Bullet Cloth Simulation

Presented by: Justin Hensley

Implemented by: Lee Howes

SIGGRAPHASIA2010
SEOUL

Course Agenda

OpenCL Review
OpenCL Development Tips
OpenGL/OpenCL Interoperability
Rigid body particle simulation
<15 minute Break>
Bullet cloth

Galaxy n-body simulation

Visualization with OpenCL

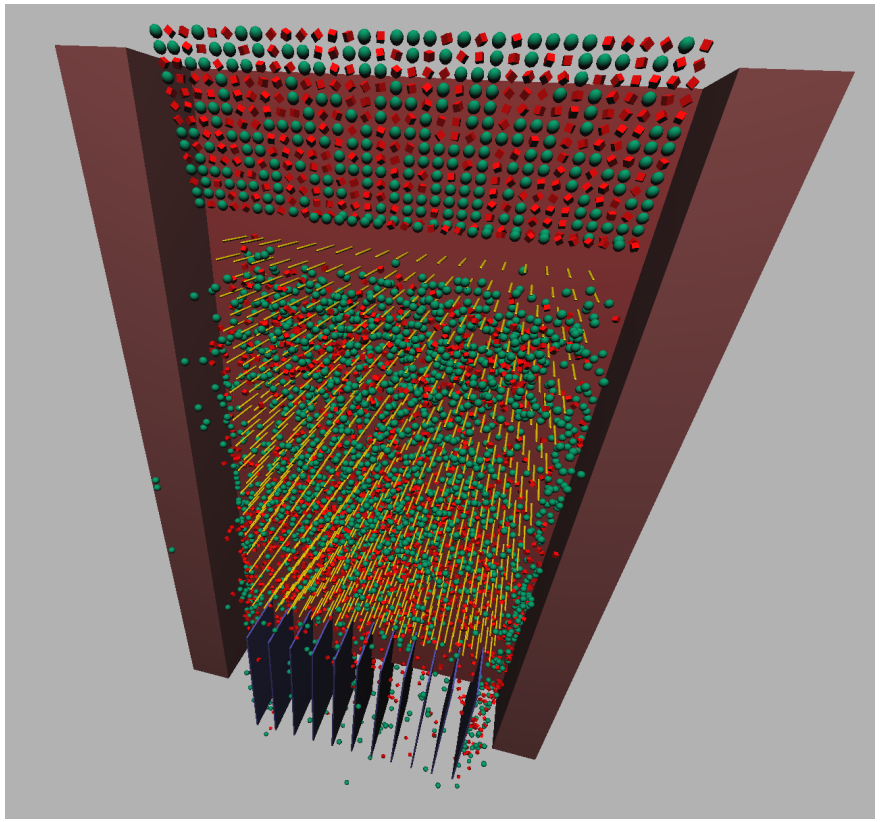
Questions

Bullet physics

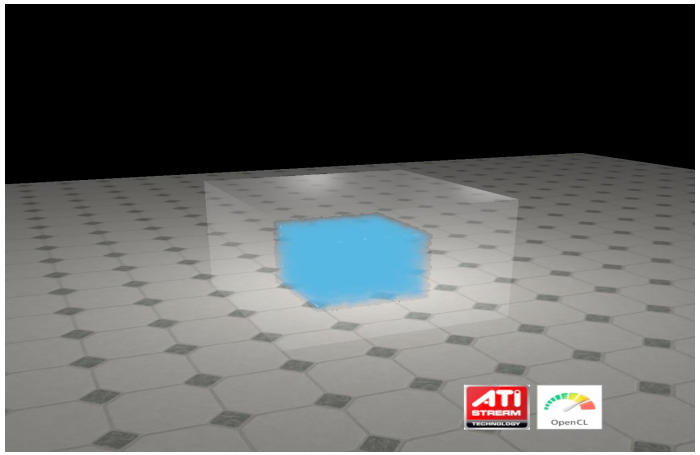
- **An open source physics SDK for games (http://bulletphysics.org)**
  - Popular physics SDK
  - Zlib license for copy-and-use openness
  - Development led by Erwin Coumans of AMD (formerly at Sony)
- **Includes**
  - Rigid body dynamics, e.g.
    - **Ragdolls, destruction, and vehicles**
  - Soft body dynamics
    - **Cloth, rope, and deformable volumes**
- **Collaborations on GPU acceleration**
  - Cloth/soft body and fluids in OpenCL and DirectCompute
  - Fully open-source contributions

© Copyright Khronos Group, 2010 - Page 3

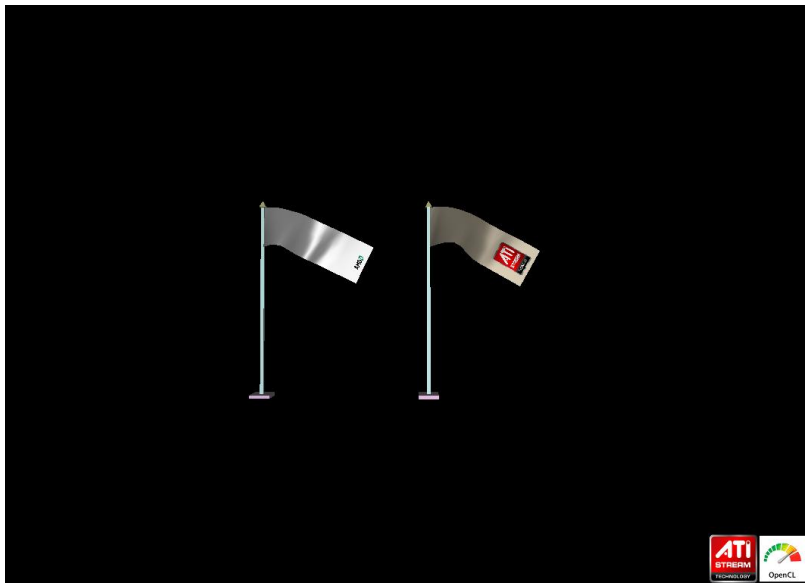
OpenCL physics is…



Rigid bodies



Fluids



Cloth

© Copyright Khronos Group, 2010 - Page 4

Introducing cloth simulation

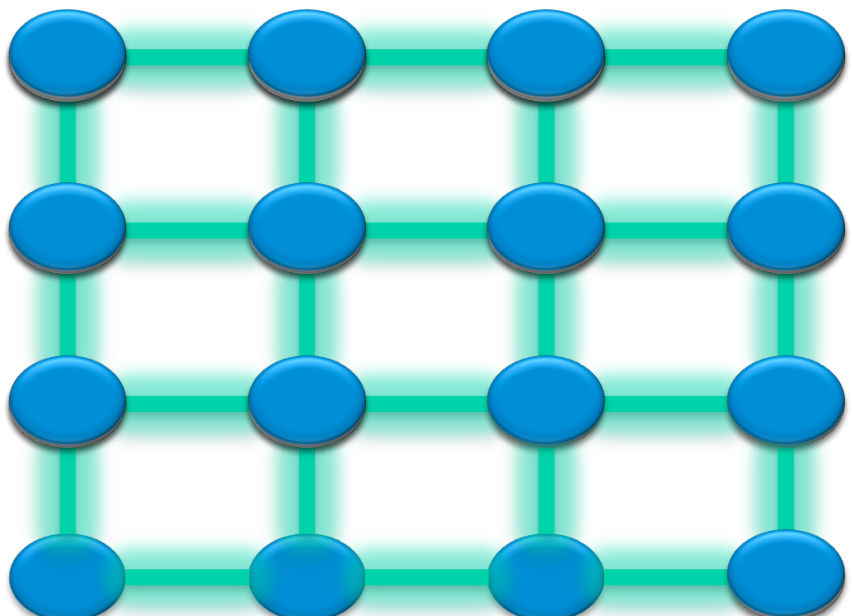
- **A subset of the possible set of soft bodies**

- **For real-time generally based on a mass/spring system**
  - Large collection of masses (particles)
  - Connect using spring constraints
  - Layout and properties change properties of cloth

© Copyright Khronos Group, 2010 - Page 5

Springs and masses

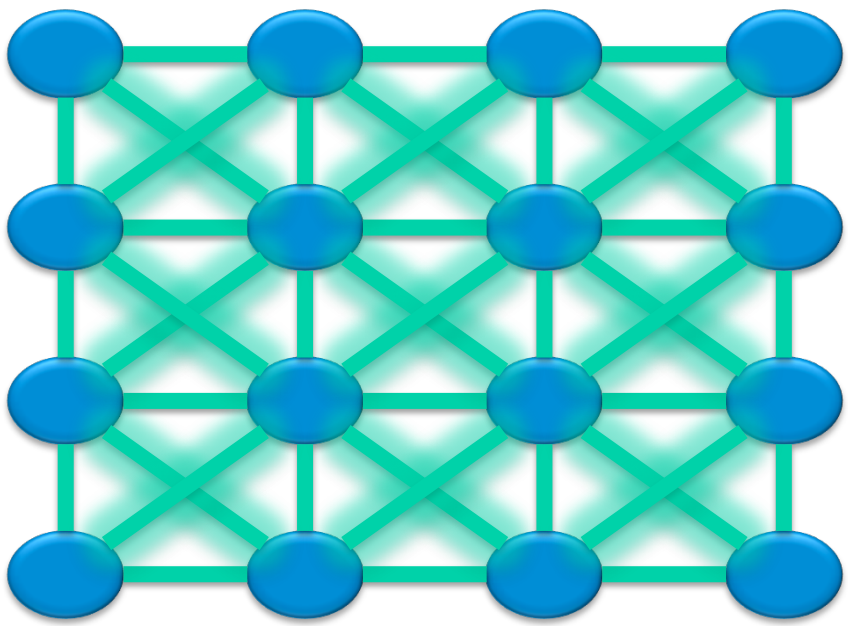
- **Three main types of springs**
  - Structural
  - Shearing
  - Bending



© Copyright Khronos Group, 2010 - Page 6

Springs and masses

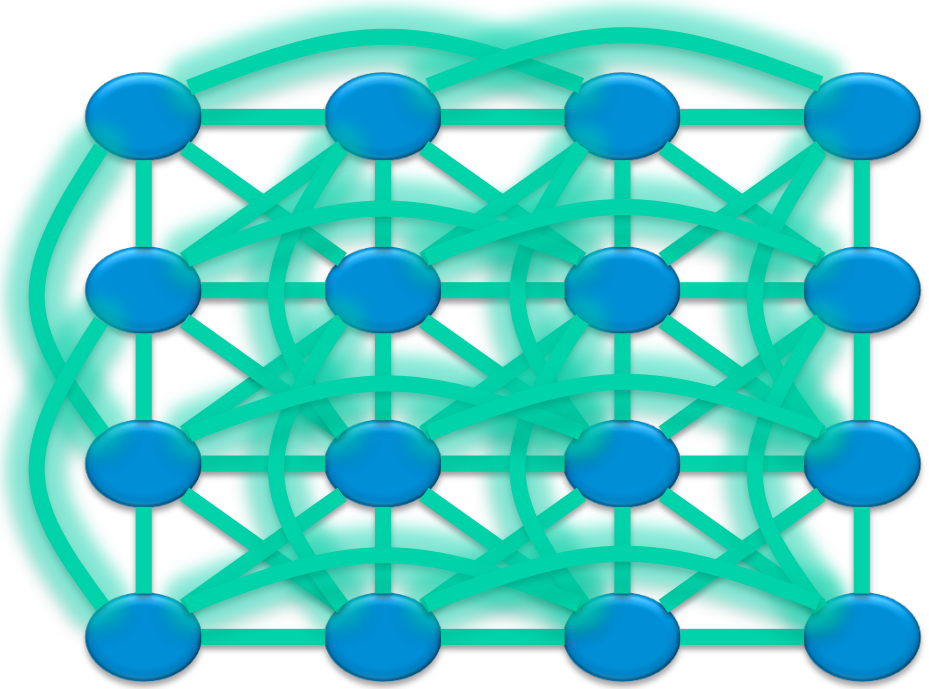
- **Three main types of springs**
  - Structural
  - Shearing
  - Bending



© Copyright Khronos Group, 2010 - Page 7

Springs and masses

- **Three main types of springs**
  - Structural
  - Shearing
  - Bending

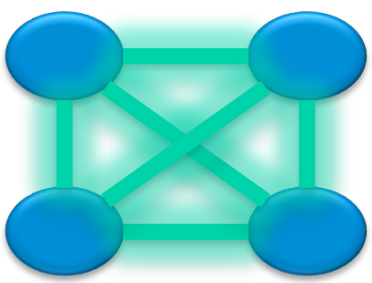


© Copyright Khronos Group, 2010 - Page 8

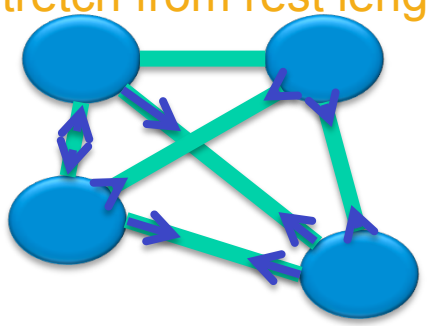
Parallelism

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

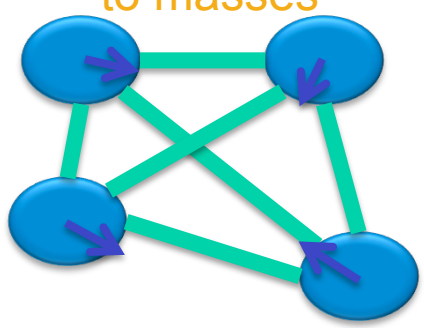
Original layout



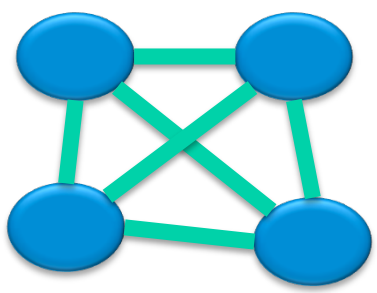
Current layout:
Compute forces as
Stretch from rest length



Apply position
corrections
to masses



Compute new
positions



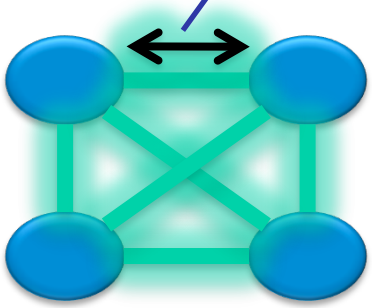
© Copyright Khronos Group, 2010 - Page 9

Parallelism

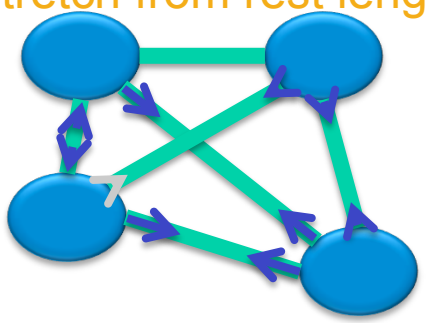
- **Large number of particles**
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles

Rest length of spring

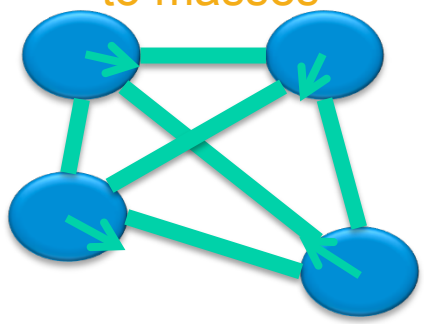
Original layout



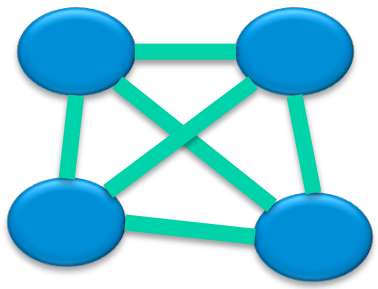
Current layout:
Compute forces as
Stretch from rest length



Apply position
corrections
to masses



Compute new
positions

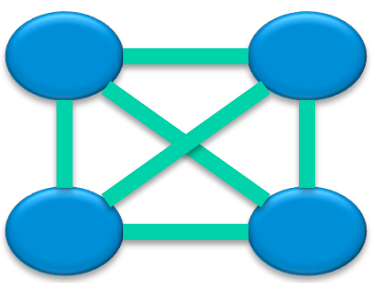


© Copyright Khronos Group, 2010 - Page 10

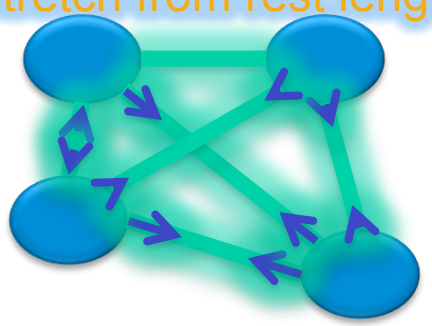
Parallelism

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

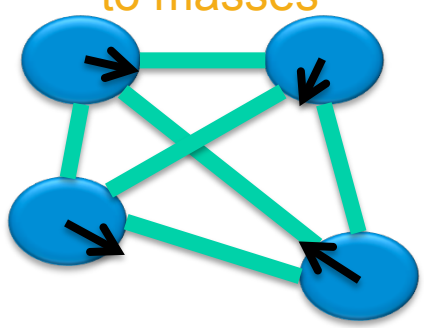
Original layout



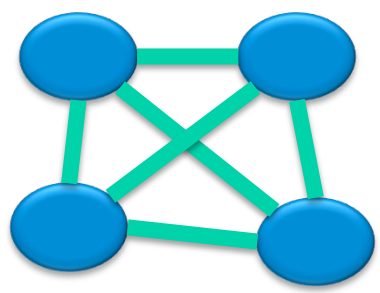
Current layout:
Compute forces as
Stretch from rest length



Apply position
corrections
to masses



Compute new
positions

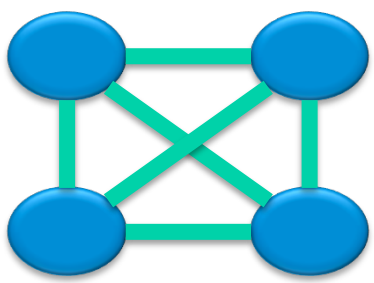


© Copyright Khronos Group, 2010 - Page 11

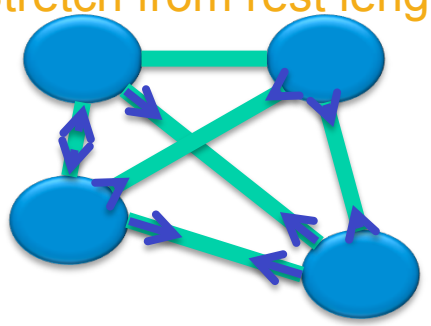
Parallelism

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

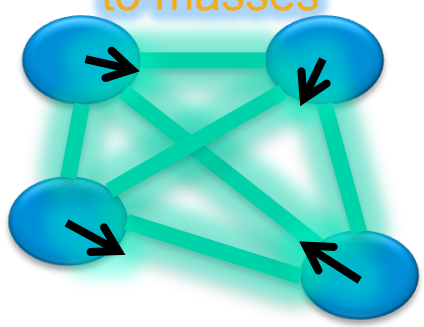
Original layout



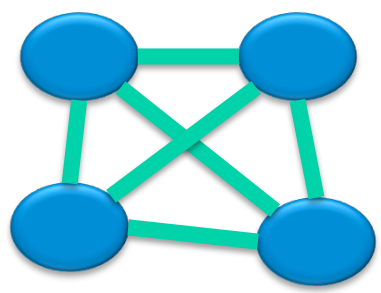
Current layout:
Compute forces as
Stretch from rest length



Apply position
corrections
to masses



Compute new
positions

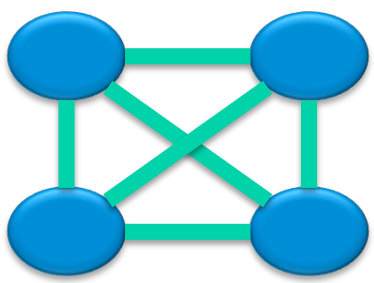


© Copyright Khronos Group, 2010 - Page 12

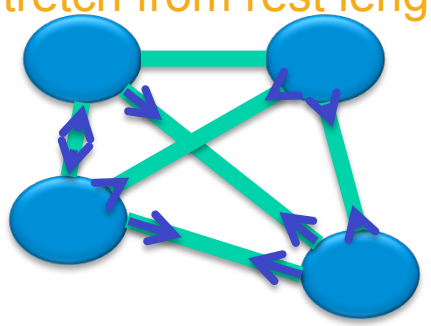
Parallelism

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

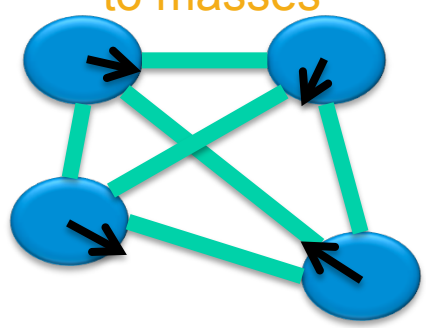
Original layout



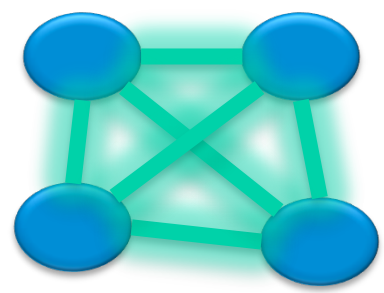
Current layout:
Compute forces as
Stretch from rest length



Apply position
corrections
to masses



Compute new
positions



© Copyright Khronos Group, 2010 - Page 13

CPU approach to simulation

- **Iterative integration over vertex positions**
  - For each spring computes a force.
  - Updates both vertices with a new position.
  - Repeat n times where n is configurable.

- **Note that the computation is serial**
  - Propagation of values through the solver is immediate.

© Copyright Khronos Group, 2010 - Page 14

The CPU approach

for each iteration

```
{
  for(int linkIndex = 0; linkIndex < numLinks; ++linkIndex)
  {
    float massLSC =
      (inverseMass0 + inverseMass1)/linearStiffnessCoefficient;
    float k = ((restLengthSquared - lengthSquared) /
            (massLSC * (restLengthSquared + lengthSquared)));

    vertexPosition0 -= length * (k*inverseMass0);
    vertexPosition1 += length * (k*inverseMass1);
  }
}
```

© Copyright Khronos Group, 2010 - Page 15

The CPU approach

```
for each iteration
{
 for(int linkIndex = 0; linkIndex < numLinks; ++linkIndex)
 {
  float massLSC =
   (inverseMass0 + inverseMass1)/linearStiffnessCoefficient;
  float k = ((restLengthSquared - lengthSquared) /
          (massLSC * (restLengthSquared + lengthSquared)));

  vertexPosition0 -= length * (k*inverseMass0);
  vertexPosition1 += length * (k*inverseMass1);
 }
}
```

© Copyright Khronos Group, 2010 - Page 16

The CPU approach

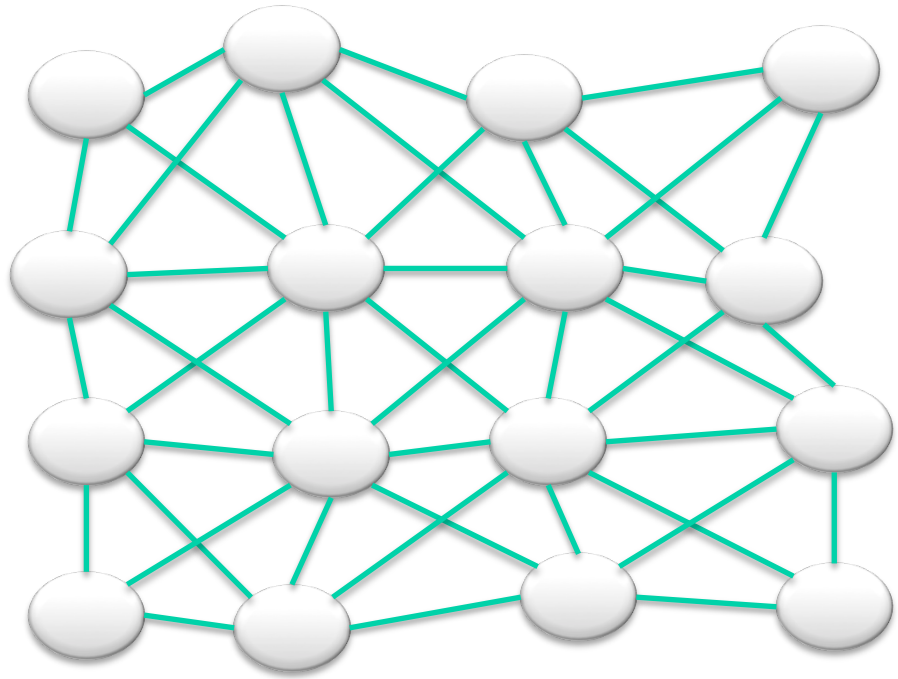
```
for each iteration
{
 for(int linkIndex = 0; linkIndex < numLinks; ++linkIndex)
 {
  float massLSC =
    (inverseMass0 + inverseMass1)/linearStiffnessCoefficient;
  float k = ((restLengthSquared - lengthSquared) /
          (massLSC * (restLengthSquared + lengthSquared)));

  vertexPosition0 -= length * (k*inverseMass0);
  vertexPosition1 += length * (k*inverseMass1);
 }
}
```

© Copyright Khronos Group, 2010 - Page 17

CPU approach to simulation

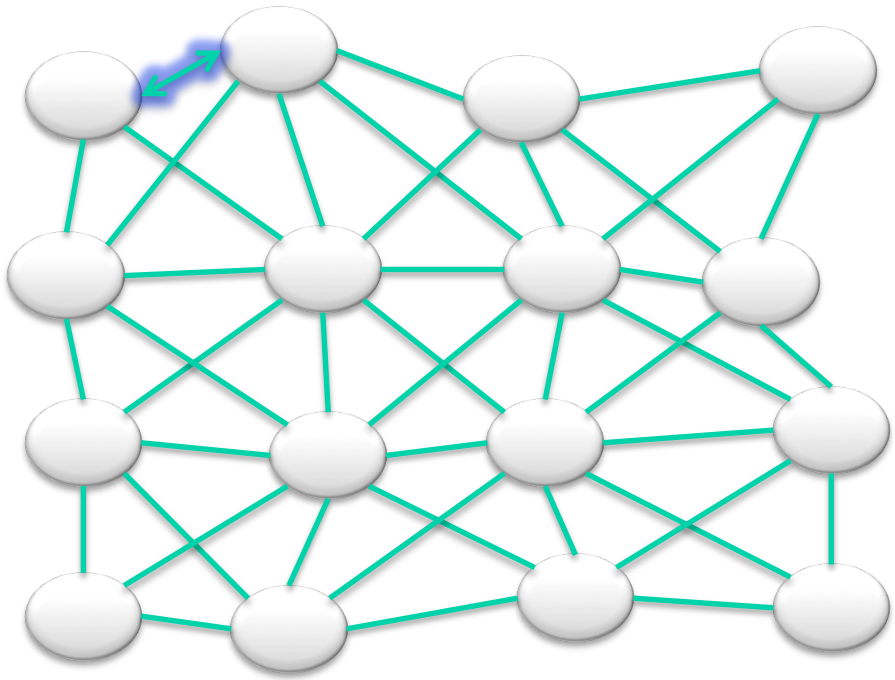
- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 18

CPU approach to simulation

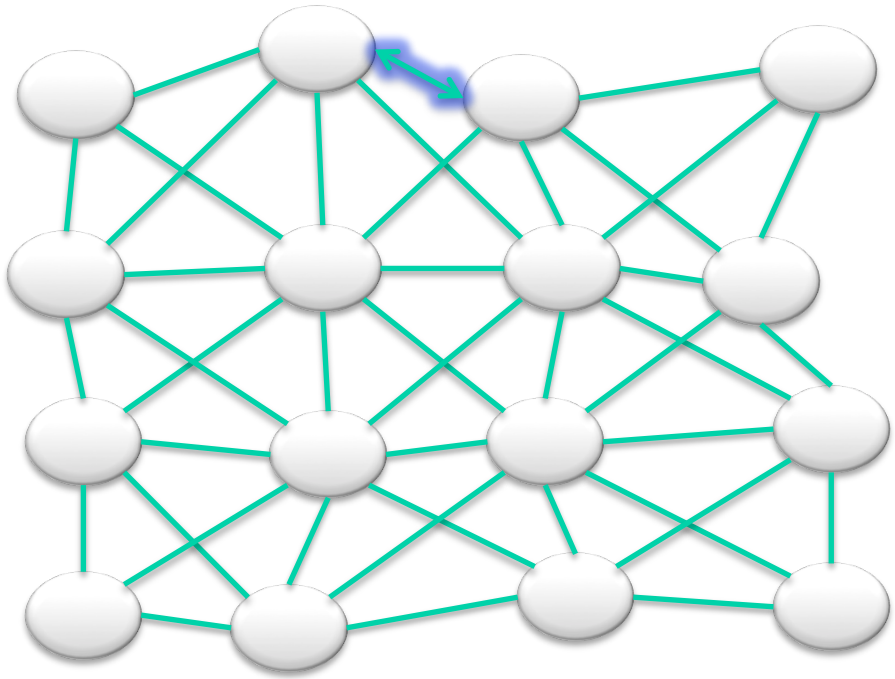
- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 19

CPU approach to simulation

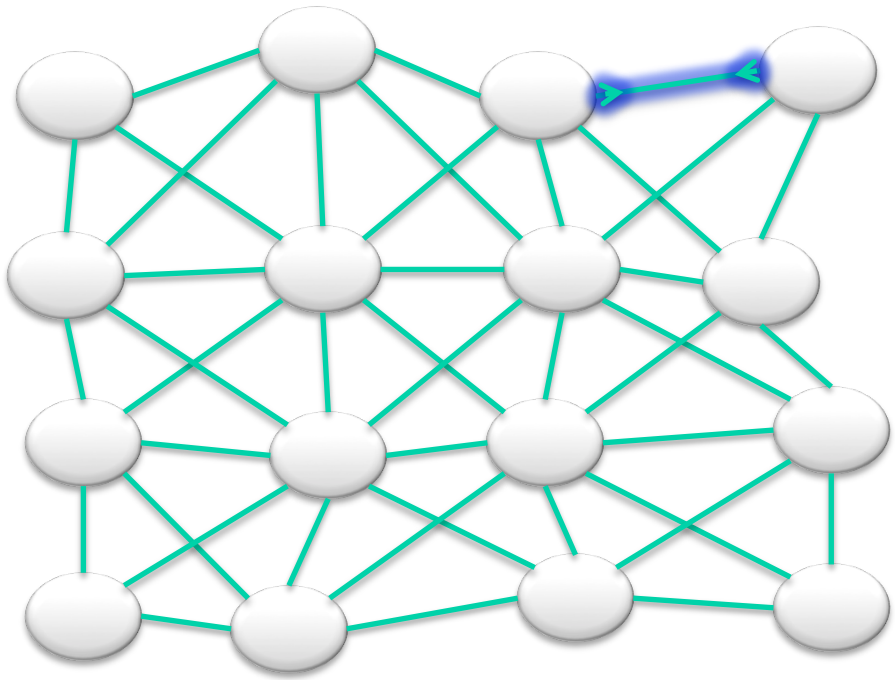
- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 20

CPU approach to simulation

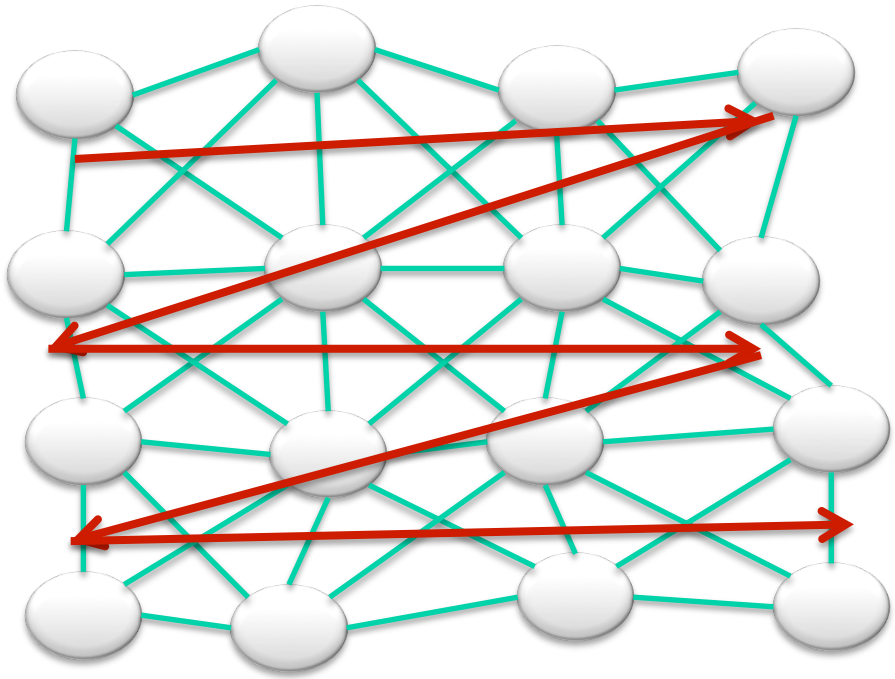
- One link at a time
- Perform updates in place
- “Gauss-Seidel” style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 21

CPU approach to simulation

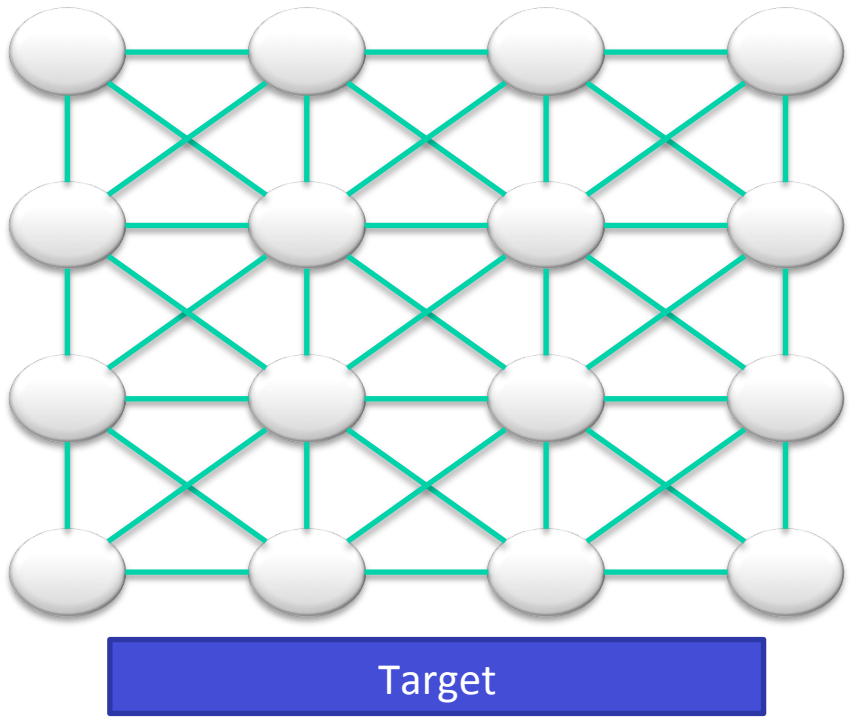
- One link at a time
- Perform updates in place
- "Gauss-Seidel" style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 22

CPU approach to simulation

- One link at a time
- Perform updates in place
- "Gauss-Seidel" style
- Conserves momentum
- Iterate n times



© Copyright Khronos Group, 2010 - Page 23

GPU parallelism

- **The CPU implementation was serial.**
  - No atomicity issues.
  - Value propagation immediate from a given update.

- **The GPU implementation is parallel within a cloth.**
  - Multiple updates to the same node create races.

© Copyright Khronos Group, 2010 - Page 24

Vertex solver: a single batch

- **Single batch**
  - Highly efficient per solver iteration
  - Can re-use position data for central node
  - Need to cleverly arrange data to allow efficient loop unrolling

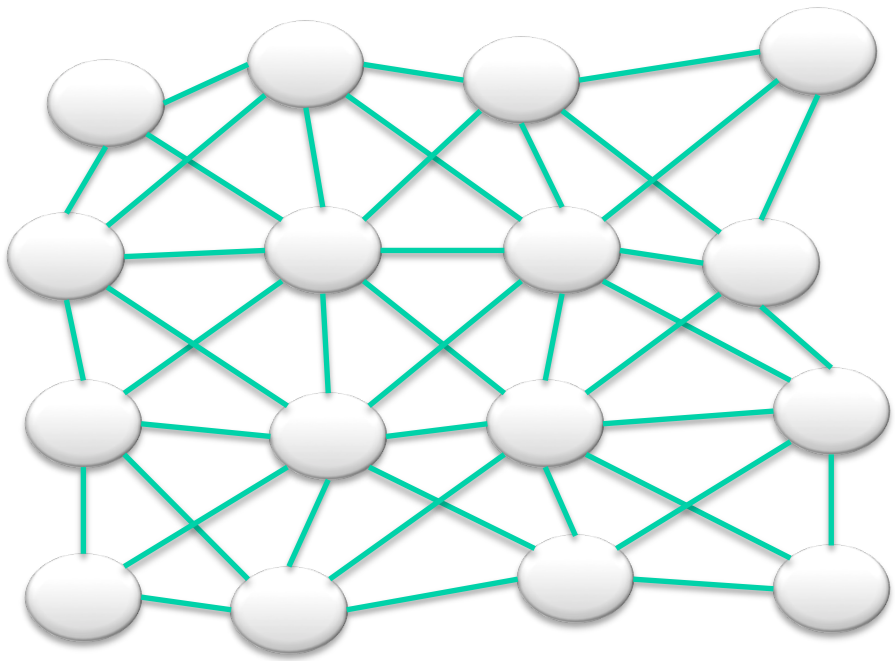
- **Double buffer the vertex data**
  - Updates will then not be seen until the next iteration

- **Write to same buffer**
  - Updates can be seen quickly, but the computation will be non-deterministic

© Copyright Khronos Group, 2010 - Page 25

Vertex solver: a single batch

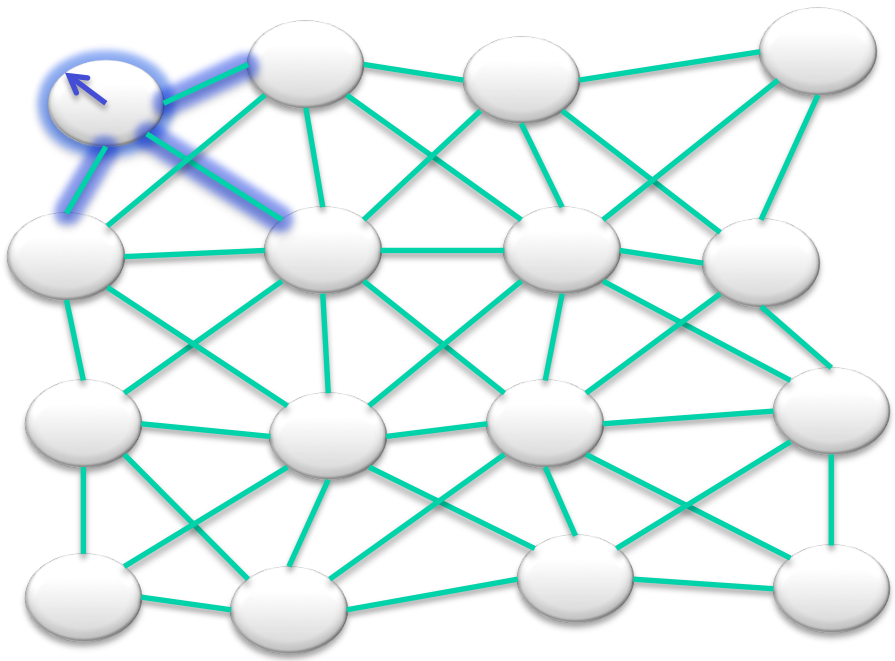
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 26

Moving to the GPU: The shader approach

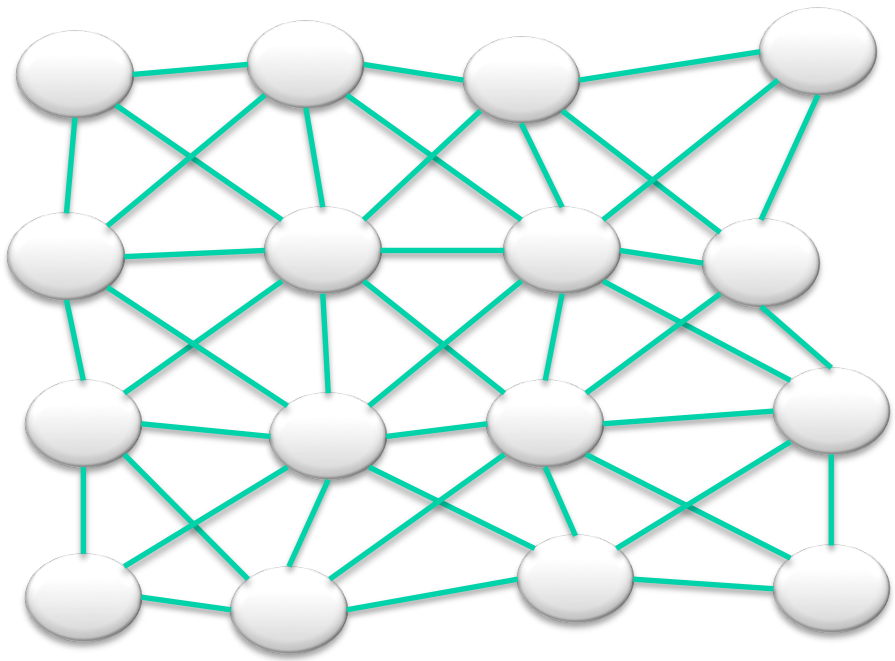
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 27

Moving to the GPU: The shader approach

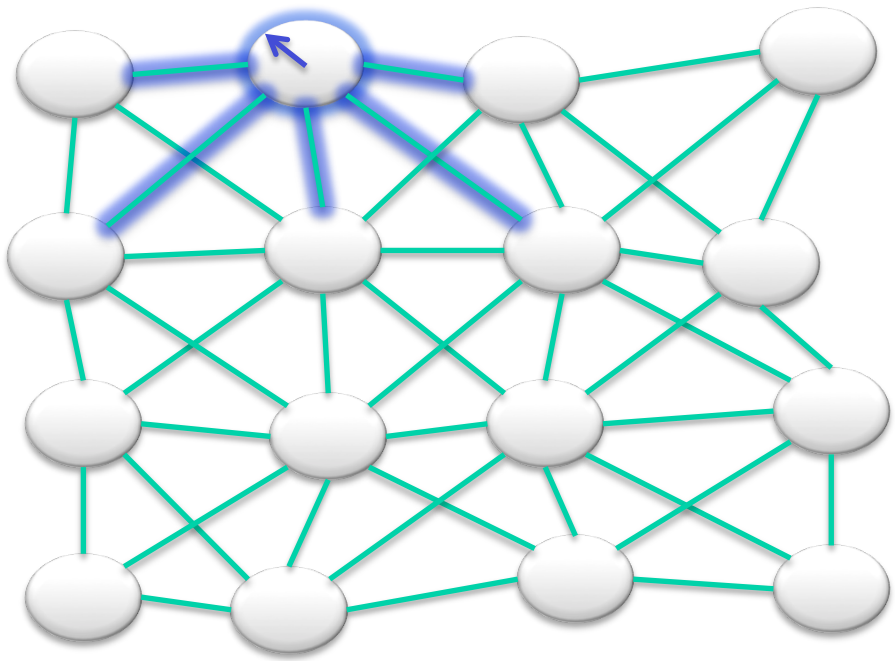
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 28

Moving to the GPU: The shader approach

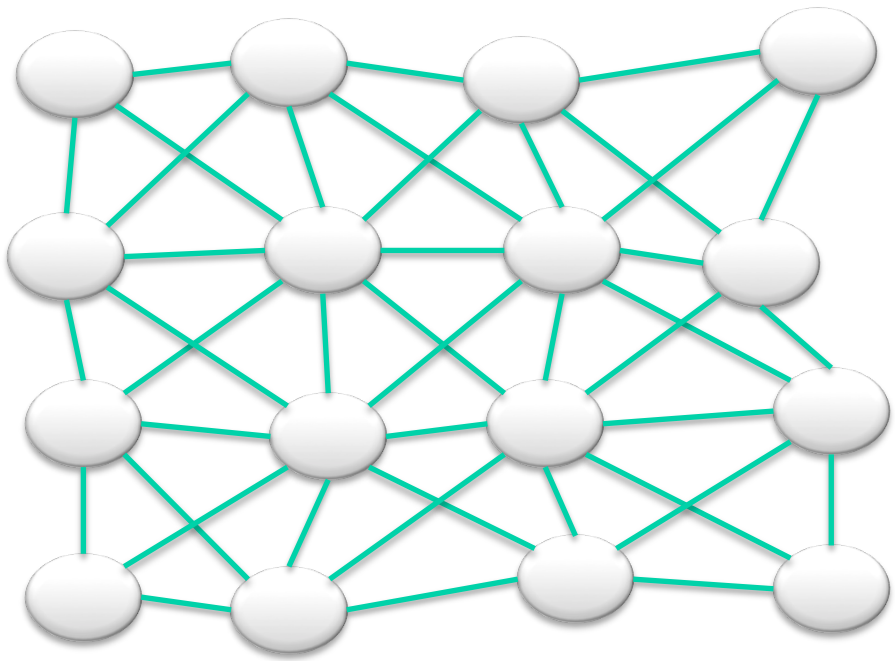
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 29

Moving to the GPU: The shader approach

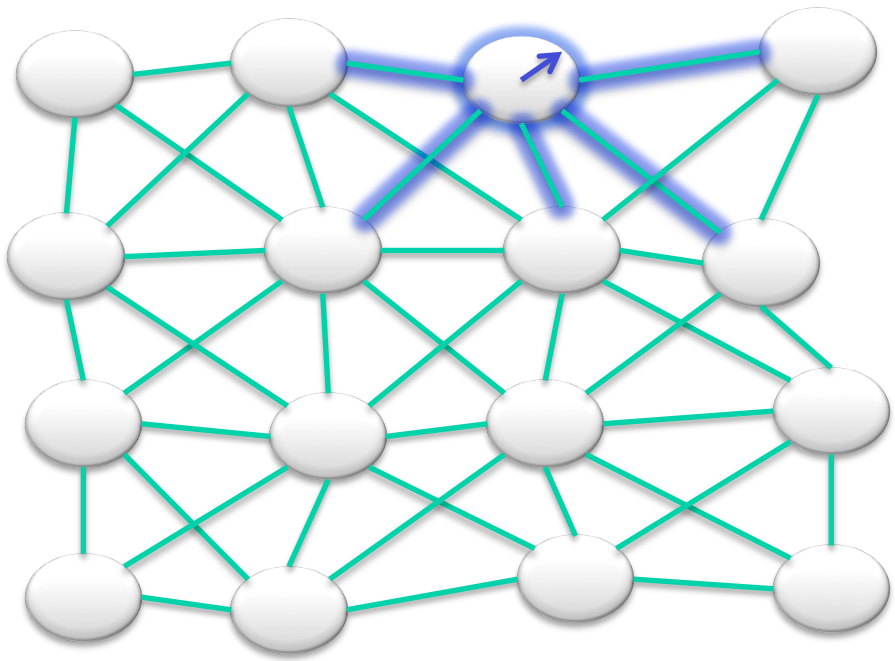
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 30

Moving to the GPU: The shader approach

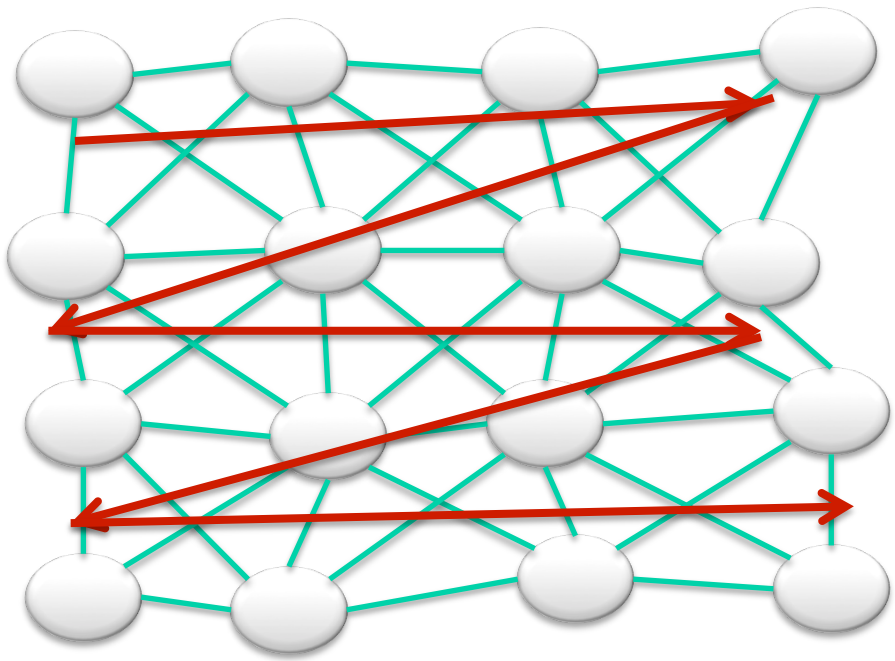
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 31

Moving to the GPU: The shader approach

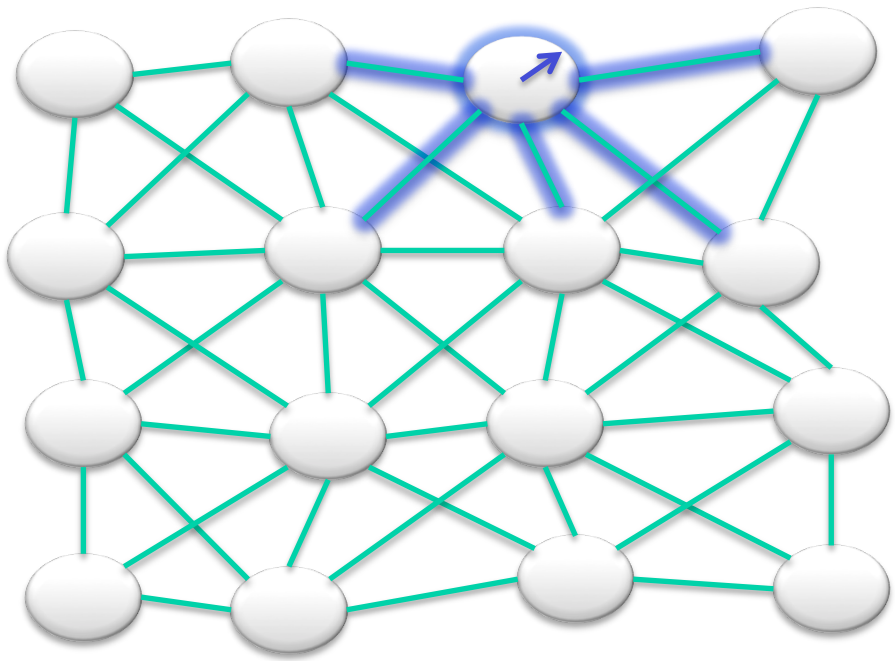
- Offers full parallelism
- One vertex at a time
- No scattered writes



© Copyright Khronos Group, 2010 - Page 32

Downsides of the shader approach

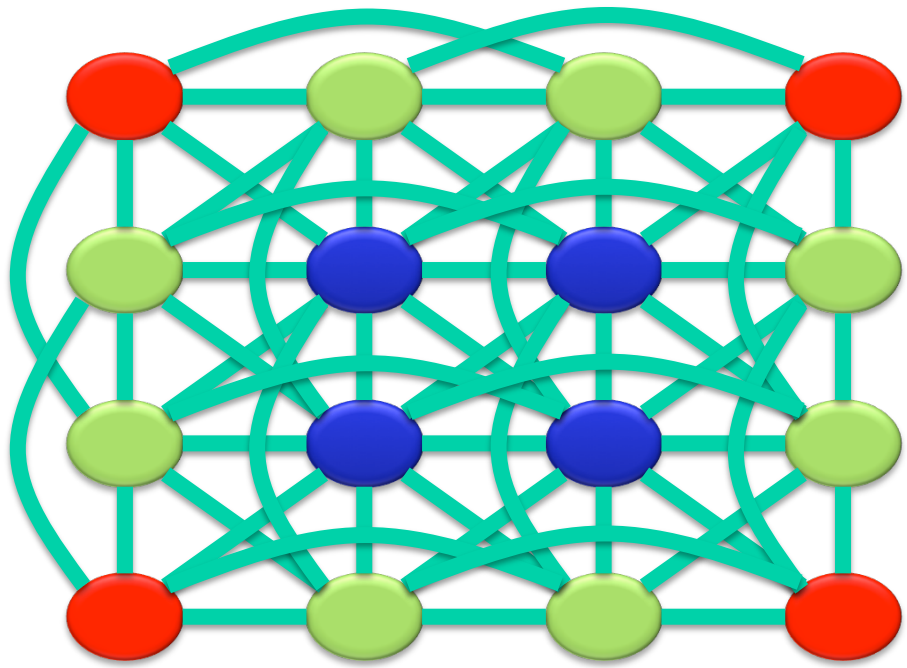
- **No propagation of updates**
  - If we double buffer
- **Or non-deterministic**
  - If we update in-place in a read/write array
- **Momentum preservation**
  - Lacking due to single-ended link updates



© Copyright Khronos Group, 2010 - Page 33

A branch divergence optimization

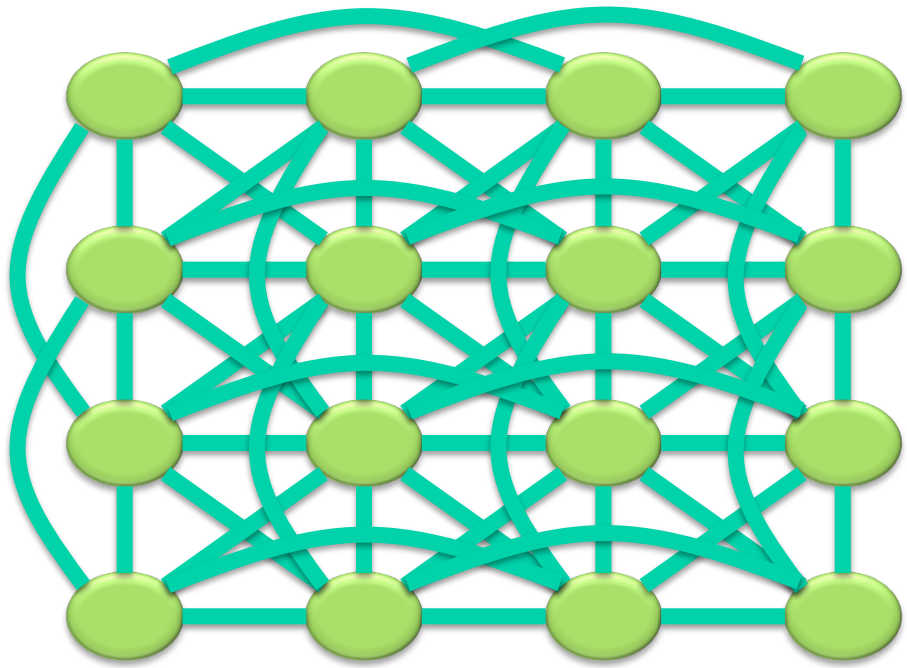
- **The GPU is a vector machine: a collection of wide SIMD cores**
  - Divergent branches across a vector hurt performance
  - Nodes have different valence/degree
  - Regular mesh
    - **Low overhead**
    - **Similar degree throughout**
  - Complicated mesh
    - **Arbitrary numerous peaks**
    - **Pack vertices by degree**



© Copyright Khronos Group, 2010 - Page 34

Batching the simulation

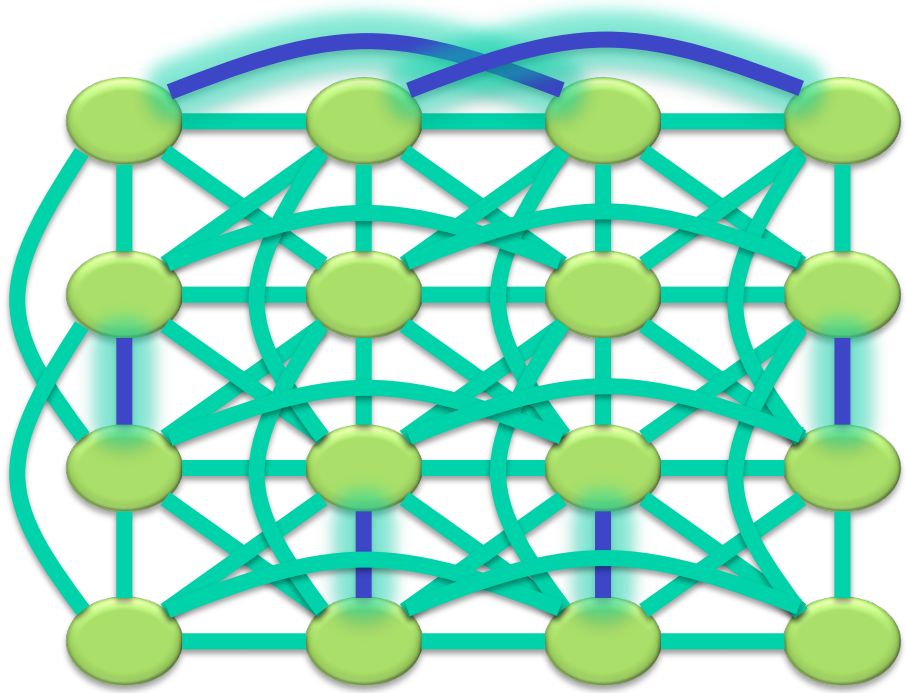
- Create independent subsets of links through graph coloring.
- Synchronize between batches



© Copyright Khronos Group, 2010 - Page 35

Batching the simulation

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

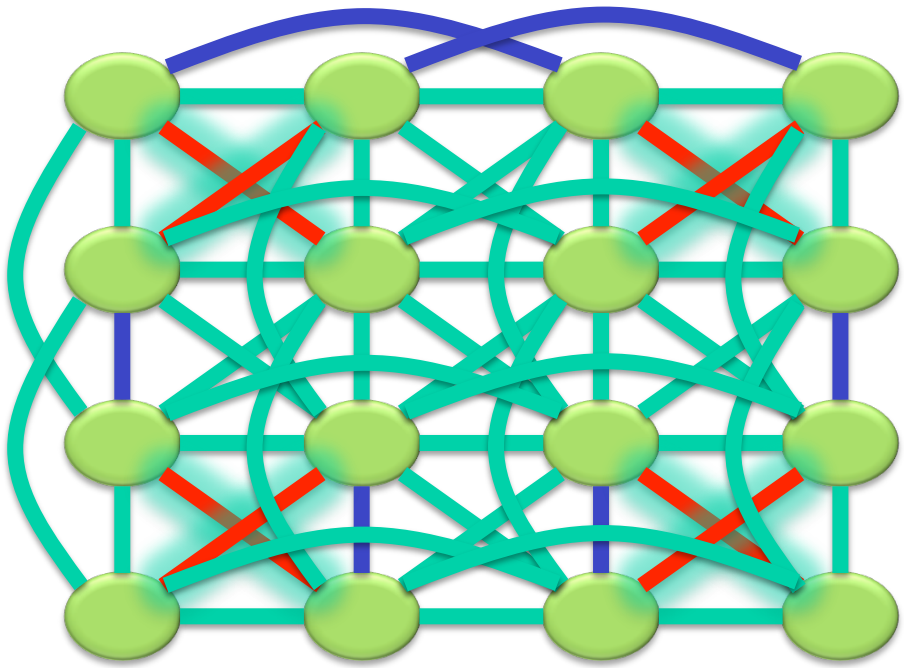


1 batch

© Copyright Khronos Group, 2010 - Page 36

Batching the simulation

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

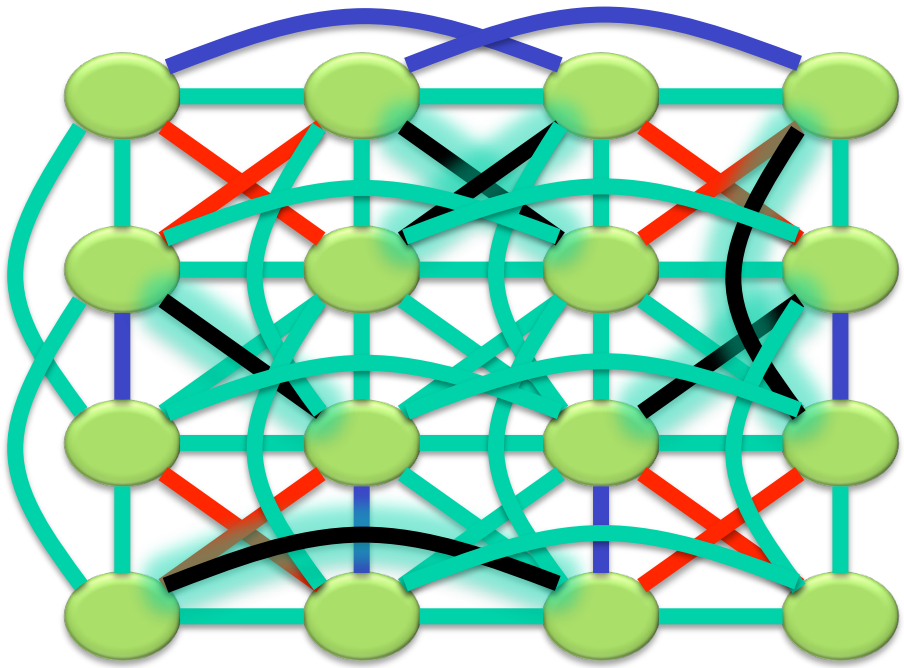


2 batches

© Copyright Khronos Group, 2010 - Page 37

Batching the simulation

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

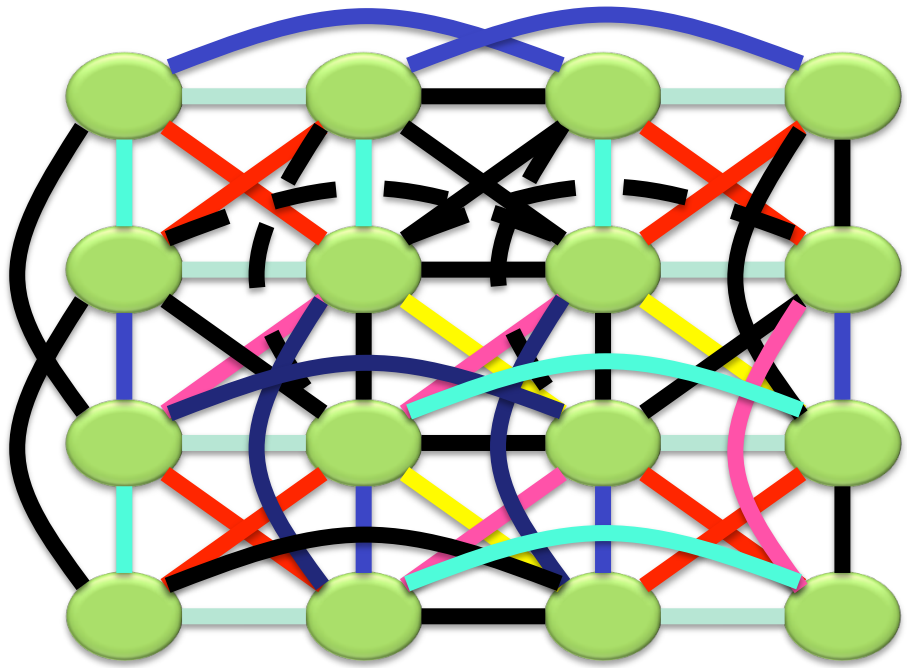


3 batches

© Copyright Khronos Group, 2010 - Page 38

Batching the simulation

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



10 batches

© Copyright Khronos Group, 2010 - Page 39

Driving batches

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start = m_linkData.m_batchStartLengths[i].first;
    int num   = m_linkData.m_batchStartLengths[i].second;
    for(int linkIndex = start; linkIndex < start + num; ++linkIndex)
    {
      …
    }
  }
}
```

Driving batches

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start = m_linkData.m_batchStartLengths[i].first;
    int num   = m_linkData.m_batchStartLengths[i].second;
    for(int linkIndex = start; linkIndex < start + num; ++linkIndex)
    {
      …
    }
  }
}
```

Driving batches

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start = m_linkData.m_batchStartLengths[i].first;
    int num   = m_linkData.m_batchStartLengths[i].second;
    for(int linkIndex = start; linkIndex < start + num; ++linkIndex)
    {
      …
    }
  }
}
```

Driving batches

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start   = m_linkData.m_batchStartLengths[i].first;
    int num     = m_linkData.m_batchStartLengths[i].second;
    for(int linkIndex = start; linkIndex < start + num; ++linkIndex)
    {
      …
    }
  }
}
```

Statically generated batches and pre-sorted buffers.

Driving batches

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start = m_linkData.m_batchStartLengths[i].first;
    int num   = m_linkData.m_batchStartLengths[i].second;
    for(int linkIndex = start; linkIndex < start + num; ++linkIndex)
    {
      …
    }
  }
}
```

This loop is now fully parallel.

Dispatching a batch

```
solvePositionsFromLinksKernel.kernel.setArg(0, startLink);
solvePositionsFromLinksKernel.kernel.setArg(1, numLinks);
solvePositionsFromLinksKernel.kernel.setArg(2, kst);
solvePositionsFromLinksKernel.kernel.setArg(3, ti);
solvePositionsFromLinksKernel.kernel.setArg(4, m_linkData.m_clLinks.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(5, m_linkData.m_clLinksMassLSC.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(6, m_linkData.m_clLinksRestLengthSquared.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(7, m_vertexData.m_clVertexInverseMass.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(8, m_vertexData.m_clVertexPosition.getBuffer());

int numWorkItems = workGroupSize*((numLinks + (workGroupSize-1)) / workGroupSize);
cl_int err = m_queue.enqueueNDRangeKernel(
        solvePositionsFromLinksKernel.kernel,
        cl::NullRange, cl::NDRange(numWorkItems), cl::NDRange(workGroupSize));
```

Dispatching a batch

```
solvePositionsFromLinksKernel.kernel.setArg(0, startLink);
solvePositionsFromLinksKernel.kernel.setArg(1, numLinks);
solvePositionsFromLinksKernel.kernel.setArg(2, k);
solvePositionsFromLinksKernel.kernel.setArg(3, d);
solvePositionsFromLinksKernel.kernel.setArg(4, m_linkData.m_clLinks.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(5, m_linkData.m_clLinksMassLSC.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(6, m_linkData.m_clLinksRestLengthSquared.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(7, m_vertexData.m_clVertexInverseMass.getBuffer());
solvePositionsFromLinksKernel.kernel.setArg(8, m_vertexData.m_clVertexPosition.getBuffer());

int numWorkItems = workGroupSize*((numLinks + (workGroupSize-1)) / workGroupSize);
cl_int err = m_queue.enqueueNDRangeKernel(
            solvePositionsFromLinksKernel.kernel,
            cl::NullRange, cl::NDRange(numWorkItems), cl::NDRange(workGroupSize));
```

Note that the number of work items is rounded to a multiple of the group size

The OpenCL link solver kernel header

```
__kernel void
SolvePositionsFromLinksKernel(
    const int startLink,
    const int numLinks,
    const float kst,
    const float ti,
    __global int2    *g_linksVertexIndices,
    __global float   *g_linksMassLSC,
    __global float   *g_linksRestLengthSquared,
    __global float   *g_verticesInverseMass,
    __global float4  *g_vertexPositions)
{
  …
}
```

The OpenCL link solver kernel body

```
int linkID = get_global_id(0) + startLink;
if( get_global_id(0) < numLinks ) {
   float massLSC = g_linksMassLSC[linkID];
   float restLengthSquared = g_linksRestLengthSquared[linkID];

   if( massLSC > 0.0f ) {
      int2 nodeIndices = g_linksVertexIndices[linkID];
      float3 position0 = g_vertexPositions[nodeIndices.x].xyz;
      float3 position1 = g_vertexPositions[nodeIndices.y].xyz;

      float inverseMass0 = g_verticesInverseMass[nodeIndices.x];
      float inverseMass1 = g_verticesInverseMass[nodeIndices.y];
```

The OpenCL link solver kernel body

```
int linkID = get_global_id(0) + startLink;
if( get_global_id(0) < numLinks ) {
    float massLSC = g_linksMassLSC[linkID];
    float restLengthSquared = g_linksRestLengthSquared[linkID];

    if( massLSC > 0.0f ) {
        int2 nodeIndices = g_linksVertexIndices[linkID];
        float3 position0 = g_vertexPositions[nodeIndices.x].xyz;
        float3 position1 = g_vertexPositions[nodeIndices.y].xyz;

        float inverseMass0 = g_verticesInverseMass[nodeIndice
        float inverseMass1 = g_verticesInverseMass[nodeIndice
```

The number of links might not be a multiple of the block size.

Changing the memory layout might be a better solution.

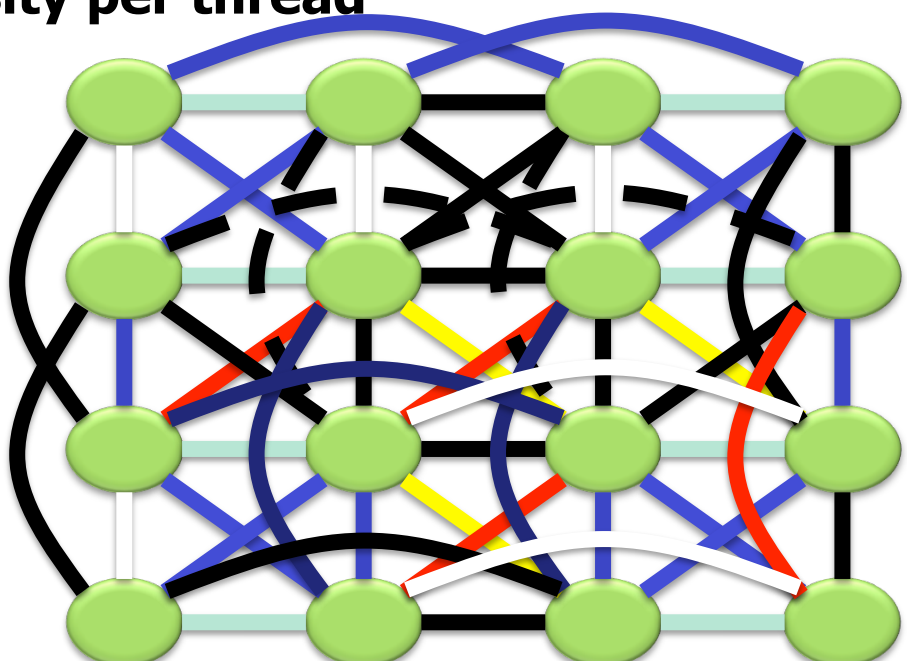
The OpenCL link solver kernel body

```
        float3 del = position1 - position0;
        float lengthSquared = dot( del, del );
        float k    = ((restLengthSquared – lengthSquared )/(massLSC*(lengthSquared ))*kst;
        position0 = position0 - del*(k*inverseMass0);
        position1 = position1 + del*(k*inverseMass1);

        g_vertexPositions[nodeIndices.x] = (float4)(position0, 0.f);
        g_vertexPositions[nodeIndices.y] = (float4)(position1, 0.f);
    }
}
```

Returning to our batching

- 10 batches: 10 OpenCL kernel enqueues/dispatches
- 1/10 links per batch
- Low compute density per thread

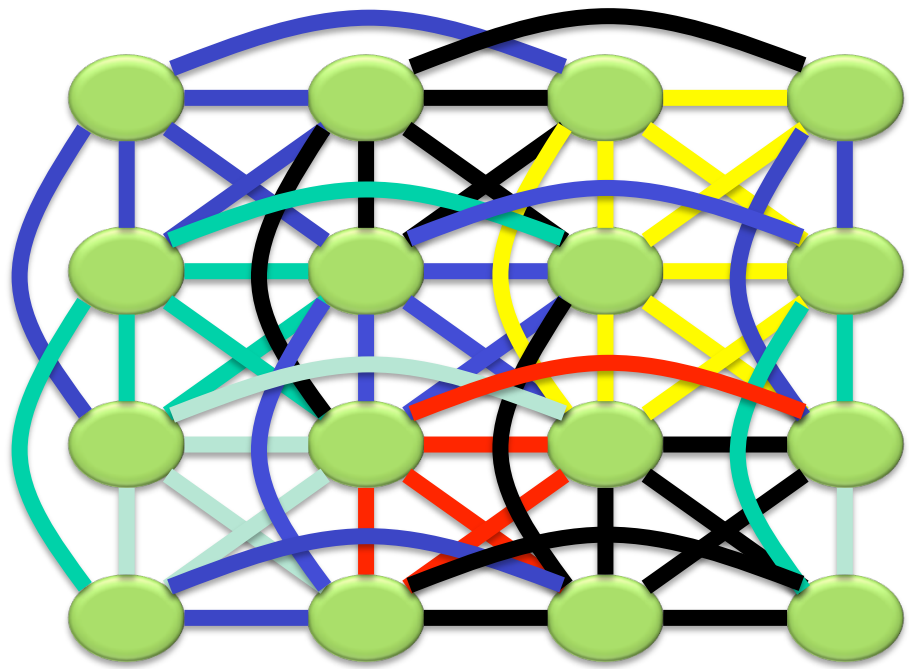


10 batches

© Copyright Khronos Group, 2010 - Page 51

Higher efficiency

- **Can create larger groups**
  - The cloth is fixed-structure
  - Can be preprocessed
- **Fewer batches/dispatches**

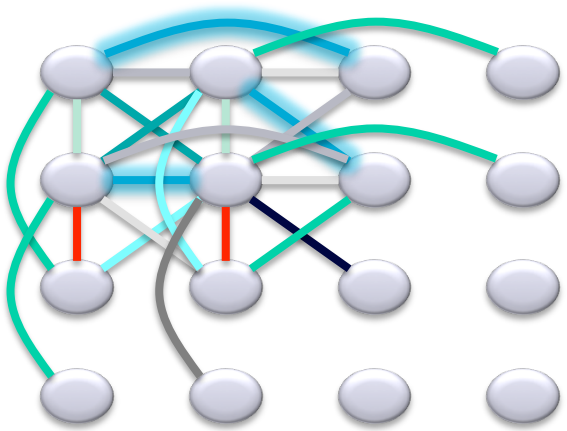


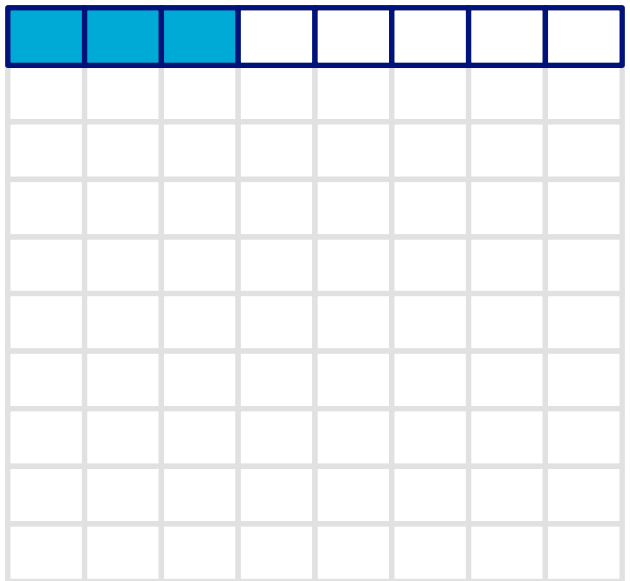
9 batches

© Copyright Khronos Group, 2010 - Page 52

Group execution

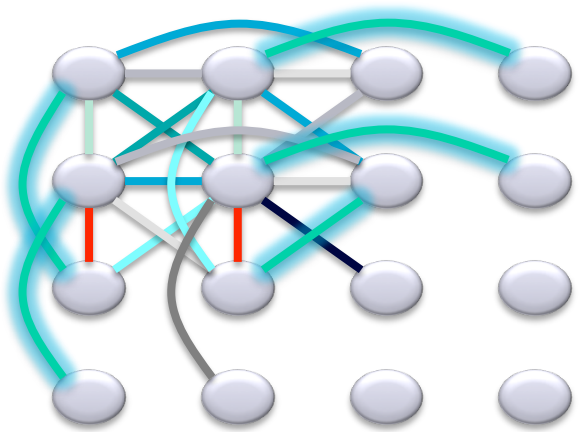
- The sequence of operations for the first batch is:

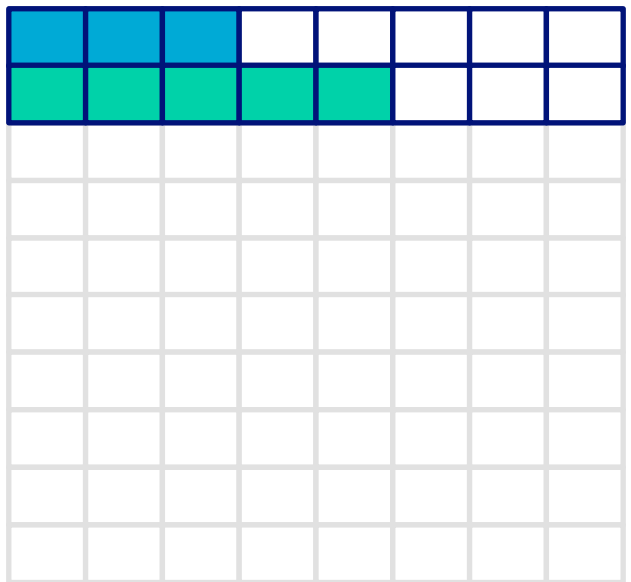




Group execution

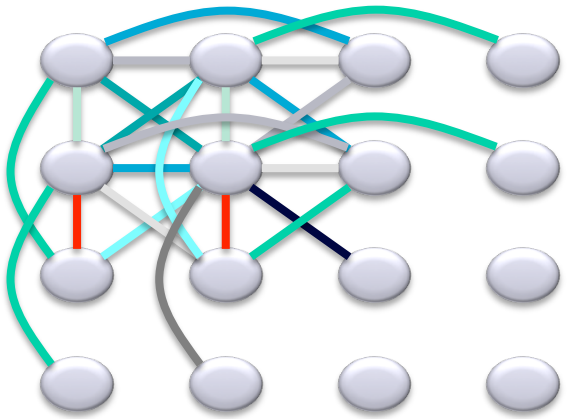
- The sequence of operations for the first batch is:

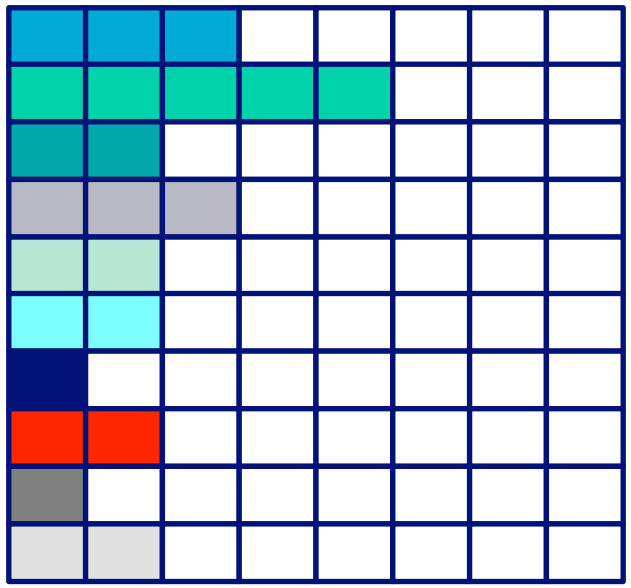




Group execution

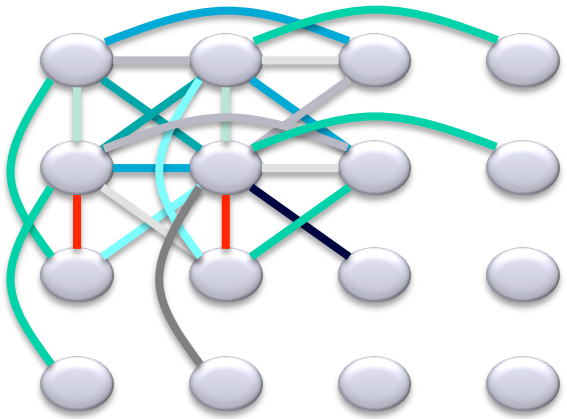
- The sequence of operations for the first batch is:



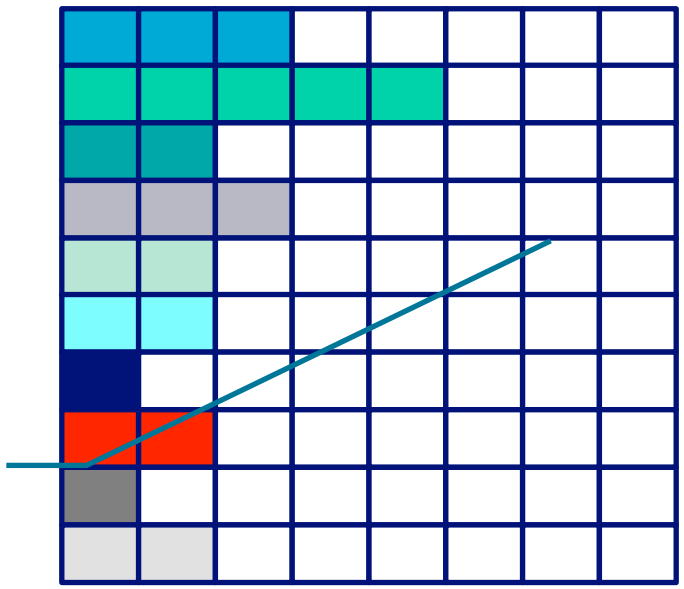


Group execution

- The sequence of operations for the first batch is:

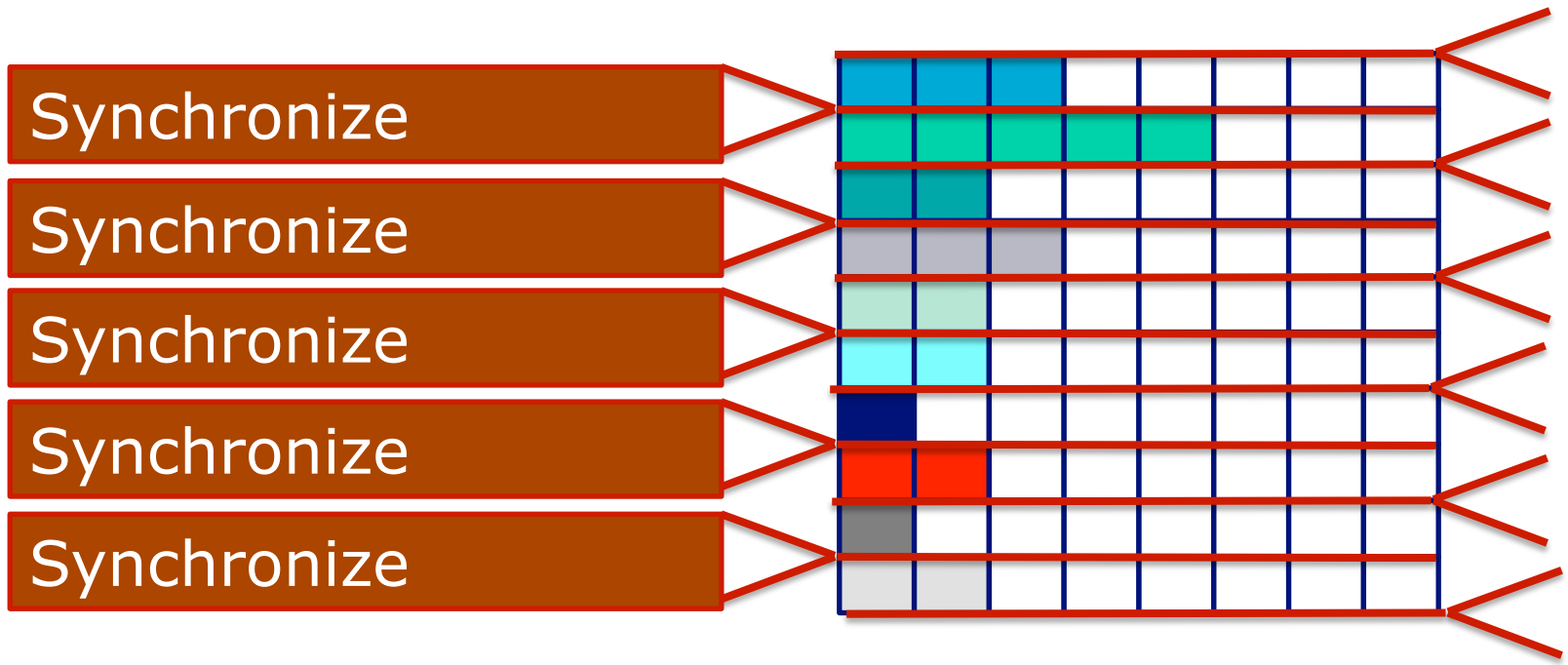


Few links so low packing efficiency:
Not a problem with larger cloth



Group execution

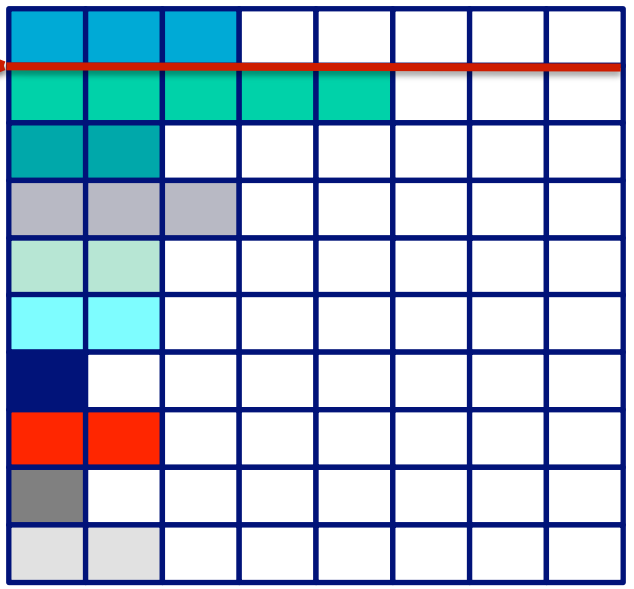
- The sequence of operations for the first batch is:



Group execution

- The sequence of operations for the first batch is:

Synchronize



```
// load
Barrier();

for( each subgroup ) {
  // Process a subgroup
  Barrier();
}

// Store
```

Why is this an improvement?

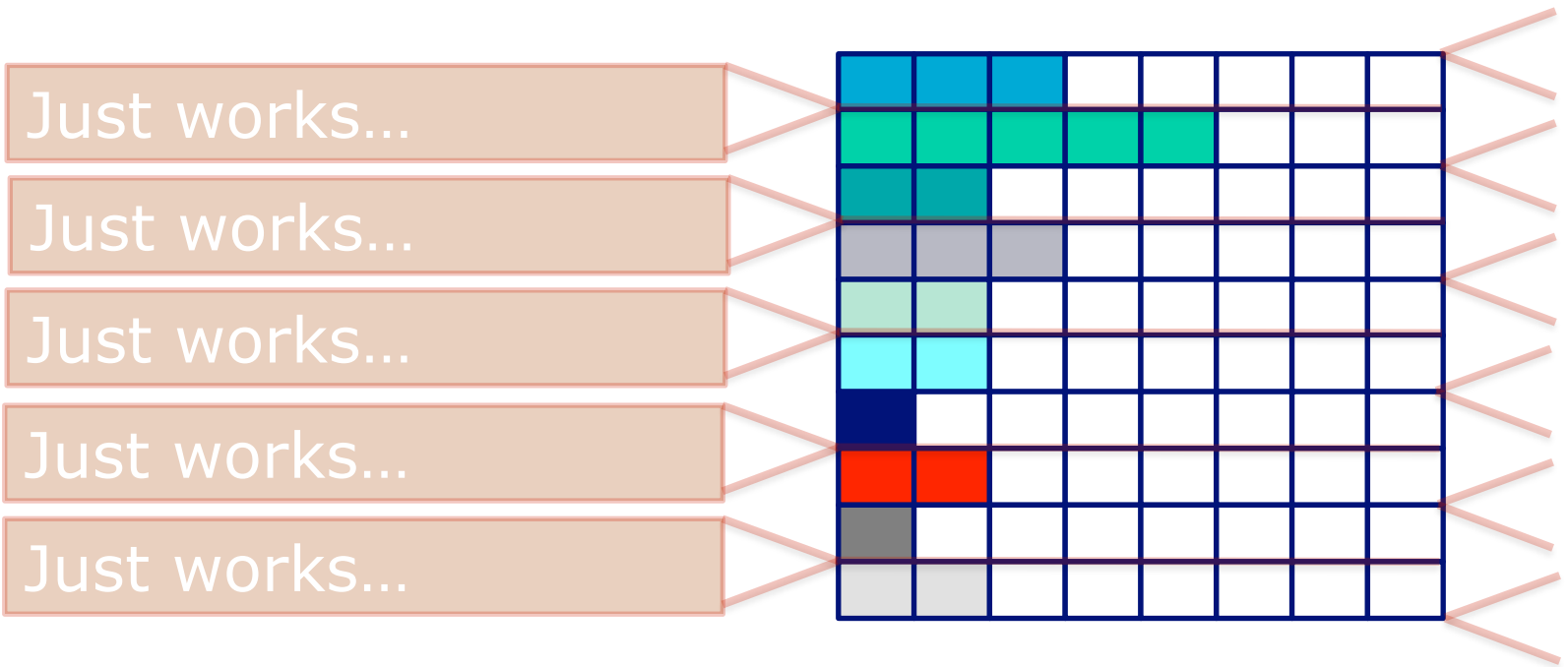
- So we still need same number batches. What have we gained?
  - The batches within a group chunk are in-shader loops
  - Only 4 shader dispatches, each with significant overhead
- The barriers will still hit performance
  - We are no longer dispatch bound, but we are likely to be on-chip synchronization bound

Exploiting the SIMD architecture

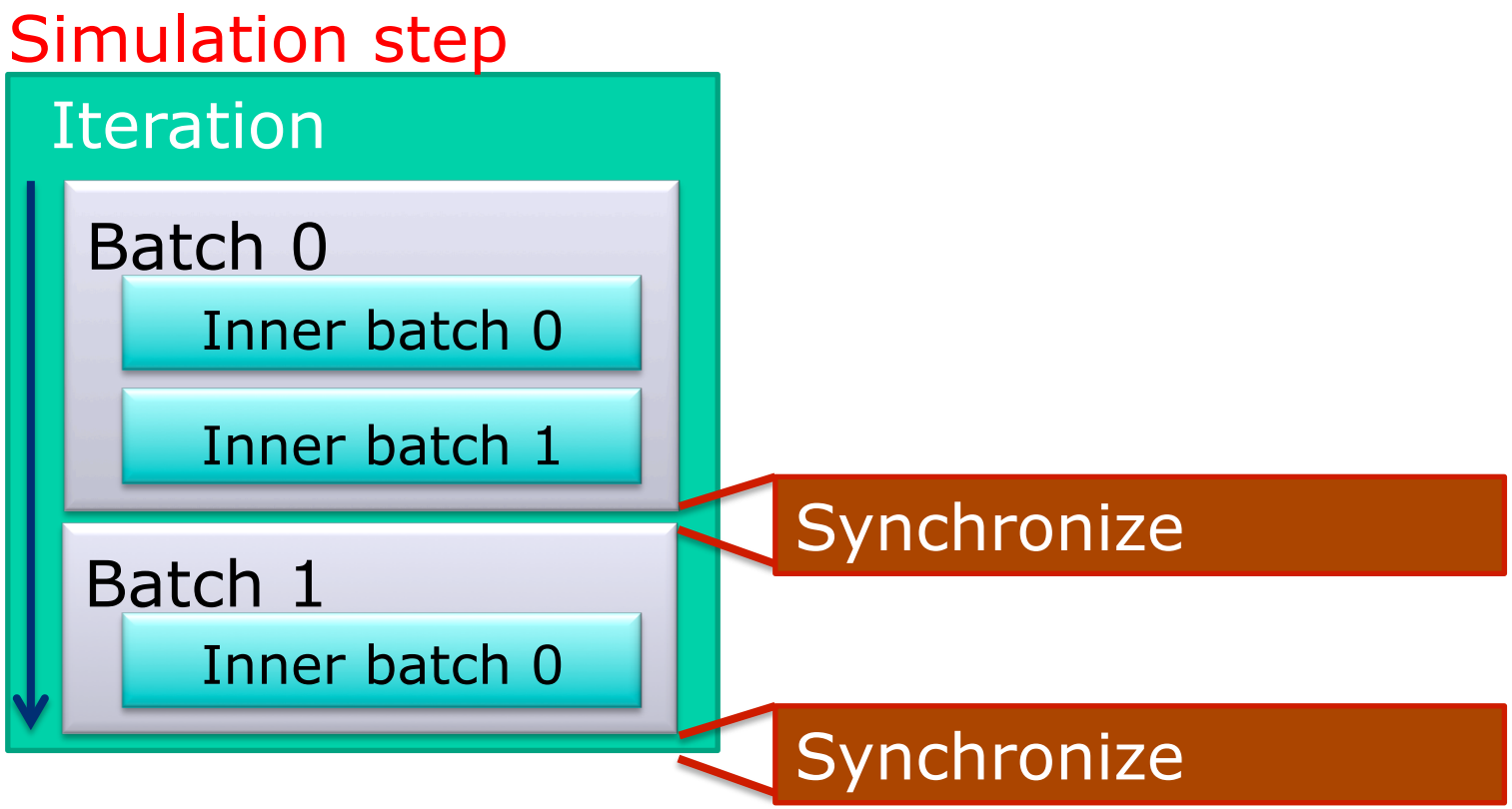
- Hardware executes 64- or 32-wide SIMD
- Sequentially consistent at the SIMD level
- Synchronization is now implicit
  - Take care
  - Execute over groups that are SIMD width or a divisor thereof

Group execution

- The sequence of operations for the first batch is:



Driving batches and synchronizing

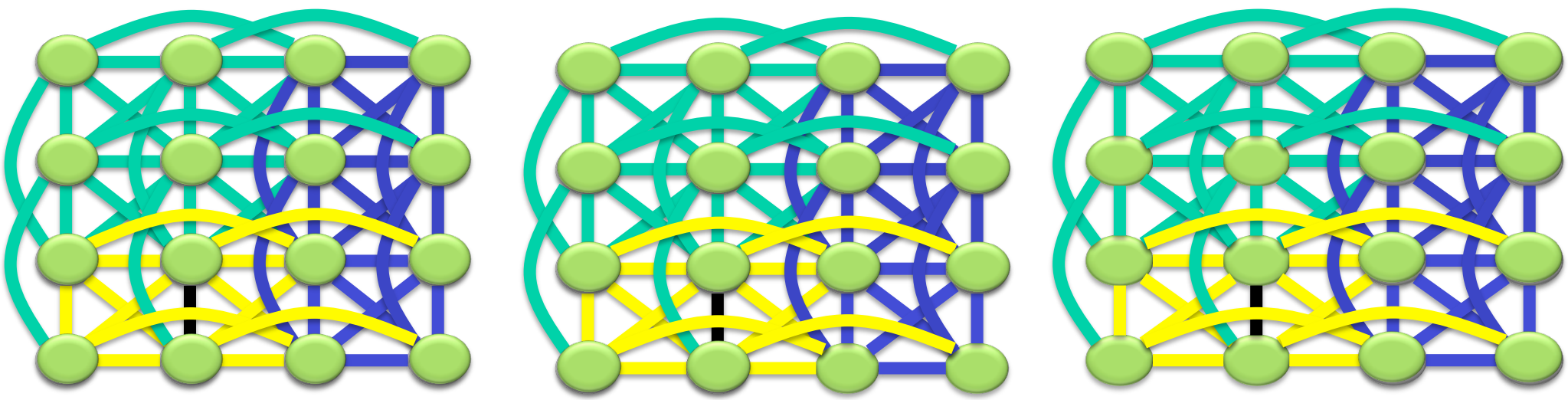


Performance gains

- 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)

Solving cloths together

- **Solve multiple cloths together in n batches**
- **Grouping**
  - Larger dispatches and reduced number of dispatches
  - Regain the parallelism that increased work-per-thread removed



© Copyright Khronos Group, 2010 - Page 64

Bullet Cloth

- **OpenCL and DirectCompute versions of cloth solver**
  - http://code.google.com/p/bullet/

© Copyright Khronos Group, 2010 - Page 65