General-Purpose Computation on Graphics Hardware
Welcome & Overview

David Luebke
NVIDIA
Introduction

• The GPU on commodity video cards has evolved into an extremely flexible and powerful processor
  - Programmability
  - Precision
  - Power

• This tutorial will address how to harness that power for general-purpose computation
Motivation: Computational Power

• GPUs are fast...
  - 3.0 GHz Intel Core2 Duo (Woodcrest Xeon 5160):
    • Computation: 48 GFLOPS peak
    • Memory bandwidth: 21 GB/s peak
    • Price: $874 (chip)
  - NVIDIA GeForce 8800 GTX:
    • Computation: 330 GFLOPS observed
    • Memory bandwidth: 55.2 GB/s observed
    • Price: $599 (board)

• GPUs are getting faster, faster
  - CPUs: 1.4× annual growth
  - GPUs: 1.7×(pixels) to 2.3× (vertices) annual growth

Courtesy Kurt Akeley, Stanford GPUbench project
Motivation: Computational Power

Courtesy John Owens, Mike Houston
An Aside: Computational Power

Why are GPUs getting faster so fast?

- Arithmetic intensity
  - The specialized nature of GPUs makes it easier to use additional transistors for computation

- Economics
  - Multi-billion dollar video game market is a pressure cooker that drives innovation to exploit this property
Motivation: Flexible and Precise

• Modern GPUs are deeply programmable
  - Programmable pixel, vertex, and geometry engines
  - Solid high-level language support

• Modern GPUs support “real” precision
  - 32 bit floating point throughout the pipeline
    • High enough for many (not all) applications
    • Both vendors committed to double precision soon
  - DX10-class GPUs add 32-bit integers
Motivation: The Potential of GPGPU

• In short:
  - The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
  - Example applications range from in-game physics simulation to conventional computational science
  - Goal: make the inexpensive power of the GPU available as a computational coprocessor
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 data parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret

• Can’t simply “port” CPU code!
  - Good news: it’s getting better (CTM, CUDA)
Course goals

• A detailed introduction to general-purpose computing on graphics hardware

• We emphasize:
  - Core computational building blocks
  - Strategies, tools, and analysis for programming GPUs
  - Tips & tricks, perils & pitfalls of GPU programming

• Case studies to bring it all together
Course Prerequisites

• Tutorial intended to be accessible to any savvy computer scientist

• Helpful but not required: familiarity with
  - Interactive 3D graphics APIs and graphics hardware
  - Data-parallel algorithms and programming

• Target audience
  - HPC researchers interested in GPGPU research
  - HPC developers interested in incorporating GPGPU techniques into their work
  - Attendees wishing a survey of this exciting field
Speakers

- David Luebke, NVIDIA
- Mark Harris, NVIDIA
- John Owens, University of California Davis
- Naga Govindaraju, Microsoft Research
- Aaron Lefohn, Neoptica
- Mike Houston, Stanford
- Mark Segal, ATI
- Ian Buck, NVIDIA
- Matt Papakipos, PeakStream
Schedule

8:30  Introduction
     **Tutorial overview, GPU architecture, GPGPU programming**

**GPU Building Blocks**

9:10  Data-Parallel Algorithms
     **Reduce, scan, scatter/gather, sort, and search**

9:30  Memory Models
     **GPU memory resources, CPU & Cell**

9:45  Data Structures
     **Static & dynamically updated data structures**

10:00 Break
Schedule

10:30  Sorting & Data Queries  Govindaraju
      Sorting networks & specializations, searching, data mining

11:00  Mathematical Primitives  Lefohn
       Linear algebra, finite difference & finite element methods

Languages & Programming Environments

11:30  GPGPU Languages  Houston
       Brook, RapidMind, Accelerator

12:00  Lunch
Schedule

1:30   Direct GPU Computing: CTM
       *CTM, Data Parallel Virtual Machine*
       Segal

1:45   Direct GPU Computing: CUDA
       *GeForce 8800, Compute Unified Driver Architecture*
       Buck

**High Performance GPGPU**

2:00   GPGPU Strategies & Tricks
       *GPU performance guidelines, scatter, conditionals*
       Owens

2:30   Performance Analysis & Arch Insights
       *GPUBench, architectural models for programming*
       Houston

3:00   Break
Schedule

**GPGPU In Practice**

3:30  Havok FX  
*Game Physics Simulation on GPUs*  
Harris

3:55  PeakStream Platform  
*Commercial GPGPU platform, HPC case studies*  
Papakipos

4:20  GPGPU Cluster Computing  
*Building GPU clusters; HMMer, GROMACS, Folding@Home*  
Houston

**Conclusion**

4:45  Question-and-answer session  
All

5:00  Wrap!
• A simplified traditional graphics pipeline
  - It’s actually highly parallel
  - Note that pipe widths vary
  - Many caches, FIFOs, and so on not shown
**GPU Fundamentals: The Recent Graphics Pipeline**

- Programmable vertex processor
- Programmable pixel processor
GPU Fundamentals: The *New* Graphics Pipeline

- Programmable primitive generation
- More flexible memory access
- And much, much more
GPU Pipeline: Transform

- Vertex processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting
GPU Pipeline: Rasterize

- **Primitive generation**
  - Convert vertices into primitives with area
    - Triangles, quadrilaterals, points

- **Rasterization**
  - Convert geometric primitives to image primitives
    - Pixels, more generally called *fragments* (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)
Introduction to GPGPU Programming

David Luebke
NVIDIA
Outline

• Data Parallelism and Stream Processing
• Computational Resources Inventory
• CPU-GPU Analogies
• Example: N-body gravitational simulation
The Importance of Data Parallelism

• GPUs are designed for graphics

• Graphics processes independent vertices & pixels
  - 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
    • For example, GeForce 8800 GTX has 128 scalar ALUs
  - Hide memory latency (with more computation)
Arithmetic Intensity

• Arithmetic intensity
  - ops per word transferred
  - Computation / bandwidth

• Best to have *high* arithmetic intensity

• Ideal GPGPU apps have
  - Large data sets
    • Amenable to streaming memory access
  - Lots of parallelism
  - High independence between data elements
Stream Processing

• 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
  - Physically-based 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
Computational Resources Inventory

- **Programmable parallel processors**
  - Vertex and fragment shader processors
  - Or unified design (ATI Xenos, NVIDIA GeForce 8800)

- **Rasterizer**
  - Mostly useful for interpolating addresses (texture coordinates) and constants

- **Texture unit**
  - Read-only memory interface
  - Optimized for coherent 2D access, rotation-invariant

- **Render to texture**
  - Write-only memory interface
  - No scatter
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
  - Vertex texture fetch
    - Random access memory for vertices
    - Arguably still not gather
Fragment Processor

- Fully programmable (SIMD)
- Processes 4-component vectors (RGBA / XYZW)
  - Caveat: GeForce 8800 is scalar instead
- 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)

- More on scatter/gather later...
CPU-GPU Analogies

- CPU programming is 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>Stream / Data Array</td>
<td>Texture</td>
</tr>
<tr>
<td>Memory Read</td>
<td>Texture Sample</td>
</tr>
</tbody>
</table>
Kernels

Kernel / loop body / algorithm step  =  Fragment Program
Feedback

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

Grid[i][j] = x;

Array Write  =  Render to Texture
GPU Simulation Overview

• Analogies lead to implementation
  - Algorithm steps are fragment programs
    • Computational *kernels*
  - Current state is 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

• **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)
  
  ```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(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

- GeForce 8800 GTX:
  - 16-bit floating point:
    - 73 fps, 122.5 GFLOPs sustained
  - 32-bit floating point:
    - 39 fps, 65.4 GFLOPS sustained
  - Teaser: 140 GFLOPS sustained (fp32, CUDA - preliminary!)

Nyland, Harris, Prins, GP² 2004 poster
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 2048 bodies makes it trickier
      - Limited by max texture size...
      - “exercise for the reader”
Computing Gravitational Forces

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

\[ r(i,j) = |\text{pos}(i) - \text{pos}(j)| \]

Force is proportional to the inverse square of the distance between bodies.
Computing Gravitational Forces

\[ F(i,j) = \frac{g M_i M_j}{r(i,j)^2}, \]
\[ r(i,j) = |\text{pos}(i) - \text{pos}(j)| \]

Coordinates \((i,j)\) in force texture used to find bodies \(i\) and \(j\) in body position texture.
Computing Gravitational Forces

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
Computing Total Force

- Have: array of (i,j) forces
- Need: total force on each particle i
  - Sum of each column of the force array
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*
Update Positions and Velocities

• Now we have a 1-D array of total forces
  - One per body

• Update Velocity
  - \( u(i, t+dt) = u(i, t) + F_{total}(i) * dt \)
  - Simple fragment shader reads previous velocity and force textures, creates new velocity texture

• Update Position
  - \( x(i, t+dt) = x(i, t) + u(i, t) * dt \)
  - Simple fragment shader reads previous position and velocity textures, creates new position texture
Summary

• Presented mappings of basic computational concepts to GPUs
  - Basic concepts and terminology
  - For introductory “Hello GPGPU” sample code, see http://www.gpgpu.org/developer

• Only the beginning:
  - Rest of course presents advanced techniques, strategies, and specific algorithms.