



Compute for Graphics

Karl Hillesland



Course Objectives

- An introduction to general compute on GPUs
- Understand how compute is related to modern GPU graphics hardware and APIs
- Learn to identify opportunities to apply general compute to graphics

Assumed Knowledge

- Modern graphics hardware and APIs
 - Shader stages
 - Shader language concepts
 - GPU memory resources
 - Basic hardware architecture concepts
- Multithreading concepts
 - Atomics and synchronization primitives
 - Memory coherency
- Real-time rendering techniques

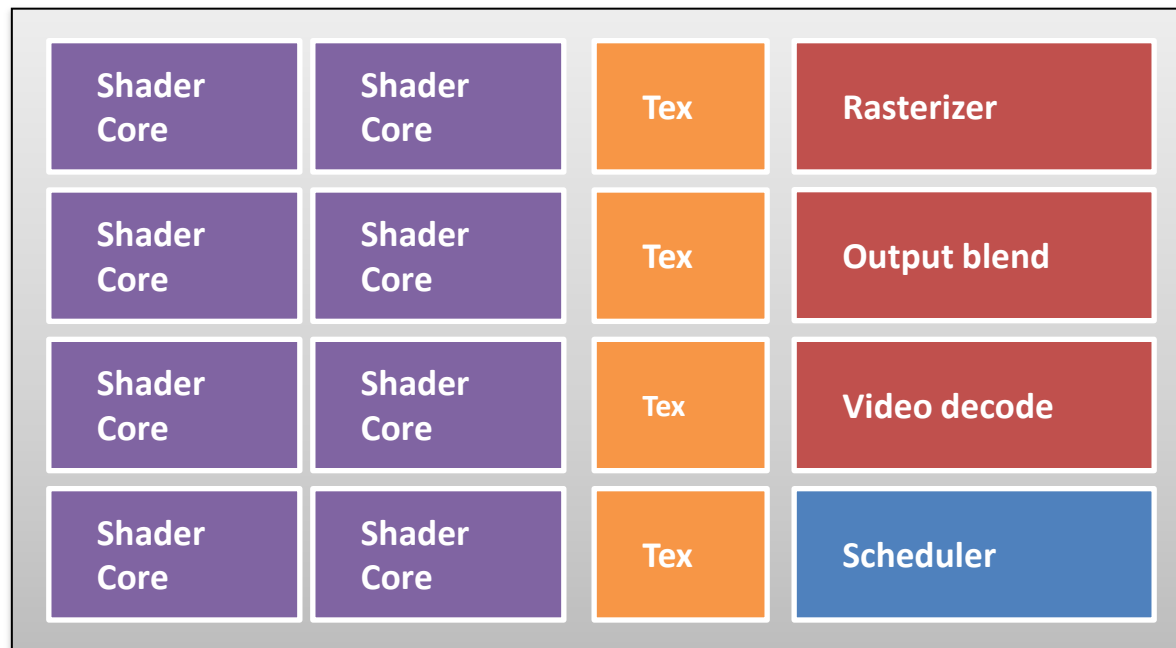
Outline

- GPU Architecture
- Compute Programming Concepts
- Compute APIs
 - “Augmented graphics”
 - DirectCompute, OpenGL Compute
 - OpenCL, CUDA
- Use Cases

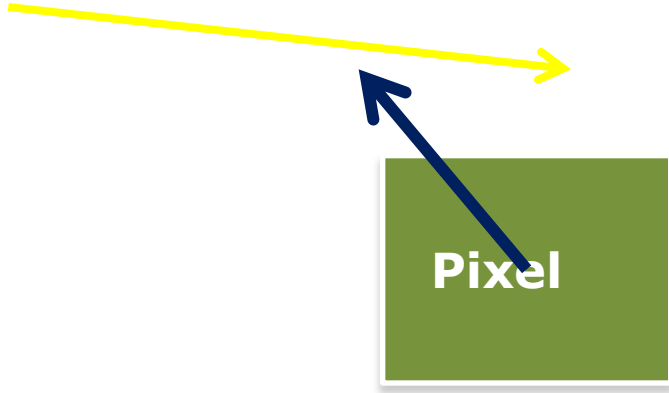
GPU ARCHITECTURE

Typical GPU

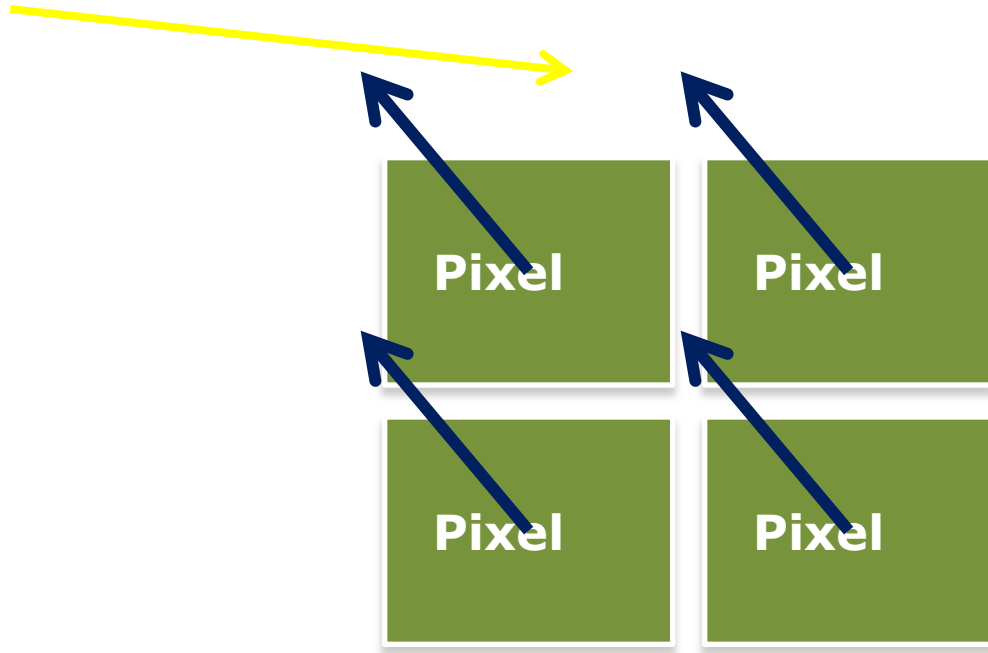
- The GPU is a multicore processor optimized for graphics workloads



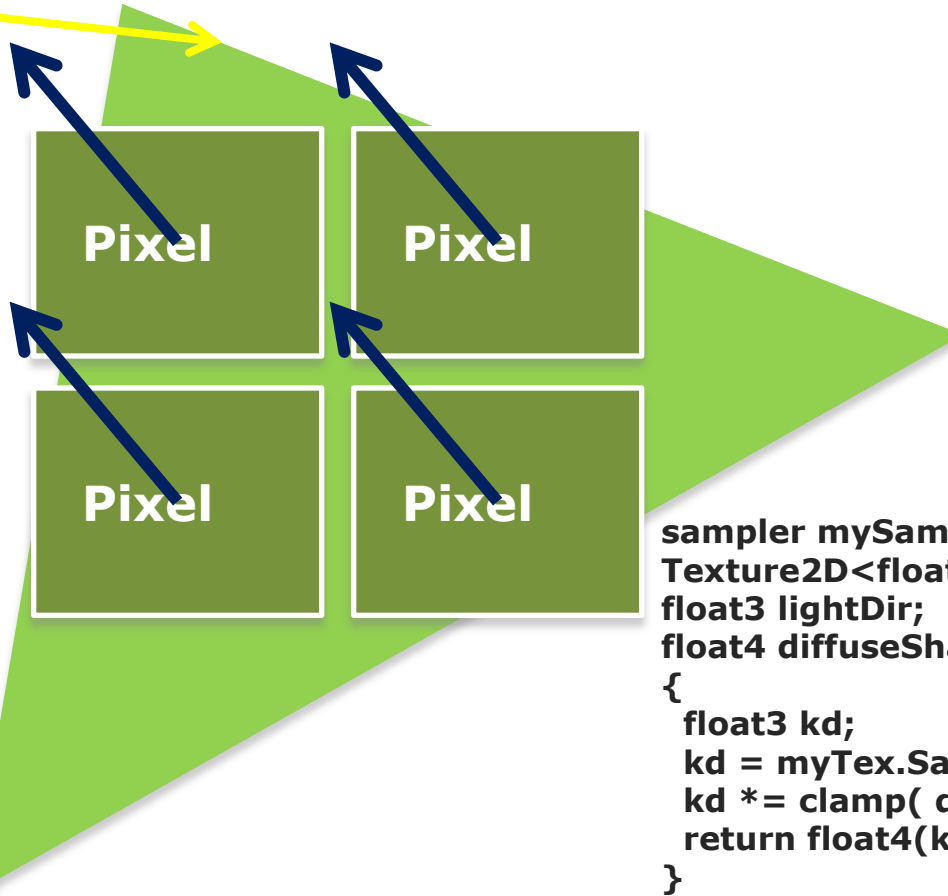
Processing pixels



Processing pixels

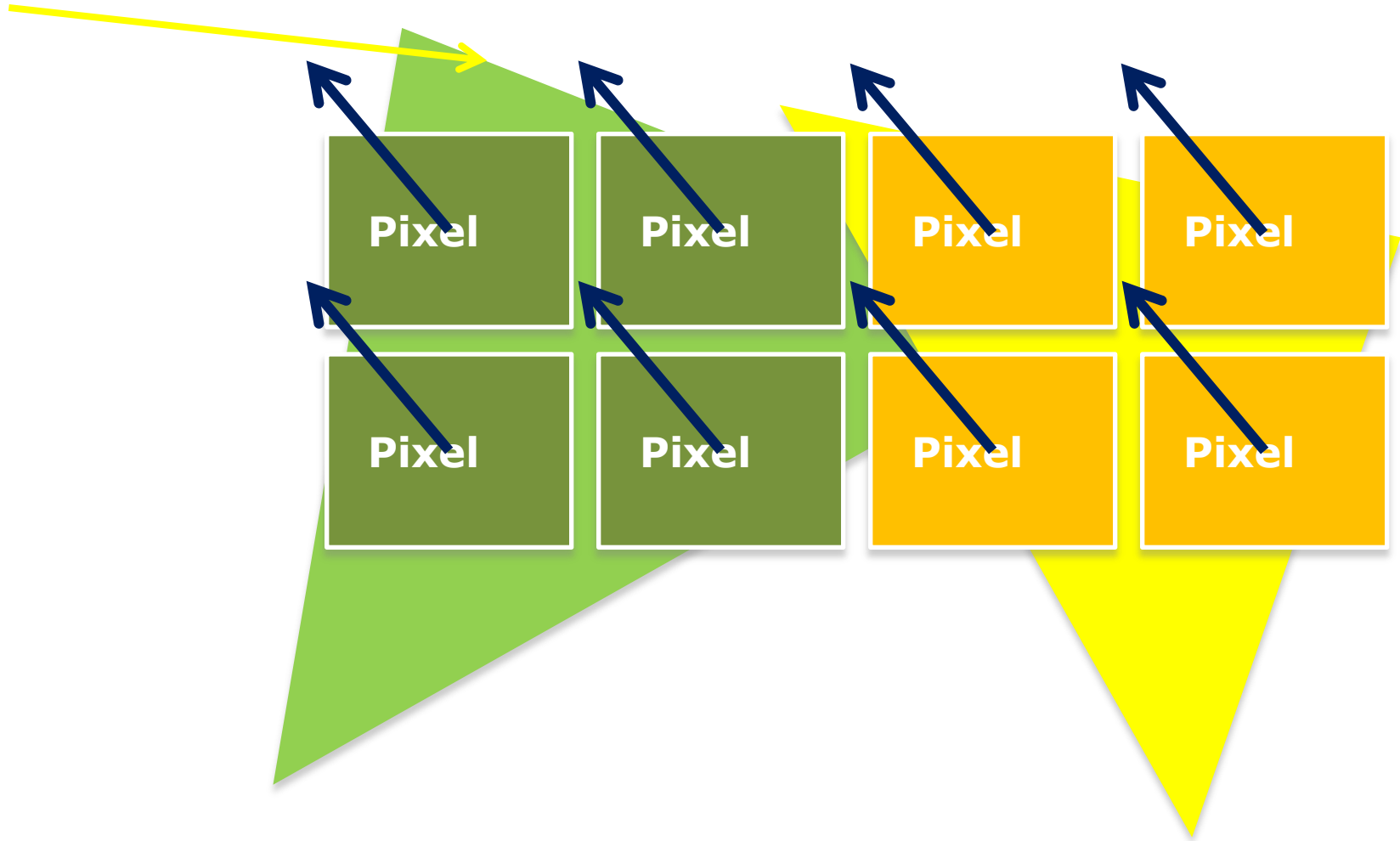


Processing pixels



```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

Processing pixels

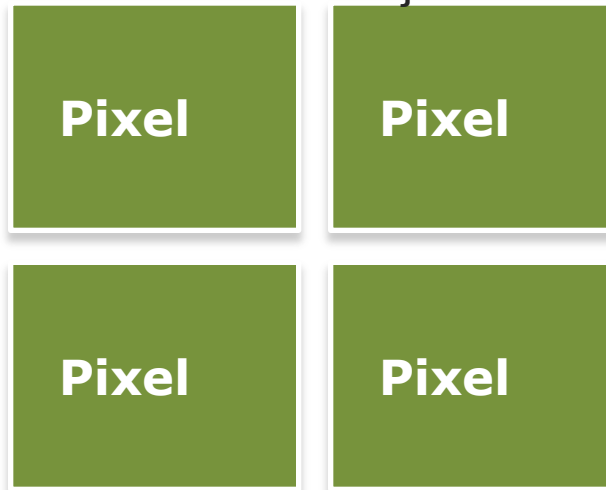


SIMD execution and its implications

SIMD pixel execution

```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

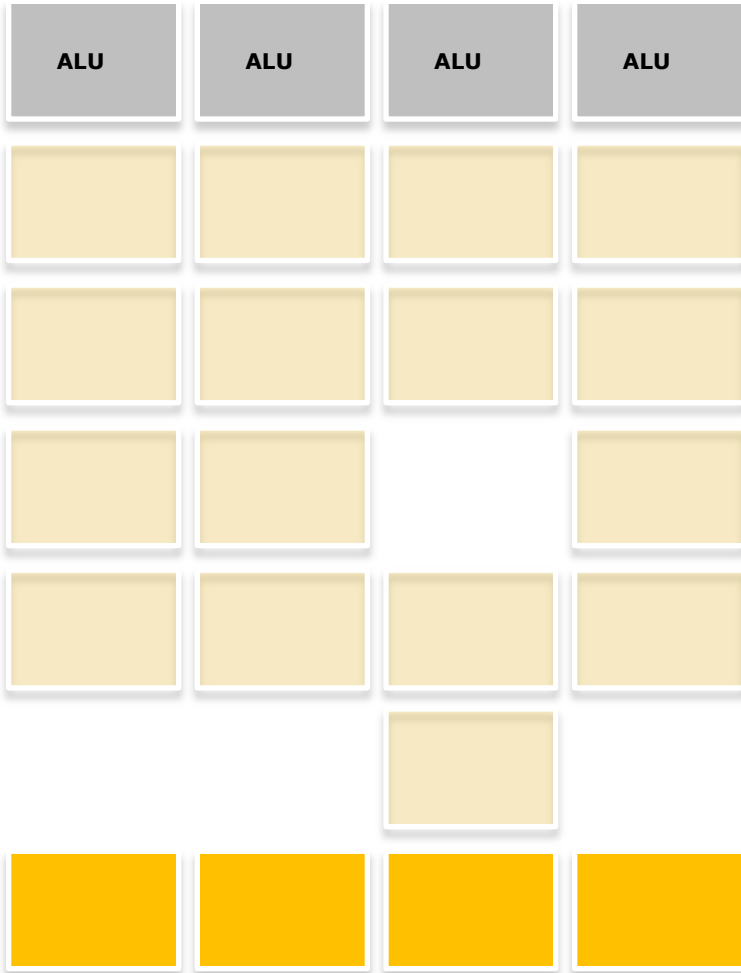
```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```



```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

Branches



```
sampler mySamp;  
Buffer<float> myTex;
```

```
float diffuseShader(  
    float threshold, float index)  
{  
    float brightness = myTex[index];  
    float output;  
    if( brightness > threshold )  
        output = threshold;  
    else  
        output = brightness;  
    return output;  
}
```

```
sampler mySamp;  
Buffer<float> myTex;
```

```
float diffuseShader(  
    float threshold, float index)  
{  
    float brightness = myTex[index];  
    float output;  
    if( brightness > threshold )  
        output = threshold;  
    else  
        output = brightness;  
    return output;  
}
```

```
sampler mySamp;  
Buffer<float> myTex;
```

```
float diffuseShader(  
    float threshold, float index)  
{  
    float brightness = myTex[index];  
    float output;  
    if( brightness > threshold )  
        output = threshold;  
    else  
        output = brightness;  
    return output;  
}
```

```
sampler mySamp;  
Buffer<float> myTex;
```

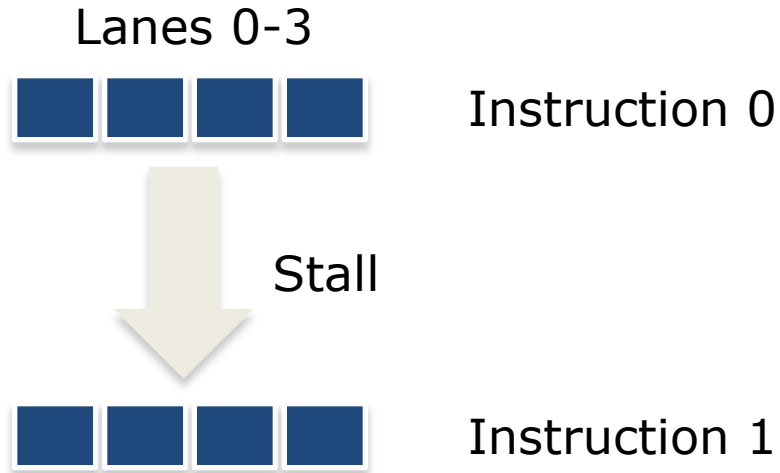
```
float diffuseShader(  
    float threshold, float index)  
{  
    float brightness = myTex[index];  
    float output;  
    if( brightness > threshold )  
        output = threshold;  
    else  
        output = brightness;  
    return output;  
}
```

Why does this matter for Compute?

- Graphics code traditionally has relatively short shaders on large triangles
 - The level of branch divergence overall will not be high
- With graphics code you can not necessarily control it
 - SIMD batches are constructed by the hardware depending on the scene properties.
- For compute you are defining your execution space
 - You choose what work is performed by which work item
 - You choose how to structure your algorithm to avoid this divergence

Throughput execution and latency hiding

Covering pipeline latency



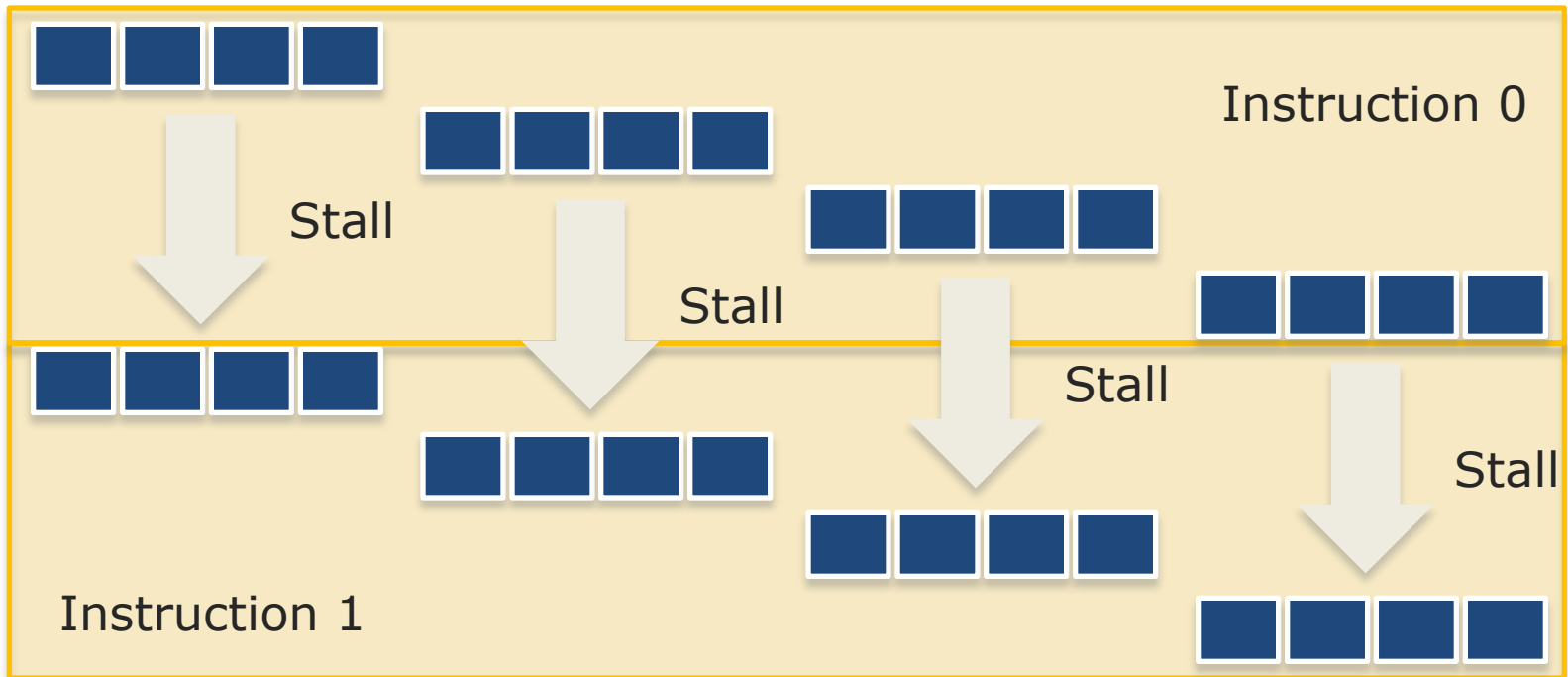
Covering pipeline latency: logical vector

Lanes 0-3

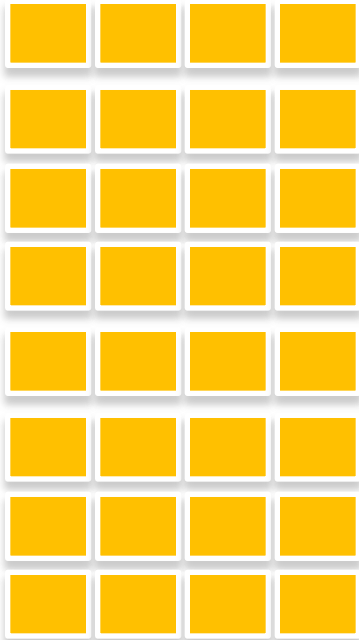
Lanes 4-7

Lanes 8-11

Lanes 12-15



Covering pipeline latency: ALU operations

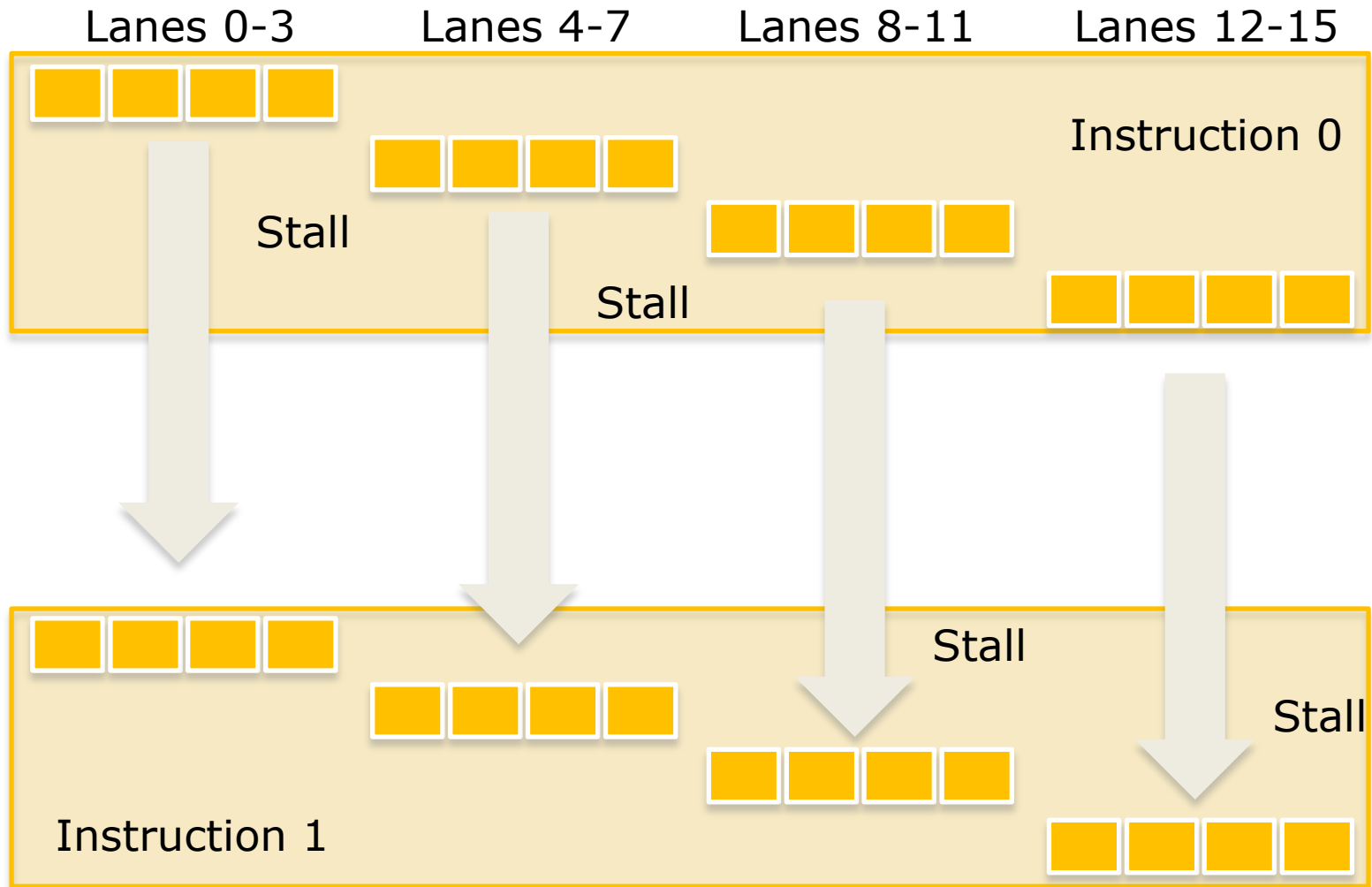


Lanes 0-3	Instruction 0
Lanes 4-7	Instruction 0
Lanes 8-11	Instruction 0
Lanes 12-15	Instruction 0
Lanes 0-3	Instruction 1
Lanes 4-7	Instruction 1
Lanes 8-11	Instruction 1
Lanes 12-15	Instruction 1

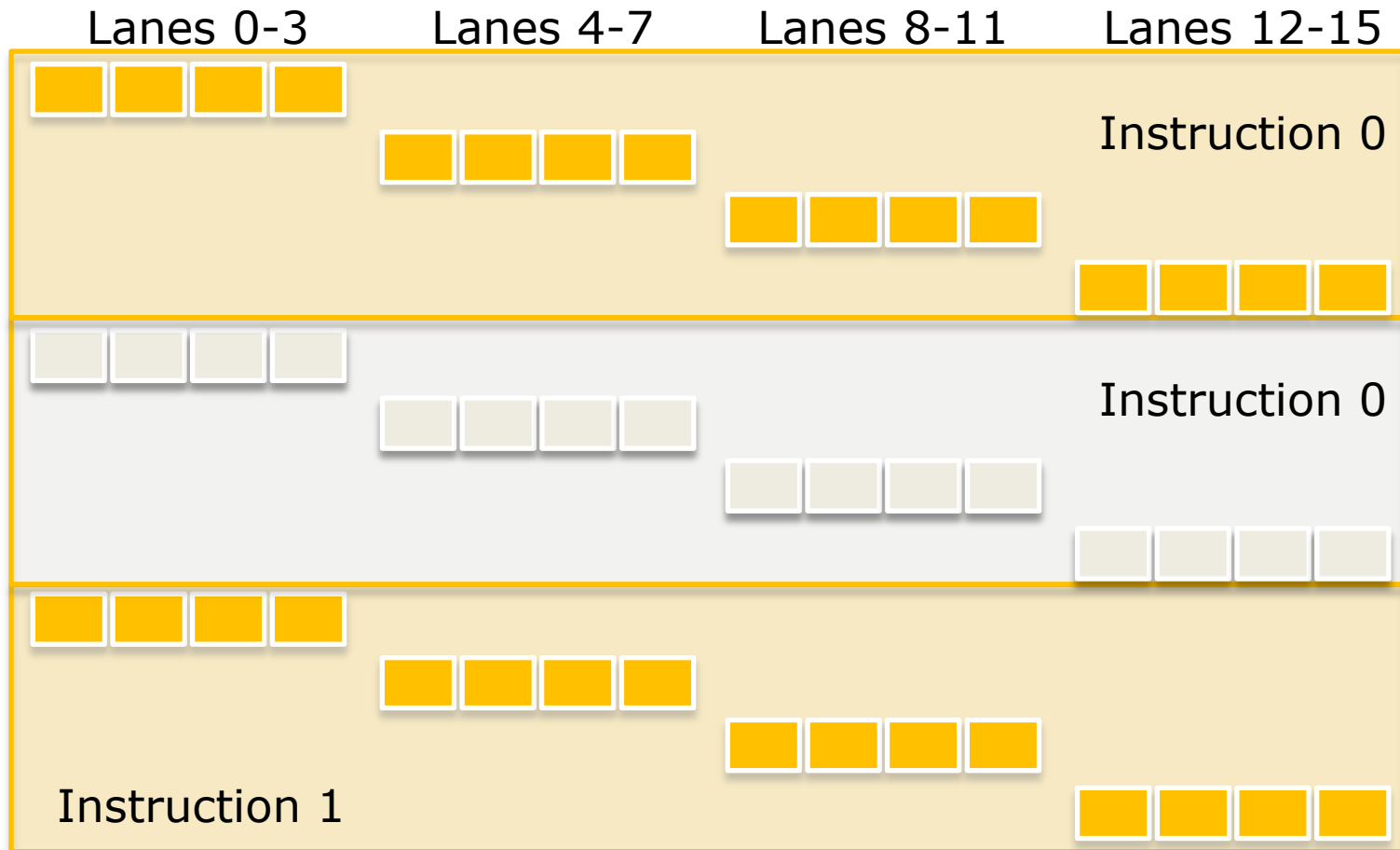
Covering memory latency



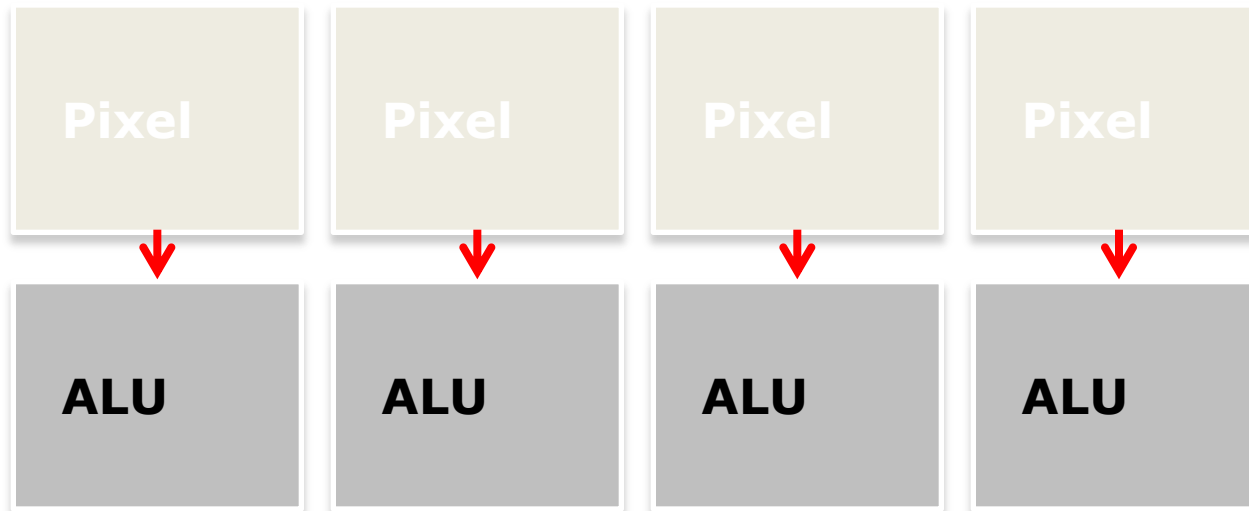
Covering memory latency: we still stall



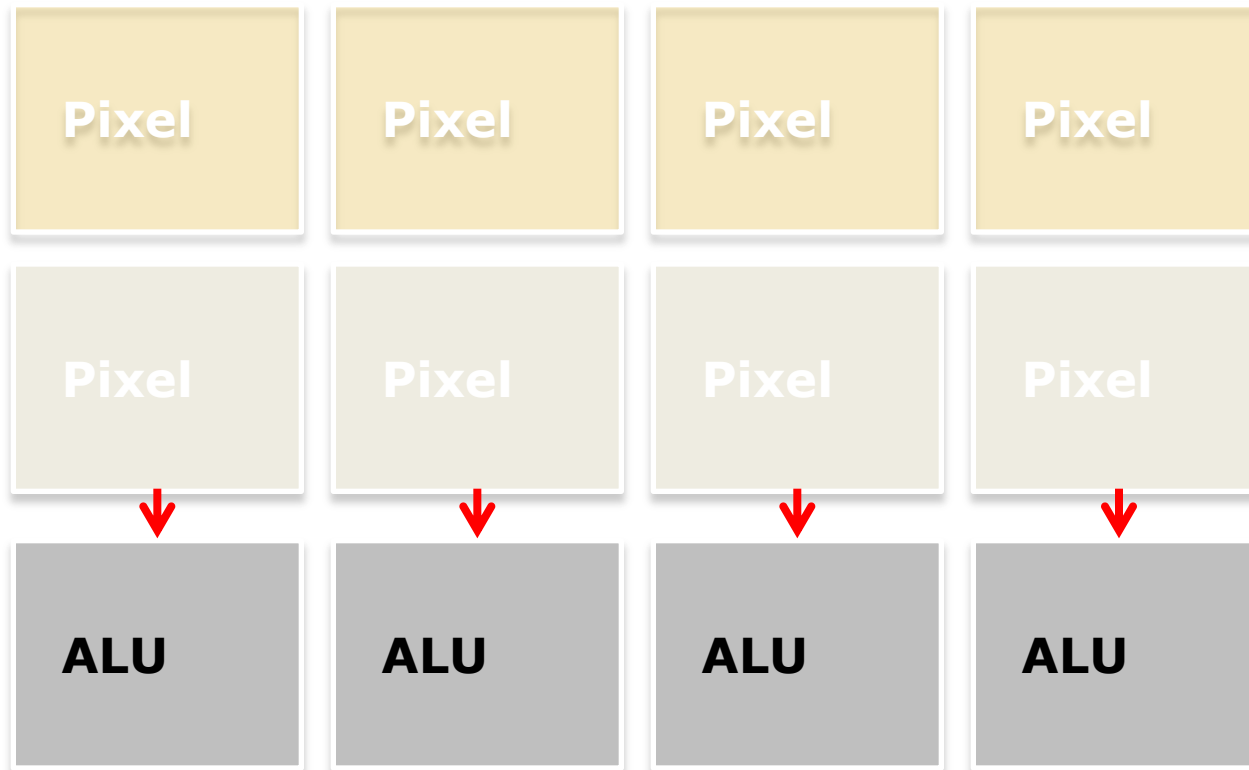
Covering memory latency: another wavefront



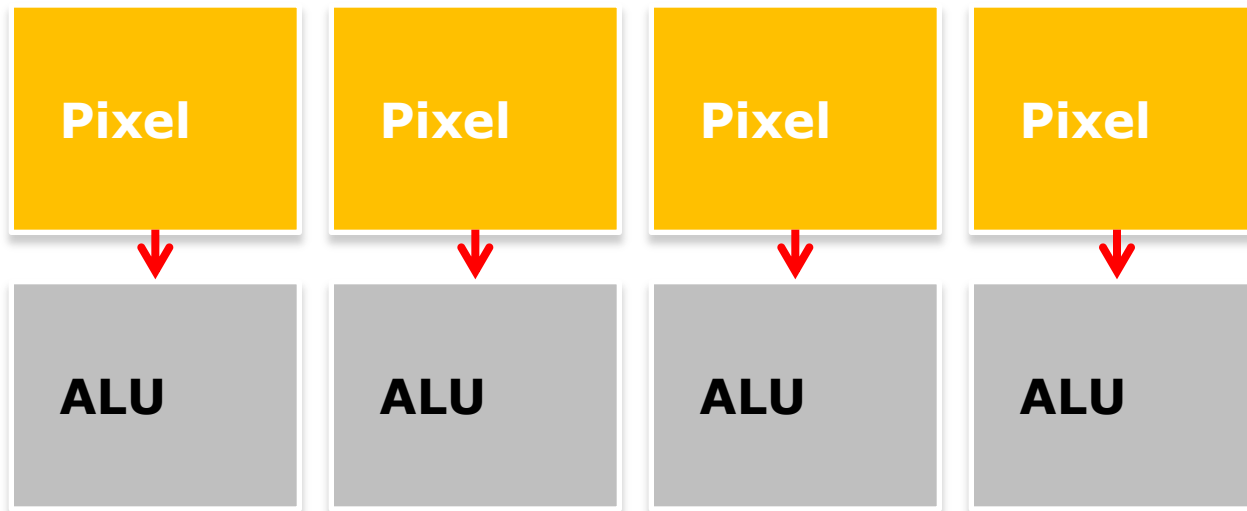
Latency hiding in the SIMD engine



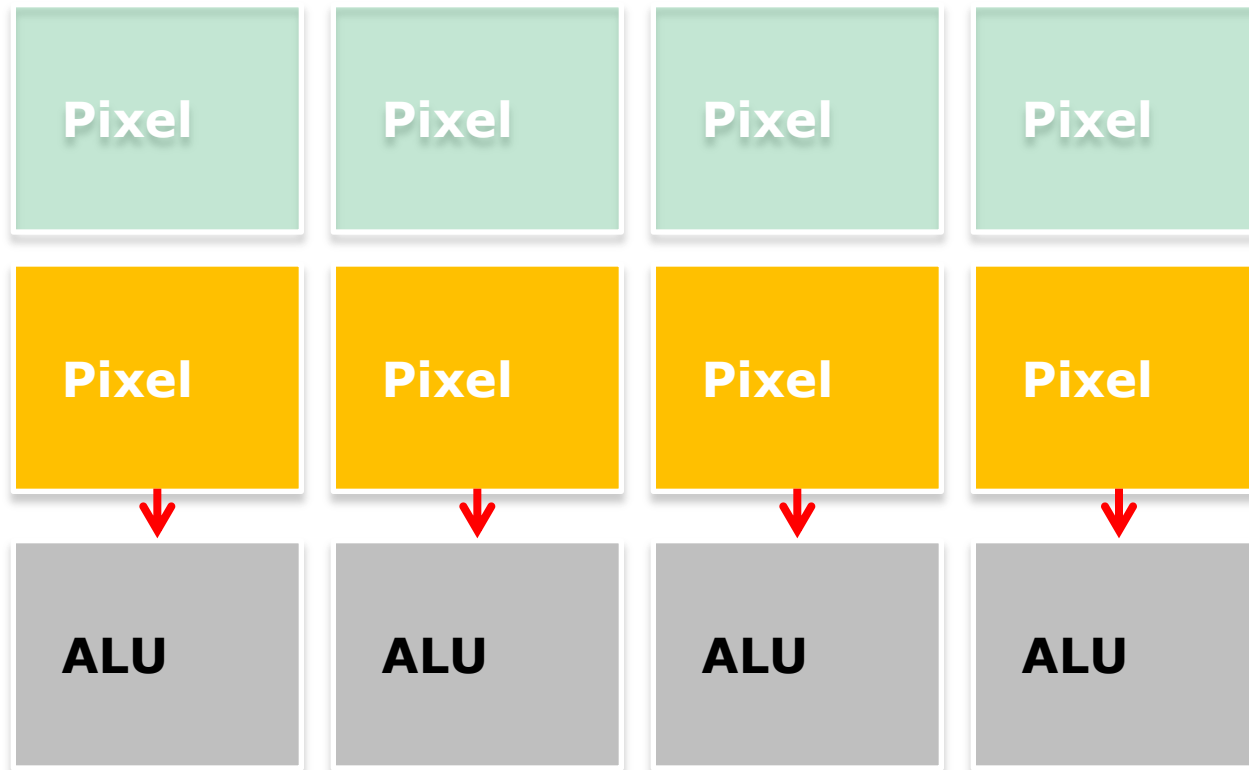
Latency hiding in the SIMD engine



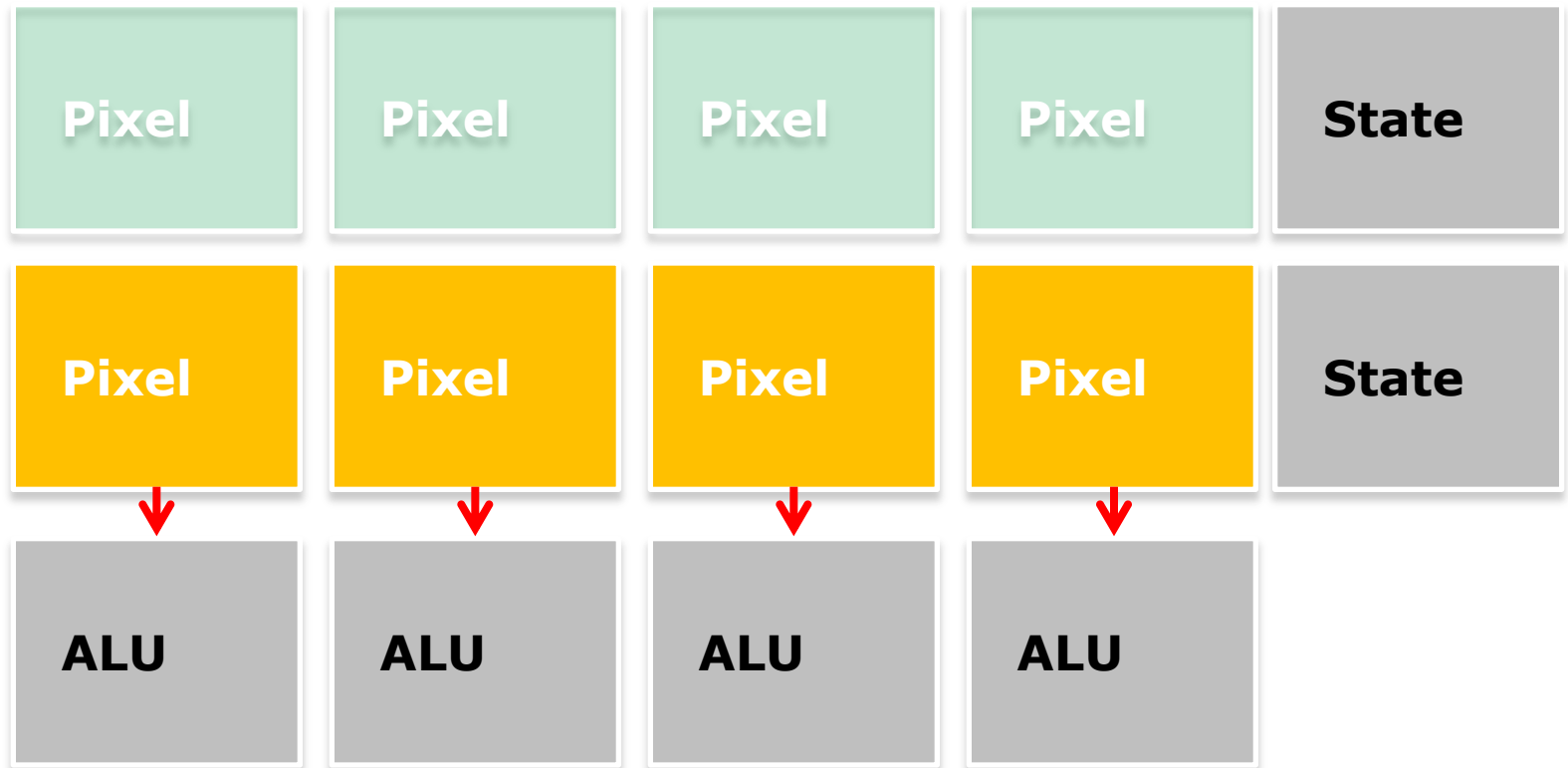
Latency hiding in the SIMD engine



A throughput-oriented SIMD engine



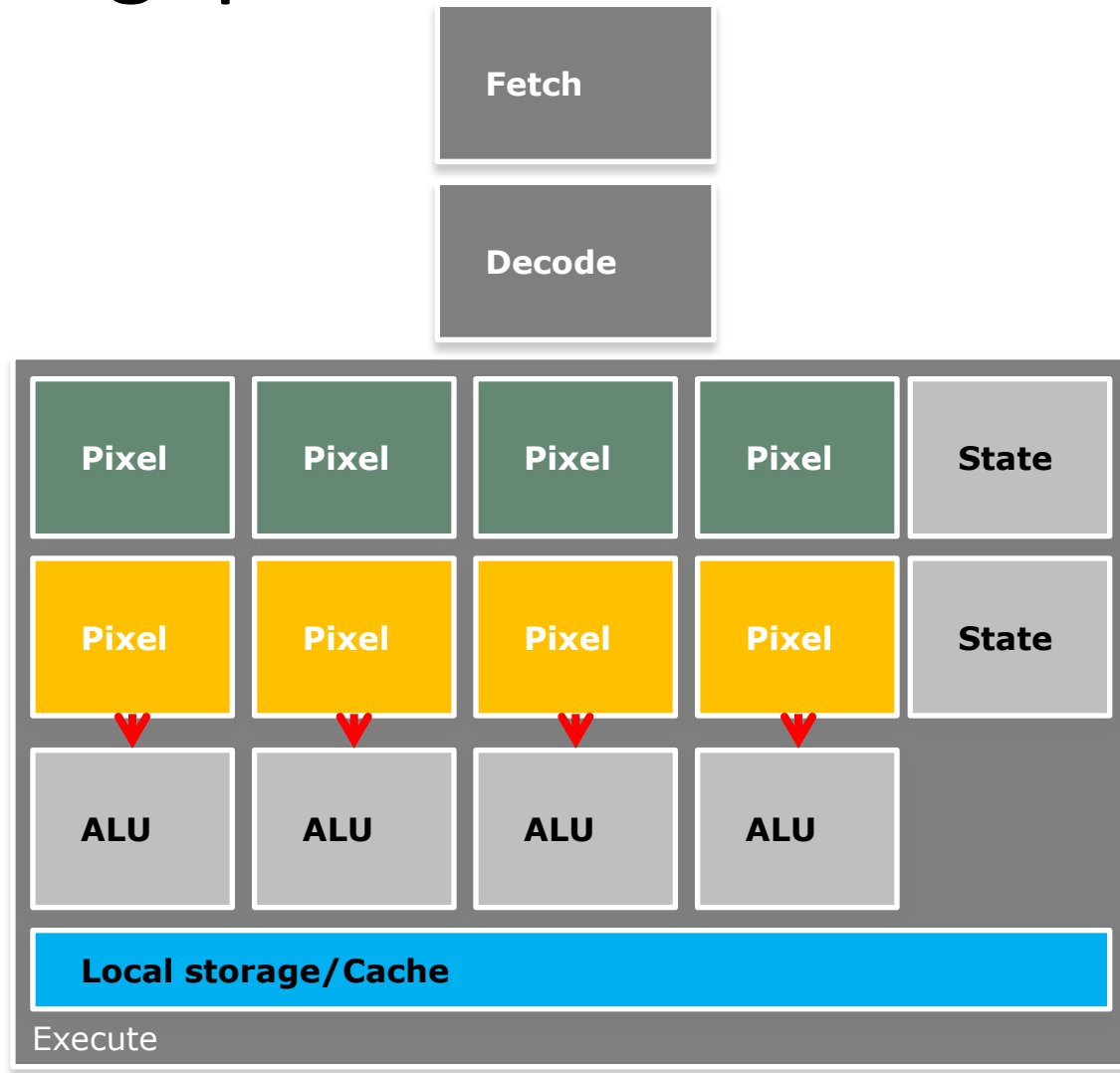
A throughput-oriented SIMD engine



Adding the memory hierarchy

- Unlike most CPUs, GPUs do not have vast cache hierarchies.
 - Caches on CPUs allow primarily for lower access latency
- Heavy multithreading reduces the latency requirement
 - Latency is not an issue, we cover that with other threads
 - Total bandwidth still an issue, even with high-latency high-speed memory interfaces

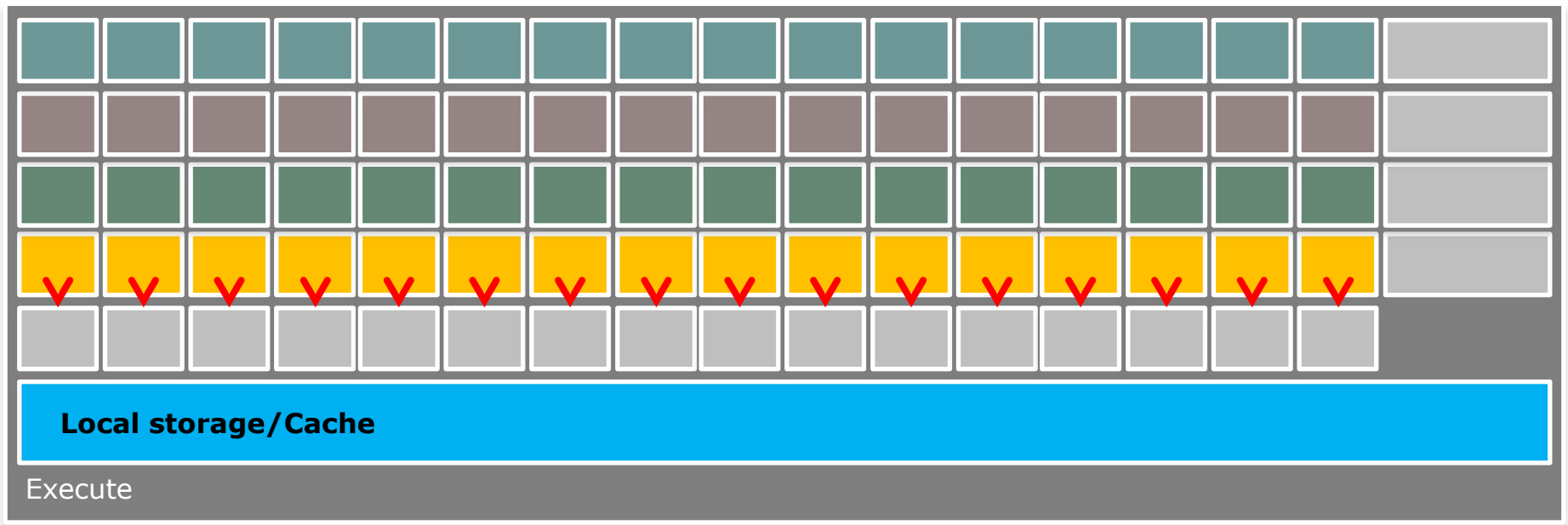
A throughput-oriented SIMD engine



A throughput-oriented SIMD engine

Fetch

Decode



The GPU shader cores



Summary

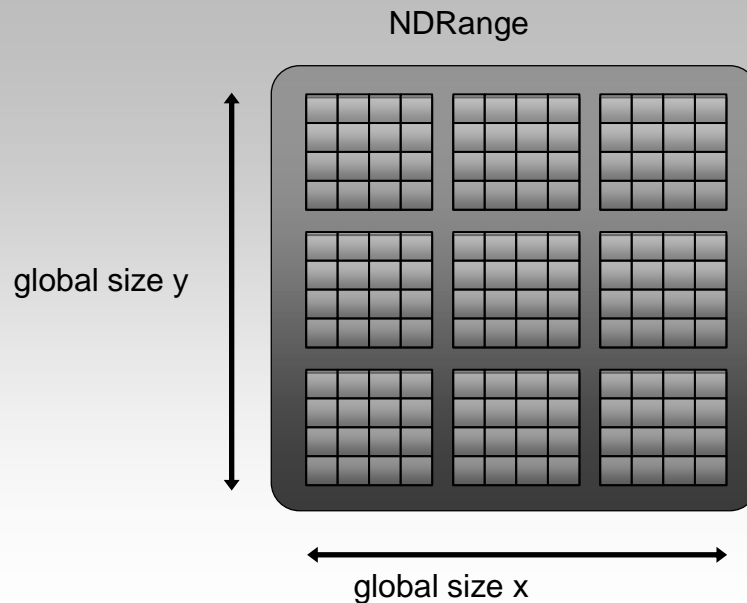
- We've:
 - looked at the basic principles of the GPU architecture in the processor design space
 - seen some of the tradeoffs that lead to GPU features

COMPUTE PROGRAMMING CONCEPTS

Common Compute Concepts

- Work Distribution
- Memory Model
- Thread Identity
- Synchronization for Data Consistency
 - Same Dispatch
 - Cross-Dispatch
 - Cross-API

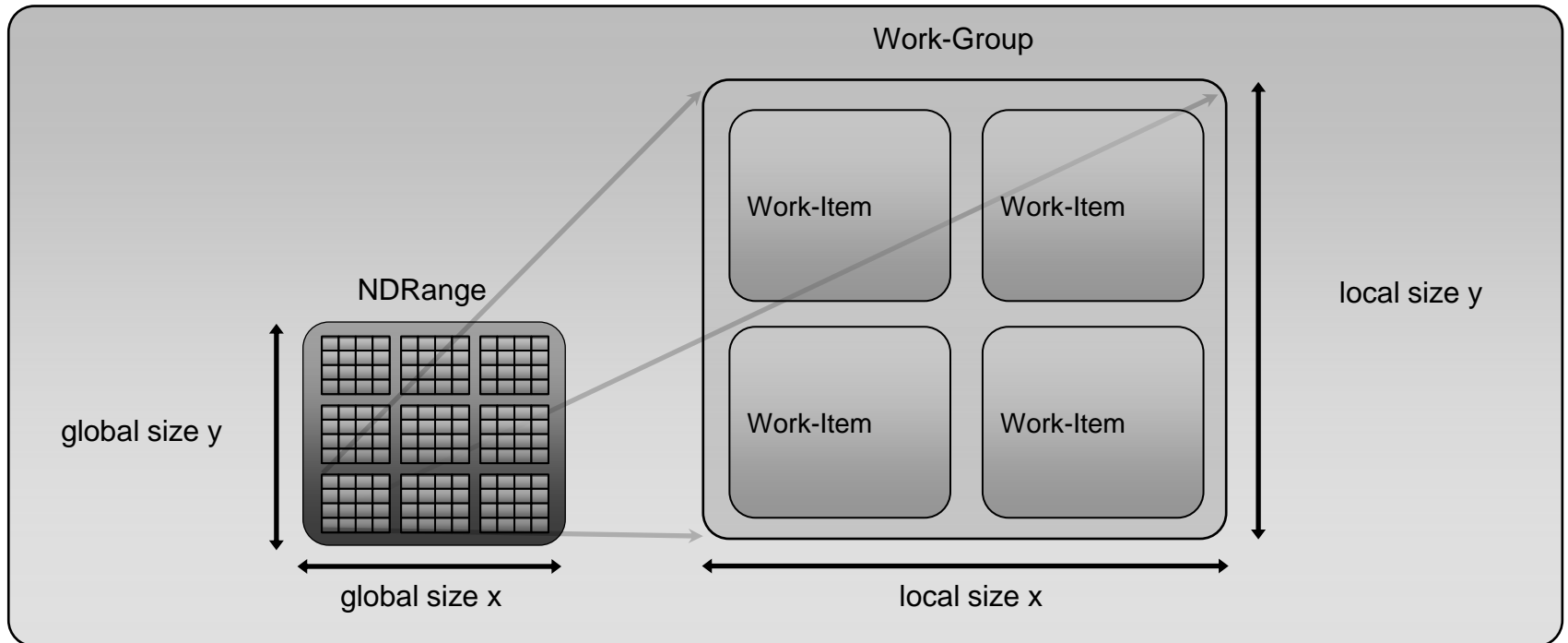
Work Distribution



Host application invokes a kernel over an index space

Index space is an N-dimensional range (where N is 1, 2, or 3)

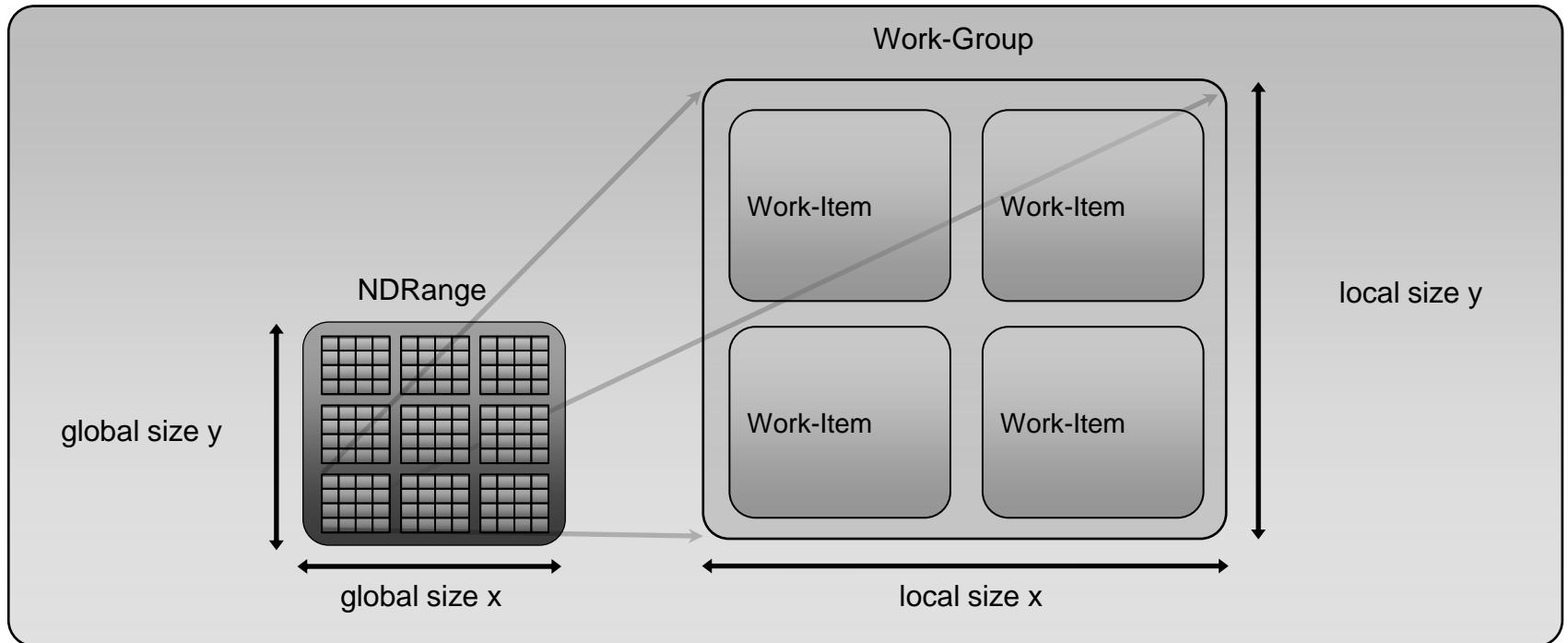
Work Distribution



Global range is executed over local work-groups

Each work-group has a collection of work-items addressable via a globally unique id

Work Distribution



Work-items share resources in a work-group

Mitigates communication costs for synchronisation -- group-wise barriers are cheap!

Work Distribution

Kernels are executed across a domain of work-items

Global dimensions define range of computation

Work-items are logically organised into work-groupsLocal

dimensions define size of work-group

Work-groups are executed in parallelWork-items in a work-group can

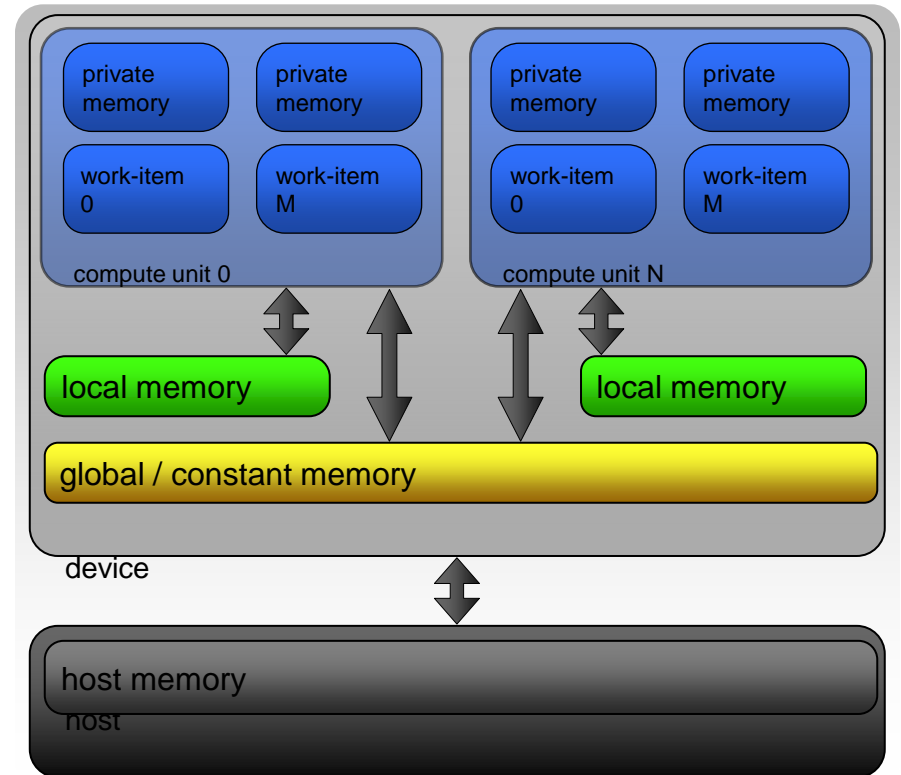
communicate to each other

Must use synchronisation to coordinate memory access

Memory Model

Address space hierarchy

All address spaces are distinct and cannot be intermixed



Memory Model

Private:

private to a single work-item

Local:

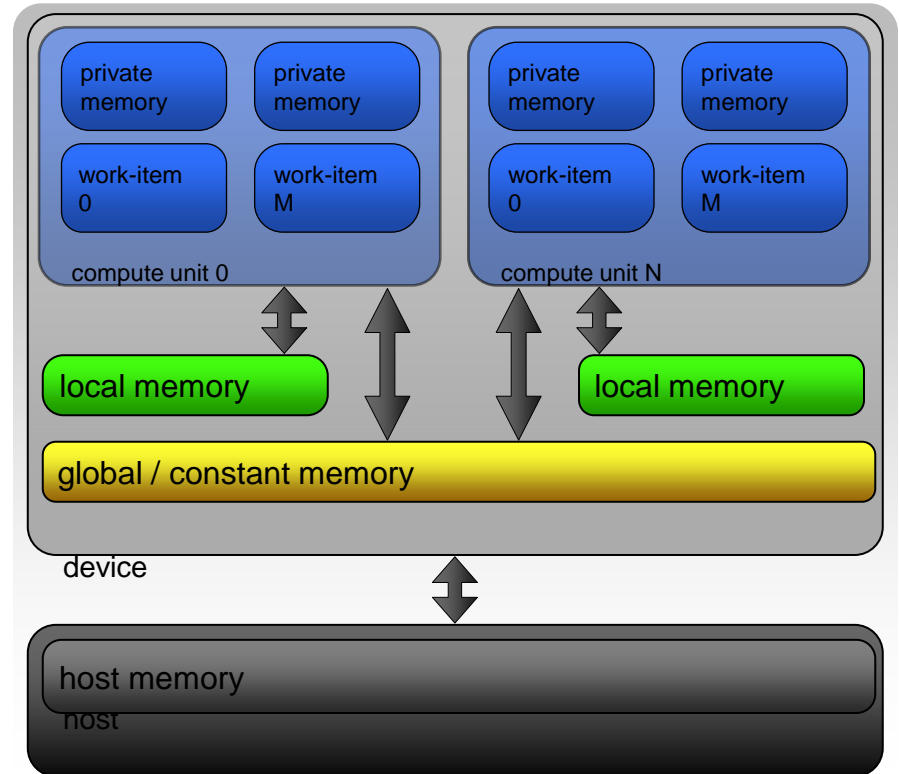
shared within a work-group

Global/Constant:

globally accessible by any work-item

Host:

accessible from host application

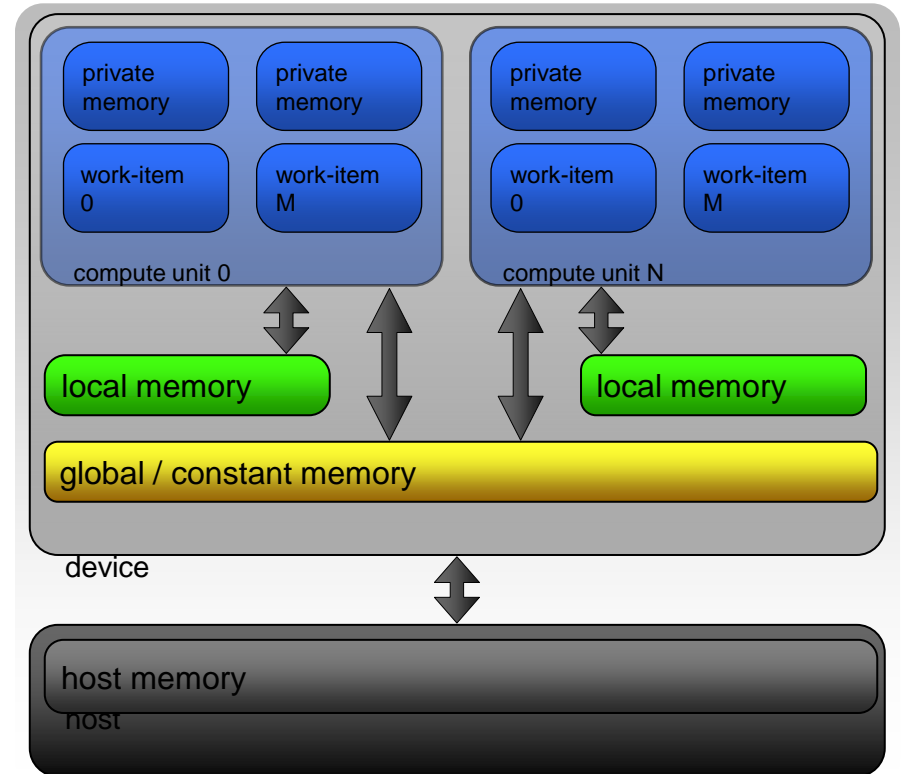


Memory Model

Data Transfer


All data movement between address spaces must be done explicitly

Applications must move data to/from
Host / **Global** / **Local** / **Private**



Thread Identity (Example)

```
__kernel void square(  
    __global float* input, __global float* output)  
{  
    size_t i = get_global_id(0);  
    output[i] = input[i] * input[i];  
}
```



Built-in methods provide access to index space addresses

Use the `get_global_id()` built-in for globally unique addresses

Use the `get_group_id()` for the logical group id spanning the ND-range

Use the `get_local_id()` for the local work-item address within a work-group

Data Consistency

Shared memory model uses relaxed consistency

State of memory visible to a work-item is **not guaranteed** to be consistent among all work-items at all times

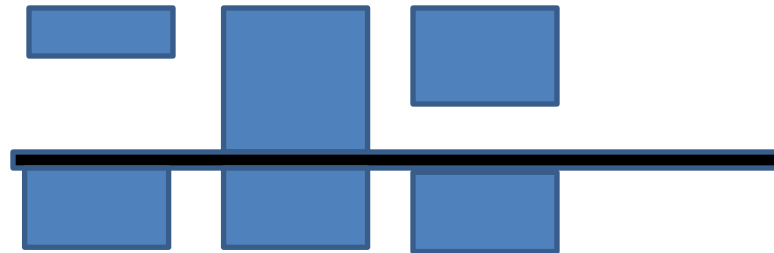
Data Consistency

If consistency is needed, synchronisation is required

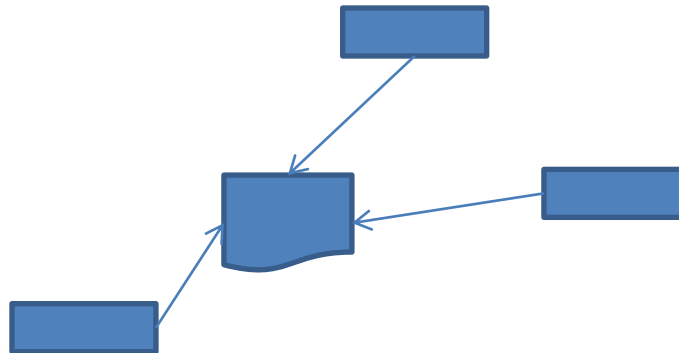
Synchronisation of memory must be done **explicitly** across all levels of the memory hierarchy in order to get the same data to be visible at any given time

Synchronization: Same Dispatch

- Barriers



- Atomics



Cross-Job Synchronization

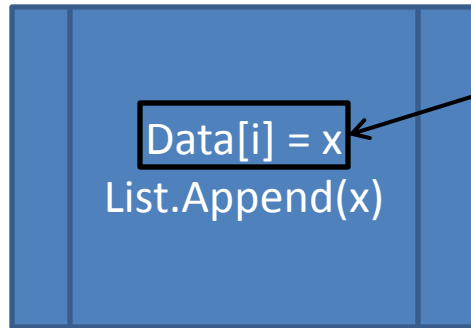
- Resource dependencies
 - Driver
 - Explicit flushes
 - Barriers
- Synchronization objects

COMPUTE APIS

Compute Choices

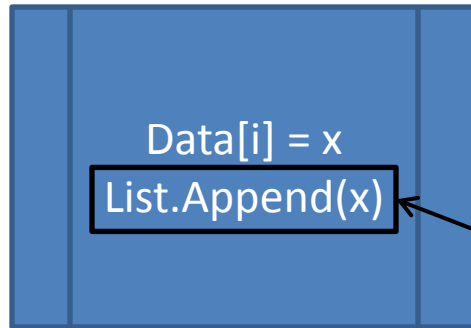
- Augmented graphics shaders
- DirectCompute
- OpenGL Compute
- OpenCL
- CUDA
- ipsc
- C++ Amp

Augmented Graphics



Scattered write
(UAV, image_load_store)

Augmented Graphics



Synchronized write
(Append buffer, atomic)

Augmented Graphics Shaders

- Scattered writes
 - DirectX Unordered Access View (UAV)
 - OpenGL image load store
- Global atomics
- No shared local memory

DirectCompute/OpenGL Compute

- Adds local memory concept
- Easy if you're already using graphics
- Uses graphics shader language

DirectCompute

```
pCtx->CSSetShader(pCS, 0, 0);  
pCtx->CSSetConstantBuffers( 0, N, ppConstants);  
pCtx->CSSetSamplers( 0, N, ppSamplers);  
pCtx->CSSetShaderResources( 0, N, ppResources );  
pCtx->CSSetUnorderedAccessViews( 0, N, ppUAVs, pCounts);  
pCtx->Dispatch( nGroupsX, nGroupsY, nGroupsZ );  
... // No sync required - will flush when used as input  
pCtx->DrawIndexed(...);
```

- Simple Synchronization at API level
- UAV ~ render target, so blocks on use as input in later draw calls

Direct Compute

```
RWStructuredBuffer<int> linearOutput;  
groupshared int var;  
  
[numthreads(64, 1, 1)]  
void ContrivedSample(  
    uint3 globalIdx : SV_DispatchThreadID,  
    uint3 localIdx : SV_GroupThreadID,  
    uint3 groupIdx : SV_GroupID )  
{  
    if(localIdx.x == 0)  
        var = 1;  
  
    GroupMemoryBarrier();  
  
    linearOutput[globalIdx.x] = var;  
}
```

OpenGL Compute

```
glBindImageTexture(0, texture, 0, GL_FALSE,  
GL_WRITE_ONLY, GL_R32I);
```

```
//... and so forth
```

```
glUseProgram(program);  
glDispatchCompute(nGroupsX, nGroupsY, nGroupsZ);  
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);  
...  
glDrawArrays(...)
```

OpenGL Compute

```
layout(r32i) image1D linearOutput;
shared int var;

layout(local_size_x = 64, local_size_y = 1, local_size_z = 1)
void ContrivedSample()
{
    const uvec3 localIdx = gl_LocalInvocationID;
    const uvec3 globalIdx = gl_GlobalInvocationID;
    const uvec3 groupIdx = gl_WorkGroupID;
    if(localIdx.x == 0)
        var = 1;

    barrier();

    imageStore(linearOutput, globalIdx.x, var);
}
```

OpenCL

- Owned by Khronos (like OpenGL)
- Cross-Platform
- CPU too
- Bit more sophisticated than Graphics Compute

OpenCL Dispatch

```
cl_int clEnqueueMethod(cl_command_queue, /* command queue */,  
    ... /* method specific parameters */,  
    cl_uint * /* number of events in wait list */,  
    const cl_event * /* event wait list */,  
    cl_event * /* returned event */)
```

All enqueue methods return an error code. Method returns CL_SUCCESS if command was enqueued successfully. All enqueue methods support an event wait-list.

Command will not get executed until all events in list are complete. All enqueue methods return an event for the command. Returned event identifies the specific command that was enqueued.

Basic OpenCL Kernel Example

```
__kernel void square(  
    __global float* input, __global float* output)  
{  
    size_t i = get_global_id(0);  
    output[i] = input[i] * input[i];  
}
```

OpenCL /GL Interop

```
// Gets GL devices that support contexts
// Showing Windows/Linux version (Apple also works)
clGetGLContextInfoKHR( ... )
    <wgl contexts>,
    CL_DEVICES_FOR_GL_CONTEXT_KHR,
    N*sizeof(cl_device_id),
    cdDeviceID, &size);

cxGPUContext = clCreateContext(
    <wgl contexts, devices, cl platform>,
    1, cdDeviceID, NULL, NULL, &ciErrNum);
cl_event e = clCreateEventFromGLsyncKHR(
```

OpenCL /GL Interop

```
clGetGLContextInfoKHR(  )
```

```
clCreateContext(<properties include GL context> )
```

```
buffer = clCreateFromGLBuffer(...)
```

```
image = clCreateFromGLTexture(...)
```

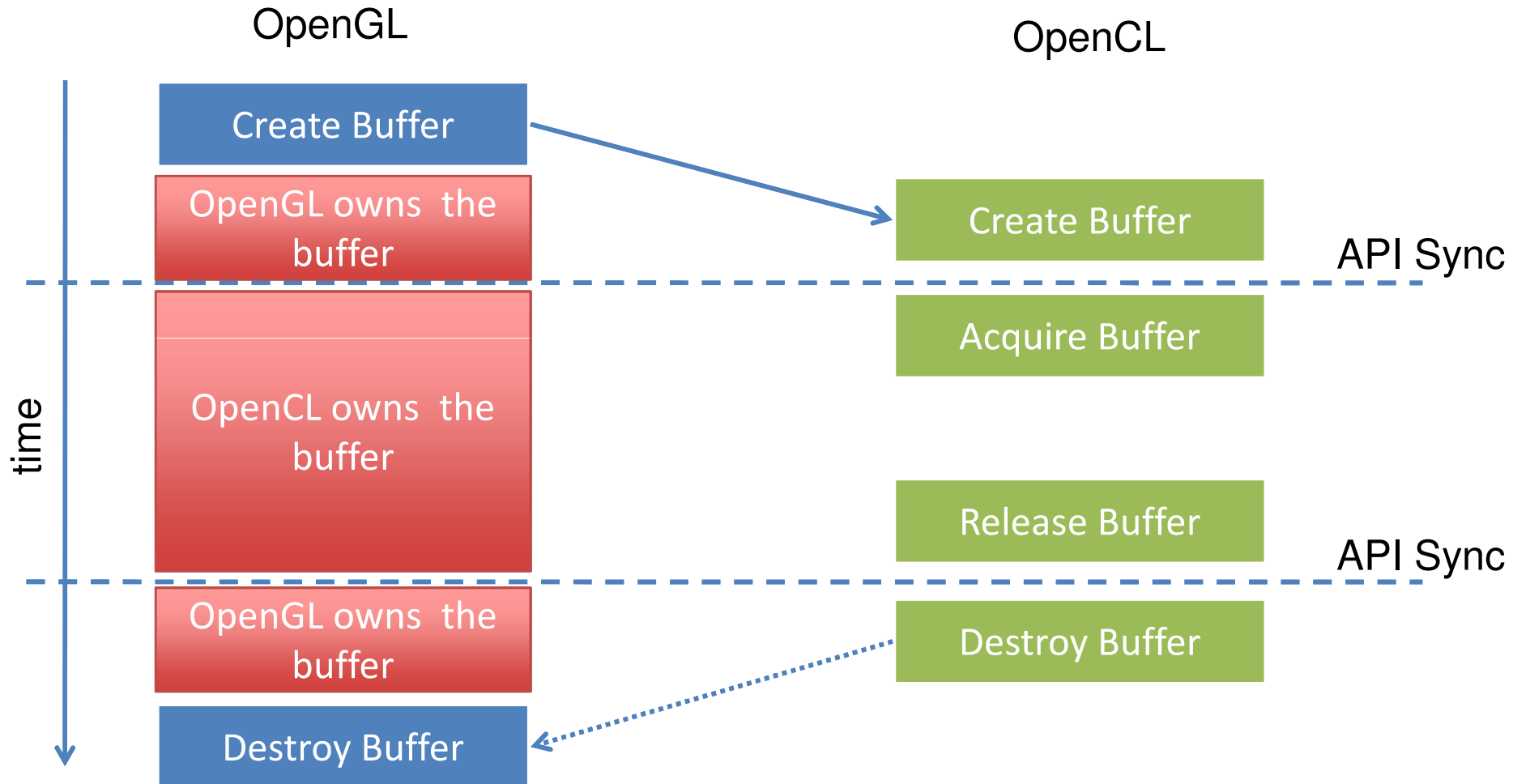
```
image = clCreateFromGLRenderbuffer(...)
```

```
clEnqueueAcquireGLObjects(..., &event)
```

```
clEnqueueReleaseGLObjects(...)
```

```
event = clCreateEventFromGLsyncKHR( ... )
```

OpenCL /GL Interop



OpenCL/DX Interop

```
// Showing D3D11, also D3D9 and D3D10
clGetDeviceIDsFromD3D11KHR(  )
clCreateContext(<properties include DX device> )

buffer = clCreateFromD3D11Buffer(...)
image = clCreateFromD3D11Texture2D(...)
image = clCreateFromD3D11Texture3D(...)

clEnqueueAcquireD3D11Objects(..., &event)
clEnqueueReleaseD3D11Objects(...)

clCreateEventFromGLsyncKHR( ... )
```

CUDA

- CPU invokation

square<<<64,1>>>(input, output);

- Kernel

```
__global__ void square(float* input, float* output)
{
    int i = blockIdx.x;
    output[i] = input[i] * input[i];
}
```

CUDA Interop

```
//GL
```

```
cudaGLSetGLDevice()
```

```
cudaGLRegisterBuffer()
```

```
cudaGLRegisterImage() // texture/renderbuffer
```

```
//DX
```

```
cudaD3D11GetDevice(&dev, adapter);
```

```
D3D11CreateDeviceAndSwapChain(adapter, ..., &d3dDevice, ...)
```

```
cudaD3D11SetDirect3DDevice(d3dDevice)
```

```
cudaGraphicsD3D11RegisterResource()
```


Use Cases

- Overview
- In Games
- Filtering
 - PS vs Compute
- Culling (lights, geometry)
- Back-end physics

Candidates – CPU Tasks

- Highly data parallel
- Output already going to GPU
 - Culling
 - “Cosmetic” physics
- Don’t add roundtrips

Candidates – From other shaders

- Easier programming model
- Local memory usage
- Watch out for switching cost

In Games (That I Know of)

- Post Effects and Filtering
 - Probably most common
- Light Culling
 - Tile Deferred: Frostbite and plenty others
 - Forward+: Dirt Showdown (Forward+)
- Variable Bit-Rate Texture Decompression (EGSR 2011)
 - Civilization V

Filtering

- Convert from PS
- Saves redundant compute
- Following Slides from talks by Jon Story and Bill Billodeau at GDC 2011 (Thanks!)
 - Seperable Filtering
 - Application: SSAO
 - Application: DOF

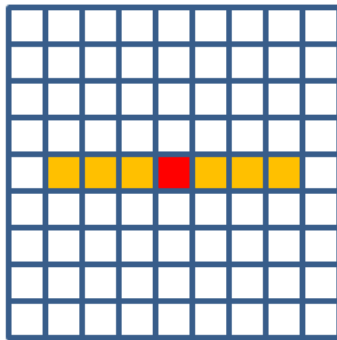
Separable Filters

- Two 1D sampling instead of full 2D
- Classically performed by the Pixel Shader
- Source image over-sampling increases with kernel size
 - Shader is usually TEX instruction limited
- In practice also “good enough” for non-separable
- Often used for bilateral cases

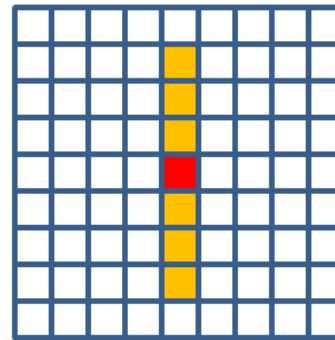
Typical Pipeline Steps



Horizontal Pass



Vertical Pass

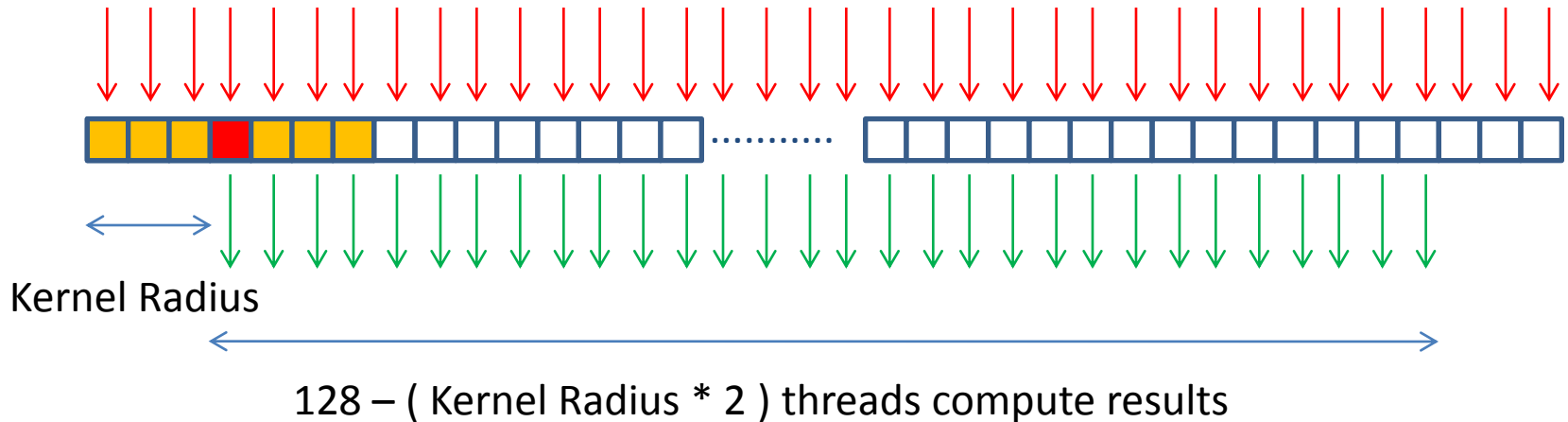


$$\text{Reads} = 2 * \text{width} * \text{pixels}$$

Basic CS Version

- Use the TGSM as a cache to reduce TEX and ALU ops

- ~~Make sure thread group size is a multiple of 64~~



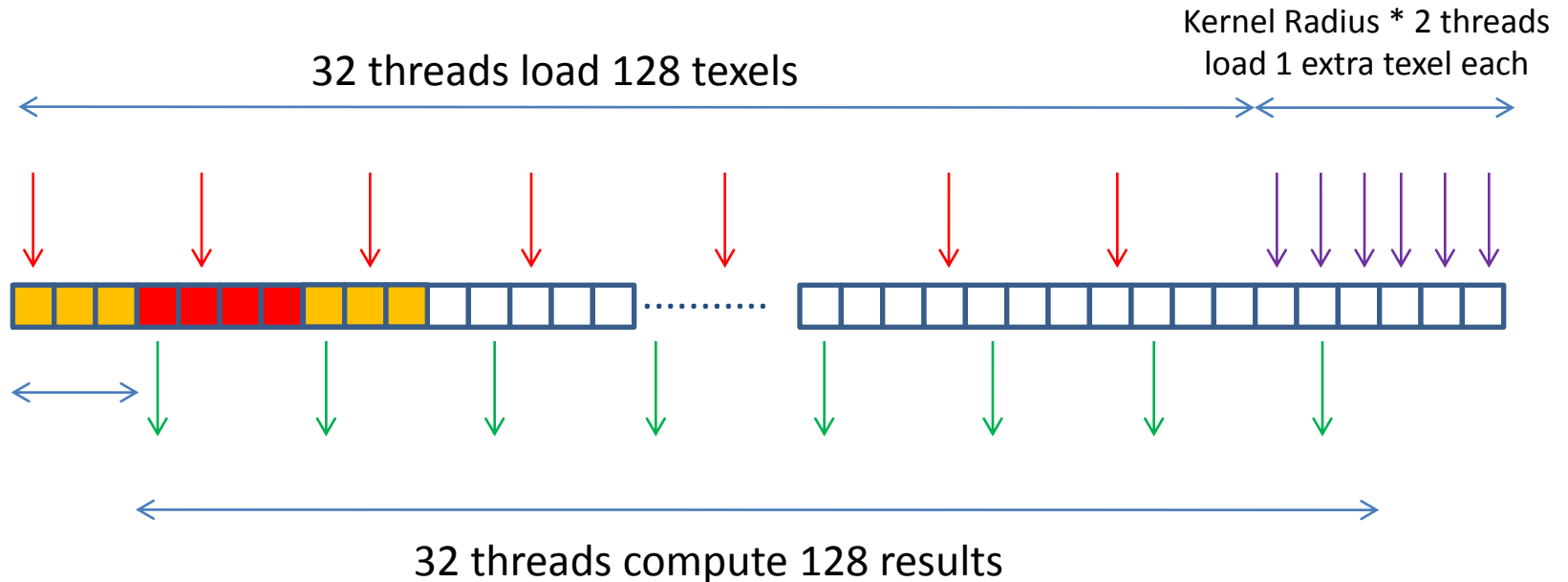
Redundant compute threads ☹️

Avoid Redundant Threads

- Should ensure that all threads in a group have useful work to do – wherever possible
- Redundant threads will not be reassigned work from another group
- This would involve a lot of redundancy for a large kernel diameter

Multiple Pixels per Thread

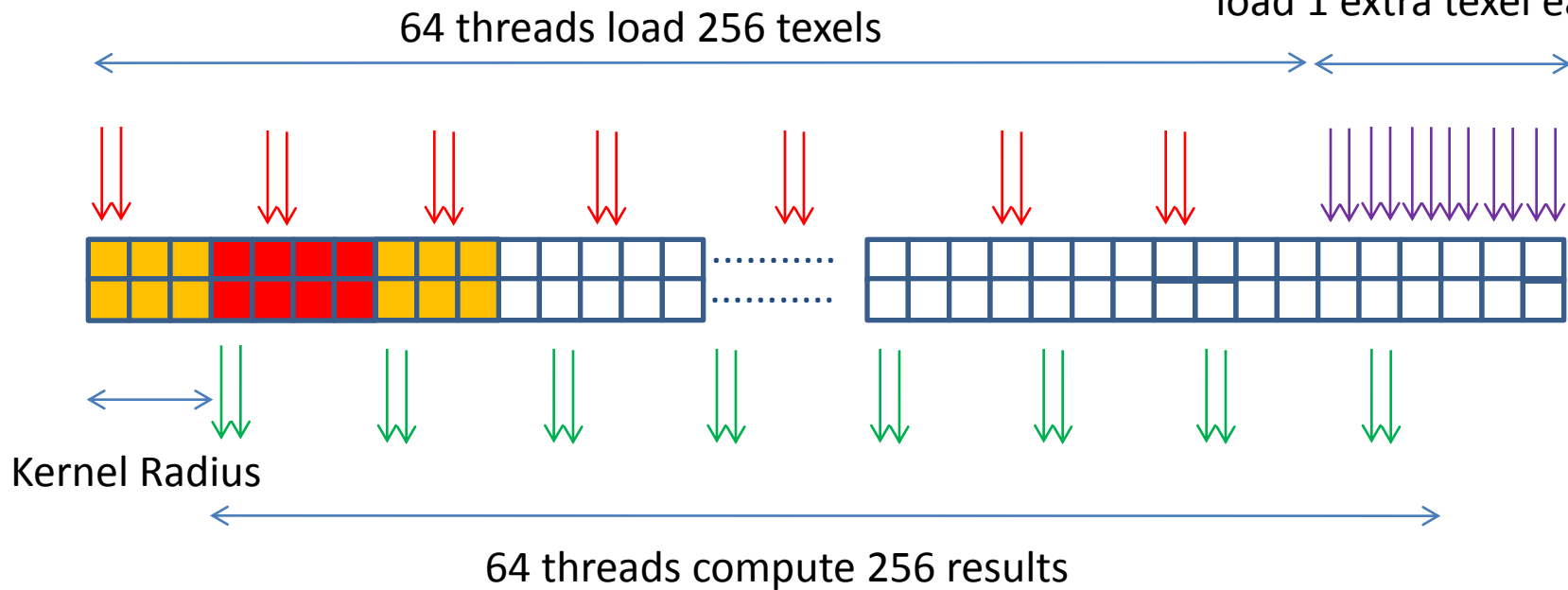
- Possible to cache TGSM reads on General Purpose Registers (GPRs)



Compute threads not a multiple of 64 ☹️

2D Thread Groups

- Process multiple lines per thread group
 - Thread group size is back to a multiple of 64
 - Better than one long line (2 or 4 works well)
 - Improved texture cache efficiency
- Kernel Radius * 4 threads
load 1 extra texel each

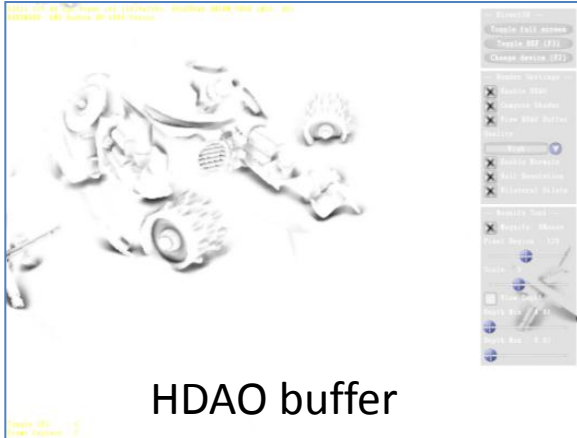


Kernel Diameter

- Kernel diameter needs to be $> N$ to see a DirectCompute win
 - Otherwise the overhead cancels out the advantage
- The larger the kernel diameter the greater the win
- Large kernels also require more TGSM

Bilateral SSAO

Depth + Normals



*



=



Perform at Half Resolution

- HDAO at full resolution is expensive
- Running at half resolution captures more occlusion – and is obviously much faster
- Problem: Artifacts are introduced when combined with the full resolution scene

Bilateral Dilate & Blur



HDAO buffer doesn't match with scene

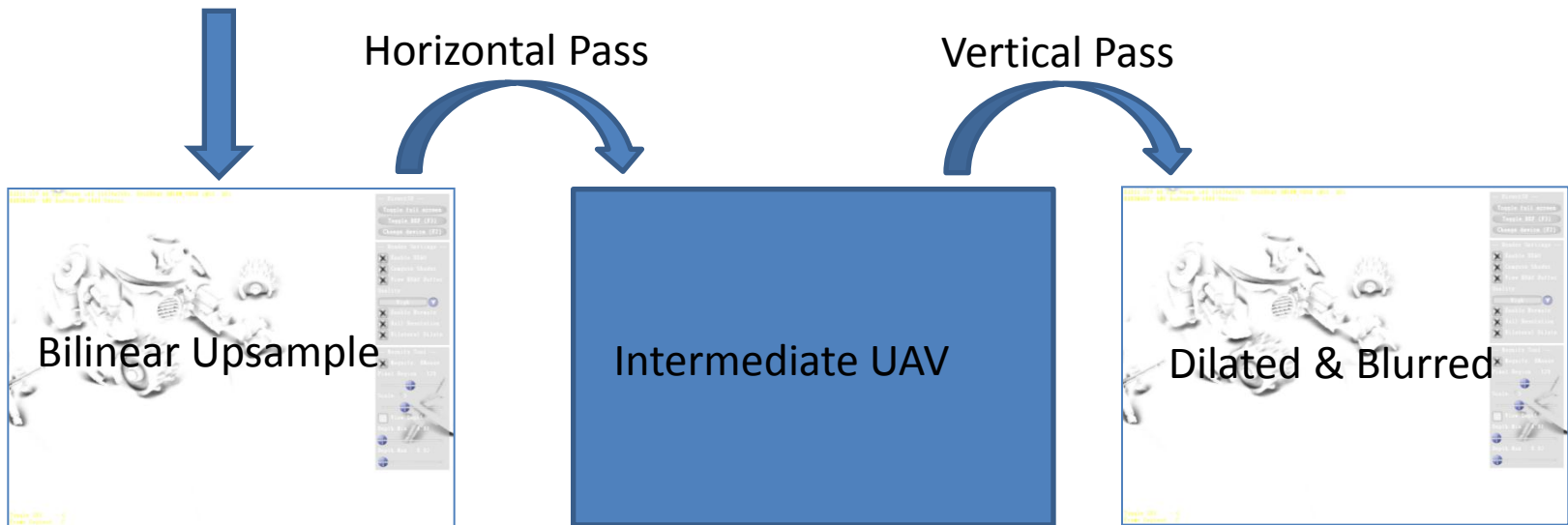


A bilateral dilate & blur
fixes the issue

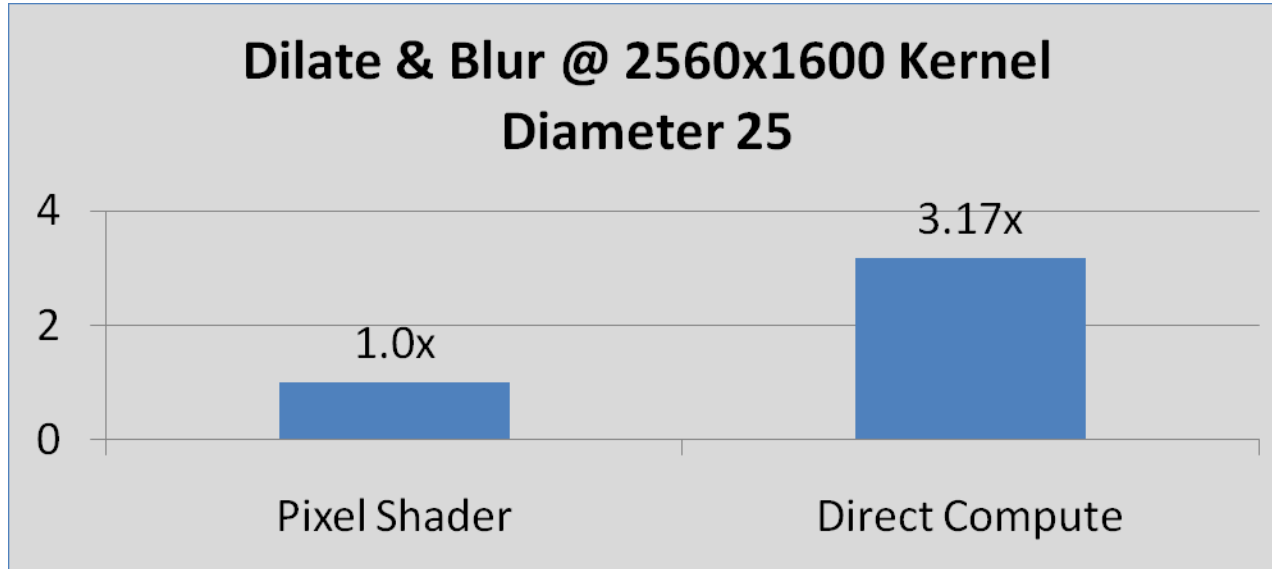
New Pipeline...

$\frac{1}{2}$ Res

Still much faster than performing at full res!



Pixel Shader vs DirectCompute



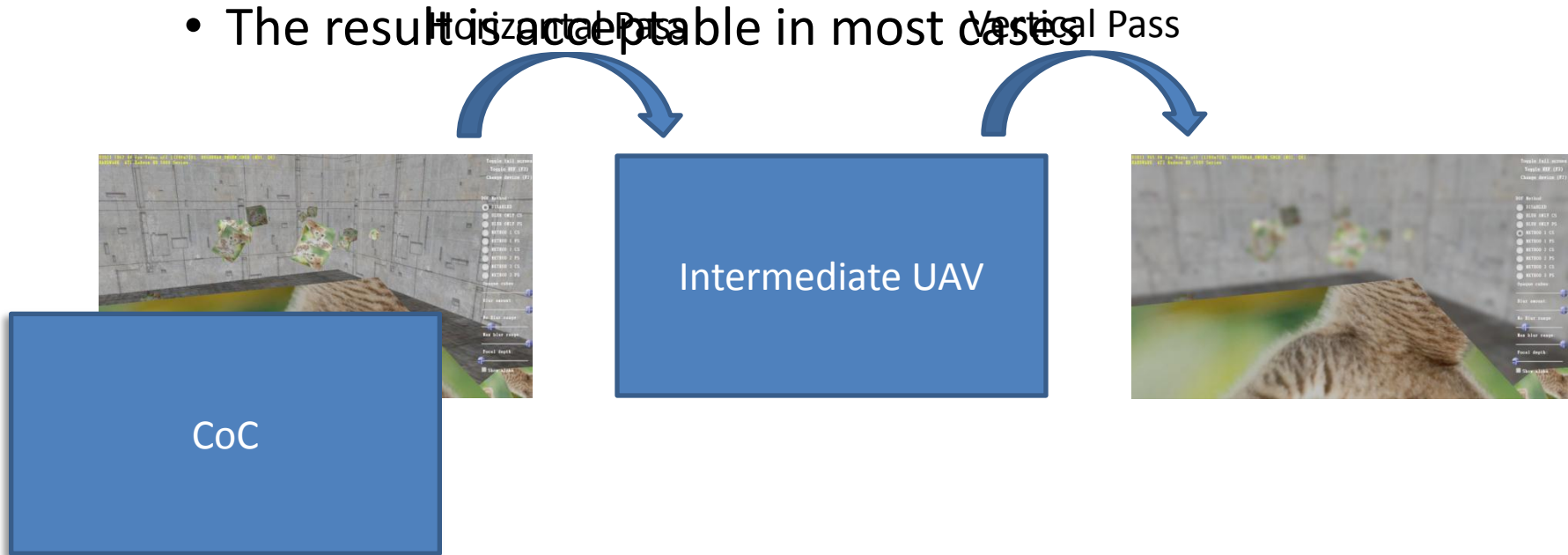
*Tested on a range of 2011 AMD and NVIDIA DX11 HW, DirectCompute is between ~2.53x to ~3.17x faster than the Pixel Shader

Depth of Field

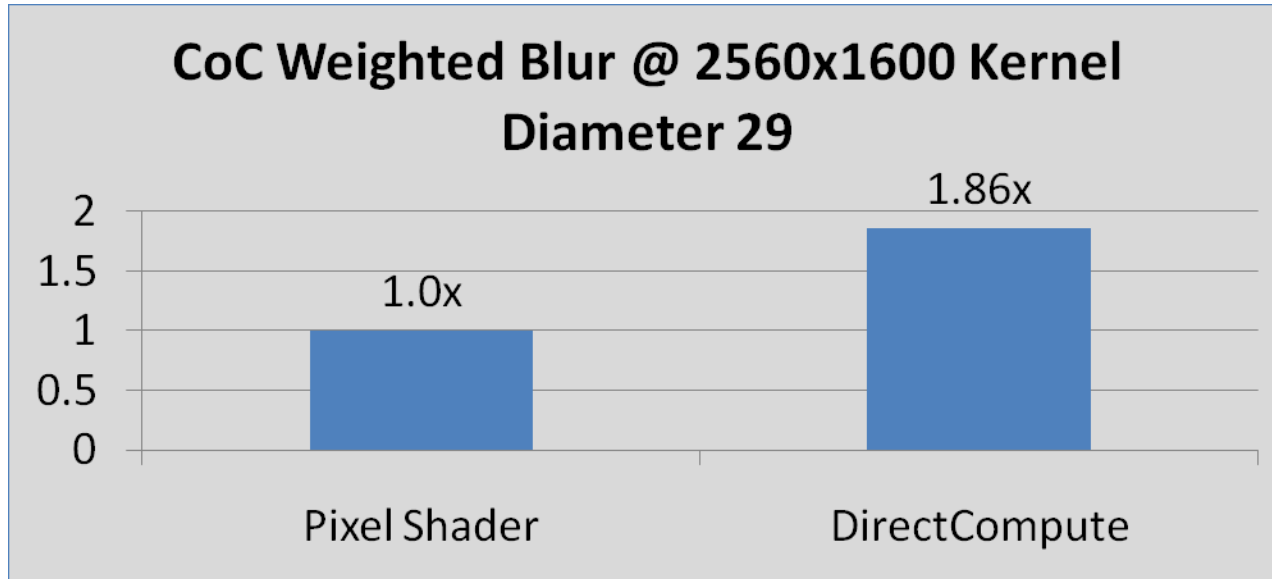
- Many techniques exist to solve this problem
- A common technique is to figure out how blurry a pixel should be
 - Often called the Circle of Confusion (CoC)
- A Gaussian blur weighted by CoC is a pretty efficient way to implement this effect

Treat as Seperable

- Combined Gaussian Blur and CoC weighting isn't a separable filter, but we can still use a separate horizontal and vertical 1D pass
 - The result is acceptable in most cases



Pixel Shader vs DirectCompute



*Tested on a range of 2011 AMD and NVIDIA DX11 HW, DirectCompute is between ~1.48x to ~1.86x faster than the Pixel Shader

Culling

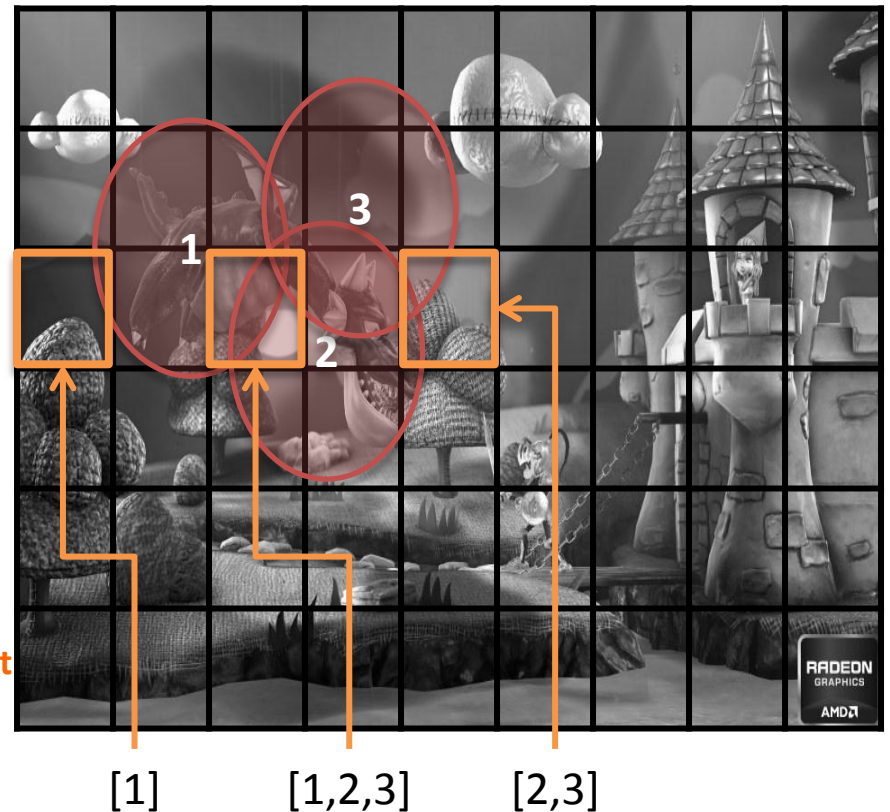
- Lights
 - Tile-Based Deferred
 - **Forward+**
- Geometry Culling

Forward+ Use



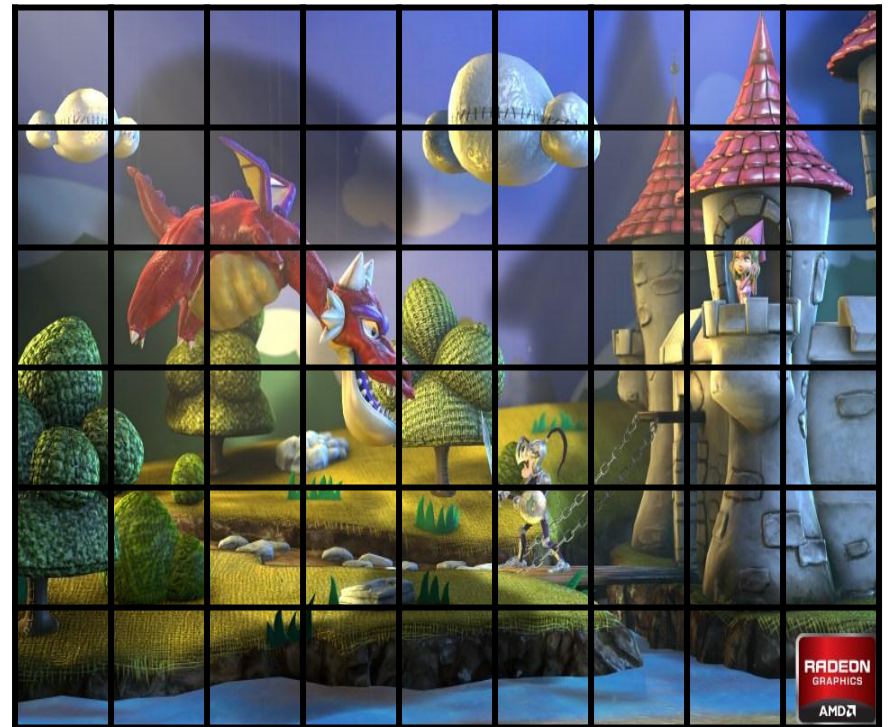
Forward+

- Depth prepass
 - Fills z buffer
 - Prevent overdraw for shading
 - Used to calculate position for light culling
- Light culling
 - Culls light per tile basis
 - Input: z buffer, light buffer
 - Output: light list per tile
- Shading
 - Geometry is rendered
 - Pixel shader
 - Iterate through light list **calculated in light**
 - Evaluates materials for the lights

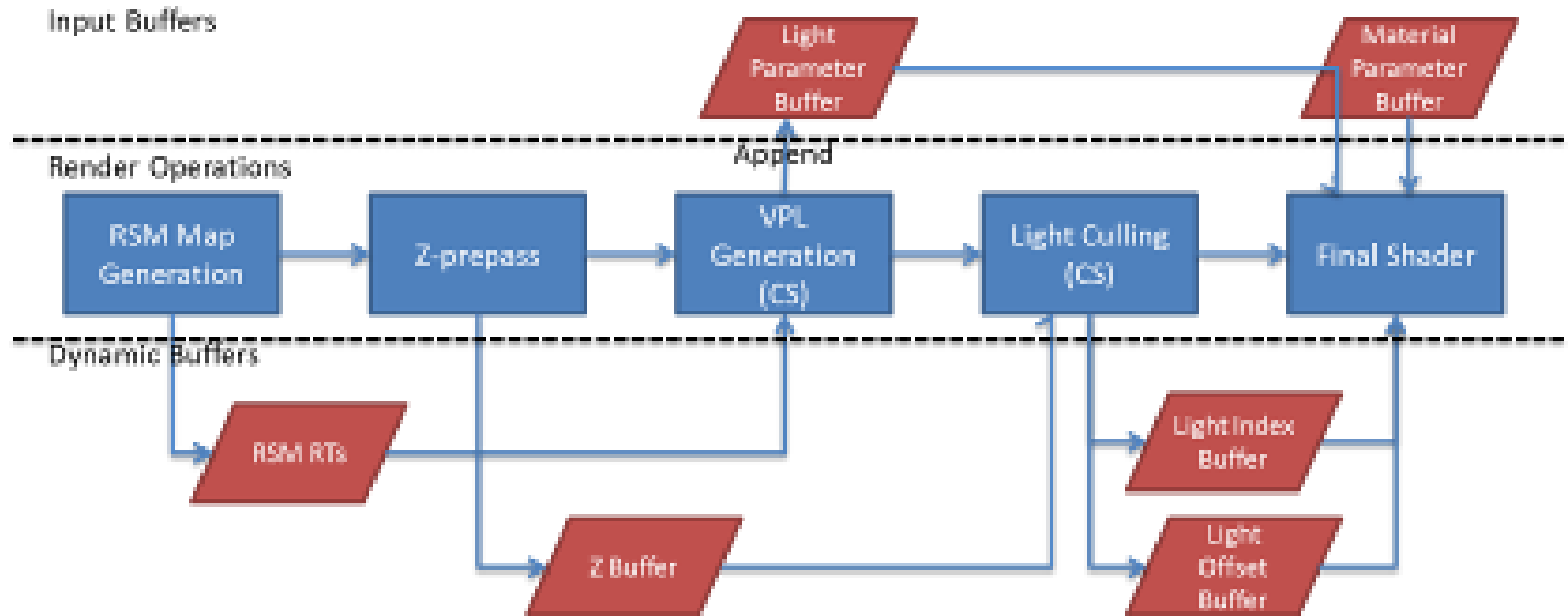


Light Culling Detail

- Single compute shader
- A thread group is executed per tile
- Calculate Z extent
- Build frustum
- 64 lights are culled in parallel
- Overlapped light indices are accumulated in TLS
- Export
 - One atomic add
 - Write light indices to a contiguous memory (\Leftrightarrow Linked list)



Leo Pipeline

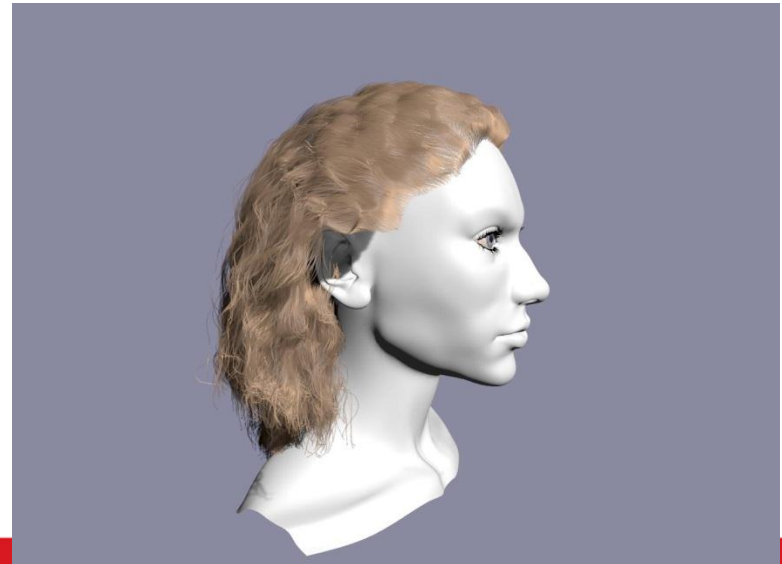
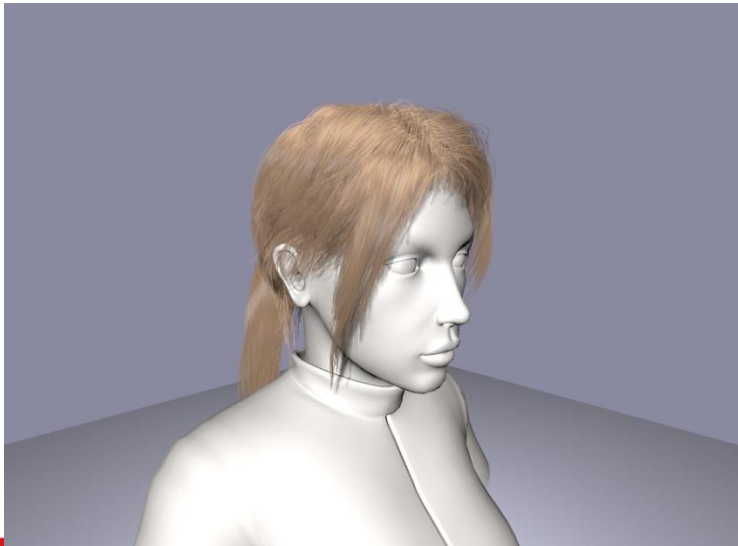


Back-end Physics

- **Hair**
- Cloth
- Deformation

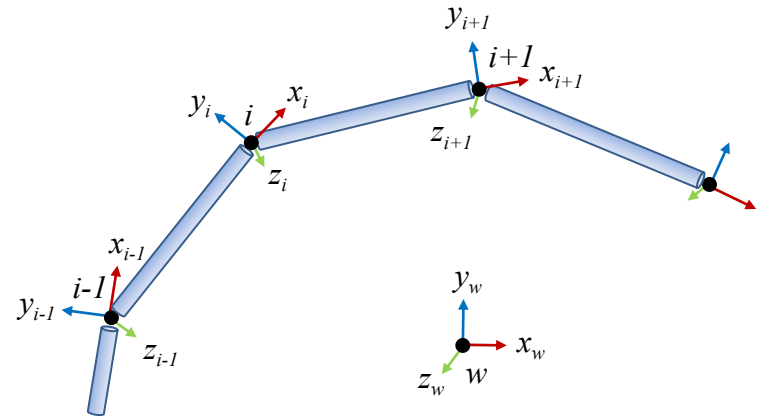
Real-time hair simulation

- Using DirectX compute shader, around 20K hair strands (1/5 of real human hair) can be simulated in less than 1 millisecond per frame.
- To preserve various hair styles, we developed global and local shape constraint methods which simulate bending and twisting effects.
- We exploits vertex-level parallel process to utilize GPU architecture.
- Various hair conditions such as wet can be simulated interactively.



Real-time hair simulation

- A strand is simulated as a polyline with local frames attached to each vertex.
- Global shape constraints are applied to each vertex and make them to return to the initial global position.
- Local shape constraints make vertices to return to initial local position w.r.t local frame.
- With both constraints, hair shapes can be maintained without tangling in weird shapes during fast moving gameplay.



Algorithm : Hair simulation outline

```

1  load hair data
2  precompute rest-state values
3  while simulation running do
4      compute forces such as gravity or wind
5      integrate
6      apply global shape constraints
7      while iteration do
8          | apply local shape constraints
9          apply edge length constraints
1         collision handling
0
    
```

Updated Slides

<http://developer.amd.com/Resources/documentation/presentations>

Thank-Yous

- Mike Houston, Ofer Rosenberg
- Justin Hensley, Derek Gertsman
- Jon Story, Bill Bilodeau
- Takahiro Harada, Jay McKee
- Dongsoo Han