GPUs - Graphics Processing Units

Minh Tri Do Dinh

Minh.Do-Dinh@student.uibk.ac.at

Vertiefungsseminar *Architektur von Prozessoren*, SS 2008
Institute of Computer Science, University of Innsbruck

July 7, 2008

This paper is meant to provide a closer look at modern Graphics Processing Units. It explores their architecture and underlying design principles, using chips from Nvidia's "Geforce" series as examples.

1 Introduction

Before we dive into the architectural details of some example GPUs, we'll have a look at some basic concepts of graphics processing and 3D graphics, which will make it easier for us to understand the functionality of GPUs

1.1 What is a GPU?

A GPU (**G**raphics **P**rocessing **U**nit) 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, since that is what modern GPUs are mainly designed for.

1.2 The anatomy of a 3D scene

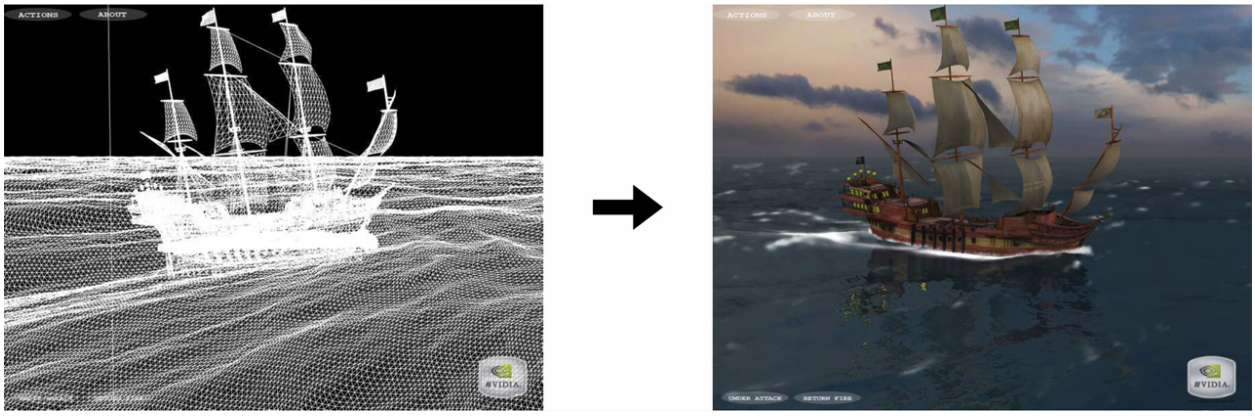


Figure 1: A 3D scene

3D scene: A collection of 3D objects and lights.

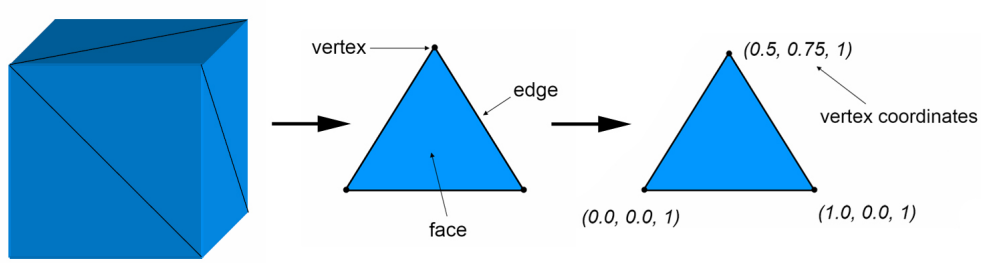


Figure 2: Object, triangle and vertices

3D objects: Arbitrary objects, whose geometry consists of triangular polygons. Polygons are composed of vertices.

Vertex: A Point with spatial coordinates and other information such as color and texture coordinates.

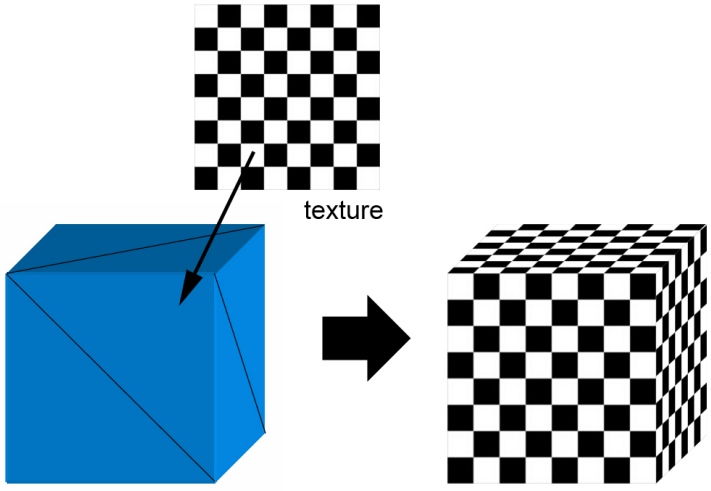


Figure 3: A cube with a checkerboard texture

Texture: An image that is mapped onto the surface of a 3D object, which creates the illusion of an object consisting of a certain material. The vertices of an object store the so-called texture coordinates (2-dimensional vectors) that specify how a texture is mapped onto any given surface.

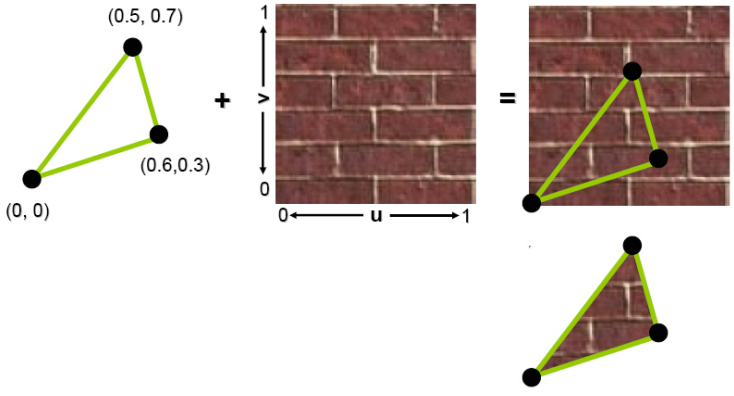


Figure 4: Texture coordinates of a triangle with a brick texture

In order to translate such a 3D scene to a 2D image, the data has to go through several stages of a "Graphics Pipeline"

1.3 The Graphics Pipeline

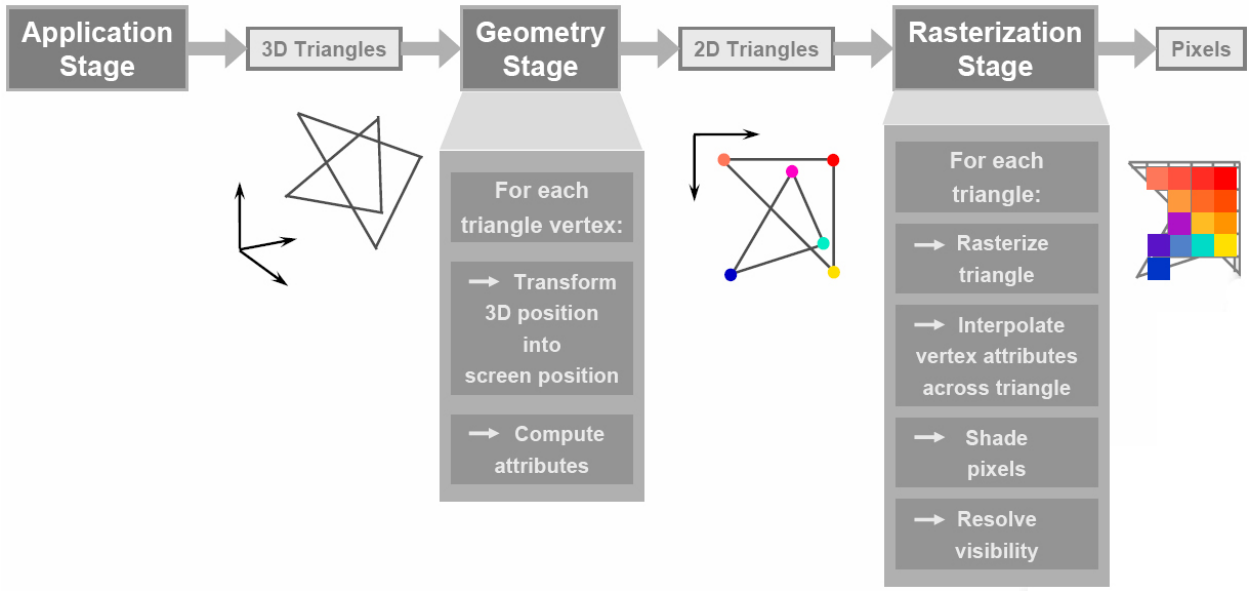


Figure 5: The 3D Graphics Pipeline

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

1.3.1 Geometry Stage

This stage is also referred to as the "Transform and Lighting" stage. In order to translate the scene from 3D to 2D, all the objects of a scene need to be transformed to various spaces - each with its own coordinate system - before the 3D image can be projected onto a 2D plane. These transformations are applied on a vertex-to-vertex basis.

Mathematical Principles

A point in 3D space usually has 3 coordinates, specifying its position. If we keep using 3-dimensional vectors for the transformation calculations, we run into the problem that different transformations require different operations (e.g.: translating a vertex requires addition with a vector while rotating a vertex requires multiplication with a 3x3 matrix). We circumvent this problem simply by extending the 3-dimensional vector by another coordinate (the w-coordinate), thus getting what is called homogeneous coordinates. This way, every transformation can be applied by multiplying the vector with a specific 4x4 matrix, making calculations much easier.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 6: Transformation matrices for translation, rotation and scaling

Lighting, the other major part of this pipeline stage is calculated using the normal vectors of the surfaces of an object. In combination with the position of the camera and the position of the light source, one can compute the lighting properties of a given vertex.

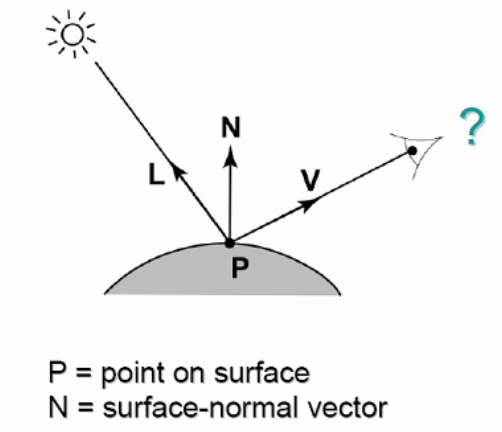
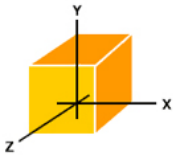


Figure 7: Calculating lighting

For transformation, we start out in the model space where each object (model) has its own coordinate system, which facilitates geometric transformations such as translation, rotation and scaling.



After that, we move on to the world space, where all objects within the scene have a unified coordinate system.

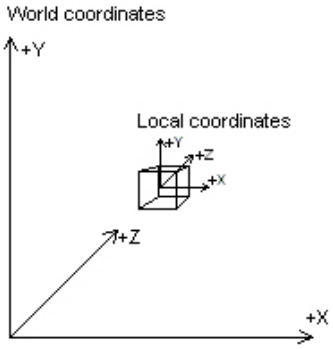


Figure 8: World space coordinates

The next step is the transformation into view space, which locates a camera in the world space and then transforms the scene, such that the camera is at the origin of the view space, looking straight into the positive z-direction. Now we can define a view volume, the so-called view frustrum, which will be used to decide what actually is visible and needs to be rendered.

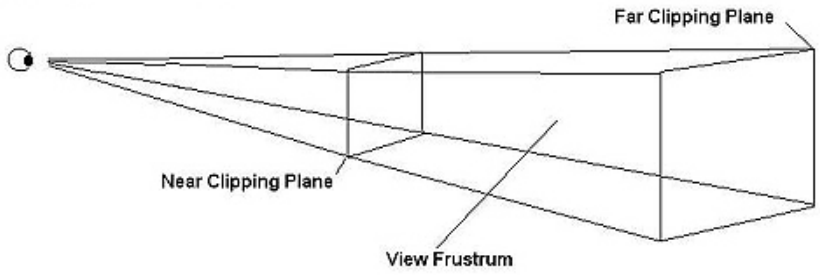


Figure 9: The camera/eye, the view frustrum and its clipping planes

After that, the vertices are transformed into clip space and assembled into primitives (triangles or lines), which sets up the so-called clipping process. While objects that are outside of the frustrum don't need to be rendered and can be discarded, objects that are partially inside the frustrum need to be clipped (hence the name), and new vertices with proper texture and color coordinates need to be created.

A perspective divide is then performed, which transforms the frustrum into a cube with normalized coordinates (x and y between -1 and 1, z between 0 and 1) while the objects inside the frustrum are scaled accordingly. Having this normalized cube facilitates clipping operations and sets up the projection into 2D space (the cube simply needs to be "flattened").

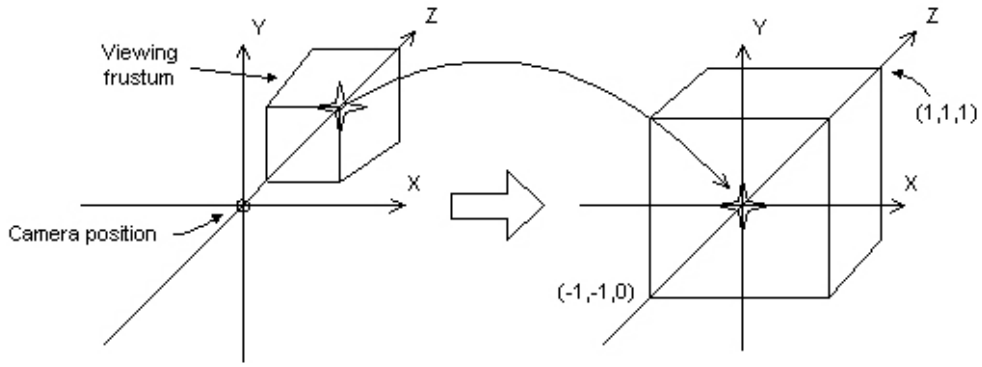


Figure 10: Transforming into clip space

Finally, we can move into screen space where x and y coordinates are transformed for proper 2D display (in a given window). (Note that the z-coordinate of a vertex is retained for later depth operations)

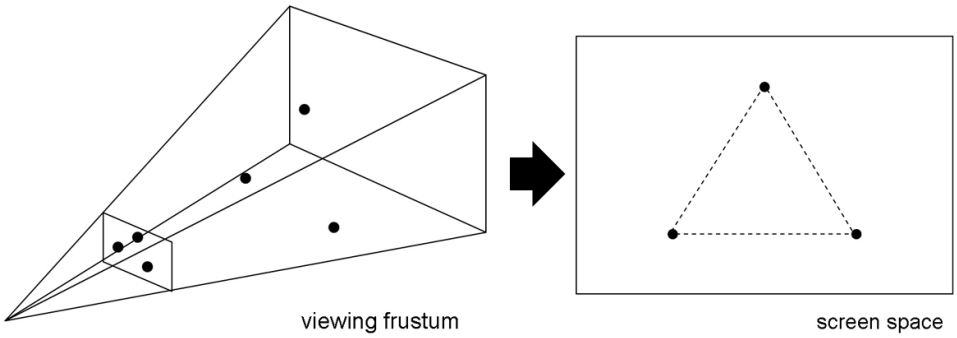


Figure 11: From view space to screen space

Note, that the texture coordinates need to be transformed as well and additionally besides clipping, surfaces that aren't visible (e.g. the backside of a cube) are removed as well (so-called back face culling).

The result is a 2D image of the 3D scene, and we can move on to the next stage.

1.3.2 Rasterization Stage

Next in the pipeline is the Rasterization stage. The GPU needs to traverse the 2D image and convert the data into a number of "pixel-candidates", so-called fragments, which may later become pixels of the final image. A fragment is a data structure that contains attributes such as position, color, depth, texture coordinates, etc. and is generated by checking which part of any given primitive intersects with which pixel of the screen. If a fragment intersects with a primitive, but not any of its vertices, the attributes of that fragment have to be additionally calculated by interpolating the attributes between the vertices.

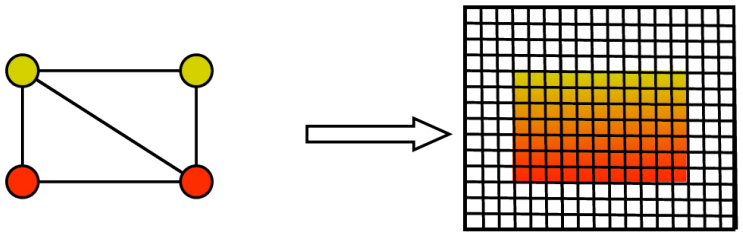


Figure 12: Rasterizing a triangle and interpolating its color values

After that, further steps can be made to obtain the final pixels. Colors are calculated by combining textures with other attributes such as color and lighting or by combining a fragment with either another translucent fragment (so-called alpha blending) or optional fog (another graphical effect).

Visibility checks are performed such as:

- Scissor test (checking visibility against a rectangular mask)
- Stencil test (similar to scissor test, only against arbitrary pixel masks in a buffer)
- Depth test (comparing the z-coordinate of fragments, discarding those which are further away)
- Alpha test (checking visibility against translucent fragments)

Additional procedures like anti-aliasing can be applied before we achieve the final result: a number of pixels that can be written into memory for later display.

This concludes our short tour through the graphics pipeline, which hopefully gives us a better idea of what kind of functionality will be required of a GPU.

2 Evolution of the GPU

Some historical key points in the development of the GPU:

- Efforts for real time graphics have been made as early as 1944 (MIT's project "Whirlwind")
- In the 1980s, hardware similar to modern GPUs began to show up in the research community ("Pixel-Planes", a 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 (e.g. Texas Instruments' TMS340)
- 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 unified shader architecture with the ATI Xenos for the XBox 360
- 2006 Nvidia launches the first unified shader GPU for the PC with the Geforce 8800

3 From Theory to Practice - the Geforce 6800

3.1 Overview

Modern GPUs closely follow the layout of the graphics pipeline described in the first section. Using Nvidia's Geforce6800 as an example we will have a closer look at the architecture of modern day GPUs.

Since being founded in 1993, the company NVIDIA has become one of the biggest manufacturers of GPUs (besides ATI), having released important chips such as the Geforce 256, and the Geforce 3.

Launched in 2004, the Geforce 6800 belongs to the Geforce 6 series, Nvidia's sixth generation of graphics chipsets and the fourth generation that featured programmability (more on that later).

The following image shows a schematic view of the Geforce 6800 and its functional units.

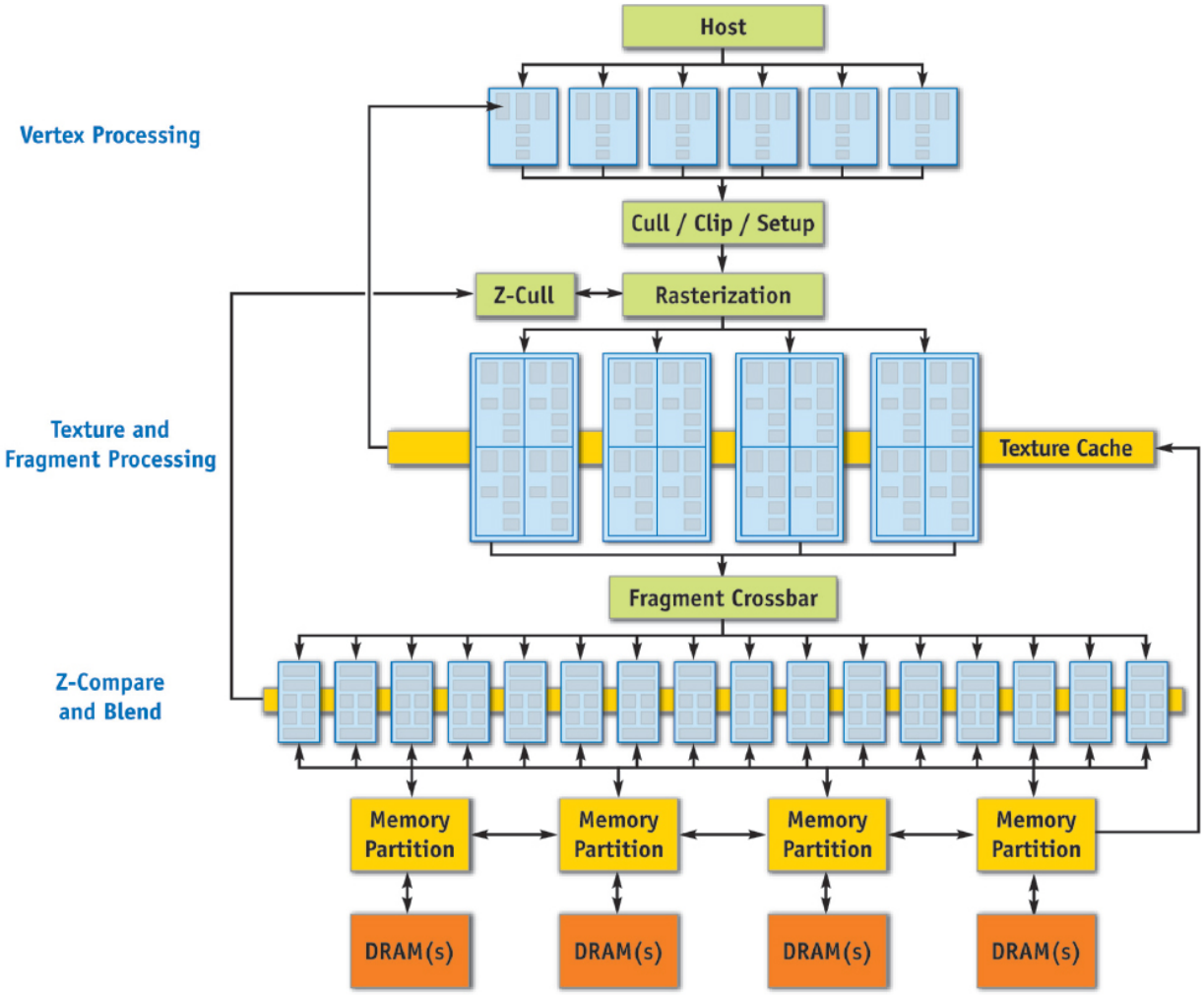


Figure 13: Schematic view of the Geforce 6800

You can already see how each of the functional units correspond to the stages of the graphics pipeline.

We start with six parallel vertex processors that receive data from the host (the CPU) and perform operations such as transformation and lighting.

Next, the output goes into the triangle setup stage which takes care of primitive assembly, culling and clipping, and then into the rasterizer which produces the fragments. The Geforce 6800 has an additional Z-cull unit which allows to perform an early fragment visibility check based on depth, further improving the efficiency.

We then move on to the sixteen fragment processors which operate in 4 parallel units and computes the output colors of each fragment.

The fragment crossbar is a linking element that is basically responsible for directing output pixels to any available pixel engine (also called ROP, short for **R**aster **O**perator), thus avoiding pipeline stalls.

The 16 pixel engines are the final stage of processing, and perform operations such as alpha blending, depth tests, etc., before delivering the final pixel to the frame buffer.

3.2 In Detail

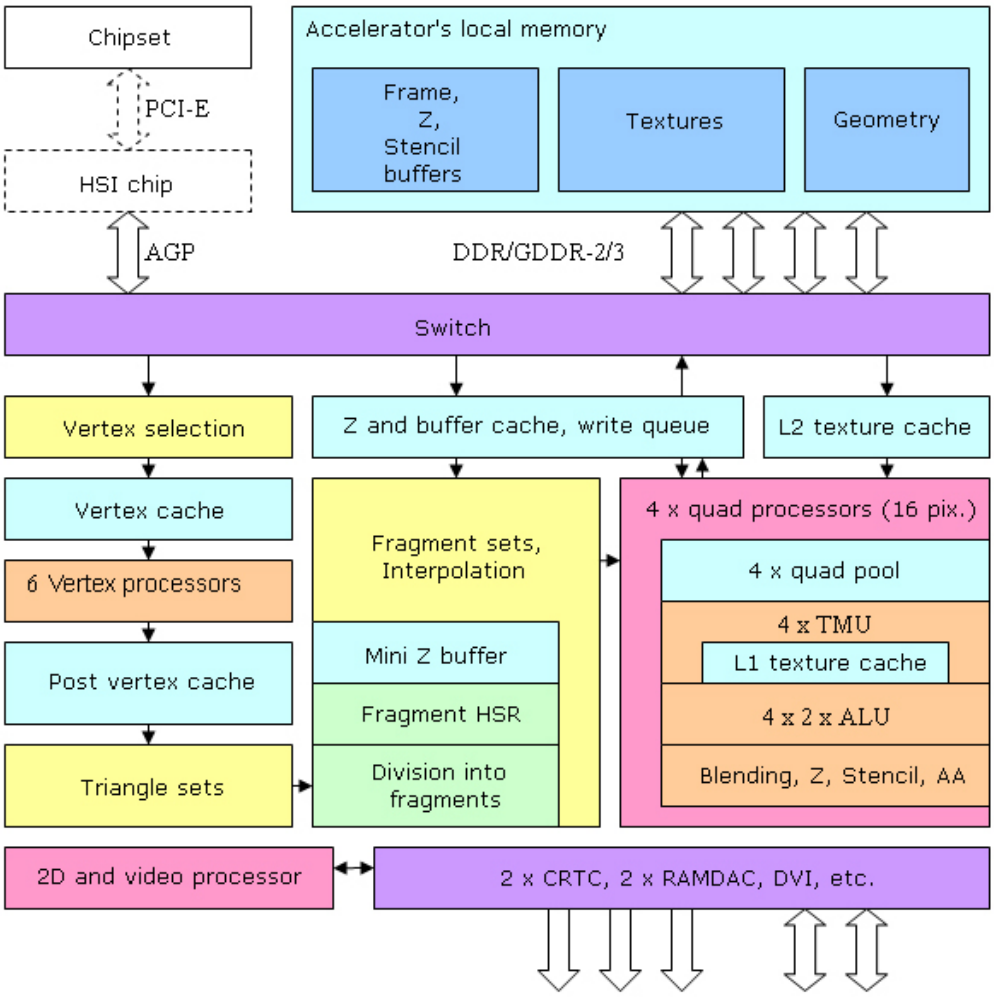


Figure 14: A more detailed view of the Geforce 6800

While most parts of the GPU are fixed function units, the vertex and fragment processors of the Geforce 6800 offer programmability which was first introduced to the geforce chipset line with the geforce 3 (2001). We'll have a more detailed look at the units in the following sections.

3.2.1 Vertex Processor

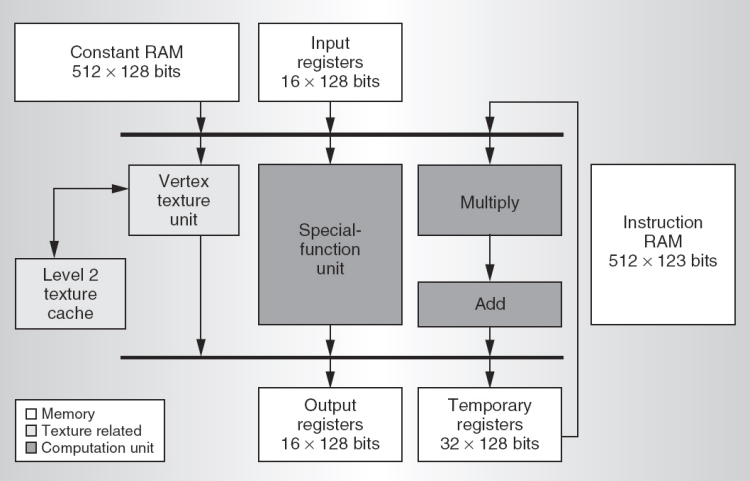


Figure 15: A vertex processor

The vertex processors are the programmable units responsible for all the vertex transformations and attribute calculations. They operate with 4-dimensional data vectors corresponding with the aforementioned homogeneous coordinates of a vertex, using 32 bits per coordinate (hence the 128 bits of a register). Instructions are 123 bits long and are stored in the Instruction RAM.

The data path of a vertex processor consists of:

- A multiply-add unit for 4-dimensional vectors
- A scalar special function unit
- A texture unit

Instruction set:
Some notable instructions for the vertex processor include:

dp4 dst, src0, src1	Computes the four-component dot product of the source registers
exp dst, src	Provides exponential 2^x
dst dest, src0, src1	Calculates a distance vector
nrm dst, src	Normalize a 3D vector
rsq dst, src	Computes the reciprocal square root (positive only) of the source scalar

Registers in the vertex processor instructions can be modified (with few exceptions):

- Negate the register value
- Take the absolute value of the register
- Swizzling (copy any source register component to any temporary register component)
- Mask destination register components

Other technical details:

- Vertex processors are MIMD units (Multiple Instruction Multiple Data)
- They use VLIW (Very Long Instruction Words)
- They operate with 32-bit floating point precision
- Each vertex processor runs up to 3 threads to hide latency
- Each vertex processor can perform a four-wide MAD (Multiply-Add) and a special function in one cycle

3.2.2 Fragment Processor

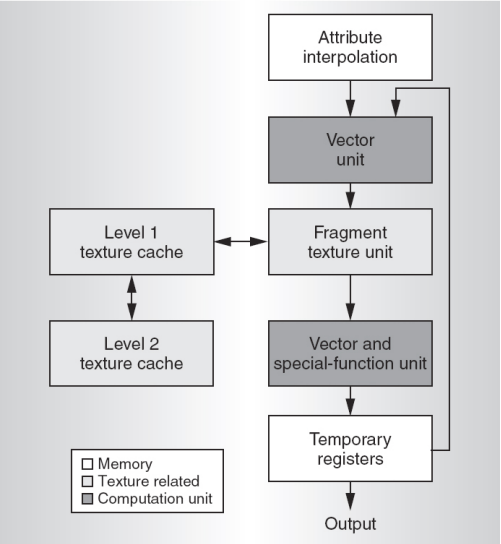


Figure 16: A fragment processor

The Geforce 6800 has 16 fragment processors. They are grouped to 4 bigger units which operate simultaneously on 4 fragments each (a so-called quad). They can take position, color, depth, fog as well as other arbitrary 4-dimensional attributes as input.

The data path consists of:

- An Interpolation block for attributes
- 2 vector math (shader) units, each with slightly different functionality
- A fragment texture unit

Superscalarity:
A fragment processor works with 4-vectors (vector-oriented instruction set), where sometimes components of the vector need be treated seperately (e.g. color, alpha). Thus, the fragment processor supports co-issueing of the data, which means splitting the vector into 2 parts and executing different operations on them in the same clock. It supports 3-1 and 2-2 splitting (2-2 co-issue wasn't possible earlier).
Additionally, it also features dual issue, which means executing different operations on the 2 vector math units in the same clock.

Texture Unit:
The texture unit is a floating-point texture processor which fetches and filters the texture data. It is connected to a level 1 texture cache (which stores parts of the textures that are used).

Shader units 1 and 2:
Each shader unit is limited in its abilities, offering complete functionality when used together.

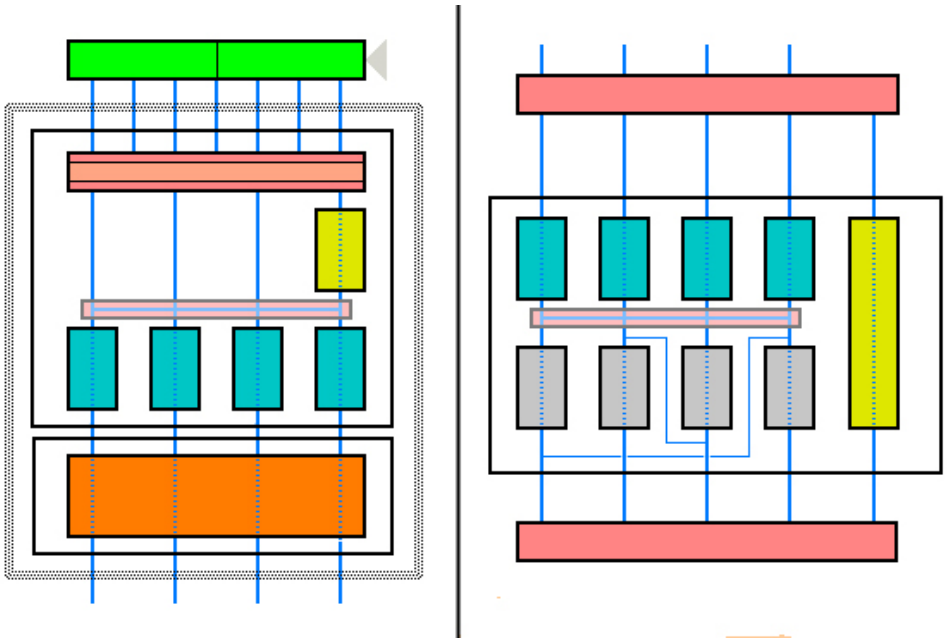


Figure 17: Block diagram of Shader Unit 1 and 2

Shader Unit 1:

Green: A crossbar which distributes the input coming eiter from the rasterizer or from the loopback
Red: Interpolators
Yellow: A special function unit (for functions such as Reciprocal, Reciprocal Square Root, etc.)
Cyan: MUL channels
Orange: A unit for texture operations (not the fragment texture unit)

The shader unit can perform 2 operations per clock:
A MUL on a 3-dimensional vector and a special function, a special function and a texture operation, or 2 MULs.
The output of the special function unit can go into the MUL channels.

The texture gets input from the MUL unit and does LOD (Level Of Detail) calculations, before passing the data to the actual fragment texture unit. The fragment texture unit then performs the actual sampling and writes the data into registers for the second shader unit.

The shader unit can simply pass data as well.

Shader Unit 2:

Red: A crossbar
Cyan: 4 MUL channels
Gray: 4 ADD channels
Yellow: 1 special function unit

The crossbar splits the input onto 5 channels (4 components, 1 channel stays free).
The ADD units are additionally connected, allowing advanced operations such as a dotproduct in one clock. Again, the shader unit can handle 2 independent operations per cycle or it can simply pass data. If no special function is used, the MAD unit can perform up to 2 operations from this list: MUL, ADD, MAD,

DP, or any other instruction based on these operations.

Instruction set:
Some notable instructions for the vertex processor include:

cmp dst, src0, src1, src2	Choose src1 if src0 >= 0. Otherwise, choose src2. The comparison is done per channel
dsx dst, src dsy dst, src	Compute the rate of change in the render target's x-direction Compute the rate of change in the render target's y-direction
sincos dst.{x|y|xy}, src0.{x|y|z|w}	Computes sine and cosine, in radians
texld dst, src0, src1	Sample a texture at a particular sampler, using provided texture coordinates

Registers in the fragment processor instructions can be modified (with few exceptions):

- Negate the register value
- Take the absolute value of the register
- Mask destination register components

Other technical details:

- The fragment processors can perform operations within 16 or 32 floating point precision (e.g. the fog unit uses only 16 bit precision for its calculations since that is sufficient)
- The quads operate as SIMD units
- They use VLIW
- They run up to 100s of threads to hide texture fetch latency (˜256 per quad)
- A fragment processor can perform up to 8 operations per cycle / 4 math operations if there's a texture fetch in shader 1

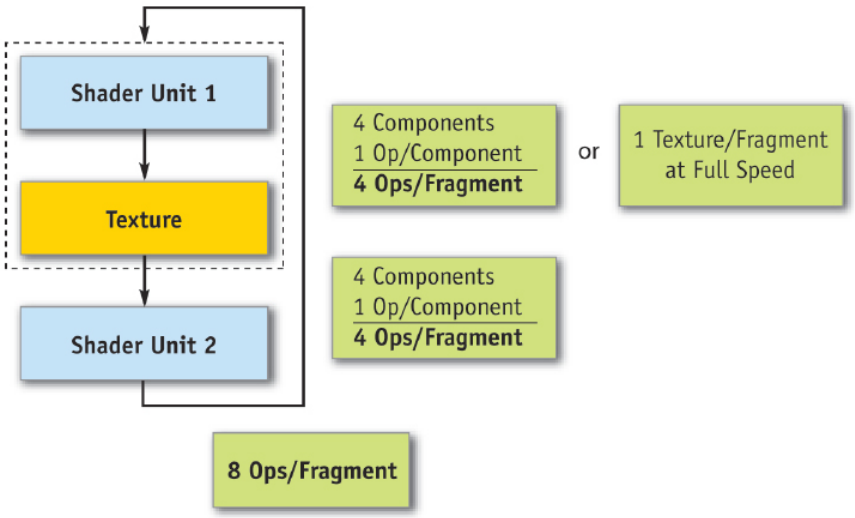


Figure 18: Possible operations per cycle

- The fragment processors have a 2 level texture cache
- The fog unit can perform fog blending on the final pass without performance penalty. It is implemented with fixed point precision since that's sufficient for fog and saves performance.
  The equation: out = FogColor * fogFraction + SrcColor * (1 - fogFraction)

- There's support for multiple render targets, the pixel processor can output to up to four seperate buffers (4x4 values, color + depth)

3.2.3 Pixel Engine

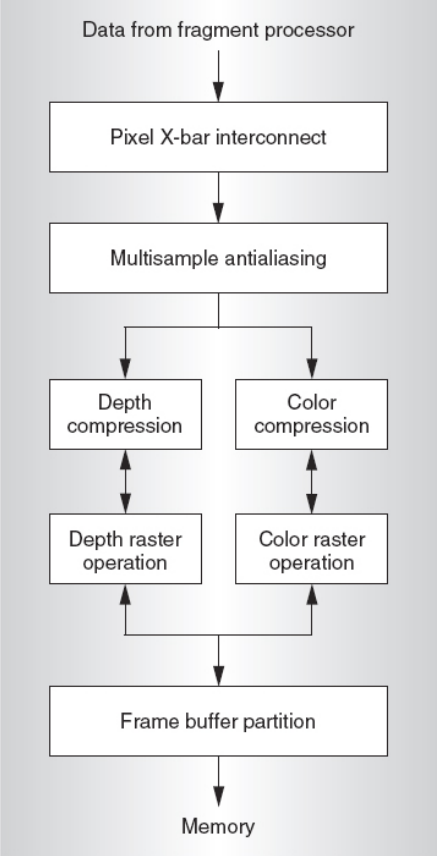


Figure 19: A pixel engine

Last in the pipeline are the 16 pixel engines (raster operators). Each pixel engine connects to a specific memory partition of the GPU. After the lossless color and depth compression, the depth and color units perform depth, color and stencil operations before writing the final pixel. When activated the pixel engines also perform multisample antialiasing.

3.2.4 Memory

From "GPU Gems 2, Chapter 30: The GeForce 6 Series GPU Architecture":

> "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, independent 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), flexible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit."

3.3 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+ Gflops 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 480 W)

4 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-called streams 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.

We know GPUs have to work with large amounts of data, the computations are simpler but they need to be fast and parallel, so it becomes clear that the stream processor architecture is very well suited for GPUs.

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.

Other GPU characteristics:

- GPUs can afford large amounts of floating point computational power since they have lower control overhead
- They use dedicated functional units for specialized tasks to increase speeds
- GPU memory struggles with bandwidth limitations, and therefore aims for maximum bandwidth usage, employing strategies like data compression, multiple threads to cope with latency, scheduling of DRAM cycles to minimize idle data-bus time, etc.
- Caches are designed to support effective streaming with local reuse of data, rather than implementing a cache that achieves 99% hit rates (which isn't feasible), GPU cache designs assume a 90% hit rate with many misses in flight
- GPUs have many different performance regimes all with different characteristics and need to be designed accordingly

4.1 The Geforce 6800 as a general processor

You can see the Geforce 6800 as a general processor with a lot of floating-point horsepower and high memory bandwidth that can be used for other applications as well.

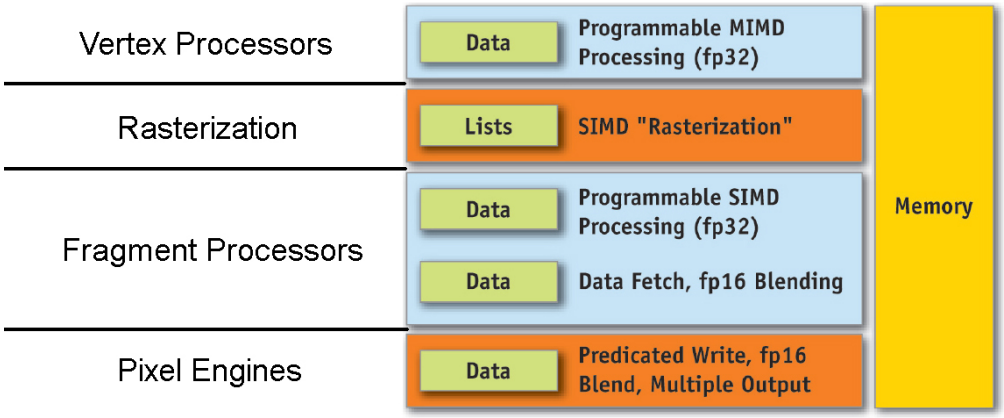


Figure 20: A general view of the Geforce 6800 architecture

Looking at the GPU that way, we get:

- 2 serially running programmable blocks with fp32 capability.
- The Rasterizer can be seen as a unit that expands the data into interpolated values (from one data-"point" to multiple "fragments").
- With MRT (Multiple Render Targets), the fragment processor can output up to 16 scalar floating-point values at a time.
- Several possibilities to control the data flow by using the visibility checks of the pixel engines or the Z-cull unit

5 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: a unified shader architecture (Note, that ATI already used this architecture in 2005 with the XBOX 360 GPU).

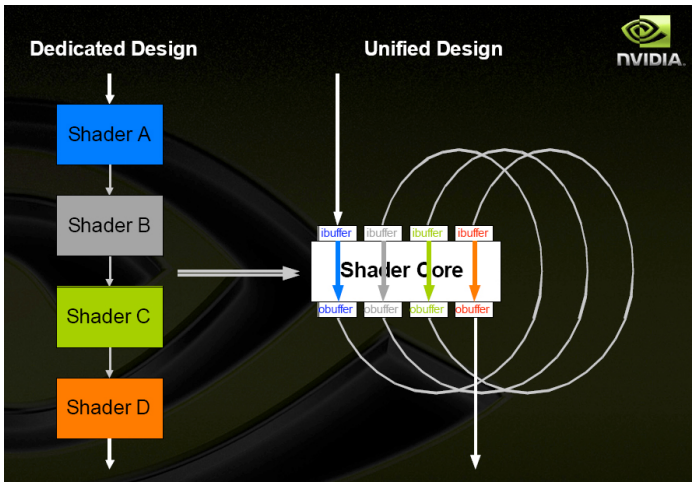


Figure 21: From dedicated to unified architecture

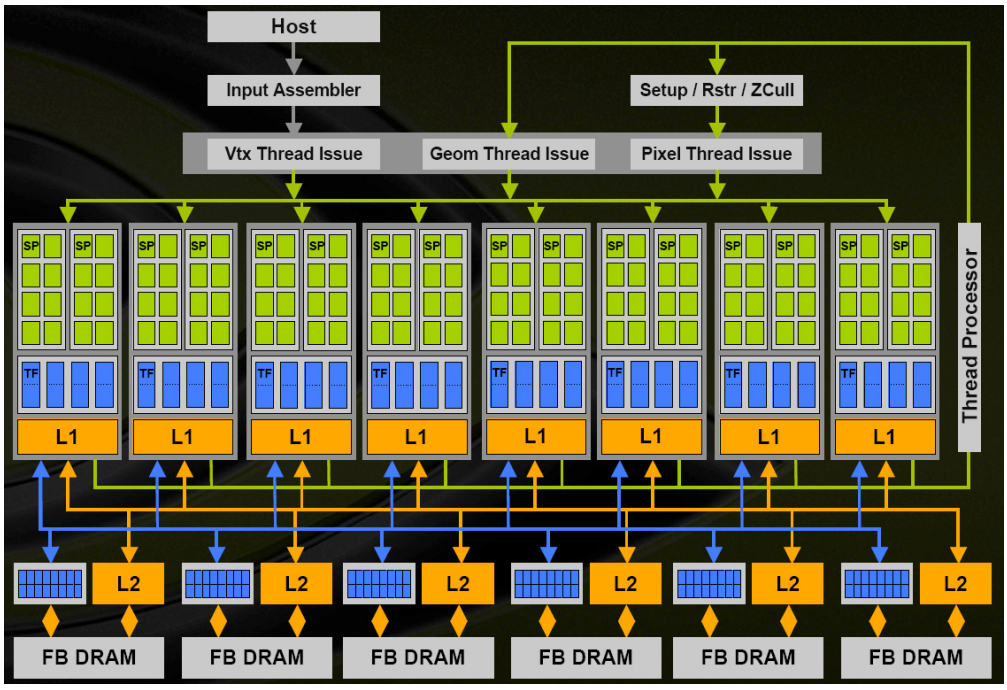


Figure 22: A schematic view of the Geforce 8800

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 shaders.
As you can see in Figure 22, 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 flows in the loop (vertex, geometry and pixel thread issue, input assembler and thread processor).

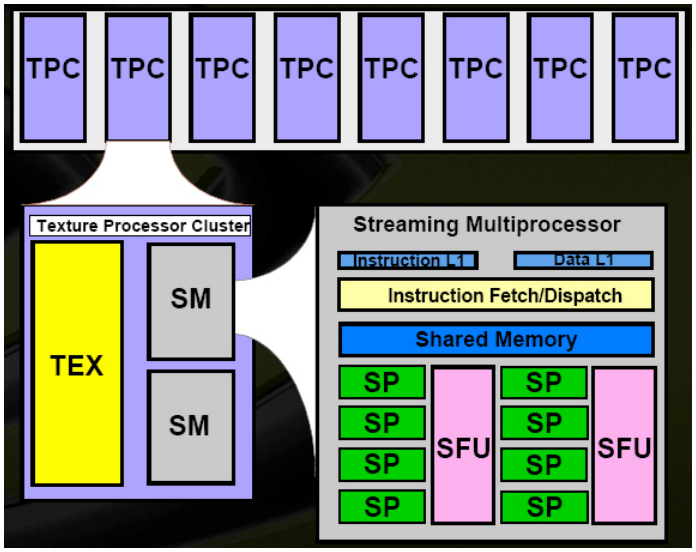


Figure 23: The streaming processor array

The streaming processor array consists of 8 texture processor clusters. Each texture processor cluster in turn consists of 2 streaming multiprocessors and 1 texture pipe.
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 a vector architecture more inefficient. They are driven by a high-speed clock that is seperate 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). The texture pipe consists of 4 texture addressing and 8 texture filtering units. It performs texture prefetching and filtering without consuming ALU resources, further increasing efficency.

It is apparent that we gain a number of advanteges with such a new architecture. For example, the old problem of constantly changing workload and one shader stage becoming a processing bottleneck is solved since the units can adapt dynamically, now that they are unified.

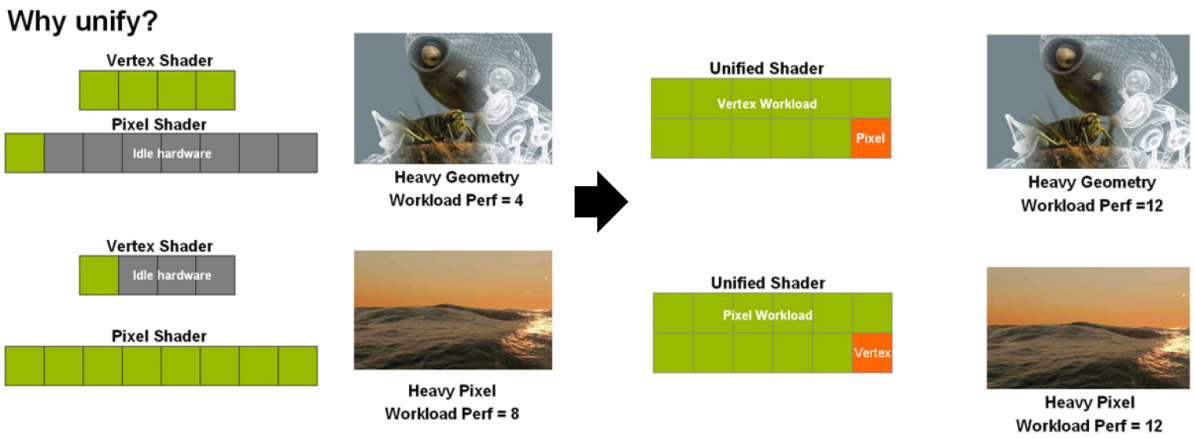


Figure 24: Workload balancing with both architectures

With a single instruction set and the support of fp32 throughout the whole pipeline, as well as the support of new data types (integer calculations), programming the GPU now becomes easier as well.

6 General Purpose Programming on the GPU - an example

We use the bitonic merge sort algorithm as an example for efficiently implementing algorithms on a GPU.

Bitonic merge sort:
Bitonic merge sort works by repeatedly building bitonic lists out of a set of elements and sorting them. A bitonic list is a concatenation of two monotonic lists, one ascending and one descending.

E.g.:
List A = (3,4,7,8)
List B = (6,5,2,1)
List AB = (3,4,7,8,6,5,2,1) is a bitonic list
List BA = (6,5,2,1,3,4,7,8) is also a bitonic list

Bitonic lists have a special property that is used in bitonic mergesort: Suppose you have a bitonic list of length 2n. You can rearrange the list so that you get two halves with n elements where each element (i) of the first half is less than or equal to each corresponding element (i+n) in the second half (or greater than or equal, if the list descends first and then ascends) and the new list is again a bitonic list. This happens by comparing the corresponding elements and switching them if necessary. This procedure is called a bitonic merge.

Bitonic merge sort works by recursively creating and merging bitonic lists that increase in their size until we reach the maximum size and the complete list is sorted. Figure 25 illustrates the process:

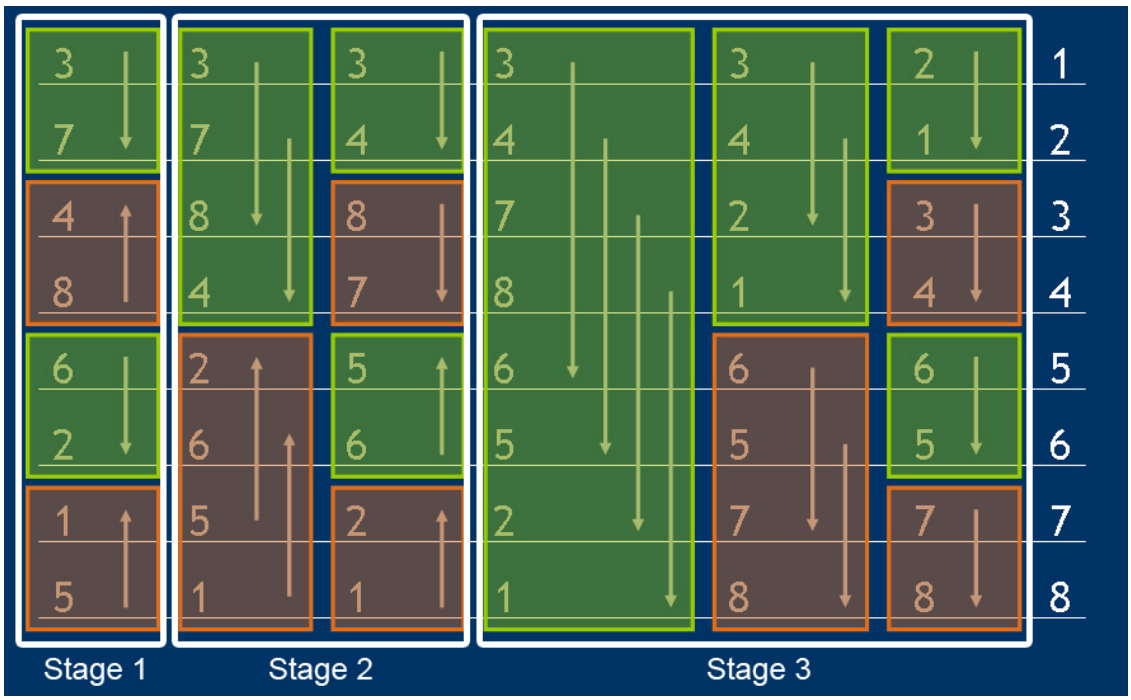


Figure 25: The different stages of the algorithm

The sorting process has a certain number of stages where comparison passes are performed. Each new stage increases the number of passes by one. This results in bitonic mergesort having a complexity of $O(n \log^2(n) + \log(n))$ which is worse than quicksort, but the algorithm has no worst-case scenario (where quicksort reaches $O(n^2)$.

The algorithm is very well suited for a GPU. Many of the operations can be performed in parallel and the length stays constant, given a certain number of elements. Now when implementing this algorithm on the GPU, we want to make use of as many resources as possible (both in parallel as well as vertically along

the pipeline), especially considering that the GPU has shortcomings as well, such as the lack of support for binary or integer operations. For example, simply letting the fragment processor stage handle all the calculations might work, but leaving all the other units unused is a waste of precious resources. A possible solution looks like this:

In this algorithm, we have groups of elements (fragments) that have the same sorting conditions, while groups next to each other operate in opposite. Now if we draw a vertex quad over two adjacent groups and set appropriate flags at each corner, we can easily determine group membership by using the rasterizer. For example, if we set the left corners to -1 and the right corners to +1, we can check where a fragment belongs to by simply looking at its sign (the interpolation process takes care of that).

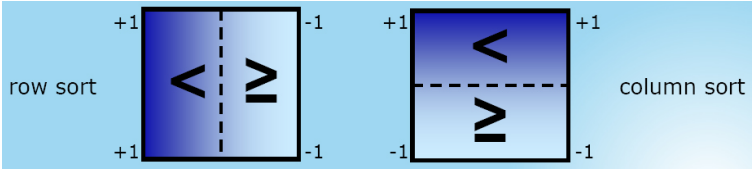


Figure 26: Using vertex flags and the interpolator to determine compare operations

Next, we need to determine which compare operation to use and we need to locate the partner item to compare. Both can again be accomplished by using the flag value. Setting the compare operation to less-than and multiplying with the flag value implicitly flips the operation to greater-equal halfway across the quad. Locating the partner item happens by multiplying the sign of the flag with an absolute value that specifies the distance between the items.

In order to sort elements of an array, we store them in a 2D texture.
Each row is a set of elements and becomes its own bitonic sequence that needs to be sorted. If we extend the quads over the rows of the 2D texture and use the interpolation, we can modulate the comparison so the rows get sorted either up or down according to their row number. This way, pairs of rows become bitonic sequences again which can besorted in the same way we sorted the columns of the single rows, simply by transposing the quads.

As a final optimization we reduce texture fetching by packing two neighbouring key pairs into one fragment, since the shader operates on 4-vectors.

Performance comparison:

std:sort: 16-Bit Data, Pentium 4 3.0 GHz			Bitonic Merge Sort: 16-Bit Float Data, NVIDIA Geforce 6800 Ultra			
N	Full Sorts/Sec	Sorted Keys/Sec	N	Passes	Full Sorts/Sec	Sorted Key/Sec
256^2	82.5	5.4 M	256^2	120	90.07	6.1 M
512^2	20.6	5.4 M	512^2	153	18.3	4.8 M
1024^2	4.7	5.0 M	1024^2	190	3.6	3.8 M

GLSL (OpenGL Shading Language) code sample, implementing the combined passes 0 and 1 for row-wise sorting of the bitonic merge sort:

```
uniform sampler2D PackedData:

// contents of the texcoord data
#define OwnPos gl_TexCoord[0].xy
#define SearchDir gl_TexCoord[0].z
#define CompOp gl_TexCoord[0].w
#define Distance gl_TexCoord[1].x
#define Stride gl_TexCoord[1].y
#define Height gl_TexCoord[1].z
#define HalfStrideMHalf gl_TexCoord[1].w

void main(void)
{
  // get self
  vec4 self = texture2D(PackedData, OwnPos);

  // restore sign of search direction and assemble vector to partner
  vec2 adr = vec2( (SearchDir < 0.0) ? -Distance : Distance , 0.0);

  // get the partner
  vec4 partner = texture2D(PackedData, OwnPos + adr);

  // switch ascending/descending sort for every other row
  // by modifying comparison flag
  float compare = CompOp * -(mod(floor(gl_TexCoord[0].y * Height), Stride) - HalfStrideMHalf);

  // x and y are the keys of the two items
  // --> multiply with comparison flag
  vec4 keys = compare * vec4( self.x, self.y, partner.x, partner.y);

  // compare the keys and store accordingly
  // z and w are the indices
  // --> just copy them accordingly
  vec4 result;
  result.xz = (keys.x < keys.z) ? self.xz : partner.xy;
  result.yw = (keys.y < keys.w) ? self.yw : partner.yw;

  // do pass 0
  compare *= adr.x;
  gl_FragColor = (result.x * compare < result.y * compare) ? result : result.yxwz;
}
```

7 Current and future developments

Nvidia's current top of the line model of graphics cards is the Geforce GTX 280 (GTX 200 series), an evolution of the unified shader architecture of the Geforce 8800, sporting almost double the shader count (from 128 to 240) and pushing up to 933 GFlops. Its launch date was the 17th of June 2008.

ATI (now merged into AMD) followed in 2007 with its first unified shader GPU for the PC (Radeon HD2900 XT), after succesfully using the architecture in the Xenos GPU for the XBox 360. Its current top of the line model is the Radeon HD 3870 X2 (which actually sports 2 GPUs on one card) which was released in January 2008.

ATI/AMD are soon to follow up with an answer to Nvidia's GTX 280: the Radeon HD 4870 (slated for somewhere around July 2008).

With the advent of the unified shader architecture, the topic of general-purpose computing on a GPU has become more and more relevant. Today, GPUs have made their way into non-graphics fields as varied as audio signal processing and weather forecasting.

Both ATI/AMD and Nvidia have made efforts for providing a programming interface for GPUs. ATI has released CTM (Close To Metal), a low level programming interface. After rewriting the software, CTM's commercial successor AMD Stream SDK was released in December 2007, now providing additional high level tools for general-purpose access to AMD graphics hardware.

Nvidia initially released the CUDA (Compute Unified Device Architecture) SDK in February 2007, a C language environment that enables programmers to write software for the GPU. In February 2008, Nvidia bought Ageia and their PhysX engine (a proprietary realtime physics engine middleware SDK) and integrated it into their CUDA framework.

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.

References

[1] Wikipedia entry on GPUs
`http://en.wikipedia.org/wiki/GPU`

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

[3] Cyril Zeller: "Introduction to the Hardware Graphics Pipeline", Presentation at ACM SIGGRAPH 2005
`http://download.nvidia.com/developer/presentations/2005/I3D/I3D_05_IntroductionToGPU.pdf`

[4] ExtremeTech 3D Pipeline Tutorial
`http://www.extremetech.com/article2/0,1697,9722,00.asp`

[5] Ashu Rege: "Introduction to 3D Graphics for Games"
`http://developer.nvidia.com/docs/IO/11278/Intro-to-Graphics.pdf`

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

[7] Mark Colbert: "GPU Architecture & CG"
`http://graphics.cs.ucf.edu/gpuseminar/seminar1.ppt`

[8] GPU Gems 2, Chapter 30: "The GeForce 6 Series GPU Architecture"
`http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf`

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

[10] www.3dcenter.de: "NV40-Technik im Detail"
`http://www.3dcenter.de/artikel/nv40_pipeline/`

[11] www.digit-life.com: “NVIDIA GeForce 6800 Ultra (NV40)”
http://www.digit-life.com/articles2/gffx/nv40-part1-a.html

[12] Austin Robison, Abe Winter: “An Overview of Graphics Processing Hardware”
http://people.cs.uchicago.edu/~robison/src/gpu_paper.pdf

[13] John Montrym, Henry Moreton: “NVIDIA GeForce 6800”, Hot Chips 16
http://www.hotchips.org/archives/hc16/2_Mon/13_HC16_Sess3_Pres1_bw.pdf

[14] Ajit Datar, Apurva Padhye: “Graphics Processing Unit Architecture”
http://www.d.umn.edu/~data0003/Talks/gpuarch.pdf

[15] Sven Schenk: “Eine Einfuehrung in die Architektur moderner Graphikprozessoren”
http://sus.ti.uni-mannheim.de/Lehre/Seminar0506/04modernGPUs.pdf

[16] Thomas Scott Crow: “Evolution of the Graphical Processing Unit”
http://www.cse.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf

[17] DirectX Developer Center: “Asm Shader Reference”
http://msdn.microsoft.com/en-us/library/bb219840(VS.85).aspx

[18] Erik Lindholm, Stuart Oberman: “NVIDIA GeForce 8800 GPU”
http://www.hotchips.org/archives/hc19/2_Mon/HC19.02/HC19.02.01.pdf

[19] www.digit-life.com: “Say Hello To DirectX 10, Or 128 ALUs In Action: NVIDIA GeForce 8800 GTX (G80)”
http://www.digit-life.com/articles2/video/g80-part1.html

[20] Richard Hough, Richard Yu: “GPU Architecture”
http://www.csl.cornell.edu/courses/ece685/slides/GPUArchitecture.ppt

[21] Technical Brief: “NVIDIA GeForce 8800 GPU Architecture Overview”
http://www.nvidia.com/object/IO_37100.html

[22] GPU Gems 2, Chapter 46: “Improved GPU Sorting”

[23] Tim Purcell: “Sorting and Searching”, SIGGRAPH 2005 GPGPU COURSE
http://www.gpgpu.org/s2005/slides/purcell.SortingAndSearching.ppt

[24] Peter Kipfer, Mark Segal, Ruediger Westermann: “UberFlow: A GPU-Based Particle Engine”
http://www.graphicshardware.org/previous/www_2004/Presentations/PeterKipfer.pdf

[25] Wikipedia entry on Nvidia
http://en.wikipedia.org/wiki/Nvidia_Corporation

[26] Wikipedia entry on ATI
http://en.wikipedia.org/wiki/ATI_Technologies_Inc.

[27] Wikipedia entry on CUDA
http://en.wikipedia.org/wiki/CUDA

[28] Wikipedia entry on CTM
http://en.wikipedia.org/wiki/Close_to_Metal

[29] William Mark, Henry Moreton: “3D Graphics Architecture Tutorial”
http://www-csl.csres.utexas.edu/users/billmark/talks/Graphics_Arch_Tutorial_Micro2004_BillMarkParts.pdf