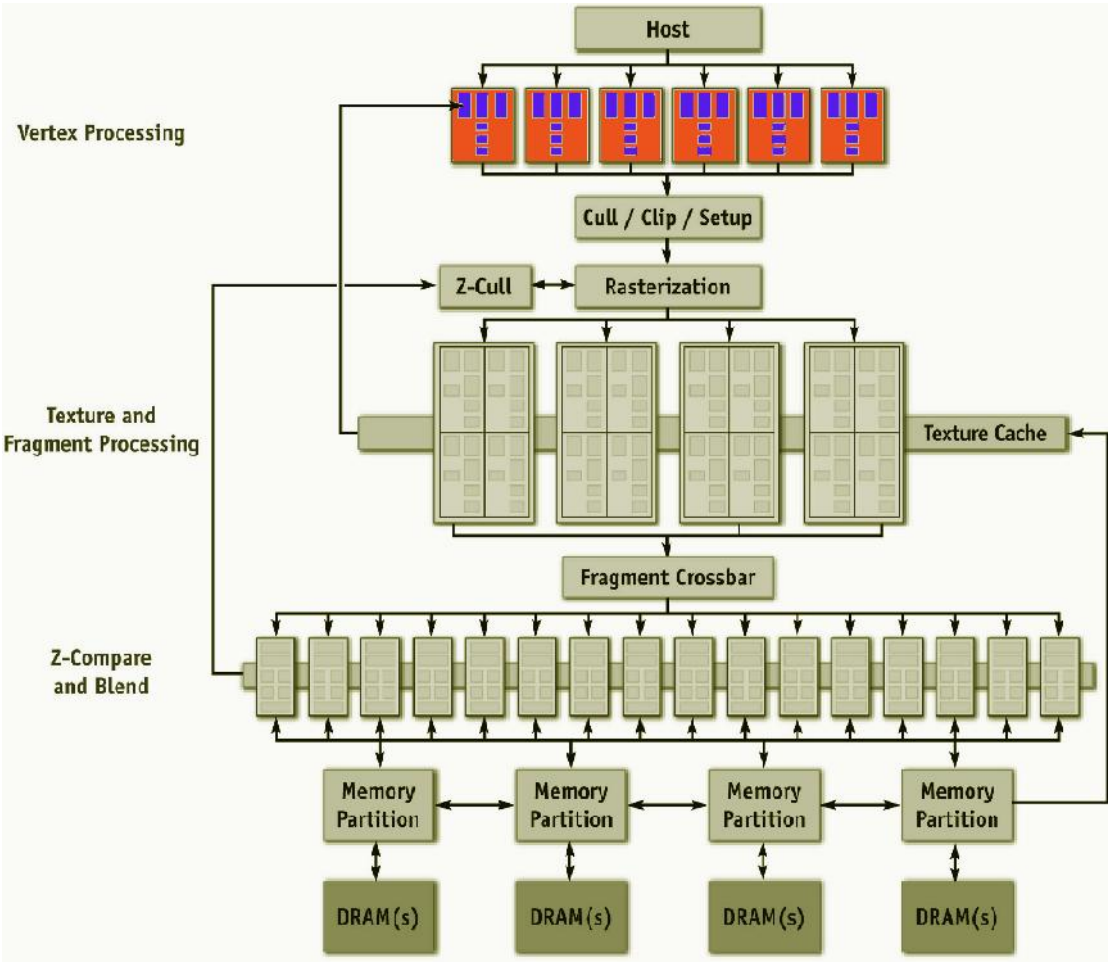


Figure12: vertex processor

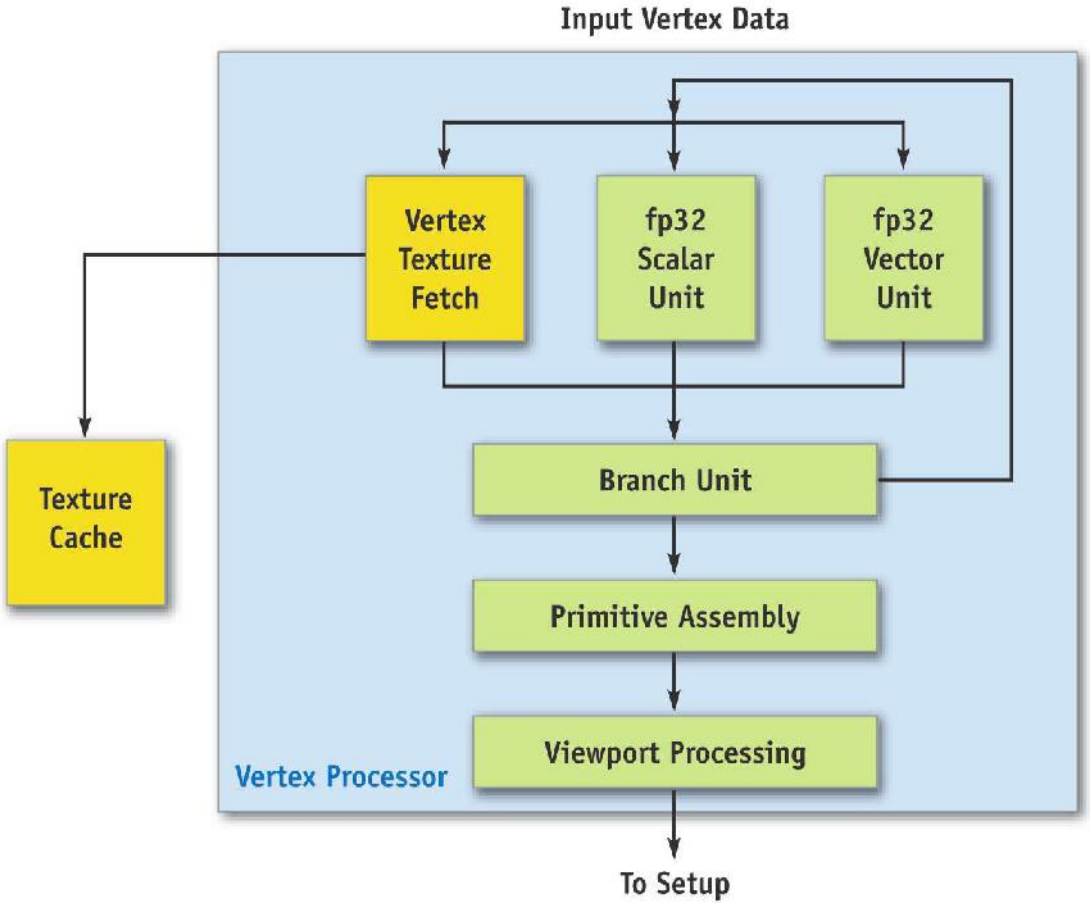


Figure13: vertex processor setup

5.3: Clipping, Z Culling and Rasterization

• Cull/clip – per primitive operation and data preparation for rasterization

• Rasterization: primitive to pixel mapping

• Z culling: quick pixel elimination based on depth

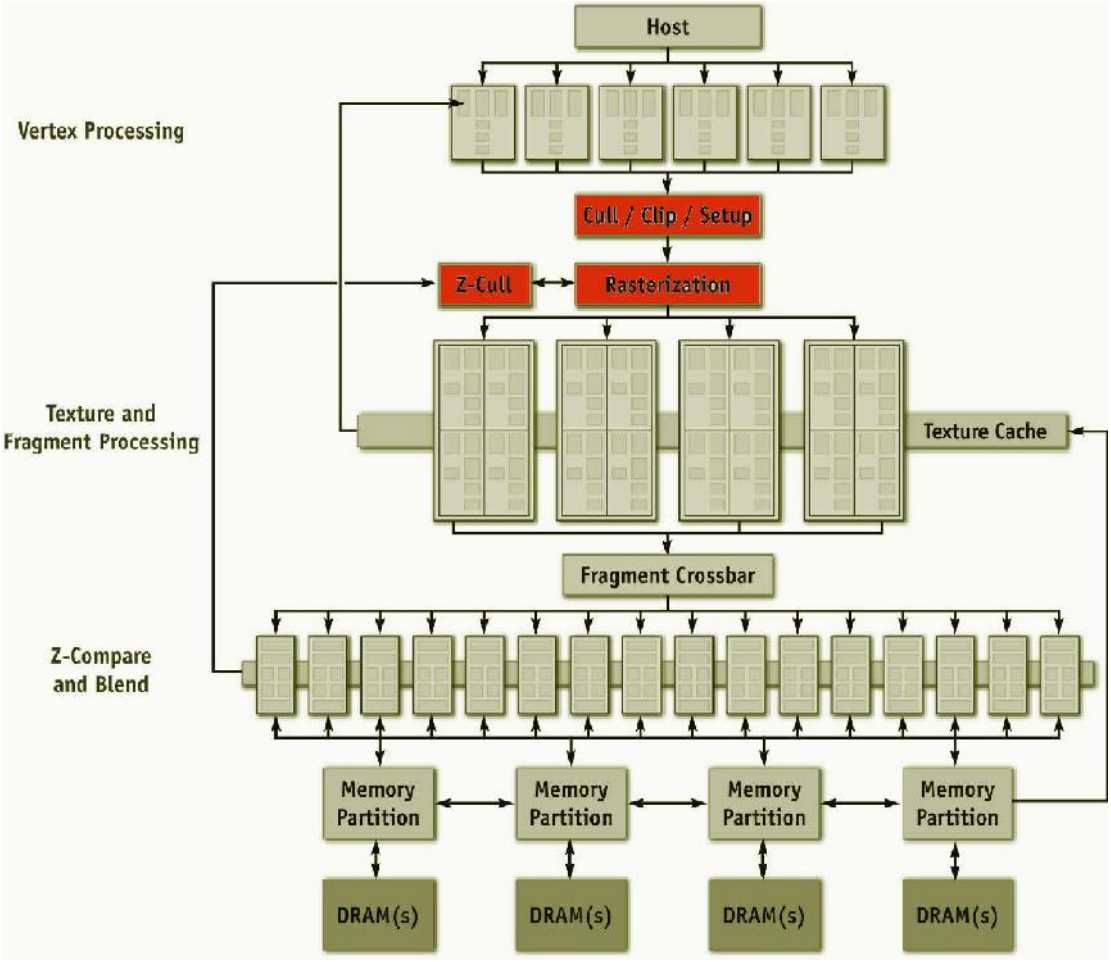


Figure14: Clipping&Culling

5.4 Fragment processor and Texel pipeline

• Texture unit can apply filters.

• Shader units can perform 8 math ops (w/o texture load) or 4 math ops (with texture load) in a clock

• Fog calculation done in the end

• Pixels almost ready for frame buffer

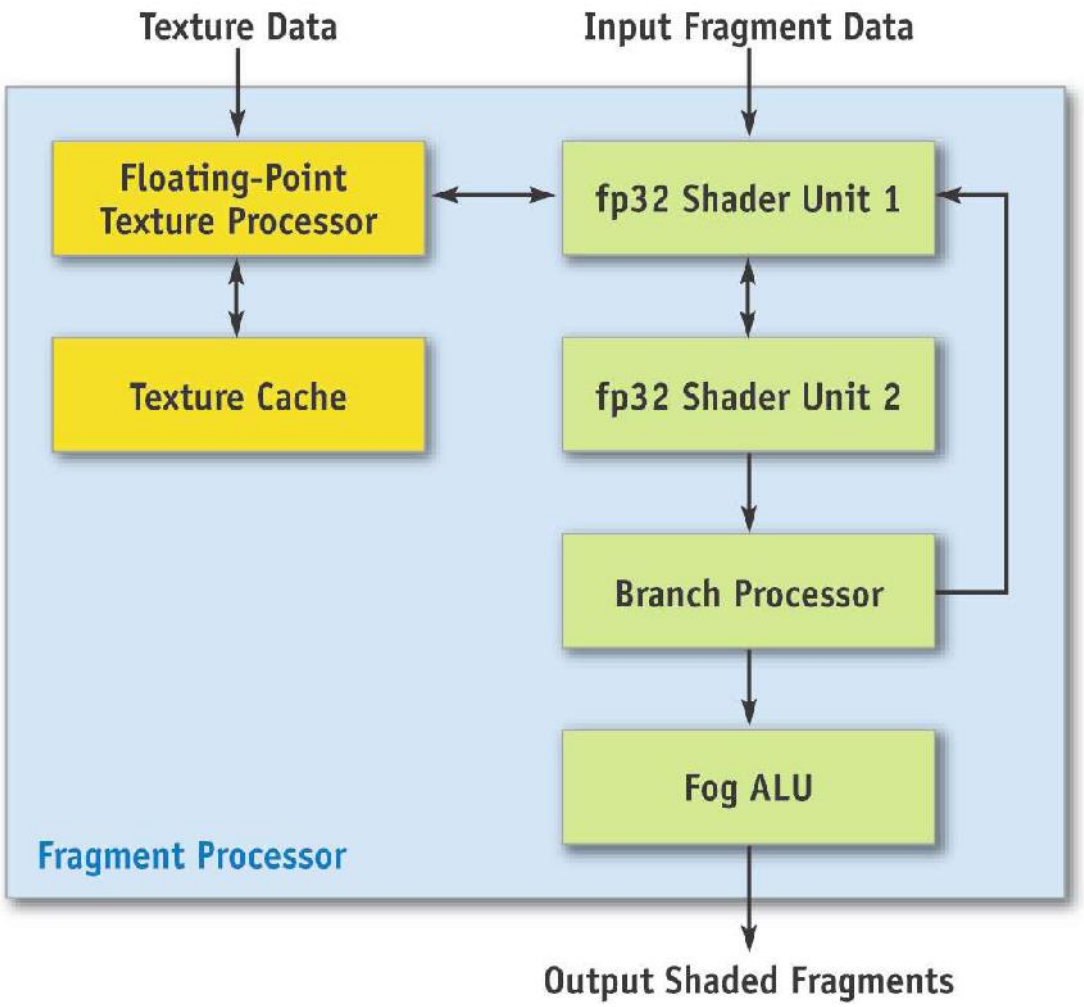


Figure15: Fragment processor &Texel pipeline

5.5: Z compares and blends

• Depth testing

• Stencil tests

• Alpha operations

• Render final color to target buffer

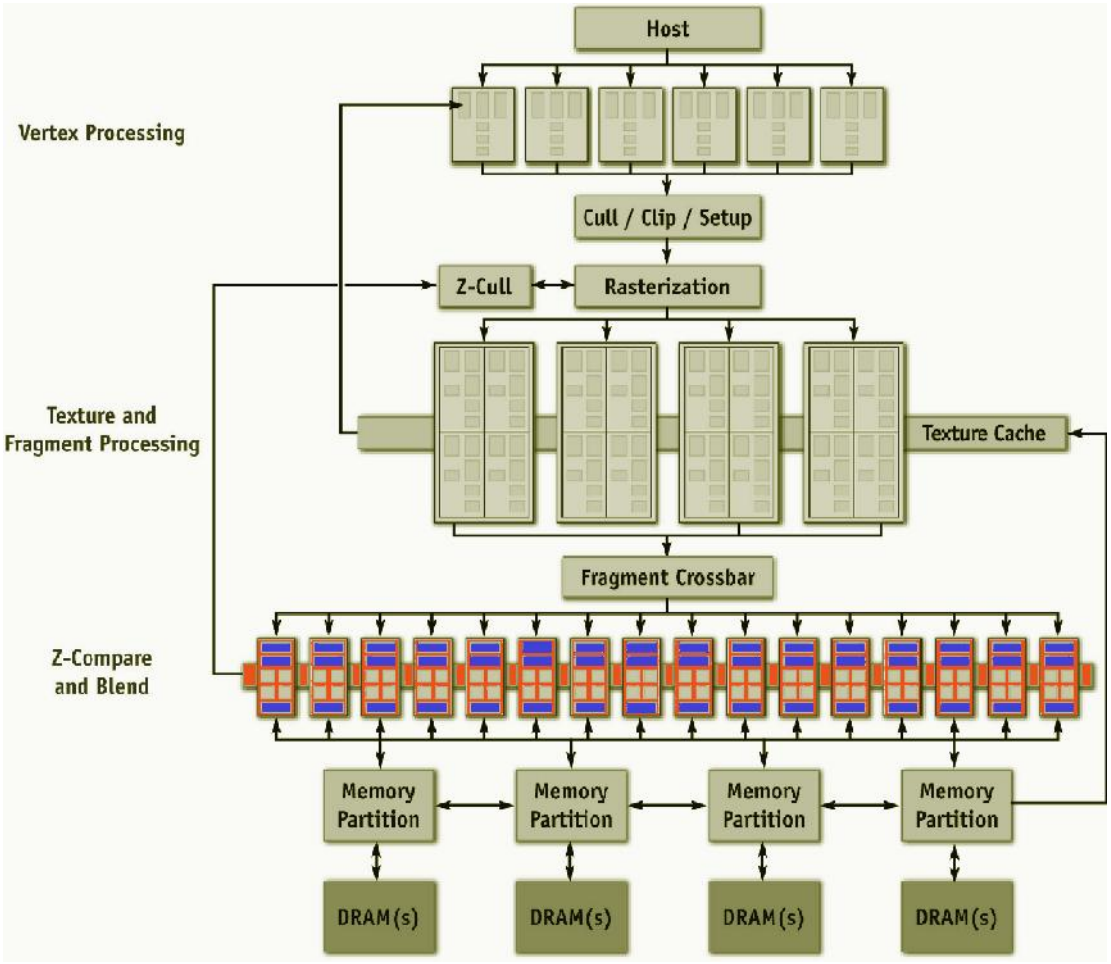


Figure16: Z compares & Blend

5.6: Features – Geometry Instancing

• Vertex stream frequency
– hardware support for looping over a subset of vertices

• Example: rendering the same object multiple times at diff locations (grass, soldiers, people in stadium)

Features – continued

• Early culling and clipping;
– cull no visible primitives at high rate

• Rasterization
– supports Point Sprite, Aliased and anti-aliasing and triangles, etc

• Z-Cull
– Allows high-speed removal of hidden surfaces

• Occlusion Query
– Keeps a record of the number of fragments passing or failing the depth test and reports it to the CPU

Features Continued

• Texturing
– Extended support for non power of two textures to match support for power of two textures - Mipmapping, Wrapping and clamping, Cube map and 3D textures.

• Shadow Buffer Support
– Fetches shadow buffer as a projective texture and performs compares of the shadow buffer data to distance from light.

Features – Shader Support

• Increased instruction count (up to 65535 instructions.)

• Fragment processor; multiple render targets.

• Dynamic flow control branching

• Vertex texturing

• More temporary registers.

5.7: GPU hardware architecture

The hardware architecture of a graphics processing unit differs from that of a normal CPU in several key aspects. These differences originate in the special conditions in the field of real time computer graphics:

• Many objects like pixels and vertices can be handled in isolation and are not interdependent.
• There are *many* independent objects (millions of pixels, thousands of vertices …).
• Many objects require expensive computations.

The GPU architectures evolved to meet these requirements.

To better understand the differences between CPU and GPU architectures we start out with CPU architecture and make several key changes until we have a GPU like architecture.

Chapter: 6

6.1: The GPU Programming

The programmable units of the GPU follow a single program multiple-data (SPMD) programming model. For efficiency, the GPU processes many elements (vertices or fragments) inparallel using the same program. Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each other. All GPU programs must be structured in this way: many parallel elements each processed in parallel by a single program. Each element can operate on 32-bit integer or floating-point data with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory (a Bather operation) and, with the newest GPUs, also write back to arbitrary locations in shared global memory (Bespatter).This programming model is well suited to straight-line programs, as many elements can be processed in lockstep running the exact same code. Code written in this manner is single instruction, multiple data (SIMD).
As shader programs have become more complex, programmers prefer to allow different elements to take different paths through the sameprogram, leading to the more general SPMD model. How is thissupported on the GPU? One of the benefits of the GPU is its large fraction of resources devoted to computation. Allowing a different execution path for each element requires a substantial amount of control hardware. Instead, today's GPUs support arbitrary control flow per thread but impose a penalty for incoherent branching. GPU vendors have largely adopted this approach. Elements are grouped together into blocks, and blocks are processed in parallel. If elements branch in different directions within a block, the hardware computes both sides of the branch for all elements in the block.

The size of the block is known as the Branch granularity and has been decreasing with recent GPU generations today; it is on the order of 16 elements. In writing GPU programs, then, branches are permitted but not free. Programmers who structure their code such that blocks have coherent branches will make the best use of the hardware.

6.2: Programming Graphics Hardware

Modern programmable graphics accelerators such as the ATI X800XT and the NVIDIA GeForce 6800 [ATI 2004b; NVIDIA 2004] feature programmable vertex and fragment processors. Each processor executes a user-specified assembly-level shader program consisting of 4-way SIMD instructions [Lindholm et al. 2001]. These instructions include standard math operations, such as 3- or 4-component dot

products, texture-fetch instructions, and a few special purpose instructions. For every vertex or fragment to be processed, the graphics hardware places a graphics primitive in the read-only input registers. The shader is then executed and the results written to the output registers. During execution, the shader has access to a number of temporary registers as well as constants set by the host application. Purcell et al. [2002] describe how the GPU can be considered a streaming processor that executes kernels, written as fragment or vertex shaders, on streams of data stored ingeometry and textures. Kernels can be written using a variety of high-level, C-like languages such as Cg, HLSL, and GLslang. However, even with these languages, applications must still execute explicit graphics API calls to organize data into streams and invoke kernels.

For example, stream management is performed by the programmer, requiring data to be manually packed into textures and transferred to and from the hardware. Kernel invocation requires the loading and binding of shader programs and the rendering of geometry. As a result, computation is not expressed as a set of kernels acting upon streams, but rather as a sequence of shading operations on graphics primitives. Even for those proficient in graphics programming, expressing algorithms in this way can be an arduous task.

These languages also fail to virtualize constraints of the underlying hardware. For example, stream elements are limited to natively-supported float, float2, float3, andfloat4 types, rather than allowing more complex user defined structures. In addition, programmers must always be aware of hardware limitations such as shader instruction count, number ofshader outputs, and texture sizes. There has been some work in shading languages to alleviate some of these constraints. Chan et al. [2002] present an algorithm to subdivide large shaders automatically into smaller shaders to circumvent shader length and input constraints, but do not explore multiple shader outputs.
McCool et al. [2002; 2004] have developed Sh, a system that allows shaders to be defined and executed using a metaprogramming language built on top of C++. Sh is intended primarily as a shading system, though it has been shown to perform other types of computation. However, it does not provide some of the basic operations common in general purpose computing, such as gathers and reductions.
In general, code written today to perform computation on GPUs is developed in a highly graphics-centric environment, posing difficulties for those attempting to map other applications onto graphics hardware.

6.3: Streaming Hardware

The stream programming model captures computational locality not present in the SIMD or vector models through the use of streams and kernels.A stream is a collection of records requiring similar computation while kernels are functions applied to each element of a stream.

A streaming processor executes a kernel over all elements of an input stream, placing the results into an output stream. Dally et al. [2003] explain how stream programming encourages the creation of applications with high arithmetic intensity, the ratio of arithmetic operations to memory bandwidth. This paper defines a similar property called computational intensity to compare CPU and GPU performance.Stream architectures are a topic of great interest in computer architecture [Bove and Watlington 1995;Gokhale and Gomersall 1997]. For example, the Imagine stream processor [Kapasi et al. 2002] demonstrated the effectiveness of streaming for a wide range of media applications, including graphics and imaging [Owens et al. 2000].
The StreamC/KernelC programming environment provides an abstraction which allows programmers to map applications to the Imagine processor [Mattson 2002]. Labonte et al. [2004] studied the effectiveness of GPUs as stream processors by evaluating the performance of a streaming virtual machine mapped onto graphics hardware. The programming model presented in this paper could easily be compiled to their virtual machine.

6.4:General-Purpose Computing on the GPU

Mapping general-purpose computation onto the GPU uses the graphics hardware in much the same way as any standard graphics application. Because of this similarity, it is both easier and more difficult to explain the process. On one hand, the actual operations are the same and are easy to follow; on the other hand, the terminology is different between graphics and general-purpose use.
Harris provides an excellent description of this mapping process. We begin by describing GPU programming using graphicsterminology, then show how the same steps are used in a general-purpose way to author GPGPU applications, and finally use the same steps to show the more simple and direct way that today's GPU computing applications are written.

Programming a GPU for Graphics: We begin with the same GPU pipeline that we described in Section II, concentrating on the programmable aspects of this pipeline.

1) The programmer specifies geometry that covers a region on the screen. The rasterizer generates a fragment at each pixel location covered by that geometry.
2) Each fragment is shaded by the fragment program.
3) The fragment program computes the value of the fragment by a combination of math operations and global memory reads from a global Btexture memory.
4) The resulting image can then be used as texture on future passes through the graphics pipeline.

Programming a GPU for General-Purpose Programs (Old): Coopting this pipeline to perform general-purpose computation involves the exact same steps but different terminology. A motivating example is a fluid simulation computed over a grid: at each time step, we compute the next state of the fluid for each grid point from the current state at its grid point and at the grid points of its neighbors.

1) The programmer specifies a geometric primitive that covers a computation domain of interest. The rasterizer generates afragment at each pixel location covered by that geometry.
2) Each fragment is shaded by an SPMD generalpurpose fragment program. (Each grid point runs the same program to update the state of its fluid.)
3) The fragment program computes the value of the fragment by a combination of math operations and Bgather accesses from global memory. (Each grid point can access the state of its neighbors from the previous time step in computing its current value.)
4) The resulting buffer in global memory can then be used as an input on future passes. (The current state of the fluid will be used on the next time step.)

Programming a GPU for General-Purpose Programs (New):
One of the historical difficulties in programming GPGPU applications has been that despite their general-purpose tasks' having nothing to do with graphics, the applications still had to be programmed using graphics APIs. In addition, the program had to structure in terms of the graphics pipeline, with the programmable units only accessible as an intermediate step in that pipeline, when the programmer would almost certainly prefer to access the programmable units directly.

The programming environments we describe in detail in Section IV are solving this difficulty by providing a more natural, direct, non graphics interface to the hardware and, specifically, the programmable units.
Today, GPU computing applications are structured in the following way.
1) The programmer directly defines the computation domain of interest as a structured grid of threads.
2) An SPMD general-purpose program computes the value of each thread.
3) The value for each thread is computed by a combination of math operations and both Bgather (read) accesses from and Bscatter (write) accesses to global memory. Unlike in the previous two methods, the same buffer can be used for both reading and writing, allowing more flexible algorithms (for example, in-place algorithms that use less memory).
4) The resulting buffer in global memory can then be used as an input in future computation.

This programming model is a powerful one for several reasons. First, it allows the hardware to fully exploit the application's data parallelism by explicitly specifying that parallelism in the program. Next, it strikes a careful balance between generality (a fully programmable routine at each element) and restrictions to ensure good performance (the SPMD model, the restrictions on branching for efficiency, restrictions on data communication between elements and between kernels/passes, and so on). Finally, its direct access to the programmable units eliminates much of the complexity faced by previous GPGPU programmers in coopting the graphics interface for general-purpose programming.

As a result, programs are more often expressed in a familiarprogramming language (such as NVIDIA's C-like syntax in their CUDA programming environment) and are simpler and easier to build and debug (and are becoming more so as the programming tools mature). The result is a programming model that allows its users to take full advantage of the GPU's powerful hardware but also permits an increasingly high-level programming model that enables productive authoring of complex applications.

6.5: Memory

The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independen memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred.
All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four independent memory partitions give the GPU a wide (256 bits), exible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit."

Performance

_ 425 MHz internal graphics clock

_ 550 MHz memory clock

_ 256-MB memory size

_ 35.2 GByte/second memory bandwidth

_ 600 million vertices/second

_ 6.4 billion texels/second

_ 12.8 billion pixels/second, rendering z/stencil-only (useful for shadow volumes and shadow buffers)

_ 6 four-wide fp32 vector MADs per clock cycle in the vertex shader, plus one scalar multifunction operation (a complex math operation, such as a sine or reciprocal square root)

_ 16 four-wide fp32 vector MADs per clock cycle in the fragment processor, plus 16 four-wide fp32 multiplies per clock cycle

_ 64 pixels per clock cycle early z-cull (reject rate)

_ 120+ Gops peak (equal to six 5-GHz Pentium 4 processors)

_ Up to 120 W energy consumption (the card has two additional power connectors, the power sources are recommended to be no less than 480W)

6.6: Computational Principles

Stream Processing:

Typical CPUs (the von Neumann architecture) suffer from memory bottlenecks when processing. GPUs are very sensitive to such bottlenecks, and therefore need a different architecture; they are essentially special purpose stream processors.

A stream processor is a processor that works with so calledstreams and kernels. A stream is a set of data and a kernel is a small program. In stream processors, every kernel takes one or more streams as input and outputs one or more streams, while it executes its operations on every single element of the input streams.

In stream processors you can achieve several levels of parallelism:

_ Instruction level parallelism: kernels perform hundreds of instructions on every stream element; you achieve parallelism by performing independent instructions in parallel

_ Data level parallelism: kernels perform the same instructions on each stream element; you achieve parallelism by performing one instruction on many stream elements at a time

_ Task level parallelism: Have multiple stream processors divide the work from one kernel

Stream processors do not use caching the same way traditional processors do since the input datasets are usually much larger than most caches and the data is barely reused - with GPUs for example the data is usually rendered and then discarded.

Continuing these ideas, GPUs employ following strategies to increase output:

Pipelining: Pipelining describes the idea of breaking down a job into multiple components that each perform a single task. GPUs are pipelined, which means that instead of performing complete processing of a pixel before moving on to the next, you fill the pipeline like an assembly line where each component performs a task on the data before passing it to the next stage. So while processing a pixel may take multiple clock cycles, you still achieve an output of one pixel per clock since you fill up the whole pipe.

Parallelism: Due to the nature of the data - parallelism can be applied on a per-vertex or per-pixel basis and the type of processing (highly repetitive) GPUs are very suitable for parallelism, you could have an unlimited amount of pipelines next to each other, as long as the CPU is able to keep them busy.

Chapter: 7

7.1: The next step the Geforce 8800

After the Geforce 7 series which was a continuation of the Geforce 6800 architecture, NVIDIA introduced the Geforce 8800 in 2006. Driven by the desire to increase performance, improve image quality and facilitate programming, the Geforce 8800 presented a significant evolution of past designs: unified shader architecture.

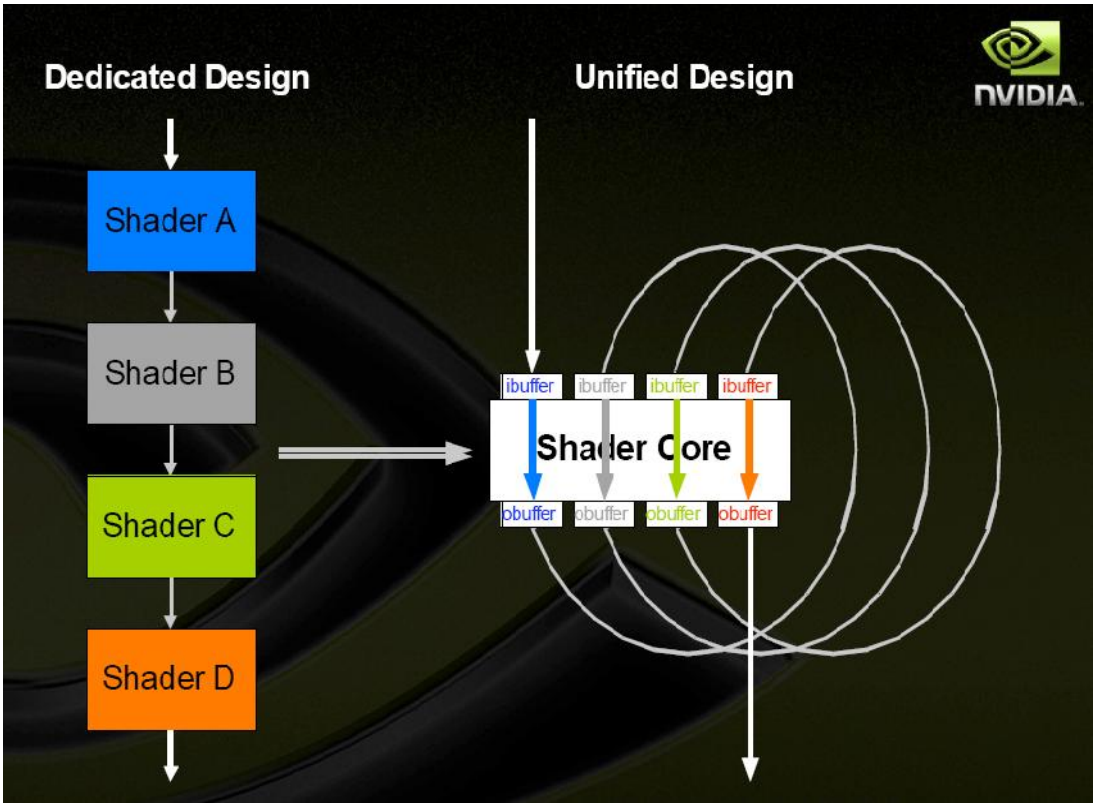


Figure17: From dedicated to unified architecture

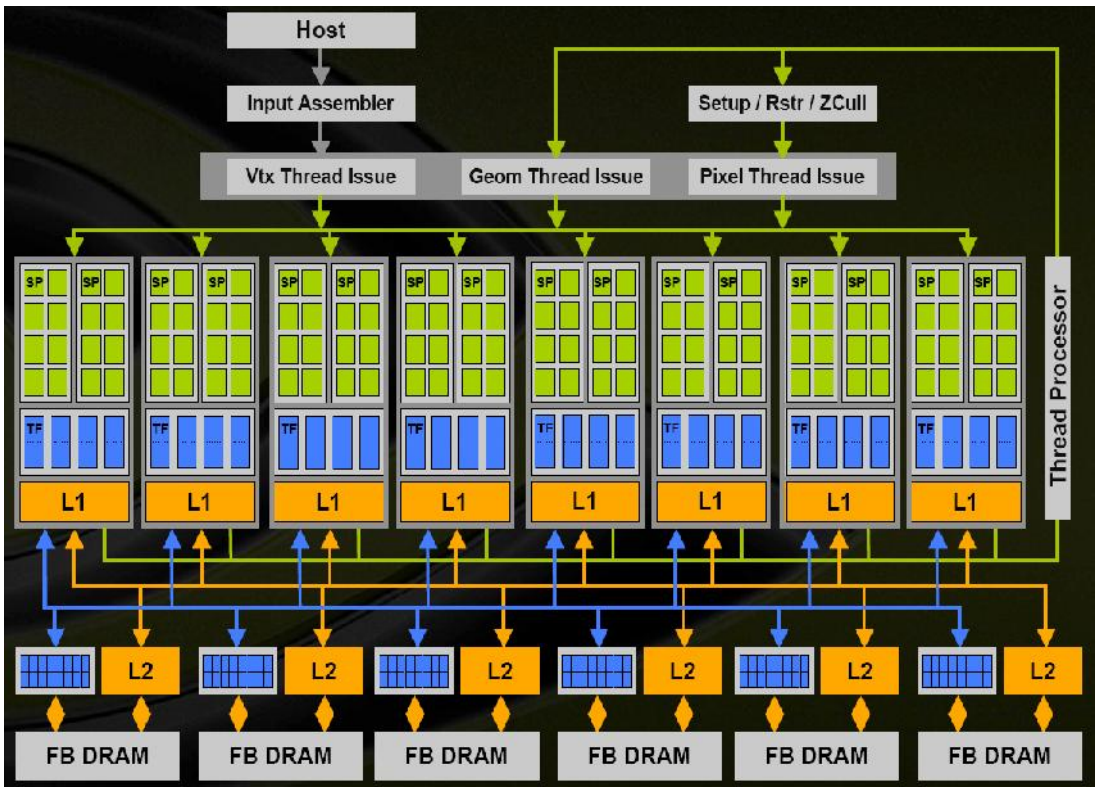


Figure18: A schematic view of the Geforce 8800

The unified shader architecture of the Geforce 8800 essentially boils down to the fact that all the different shader stages become one single stage that can handle all the different shader.

Instead of different dedicated units we now have a single streaming processor array. We have familiar units such as the raster operators (blue, at the bottom) and the triangle setup, rasterization and z-cull unit. Besides these units we now have several managing units that prepare and manage the data as it owe in the loop (vertex, geometry and pixel thread issue, input assembler and thread processor).

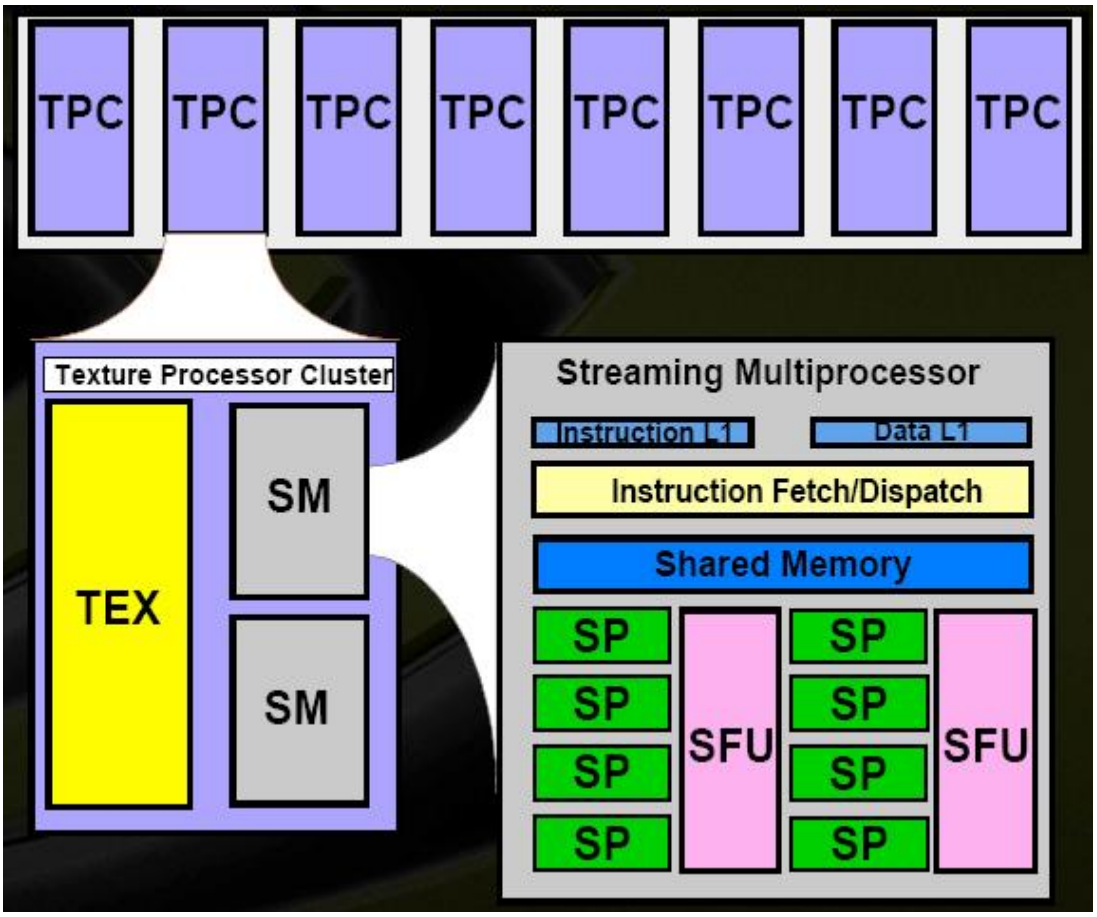


Figure19: The streaming processor array

The streaming processor array consists of 8 texture processor clusters. Each texture processor cluster in turn consists of 2 streaming multiprocessors and 1 texture pipes. A streaming multiprocessor has 8 streaming processors and 2 special function units. The streaming processors work with 32-bit scalar data, based on the idea that shader programs are becoming more and more scalar, making vector architecture more inefficient. They are driven by a high-speed clock that is separate from the core clock and can perform a dual-issued MUL and MAD at each cycle. Each multiprocessor can have 768 hardware scheduled threads; grouped together to 24 SIMD "warps" (A warp is a group of threads).

7.2: Compare CPU and GPU

CPU and GPU architectures share the same basic execution model. Fetch and decode an instruction, execute it and use some kind of execution context or registers to operate upon. But the GPU architecture differs from the CPU architecture in the three key concepts introduced in the previous chapter:

• No focus on single instruction stream performance
• Share instructions between streams (SIMD)
• Interleave streams to hide latency

These concepts originate in the special conditions of the computer graphics domain. However some of the ideas behind these concepts can also be found in modern day CPUs. Usually inspired by other problem domains with similar conditions.
The MMX and later the SEE instruction sets are also SIMD (single instruction multiple data) based for example. SEE allows working on multiple integer or single precision floating point values simultaneously. These instructions were added to allow faster processing of video and audio data. In that case the raw volume of multimedia data forced new optimizations into the CPU architecture.

However most CPU programs do not use these optimizations by default. Usually programming languages focus on well known development paradigms. Vectorization or data parallelism unfortunately isn't such a well known paradigm. In the contrary, it is usually only adopted when necessary because it adds complexity to a program. Therefore very much software does not use SIMD style instructions even if the problemdomain would offer it. Especially in a time where development time is considered very expensive and performance cheap it's hard to justify proper optimization. This situation gave SIMD instructions on the CPU an add-on characteristics instead of being the instructions of choice to efficiently solve certain problems.

Out of different motivations CPU and GPU architectures moved into the same direction in regards to SIMD optimization.13Another similarity between CPUs and GPUs it the current development towards multicore and many core CPUs. Out of certain physical limitations (speed of light, leakage voltage in very small circuits) CPUs can no longer increase the performance of a single instruction stream. CPU frequency and with it the instructions per second cannot be increased indefinitely.
The increased power consumption of high frequencies will damage the circuits and require expensive cooling. The Net burst architecture (Pentium 4) was designed for very high frequencies of up to 10 GHz. However the excessive power consumption limited the frequencies to about 3 to 4 GHz.Frequencies of up to 7 GHz were achieved under special cooling conditions for a very short time (30 seconds up to 1 minute before the CPU burned out).

In order to integrate more and more cores onto a single CPU speed and complexity of a single core is usually reduced. Therefore single core CPUs usually have a higher performance for one instruction stream than the more modern quad core CPUs. This is similar to the idea used in the GPU architecture: many simple processing cores are more effective than one large core. While CPU cores are still far more complex than GPU cores they might develop into more simple layouts to allow better scaling. GPU cores on the other hand might evolve into more complex layouts to provide more developer friendliness. However the idea of horizontal scaling is present on both architectures.

7.3: THE FUTURE OF GPU COMPUTING

With the rising importance of GPU computing, GPU hardware and software are changing at a remarkable pace. In the upcoming years, we expect to see several changes to allow more flexibility and performance from future GPU computing systems:

• At Supercomputing 2006, both AMD and NVIDIA announced future support for double-precision floating-point hardware by the end of 2007. The addition of double-precision support removes one of the major obstacles for the adoption of the GPU in many scientific computing applications.

• Another upcoming trend is a higher bandwidth path between CPU and GPU. The PCI Express bus between CPU and GPU is a bottleneck in many applications, so future support for PCI Express 2, Hyper Transport, or other high-bandwidth connections is a welcome trend.
Sony's PlayStation 3 and Microsoft's XBox 360 both feature CPU–GPU connections with substantially greater bandwidth than PCI Express, and this additional bandwidth has been welcomed by developers.

• Such as AMD's Fusion, that places both the CPU and GPU on the same die. Fusion is initially targeted at portable, not high-performance, systems, but the lessons learned from developing this hardware and its heterogeneous APIs will surely be applicable to future single-chip systems built for performance. One open question is the fate of the GPU's dedicated high-bandwidth memory system in a computer with a more tightly coupled CPU and GPU.

• Pharr notes that while individual stages of the graphics pipeline are programmable, the structure of the pipeline as a whole is not [32], and proposes future architectures that support not just programmable shading but also a programmable pipeline. Such flexibility would lead to not only a greater variety of viable rendering approaches but also more flexible general-purpose processing.

• Systems such as NVIDIA's 4-GPU Quadroplex are well suited for placing multiple coarse-grained GPUs in a graphics system. On the GPU computing side, however, fine-grained cooperation between GPUs is still an unsolved problem. Future API support such as Microsoft's Windows Display Driver Model 2.1 will help multiple GPUs to collaborate on complex tasks, just as clusters of CPUs do today.

It is apparent that the market for GPUs is very much alive and moving fast. GPUs actually scale well beyond Moore's law, doubling their speed almost twice a year. With such a rapid development we can certainly expect to see quite some interesting things to come in this field of processing.

Chapter: 8

8.1: Conclusion

The presented the most efficient currently known approach in encryption and decryption of messages with AES on programmable graphics processing units. While the study suggested that the traditional graphics hardware architectures could now be compared with optimized sequential solutions on the CPU. There is a multitude of reasons to consider using a neural based solution to real-world problems, and several of them have been outlined in this thesis. Coupled with commercial o®-the-shelf graphics hardware found in many personal computers, significant improvements in simulation performance can be realized. Although more power is not a panacea, it is a good step towards allowing us to solve larger, harder, and ultimately more interesting problems and questions. Unlike previous related work, for thefirst time a GPU implementation of AES performs theencryption and decryption of the input data without the CPU to keep busy in the meantime. So this effort has to be consideredas the proof that the modern unified GPUarchitecture can perform as an efficient cryptographic acceleration board. Future work will include efficient implementations of other common symmetric algorithms. GPU implementations of hashing and public key algorithms may also be implemented, in order to create a complete cryptographic framework accelerated by the GPU.

8.2: References

1. Wikipedia. Graphics processing unit. http://en.wikipedia.org/
wiki/Graphics processing unit, 2012.

2. Wikipedia. John hopfield. http://en.wikipedia.org/wiki/John Hopfield, 2012.

3. Wikipedia entry on GPUs http://en.wikipedia.org/wiki/GPU

4. Kees Huizing, Han-Wei Shen: \The Graphics Rendering Pipeline" http://www.win.tue.nl/~keesh/ow/2IV40/pipeline2.pdf

5. Cyril Zeller: \Introduction to the Hardware Graphics Pipeline", Presentation at ACM SIGGRAPH 2012
http://download.nvidia.com/developer/presentations/2005/I3D/I3D_05
IntroductionToGPU.pdf

6. ExtremeTech 3D Pipeline Tutorial
http://www.extremetech.com/article2/0,1697,9722,00.asp

7. DirectX Developer Center: \The Direct3D Transformation Pipeline"
http://msdn.microsoft.com/en-us/library/bb206260(VS.85).aspx

8. Mark Colbert: \GPU Architecture & CG"
http://graphics.cs.ucf.edu/gpuseminar/seminar1.ppt

9. GPU Gems 2, Chapter 30: \The GeForce 6 Series GPU Architecture"
http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

10. IEEE Micro, Volume 25, Issue 2 (March 2005): \The GeForce 6800"
http://portal.acm.org/citation.cfm?id=1069760

11. www.3dcenter.de: \NV40-Technik in Detail"
http://www.3dcenter.de/artikel/nv40_pipeline/