The Way of the GPU
(based on GPGPU SIGGRAPH Course)

CS334
Spring 2023

Daniel G. Aliaga
Department of Computer Science
Purdue University
Computer Graphics Pipeline

Geometry

- Modeling Transformation: Transform into 3D *world* coordinate system
- Lighting: Simulate illumination and reflectance
- Viewing Transformation: Transform into 3D *camera* coordinate system
- Clipping: Clip primitives outside camera’s view
- Projection: Transform into 2D camera coordinate system
- Scan Conversion: Draw pixels (incl. texturing, hidden surface…)

Image
Today, we have GPUs...

(GPU = graphical processing unit)
Some history and context...

- 1980s
- 1990s and programming shading
- 2000s and PCs
- GPUs...
Motivation: Computational Power

- Why are GPUs fast?
  - Arithmetic intensity: the specialized nature of GPUs makes it easier to use additional transistors for computation not cache
  - Economics: multi-billion dollar video game market is a pressure cooker that drives innovation
Motivation: Flexible and Precise

- **Modern GPUs are deeply programmable**
  - Programmable pixel, vertex, video engines
  - Solidifying high-level language support
- **Modern GPUs support high precision**
  - 32 bit floating point throughout the pipeline
  - High enough for many (not all) applications
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
- Can’t simply “port” CPU code!
Diagram of a Modern GPU

Input from CPU

Host interface
Geometry/Vertex processing
Triangle setup
Pixel processing
Memory Interface

fast memory
fast memory
fast memory
fast memory
nVIDIA GPU

• **GTX 3090 Founder’s Edition**
  – 10496 (CUDA) cores @ 1.7GHz (i.e., mini processors)
  – 936 GB/sec (memory bandwidth)
  – 36 TFLOPS (shader)
  – 24 GB video memory
  – 7680x4320 pixels
  – 350W power
  – 91°C max GPU temp
  – $1500-$3000
nVIDIA GPU

- GeForce 256 (from 1999)
  - 120 MHz
  - 4.8 GB/sec (memory bandwidth)
  - 32 MB memory
  - $100
Before...

  - 2-4 raster boards (i.e., boards used in parallel)
  - 0.8 GB/sec (memory bandwidth)
  - 0.000640 TFLOPS
  - 2560x2048 pixels
  - ?? power
  - ?? max GPU temp
  - $390,000
Before

- SGI Personal IRIS 4D (1985)
  - 0.000000940 TFLOPS
  - $68000
Before

- **IBM PC 5150 (~1985)**
  - 0.000004.77 GHz
  - 16-640 KB
  - ~200W power
Modern GPU has more ALU’s

Figure 1-2. The GPU Devotes More Transistors to Data Processing
GPU Pipeline: Transform

- Vertex/Geometry processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-primitive and per-vertex lighting
GPU Pipeline: Rasterize
(typically not programmable)

• Rasterizer
  – Convert geometric rep. (vertex) to image rep. (fragment)
    • Fragment = image fragment
      – Pixel + associated data: color, depth, stencil, etc.
  – Interpolate per-vertex quantities across pixels
GPU Pipeline: Shade

• Fragment processors (multiple in parallel)
  – Compute a color for each pixel
  – Optionally read colors from textures (images)
GPU Programming Languages

• Many options!
  – A while ago: “Renderman”
  – cG (from NVIDIA)
  – GLSL (GL shading Language)
  – CUDA (more general that graphics)...

• Lets focus first on the concept, later on the language specifics...
GLSL Demo

- [http://glslsandbox.com/](http://glslsandbox.com/)

(backup: [https://www.youtube.com/watch?v=9ETfgTD6L2I](https://www.youtube.com/watch?v=9ETfgTD6L2I) [https://www.youtube.com/watch?v=8gHx7nMCVp4](https://www.youtube.com/watch?v=8gHx7nMCVp4) [https://www.youtube.com/watch?v=t2yPfenzkII](https://www.youtube.com/watch?v=t2yPfenzkII) [https://www.youtube.com/watch?v=M_FsjL9j0HY](https://www.youtube.com/watch?v=M_FsjL9j0HY))
Mapping Parallel Computational Concepts to GPUs

• GPUs are designed for graphics
  – Highly parallel tasks

• GPUs process *independent* vertices & fragments
  – Temporary registers are zeroed
  – No shared or static data
  – No read-modify-write buffers

• Data-parallel processing
  – GPUs architecture is ALU-heavy
    • Multiple vertex & pixel pipelines, multiple ALUs per pipe
  – Hide memory latency (with more computation)
Example: Simulation Grid

- Common GPGPU computation style
  - Textures represent computational grids = streams
- Many computations map to grids
  - Matrix algebra
  - Image & Volume processing
  - Physically-based simulation
  - Global Illumination
    - ray tracing, photon mapping, radiosity
- Non-grid streams can be mapped to grids
e.g.: Scatter vs. Gather

- Grid communication
  - Grid cells share information
Vertex Processor

• Fully programmable (SIMD / MIMD)
• Processes 4-vectors (RGBA / XYZW)
• Capable of scatter but not gather
  – Can change the location of current vertex
  – Cannot read info from other vertices
  – Can only read a small constant memory
• Latest GPUs: Vertex Texture Fetch
  – Random access memory for vertices
  – \(\approx\)Gather (But not from the vertex stream itself)
Fragment Processor

- Fully programmable (SIMD)
- Processes 4-component vectors (RGBA / XYZW)
- Random access memory read (textures)
- Capable of gather but not scatter
  - RAM read (texture fetch), but no RAM write
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - Direct output (fragment processor is at end of pipeline)
GPU Simulation Overview

• A Simulation:
  – Its algorithm steps are fragment programs
    • Called Computational *kernels*
  – Current state is stored in textures
  – Feedback via “render to texture”

• Question:
  – How do we invoke computation?
Invoking Computation

• Must invoke computation at each pixel
  – Just draw geometry!
  – Most common GPGPU invocation is a full-screen quad

• Other Useful Analogies
  – Rasterization = Kernel Invocation
  – Texture Coordinates = Computational Domain
  – Vertex Coordinates = Computational Range
Typical “Grid” Computation

• Initialize “view” (so that pixels:texels::1:1)
  
  ```
  glmMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glmMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glmOrtho(0, 1, 0, 1, 0, 1);
  glViewport(0, 0, outTexResX, outTexResY);
  ```

• For each algorithm step:
  – Activate render-to-texture
  – Setup input textures, fragment program
  – Draw a full-screen quad (1x1)
Example: N-Body Simulation

- Brute force 😞
- N = 8192 bodies
- $N^2$ gravity computations
- 64M force comps. / frame
- ~25 flops per force
- 10.5 fps
- 17+ GFLOPs sustained in this example
Computing Gravitational Forces

• Each body attracts all other bodies
  – $N$ bodies, so $N^2$ forces

• Draw into an $N\times N$ buffer
  – Pixel $(i,j)$ computes force between bodies $i$ and $j$
  – Very simple fragment program
    • More than $N=2048$ bodies is tricky
    • Why?
Computing Gravitational Forces

\[ F(i,j) = \frac{gM_i M_j}{r(i,j)^2}, \]

where \( r(i,j) = |\text{pos}(i) - \text{pos}(j)| \).

Force is proportional to the inverse square of the distance between bodies.
float4 force(float2 ij : WPOS,
    uniform sampler2D pos) : COLOR0
{
    // Pos texture is 2D, not 1D, so we need to
    // convert body index into 2D coords for pos tex
    float4 iCoords = getBodyCoords(ij);
    float4 iPosMass = texture2D(pos, iCoords.xy);
    float4 jPosMass = texture2D(pos, iCoords.zw);
    float3 dir = iPos.xyz - jPos.xyz;
    float r2 = dot(dir, dir);
    dir = normalize(dir);
    return dir * g * iPosMass.w * jPosMass.w / r2;
}
Computing Total Force

- Have: array of (i, j) forces
- Need: total force on each particle i
  - Sum of each column of the force array
- Can do all N columns in parallel

This is called a Parallel Reduction
Geometry processing on GPUs

• so far: GPGPU limited to texture output

• new APIs allow geometry generation on GPU
Examples

Fluid Simulation

Particles

3D Smoke & Fire

Water Simulation

Grid displacement

3D Water Surfaces
Examples

Fluid Simulation

Particles

3D Smoke & Fire

Water Simulation

Grid displacement

3D Water Surfaces

Point Compression

Point Decompression

Point Rendering
High Level Shading Languages

• Cg, HLSL, & OpenGL Shading Language
  – Cg:
    • http://www.nvidia.com/cg
  – HLSL:
  – OpenGL Shading Language:
    • http://www.3dlabs.com/support/developer/ogl2/whitepapers/index.html
‘printf’ Debugging

• MOV suspect register to output
  – Comment out anything else writing to output
  – Scale and bias as needed
• Recompile
• Display/readback frame buffer
• Check values
• Repeat until error is (hopefully) found
‘printf’ Debugging Examples
‘printf’ Debugging Examples
Asgn 2

- [https://docs.google.com/document/d/1ZuVCdv5MxMrgdEMp8HrXrCnMwSoArbrSBu20oyyaOil/edit?usp=sharing](https://docs.google.com/document/d/1ZuVCdv5MxMrgdEMp8HrXrCnMwSoArbrSBu20oyyaOil/edit?usp=sharing).