Game Developer’s Perspective on OpenCL

Eric Schenk, EA
August 2009
Motivation
Motivation

• Supports a variety of compute resources
  - CPUs, GPUs, SPUs, accelerators
  - Unifies programming for devices that have very different environments
  - Provides uniform interface to non-fixed platforms (e.g. PC/Mac)
Motivation

• Open standard
  - Many vendors will support directly
  - Potential for 3rd party implementations if no vendor support
  - Embedded platform support in spec
Motivation

• Concurrent programming model
  - Command queue(s) per device with support for dependencies across queues
  - Data parallel and SIMD support in a portable notation,
  - superior to intrinsics
  - Low-level programming model, minimalist abstractions
How to apply OpenCL to games
How to apply OpenCL to games

• Don’t try speed up everything
  - Create new content/features that use the excess compute capacity

• Some cases are easy
  - We already parallelize them

• Some cases are harder
  - But with OpenCL it will be easier to try them.
Dodging Amdahl’s Law

• Amdahl’s Law
  - “The speedup of a program using multiple processors is limited by the sequential fraction of the program”

• So, massively parallel hardware has limited benefit?
  • No
    - The game already runs on today’s hardware.
    - More compute power => add more content/features.
    - Games have 100’s or 1000’s of algorithms to target.

• OpenCL makes the coding easier
The “Easy” cases

- **Rendering**
  - Rasterization / shading already runs on graphics hardware
  - Visibility culling
  - Procedural geometry

- **Codec decompression**
  - Animation, audio, video, etc.

- **Animation blending**

- **Audio mixing**

- **Rigid body physics integration**

- **Some collision calculations**

- **Etc.**
Then it gets harder...

• AI
  - Path finding
  - Search
  - Pattern matching
  - Fuzzy logic / neural nets for AI
  - Massive scale AI behavior on multiple actors
  - Speculative AI paths

• Animation
  - High level motion planning
  - Detailed crowd simulation with individual behaviors

• Physics
  - Broad phase collision systems
  - Character to character interaction
  - High quality liquid/smoke.

• Asset creation
  - Landscape, vegetation, architecture, etc.
Early Experiences
Early Experiences

• Too early to tell you about shipped game code
• We have done some feasibility experiments

• Skate 2 Cloth Experiment
  - Pulls the character skinning and cloth physics out of EA’s Skate 2 and embeds it into a stand alone demo

• Goal:
  - Exercise OpenCL on “real” code
  - Extract game subsystem and port key algorithms to OpenCL
  - Leave intact as much original code and API as possible
What does it do?

• Play back recorded skeleton poses
• Skin character model to pose
• Apply cloth physics to portion of the skinned model
  - Integrator for gravity
  - Particle-to-particle spring system
  - Constrained to underlying skinned model
• Render via OpenGL
Demo Task Graph

- Simple sequential execution graph
- In-order, data parallel tasks
- Render inputs are double buffered to OpenCL tasks could, in theory, begin prior to render completion
Enqueue Instead of Execute

- In original code each box represents a function call
- Demo abstracts OpenCL kernel invocations as functors
  - Functors take an event parameter to be dependent on, and return their own event
  - Functors enqueue a kernel—upon return the kernel may not have completed or even begun
Enqueue Instead of Execute

• A function like this
  
  ```cpp
this->TCVIntegrate(dt, deltaPos);
  ```

• Became an enqueue functor call like this
  
  ```cpp
if (mIntegratorFunc != NULL)
    event = (*mIntegratorFunc)(event, dt, deltaPos);
  ```

• The original code remains as a fallback, slightly modified
  
  ```cpp
else {
    this->LockBuffers();
    this->TCVIntegrate(dt, deltaPos);
    this->UnlockBuffers();
}
```
Memory Objects

• Original data was organized in aligned arrays of structs

• `cl_mem` objects were created for each array
  - `CL_MEM_USE_HOST_PTR` to avoid additional allocations and copying (when using CPU device)
  - OpenGL vertex buffers allocated by GL and bound to OpenCL via GL/CL interop
Data Flow Through Kernels

Host → V→P → Drivers → Constraints → Driver Constraint → WriteBack → Render

SkinModel → Integrator → Distance Constraint → Particle → P→V

Pose ← Src Verts → Dst Verts

Display
Complete Command Queue

- Buffer write used to move data into pose buffer
- Acquire / release to give OpenCL access to GL buffers
Kernels

• **Skinning**
  - Generates vertex buffer and drivers from pose

• **Integrator**
  - Acceleration due to gravity

• **Distance constraint**
  - Springs between cloth particles

• **Driver constraint**
  - Keeps cloth near underlying ‘skin’

• **Writeback**
  - Output cloth positions to vertex buffer
Vectorization

• Wrote two variants of each kernel: Scalar and vector
  - Some hardware does much better with SIMD code

• “Structure of Arrays” (SoA) style math
  - Memory data layout unchanged, rearranged at load/store
  - AoS float4 contains “xyzw”; SoA float4 contains “xxxx”

• Two key techniques employed: Transpose and select
4x4 Transpose

- Used to convert between AoS and SoA
- Four float4 AoS values loaded into one float16
- Post transpose the four float4 parts are SoA

```
float16 transpose (float16 m) {
    float16 t;
    t.even = m.lo;  t.odd = m.hi;
    m.even = t.lo;  m.odd = t.hi;
    return m;
}
```
Select vs. Branch

- Branching is bad, for many reasons
- `select(a, b, c)` efficiently chooses between “a” and “b” based on “c” element-wise fashion
- Comparisons like `isgreater(a, b)` are set up to output to “c”
Scalar Integrator

```c
void perform_integrator (int pIdx, float4 vsr, float4 acc, float dt, __global Particle* particles, __global short* indices, __global IntegratorState *iState) {
    float4 curPos, curPrevPos, nextPos;
    curPos = particles[pIdx].mPos;
    curPrevPos = particles[pIdx].mPrevPos;
    float vsr = (1.0f - ctp->mVerticalSpeedDampening);

    // TIME CORRECTED VERLET
    // xi+1 = xi + (xi - xi-1)*(dti/dti-1) + a*dti* dti
    if (((indices[pIdx]) >= 0 && (curPrevPos.w > 0.0f)) { 
        nextPos = curPos;
        nextPos -= curPrevPos;
        nextPos *= dt / iState->mLastDT;
        nextPos.y *= vsr;
        nextPos += acc;
        particles[pIdx].mPrevPos = curPos;
        particles[pIdx].mPos = curPos + nextPos;
    }
}
```
Vector Integrator

```c
void perform_vector_integrator (  
    float4 vdt_ratio,  
    float4 acc,  
    int numParticles,  
    int pIdx,  
    __global Particle *ptrParticle,  
    __global uint *mapped)
{
    // load 4 particle positions and previous positions
    // transpose particles from Aos -> SoA
    // extract locked flags for particles
    // TIME CORRECTED VERLET:
    //   xi+1 = xi + (xi - xi-1) * (dti / dti-1)
    //       + a * dti * dti
    // select between unchanged (if locked)
    //   and new location (if not locked)
    // transpose SoA -> AoS
    // store particles back
}
```
Vector Integrator

// load particle position and previous position
float16 curPos, curPrevPos;
curPos.s0123 = ptrParticle[0].mPos;
curPrevPos.s0123 = ptrParticle[0].mPrevPos;
curPos.s4567 = ptrParticle[1].mPos;
curPrevPos.s4567 = ptrParticle[1].mPrevPos;
curPos.s89ab = ptrParticle[2].mPos;
curPrevPos.s89ab = ptrParticle[2].mPrevPos;
curPos.scdef = ptrParticle[3].mPos;
curPrevPos.scdef = ptrParticle[3].mPrevPos;

// transpose particles from Aos -> SoA
curPos = transpose(curPos);
curPrevPos = transpose(curPrevPos);

// extract locked flags for particles
uint4 mask = (uint4)isgreater(curPrevPos.scdef, (float4)0.0f) & ~ComputeMappedMask(mapped);
Vector Integrator

// TIME CORRECTED VERLET
float4 next_x = ((curPos.s0123 - curPrevPos.s0123) * vdt_ratio + (curPos.s0123 + acc.x));
float4 next_y = ((curPos.s4567 - curPrevPos.s4567) * vdt_ratio + (curPos.s4567 + acc.y));
float4 next_z = ((curPos.s89ab - curPrevPos.s89ab) * vdt_ratio + (curPos.s89ab + acc.z));

// select between unchanged (if locked)
// and new location (if not locked)
curPrevPos.s0123 = select(
    curPrevPos.s0123, curPos.s0123, mask);
curPrevPos.s4567 = select(
    curPrevPos.s4567, curPos.s4567, mask);
curPrevPos.s89ab = select(
    curPrevPos.s89ab, curPos.s89ab, mask);
curPos.s0123 = select(curPos.s0123, next_x, mask);
curPos.s4567 = select(curPos.s4567, next_y, mask);
curPos.s89ab = select(curPos.s89ab, next_z, mask);
// transpose SoA -> AoS
curPos = transpose(curPos);
curPrevPos = transpose(curPrevPos);

// store particles back
ptrParticle[0].mPos = curPos.s0123;
    ptrParticle[0].mPrevPos = curPrevPos.s0123;
ptrParticle[1].mPos = curPos.s4567;
    ptrParticle[1].mPrevPos = curPrevPos.s4567;
ptrParticle[2].mPos = curPos.s89ab;
    ptrParticle[2].mPrevPos = curPrevPos.s89ab;
ptrParticle[3].mPos = curPos.scdedf;
    ptrParticle[3].mPrevPos = curPrevPos.scdedf;
Work-Items and Workgroups

- **Data parallel model in OpenCL is based on work-items**
  - Each work-item is given its own index (in up to three dimensions)
  - Work-items are organized into uniformly sized “workgroups”
  - Workgroup size is limited by device and kernel

- **Work-items in a workgroup execute in parallel and share**
  - Local memory
  - Barriers
  - Fences
Per Kernel Index Space

• This demo’s kernels are all in one-dimensional spaces

• Each kernel uses its own space
  - Skinning: Complete vertex array
  - Integrator: Particle array
  - Driver constraint: Driver array
  - Distance constraint: Constraints array
  - Writeback: Clothed portion of vertex array
Distance Constraint: Limited Parallelism

- Each distance constraint modifies two particles
  - Modifying a particle from multiple constraints concurrently gives incorrect results

- Original constraints organized into “octets”
  - Eight-at-once is far too few to keep GPU busy
## CPU Device Performance

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>Scalar Task</th>
<th>Scalar DP (8 Cores)</th>
<th>Vector DP (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overall</strong></td>
<td>1.00</td>
<td>2.72</td>
<td>17.03</td>
<td>17.27</td>
</tr>
<tr>
<td><strong>Skinning</strong></td>
<td>1.00</td>
<td>2.98</td>
<td>20.98</td>
<td>20.98*</td>
</tr>
<tr>
<td><strong>Integrator</strong></td>
<td>1.00</td>
<td>1.53</td>
<td>1.48</td>
<td>1.10</td>
</tr>
<tr>
<td><strong>Distance</strong></td>
<td>1.00</td>
<td>1.34</td>
<td>2.54</td>
<td>2.69</td>
</tr>
<tr>
<td><strong>Driver</strong></td>
<td>1.00</td>
<td>1.14</td>
<td>5.58</td>
<td>8.83</td>
</tr>
<tr>
<td><strong>WriteBack</strong></td>
<td>1.00</td>
<td>1.01</td>
<td>7.26</td>
<td>7.26*</td>
</tr>
</tbody>
</table>

*No vector version available*
## GPU Device Performance

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>Best CPU (8 Cores)</th>
<th>Unoptimized GTX285</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>1.00</td>
<td>17.27</td>
<td>4.11</td>
</tr>
<tr>
<td>Skinning</td>
<td>1.00</td>
<td>20.98</td>
<td>4.39</td>
</tr>
<tr>
<td>Integrator</td>
<td>1.00</td>
<td>1.53</td>
<td>1.10</td>
</tr>
<tr>
<td>Distance</td>
<td>1.00</td>
<td>2.69</td>
<td>0.74</td>
</tr>
<tr>
<td>Driver</td>
<td>1.00</td>
<td>8.83</td>
<td>3.24</td>
</tr>
<tr>
<td>WriteBack</td>
<td>1.00</td>
<td>7.26</td>
<td>17.69</td>
</tr>
</tbody>
</table>

©Copyright Electronic Arts, 2009 - Page 33
Optimizations – Algorithmic

• Each distance constraint modifies two particles
  - Modifying a particle from multiple constraints concurrently gives incorrect results.
  - Original algorithm allows for at most eight particles in a work-group. This cripples GPU performance.

• Reordering constraints to maximize the workgroup size (7x) improved kernel performance dramatically

• Limited by algorithm,
  - Alternatives should be investigated
Optimizations – Work per Task

• Larger simultaneous data set sizes provide the GPU with substantially more work
  - CPU benefits as well, but drops off with larger data sets

• Real game: Process all characters as a single batch

• Demo: Simple geometry replication (25x)
Optimizations – Bandwidth

• Memory access is crucial in bandwidth limited kernels (and most will be bandwidth limited)

• GL/CL integration

• CL_MEM_COPY_HOST_PTR for GPU

• Current GPU memory controllers are optimized for a specific set of memory access patterns
GPU Memory and Execution Optimization

- Burst reads from memory are essential
- Bursts only happen on sequential accesses
- Bursts are created by coalescing smaller reads
- Can only coalesce 4-, 8-, or 16-byte reads
- Coalesces across work-items in a workgroup

- Solution: Optimized transfer from global to local memory, then operate on local memory
- This pattern must currently be explicitly coded for
GPU Memory and Execution Optimization

Buffer of 80-byte vertex structs
(5 x 16-byte float4)
GPU Memory and Execution Optimization

Buffer of 80-byte vertex structs
(5 x 16-byte float4)

Local memory
Kernel barriers
Kernel fences
GPU Memory and Execution Optimization

Buffer of 80-byte vertex structs
(5 x 16-byte float4)

Local memory
Kernel barriers
Kernel fences
GPU Memory and Execution Optimization

Buffer of 80-byte vertex structs
(5 x 16-byte float4)

Local memory
Kernel barriers
Kernel fences

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GPU Memory and Execution Optimization

Buffer of 80-byte vertex structs
(5 x 16-byte float4)

Local memory
Kernel barriers
Kernel fences
## GPU Device Performance

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>Best CPU (8 Cores)</th>
<th>Unoptimized GTX285</th>
<th>Optimized GTX285</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>1</td>
<td>17.27</td>
<td>4.11</td>
<td>45.15</td>
</tr>
<tr>
<td>Skinning</td>
<td>1</td>
<td>20.98</td>
<td>4.39</td>
<td>83.56</td>
</tr>
<tr>
<td>Integrator</td>
<td>1</td>
<td>1.53</td>
<td>1.1</td>
<td>4.6</td>
</tr>
<tr>
<td>Distance</td>
<td>1</td>
<td>2.69</td>
<td>0.74</td>
<td>1.73</td>
</tr>
<tr>
<td>Driver</td>
<td>1</td>
<td>8.83</td>
<td>3.24</td>
<td>14.45</td>
</tr>
<tr>
<td>WriteBack</td>
<td>1</td>
<td>7.26</td>
<td>17.69</td>
<td>56.6</td>
</tr>
</tbody>
</table>
Performance Summary

• Performance tuning is essential

• Performance characteristics are not hidden by abstraction layer

• Expect to write multiple variations and choose empirically at runtime