Graphics Processing Unit
What is a GPU?

• 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
GPU vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially.
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds.
- A GPU is for the most part deterministic in its operation (though this is quickly changing).
- GPUs have much deeper pipelines (several thousand stages vs 10-20 or so for CPUs).
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs.
GPU Evolution

• 1980’s – No GPU. PC used VGA controller
• 1990’s – Add more function into VGA controller
• 1997 – 3D acceleration functions:
  - Hardware for triangle setup and rasterization
  - Texture mapping
  - Shading
• 2000 – A single chip graphics processor (beginning of GPU term)
• 2005 – Massively parallel programmable processors
• 2007 – CUDA (Compute Unified Device Architecture)
GPU Trends

• OpenGL – an open standard for 3D programming
• DirectX – a series of Microsoft multimedia programming interfaces
• New GPU’s are being developed every 12 to 18 months
• New idea of visual computing:
  combines graphics processing and parallel computing
• Heterogeneous System – CPU + GPU
• GPU evolves into scalable parallel processor
• GPU Computing: GPGPU and CUDA
• GPU unifies graphics and computing
• GPU visual computing application: OpenGL, and DirectX
Motherboard Bus Interface speeds

- PCI – peripheral component interconnect
  Originally: 133 MB/sec
  Recently: 512 MB/sec
  Upstream bandwidth 256MB/s peak

- AGP: Advanced Graphics Port – an interface between the computer core logic and the graphics processor
  AGP 1x: 266 MB/sec – twice as fast as PCI
  AGP 2x: 533 MB/sec
  AGP 4x: 1 GB/sec  AGP 8x: 2 GB/sec
  256 MB/sec readback from graphics to system

- PCIe: PCI-Express – a faster interface between the computer core logic and the graphics processor
  - v. 1.x (2.5 GT/s): 250 MB/s (×1) - 4 GB/s (×16)
  - v. 2.x (5 GT/s): 500 MB/s (×1) - 8 GB/s (×16)
  - v. 3.x (8 GT/s): 985 MB/s (×1) - 15.75 GB/s (×16)
  - v. 4.0 (16 GT/s): 1.969 GB/s (×1) - 31.51 GB/s (×16)

GT = Gigatransfers
Graphics Definitions

**Graphics primitive** - An elementary graphics building block, such as a point, line or arc. In a solid modeling system, a cylinder, cube and sphere are examples.

**Transform** is the task of converting spatial coordinates, which in this case involves moving three-dimensional objects in a virtual world and converting the coordinates to a two-dimensional view. **Clipping** means only drawing things that might be visible to the viewer. **Lighting** is the task of taking light objects in a virtual scene, and calculating the resulting colour of surrounding objects as the light falls upon them.

A **pixel shader** serves to manipulate a pixel color, usually to apply an effect on an image, for example; realism, bump mapping, shadows, and explosion effects. It is a graphics function that calculates effects on a per-pixel basis. Depending on resolution, an excess of 2 million pixels may need to be rendered, lit, shaded, and colored for each frame.

A **vertex shader** is a graphics processing function used to add special effects to objects in a 3D environment by performing mathematical operations on the objects' vertex data. Each vertex can be defined by many different variables. Vertices may also be defined by colors, textures, and lighting characteristics. Vertex Shaders don't actually change the type of data; they simply change the values of the data, so that a vertex emerges with a different color, different textures, or a different position in space.
Graphics Definitions

**interpolation** is a process where the software adds new pixels to an image based on the color values of the surrounding pixels. Interpolation is used when an image is upsampled to increase its resolution. Resampling through interpolation is not ideal and often results in a blurry image.

**Rasterization** - is the process of taking an image described in a vector graphics format and converting it into a raster image (pixels or dots) for output on a video display.

**Culling** – a GPU pipeline step that determines whether a polygon of a graphical object is visible.

**Viewport** - the 2D rectangle used to project the 3D scene to the position of a virtual camera. A viewport is a region of the screen used to display a portion of the total image to be shown.

**Z-buffer** - also known as depth buffering, is the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden.

**Fragment (pixel) shader** – a graphics processing function a computer program that is used to do shading: the production of appropriate levels of color within an image.

**Geometry shader** is a relatively new type of shader. This type of shader can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline. They take as input a whole primitive, possibly with adjacency information. For example, when operating on triangles, the three vertices are the geometry shader's input. The shader can then emit zero or more primitives, which are rasterized and their fragments ultimately passed to a pixel shader.
Barycentric coordinates

Barycentric coordinates are coordinates defined by the vertices of a simplex.

Barycentric or areal coordinates are extremely useful in engineering applications involving triangular subdomains. These make analytic integrals often easier to evaluate, and Gaussian quadrature tables are often presented in terms of area coordinates.

A simplex (plural simplexes or simplices) or n-simplex is an n-dimensional analogue of a triangle.

A 3-simplex or tetrahedron
STREAM PROCESSOR

- Stream processors are highly efficient computing engines that perform calculations on an input stream and produces an output stream that can be used by other stream processors.
- Stream processors can be grouped in close proximity, and in large numbers, to provide immense parallel processing power.

The **viewing frustum** is a geometric representation of the volume visible to the virtual camera. Naturally, objects outside this volume will not be visible in the final image, so they are discarded. Often, objects lie on the boundary of the viewing frustum. These objects are cut into pieces along this boundary in a process called clipping, and the pieces that lie outside the frustum are discarded as there is no place to draw them.
The host interface is the communication bridge between the CPU and the GPU. It receives commands from the CPU and also pulls geometry information from system memory. It outputs a stream of vertices in object space with all their associated information (normals, texture coordinates, per vertex color etc).

Code Snippet

```cpp
....
gBegin(GL_TRIANGLES);
TexCoord2f(1,0); glVertex3f(0,1,0);
TexCoord2f(0,1); glVertex3f(-1,-1,0);
TexCoord2f(0,0); glVertex3f(1,-1,0);
End();
...```

GPU pipeline example
GPU pipeline example

1. Program/API
2. Driver
3. GPU Front End
4. Vertex Processing
5. Primitive Assembly
6. Rasterization & Interpolation
7. Fragment Processing
8. Raster Operations
9. Framebuffer(s)
GPU pipeline example

Program/API

Driver

Bus

GPU Front End

Vertex Processing

Primitive Assembly

Rasterization & Interpolation

Fragment Processing

Raster Operations

Framebuffer(s)
GPU pipeline example

Program/API

Driver

GPU Front End

Vertex Processing

Primitive Assembly

Rasterization & Interpolation

Fragment Processing

Raster Operations

Framebuffer(s)

screen space
GPU pipeline example

1. Program/API
2. Driver
3. GPU Front End
4. Vertex Processing
5. Primitive Assembly
6. Rasterization & Interpolation
7. Fragment Processing
8. Raster Operations
9. Framebuffer(s)
GPU pipeline example
Adding Programmability to the Graphics Pipeline

Vertex and fragment processing, and now triangle set-up, are programmable. The programmer can write programs that are executed for every vertex as well as for every fragment. This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications.
Modern GPU Architecture

- Input from CPU
- Host interface
- Vertex processing
- Triangle setup
- Pixel processing
- Memory Interface
- 64bits to memory
- 64bits to memory
- 64bits to memory
- 64bits to memory
The CPU and GPU inside the PC work in parallel with each other.

There are two “threads” going on, one for the CPU and one for the GPU, which communicate through a command buffer:

- If this command buffer is drained empty, we are CPU limited and the GPU will idle while waiting for new input.
- If the command buffer fills up, the CPU will idle waiting for the GPU to consume it, and we are effectively GPU limited.
Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model. In the CPU program below, the object is not drawn after statement A and before statement B:

- Statement A
- API call to draw object
- Statement B

Instead, all the API call does, is to add the command to draw the object to the GPU command buffer.

This leads to a number of synchronization considerations:

In the figure below, the CPU must not overwrite the data in the “yellow” block until the GPU is done with the “black” command, which references that data:
Modern APIs implement semaphore style operations to keep this from causing problems.

If the CPU attempts to modify a piece of data that is being referenced by a pending GPU command, it will have to idle waiting, until the GPU is finished with that command.

While this ensures correct operation it is not good for performance since there are a million other things we’d rather do with the CPU instead of idling.

The GPU will also drain a big part of the command buffer thereby reducing its ability to run in parallel with the CPU.

One way to avoid these problems is to inline all data to the command buffer and avoid references to separate data:

However, this is also bad for performance, since we may need to copy several Mbytes of data instead of merely passing around a pointer.
A better solution is to allocate a new data block and initialize that one instead, the old block will be deleted once the GPU is done with it.

Modern APIs do this automatically, provided you initialize the entire block (if you only change a part of the block, renaming cannot occur).

Better yet, allocate all your data at startup and don’t change them for the duration of execution (not always possible, however).
CPU/GPU Interface

- Since the GPU is highly parallel and deeply pipelined, try to dispatch large batches with each drawing call.
- Sending just one triangle at a time will not occupy all of the GPU’s several vertex/pixel processors, nor will it fill its deep pipelines.
- Since all GPUs today use the zbuffer algorithm to do hidden surface removal, rendering objects front-to-back is faster than back-to-front (painters algorithm), or random ordering.
- Of course, there is no point in front-to-back sorting if you are already CPU limited.
GPUs evolved as hardware and software algorithms evolve.
Early Graphics

- Originally, no specialized graphics hardware
- All processing in software on CPU,
- Results transmitted to frame buffer
  - first, external frame buffers
  - later, internal frame buffers.
More detailed pipeline

Simple functionality transferred to specialized hardware.

- Geometry data
- Transform & lighting
- Culling, perspective divide, viewport mapping
- Rasterization
- Simple texturing
- Depth test
- Frame buffer blending
Add more functionality to GPU.

Simple functionality transferred to specialized hardware.

- Geometry data
- Transform & lighting
- Culling, perspective divide, viewport mapping
- Rasterization
- Simple texturing
- Depth test
- Frame buffer blending
Fixed function GPU pipeline

• Pipeline implemented in hardware
• Each stage does fixed task
• Tasks are parameterized
• Inflexible – fixed, parameterized functions
• Vector-matrix operations (some parallelism).
Technology advances

• Hardware gets cheaper, smaller, and more powerful

• Parallel architectures develop

• Graphics processing get more sophisticated (environmental mapping, displacement mapping, sub-surface scattering)

• Need more flexibility in GPUs.
Make this programmable: Vertex Shader

Geometry data

Transform & lighting

Culling, perspective divide, viewport mapping

Rasterization

Complex texturing

Depth test, alpha test, stencil test

Frame buffer blending

Make this programmable: Fragment Shader
Geometry data

- Culling, perspective divide, viewport mapping
- Rasterization
- Vertex Shader
- Fragment Shader
- Vertex Shader
- Fragment Shader
- Vertex Shader
- Fragment Shader

Introduce parallelism: add multiple units

- Alpha test, depth test, stencil test
- Frame buffer blending
Graphics Programming languages

- OpenGL and DirectX provide an abstraction of the hardware.

**Graphics pipelines for last 20 years**

*Processor per function*

- **Vertex**
  - T&L evolved to vertex shading

- **Triangle**
  - Triangle, point, line – setup

- **Pixel**
  - Flat shading, texturing, eventually Pixel shading

- **ROP**
  - Blending, Z-buffering, antialiasing

- **Memory**
  - Wider and faster over the years
Trend from pipeline to data parallelism

Pipe (Clark “Geometry Engine” (1983))
- Coordinate Transform
- 6-plane Frustum Clipping
- Divide by w Viewport

SGI 4D/GTX (1988)
- Coord, normal Transform
- Lighting
- Clip testing
- Clipping state
- Divide by w (clipping)
- Viewport
- Prim. Assy.
- Backface cull

SGI RealityEngine (1992)
- Command Processor
- Round-robin Aggregation
Shading language

- Shade trees -> Pixar’s Renderman shader

Figure 1a. Shade tree for copper.
Shader Language

• Low level (like assembler) but high-level language compilers: nVidia’s Cg
• 4 component floating point data type
• SIMD
Cg: C-based graphics program

- Array & structures
- Flow control
- Vectors & matrices
- No memory allocation, file I/O
Next: unify shaders

- One set of shaders
- Allocate to either vertices or fragments
Impact of Unified Shaders

All shading processes performed by a unified set of processors
  Fewer bottle-necks (i.e. in case of vertex or pixel dominant scenes)
  Better hardware utilization
Hardware architecture no longer reflects the graphics pipeline
Greater flexibility makes GPUs eligible for nongraphics applications (game physics, scientific applications)

Basically makes the GPU a massively parallel stream multiprocessor!
FIGURE A.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors. Copyright © 2009 Elsevier, Inc. All rights reserved.
Processor Array

TPC – texture processor cluster
SP – streaming processor
ROP – raster operation processor
SM – streaming multiprocessor
SFU – special fn unit
nVidia G80 GPU
Architecture Overview

• 16 Multiprocessors Blocks
• Each MP Block Has:
  • 8 Streaming Processors (IEEE 754 spfp compliant)
  • 16K Shared Memory
  • 64K Constant Cache
  • 8K Texture Cache
• Each processor can access all of the memory at 86Gb/s, but with different latencies:
  • Shared – 2 cycle latency
  • Device – 300 cycle latency
Graphics Demos

Unreal Engine 4 – Paris 2015 demo
https://www.youtube.com/watch?v=QS1HQFizDx4

Nvidia Face Works – Titan Z
https://www.youtube.com/watch?v=7fqEAzMZhJl
https://www.youtube.com/watch?v=z0cZin2xDmQ

Unreal Engine 4 – 2015 Titan Z demo
https://www.youtube.com/watch?v=XISqvBVyASo
GPGPU

- GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine.

- The raw computational power of a GPU dwarfs that of the most powerful CPU, and the gap is steadily widening.

- Make GPU more general – adapt certain types of programs to it’s pipelined, parallel architecture

- Nvidia GeForce 8800 chip achieves a sustained 330 billion floating-point operations per second (Gflops) on simple benchmarks

- Cost-effective: graphics driving demand up, supply up, price down for GPUs

- Finding uses in non-graphics applications.
What is the GPU Good at?

The GPU is good at data-parallel processing

The same computation executed on many data elements in parallel – low control flow overhead with high SP floating point arithmetic intensity

Many calculations per memory access

Currently also need high floating point to integer Ratio

High floating-point arithmetic intensity and many data elements mean that memory access latency can be hidden with calculations instead of big data caches –

Still need to avoid bandwidth saturation!
GPGPU Applications

Scientific computing and physical simulation
   Solving PDEs
   Reaction-Diffusion
Fluid and molecular dynamics
   N-body simulation
Signal processing
   FFT, DCT, video processing
Geometric computations
Distance computations
   Collision detection
Proximity computations
Computer vision
   Real-time feature tracking
Financial forecasting
Database computations
   Many more
Example: Crack the Windows Vista logon password

**Encrypted using NTLM hashing**

- Microsoft authentication protocol
- Random challenge-response Sequence
- Considered hard to crack

**Brute force technique required**

Send many, many requests until you score a right guess
Hence a lot of computing power required

**With $150 graphics card, just 3-5 days**

Speedup to high-end dual-core CPU: 25x
The Problem: Difficult To Use

GPUs designed for & driven by video games
  Programming model unusual
  Programming idioms tied to computer graphics
  Programming environment tightly constrained

Underlying architectures are:
  Inherently parallel
  Rapidly evolving (even in basic feature set!)
  Largely secret

You cannot simply “port” CPU code!
Programming the GPU for non-graphics applications

The past (until 2005):
- Graphics API - cumbersome when you don’t actually want graphics…
- Cast input data into textures
- perform computation with shaders
- Memory accesses done as pixels
- Reshape algorithm to work around hardware limitations (i.e. no scatter)

The present
- High level language extensions
- More flexible hardware
- GPGPU SDKs from GPU vendors
  ATI’s CTM (Close-to-Metal)
  Nvidia’s CUDA (Compute Unified Driver Architecture)
CUDA sees the G80 as this:
CUDA Programming Model

The GPU is viewed as a compute device that:

- Is a coprocessor to the CPU or host
- Has its own DRAM (device memory)
- Runs many threads in parallel Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

Differences between GPU and CPU threads

- GPU threads are extremely lightweight
- Very little creation overhead
- GPU needs 1000s of threads for full efficiency
- Multi-core CPU needs only a few
New Platform: Tesla

**Tesla C870 GPU Computing Processor**
- One Tesla GPU (128 thread processors)
- 1.5 GB dedicated memory
- Fits in one full-length, dual slot with one open PCI Express x16 slot

**Tesla D870 Deskside GPU Computing System**
- Two Tesla GPUs (128 thread processors per GPU)
- 3 GB system memory (1.5 GB dedicated memory per GPU)
- Quiet operation (40dB) suitable for office environment
- Connects to host via cabling to a low power PCI Express x8 or x16 adapter card
- Optional rack mount kit

**Tesla S870 GPU Computing System**
- Four Tesla GPUs (128 thread processors per GPU)
- 6 GB of system memory (1.5 GB dedicated memory per GPU)
- Standard 19", 1U rack-mount chassis
- Connects to host via cabling to a low power PCI Express x8 or x16 adapter card
- Configuration: 2 PCI Express connectors driving 2 GPUs each (4 GPUs total)
Sony PS3 Graphics

- Processing
  - 3.2Ghz Cell: PPU and 7 SPUs
    - PPU: PowerPC based, 2 hardware threads
    - SPUs: dedicated vector processing units
  - RSX®: high end GPU

- Data flow
  - IO: BluRay, HDD, USB, Memory Cards, GigaBit ethernet
  - Memory: main 256 MB, video 256 MB
  - SPUs, PPU and RSX® access main via shared bus
  - RSX® pulls from main to video
Cell Processor
Sony RSX Graphics processor

- Based on a high end NVidia chip
  - Fully programmable pipeline: shader model 3.0
  - Floating point render targets
  - Hardware anti-aliasing (2x, 4x)
- 256 MB of dedicated video memory
- PULL from the main memory at 20 GB/s
- HD Ready (720p/1080p)
  - 720p = 921,600 pixels
  - 1080p = 2,073,600 pixels

→ a high end GPU adapted to work with the Cell Processor and HD displays
**XBOX 360**

- 512 MB system memory
- IBM 3-way symmetric core processor
- ATI GPU with embedded EDRAM
- 12x DVD
- Optional Hard disk
- Custom silicon designed by ATi Technologies Inc.
- 500 MHz, 338 million transistors, 90nm process
- Supports vertex and pixel shader version 3.0+
- Includes some Xbox 360 extensions
XBOX 360

- 10 MB embedded DRAM (EDRAM) for extremely high-bandwidth render targets
  - Alpha blending, Z testing, multisample antialiasing are all free (even when combined)
- Hierarchical Z logic and dedicated memory for early Z/stencil rejection
- GPU is also the memory hub for the whole system
  - 22.4 GB/sec to/from system memory
  - 48 shader ALUs shared between pixel and vertex shading (unified shaders)
  - Each ALU can co-issue one float4 op and one scalar op each cycle
- Non-traditional architecture
- 16 texture samplers
- Dedicated Branch instruction execution
XBOX 360

- 2x and 4x hardware multi-sample anti-aliasing (MSAA)
- Hardware tessellator
  - N-patches, triangular patches, and rectangular patches
- Can render to 4 render targets and a depth/stencil buffer simultaneously

**GPU workflow:**
- Consumes instructions and data from a command buffer
- Ring buffer in system memory
- Managed by Direct3D, user configurable size (default 2 MB)
- Supports indirection for vertex data, index data, shaders, textures, render state, and command buffers
- Up to 8 simultaneous contexts in-flight at once
- Changing shaders or render state is inexpensive, since a new context can be started up easily
GPU Workflow

- Threads work on units of 64 vertices or pixels at once
- Dedicated triangle setup, clipping, etc.
- Pixels processed in 2x2 quads
- Back buffers/render targets stored in EDRAM
  - Alpha, Z, stencil test, and MSAA expansion done in EDRAM module
- EDRAM contents copied to system memory by “resolve” hardware
- Write 8 pixels or 16 Z-only pixels to EDRAM
- With MSAA, up to 32 samples or 64 Z-only samples
- Reject up to 64 pixels that fail Hierarchical Z testing
- Vertex fetch sixteen 32-bit words from up to two different vertex streams
- 16 bilinear texture fetches
- 48 vector and scalar ALU operations
- Interpolate 16 float4 shader interpolants
- 32 control flow operations
- Process one vertex, one triangle
- Resolve 8 pixels to system memory from EDRAM
Direct3D 9+ on XBOX 360

- Communicates with GPU via a *command buffer*
- Ring buffer in system memory
- Direct Command Buffer Playback support
- Ring buffer allows the CPU to safely send commands to the GPU
- Buffer is filled by CPU, and the GPU consumes the data
PS4 vs XBOX ONE

• The Xbox has a more powerful CPU - The PS4 has a more powerful GPU.

• Xbox One has a custom 1.75GHz AMD 8-core CPU, a last-minute upgrade over its original 1.6GHz processor.
• The PS4 CPU remained clocked at 1.6GHz and contains a similar custom AMD 8-core CPU with x86 based architecture.

• PS4 boasts a 1.84 teraflop GPU that's based on AMD's Radeon technology.
• The Xbox One graphics chip, also with an AMD Radeon GPU, has a pipeline for 1.31 teraflops.

• Both systems have 8GB of RAM overall. But they allocate that memory to developers differently.
• PS4 has a distinct advantage with faster 8GB GDDR5 memory, while Xbox One went with the slower bandwidth of the 8GB DDR3 variety

• PS4 reserves up to 3.5GB for its operating system, leaving developers with 4.5GB, according to documentation. They can sometimes access an extra 1GB of "flexible" memory when it's available, but that's not guaranteed.
• Xbox One's "guaranteed memory" amounts to a slightly higher 5GB for developers, as Microsoft's multi-layered operating system takes up a steady 3GB. It eeks out a 0.5GB win with more developer-accessible memory than PS4, unless you factor in Sony's 1GB of "flexible" memory at times. Then it's 0.5GB less.
Nvidia GPGPU demo:

Procedural generation

https://www.youtube.com/watch?v=XSHBn7hOyDw