GPU for Game Computing

Avi Bleiweiss
Overview

• GPU transition
  • CISC to RISC

• Dynamic architecture
  • Compute, bandwidth

• Data parallel
  • Often algorithm redesign

• Game computing
  • Simulation, AI, Audio
Outline

- GPU, processor array
- Compute abstraction
- Physics simulation
- Mesh video mapping
- Results, summary

  - FYSI, physics simulation case study
Outline

- GPU, processor array
- Compute abstraction
- Physics simulation
- Mesh video mapping
- Results, summary
- Deeply programmable
  - Vertex, geometry, pixel
- Precise and flexible
  - 32 bit IEEE float
- Unified shader model
- Compute pipeline
  - Simplified modality, API
• Raw compute power
  • 48 pixel engines, each
  • Executes 4 float mad per cycle
  • @650 MHz -> 250 GFlops

• Shader model 3.0
  • Dynamic branching

• Shader model 4.0
  • Integer arithmetic
Bandwidth

- Memory
  - 256 bits wide, double edge
  - @775 MHz -> **50 GBytes/sec**
  - 0.5 GByte video memory
- I/O, PCI Express x16
  - 4 GBytes/sec
  - Multi-GPU shared resources
- Relatively small caches
Data Concurrency

- Grid formation
  - ALU heavy architecture
- Stream data memory system
- Many in flight threads
  - Hide memory latency
- No static data
- No read-modify-write buffers
Scatter/Gather

- Scatter
  - Indirect memory write \((data[i] = x)\)
  - Populate data structures
  - Usually performed on CPU

- Gather
  - Indirect memory read \((x = data[i])\)
  - Access of data structures
  - Maps onto texture fetch
Grid Framework

- Input, multiple arrays
  - Different resolutions
- Grid computation
  - No inter-cell dependencies
- Kernels
  - Applied to each grid element
  - High arithmetic intensity
- Output, multiple arrays
  - Must have same resolution
Cost/Performance

- Porting to GPU non-trivial
  - Significant speedup payoff
- Efficient data structures
  - In video memory, sliver
- GPU for computing cheap
  - None of display, texture filtering
- Multi-GPU
  - Load partition, shared resources

3x3 Grid

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 active elements
2 sliver elements
Limitations

• No double precision float
  • Single float good enough
• No pixel scatter
  • Self-modifying location expensive
• Dedicated branch units
  • Both sides executed
• Grid outputs limited
  • 4/8 in DirectX 9/10, respectively
Outline

- GPU, processor array
- Compute abstraction
- Physics simulation
- Mesh video mapping
- Results, summary
Graphics API

- Complex, non-intuitive
  - for general computation
- Quest for RISC API
  - Reduced interface set core
- Full screen quad
- RISC API for rendering
  - Ray tracing
Goals

- Hide graphics API
  - Underlying DirectX 9 & 10
- Evolving hardware seamless
- Scalable
  - Multi GPU support
- Automatic multi passing
  - Overcome GPU limitations
Compute API

- Opaque device pointer(s)
  - Single, multi-GPU
- Interface objects

<table>
<thead>
<tr>
<th>Abstract Object</th>
<th>GPU Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Texture (2D/3D)</td>
</tr>
<tr>
<td>Kernel</td>
<td>Shader</td>
</tr>
<tr>
<td>Scratch</td>
<td>Target (2D/3D)</td>
</tr>
</tbody>
</table>

- Object actions
  - Create, assign, remove
Data types
  - Scalars, vectors, matrices

Load, store
  - Resource, scratch, respectively

Kernel parameters
  - One time compile overhead

Compute invocation
  - Across grid cells

Debug, scratch dump
• A single format
  • For both resource, scratch
• Four float components
• Three component vector
  • Alpha channel for control
• Three/four squared matrix
  • Three dimensional array

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>slice #0</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>slice #1</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>slice #2</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>
• Iterative
  • Recirculate scratches
• Parallel setup paths
  • Resources, kernels
• Implicit correlation
  • Scratches, kernel
• Multi pass compute
  • Single kernel execution
• Results CPU access
Grid parallel traversal
- Tiled pattern, ownership
- Point sampled grid cells
  - VPOS for cell index
Relative resource address
- Multi resolution resources
Dependent data access
- Multi pass reduction
Multi GPU

- Compute bound process
- Asymmetric processing
- Computation distribution
  - Simple grid subdivision
- Synchronization overhead
  - PCI Express contention
- Shared resources
Outline

• GPU, processor array
• Compute abstraction
• Physics simulation
• Mesh video mapping
• Results, summary
• Game engine
  • Simulation, visual rendering

• Physics scope
  • Rigid, deformable bodies

• Physics platforms
  • Multi core CPU, PPU, GPU

• GPU assisted physics
  • Game play, effects
FYSI

- Physics scene description format
  - *FYSL*, GPU oriented
- Consistent I/O abstraction
  - Simulation input and results
- Stream API
  - Asynchronous, discrete simulation
- GPU, CPU implementation
  - Performance analysis
FYSI (cnt’d)

- Iterative *physics pipeline*
  - *System Setup, Solver, Collision*
- Physics to rendering interface
  - Position, transform update
- Compute Abstraction Layer
  - On top of DirectX API
// instantiate Fysi, attach a Fysi done callback
IFysi* m_ifysi = new IFysi();
FysiUpdate* m_update = new FysiUpdate();
m_ifysi->attach(m_update);

// initialize methods
m_ifysi->init(IFysi::File, IFysi::DX9, IFysi::File);
m_ifysi->setProcessor(IFysi::GPU);
m_ifysi->setFlag(IFysi::LastStep, true);

m_ifysi->setDevice(pDevice);

// setup
std::string description = "rigid.xml";
m_ifysi->scene(description.c_str());

// simulate (fork as working thread)
Int numSteps = 10;
m_ifysi->simulate(numSteps);

// working thread loop/sleep
while(!updated) ...;

if(!m_update->getStatus()) std::cerr << m_fysi->getError();
else {
    for(int i = 0; i < numSteps; i++) {
        std::cout << m_fysi->getResults(i);
    }
}
• Simulation control
  • Type, time step
• Actors, hierarchical
  • Bounding volumes, meshes
  • Linear, angular motion
• Joints, motion constraints
  • Spring, distance
• Feedback
  • Path decision
<Actor>
  <Name>a0</Name>
  <Type>static</Type>
  <Id>0</Id>
  <Shape>
    <Box>
      <Position>-1. 0 -1.</Position>
      <Radius>2. 5 -1.5</Radius>
    </Box>
  </Shape>
  <Dynamics>
    <Linear>
      <Mass>10</Mass>
      <Velocity>10. 0 0</Velocity>
      <Force>5 1.2 -.5</Force>
    </Linear>
  </Dynamics>
</Actor>

<Joint>
  <Distance>
    <ActorPair>0 1</ActorPair>
    <Min>.1</Min>
    <Max>1.</Max>
  </Distance>
</Joint>

<Joint>
  <Spring>
    <ActorPair>1 2</ActorPair>
    <RestLength>2.3</RestLength>
    <Stiffness>1.5</Stiffness>
    <Damping>3.2</Damping>
  </Spring>
</Joint>

<Actor definition> Actor definition </Actor definition>

<Joints definition> Joints definition </Joints definition>
Physics Pipeline

- **initial scene (FYSL)**
  - **System Setup**
    - current state resolution
  - **Solver**
    - next state resolution
  - **Collision**
    - detection and response

- **results (FYSL)**

flowchart: per step
• Grid based simulation
  • Euler method \( (x(t + dt) = x(t) + t \times \frac{dx}{dt};) \)
• Geometry, property resources
  • Texture array
• Portable shading library
• Resume from last step
  • Result caching
• Adaptive time step
GPU Dynamics

Future Workflow within Maya Environment

Scene Creation

Create Geometry Shading Lighting
Create Dynamics

Switch

Conform to Ultra Real ATI Maya GPU Dynamics Scene

ATI - GPU Maya Dynamics Simulator

High Level of Scene complexity
Interactive GPU based Hardware Simulated Dynamics Scene Playback

CPU based Maya Dynamics Solver

Current Workflow path in Maya

Edit Geometry Shading Lighting
Edit Dynamics

User Review
Outline

- GPU, processor array
- Compute abstraction
- Physics simulation
- Mesh video mapping
- Results, summary
Geometry

• Object space representation
  • Precise, CPU equivalent
• Geometry in video memory
  • Multiple arbitrary meshes
• GPU texture addressability
  • 4K by 4K for 2D
• Mesh representation
  • 1D vertex and index buffers
• Vertex and index buffers
  • Positions and faces, respectively
• Counter clockwise triangles

```
<table>
<thead>
<tr>
<th>vertex buffer</th>
<th>index buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>position # 0</td>
<td>face # 0 (0, 1, 2)</td>
</tr>
<tr>
<td>position # 1</td>
<td>face # 1 (2, 1, 3)</td>
</tr>
<tr>
<td>position # 2</td>
<td>face # 2 (1, 4, 3)</td>
</tr>
<tr>
<td>position # 3</td>
<td>face # 3 (3, 4, 5)</td>
</tr>
<tr>
<td>position # 4</td>
<td>face # 4 (5, 4, 6)</td>
</tr>
<tr>
<td>position # 5</td>
<td></td>
</tr>
<tr>
<td>position # 6</td>
<td></td>
</tr>
</tbody>
</table>
```
Mesh as a resource
- 2D video memory array

Three floats element
- \{x,y,z\} and \{i0,i1,i2\}

Arbitrary dimensions
- Padded as necessary

Dependent texture
- For each triangle access

vertex buffer (3x3)  index buffer (3x2)
Compositing

- Many mesh scene
- 16 textures constraint
- A single super mesh
  - Coalesces multiple meshes
  - Vertex and index uber buffer
- Index buffer offset
- Boundary sub mesh id
Traversal

- Low creation overhead
- Sub mesh collision
  - Tri-tri intersection, contact
- Avoids self intersection
- Multi pass compute
  - Intersect, response, integrate
- Dependent texture access
  - Face to vertex fetch
Outline

• GPU, processor array
• Compute abstraction
• Physics simulation
• Mesh video mapping
• Results, summary
Goals

- GPU superior to CPU
  - Collision detection, response

- Understand compute behavior
  - Grid distribution
  - Flow control, multi pass

- Scalability across GPUs
  - More pipes, ALUs
Methodology

- Single CPU
  - No dual core, no SSE2
- System performance
  - Physics process, no rendering
- Shape based benchmarks
  - Linear motion
- Absolute, normalized results
## Processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pipes</th>
<th>ALUs</th>
<th>Core (MHz)</th>
<th>Memory (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>1</td>
<td>1(4)</td>
<td>3400</td>
<td>533</td>
</tr>
<tr>
<td>Radeon X1600</td>
<td>4</td>
<td>3</td>
<td>600</td>
<td>700</td>
</tr>
<tr>
<td>Radeon X1800</td>
<td>16</td>
<td>1</td>
<td>625</td>
<td>750</td>
</tr>
<tr>
<td>Radeon X1900</td>
<td>16</td>
<td>3</td>
<td>650</td>
<td>775</td>
</tr>
<tr>
<td>GeForce 7800</td>
<td>24</td>
<td>2</td>
<td>580</td>
<td>865</td>
</tr>
<tr>
<td>GeForce 7900</td>
<td>24</td>
<td>2</td>
<td>650</td>
<td>800</td>
</tr>
</tbody>
</table>
## Arithmetic Intensity

<table>
<thead>
<tr>
<th>Shader Type</th>
<th>Ops</th>
<th>ALU</th>
<th>Texture</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume-volume</td>
<td>119</td>
<td>87</td>
<td>5</td>
<td>17.40</td>
</tr>
<tr>
<td>mesh-mesh</td>
<td>300</td>
<td>241</td>
<td>8</td>
<td>30.13</td>
</tr>
<tr>
<td>mesh update</td>
<td>67</td>
<td>53</td>
<td>7</td>
<td>7.57</td>
</tr>
<tr>
<td>impact</td>
<td>102</td>
<td>72</td>
<td>19</td>
<td>3.79</td>
</tr>
</tbody>
</table>
## GPU Scalability

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU</th>
<th>X1600</th>
<th>X1800</th>
<th>X1900</th>
<th>7800</th>
<th>7900</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabb256x256</td>
<td>101404</td>
<td>47 (1.000)</td>
<td>31 (1.516)</td>
<td>15 (3.133)</td>
<td>203 (0.231)</td>
<td>220 (0.213)</td>
</tr>
<tr>
<td>sphere256x256</td>
<td>74793</td>
<td>31 (1.000)</td>
<td>15 (2.067)</td>
<td>15 (2.067)</td>
<td>188 (0.164)</td>
<td>201 (0.154)</td>
</tr>
<tr>
<td>mesh64x64</td>
<td>197508</td>
<td>32 (1.000)</td>
<td>16 (2.000)</td>
<td>30 (1.066)</td>
<td>187 (0.171)</td>
<td>247 (0.129)</td>
</tr>
<tr>
<td></td>
<td>(62471 tris)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tetrahedron128x128</td>
<td>562768</td>
<td>125 (1.000)</td>
<td>110 (1.136)</td>
<td>47 (2.659)</td>
<td>703 (0.177)</td>
<td>482 (0.259)</td>
</tr>
</tbody>
</table>

Compute draw calls (figures in msec)
Observations

• CPU wins for small grids
  • Expected, setup overhead

• GPU faster elsewhere
  • Up to an order of magnitude

• GPU Scalability
  • Flow control might be stalling

• GeForce 7800/7900 slower
  • Across benchmarks
• Unified shader architecture

• Higher concurrency level
  • 64 pixel engines
  • Flexible mix of scalar/vector

• DirectX 10
  • Constant buffers
  • Texture array, indexing
  • Non-power-of-2 3D textures
  • 3D render target
Summary

- GPU game computing
  - Still a challenge
- Parallel programming
  - Macro and micro level
- Software productivity
  - Tools, tools, tools…
- Emerging CPU/GPU platforms
  - Adaptive load balance
Thank You!

Questions?