GPU Architecture
This Week: GPU Programming

1. Processor Architecture
2. GPU Register Model
3. Shader Launch
4. GLSL Language
5. Implicit Surfaces
6. Sphere Tracing Algorithm
7. Leveraging Reference Frames
8. Procedural Texture
GPU ARCHITECTURE (++CS237)
State

- **Program Counter** a.k.a. Instruction Pointer ("PC" or "IP")
- [Stack pointer ("SP"), Base pointer ("BP"), Condition codes]
- General-purpose **registers** ("reg" or "GPR")
- Fast, small **memory** (today: on-chip caches and local/shared memory)
- Slow, large **memory** (today: off-chip DRAM…and network and disk)
Generic Processor

Main Memory

i$ Instructions

d$ Data

Decode & Dispatch

Register File

ALU

Control

Processor

$ = "cache"

ALU = Arithmetic Logic Unit
Streaming Vector Processor

**Simplify**
- No out-of-order, branch prediction, etc.
- Many ALUs and registers in saved space

**Fast context switch**
- 100,000s of threads
- Swap during memory latency

**Special stream in/out path**
- Bypass cache
- Self-synchronized by scheduling

**Vectorize instructions**
- Execute each 32x
- Amortize instruction fetch & decode
Streaming Vector Processor

Implications:

- Explicit memory load is slow
- Explicit memory store is very slow (and unsafe)
- Stack is slow (affects context swap)
- Pointers are slow (most data are in registers!)
- Computed array indexing is slow (same as above)
- Divergence is slow (splits vectors)
- Everything else is very fast!

Simplify
- No out-of-order, branch prediction, etc.
- Many ALUs and registers in saved space

Fast context switch
- 100,000s of threads
- Swap during memory latency

Special stream in/out path
- Bypass cache
- Self-synchronized by scheduling

Vectorize instructions
- Execute each 32x
- Amortize instruction fetch & decode
HLGL/GLSL

• **Designed to prevent writing slow programs**
  – Stack is slow → no recursion, fully-inlined program
  – Pointers are slow → no pointers
  – Memory store is slow → hard to use
  – Memory load is slow → abstract through “texture” functions
  – Abstractions might hide slow code → only C-level features

• **Result:** awkward, but fast

• *You can* use other languages for GPU programming (CUDA, OpenCL, C++ -> SPIRV, etc.)
GLSL Syntax Tips

• “in out” = C++ &
• “const” = C++ “static const”…must be compile-time evaluable
• Use #define instead of typedef
• Maximum single-precision float and int (no double or long)
• No heap allocation
• No Doxygen support
GLSL Inlines Everything

- Small, fixed-length loops unroll completely
- Branches on compile-time constants are free
- Dead code is free
- No pointers
- No exceptions
- No recursion allowed (but you can build your own slow stack)
- No function pointers (and no classes or methods, but you can use switch)
- No pointers + no recursion = no data structures except fixed-length array and struct
GLSL Hidden Performance

- More registers = fewer threads = slower
- Float is much faster than int!
- Some intrinsics are very fast:
  - normalize()
  - dot()
- Multiply-add is a single operation
- Branches and loops have high overhead and create divergence
  - But conditional assignment is inexpensive!
- Computed array indices force the array out of registers into cache (slow!)
GPU SHADER LAUNCH
Shader Launch Modes

• “Compute” mode
  – Iterate over 1D, 2D, or 3D rectangular bounds
  – Like G3D::Thread::runConcurrently on CPU

• “Graphics” mode
  – Iterate over an indexed triangle list’s vertices, geometry, and pixels
  – Special depth test support for pixels
  – Automatic clipping to 3D frustum
  – Some other exotic cases (tessellated patches, stencils, paths, axis-aligned rectangles)
G3D Syntax
for a pixel shader on a full-screen rectangle in “Graphics” mode

// Prepare for the 2D rectangle
rd->push2D();

// Pass arguments from the CPU to the GPU
Args args;
args.setUniform("center", Point3(x, y, z));
args.setUniform("environmentMap", environmentMap, Sampler::cubeMap());
...

// Set the indexed triangle list to two triangles covering the near plane
args.setRect(rd->viewport());

LAUNCH_SHADER("mycode.pix", args);
rd->pop2D();
THE RASTERIZER