エンターテインメントの未来がここにある
Compile -Future Entertainment-
CEDEC
CESA Developers Conference
2010

Advanced DirectX® 11 technology: DirectCompute by Example

Justin Hensley

Lee Howes

Advanced Micro Devices Inc.

September 2nd 2010

DirectX 11 Basics

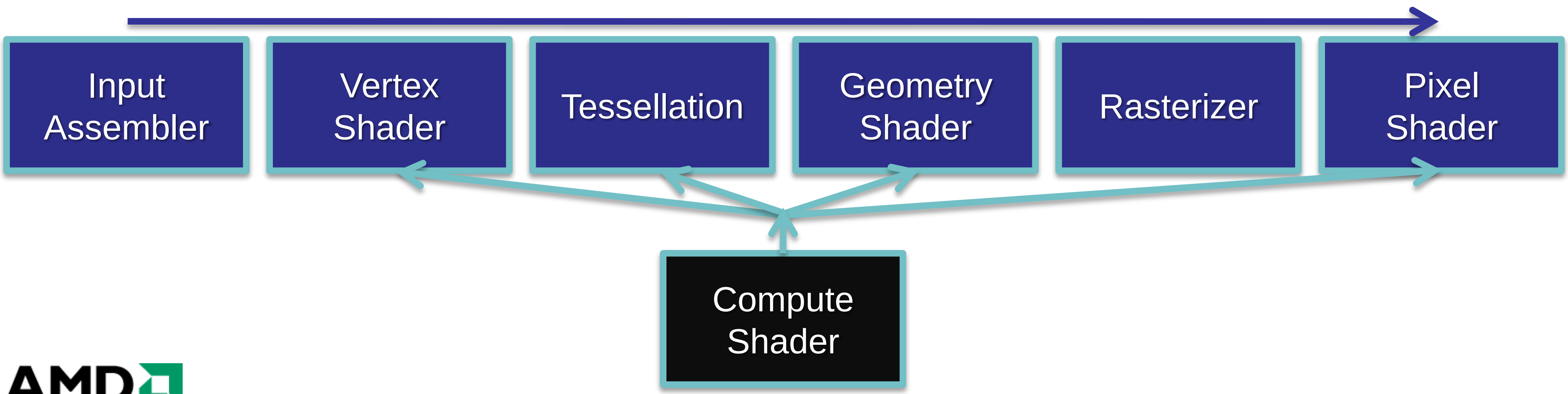
- New API from Microsoft
  - Released alongside Windows® 7
  - Runs on Windows Vista® as well
- Supports downlevel hardware
  - **DirectX9**, **DirectX10**, **DirectX11**-class HW supported
  - Exposed features depend on GPU
- Allows the use of the same API for multiple generations of GPUs
  - However Windows Vista/Windows 7 required
- Lots of new features…

What is DirectCompute?

- DirectCompute brings GPGPU to DirectX
- DirectCompute is both separate from and integrated with the DirectX graphics pipeline
  - Compute Shader
  - Compute features in Pixel Shader
- Potential applications (応用分野)
  - Physics
  - AI
  - Image processing

DirectCompute – part of DirectX

- DirectX 11 helps efficiently combine Compute work with graphics
  - Sharing of buffers is trivial
  - Work graph is scheduled efficiently by the driver



DirectCompute Features

- Scattered writes
- Atomic operations
- Append/consume buffer
- Shared memory (local data share)
- Structured buffers
- Double precision (if supported)

DirectCompute by Example

- Order Independent Transparency (OIT)
  - Atomic operations
  - Scattered writes
  - Append buffer feature

- Bullet Cloth Simulation
  - Shared memory
  - Shared compute and graphics buffers

Order Independent Transparency

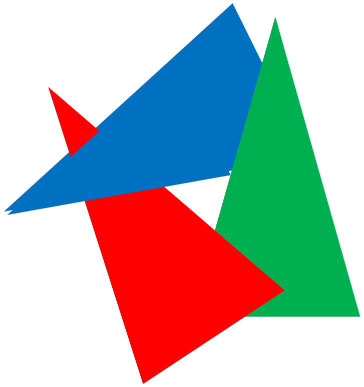
Transparency Problem

- Classic problem in computer graphics
- Correct rendering of semi-transparent geometry requires sorting – blending is an order dependent operation

  半透明な物体の正確なレンダリングはそれぞれのトライアングルの視点からの距離でのソートが必要
- Sometimes sorting triangles is enough but not always

  しかしトライアングルのソートでは不十分なときがある
  - Difficult to sort: Multiple meshes interacting (many draw calls)
  - Impossible to sort: Intersecting triangles (must sort fragments)

パワーポイントでこれをどうやって作るか知ってるかい?大変だったんだ、これが。



Try doing this in PowerPoint!

Background

- A-buffer – Carpenter ‘84
  - CPU side linked list per-pixel for anti-aliasing
- Fixed array per-pixel
  - F-buffer, stencil routed A-buffer, Z^3 buffer, and k-buffer, Slice map, bucket depth peeling
- Multi-pass
  - Depth peeling methods for transparency
- Recent
  - Freepipe, PreCalc [DirectX11 SDK]

OIT using Per-Pixel Linked Lists

- Fast creation of linked lists of arbitrary size on the GPU using D3D11
  - Computes correct transparency
    正しい半透明の描画が可能
- Integration into the standard graphics pipeline
  - Demonstrates **compute** from **rasterized data**
  - DirectCompute features in Pixel Shader
  - Works with depth and stencil testing
  - Works with and without MSAA
- Example of programmable blend

Linked List Construction

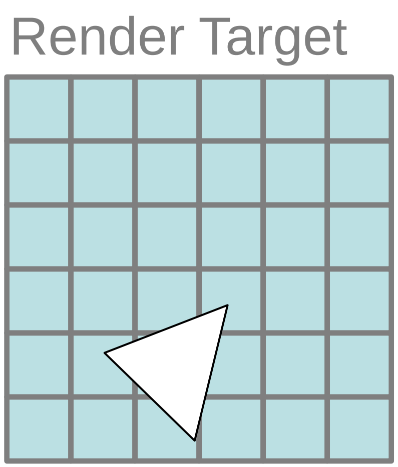
- Two Buffers
  - Head pointer buffer
    - addresses/offsets
    - Initialized to end-of-list (EOL) value (e.g., -1)
  - Node buffer
    - arbitrary payload data + “next pointer”
      任意のデータと次のデータへのポインタ
- Each shader thread
  1. Retrieve and increment global counter value
     グローバルのカウンターを取り出し、インクリメント
  2. Atomic exchange into head pointer buffer
     ヘッドのポインタバッファをアトミックエクスチェンジ
  3. Add new entry into the node buffer at location from step 1
     新しいデータをステップ1での場所に書き込む

Algorithm Overview

0. Render opaque scene objects
1. Render transparent scene objects
2. Screen quad resolves and composites fragment lists

Step 0 – Render Opaque

- Render all opaque geometry normally
  不透明物体の描画を普通に行う



Algorithm Overview

0. Render opaque scene objects
1. Render transparent scene objects

   半透明物体の描画

   – All fragments are stored using per-pixel linked lists

     全てのフラグメントはピクセルごとのリンクリストとして書き込まれる

   – Store fragment's: color, alpha, & depth
2. Screen quad resolves and composites fragment lists

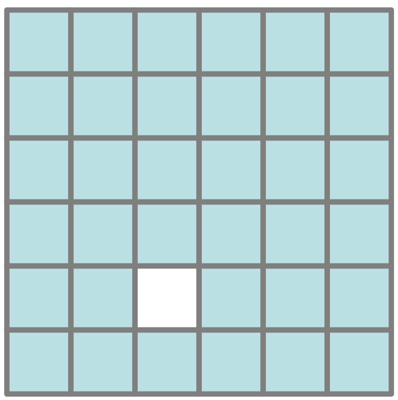
Setup

- Two buffers
  - Screen sized head pointer buffer
    スクリーンサイズのヘッドポインタバッファ
  - Node buffer – large enough to handle all fragments
    ノードバッファ（全てのフラグメントを格納するのに十分な大きさ）

- Render as usual
  不透明物体を普通に描画
- Disable render target writes
  レンダーターゲットへの書き込みを無効化
- Insert render target data into linked list
  半透明物体のデータをリンクリストに書き込み

Step 1 – Create Linked List

Render Target

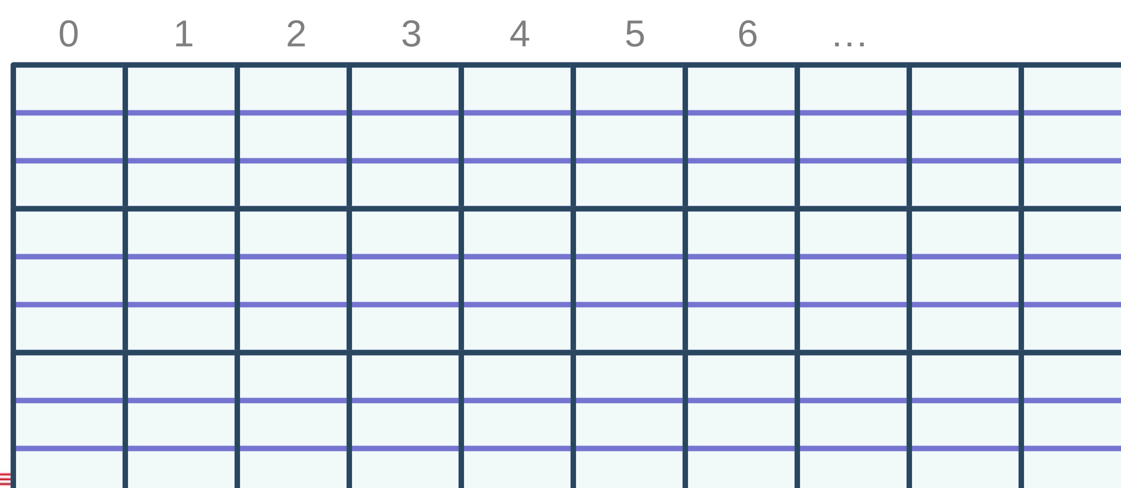


Head Pointer Buffer

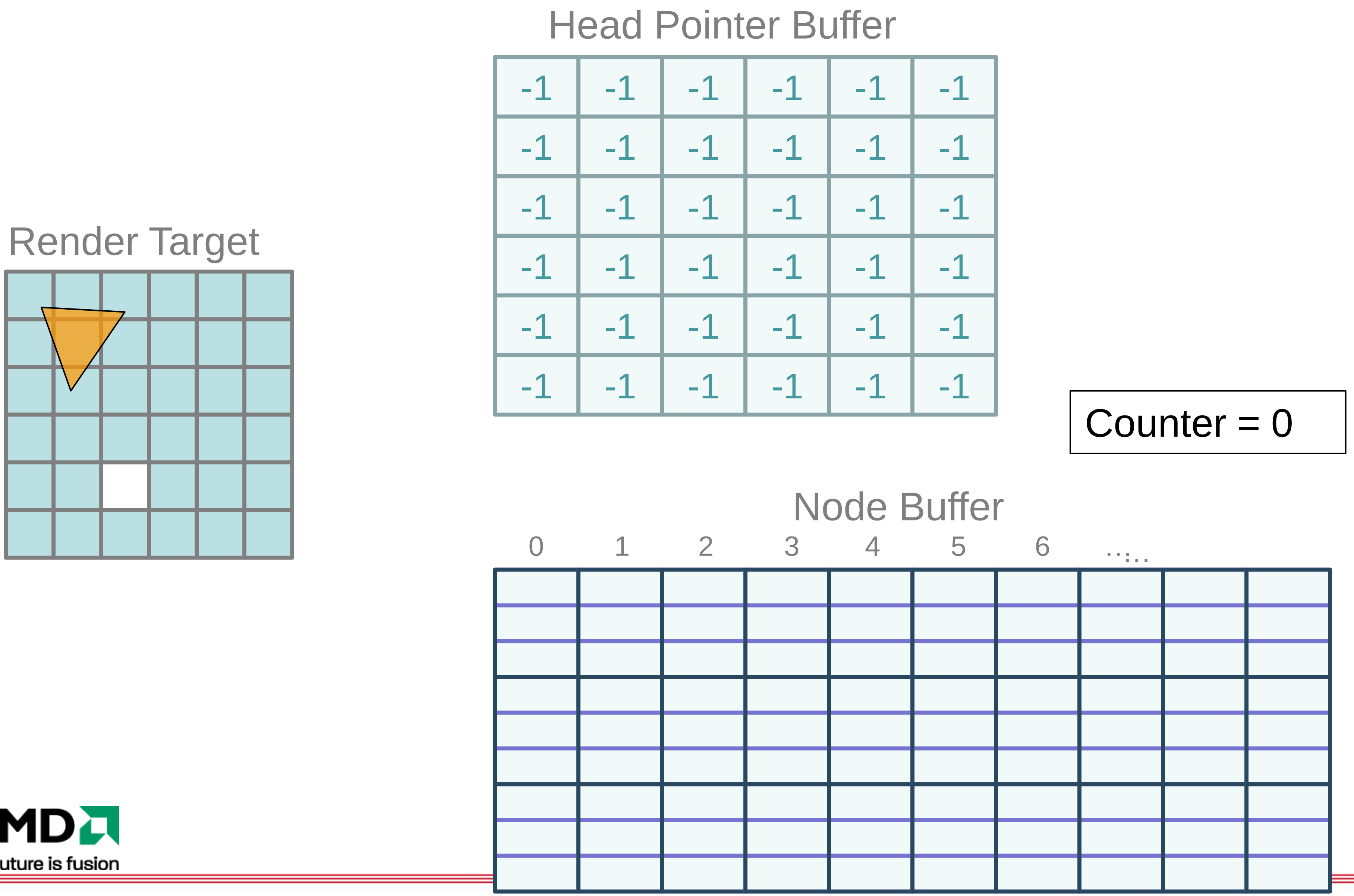
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

Counter = 0

Node Buffer

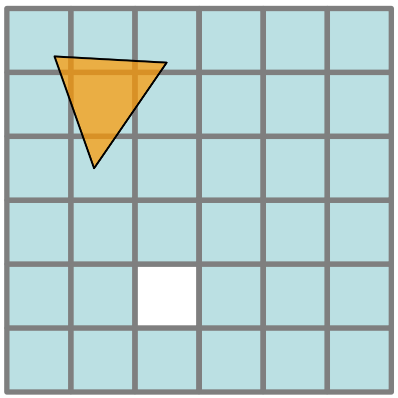


Step 1 – Create Linked List



Step 1 – Create Linked List

Render Target



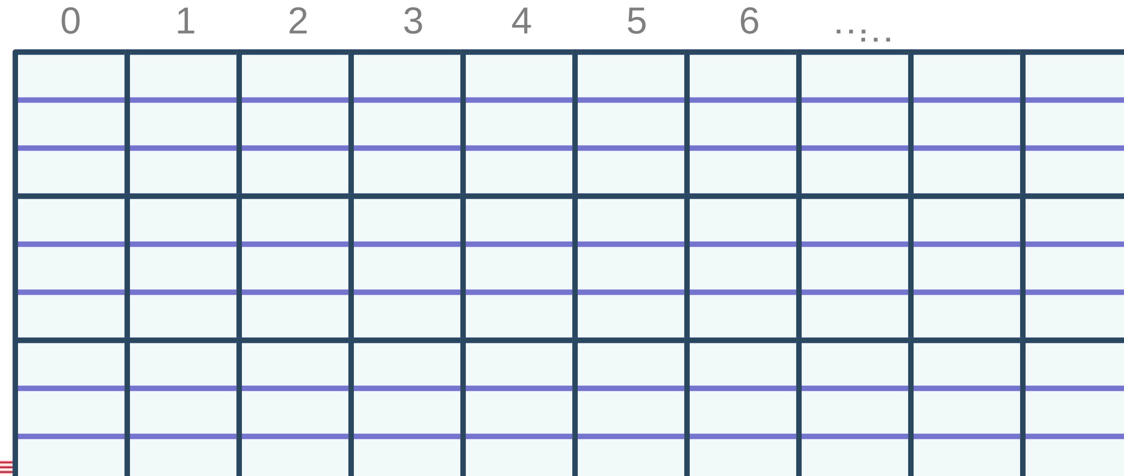
Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

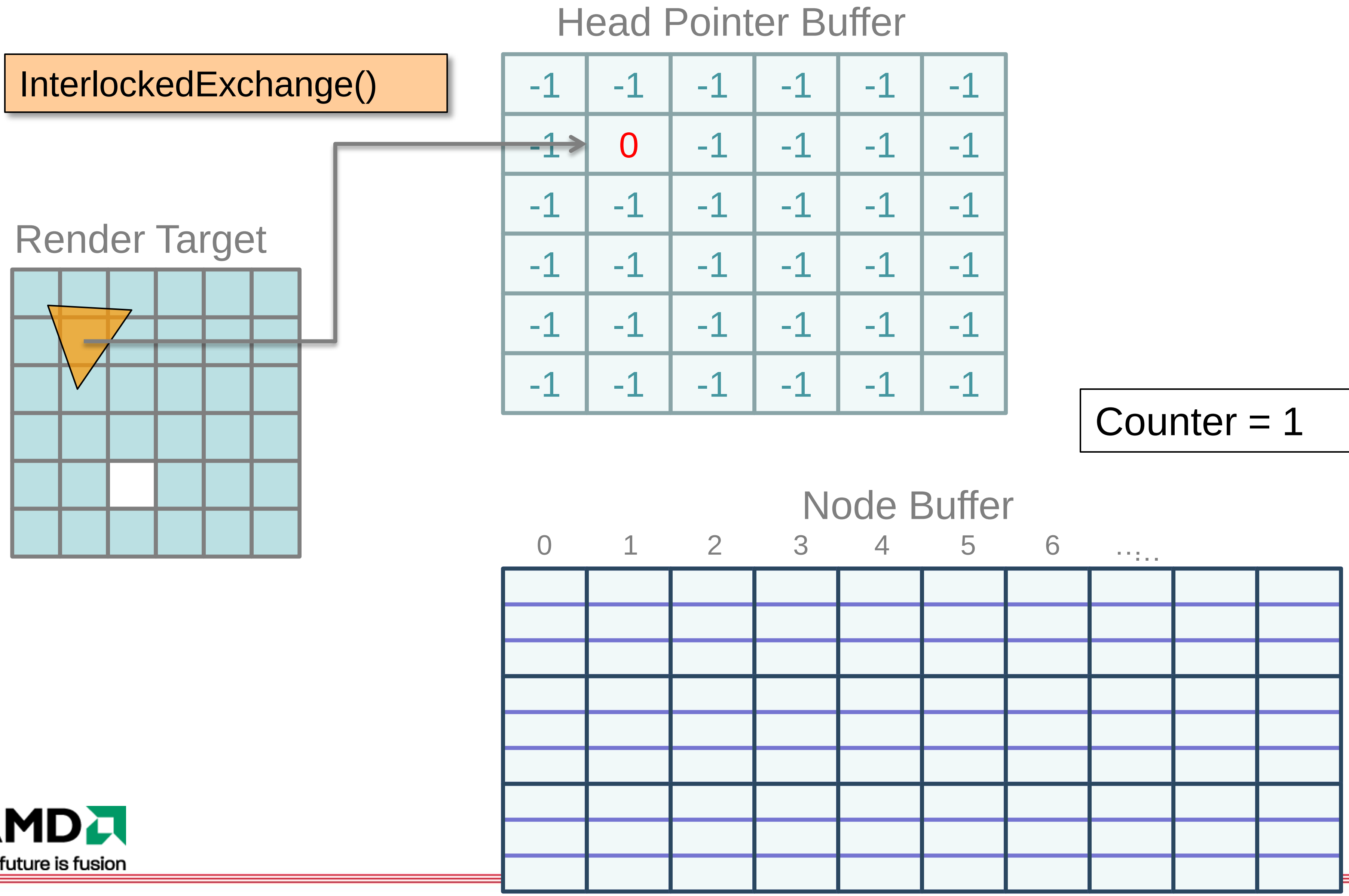
IncrementCounter()

Counter = 1

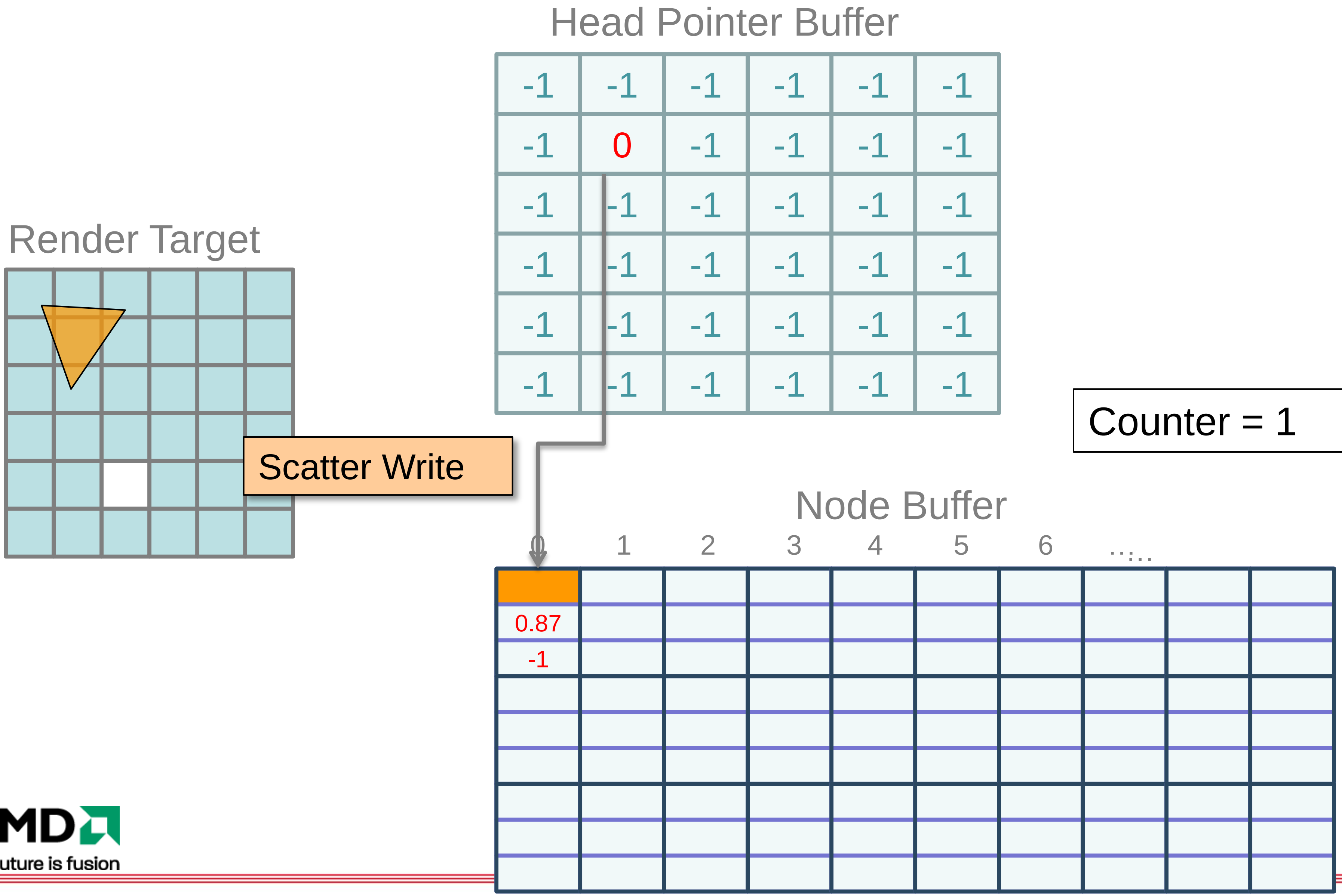
Node Buffer



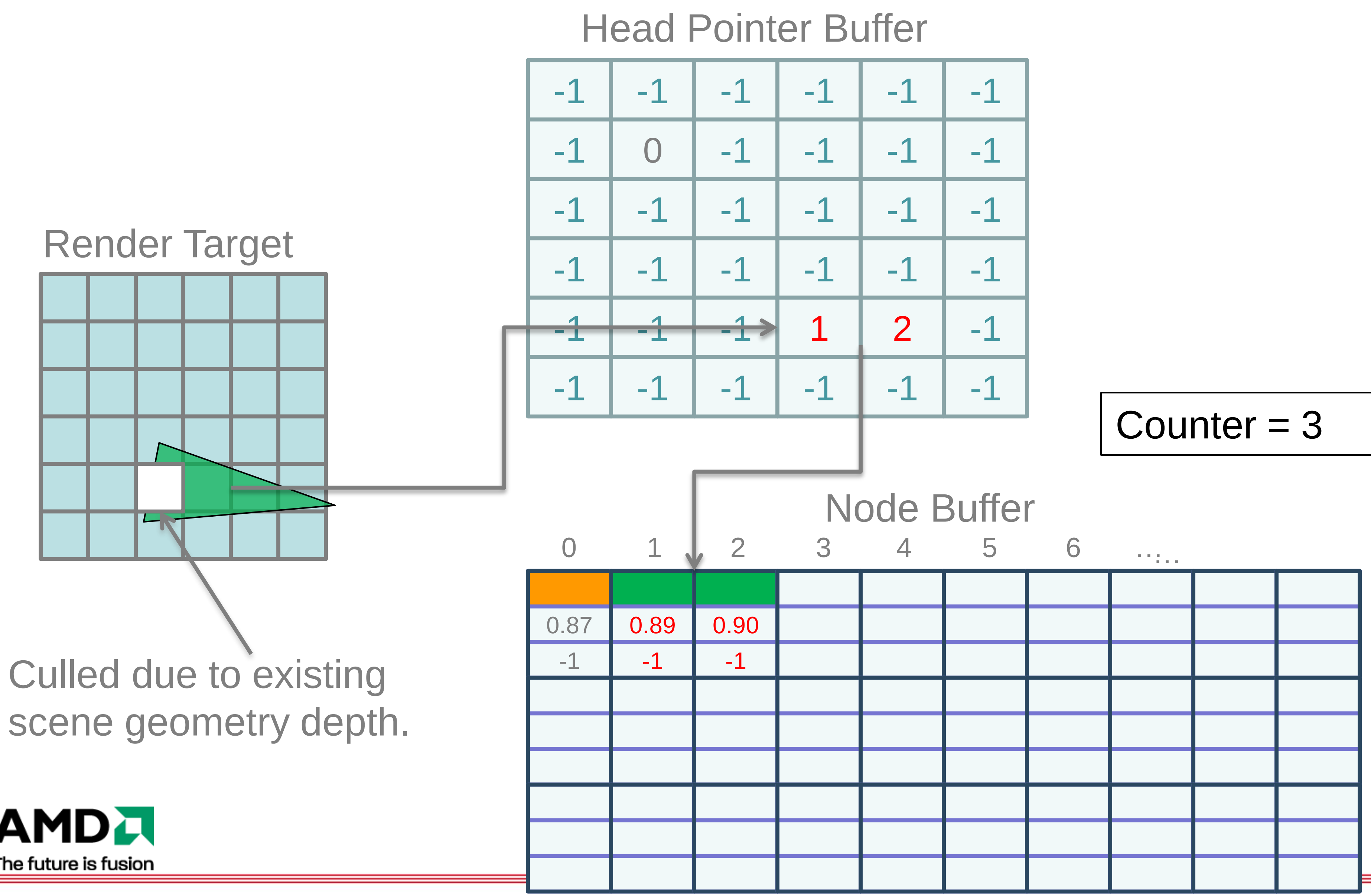
Step 1 – Create Linked List



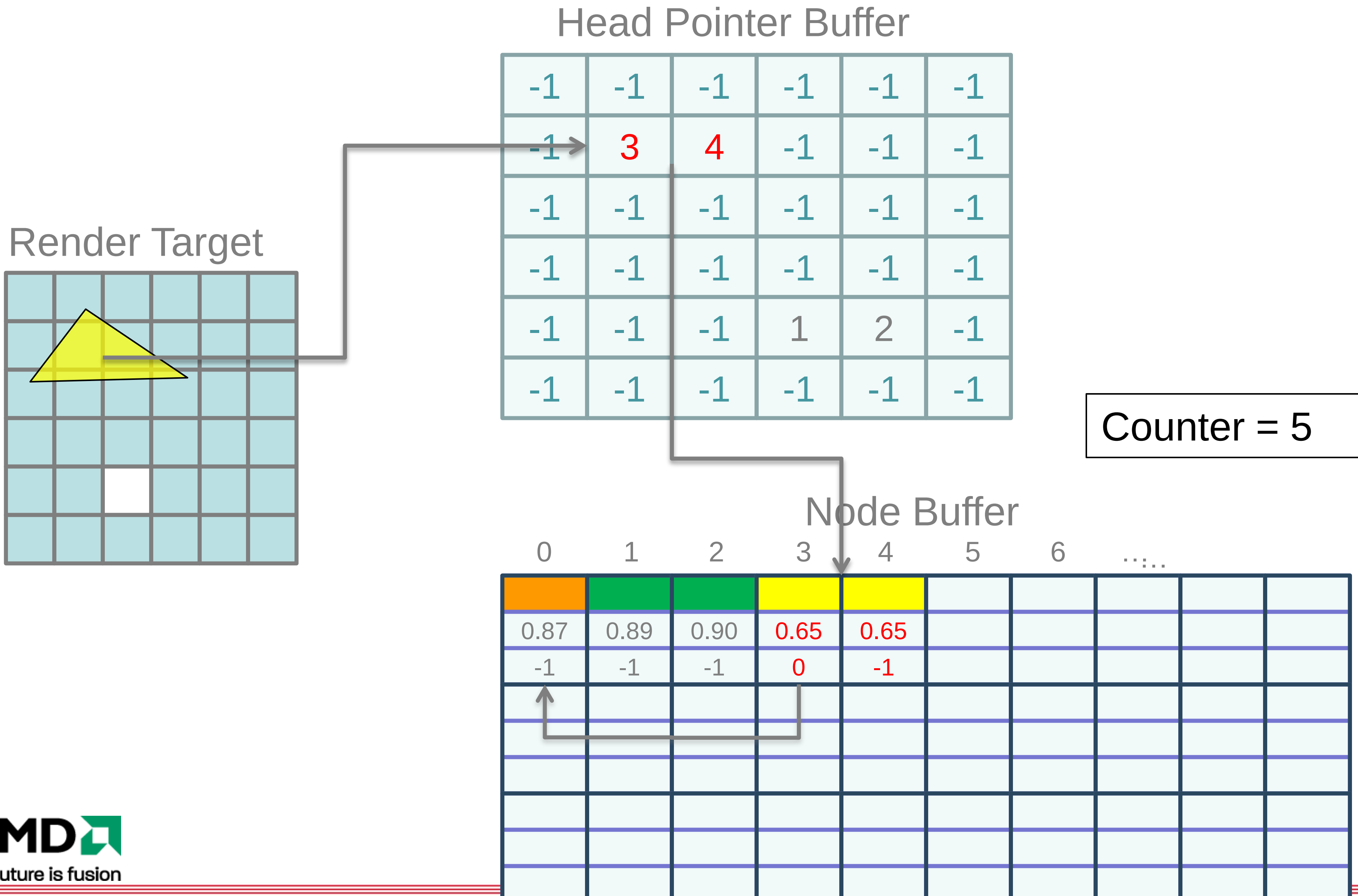
Step 1 – Create Linked List



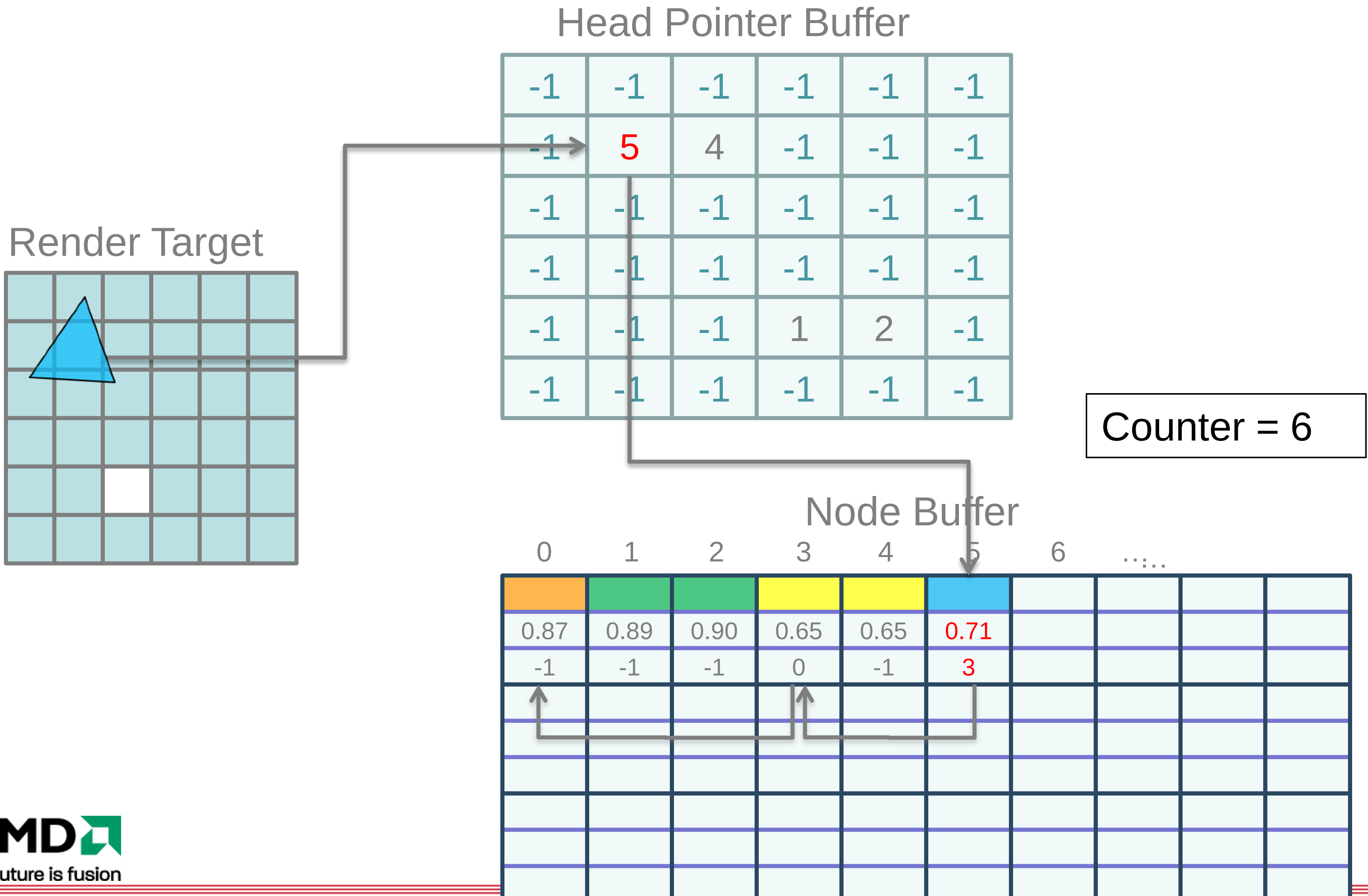
Step 1 – Create Linked List



Step 1 – Create Linked List



Step 1 – Create Linked List



The future is fusion

Node Buffer Counter

- Counter allocated in GPU memory (i.e. a buffer)
  - Atomic updates
  - Contention issues
- DirectX11 Append feature
  - Compacted linear writes to a buffer
  - Implicit writes
    - Append()
  - Explicit writes
    - IncrementCounter()
    - Standard memory operations
  - Up to 60% faster than memory counters

Code Example

```
RWStructuredBuffer    RWStructuredCounter;
RWTexture2D<int>      tRWFragmentListHead;
RWTexture2D<float4>   tRWFragmentColor;
RWTexture2D<int2>     tRWFragmentDepthAndLink;

[earlydepthstencil]
void PS( PsInput input )
{
    float4 vFragment = ComputeFragmentColor(input);
    int2   vScreenAddress = int2(input.vPositionSS.xy);

    // Get counter value and increment
    int nNewFragmentAddress = RWStructuredCounter.IncrementCounter();
    if ( nNewFragmentAddress == FRAGMENT_LIST_NULL ) { return; }

    // Update head buffer
    int nOldFragmentAddress;
    InterlockedExchange(tRWFragmentListHead[vScreenAddress], nNewHeadAddress,
    nOldHeadAddress );

    // Write the fragment attributes to the node buffer
    int2 vAddress = GetAddress( nNewFragmentAddress );
    tRWFragmentColor[vAddress] = vFragment;
    tRWFragmentDepthAndLink[vAddress] = int2(
        int(saturate(input.vPositionSS.z))*0x7fffffff), nOldFragmentAddress );

    return;
}
```

The future is fusion

Algorithm Overview

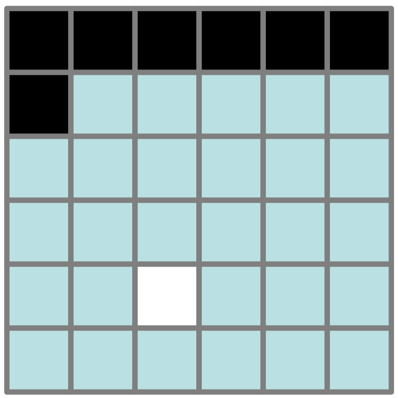
0. Render opaque scene objects
1. Render transparent scene objects
2. Screen quad resolves and composites fragment lists
   - Single pass
   - Pixel shader sorts associated linked list (e.g., insertion sort)
   - Composite fragments in sorted order with background
   - Output final fragment

Step 2 – Render Fragments

Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target

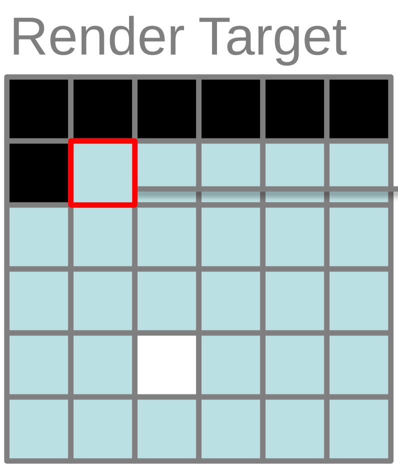


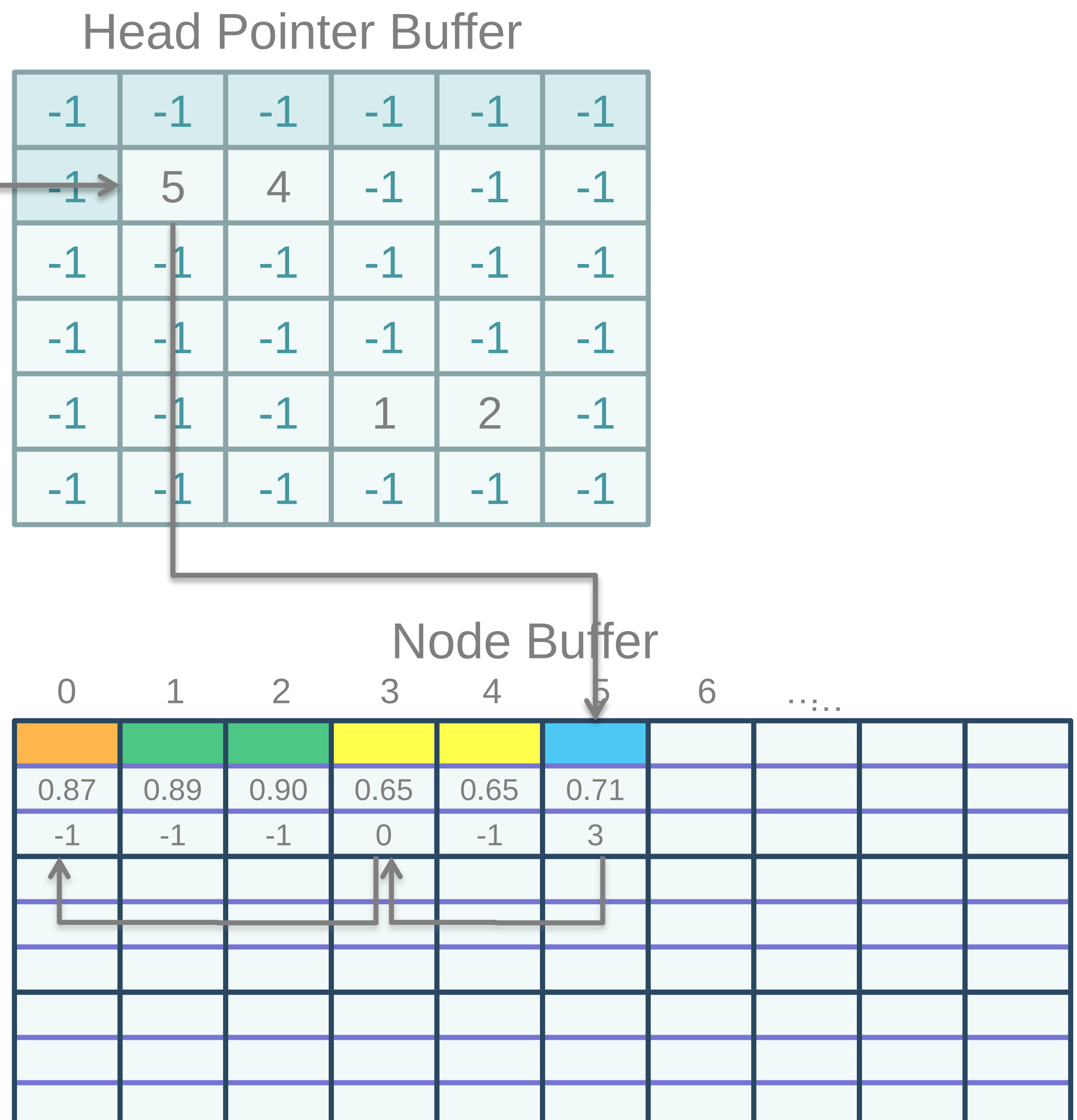
Node Buffer

0	1	2	3	4	5	6	……
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

(0,0)->(1,1):
Fetch Head Pointer: -1
-1 indicates no fragment to render

Step 2 – Render Fragments





(1,1):
Fetch Head Pointer: 5
Fetch Node Data (5)
Walk the list and store in temp array

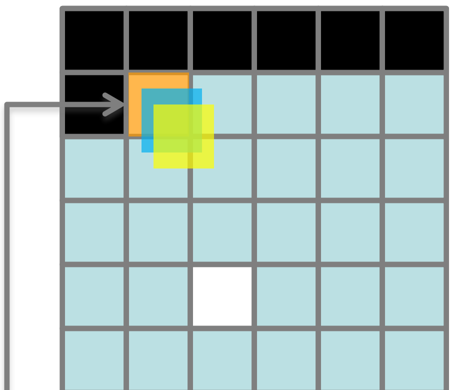
0.71	0.65	0.87

Step 2 – Render Fragments

Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target



(1,1):
Sort temp array
Blend colors and write out

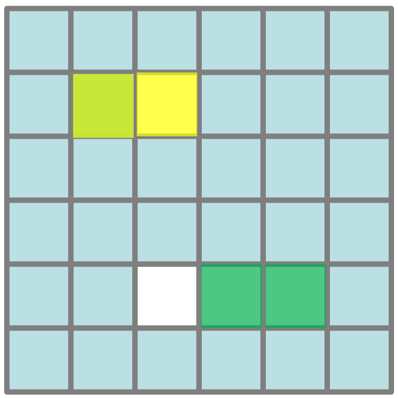
0.65	0.71	0.87

Node Buffer

0	1	2	3	4	5	6	……		
0.87	0.89	0.90	0.65	0.65	0.71				
-1	-1	-1	0	-1	3				

Step 2 – Render Fragments

Render Target



Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Node Buffer

0	1	2	3	4	5	6	…:..		
0.87	0.89	0.90	0.65	0.65	0.71				
-1	-1	-1	0	-1	3				

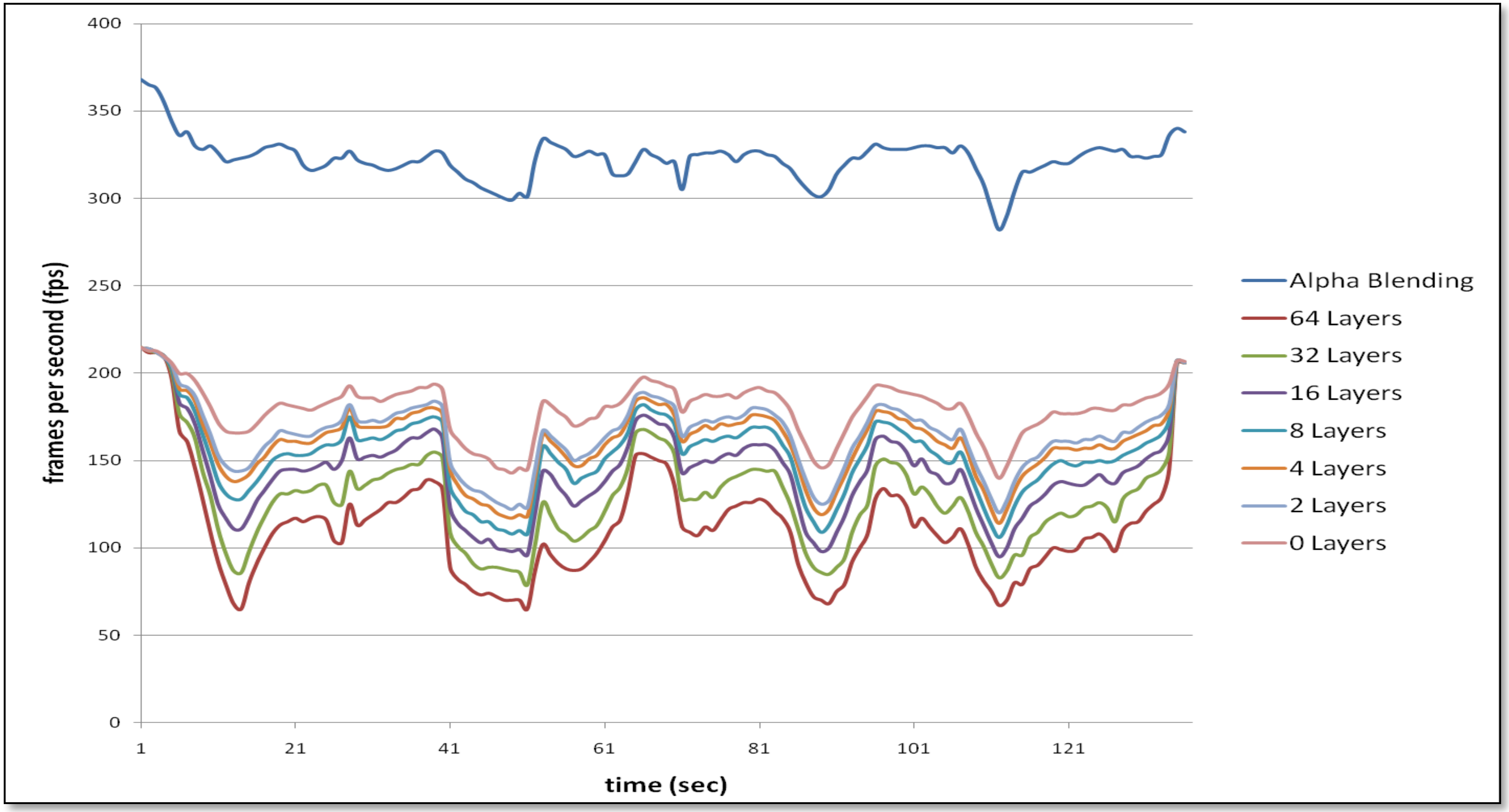
Anti-Aliasing

- Store coverage information in the linked list
- Resolve per-sample
  – Execute a shader at each sample location
  – Use MSAA hardware
- Resolve per-pixel
  – Execute a shader at each pixel location
  – Average all sample contributions within the shader

Mecha Demo

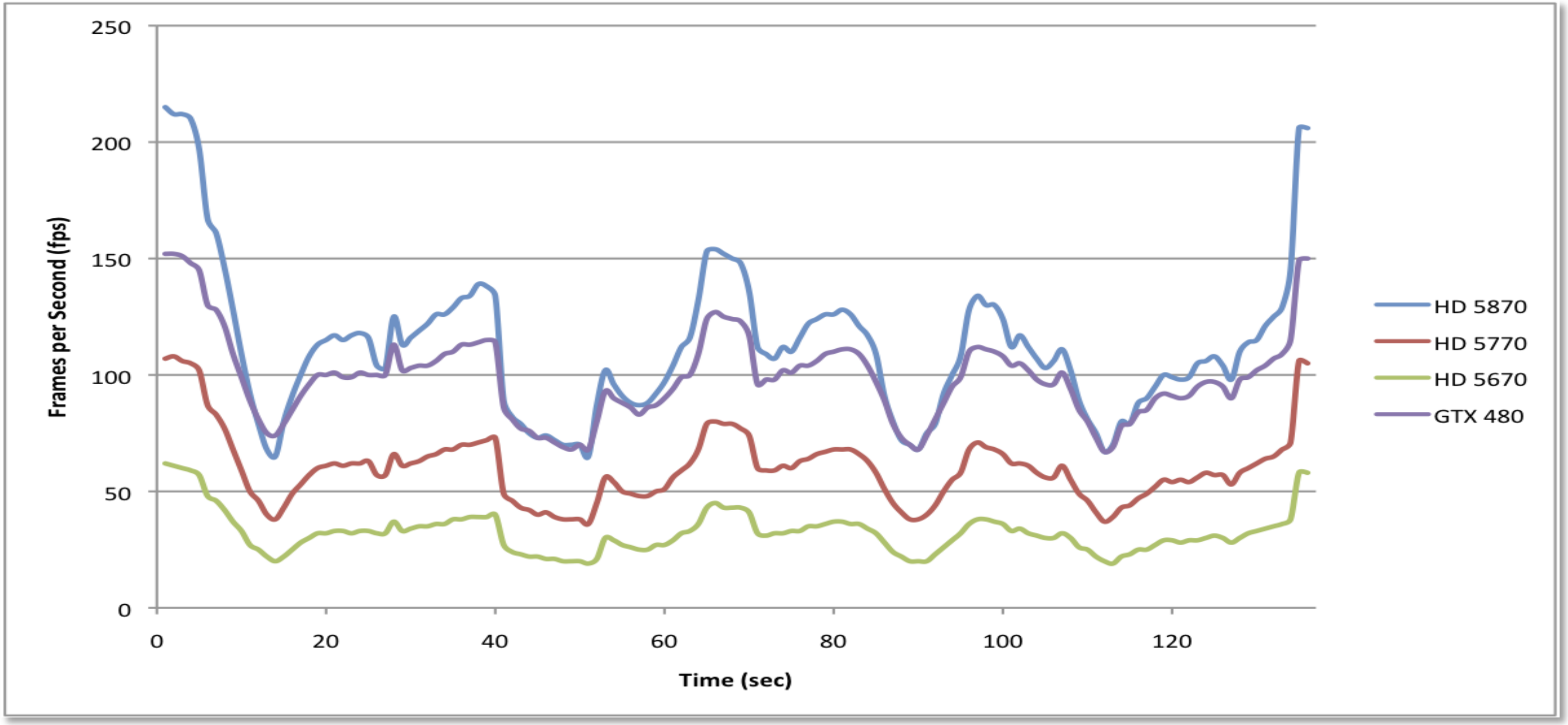
- 602K scene triangles
  - 254K transparent triangles

Layers



400
350
300
250
200
150
100
50
0
frames per second (fps)
1
21
41
61
81
101
121
time (sec)
Alpha Blending
64 Layers
32 Layers
16 Layers
8 Layers
4 Layers
2 Layers
0 Layers

Scaling



Frames per Second (fps)
250
200
150
100
50
0
0
20
40
60
80
100
120
Time (sec)
HD 5870
HD 5770
HD 5670
GTX 480

Future Work

- Memory allocation
- Sort on insert
- Other linked list applications
  - Indirect illumination
  - Motion blur
  - Shadows
- More complex data structures

Bullet Cloth Simulation

概要 Overview

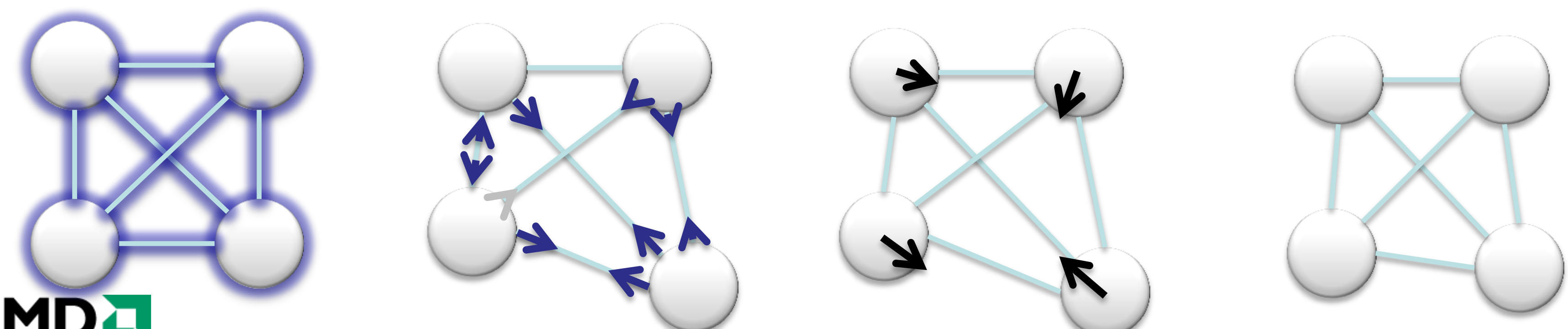
- 布のシミュレーションについて
- 2つの解法
  - バッチソルバ(Batched Solver)
  - SIMDバッチソルバ(SIMD Batched Solver)
- そしてもう一つ
  - データのGPU上での直接のコピー(GPU Copy)

DirectCompute for physics

- DirectCompute in the Bullet physics SDK
  - 布のシミュレーションの紹介
  - An introduction to cloth simulation
  - DirectComputeでの実装のティップス
  - Some tips for implementation in DirectCompute

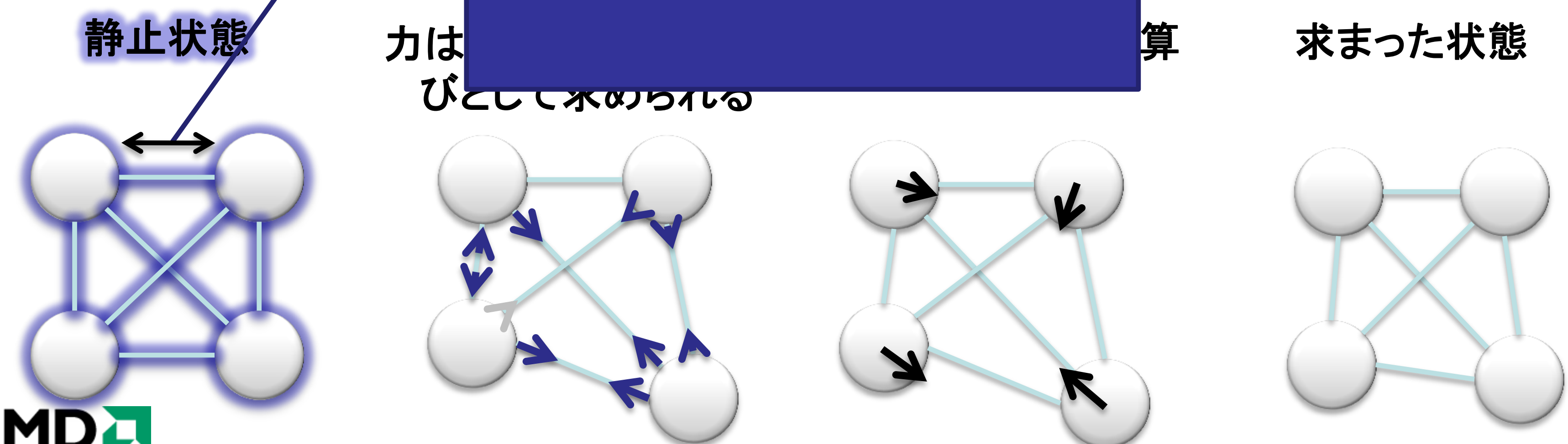
Cloth simulation

- Large number of particles
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles
- 大量のパーティクル
  - 並列処理に適している
  - それぞれのバネをつなぐパーティクルに拘束条件を適用



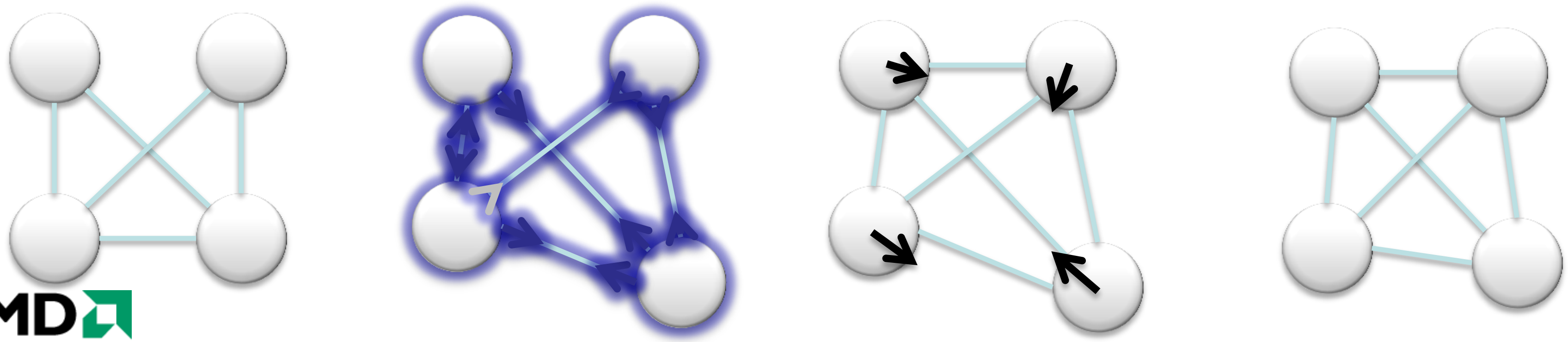
Cloth simulation

- Large number of particles
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles
- 大量のパーティクル
  - 並列処理に
  - それぞれの　束条件を適用



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

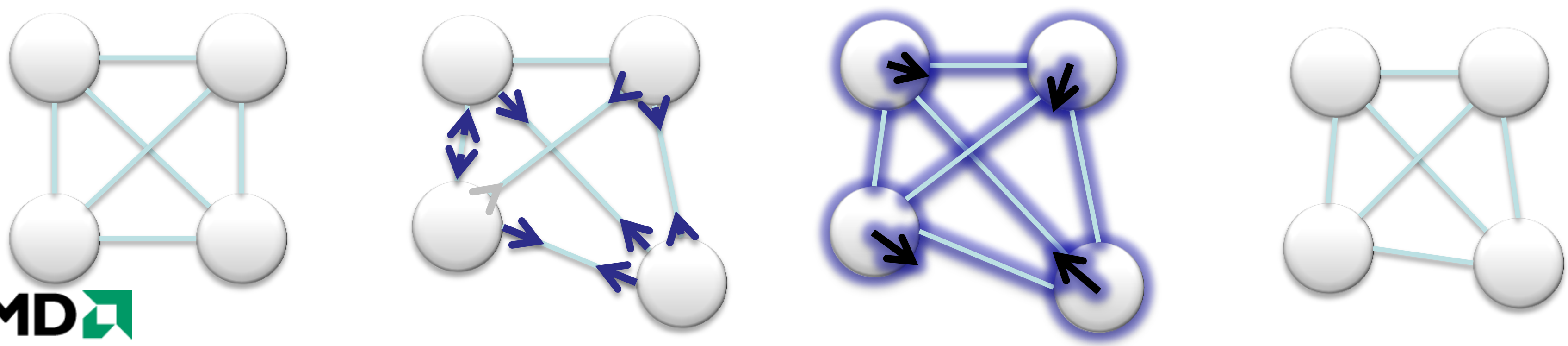
- For each simulation iteration:
  - Compute forces in each link based on its length
  - Correct positions of masses/vertices from forces
  - Compute new vertex positions

静止状態

現在の状態
力は静止状態からの伸びとして求められる

位置の補正量の計算

求まった状態



Cloth simulation steps

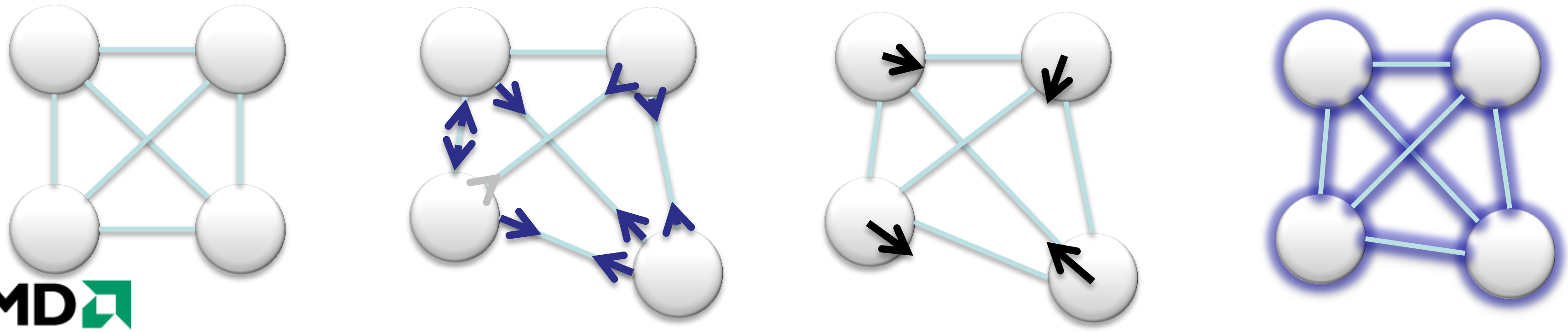
- For each simulation iteration:
  - Compute forces in each link based on its length
  - Correct positions of masses/vertices from forces
  - Compute new vertex positions

静止状態

現在の状態
力は静止状態からの伸びとして求められる

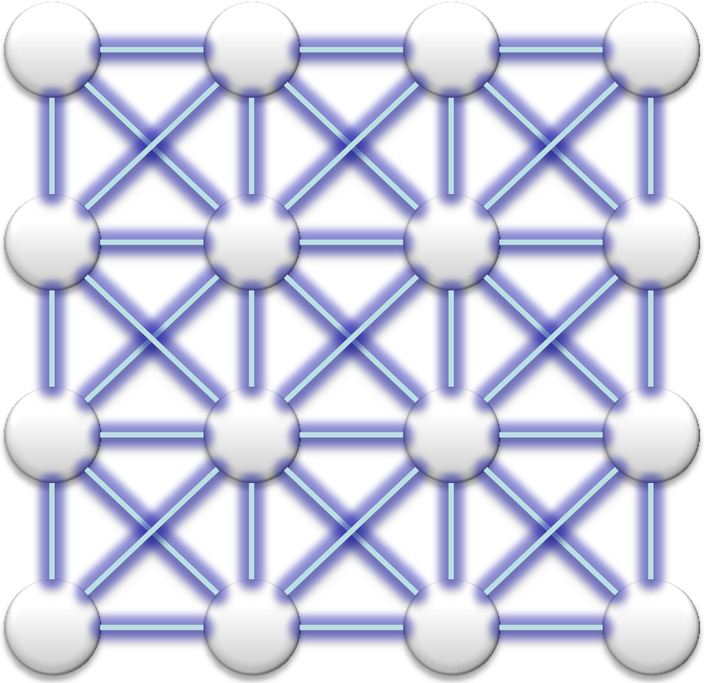
位置の補正量の計算

求まった状態



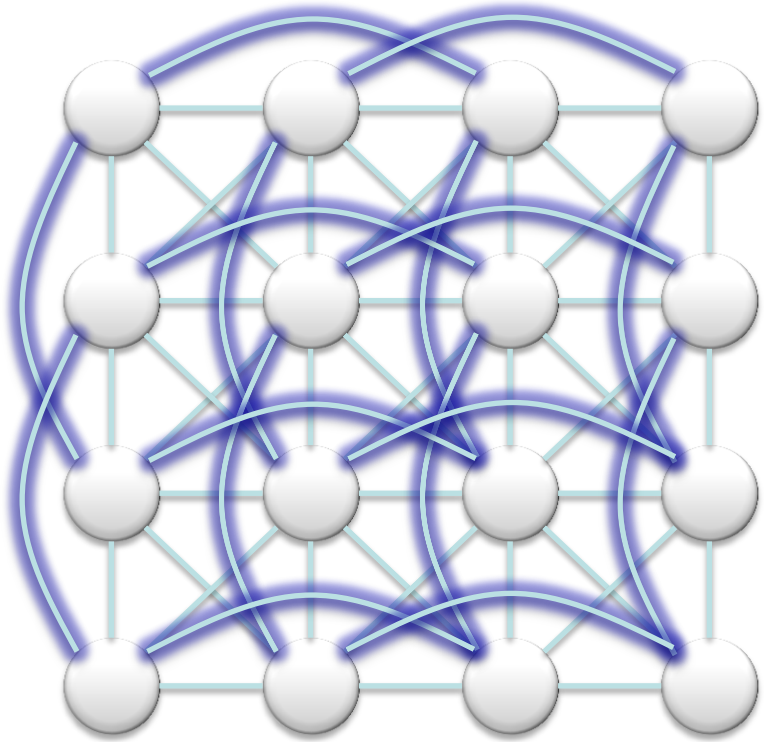
Springs and masses

- Two or three main types of springs
  - Structural/shearing 構造バネ
  - Bending 曲げバネ



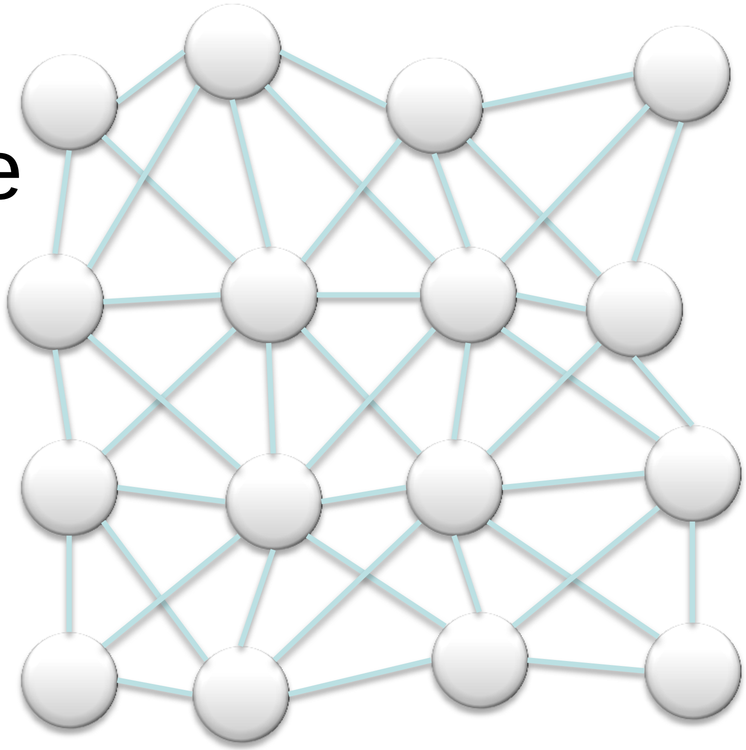
Springs and masses

- Two or three main types of springs
  - Structural/shearing 構造バネ
  - Bending 曲げバネ



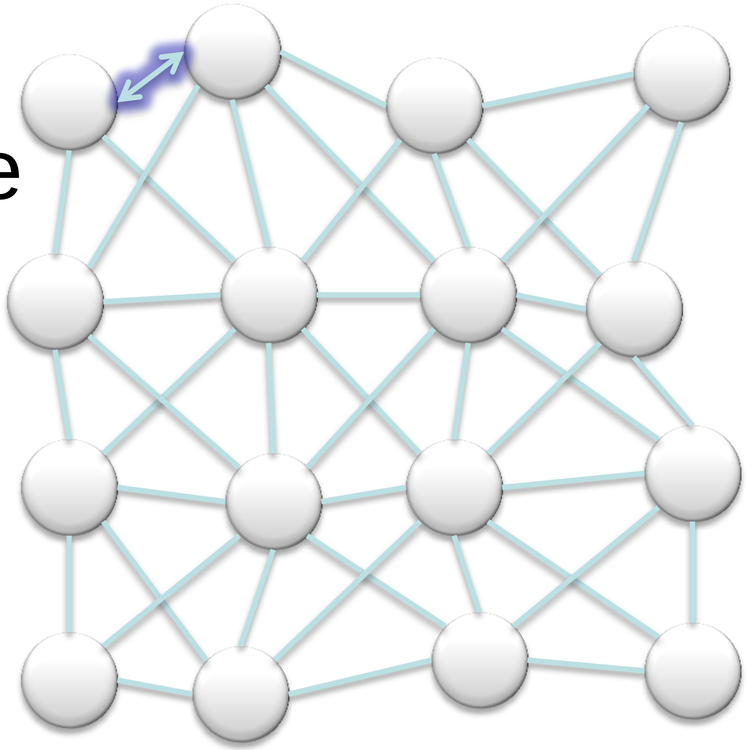
CPU approach to simulation

- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



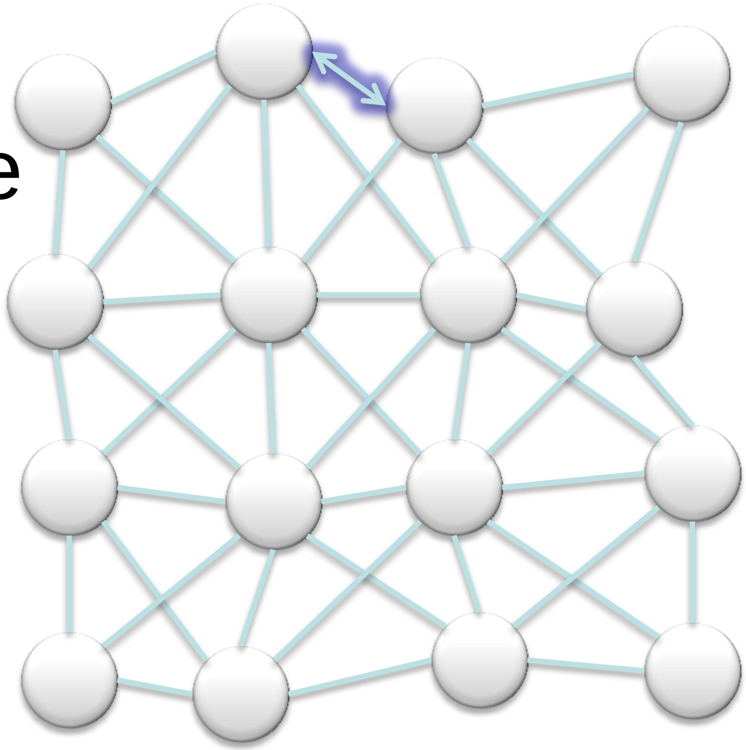
CPU approach to simulation

- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



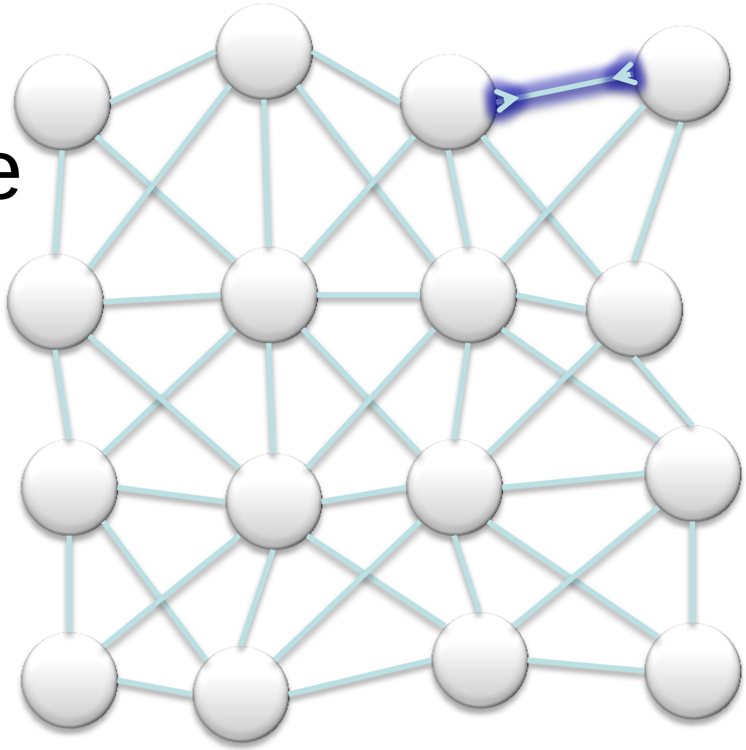
CPU approach to simulation

- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



CPU approach to simulation

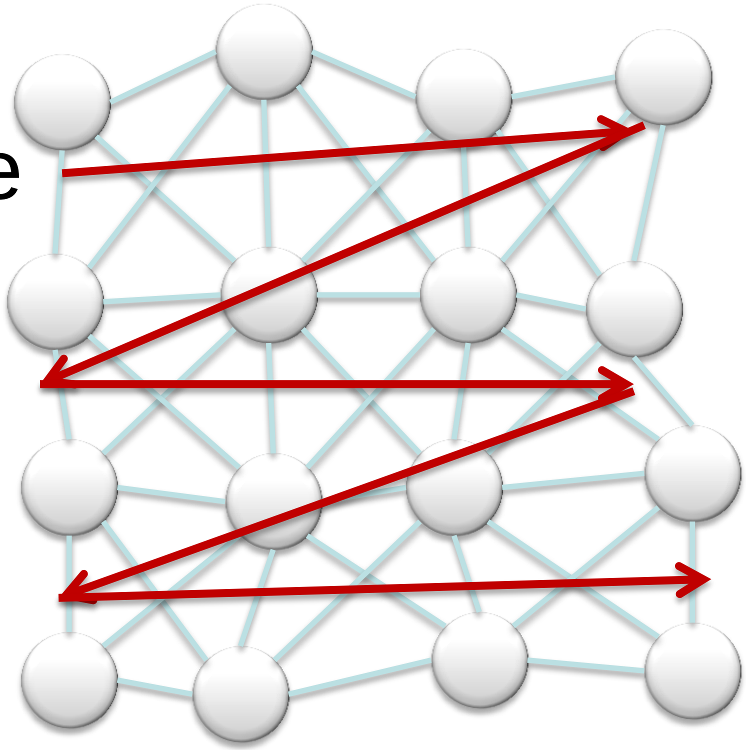
- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



AMD
The future is fusion

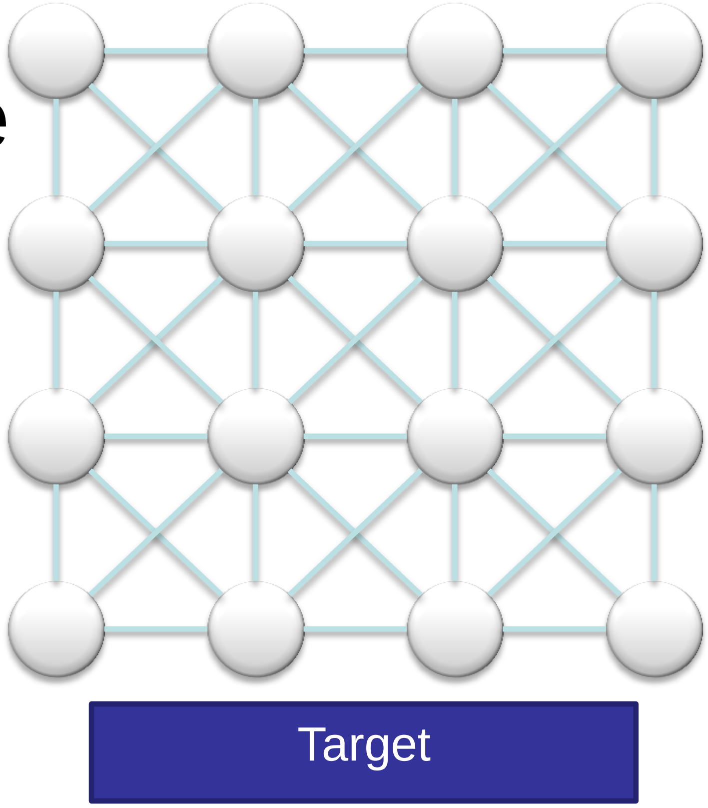
CPU approach to simulation

- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



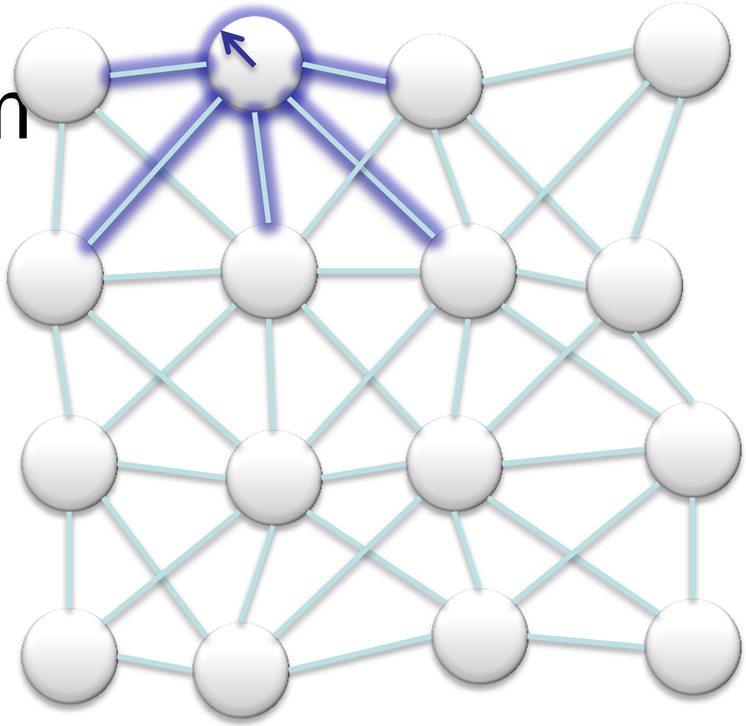
CPU approach to simulation

- One link at a time
- リンク一つずつ解く
- Perform updates in place
- そして座標を更新
- “Gauss-Seidel” style
- ガウスザイデル法
- Conserves momentum
- Iterate n times



Moving to the GPU: The pixel shader approach

- Offers full parallelism
- 完全な並列性
- One vertex per work-item
- All vertices together
- 全ての頂点を一度に解く
- No scattered writes

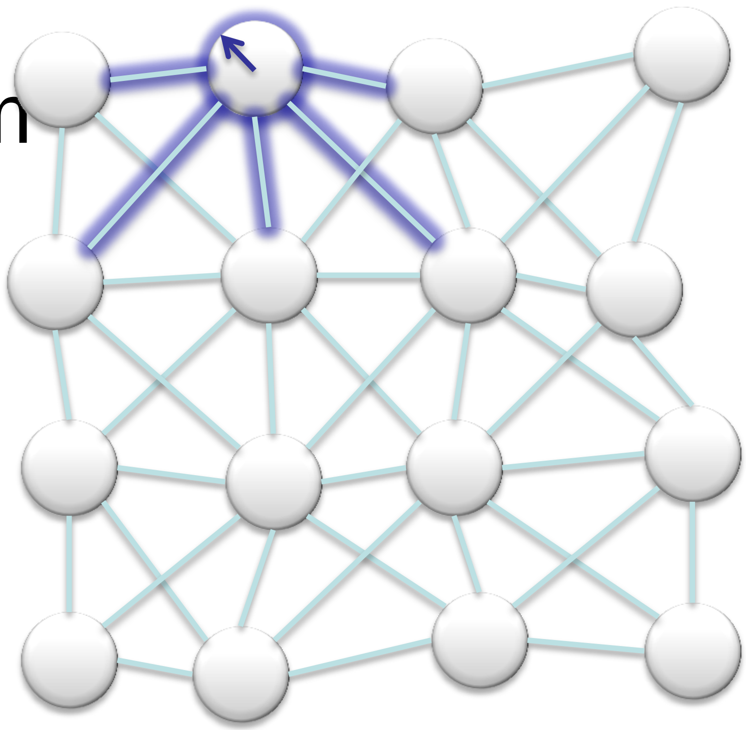


- Poor convergence
- 収束が悪いという問題がある

Moving to the GPU: The pixel shader approach

- Offers full parallelism
- 完全な並列性
- One vertex per work-item
- All vertices together
- 全ての頂点を一度に解く
- No scattered writes

- Poor convergence
- 収束が悪いという問題がある



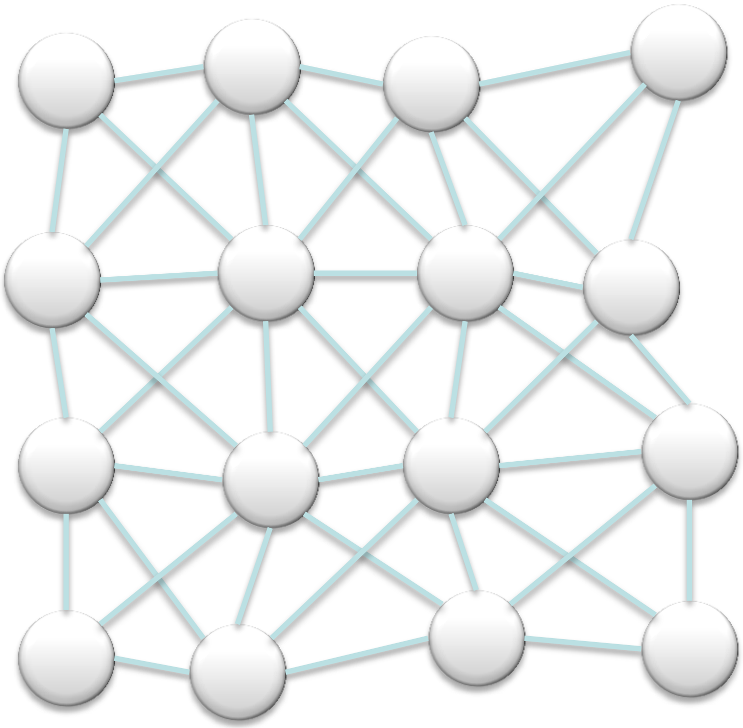
AMD
The future is fusion

Can DirectCompute 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

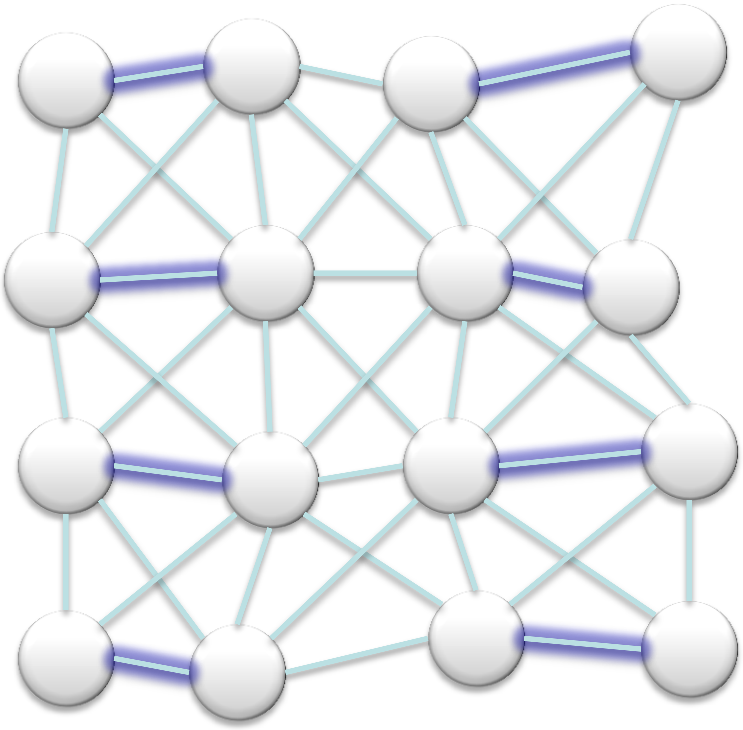
Execute independent subsets in parallel

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



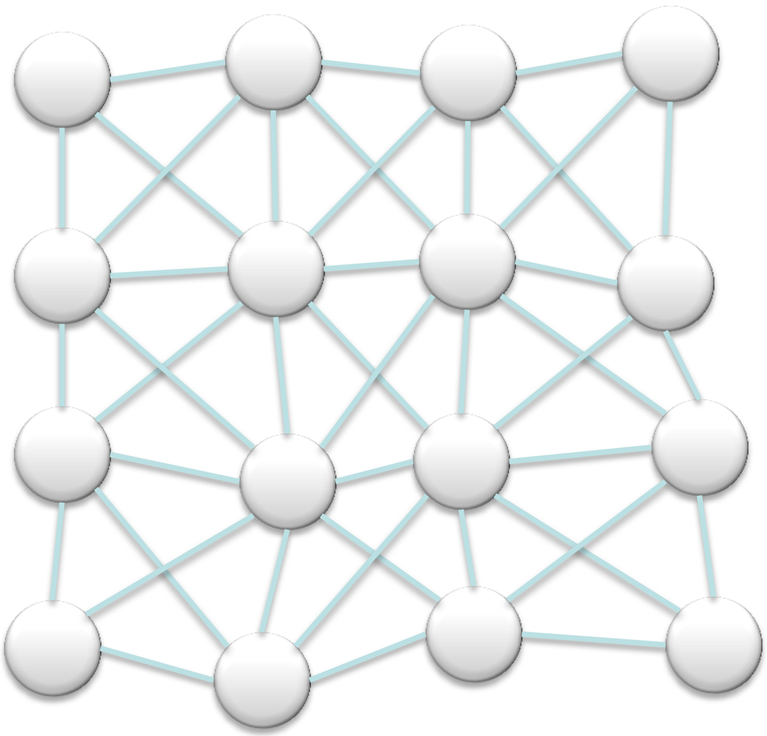
Execute independent subsets in parallel

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



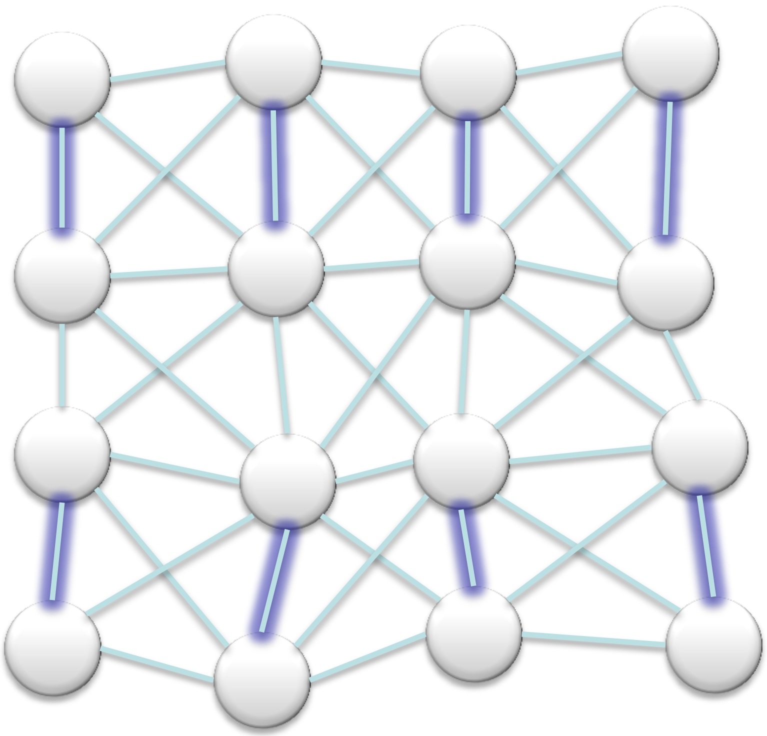
Execute independent subsets in parallel

- All links act at both ends
- Batch links
  - – No two links in a given batch share a vertex
  - – No data races



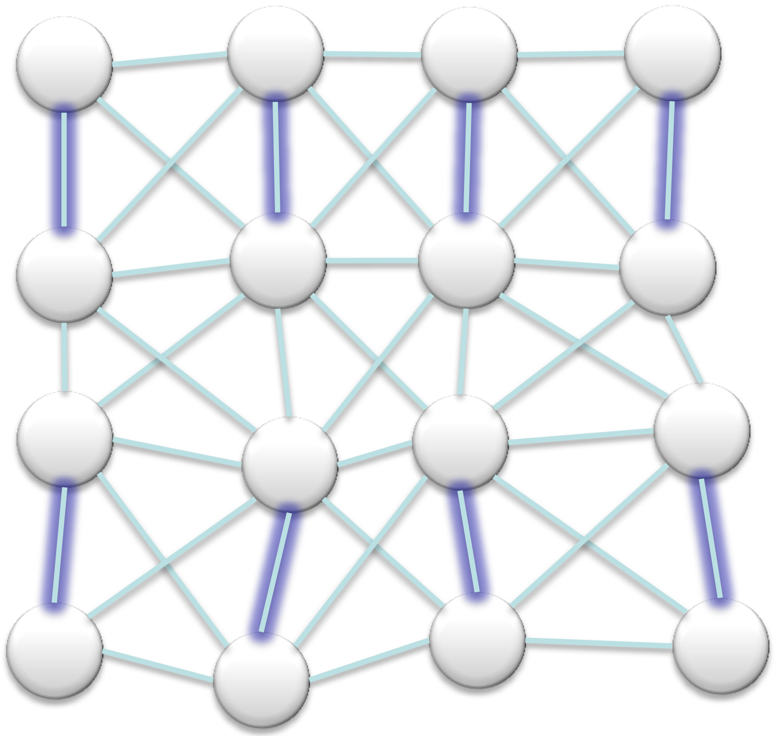
Execute independent subsets in parallel

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



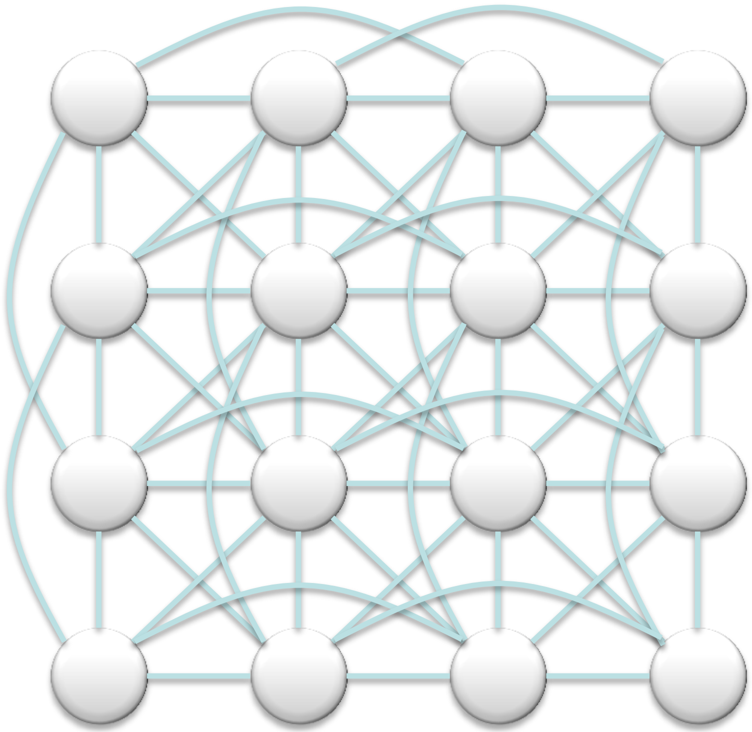
Execute independent subsets in parallel

- 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

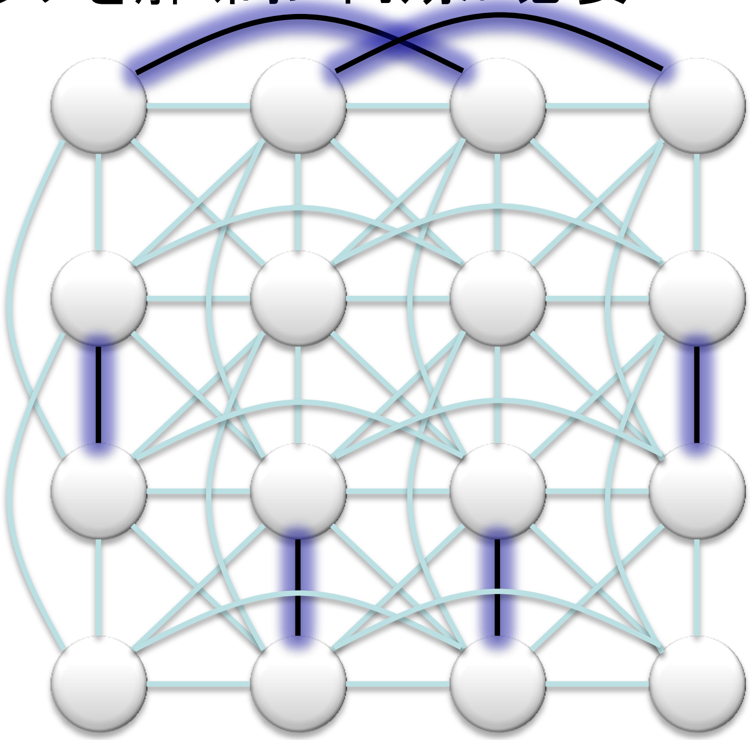
- Create independent subsets of links through graph coloring.
- Synchronize between batches
- 実際の布のメッシュでは多くのバッチが必要
  - グラフカラーリングで独立なサブセットを作る必要がある
  - それぞれのバッチを解く間に同期が必要



AMD
The future is fusion

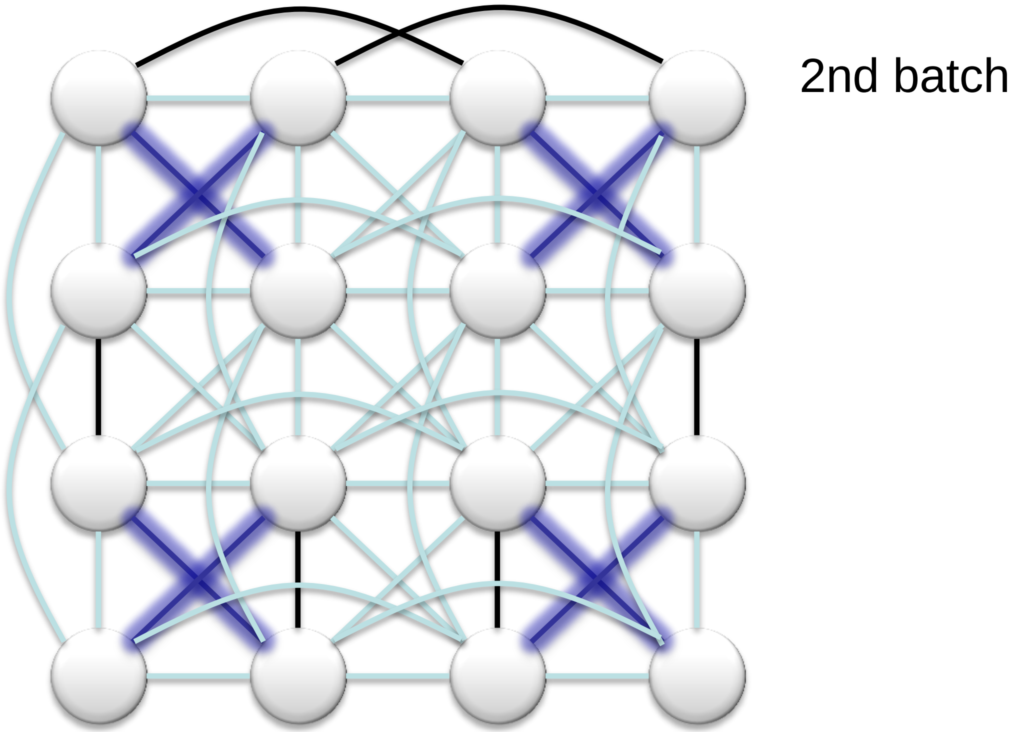
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
- 実際の布のメッシュでは多くのバッチが必要
  - グラフカラーリングで独立なサブセットを作る必要がある
  - それぞれのバッチを解く間に同期が必要



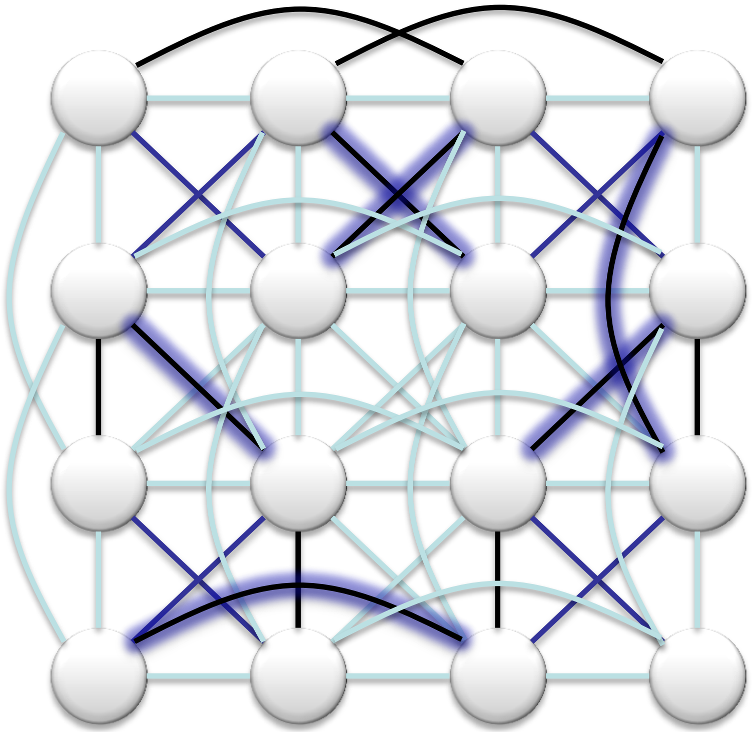
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
- 実際の布のメッシュでは多くのバッチが必要
  - グラフカラーリングで独立なサブセットを作る必要がある
  - それぞれのバッチを解く間に同期が必要



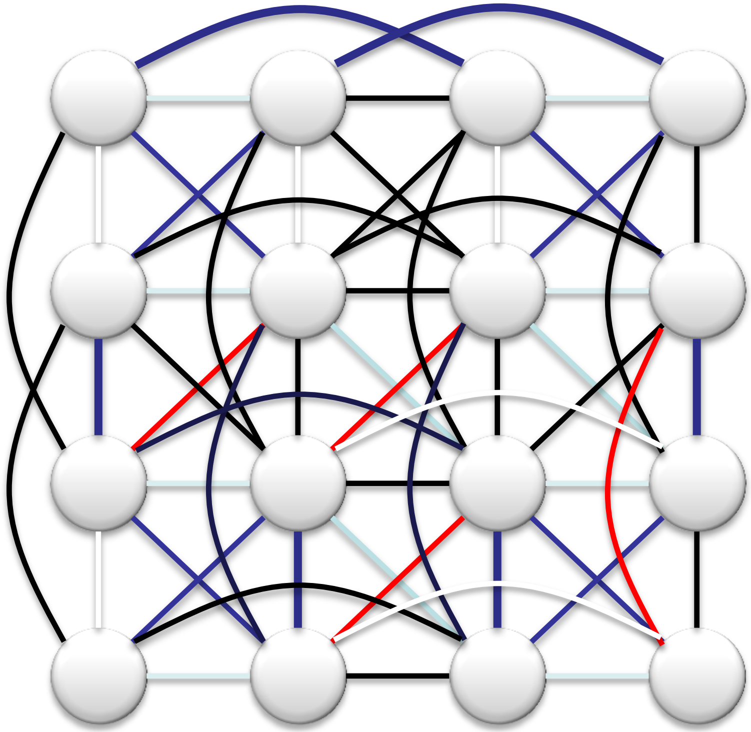
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
- 実際の布のメッシュでは多くのバッチが必要
  - グラフカラーリングで独立なサブセットを作る必要がある
  - それぞれのバッチを解く間に同期が必要



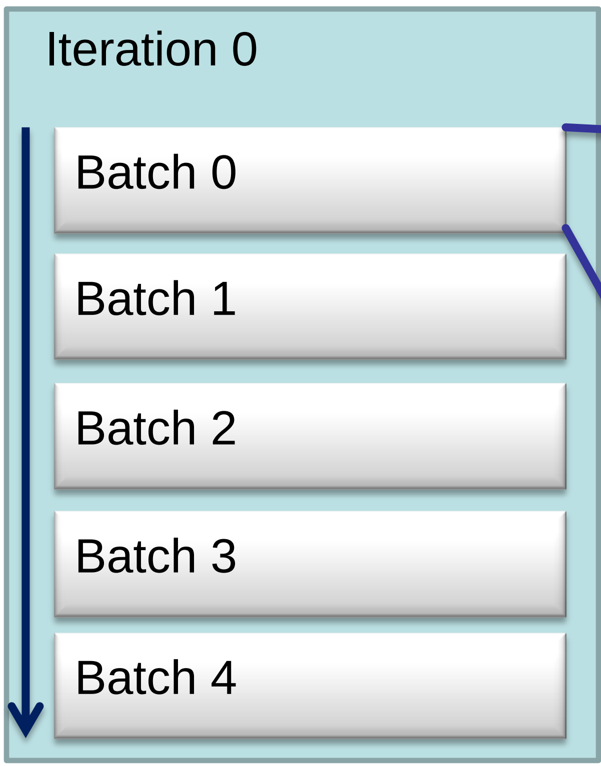
On a real cloth mesh we need many batches

- Create independent subsets of links through graph coloring.
- Synchronize between batches
- 実際の布のメッシュでは多くのバッチが必要
  - グラフカラーリングで独立なサブセットを作る必要がある
  - それぞれのバッチを解く間に同期が必要



10 batches

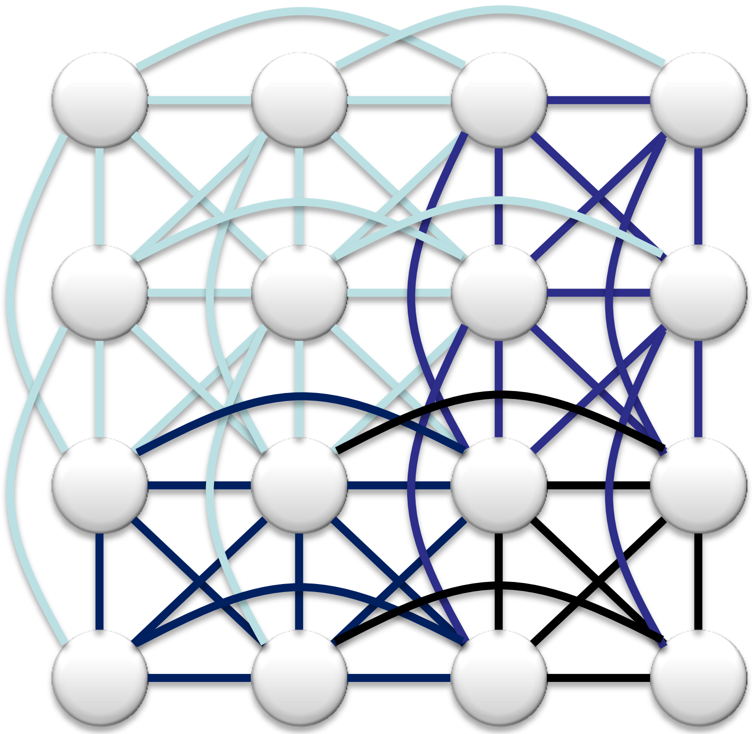
Driving batches and synchronizing




```
// Execute the kernel
context->CSSetShader(
  solvePositionsFromLinksKernel.kernel, NULL, 0 );

int numBlocks =
  (constBuffer.numLinks + (blockSize-1)) / blockSize;

context->Dispatch( numBlocks , 1, 1 );
```

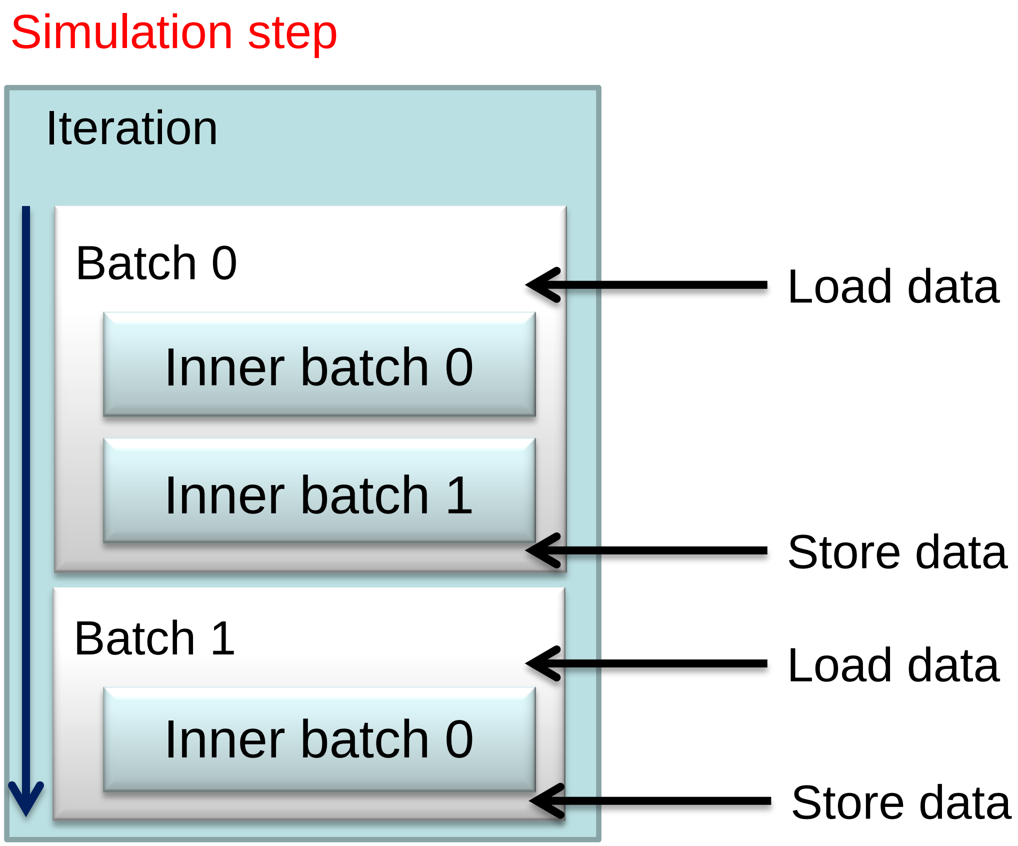
Packing for higher efficiency

- Can create clusters of links
  – The cloth is fixed-structure
  – Can be preprocessed
- Apply a group per DirectCompute “thread” group



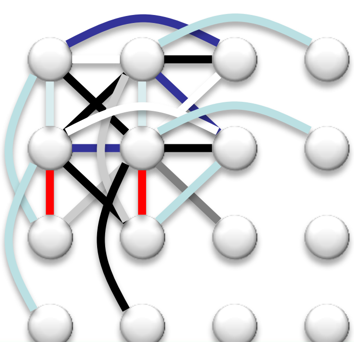
Driving batches and synchronizing

- The next feature of DirectCompute: **shared memory**



Solving in shared memory

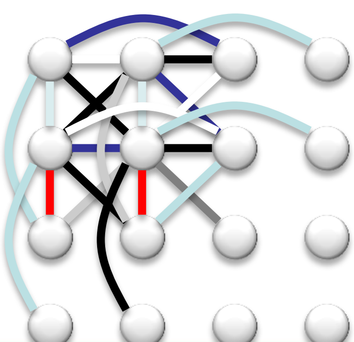
```
groupshared float4 positionSharedData[VERTS_PER_GROUP];

[numthreads(GROUP_SIZE, 1, 1)]
void
SolvePositionsFromLinksKernel( … uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID … )
{
  for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    positionSharedData[vertex] = g_vertexPositions[vertexAddress];
  }

  ... // Perform computation in shared buffer

  for( int vertex = GTid.x; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    g_vertexPositions[vertexAddress] = positionSharedData[vertex];
  }
}
```



Solving in shared memory

```
groupshared float4 positionSharedData[VERTS_PER_GROUP];

[numthreads(GROUP_SIZE, 1, 1)]
void
SolvePositionsFromLinksKernel( … uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID … )
{
  for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[

    positionSharedData[vertex] = g_vertexPositions[ver
  }

  ... // Perform computation in shared buffer

  for( int vertex = GTid.x; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    g_vertexPositions[vertexAddress] = positionSharedData[vertex];
  }
}
```

Define a “groupshared” buffer for shared data storage

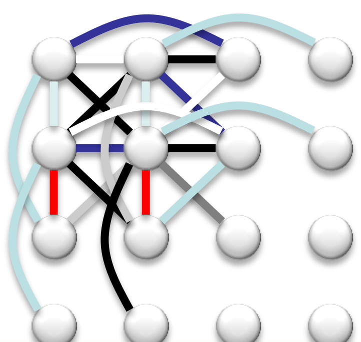


AMD The future is fusion

Solving in shared memory

```
groupshared float4 positionSharedData[VERTS_PER_GROUP];

[numthreads(GROUP_SIZE, 1, 1)]
void
SolvePositionsFromLinksKernel( … uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID … )
{
  for( int vertex = laneInWavefront; vertex < vertices
  {
    int vertexAddress = g_vertexAddressesPerWavefront[

    positionSharedData[vertex] = g_vertexPositions[ver
  }

  ... // Perform computation in shared buffer

  for( int vertex = GTid.x; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    g_vertexPositions[vertexAddress] = positionSharedData[vertex];
  }
}
```

Data will be shared across a group of threads with these dimensions



AMD
The future is fusion

Solving in shared memory

```
groupshared float4 positionSharedData[VERTS_PER_GROUP];

[numthreads(GROUP_SIZE, 1, 1)]
void
SolvePositionsFromLinksKernel( … uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID … )
{
  for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];
    positionSharedData[vertex] = g_vertexPositions[vertexAddress];
  }

  ... // Perform computation in shared buffer

  for( int vertex = GTid.x; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    g_vertexPositions[vertexAddress] = positio
  }
}
```

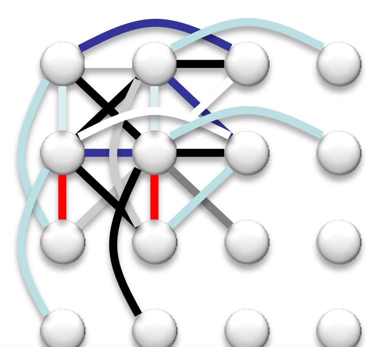
Load data from global buffers into the shared region

AMD The future is fusion

Solving in shared memory

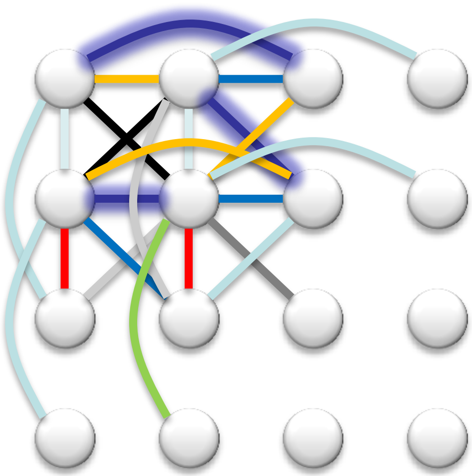
```
groupshared float4 positionSharedData[VERTS_PER_GROUP];

[numthreads(GROUP_SIZE, 1, 1)]
void
SolvePositionsFromLinksKernel( … uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID … )
{
  for( int vertex = laneInWavefront; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    positionSharedData[vertex] = g_vertexPositio
  }

  ... // Perform computation in shared buffer

  for( int vertex = GTid.x; vertex < verticesUsedByWave; vertex+=GROUP_SIZE )
  {
    int vertexAddress = g_vertexAddressesPerWavefront[groupID*VERTS_PER_GROUP + vertex];

    g_vertexPositions[vertexAddress] = positionSharedData[vertex];
  }
}
```

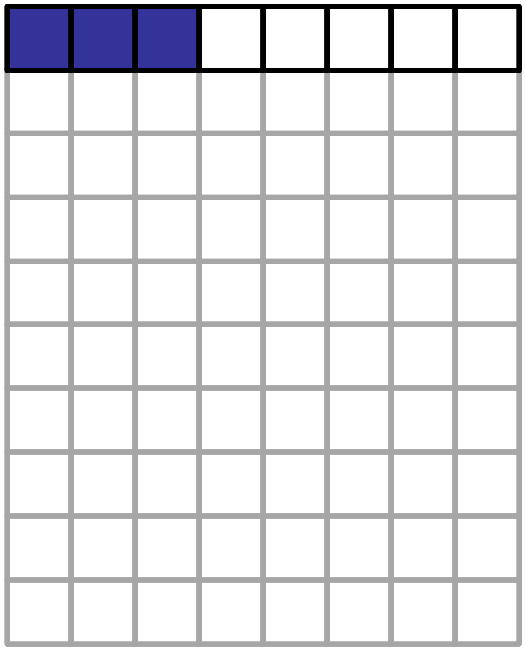
Write back to the global buffer after computation



Group execution

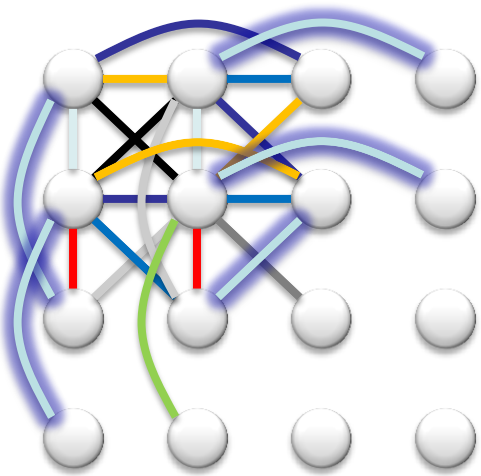
- The sequence of inner batch operations for the first cluster is:
- それぞれのクラスタ内で独立なバッチセットを作成
  - – バッチセットは右図のようになる

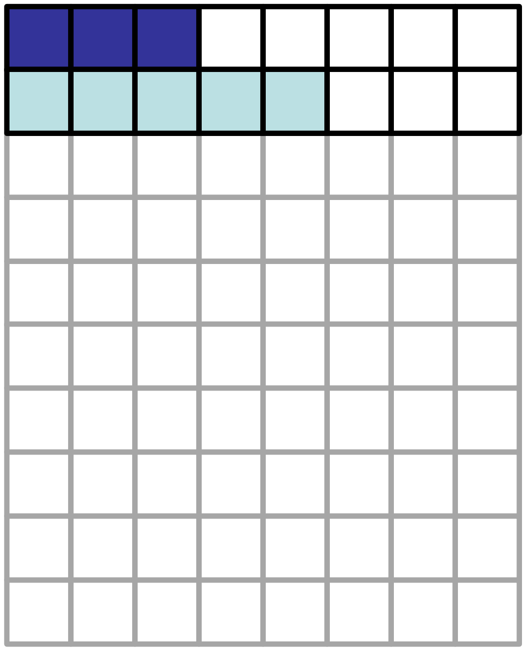




Group execution

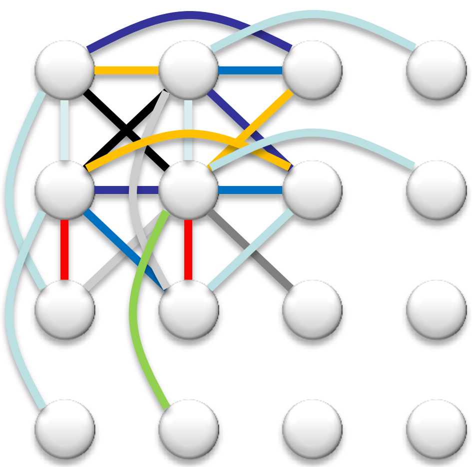
- The sequence of inner batch operations for the first cluster is:
- それぞれのクラスタ内で独立なバッチセットを作成
  - バッチセットは右図のようになる

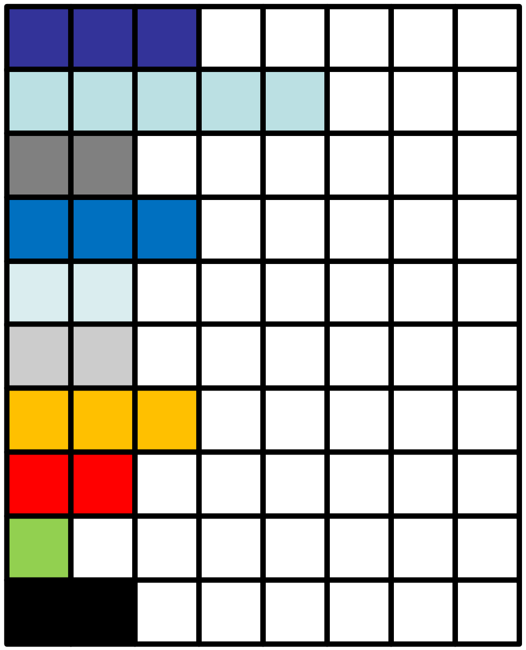




Group execution

- The sequence of inner batch operations for the first cluster is:
- それぞれのクラスタ内で独立なバッチセットを作成
  - バッチセットは右図のようになる



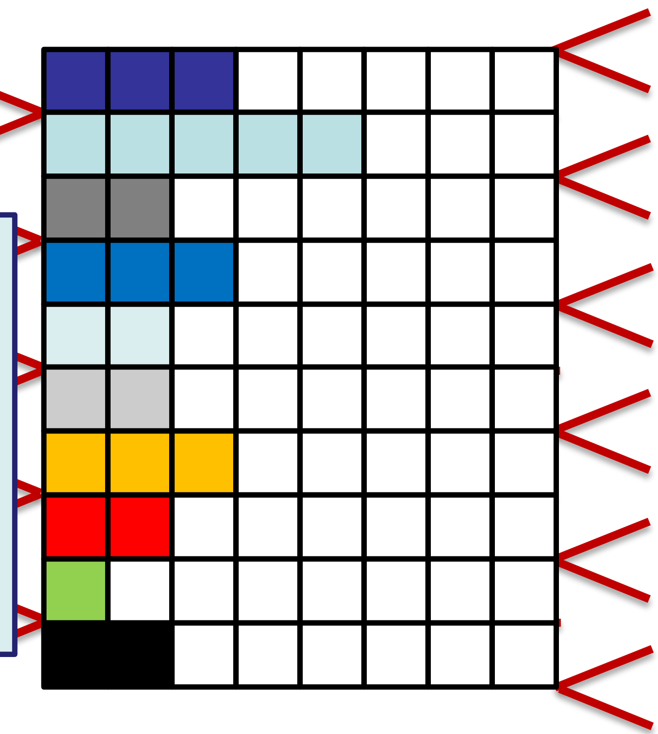


Group execution

- The sequence of inner batch operations for the first cluster is:
- それぞれのクラスタ内で独立なバッチセットを作成
  - バッチセットは右図のようになる

Synchronize…

```
// load
AllMemoryBarrierWithGroupSync();

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

// Store
```



Why is this an improvement?

- In-cluster batches now in-shader loop
- グローバルな同期を取らなければならなかった計算がクラスタ内の同期だけで済む

- Only 4 shader dispatches: less overhead
- 4回の同期、シェーダ実行で済む為、少ないオーバーヘッド

- Barrier synchronization is still slow
- クラスタ内のバリアを使った同期はまだ遅い
- しかし

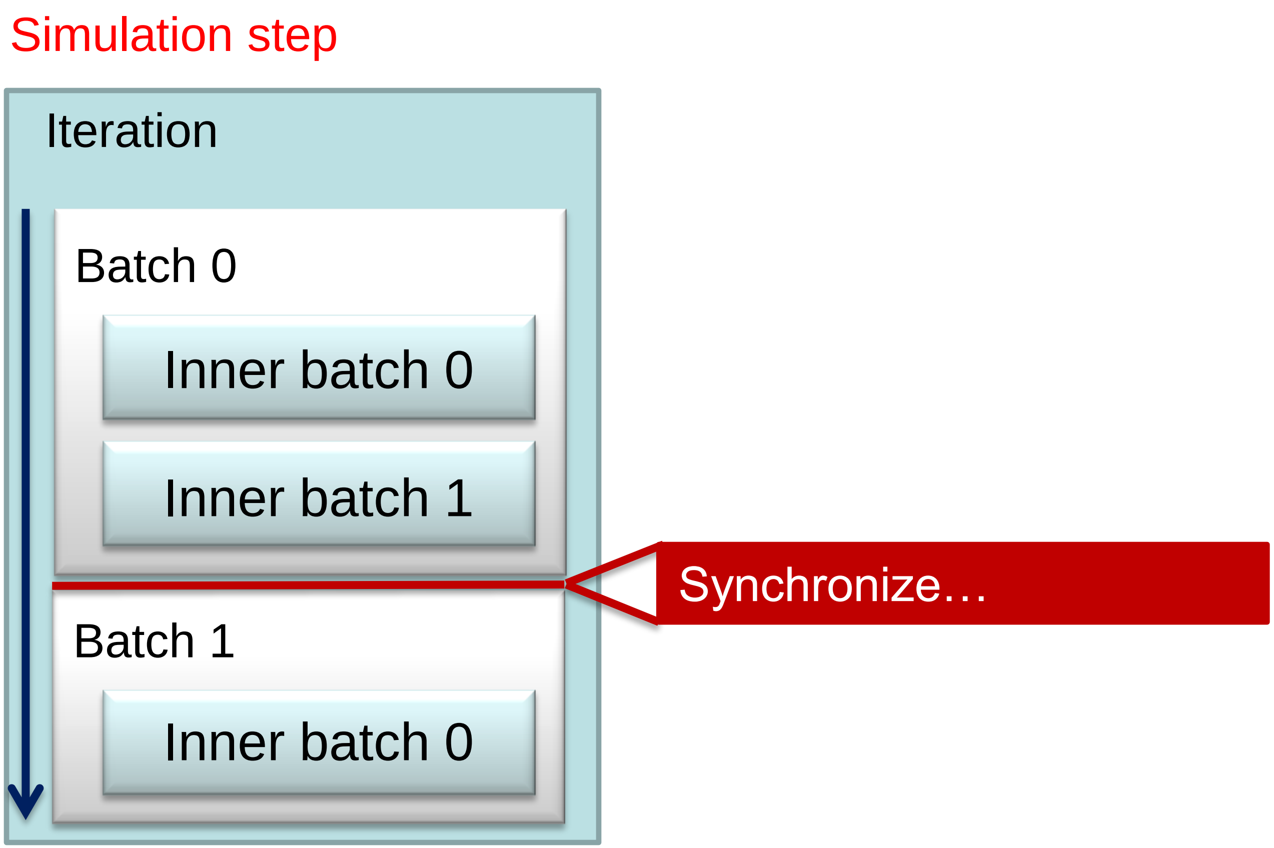
Exploiting the SIMD architecture

- Hardware executes 64- or 32-wide SIMD
- AMDのハードウェアは64SIMDで実行

- Sequentially consistent at the SIMD level
  – So clusters can run on SIMDs, not groups

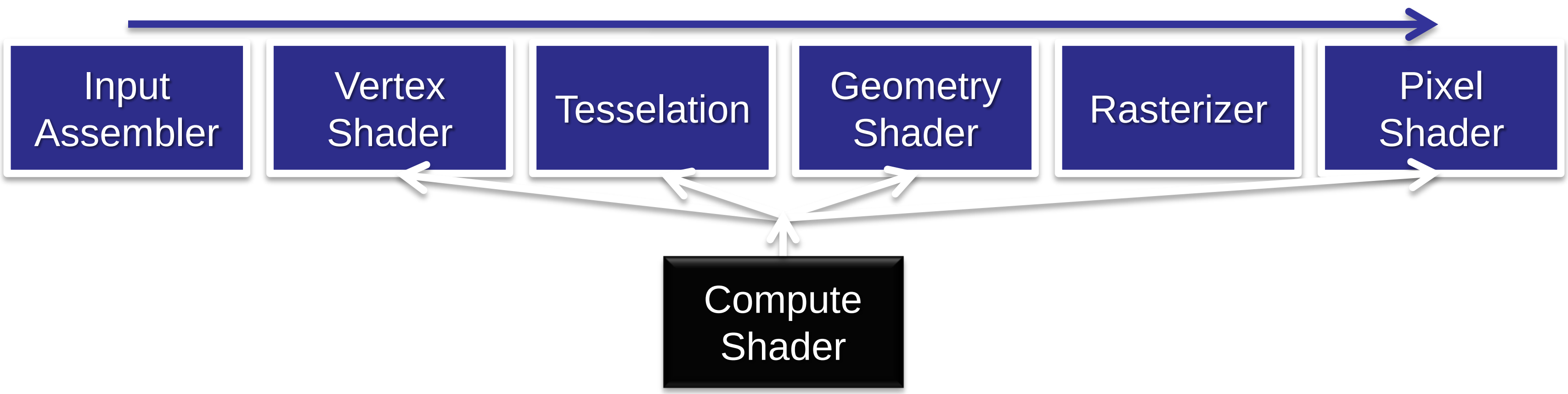
- Synchronization is now implicit
- そのため各クラスタで64スレッドごと立ち上げれば同期を明示的に導入しなくても良い

Driving batches and synchronizing



One more thing…

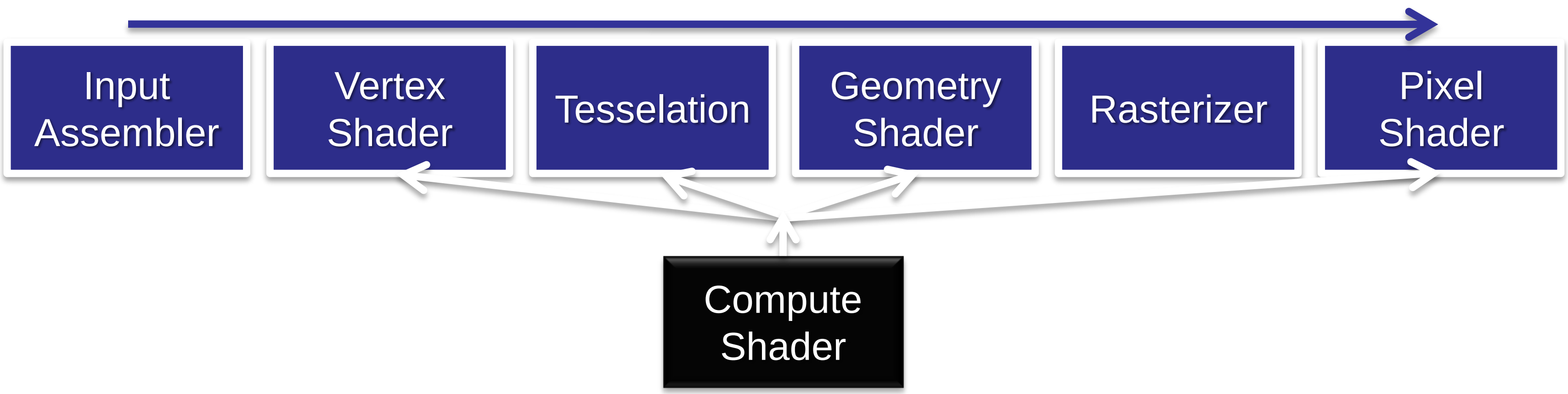
- Remember the tight pipeline integration?



- How can we use this to our advantage?
- Write directly to vertex buffer!
- コンピュートシェーダを使ってGPU上の布の頂点データを計算しているので直接バーテックスバッファに書き出すことが可能

One more thing…

- Remember the tight pipeline integration?



- How can we use this to our advantage?
- Write directly to vertex buffer!
- コンピュートシェーダを使ってGPU上の布の頂点データを計算しているので直接バーテックスバッファに書き出すことが可能

Create a vertex buffer

```
// Create a vertex buffer with unordered access support
D3D11_BUFFER_DESC bd;
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = vertexBufferSize * 32;
bd.BindFlags =
  D3D11_BIND_VERTEX_BUFFER |
  D3D11_BIND_UNORDERED_ACCESS
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
bd.StructureByteStride = 32;
hr = m_d3dDevice->CreateBuffer(&bd, NULL, &m_Buffer);

// Create an unordered access view of the buffer to allow writing
D3D11_UNORDERED_ACCESS_VIEW_DESC uavbuffer_desc;
ud.Format = DirectXGI_FORMAT_UNKNOWN;
ud.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
ud.Buffer.NumElements = vertexBufferSize;
hr = m_d3dDevice->CreateUnorderedAccessView(m_Buffer, &ud, &m_UAV);
```

Vertex buffer also bound for unordered access.

Scattered writes!

Performance gains

- For 90,000 links:

– Rendering and copy only	2.98	ms/frame
– バッチソルバ (Batched solver)	3.84	ms/frame
– SIMDバッチソルバ (SIMD batched solver)	3.22	ms/frame
– SIMDバッチソルバとGPU上でのコピー (SIMD with GPU copy)	0.617	ms/frame

- 3.5x improvement in solver alone

Based on internal numbers

Thanks

- Justin Hensley
- Holger Grün
- Nicholas Thibieroz
- Erwin Coumans

References

- Yang J., Hensley J., Grün H., Thibieroz N.: Real-Time Concurrent Linked List Construction on the GPU. In Rendering Techniques 2010: Eurographics Symposium on Rendering (2010), vol. 29, Eurographics.
- Grün H., Thibieroz N.: OIT and Indirect Illumination using DirectX11 Linked Lists. In Proceedings of Game Developers Conference 2010 (Mar. 2010). http://developer.amd.com/gpu_assets/OIT%20and%20Indirect%20Illumination%20using%20DirectX11%20Linked%20Lists_forweb.ppsx
- http://developer.amd.com/samples/demos/pages/ATIRadeonHD5800SeriesRealTimeDemos.aspx
- http://bulletphysics.org

AMD
The future is fusion

Trademark Attribution

AMD
The future is fusion