GPGPU: Beyond Graphics
Mark J. Harris

Motivation: Computational Power
- GPUs are fast...
  - 3 GHz Pentium4 theoretical: 6 GFLOPS
  - 5.96 GB/sec peak
  - GeForce FX 5900 observed*: 20 GFLOPs
  - 25.6 GB/sec peak
  - GeForce 6800 Ultra observed*: 40 GFLOPs
  - 35.2 GB/sec peak

  *Observed on a synthetic benchmark:
    - A long pixel shader with nothing but MUL instructions

Courtesy Ian Buck

GPU performance increasing faster CPU
- CPUs: annual growth ~1.5× → decade growth ~ 60× Moore’s law
- GPUs: annual growth > 2.0× → decade growth > 100×
  - Much faster than Moore’s law

Why are GPUs getting faster so fast?
- Arithmetic intensity
  - 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

Graph courtesy Avneesh Sud, UNC

GPU floating point power is following a similar trend!

Moore’s Law?
Motivation: Flexible and precise

- Modern GPUs are programmable
- Programmable pixel and vertex engines
- High-level language support
- Modern GPUs support high precision
- 32-bit floating point throughout the pipeline
- High enough for many (not all) applications

Motivation: The Potential of GPGPU

- The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
- Example applications (from www.GPGPU.org)
  - Advanced Rendering: Global Illumination, Image-based Modeling
  - Computational Geometry
  - Computer Vision
  - Image And Volume Processing
  - Scientific Computing: physically-based simulation, linear system solution, FEMs
  - Stream Processing
  - Database queries
  - Monte Carlo Methods

The Problem: Difficult To Use

- GPUs designed for and driven by graphics
- Programming model is unusual & tied to graphics
- Programming environment is tightly constrained
- Underlying architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret
- Can’t simply “port” code written for the CPU!

Mapping Computational Concepts to GPUs

- Rest of the Talk:
  - Data Parallelism and Stream Processing
  - Computational Resources Inventory
  - CPU-GPU Analogies
  - Flow Control Techniques
  - Examples and Future Directions

Importance of Data Parallelism

- GPU: Each vertex / fragment is independent
- Temporary registers are zeroed
- No static data
- No read-modify-write buffers
- Data parallel processing
- Best for ALU-heavy architectures: GPUs
- Multiple vertex & pixel pipelines
- Hide memory latency (with more computation)
- GPU is a Stream Processor

Arithmetic Intensity

- Lots of ops per word transferred
- Graphics pipeline
  - Vertex
    - BW: 1 triangle = 32 bytes;
    - OP: 100-500 f32-ops / triangle
  - Rasterization
    - Create 16-32 fragments per triangle
  - Fragment
    - BW: 1 fragment = 10 bytes
    - OP: 300-1000 i8-ops/fragment

Courtesy of Ian Buck

Courtesy of Pat Hanrahan
Data Streams & Kernels

Streams
- Collection of records requiring similar computation
  - Vertex positions, Voxels, FEM cells, etc.
- Provide data parallelism

Kernels
- Functions applied to each element in stream
  - transforms, PDE, …
- Few dependencies between stream elements
- Encourage high Arithmetic Intensity

Example: Simulation Grid

Common GPGPU computation style
- Textures represent computational grids = streams
- Many computations map to grids
  - Matrix algebra
  - Image & Volume processing
  - Physical simulation
  - Global Illumination
  - ray tracing, photon mapping, radiosity
- Non-grid streams can be mapped to grids

Stream Computation

Grid Simulation algorithm
- Made up of steps
- Each step updates entire grid
- Must complete before next step can begin
- Grid is a stream, steps are kernels
- Kernel applied to each stream element

Scatter vs. Gather

Grid communication (a necessary evil)
- Grid cells share information
- Two ways:
  - Scatter
  - Gather

Computational Resources Inventory

Programmable parallel processors
- Vertex & Fragment pipelines
- Rasterizer
  - Mostly useful for interpolating addresses (texture coordinates) and per-vertex constants
- Texture unit
  - Read-only memory interface
  - Render to texture
  - Write-only memory interface

Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex (scatter)
  - Cannot read info from other vertices (gather)
  - Small constant memory
- New GeForce 6 Series features:
  - Pseudo-gather: read textures in the vertex program
  - MIMD: independent per-vertex branching, early exit
Fragment Processor

- Fully programmable (SIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of gather but not scatter
  - Random access memory read (textures)
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
- More fragment pipelines than vertex pipelines
- RAM read
- Direct output
- GeForce 6 Series adds SIMD branching
- GeForce FX only has conditional writes

CPU-GPU Analogies

- CPU programming is (assumed) familiar
- GPU programming is graphics-centric
- Analogies can aid understanding

**CPU-GPU Analogies**

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Read</td>
<td>Texture Sample</td>
</tr>
</tbody>
</table>

Feedback

- Each algorithm step depend on the results of previous steps
- Each time step depends on the results of the previous time step

**CPU-GPU Analogies**

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Render to Texture</td>
</tr>
</tbody>
</table>

Array Write = Render to Texture
Tutorial 5: Programming Graphics Hardware

**GPU Simulation Overview**
- Analogies lead to implementation
- Algorithm steps are fragment programs
- Computational “kernels”
- Current state variables stored in textures
- Feedback via render to texture
- One 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

**Standard “Grid” Computation**
- Initialize “view” (so that pixels:texels::1:1)
  
```c
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, 1, 0, 1, 0, 1);
  glViewport(0, 0, gridResX, gridResY);
```
- For each algorithm step:
  - Activate render-to-texture
  - Setup input textures, fragment program
  - Draw a full-screen quad (1 unit x 1 unit)

**Example: “Disease”**
- Chemical reaction-diffusion simulation
- Generate dynamic normal map from the result
- Add creepy effects to characters!
- Available at GPGPU.org

**Example: Fluid Simulation**
- Navier-Stokes fluid simulation on the GPU
- Based on Stam’s “Stable Fluids”
- Vorticity Confinement step
  
```c
  [Fedkiw et al., 2001]
```
- Interior obstacles
  - With zero branches!
- Available at GPGPU.org

“Fast Fluid Dynamics Simulation on the GPU”, Mark Harris. In GPU Gems

**Per-Fragment Flow Control**
- No true branching on GeForce FX
  - Simulated with conditional writes: every instruction is executed, even in branches not taken
- GeForce 6 Series has SIMD branching
- Deep pipelines make branch incoherence expensive
- All pixels “in flight” wait for longest branch
  - Good to use when large blocks of pixels will take the same branch
  - Not good with noise!
**Fragment Flow Control Techniques**

- Try to move decisions up the pipeline
- Replace with math
- Occlusion Query
- Domain decomposition
- Z-cull
- Pre-computation

**Branching with Occlusion Query**

- OQ counts the number of fragments written
- Use it for iteration termination
  ```
  Do { // outer loop on CPU
     BeginOcclusionQuery { // Render with fragment program that
       // discards fragments that satisfy
       // termination criteria
     } EndQuery
  } While query returns > 0
  ```
- Can be used for subdivision techniques

**Domain Decomposition**

- Avoid branches where outcome is fixed
  - One region is always true, another false
  - Separate Fragment Programs for each region
    - No branches!
- Example: boundaries

**Z-Cull**

- In early pass, modify depth buffer
  - Write depth=0 for pixels that should not be modified by later passes
  - Write depth=1 for rest
- Subsequent passes
  - Enable depth test (GL_LESS)
  - Draw full-screen quad at z=0.5
  - Only pixels with previous depth=1 will be processed
- Available on GeForce 6 Series
  - Shader depth replace disables Z-Cull on NV3X

**Pre-computation**

- Pre-compute anything that will not change every iteration!
- Example: arbitrary boundaries
  - When user draws boundaries, compute texture containing boundary info for cells
  - e.g. Offsets for applying PDE boundary conditions
  - Reuse that texture until boundaries modified
  - GeForce 6 Series: combine with Z-cull for higher performance!

**Current GPGPU Limitations**

- Programming is difficult
  - Limited memory interface
  - Usually “invert” algorithms (Scatter → Gather)
  - Not to mention that you have to use a graphics API...
- Limited bandwidth from GPU to CPU
  - PCI-Express will help
  - GeForce 6 Quadro boards will support 1
  - Frame buffer read can cause pipeline flush
  - Avoid frequent communication to CPU
Brook for GPUs

- A step in the right direction
- Moving away from graphics APIs
- Stream programming model
- Enforce data parallel computing: streams
- Encourage arithmetic intensity: kernels
- C with stream extensions
- Cross compiler compiles to HLSL and Cg
- GPU becomes a streaming coprocessor
See SIGGRAPH 2004 Paper and
- http://graphics.stanford.edu/projects/brook
- http://www.sourceforge.net/projects/brook

New Functionality Overview

- Vertex Programs
  - Vertex Textures: gather
  - MIMD processing: full-speed branching
- Fragment Programs
  - Looping, branching, subroutines, indexed input arrays, explicit texture LOD, facing register
- Multiple Render Targets
  - More outputs from a single shader
- Fewer passes, side effects
  - “Deferred Computation”

New Functionality Overview

- VBO / PBO & Superbuffers
  - Feedback texture to vertex input
  - Render simulation output as geometry
  - Not as flexible as vertex textures
    - No random access, no filtering
- Demos
- PCI-Express
  - Higher GPU ↔ CPU bandwidth

Example: Cloth Simulation

- Cloth Simulation
  - Demo by Simon Green
  - Simulation in fragment program
  - Use PBO/VBO to cast texture as vertex array for rendering

Example: Particle Simulation

- 1 Million Particles at 30 Hz
  - Demo by Simon Green
Example: OQ-based subdivision

Example: GPU Radiosity

The Future

More Information

- Increasing flexibility
- Always adding new features
- Improved vertex, fragment languages
- Easier programming
- Non-graphics APIs and languages?
  - Brook for GPUs
    - http://graphics.stanford.edu/projects/brookgpu

- Increasing performance
  - More vertex & fragment processors
  - GFLOPs, GFLOPs, GFLOPs!
  - Fast approaching TFLOPs!
  - Supercomputer on a chip
  - Start planning ways to use it!

- Massive multi-GPU Supercomputers /Clusters?
  - Very low cost per GFLOP.
  - Today: 40 GFLOPs coprocessor for $500

- GPGPU news, research links and forums
  - www.GPGPU.org
- Questions?
  - mharris@nvidia.com