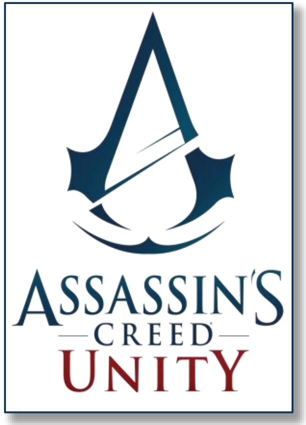
Ubisoft Cloth Simulation: Performance Post-mortem & Journey from C++ to Compute Shaders

Alexis Vaisse
Lead Programmer – Ubisoft

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Motion Cloth

- Cloth simulation developed by Ubisoft
- Used in:



UNANNOUNCED

UNANNOUNCED

UNANNOUNCED

UNANNOUNCED

Agenda

- Cloth Simulation Performance Post-mortem

What is the solution?

- Journey from C++ to Compute Shaders

Cloth simulation performance post-mortem

- The cloth simulation itself is quite fast
- But it requires a lot of processing before and after

Simulation	~ 40%
Pre- & Post-simulation	~ 60%

Cloth simulation performance post-mortem

- Skinning
- Interpolation system
- Mapping
- Tangent space
- Critical path

Skinning

In an ideal world:

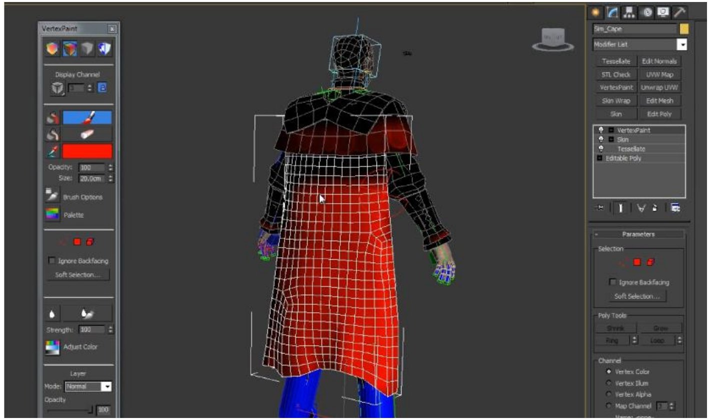
- Set a material on the cloth
- Let the simulation do the job

Skinning

In practice:

- We need to control the cloth
- The cloth must look impressive even when the character's movement is not physically realistic
- The skinned vertices are heavily used to control the cloth

Skinning

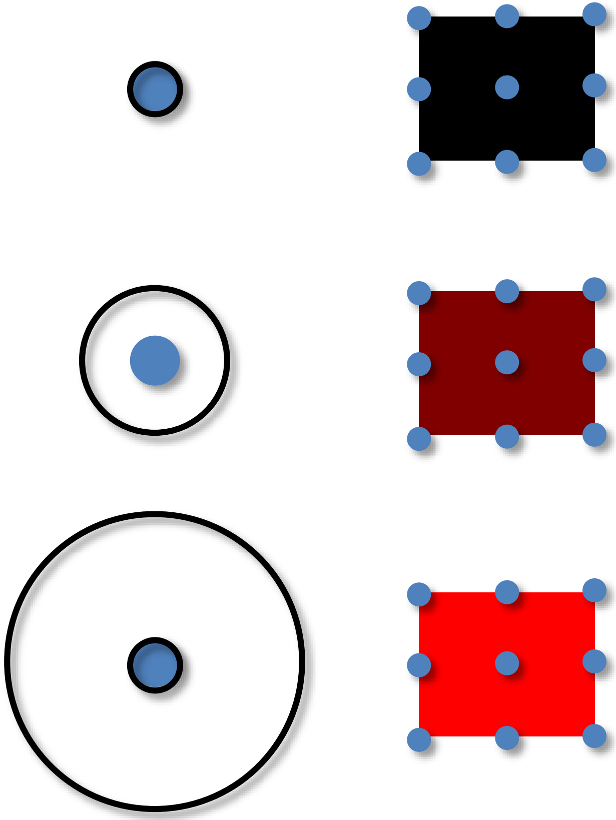


Maximum distance constraints:

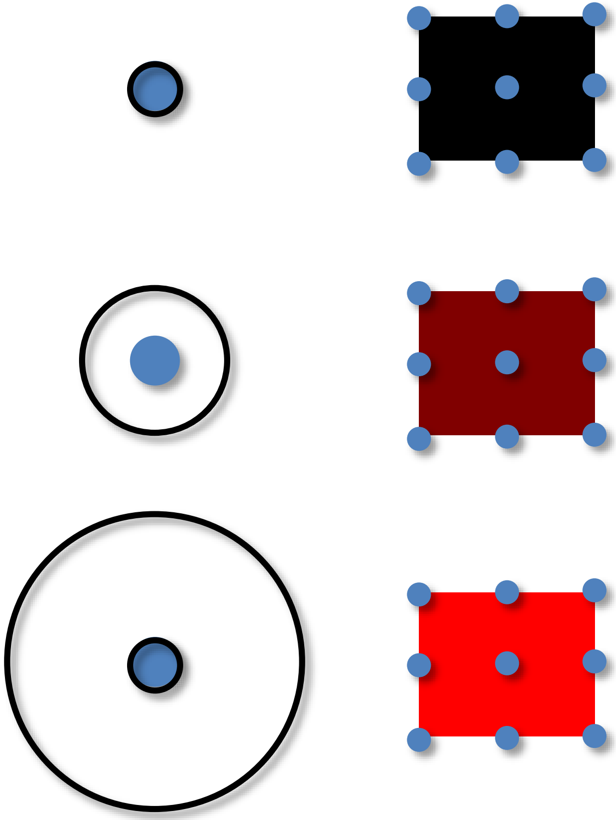
- Maximum displacement of each vertex
- Relatively to its skinned position
- Controlled by a vertex paint layer

Skinning

- The simulated vertex can move inside a sphere centered around the skinned vertex
- The radius of the sphere depends on the color at the vertex in the vertex paint layer



Skinning



Skinning

Skinning is also used by:

- Blend constraints
- Levels of detail

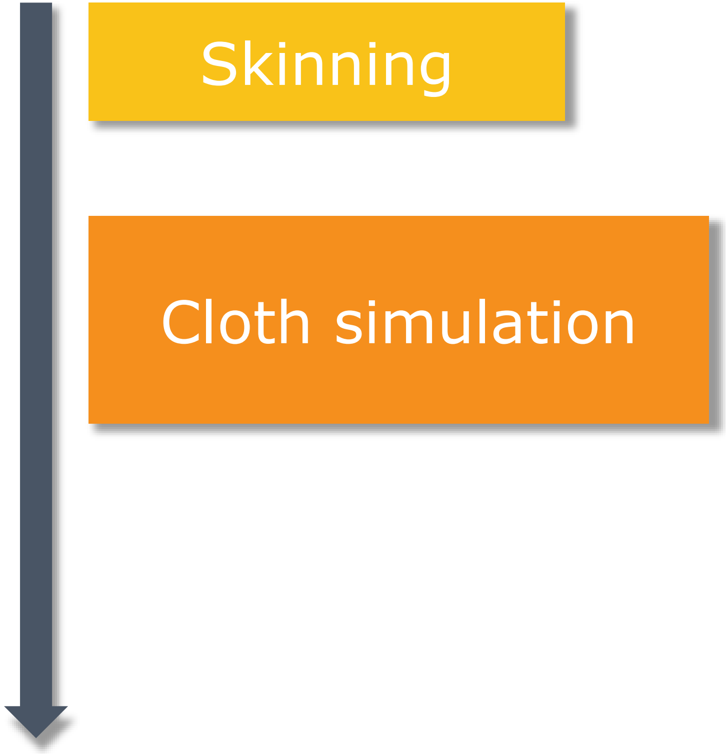
Skinning

➜ We definitely need to compute skinning

- ~~Compute on the GPU then transfer~~
  - ➜ Serious synchronization issues
- Compute on the CPU
  - ➜ Most of the time before the simulation

Cloth simulation performance post-mortem

- Skinning
- Interpolation system
- Mapping
- Tangent space
- Critical path



Interpolation system

Game frame rate ≠ simulation frame rate

Game frame rate:

- Usually locked to 30 fps
- But can be lower in a few specific places on consoles
- Can be lower and fluctuate on PC
- Also fluctuates a lot during the production of the game

Interpolation system

Game frame rate ≠ simulation frame rate

Simulation frame rate:

- Must be fixed (limitation of the algorithm)
- 30 fps if no collision or slow pace
  - → Flags, walking characters
- 60 fps if fast moving collision objects
  - → Running or playable characters

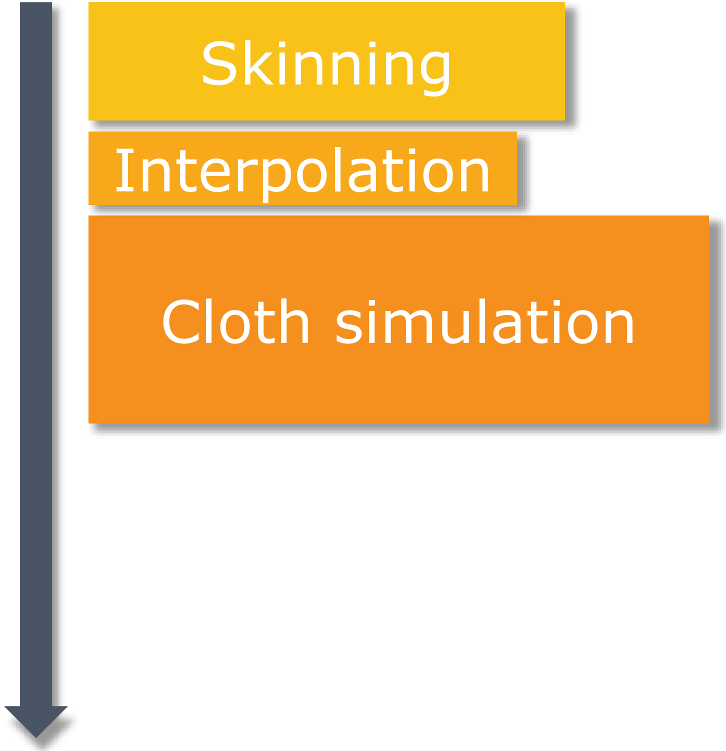
Interpolation system

- Cloth simulation called several times per frame
- Interpolate:
  - The skinned vertices (position and normal)
  - Collision objects (position and orientation)

➜ Still quite cheap compared to skinning

Cloth simulation performance post-mortem

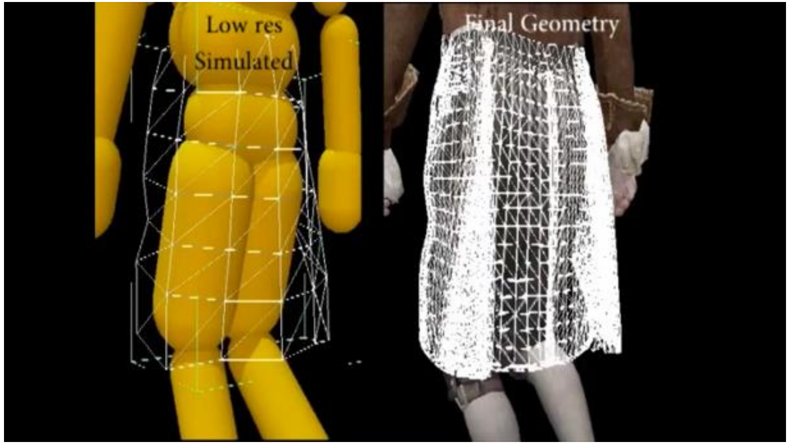
- Skinning
- Interpolation system
- Mapping
- Tangent space
- Critical path



Mapping

WHAT?

- Map a high-res visual mesh
- To a lower-res simulated mesh



Mapping

WHY?

- Simulating a high-res mesh is too costly
- It doesn't give good results
  - → Too silky, too light
- Ability to update the visual mesh without breaking the cloth setup

Mapping

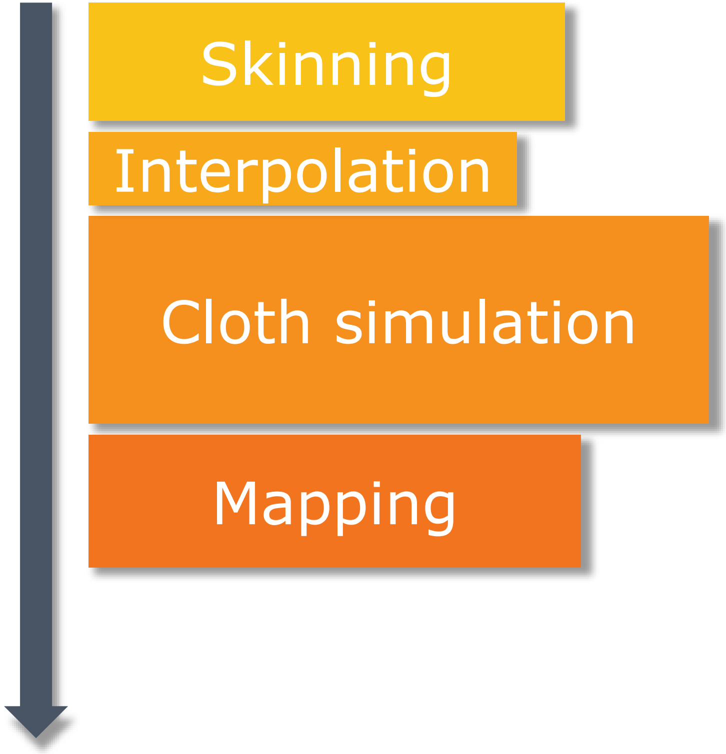
COST?

- Compute position and normal of each visual vertex
- Mapping ~ 10x faster than simulation
- But high-res mesh can have 10x more vertices!

➥ Up to same cost or even higher in worst cases

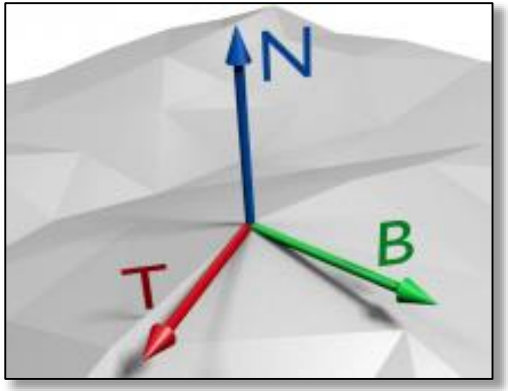
Cloth simulation performance post-mortem

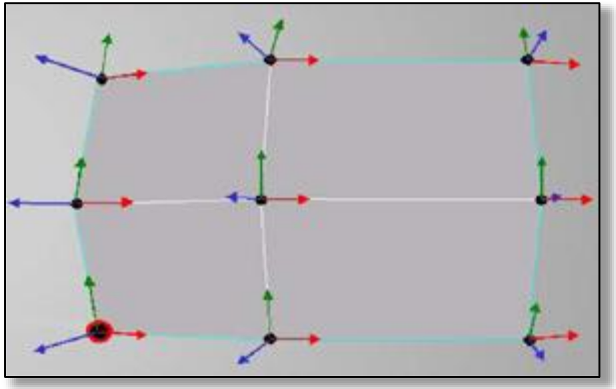
- Skinning
- Interpolation system
- Mapping
- Tangent space
- Critical path



Tangent space

- Tangent space is required for normal mapping



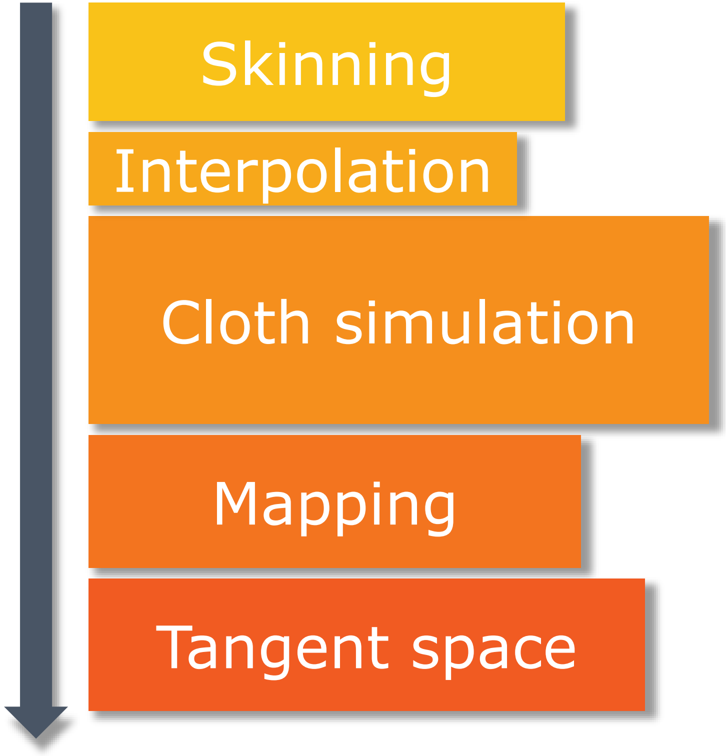


Tangent space

- Tangent space is required for normal mapping
- Compute it on CPU ← Most of the time taken after the simulation
  - Costly
- Compute it on the GPU
  - Requires specific shaders

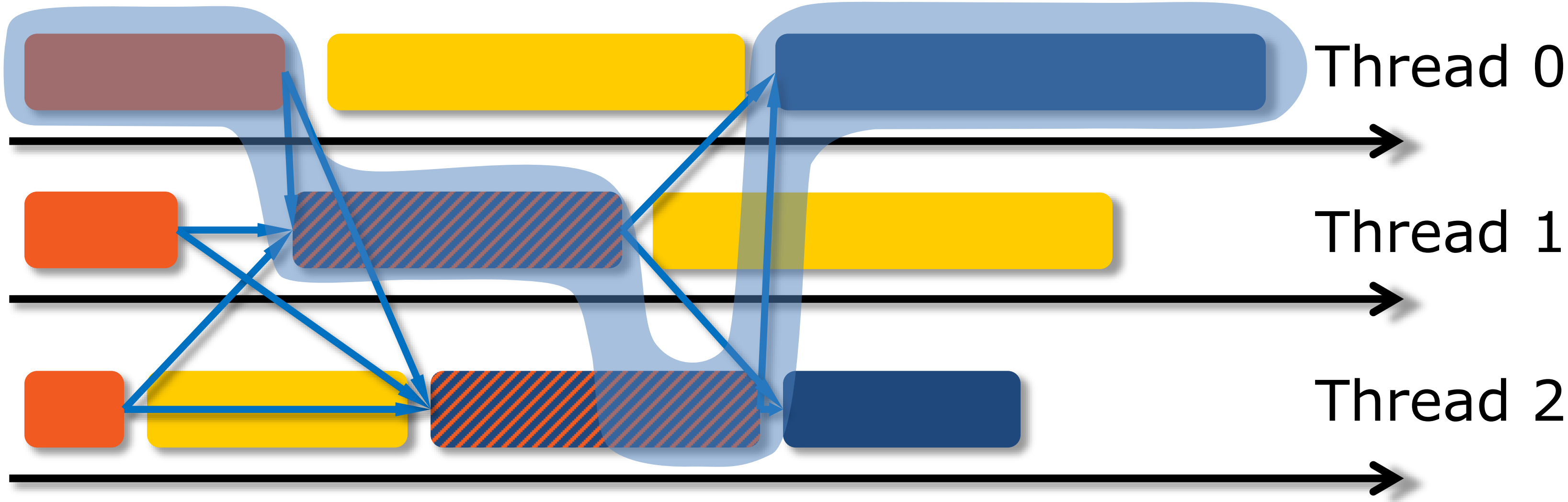
Cloth simulation performance post-mortem

- Skinning
- Interpolation system
- Mapping
- Tangent space
- Critical path



Critical path

WHAT IS CRITICAL PATH?



Critical path

- Adding a task on the critical path
  - ➜ Bigger duration for the game engine loop
- Adding a task outside the critical path
  - ➜ Doesn’t change the engine loop’s duration
  - ➜ It’s “free”
    - Unless task is too big
    - Unless perfect balancing

Critical path

Is cloth simulation on the critical path?

- Scenario 1: cloth doesn’t need skinning

Critical path

Is cloth simulation on the critical path?

- Scenario 1: cloth doesn’t need skinning
- Dependency:

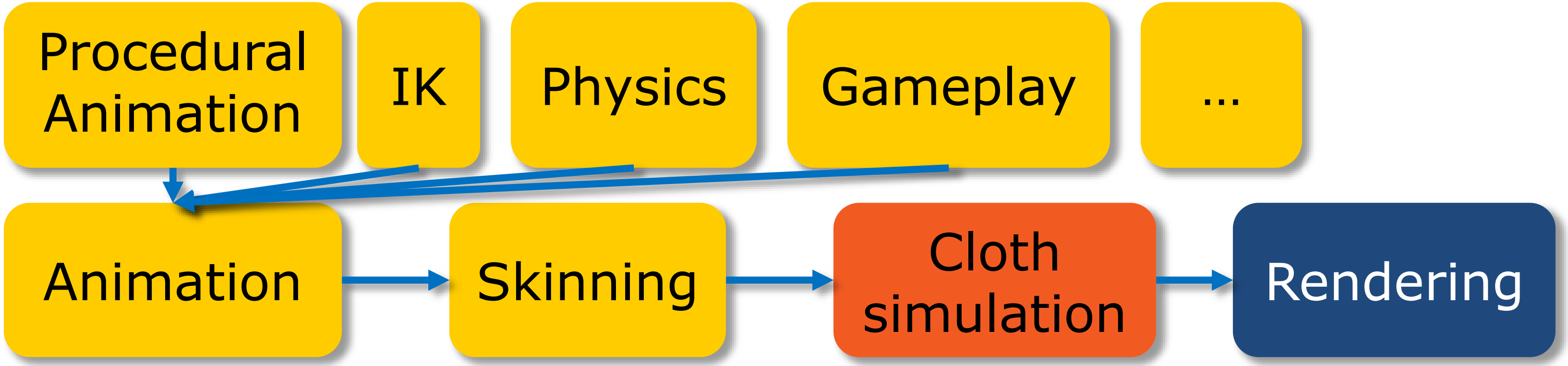
Cloth simulation → Rendering

Not on the critical path

Critical path

Is cloth simulation on the critical path?

- Scenario 2: cloth does need skinning



Critical path

Is cloth simulation on the critical path?

- Scenario 2: cloth does need skinning

  → Most of the time on the critical path

Consequence:

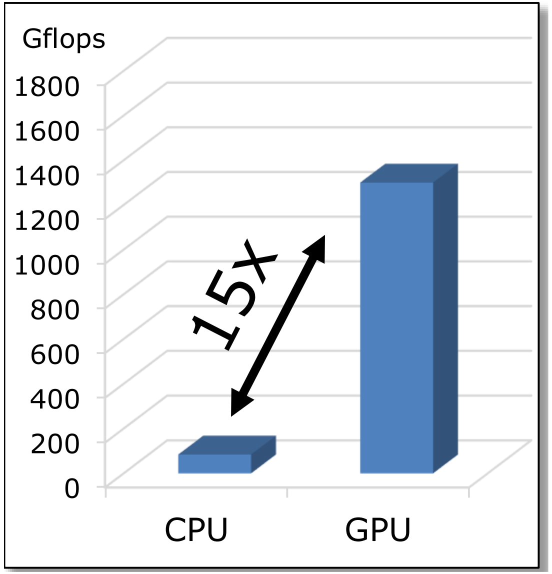
Hey! The game is too slow!

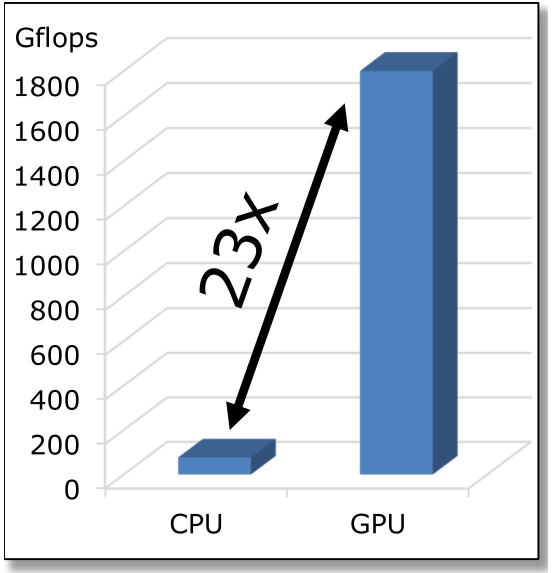
Use more aggressive cloth levels of detail, and it's fixed!

- Cloth Simulation Performance Post-mortem

What is the solution?

Peak power: Xbox One PS4





- Cloth Simulation
  Performance Post-mortem

What is the solution?

- Journey from C++ to Compute Shaders

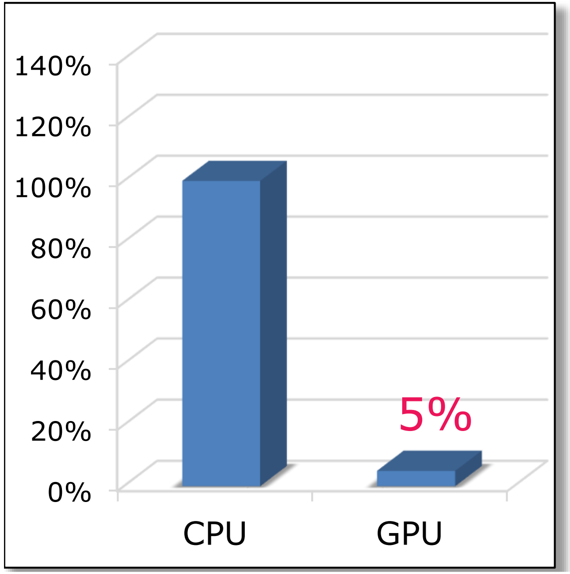
Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

The first attempts

Integrate velocity	Compute Shader
Resolve some constraints	Compute Shader
Resolve collisions	Compute Shader
Resolve some more constraints	Compute Shader
Do some other funny stuffs	Compute Shader
...	Compute Shader

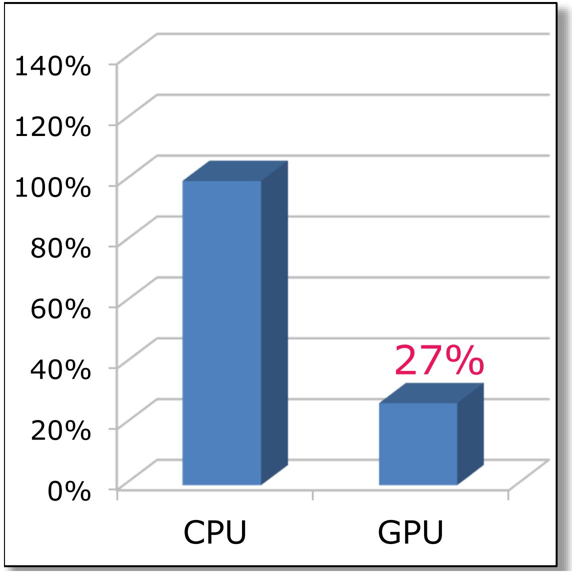
The first attempts



- The GPU version is 20x slower than the CPU version!!
- Too many “Dispatch” calls
- Bottleneck = CPU

The first attempts

- Merge several cloth items to get better performance
- It's better, but it's not enough
- Problem: all cloth items must have the same properties

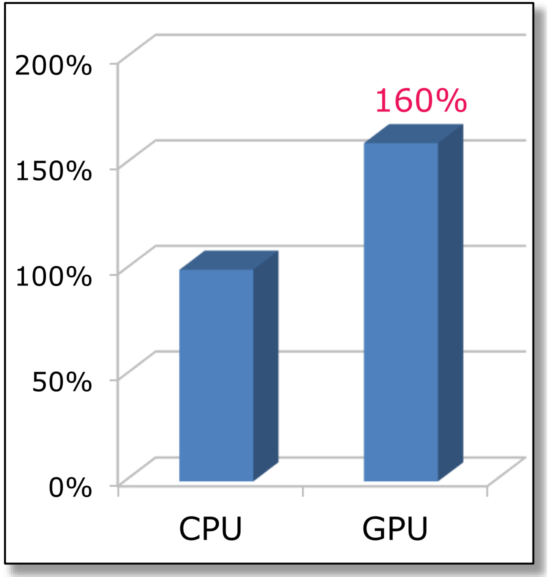


Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

A new approach

- A single huge compute shader to simulate the entire cloth
- Synchronization points inside the shader
- A single "Dispatch" call instead of 50+
- Simulate several cloth items (up to 32) using a single "Dispatch" call
- The GPU version is now faster than the CPU version



Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

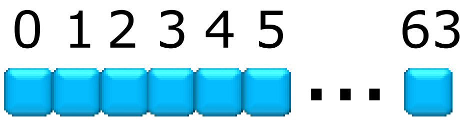
The shader

- 43 .hlsl files
- 3,400 lines of code

  (+ 800 lines for unit tests & benchmarks)
- Compiled shader code size = 75 KB

The shader – Easy parts

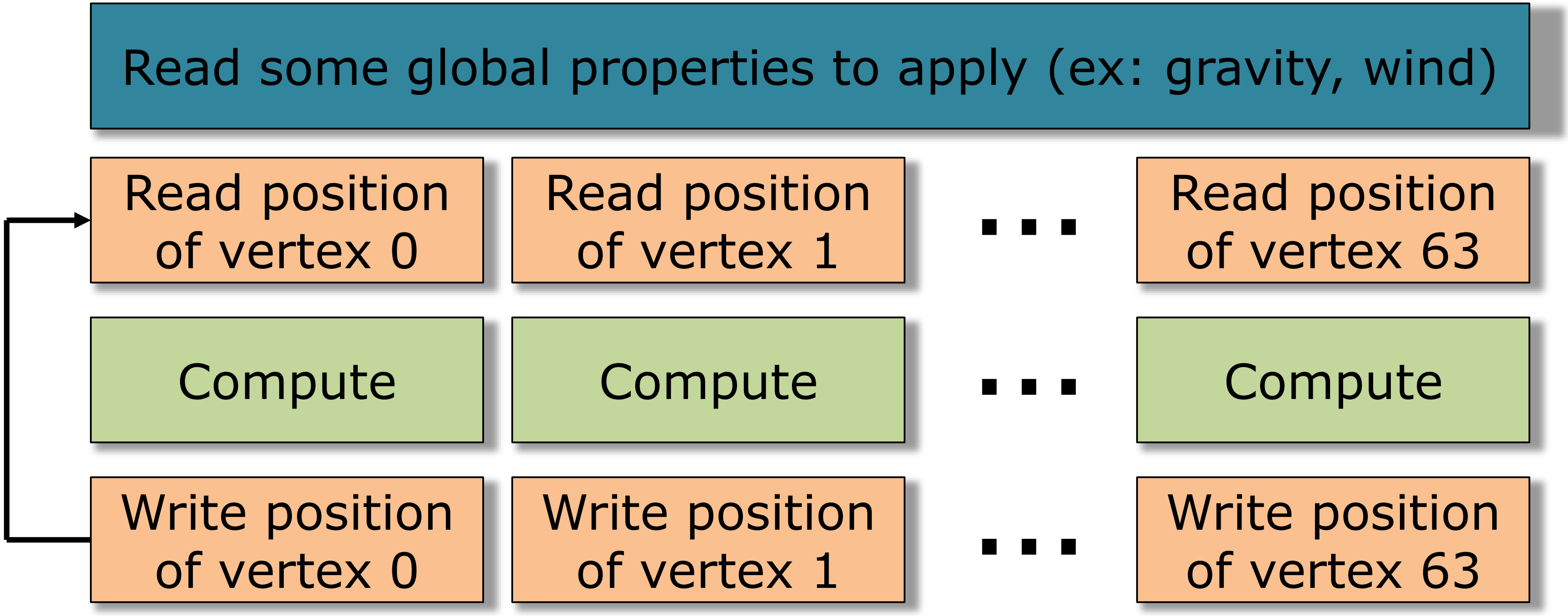
- Thread group:



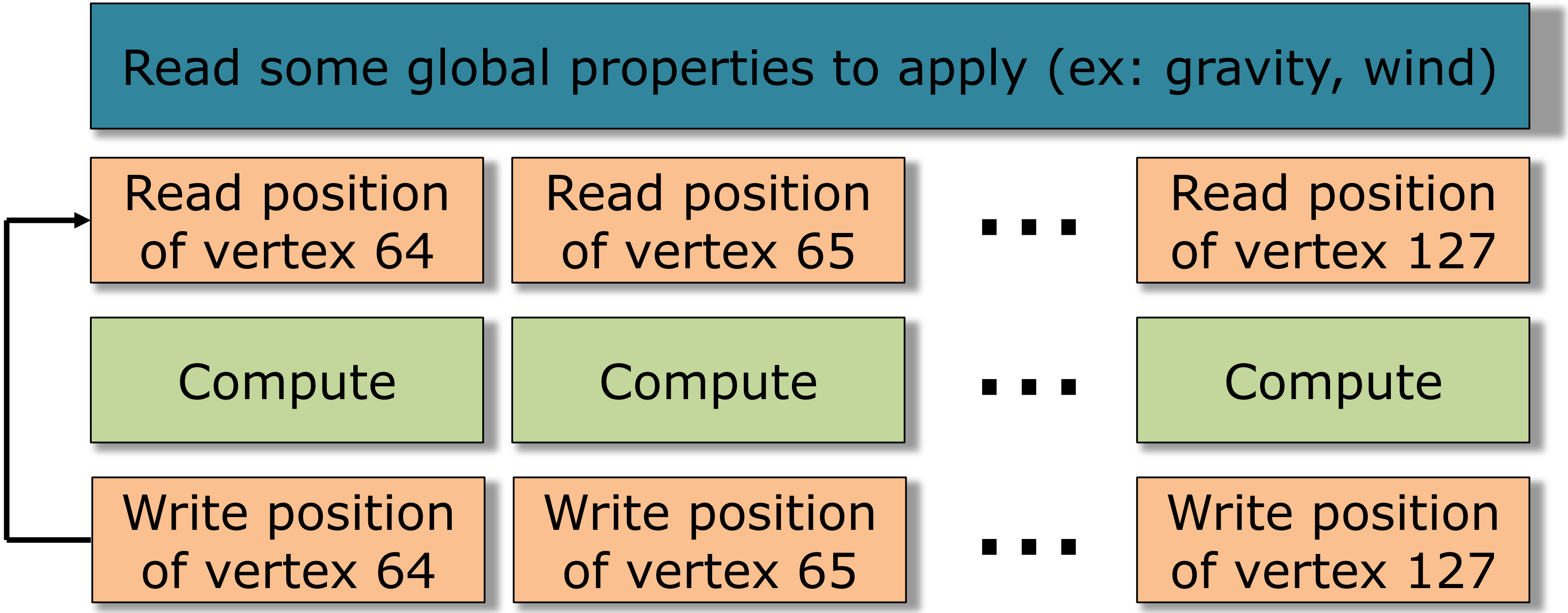
- We do the same operation on 64 vertices at a time

There must be no dependency between the threads

The shader – Easy parts



The shader – Easy parts



The shader – Easy parts

Read position of vertex 0	Read position of vertex 1	...	Read position of vertex 63
Read property for vertex 0	Read property for vertex 1	...	Read property for vertex 63
Compute	Compute	...	Compute
Write position of vertex 0	Write position of vertex 1	...	Write position of vertex 63

The shader – Easy parts

Read property for vertex 0 | Read property for vertex 1 | ... | Read property for vertex 63

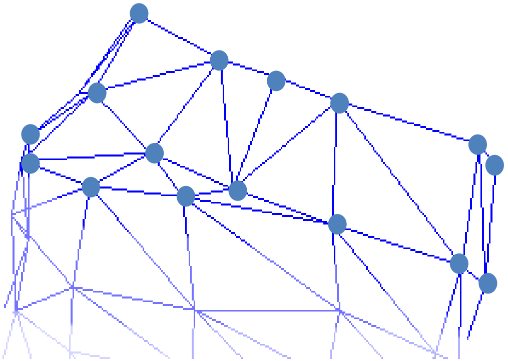
Ensure contiguous reads to get good performance

Coalescing = 1 read instead of 16

i.e. use Structure of Arrays (SoA) instead of Array of Structures (AoS)

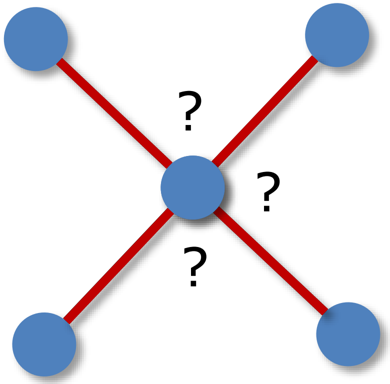
The shader – Complex parts

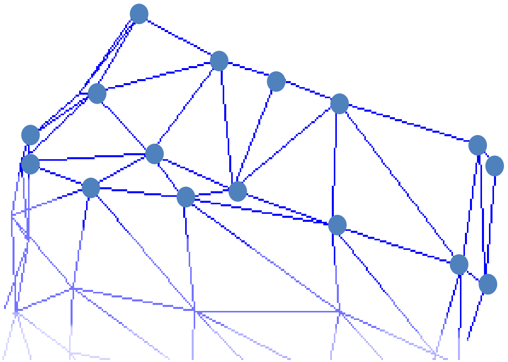
- A binary constraint modifies the position of 2 vertices



The shader – Complex parts

- Binary constraints:





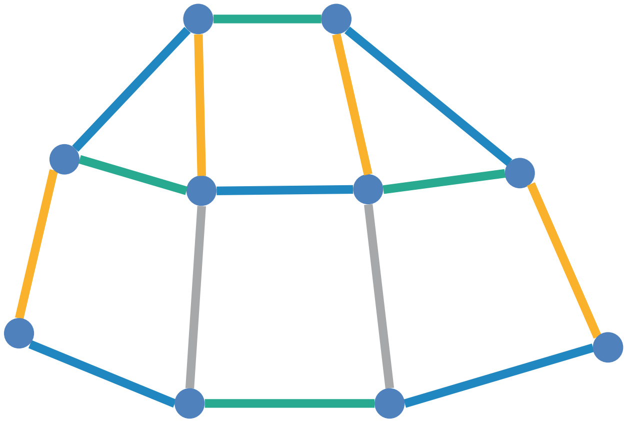
- 4 constraints updating the position of the same vertex

  → 4 threads reading and writing at the same location

  → Undefined behavior

The shader – Complex parts

- Binary constraints:



Group 1

GroupMemoryBarrierWithGroupSync()

Group 2

GroupMemoryBarrierWithGroupSync()

Group 3

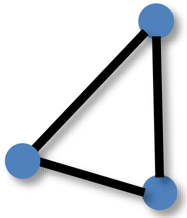
GroupMemoryBarrierWithGroupSync()

Group 4

The shader – Complex parts

- Collisions: Easy or not?
  - Collisions with vertices → Easy
  - Collisions with triangles

→ Each thread will modify the position of 3 vertices



→ You have to create groups and add synchronization

Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

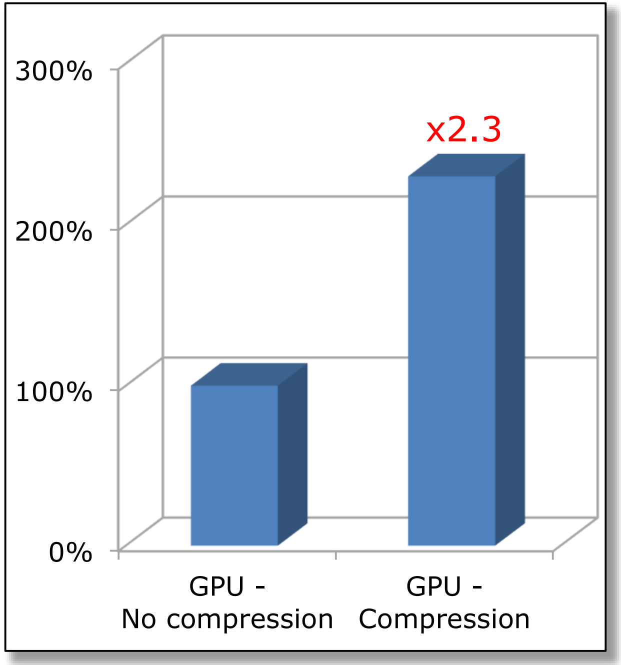
Optimizing the shader

- General rule:

Bottleneck = memory bandwidth

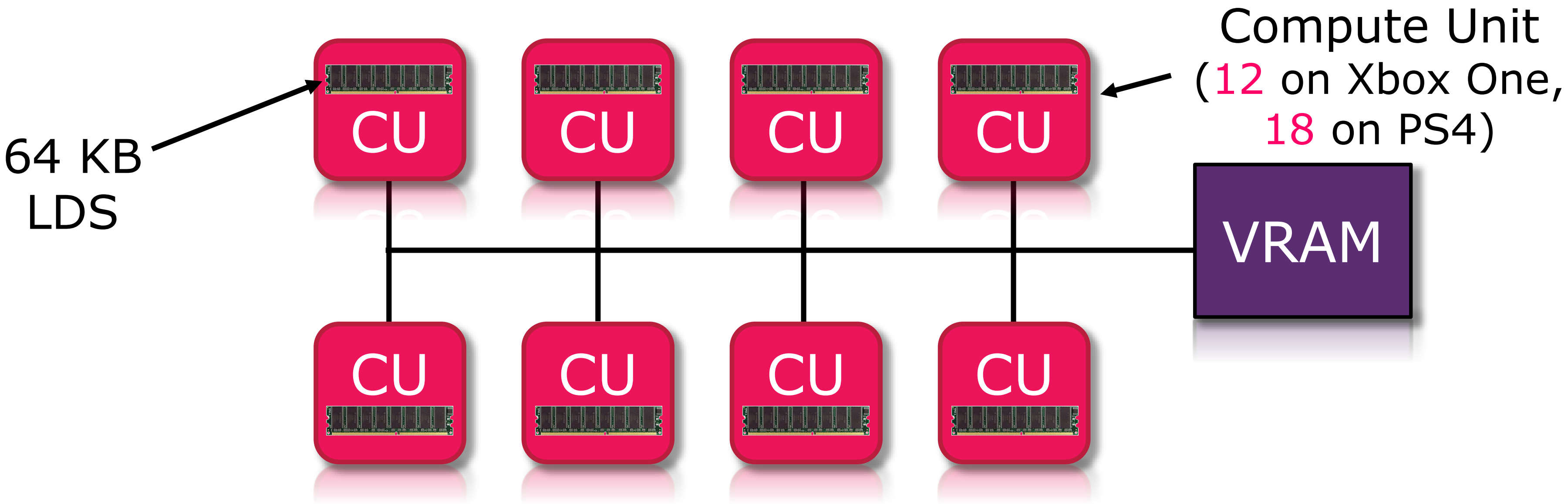
- Data compression:

	CPU	GPU
Vertex	128 bits (4 floats)	64 bits (21:21:21:1)
Normal	128 bits (4 floats)	32 bits (10:10:10)



Optimizing the shader

- Use Local Data Storage (aka Local Shared Memory)



Optimizing the shader

- Store vertices in Local Data Storage

Copy vertices from VRAM to LDS

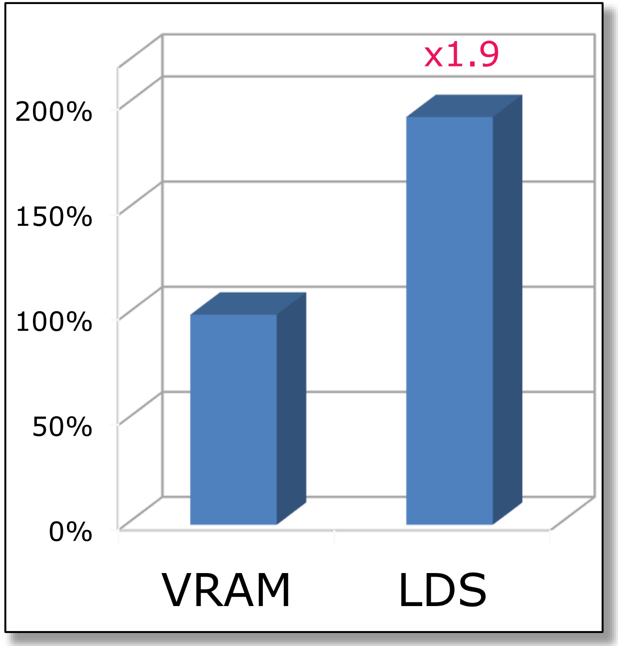
Step 1 – Update vertices

Step 2 – Update vertices

...

Step n – Update vertices

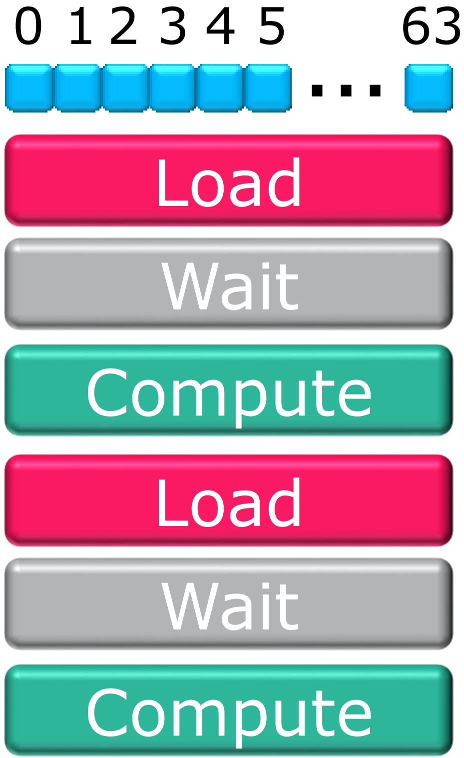
Copy vertices from LDS to VRAM



Optimizing the shader

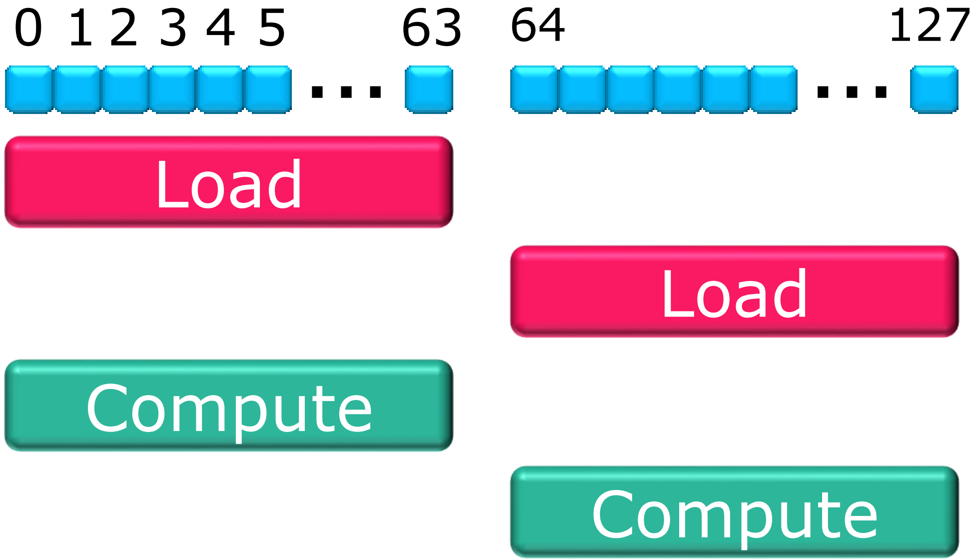
Use bigger thread groups:

- With 64 threads, the GPU is waiting for the memory most of the time



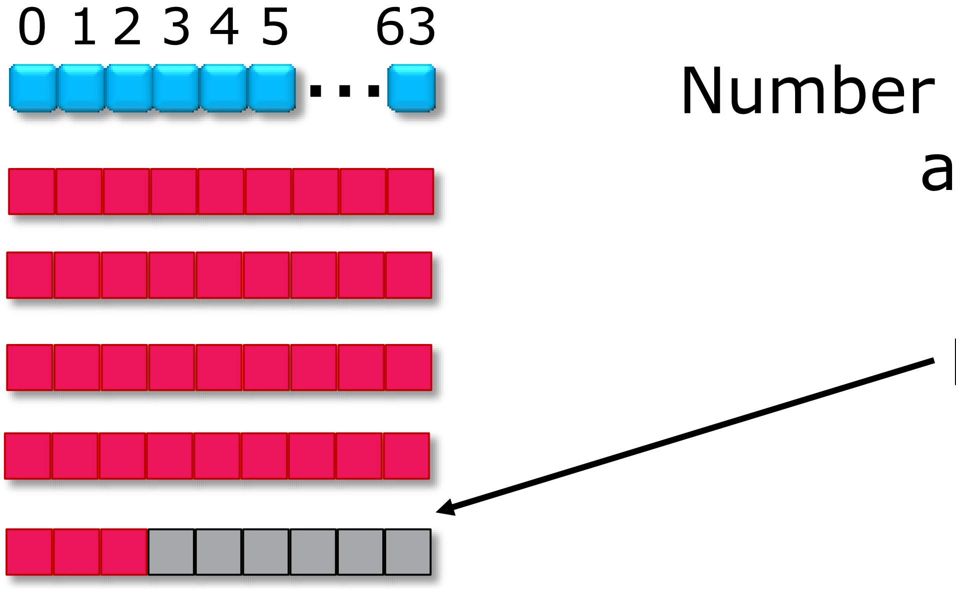
Optimizing the shader

Use bigger thread groups:



- With 256 or 512 threads, we hide most of the latency!
- But…

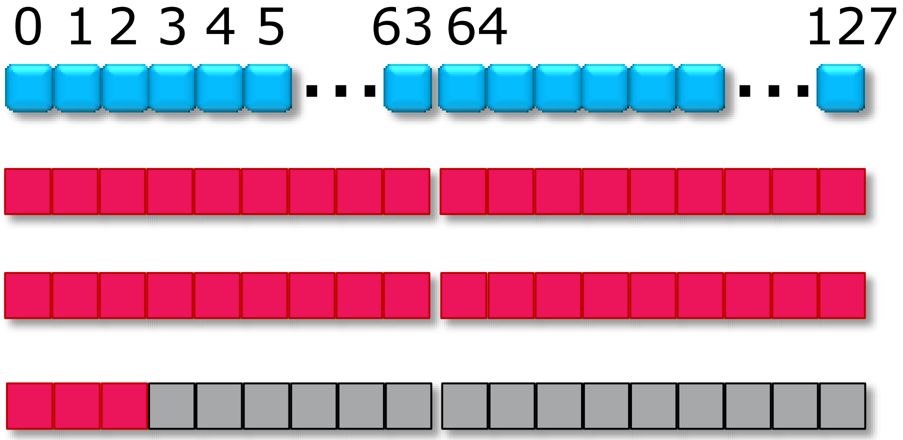
Optimizing the shader



Number of vertices usually not a multiple of 64

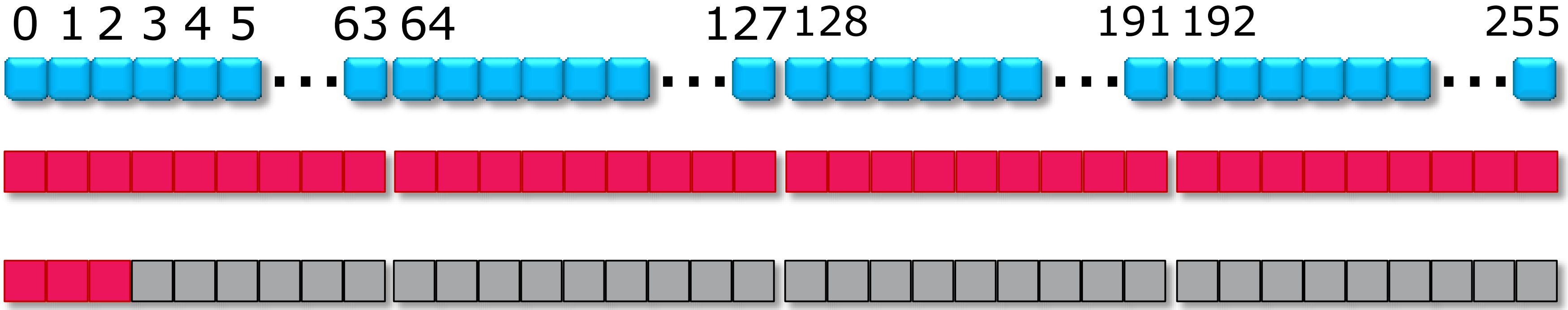
Dummy vertices
=
Useless work!

Optimizing the shader



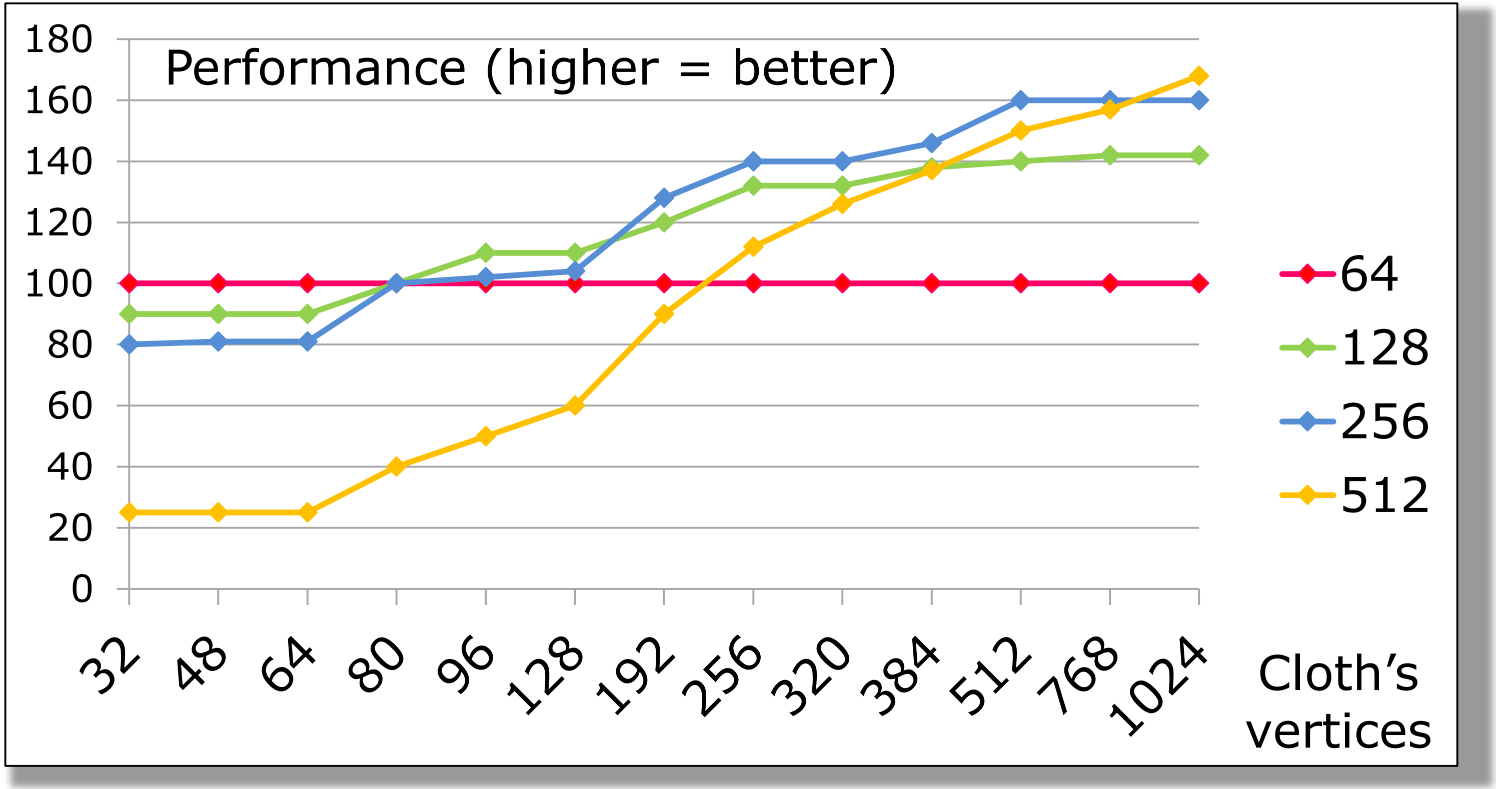
Bigger thread group = more dummy vertices

Optimizing the shader



Bigger thread group = more dummy vertices

Optimizing the shader

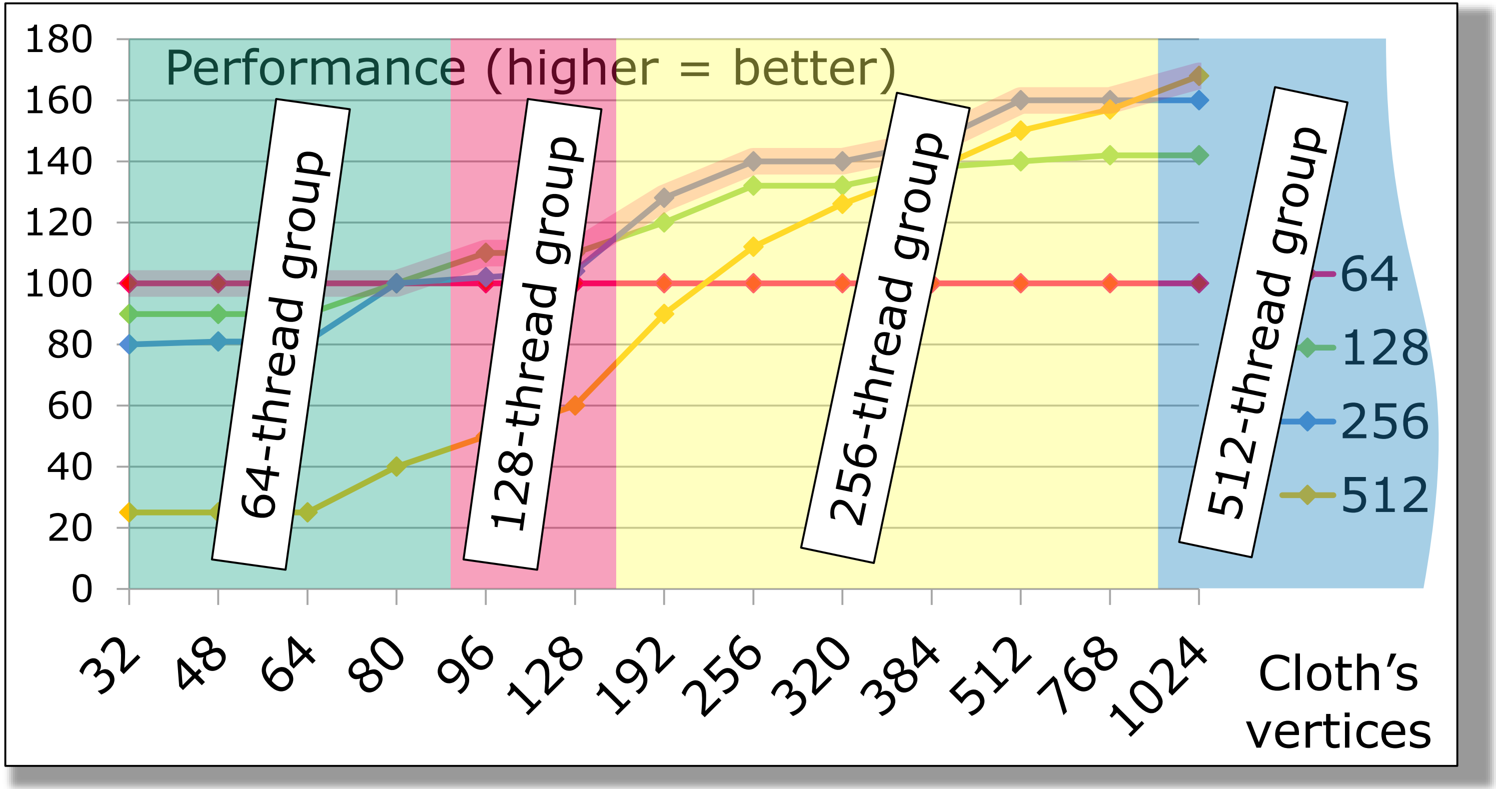


Optimizing the shader

To get the best performance:

- Use several shaders with different thread group sizes
- Use the most efficient shader depending on the number of vertices of the cloth

Optimizing the shader



Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

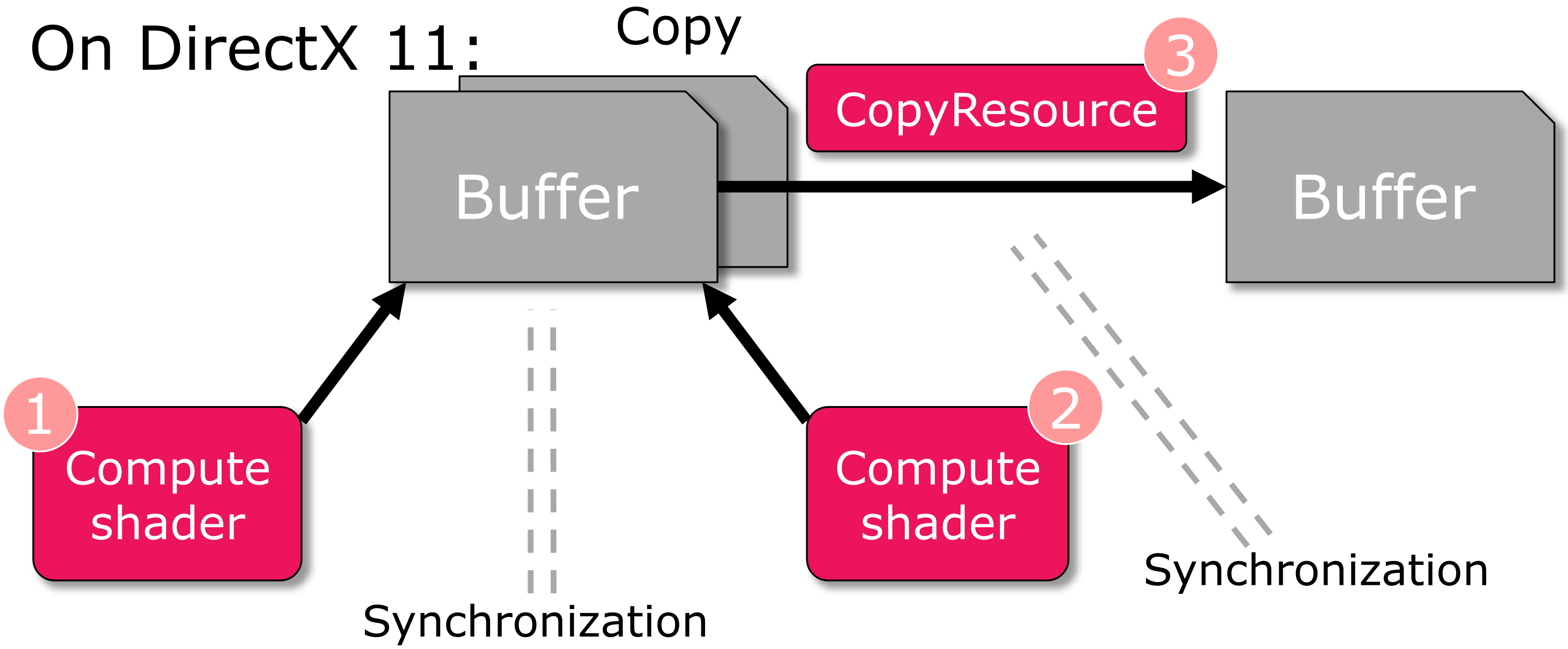
The PS4 version

- Porting from HLSL to PSSL is easy:

```
#ifdef __PSSL__
   #define numthreads NUM_THREADS
   #define SV_GroupIndex S_GROUP_INDEX
   #define SV_GroupID S_GROUP_ID
   #define StructuredBuffer RegularBuffer
   #define RWStructuredBuffer RW_RegularBuffer
   #define ByteAddressBuffer ByteBuffer
   #define RWByteAddressBuffer RW_ByteBuffer
   #define GroupMemoryBarrierWithGroupSync ThreadGroupMemoryBarrierSync
   #define groupshared thread_group_memory
#endif
```

The PS4 version

- On DirectX 11:



The PS4 version

- On PS4:

− No implicit synchronization, no implicit buffer duplication
You have to manage everything by yourself

+ Potentially better performance because you know when you have to sync or not

Also available on Xbox One
(use fast semantics contexts)

The PS4 version

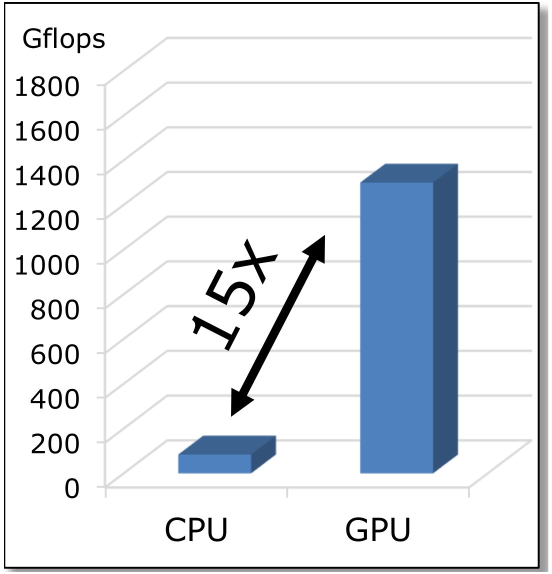
- We use labels to know if a buffer is still in use by the GPU
- Still used → Automatically allocate a new buffer
- "Used" means used by a compute shader or a copy
- We also use labels to know when a compute shader has finished, to copy the results

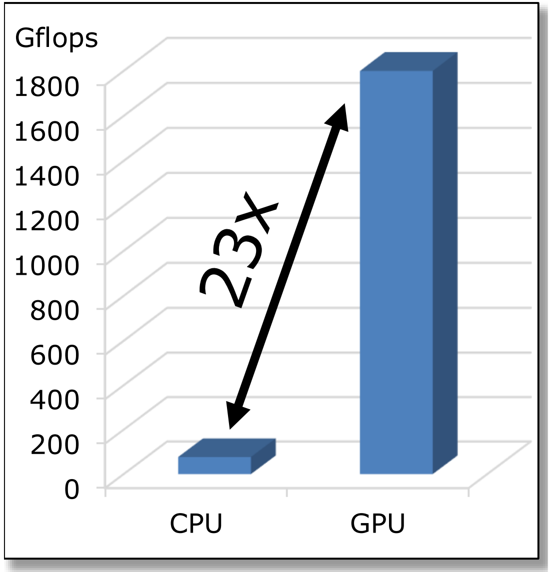
Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

What you can do in compute shader

Peak power: Xbox One PS4





What you can do in compute shader

- ➕ Using DirectCompute, you can do almost everything in compute shader
- ➖ The difficulty is to get good performance

What you can do in compute shader

- Efficient code = you work on 64+ data at a time
- If you have less data:

```
if (threadIndex < 32)
{
   ...
};
```

```
if (threadIndex == 0)
{
   ...
};
```

```
// Read the same data on all threads
// All threads do the same computation
// They write the same result
...
```

But this is likely to be the bottleneck of the shader!

What you can do in compute shader

- Example: collisions
- On the CPU:

Compute a bounding volume
(ex: Axis-Aligned Bounding Box)

Use it for an early rejection test

Use an acceleration structure
(ex: AABB Tree) to improve performance

What you can do in compute shader

- Example: collisions
- On the GPU:

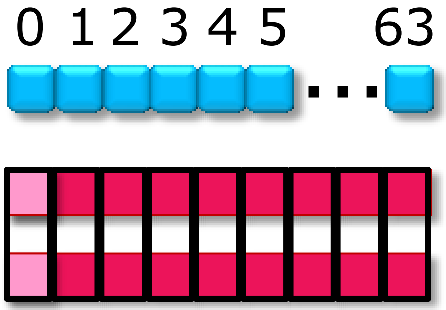
Compute a bounding volume
(ex: Axis-Aligned Bounding Box)

↑

Just doing this can be more costly than computing the collision with all vertices!!!

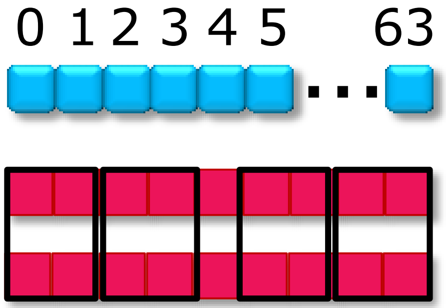
What you can do in compute shader

- Compute 64 sub-AABoxes



What you can do in compute shader

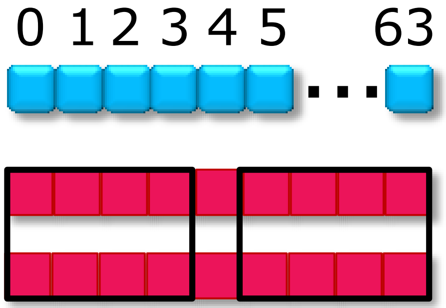
- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes



We use only 32 threads for that

What you can do in compute shader

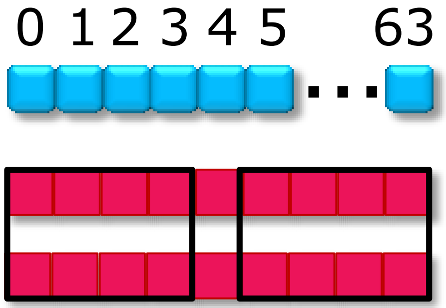
- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes



We use only 16 threads for that

What you can do in compute shader

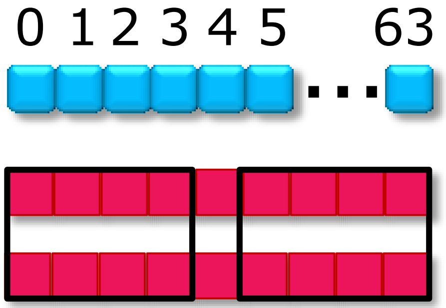
- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes



We use only 8 threads for that

What you can do in compute shader

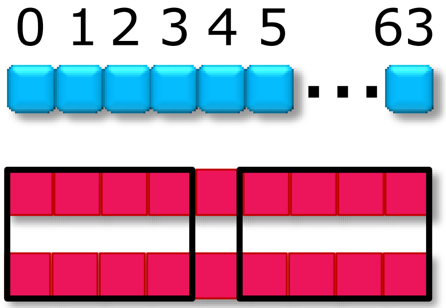
- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes



We use only 4 threads for that

What you can do in compute shader

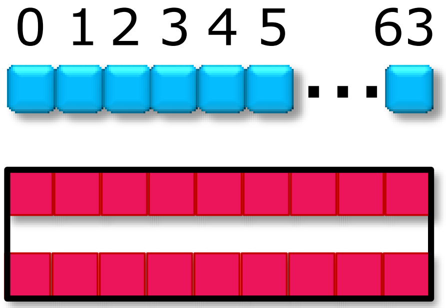
- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes



We use only 2 threads for that

What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes
- Reduce down to 1 AABox



We use a single thread for that

What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes
- Reduce down to 1 AABox

This is ~ as costly as computing the collision with 7 x 64 = 448 vertices!!

What you can do in compute shader

- Atomic functions are available

→ You can write lock-free thread-safe containers

- Too costly in practice

→ The brute-force approach is almost always the fastest one

What you can do in compute shader

Conclusion:

> Port an algorithm to the GPU
> only if you find a way
> to handle 64+ data at a time
> 95+% of the time

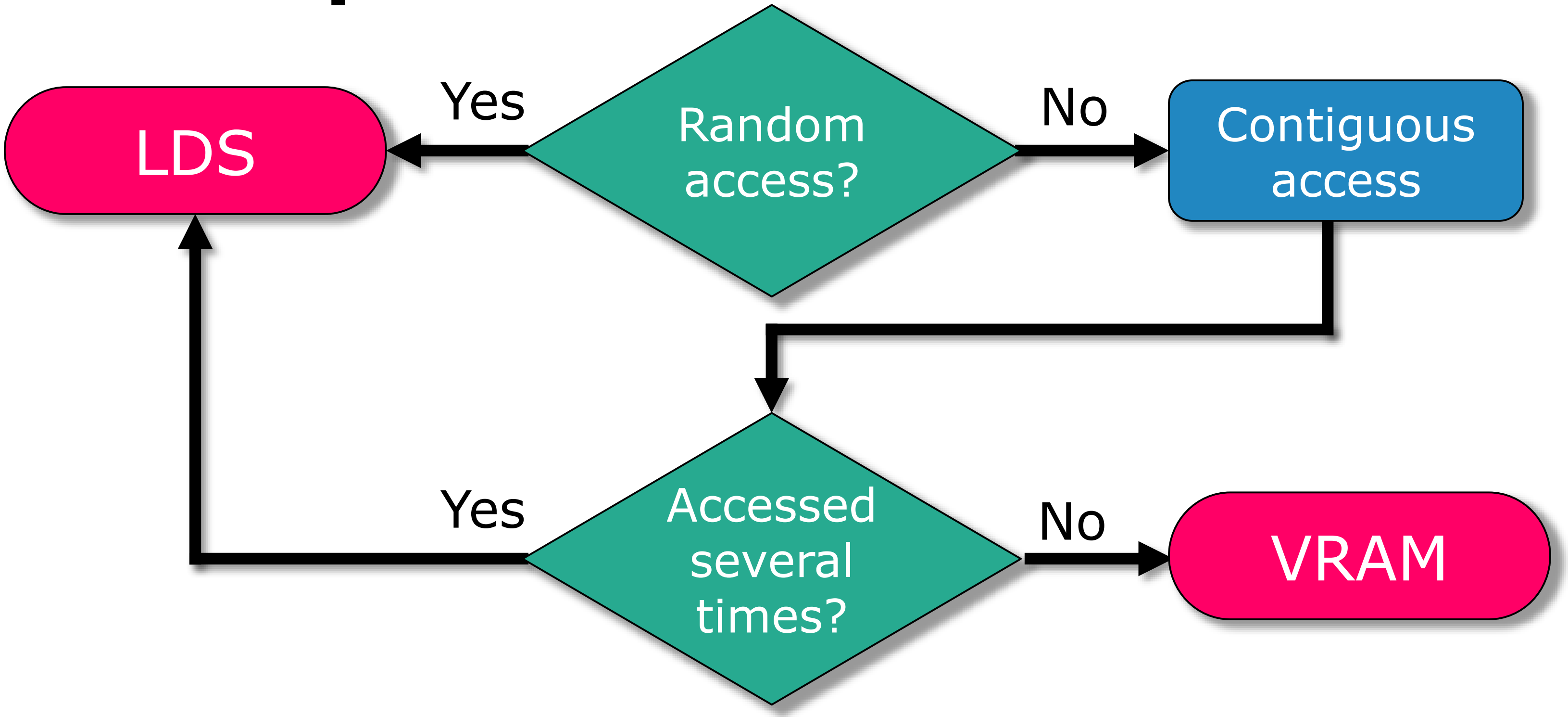
Journey from C++ to Compute Shaders

- The first attempts
- A new approach
- The shader – Easy parts – Complex parts
- Optimizing the shader
- The PS4 version
- What you can & cannot do in compute shader
- Tips & Tricks

Sharing code between C++ & hlsl

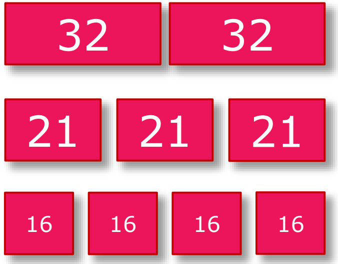
```
#if defined( _WIN32)      || defined(_WIN64)
  || defined(_DURANGO) || defined(__ORBIS__)
    typedef unsigned long uint;
    struct float2 { float x, y;        };
    struct float3 { float x, y, z;     };
    struct float4 { float x, y, z, w;  };
    struct uint2  { uint x, y;         };
    struct uint3  { uint x, y, w;      };
    struct uint4  { uint x, y, z, w;   };
#endif
```

What to put in LDS?



Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB



  2 thread groups can run simultaneously on the same compute unit

- Less memory used in LDS

  More thread groups can run in parallel

Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB
- Less memory used in LDS

➔ More thread groups can run in parallel

- 256- or 512-thread groups: No visible impact
- 64- or 128-thread groups:
  Visible impact on performance

Optimizing bank access in LDS?

- LDS is divided into several banks (16 or 32)
- 2 threads accessing the same bank → Conflict

→ Visible impact on performance on older PC hardware

→ Negligible on Xbox One, PS4 and newer PC hardware

Beware the compiler

```
CopyFromVRAMToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

//CopyFromLDSToVRAM();
```

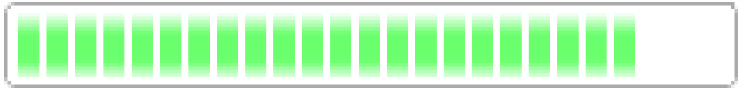
Beware the compiler

```
CopyFromVRAMToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

CopyFromLDSToVRAM();
```



The last copy
takes all the time

This doesn't
make sense!

Beware the compiler

```
CopyFromVRAMToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

//CopyFromLDSToVRAM();
```

- Data written in LDS are never used
- The shader compiler detects it

➔ It removes the entire code

Optimizing compilation time

```
float3 fanBlades[10];
for (uint i = 0; i < 10; ++i)
{
    Vertex fanVertex = GetVerte
    fanBlades[i] = fanVertex.m_
}

float3 normalAccumulator = cross(fanBlades[0], fanBlades[1]);
for (uint j = 0; j < 8; ++j)
{
        float3 triangleNormal = cross(fanBlades[j+1], fanBlades[j+2]);
        uint isTriangleFilled = neighborFan.m_FilledFlags & (1 << j);
        if (isTriangleFilled) normalAccumulator += triangleNormal;
}
```

Shader compilation time	
Loop	**19″**
Manually unrolled	**6″**

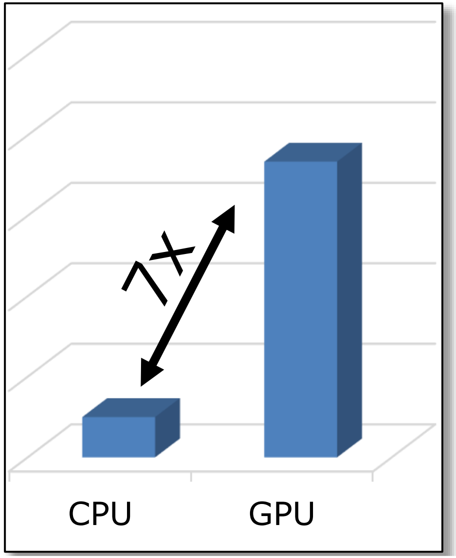
Iteration time

- It's really hard to know which code will run the fastest
- The "best" method:
  - Write 10 versions of your feature
  - Test them
  - Keep the fastest one
- A fast iteration time really helps

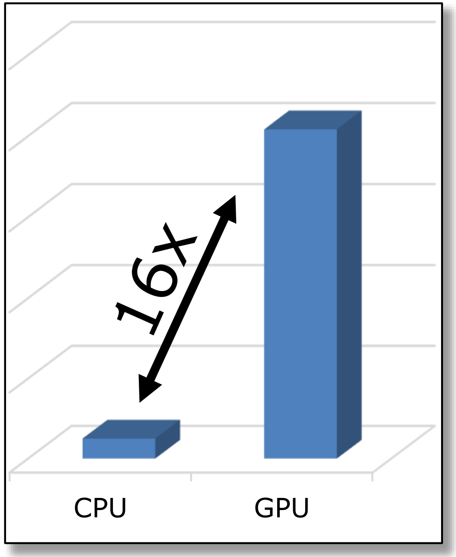
- Loops ordering
- Which data to compress?
- Which data to put in LDS?
- Unroll loops?
- Change data organization?

Bonus: final performance

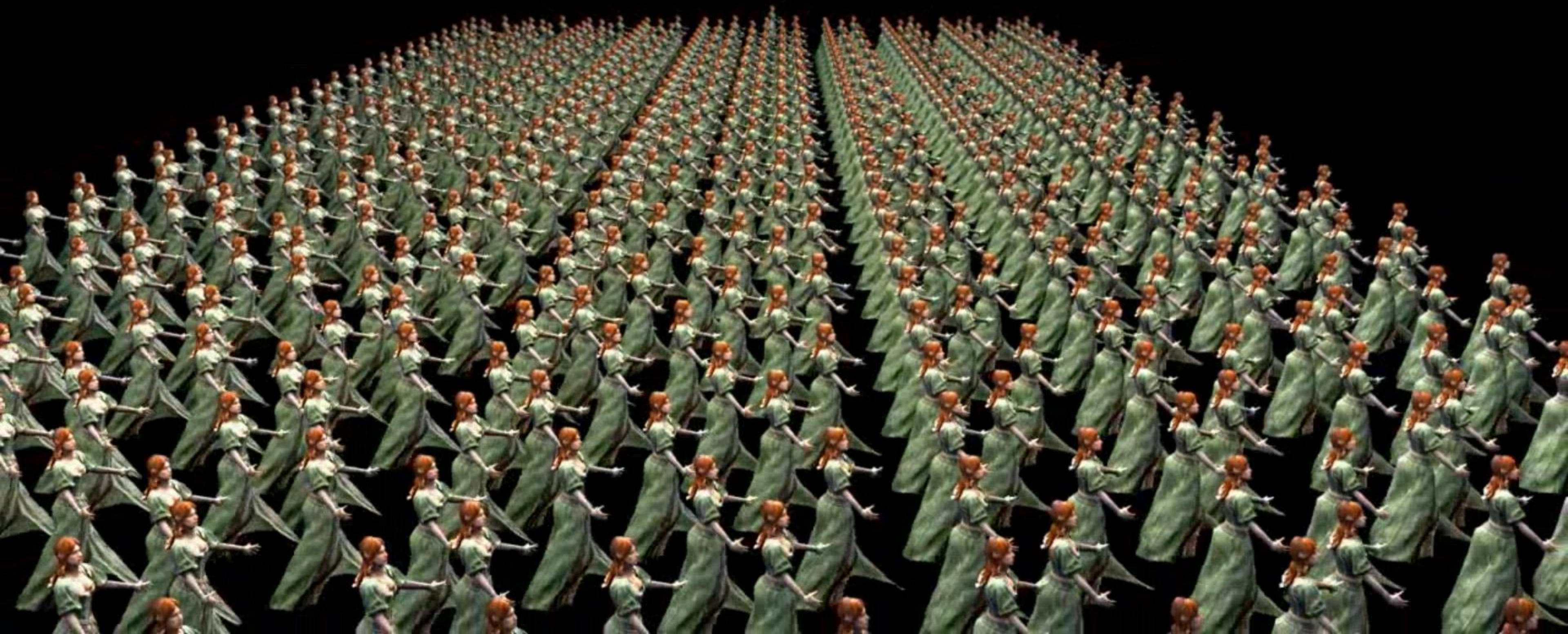
Xbox One



PS4



PS4 – 2 ms of GPU time – 640 dancers



Thank you!

MOTION

UBISOFT®