OpenCL
Cloth Simulation in the Bullet Physics SDK

Lee Howes, AMD
OpenCL for physics

• OpenCL in the Bullet physics SDK
  - An introduction to cloth simulation
  - Some tips for implementation in OpenCL on wide SIMD architectures
  - A demonstration of the current state of development
Cloth simulation

- **Large number of particles**
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles
Cloth simulation

- **Large number of particles**
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles

Original layout

Compute forces as stretch from rest length

Apply position corrections to masses

Rest length of spring

Compute new positions
Cloth simulation steps

- For each simulation iteration:
  - Compute forces in each link based on its length
  - Correct positions of masses/vertices from forces
  - Compute new vertex positions

Original layout

Current layout: Compute forces as stretch from rest length

Apply position corrections to masses

Compute new positions
Cloth simulation steps

- For each simulation iteration:
  - Compute forces in each link based on its length
  - Correct positions of masses/vertices from forces
  - Compute new vertex positions
Cloth simulation steps

For each simulation iteration:
- Compute forces in each link based on its length
- Correct positions of masses/vertices from forces
- Compute new vertex positions
Springs and masses

- Two or three main types of springs
  - Structural/shearing
  - Bending
Springs and masses

• Two or three main types of springs
  - Structural/shearing
  - Bending
CPU approach to simulation

- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times
CPU approach to simulation

• One link at a time
• Perform updates in place
• “Gauss-Seidel” style
• Conserves momentum
• Iterate n times
CPU approach to simulation

- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times
CPU approach to simulation

• One link at a time
• Perform updates in place
• “Gauss-Seidel” style
• Conserves momentum
• Iterate n times
CPU approach to simulation

• One link at a time
• Perform updates in place
• “Gauss-Seidel” style
• Conserves momentum
• Iterate n times
CPU approach to simulation

- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times
Moving to the GPU:
The pixel shader approach

• Offers full parallelism
• One vertex at a time
• No scattered writes
Moving to the GPU: The pixel shader approach

• Offers full parallelism
• One vertex at a time
• No scattered writes
Moving to the GPU: The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Moving to the GPU:
The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Moving to the GPU: The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Moving to the GPU: The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Moving to the GPU: The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Moving to the GPU: The pixel shader approach

- Offers full parallelism
- One vertex at a time
- No scattered writes
Downsides of the pixel shader approach

• No propagation of updates
  - If we double buffer

• Or non-deterministic
  - If we update in-place in a read/write array

• Momentum preservation
  - Lacking due to single-ended link updates
Can OpenCL help?

- Offers scattered writes as a feature as we saw earlier
- The GPU implementation could be more like the CPU
  - Solver per-link rather than per-vertex
  - Leads to races between links that update the same vertex
Execute independent subsets in parallel

• All links act at both ends

• Batch links
  - No two links in a given batch share a vertex
  - No data races
Execute independent subsets in parallel

• All links act at both ends
• Batch links
  - No two links in a given batch share a vertex
  - No data races
Execute independent subsets in parallel

- All links act at both ends
- Batch links
  - No two links in a given batch share a vertex
  - No data races
Execute independent subsets in parallel

- All links act at both ends
- Batch links
  - No two links in a given batch share a vertex
  - No data races
Execute independent subsets in parallel

• All links act at both ends

• Batch links
  - No two links in a given batch share a vertex
  - No data races
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
On a real cloth mesh we need many batches

• Create independent subsets of links through graph coloring.
• Synchronize between batches
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
On a real cloth mesh we need many batches

• Create independent subsets of links through graph coloring.
• Synchronize between batches
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
Driving batches and synchronizing

Simulation step

Iteration 0
- Batch 0
- Batch 1
- Batch 2
- Batch 3
- Batch 4

Iteration 1
- Batch 0
- Batch 1
- Batch 2
- Batch 3
- Batch 4

Iteration 2
- Batch 0
- Batch 1
- Batch 2
- Batch 3
- Batch 4
Driving batches and synchronizing

Iteration 0

Batch 0
Batch 1
Batch 2
Batch 3
Batch 4

ciErrNum = clSetKernelArg(solvePositionsFromLinksKernel, 0, sizeof(int), &startLink);
...
ciErrNum = clSetKernelArg(solvePositionsFromLinksKernel, 8, sizeof(cl_mem), &m_vertexData.m_clVertexPosition.m_buffer);

size_t workItems = groupSize*((numLinks + (groupSize-1)) / groupSize);
ciErrNum = clEnqueueNDRangeKernel(queue, solvePosKernel, 1, NULL, &workItems, &groupSize, 0, 0, 0);
Driving batches and synchronizing

Simulation step

Iteration

Batch 0
Batch 1
Batch 2
Batch 3
Batch 4

Wait a bit…
Driving batches and synchronizing

Simulation step

Iteration

Batch 0

Batch 1

Batch 2

Batch 3

Batch 4

Wait a bit…

Wait a bit…
Driving batches and synchronizing

Simulation step

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Batch 0</th>
<th>Batch 1</th>
<th>Batch 2</th>
<th>Batch 3</th>
<th>Batch 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Wait a bit…</td>
<td></td>
<td>Wait a bit…</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wait a bit…</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wait a bit…</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wait a bit…</td>
</tr>
</tbody>
</table>

Remember, 10 batches!
Returning to our batching

- 10 batches: 10 kernel dispatches
- 1/10 links per batch
- Low compute density per work item
Solving cloths together

• Solve multiple cloths together in n batches
• Grouping
  - Larger and reduced number of dispatches
  - Regain the parallelism that increased work-per-work-item removed
Packing for higher efficiency

• Can create larger groups
  - The cloth is fixed-structure
  - Can be preprocessed

• Fewer batches/dispatches

• Less parallelism
Local memory

- We’ve made use of scattered writes

- The next feature of OpenCL: local memory
  - Load data at the start of a block
  - Compute over multiple links together
  - Write data out again
Driving batches and synchronizing

Simulation step

Iteration

Batch 0
- Inner batch 0
- Inner batch 1

Batch 1
- Inner batch 0

Iteration

Batch 0
- Inner batch 0
- Inner batch 1

Batch 1
- Inner batch 0

Iteration

Batch 0
- Inner batch 0
- Inner batch 1

Batch 1
- Inner batch 0
How can we improve the batching?

So let’s look at the batching we saw before:

There are 4 batches:
- If we do this per group we need 3 groups rather than three OpenCL “work items” (which are, of course, NOT threads)
So let’s look at the batching we saw before:

There are 4 batches:
- If we do this per group we need 3 groups rather than three OpenCL “work items” (which are, of course, NOT threads)
So let’s look at the batching we saw before:

- There are 4 batches:
  - If we do this per group we need 3 groups rather than three OpenCL “work items” (which are, of course, NOT threads)
Solving in shared memory

__kernel void SolvePositionsFromLinksKernel( ...
    __global int *g_addresses,
    __global float4 g_positions,
    __local float4 *localPositions
    ...
) {
    for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
    {
        int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];

        localPositions[vertex] = g_positions[vertexAddress];
    }

    ... // Perform computation in shared buffer

    for( int vertex = get_local_id(0); vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
    {
        int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];

        g_positions[vertexAddress] = localPositions[vertex];
    }
}
__kernel void
SolvePositionsFromLinksKernel( ...
  __global int *g_addresses,
  global float4 g_positions,
  _local float4 *localPositions
...
}{
  for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
    localPositions[vertex] = g_positions[vertexAddress];
  }
  ...
  // Perform computation in shared buffer
  for( int vertex = get_local_id(0); vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
    g_positions[vertexAddress] = localPositions[vertex];
  }
}
Solving in shared memory

```c
__kernel void SolvePositionsFromLinksKernel( ...
    __global int *g_addresses,
    __global float4 g_positions,
    __local float4 *localPositions
    ...
) {

    for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
    {
        int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
        localPositions[vertex] = g_positions[vertexAddress];
    }

    /* Perform computation in shared buffer */

    for( int vertex = get_local_id(0); vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
    {
        int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
        g_positions[vertexAddress] = localPositions[vertex];
    }

    /* Load data from global buffers into the shared region */
```

- Page 51
Solving in shared memory

__kernel void SolvePositionsFromLinksKernel( ...
    __global int *g_addresses,
    __global float4 g_positions,
    __local float4 *localPositions ...
)
{
    for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
    {
        int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
        localPositions[vertex] = g_positions[vertexAddress];
    }
    ...
    // Perform computation in shared buffer

for( int vertex = get_local_id(0); vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
{
    int vertexAddress = g_addresses[groupID*VERTS_PER_GROUP + vertex];
    g_positions[vertexAddress] = localPositions[vertex];
}

Write back to the global buffer after computation
Group execution

- The sequence of operations for the first batch is:
Group execution

- The sequence of operations for the first batch is:
Group execution

• The sequence of operations for the first batch is:
Group execution

- The sequence of operations for the first batch is:

Few links so low packing efficiency:
Not a problem with larger cloth
Group execution

• The sequence of operations for the first batch is:
Group execution

• The sequence of operations for the first batch is:

```c
// load
barrier(CLK_LOCAL_MEM_FENCE);

for ( each subgroup ) {
    // Process a subgroup
    // Barrier is safe because entire group executes
    // whole loop
    barrier(CLK_LOCAL_MEM_FENCE);
}

// Store
```
Why is this an improvement?

• So we still need 10*4 batches. What have we gained?
  - The batches within a group chunk are in-shader loops
  - Only 4 shader dispatches, each with significant overhead

• The barriers will still hit performance
  - We are no longer dispatch bound, but we are likely to be on-chip synchronization bound
Exploiting the SIMD architecture

• **Hardware executes 64- or 32-wide SIMD**
  - We’re speaking of DX11 class hardware here. ARM’s OpenCL architectures do not.
  - Unfortunately OpenCL isn’t very portable if you want performance...

• **Sequentially consistent at the SIMD level**

• **Synchronization is now implicit**
  - Take care
  - Execute over groups that are SIMD width or a divisor thereof
Group execution

- The sequence of operations for the first batch is:
Driving batches and synchronizing

Simulation step

Iteration

Batch 0

Inner batch 0

Inner batch 1

Batch 1

Inner batch 0

Synchronize

Synchronize
Performance gains

• For 90,000 links:
  - No solver running in 2.98 ms/frame
  - Fully batched link solver in 3.84 ms/frame
  - SIMD batched solver 3.22 ms/frame
  - CPU solver 16.24 ms/frame

• 3.5x improvement in solver alone

• (67x improvement CPU solver)
One more thing…

• Remember that OpenCL often runs on the GPU
  - Indeed our optimizations target precisely this case

The driver mixes OpenCL and OpenGL execution
Efficiently output vertex data

• Cloth simulation updates vertex positions
  - Generated on the GPU
  - Need to be used on the GPU for rendering
  - Why not keep them there?

• Large amount of data to update
  - Many vertices in fine simulation meshes
  - Normals and other information present
Create a vertex buffer

// Construct VBO
glGenBuffers(1, &clothVBO);
glBindBuffer(GL_ARRAY_BUFFER, clothVBO);

// Do initial upload to ensure that the buffer exists on the device
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_struct)*width*height, &cpu_buffer[0], GL_DYNAMIC_DRAW);
int error = glGetError();
glBindBuffer(GL_ARRAY_BUFFER, 0);

...

// Create the CL buffer from a GL buffer
m_buffer = clCreateFromGLBuffer(m_context, CL_MEM_WRITE_ONLY, openGLVBO, &ciErrNum);

...

// Asynchronously acquire and release the CL buffer that references a GL object
// to correctly synchronize with OpenGL

ciErrNum = clEnqueueAcquireGLObjects(m_cqCommandQue, 1, &m_buffer, 0, 0, NULL);
ciErrNum = clEnqueueNDRangeKernel(m_cqCommandQue, outputKernel, 1, NULL, &numWorkItems, &workGroupSize, 0, 0, 0);
ciErrNum = clEnqueueReleaseGLObjects(m_cqCommandQue, 1, &m_buffer, 0, 0, 0);
Performance gains

• For 90,000 links with copy on GPU:
  - No solver running in 0.58 ms/frame
  - Fully batched link solver in 0.82 ms/frame
  - SIMD batched solver 0.617 ms/frame

• 6.5x improvement in solver
• 6.5x improvement from CPU copy alone
• 23x improvement over simpler solver with host copy