Visual Effects Production on the Cell/BE

Andrew Clinton, Side Effects Software
Visual Effects Production

1. Animation – Character, Keyframing
2. Dynamics Simulation – Fluids, RBD, Cloth
3. Rendering – Raytracing
4. Compositing
Software Products

- Houdini: 3D Visual Effects production software supporting all stages of content creation
- Houdini Batch: Command-line interface to Houdini with python scripting
- Mantra: Bundled production rendering application
Render Farms

- Final software renders of visual effects shots are often distributed to a network of servers for processing
  - Individual frames can often require hours of computation
- Complex simulations or compositing operations also use the render farm
  - Simulation is more difficult to parallelize due to inter-frame dependencies
Cell/BE Based Render Farm

- In co-operation with Sony, we have ported some of the most computationally expensive portions of our software to the Cell/BE
  - Fluid Simulation
  - Rendering
  - Compositing
Project Goals

- Houdini Batch / Mantra to run under Linux on the Cell/BE
- Apply SPU and GPU-based optimizations to accelerate bottleneck algorithms
- Produce the same output for rendering/compositing as other supported platforms (Linux/Windows x86)
  - Simulation results are sensitive to floating point roundoff, so identical results are not expected between platforms
Development Approach

- **Top-down**
  - profile common customer scenes to determine the most important optimizations
  - Simplify data structures
  - Optimize these algorithms first

- **Bottom-up**
  - development of a robust, abstract library of data structures to simplify SPU programming

- **Tools:** C++, GCC, Sony PASuite, oprofile
Target Algorithms

- Rendering (SPU)
  - Raytracing – shadows, reflections, refractions
  - Ambient Occlusion
  - Global Illumination
  - Micropolygon Rendering – motion blur, depth of field
- Fluid Simulation (SPU)
  - Voxel operations (Advect, Diffuse, etc.)
  - Sparse matrix solver
- Compositing (SPU/GPU)
  - Blur, Defocus, Over, etc. - SPU
  - Noise, Lighting, Fog, etc. - GPU
Raytracing
Raytracing Algorithm

- Build KD-Tree
- Generate rays
- Calculate intersections
  - Performed on SPU
Raytracing on the SPU

**PPU**
- Geometry
- KD-Tree

**Local Store**
- Software Managed Cache (112Kb)
- Double Buffers (10Kb)
- Code (128Kb)
## Raytracing Benchmark (million rays/second)

### Incoherent Rays

<table>
<thead>
<tr>
<th></th>
<th>Box (Hit%: 0.94)</th>
<th>Godzilla (Hit%: 0.28)</th>
<th>Happy Buddha (Hit%: 0.87)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core 2 2.66 GHz (1 processor)</td>
<td>1.68</td>
<td>0.74</td>
<td>0.20</td>
</tr>
<tr>
<td>Cell/BE PPU 64-bit</td>
<td>0.19</td>
<td>0.16</td>
<td>0.05</td>
</tr>
<tr>
<td>Cell/BE 8 SPEs</td>
<td>14.45</td>
<td>1.95</td>
<td>0.64</td>
</tr>
</tbody>
</table>

### Coherent Rays

<table>
<thead>
<tr>
<th></th>
<th>Box (Hit%: 0.23)</th>
<th>Godzilla (Hit%: 0.25)</th>
<th>Happy Buddha (Hit%: 0.21)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core 2 2.66 GHz (1 processor)</td>
<td>4.23</td>
<td>1.67</td>
<td>1.81</td>
</tr>
<tr>
<td>Cell/BE PPU 64-bit</td>
<td>0.51</td>
<td>0.27</td>
<td>0.31</td>
</tr>
<tr>
<td>Cell/BE 8 SPEs</td>
<td>27.02</td>
<td>5.90</td>
<td>6.05</td>
</tr>
</tbody>
</table>
Ambient Occlusion
 Ambient Occlusion

- Offload ray generation in addition to ray intersection algorithm to SPU
- Render Time comparison (256 samples/pixel):
  - 2:35 (Intel quad core2, 4 threads)
  - 5:16 (Cell/BE, 8 SPUs)
- Why slower?
  - Divergent rays and heavy geometry – non-ideal SIMD efficiency and software managed cache overhead
  - KD Tree construction performed on PPU
Micropolygon Sampling
Micropolygon Sampling

- Specialization of raytracing for eye rays
- Supports fast motion blur, depth of field
- Too much code to fit into the same spu program as the raytracer
  - Use overlays! Overlay switch time close to 4us
    - 100 times faster than using libspe2 to switch programs
Micropolygon Sampling Results

- Micropolygon sampling performance is considerably better on the SPUs as opposed to the PPU
- However, SPU sampling is not yet competitive with Core2 processor performance
  - In-memory representation using linked sample lists not easily ported to SPU
  - Render order dependencies limit the amount of data available to send to SPUs
  - Need to consider data structure redesign to take full advantage of SPU performance
Fluid Simulation
Fluid Simulation

- Primarily 2 classes of bottleneck algorithms
  - Voxel processing (eg. advection, diffusion, etc)
  - Sparse matrix solver (project)
- Each simulation frame executes a sequence of operations applied to a varying number of scalar and vector fields
Voxel Operations

- 1Kb per voxel tile plane (16 2D slices per tile)
  - Double buffer streaming inputs/outputs
  - Use software managed cache for random access with 1 tile plane per page
- 3D field = 3 scalar fields
Advection

- Use a velocity field to transform a scalar field (e.g., density) through time
  - 4 random access inputs (velocity x, y, z; scalar field)
  - 1 streaming output
Sparse Matrix Solver

- Uses Jacobi iterations
- Within each iteration, compute:
  - Sparse matrix multiply
  - Vector scale/offset
  - L2 norm calculation
- Normally vector sizes are very large (for a $128^3$ simulation, about 1.5 million entries)
  - Use double buffering to stream inputs and outputs to memory
Fluid Simulation Results

<table>
<thead>
<tr>
<th>Processor / Sim Stage</th>
<th>CPU Core 2 2.66 GHz (4 processors)</th>
<th>Cell/BE 8 SPEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project (Sparse Solver)</td>
<td>18.17</td>
<td>25.38</td>
</tr>
<tr>
<td>Diffuse</td>
<td>3.59</td>
<td>2.42</td>
</tr>
<tr>
<td>Buoyancy</td>
<td>0.29</td>
<td>0.06</td>
</tr>
<tr>
<td>Advect</td>
<td>1.64</td>
<td>1.77</td>
</tr>
<tr>
<td>Enforce</td>
<td>0.42</td>
<td>0.20</td>
</tr>
<tr>
<td>Total</td>
<td>24.11</td>
<td>29.83</td>
</tr>
</tbody>
</table>

- Results are for a single simulation step of a $128^3$ simulation
- Values in seconds
## Floating Point Evaluation (no dma)

<table>
<thead>
<tr>
<th>Algorithm / Processor</th>
<th>Quad Intersection</th>
<th>GAS Diffuse</th>
<th>VM Multiply</th>
<th>SPU Ideal (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell/BE SPU</td>
<td>0.52s</td>
<td>0.43s</td>
<td>0.31s (3.2 GFLOPS)</td>
<td>1.36s (23.5 GFLOPS)</td>
</tr>
<tr>
<td>Cell/BE PPU</td>
<td>2.38s</td>
<td>N/A</td>
<td>1.67s (0.6 GFLOPS)</td>
<td>19.3s (1.7 GFLOPS)</td>
</tr>
<tr>
<td>Intel Core2</td>
<td>0.45s</td>
<td>0.43s</td>
<td>0.26s (3.8 GFLOPS)</td>
<td>1.92s (16.7 GFLOPS)</td>
</tr>
<tr>
<td>Speedup (*)</td>
<td>1.73x</td>
<td>2.00x</td>
<td>1.61x</td>
<td>2.82x</td>
</tr>
</tbody>
</table>

### Notes:
- SPU Ideal is a benchmark of the ideal SPU FLOPS (it uses only spu_madd instructions)
- Speedup compares Cell/BE performance (with 8 SPUs) against Intel Core2 performance (with 4 Core2 processors)
  - Assuming linear scaling
Bandwidth Evaluation
(no computation)

- Cell 8 SPUs: 18.0 Gb/s (25.6 Gb/s)
- Cell 1 PPU: 1.5 Gb/s
- Intel Core2 DDR2 800Mhz: 4.4 Gb/s
  - 25.0 Gb/s cached
Basic Performance Assessment

- In ideal scenario, cell processor has peak floating point throughput 2.8x current-generation Intel quad-core cpus.
  - Real-world algorithms produce floating point speedup up to 2.0x
- Ideal speedup can only be observed when:
  - Algorithm is fully vectorized, and has high vector utilization
  - Algorithm is compute-bound or has well-behaved dma access (streaming or prefetched)
Conclusions

- PPU-only port of software to Cell/BE is easy
- SPU port can show competitive performance to Intel 4-core system
  - ambient occlusion
  - fluid simulation
  - compositing blur
- Cell/BE algorithms produce visually identical results to existing x86 platforms
- The Cell/BE can accommodate the flexibility inherent in Houdini and visual effects production in general
Questions?
Appendix A: SPU Programming

- **SPU side:**
  - N-Buffering
  - Software managed cache
  - 4-component vectors
  - Overlays
  - DMA/Heap debugging

- **PPU side:**
  - SPU thread set start/stop
  - Mailbox communication
  - Scheduling, SPU allocation