



GPU TECHNOLOGY CONFERENCE

Direct Compute - Bring GPU Computing to the Mainstream

San Jose, CA | October 1, 2009
Tianyun Ni

Outline

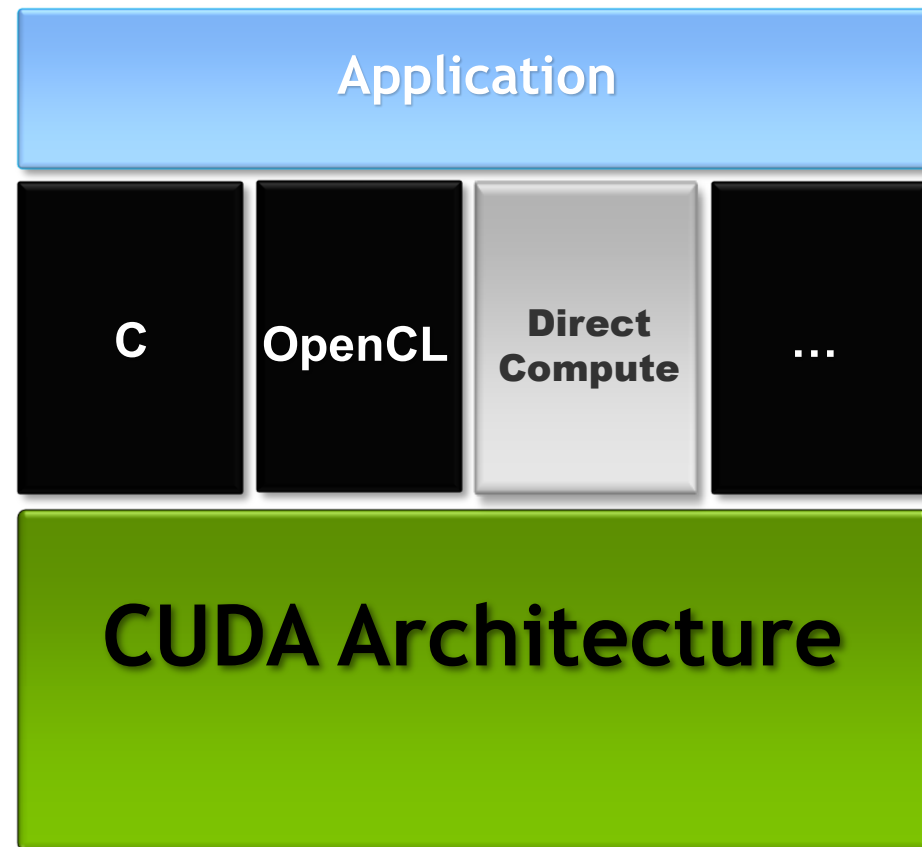
- Introduction
- Case study: Rolling Box Blur
- Applications
- Optimizations

Why Do We Need DirectCompute?

- Designed for general purpose computing
 - Enables more general data structures
 - Enables more general algorithms
- Enables better graphics
- Full-cross vendor support

What Is DirectCompute?

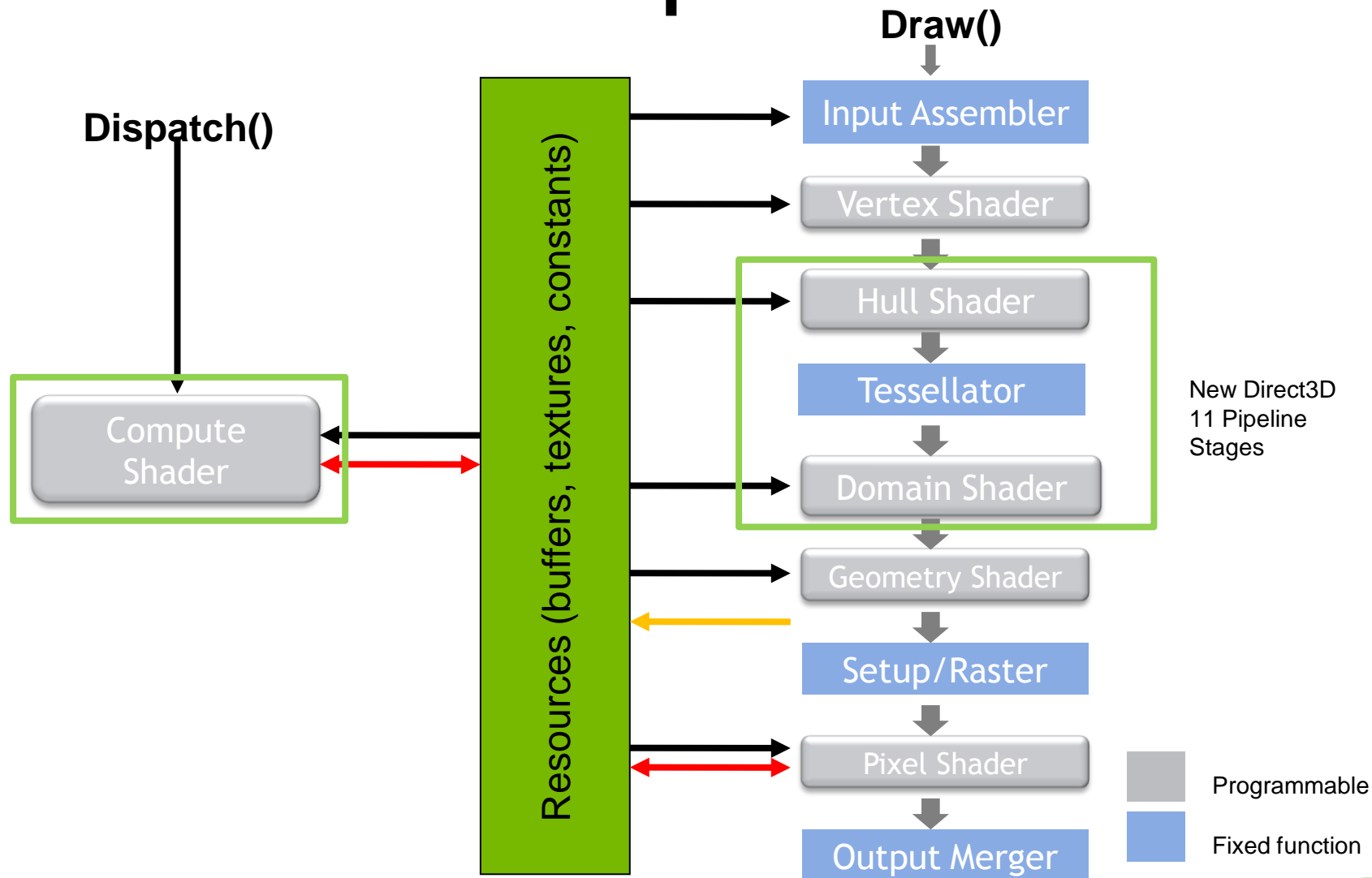
- Microsoft's standard GPU-Computing platform
 - For Windows Vista and Windows 7
 - On DX10 and DX11 hardware generations
- Another realization of the **CUDA architecture**
 - Sibling to OpenCL and NVIDIA's CUDA C extensions
 - Shares many concepts, idioms, algorithms and optimizations



What Is DirectCompute?

- DirectX Compute exposes the compute functionality of the GPU as a new type of shader - the compute shader
 - Not attached specifically to any stage of the graphics pipeline
 - Impose different policies and reduce the complexity of interactions
 - Full inter-operability with all D3D resources
 - A compute shader invocation through dispatching a specified regular grid of threads

What Is DirectCompute?

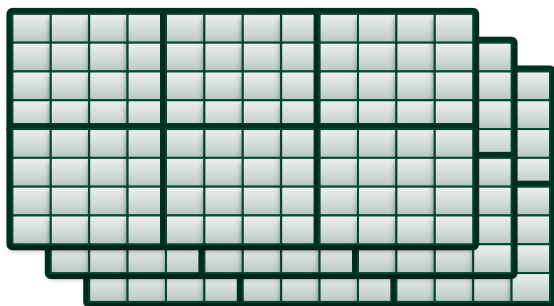


Features supported only in Compute Shader

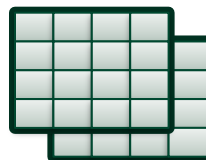
- Ability to decouple thread invocations from input or output domains
- Ability to share data between threads
- Random access writes
- Ability to synchronize a group of threads

Programming Model Overview

DirectCompute programs decompose parallel work into **groups** of **threads**, and **dispatch** many thread groups to solve a problem.



Dispatch: 3D grid of thread groups. Hundreds of thousands of threads.



Thread Group: 3D grid of threads. Tens or hundreds of threads.

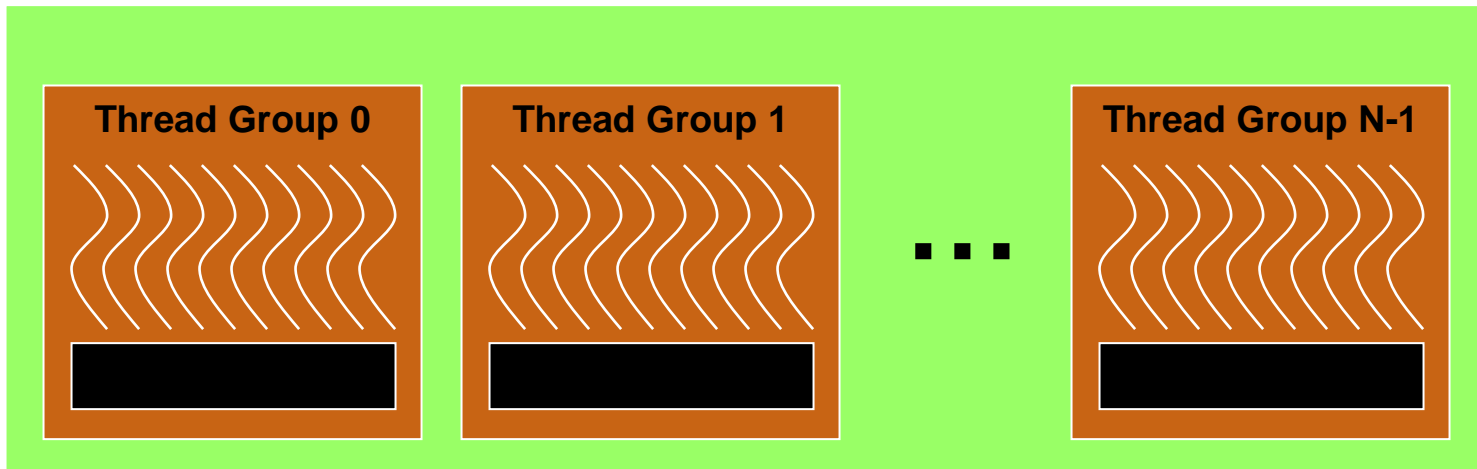
numThreads(nX, nY, nZ)



Thread: One invocation of a shader.

SV_ThreadID,
SV_ThreadGroupID,
SV_ThreadIDinGroup

Parallel Execution Model



- Threads in the same group run concurrently
- Threads in different groups **may** run concurrently

Parallel Execution Model

- Thread groups are not preemptively multitasked
 - One thread must never wait for a thread in a different group; this will deadlock
- Compute Shader Invocation
 - Dispatch()
 - DispatchIndirect(): take the number of thread groups directly from a buffer resource

Resources in DirectCompute

- Data are stored in resources (Buffers and Textures)
- A new set of resources: unordered access resources
- New operations on buffers : Append / Consume
- Structured buffer: a new type of buffer that contains elements of a structure type
 - contains elements of a structure type
 - used as an array of struct typed elements

Resource Views in DirectCompute

- Resources (except for constant buffers) are accessed via “views”
 - Unordered access resources are accessed through an unordered access view
- SRV and UAV accesses are automatically bounds checked
- New resource and view types allow scatter and gather in computer shaders

Shared Variables

- Variables declared as **groupshared** are accessible to all threads in a thread group
- Total groupshared variable size is limited:
 - CS4.x: 16KB per thread group
 - CS5.0: 32KB per thread group
- Shared variables are stored in on-chip (SM) memory
 - Cache to minimize off-chip bandwidth use
 - Efficient communication with a thread group

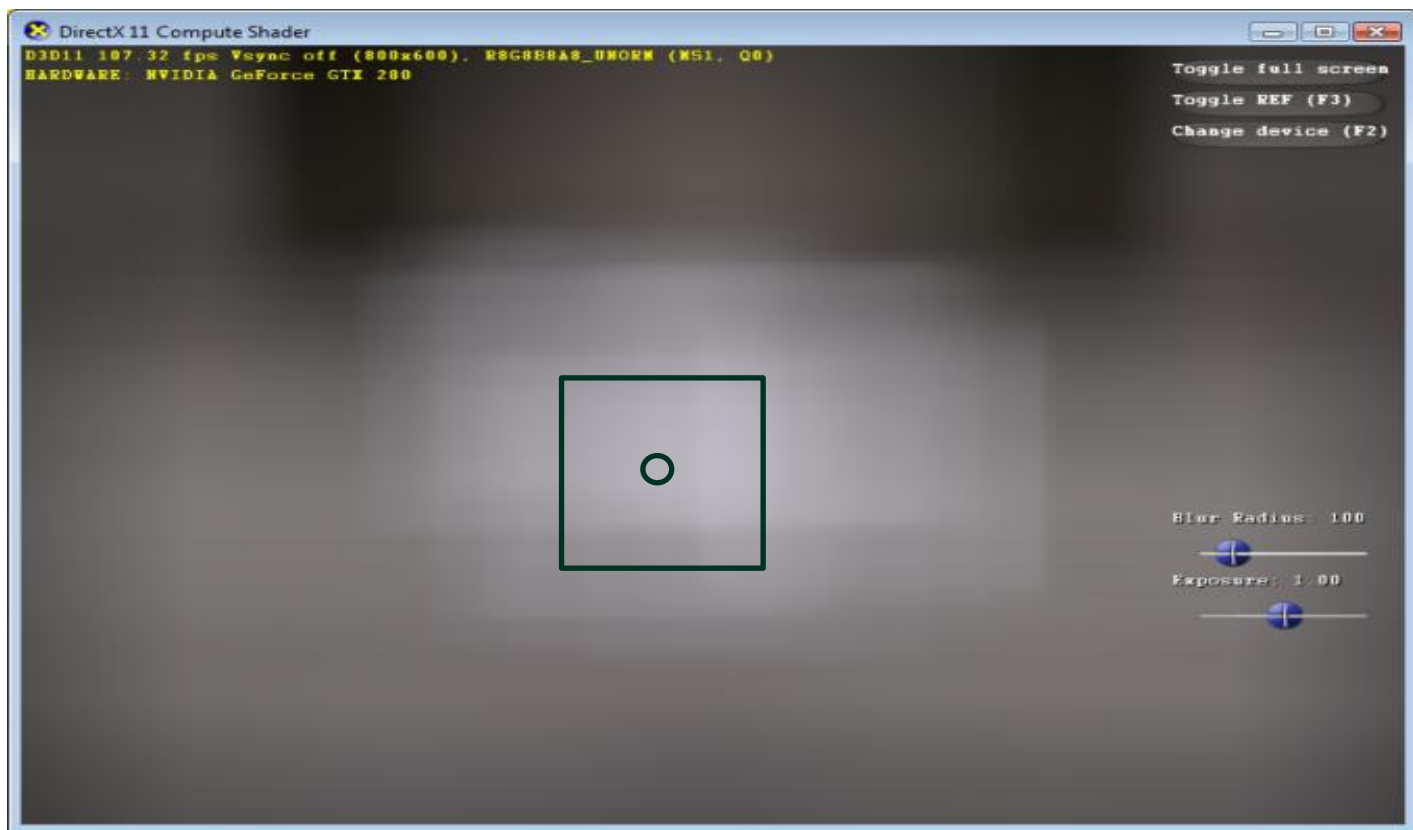
Thread Synchronization

- **GroupMemoryBarrierWithGroupSync()**
 - As each thread executes the barrier, it pauses
 - After all threads in the group have arrived at the barrier, they all unpause and continue running
 - Barriers are not allowed in divergent code: if some threads don't reach the barrier, the ones that do will wait forever
- **Atomic intrinsics**
 - InterlockedAdd
 - InterlockedMin
 - InterlockedMax
 - InterlockedOr
 - InterlockedXOr
 - InterlockedCompareWrite
 - InterlockedCompareExchange

Outline

- Introduction (Why, What, How)
- Case study: Rolling Box Blur
- Applications
- Optimizations

What is Box Blur?



What is Box Blur?

```
void BoxBlurAtOnePixel(int radius, const float4* X, float4* Y) {  
    for (int i = r-radius; i < r+radius; ++i)  
        for (int j = c-radius; j < c+radius; ++j)  
            Y[r][c] += X[i][j];  
  
    Y[r][c] /= (radius+1)^2;  
}
```

Reduce a 2D blur problem to 2 1D blurs

- Two blur directions: horizontal, and vertical

DirectCompute Implementation

- Set the render target to our own texture
- Render scene to texture
- Allocate memory and copy input data to the GPU
- Horizontal blur
 - `Dispatch(group_dimX, group_dimY, 1);`
 - Compute Shader
 - process box filtering in X
 - output the result to a RWStructuredBuffer
- Vertical Blur
- render the final image

Horizontal Blur

blur.cpp:

```
int dimX= int(ceil((float)pBackBufferDesc->Width / (128 - kernelhalf * 2)));
int dimY=pBackBufferDesc->Height;
Dispatch( dimX, dimY, 1 );
```

blurCS.hlsl:

```
Texture2D<float4>          InputTex : register( t0 );      }   Input to compute shader
cbuffer Params {...}

RWStructuredBuffer<float4>  Output;                          }   Output to compute shader

#define groupthreads 128
groupshared float4 temp[groupthreads ];
[numthreads(groupthreads ,1,1)]
void CSHorizontalBlur( uint3 Gid : SV_GroupID, uint GI : SV_GroupIndex )
{

}
```

Horizontal Blur

```
groupshared float4 temp[groupthreads];
[numthreads( groupthreads, 1, 1 )]
void CSHorizontalBlur( uint3 Gid : SV_GroupID, uint GI : SV_GroupIndex )
{
    int2 coord = int2( GI - kernelhalf + (groupthreads - kernelhalf * 2) * Gid.x, Gid.y );
    coord = clamp( coord, int2(0, 0), int2(g_inputsize.x-1, g_inputsize.y-1) );
    temp[GI] = InputTex.Load( int3(coord, 0) );
```

Load data in one thread
group to the shared memory

```
GroupMemoryBarrierWithGroupSync();
```

Prevent RAW data hazard

```
if ( GI >= kernelhalf &&
    GI < (groupthreads - kernelhalf) &&
    ( (Gid.x * (groupthreads - 2 * kernelhalf) + GI - kernelhalf) < g_outputsize.x) )
{
    float4 vOut = 0;
    [unroll]
    for ( int i = -kernelhalf; i <= kernelhalf; ++i )
        vOut += temp [GI + i];
    vOut /= (2*kernelhalf+1);
```

Horizontal blur

```
Output[GI - kernelhalf + (groupthreads - kernelhalf * 2) * Gid.x + Gid.y * g_outputsize.x]
    = float4(vOut.rgb, 1.0f);
```

Write the blurred
data to output buffer

Outline

- Introduction
- Case study: Rolling Box Blur
- Applications
- Optimizations

Applications

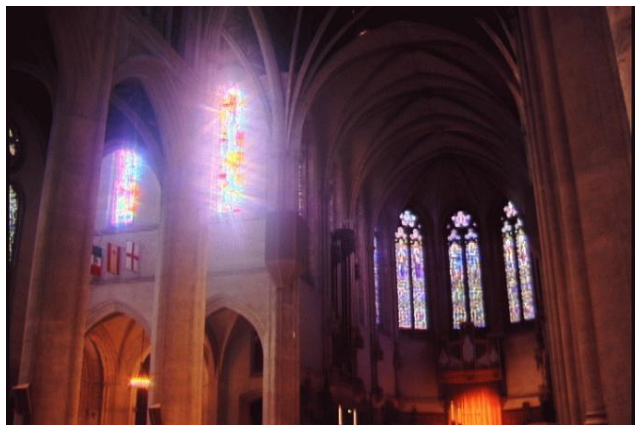
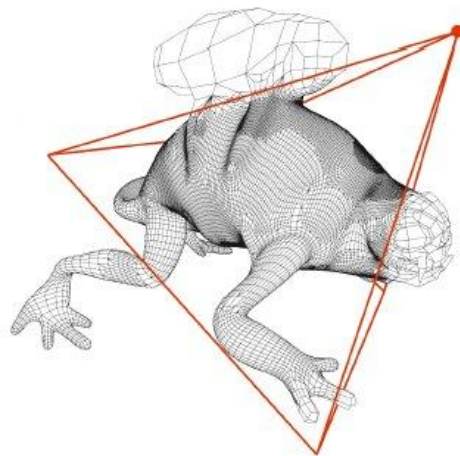
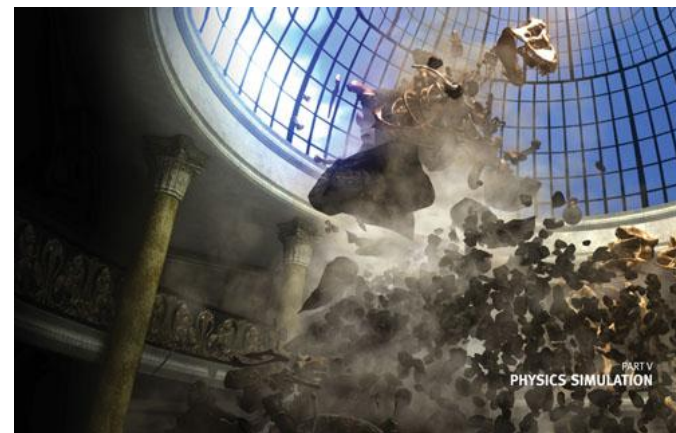


Image Processing



Procedural Geometry



Physics and Animation



Rendering



AI

...

Convolution-based Image Processing

- Depth of field, HDR bloom, motion blur, ...
- Implemented with various forms of image convolutions.



Halo 3 © Bungie Studios



Crysis © Crytek GmbH
© 2009 NVIDIA CORPORATION

Convolution-based Image Processing

- Convolution process
 - Each texture read is multiplied by the kernel weight and then summed into the final pixel value
 - All but one of the texels that are shared by its neighbor pixel
 - Shared memory in DirectCompute
- Convolution Soup: A Case Study in CUDA Optimization
 - October 2, 10:30-11:30, in California room

Fast Fourier Transform

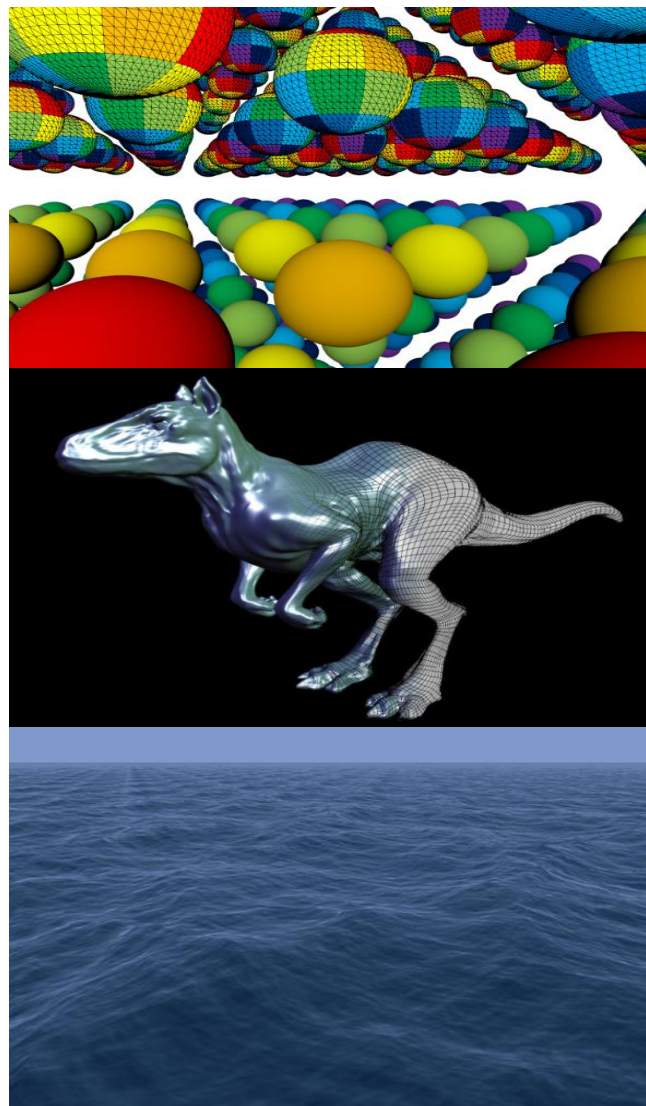
- FFTs are used in many image processing and video processing applications
- FFTs require a high number of texture reads per floating point calculation
- > 2x than the implementation without using shared register storage

Reduction

- Reduction operations are useful for tone mapping, histograms, prefix sums, and summed area tables.
- Shared storage can be used to limit the number of passes that must be performed by these algorithms.

Geometry Processing

- Mesh deformation
- Tessellation
- Recursive subdivision
- Water surface rendering



Subdivision Surfaces

- **Real-Time Reyes-Style Adaptive Surface Subdivision**

Anjul Patney and John D. Owens

http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=952

- **Real-Time View-Dependent Rendering of Parametric Surfaces**

Christian Eisenacher, Quirin Meyery, Charles Loop

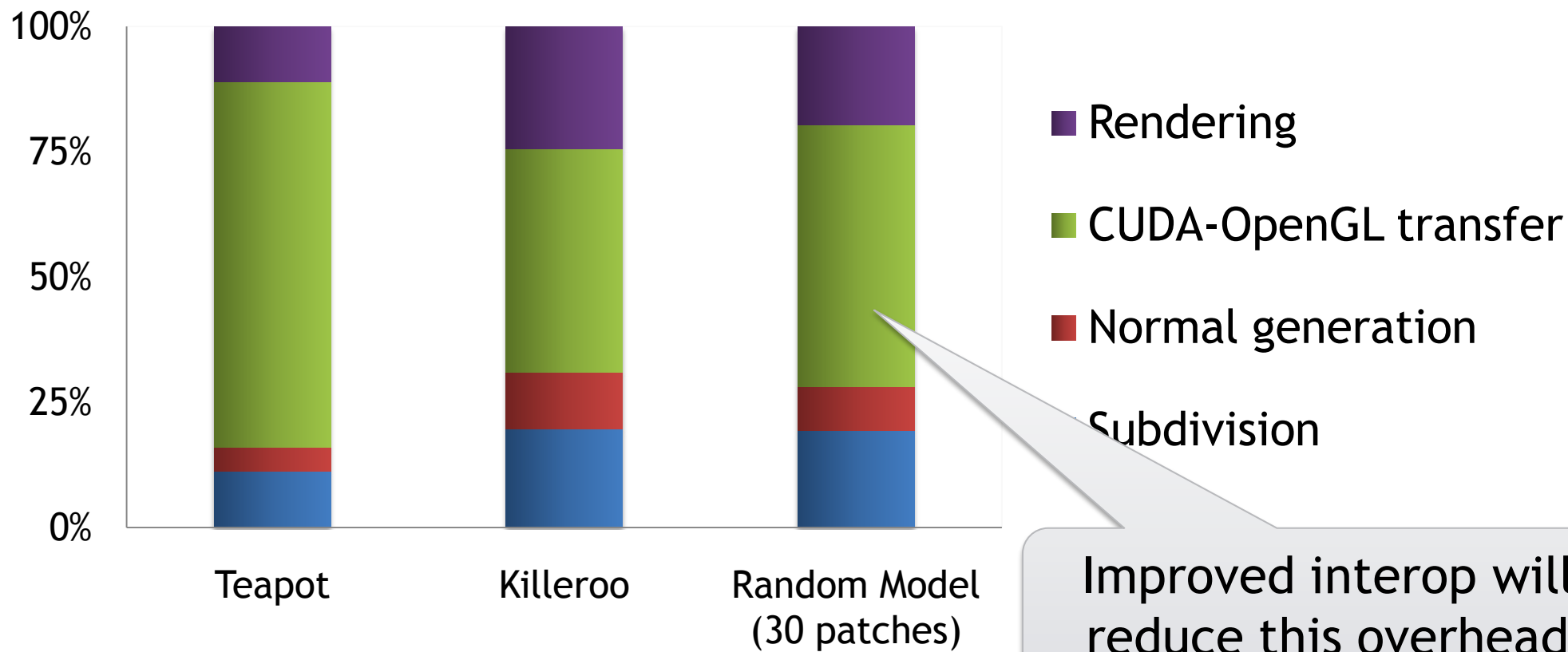
<http://research.microsoft.com/en-us/um/people/cloop/eisenetal2009.pdf>

- **Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces**

Anjul Patney, Mohamed S. Ebeida, John D. Owens

http://www.idav.ucdavis.edu/publications/print_pub?pub_id=964

Subdivision Surfaces



Physics

- Physics is highly parallel
 - Rigid bodies, fluids, cloth simulation, etc.
- Accelerated by Compute Shader
 - Inter-thread sharing, and scattered writes
- Rigid Body, Cloth and Fluid Physics for CUDA
 - October 2, 1:30-2:30 pm, in Garden Room

Rendering

- Lighting Models
 - Share the results of lighting computations.
 - Linear algebra operations also benefit from inter-thread sharing
- Deferred Shading
 - Graphics pipeline rasterizes gbuffers for opaque surfaces
 - Compute pipeline uses gbuffers, culls light sources, computes lighting & combines with shading

Deferred Shading

- The screen is divided into tiles
 - One thread group per tile; one thread per pixel
 - Determine light sources intersection with each tile
 - Only apply the visible light sources on pixels in each tile
- Shared memory → Read gbuffers & depth only **once**
- Parallel Graphics in Frostbite –Current & Future
SIGGRAPH 09 Course: Beyond Programmable Shading
<http://s09.idav.ucdavis.edu/talks/04-JAndersson>
ParallelFrostbite-Siggraph09.pdf

Others

- Particle system
- Sorting
- Order independent transparency
- Video encoding
- Scientific computing

Outline

- Introduction
- Case study: Rolling Box Blur
- Applications
- Optimizations
 - Memory optimization
 - Execution configuration optimization

Minimizing host-device data transfer

- Minimize switch between running a compute shader and a graphics shader
- Minimize host-device data transfer
 - Host device data transfer has much lower bandwidth than global memory access.
 - Group transfer: One large transfer much better than many small ones

Memory Regions

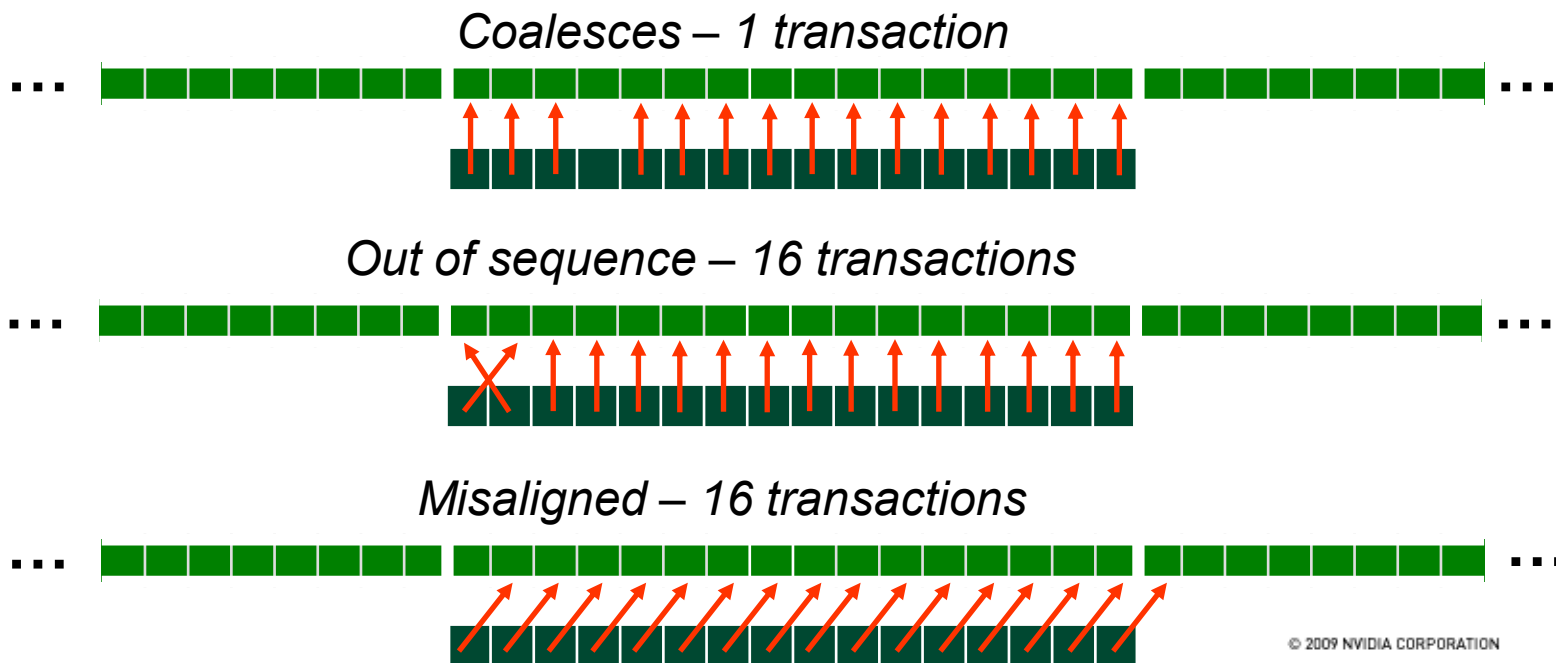
	Scope	Lifetime	Access	Location
Local scalar	thread	thread	read/write	on-chip (register)
Local array	thread	thread	read/write	off-chip
Shared variable	group	group	read/write	on-chip (RAM)
Constant	global	persistent	read-only	on-chip (cache)
Shader Resource View	global	persistent	read-only	off-chip
Unordered Access View	global	persistent	read/write	off-chip

Global Memory Access

- Reduce global memory access → most important performance consideration!
 - Use shared memory
 - Ensure global memory accesses are coalesced whenever possible.

Coalescing in Compute Capability 1.0 and 1.1

- Reads and writes to structured buffers should be linear and aligned



Shared Memory Access

- Shared memory Latency $\sim 100\times$ smaller than global memory
 - Think of it as a small (but fast) user-managed cache
 - Use shared memory to save bandwidth
 - Reads and writes to shared memory are very fast compared to global (buffer) loads and stores
 - Re-use data in the shared memory as much as possible

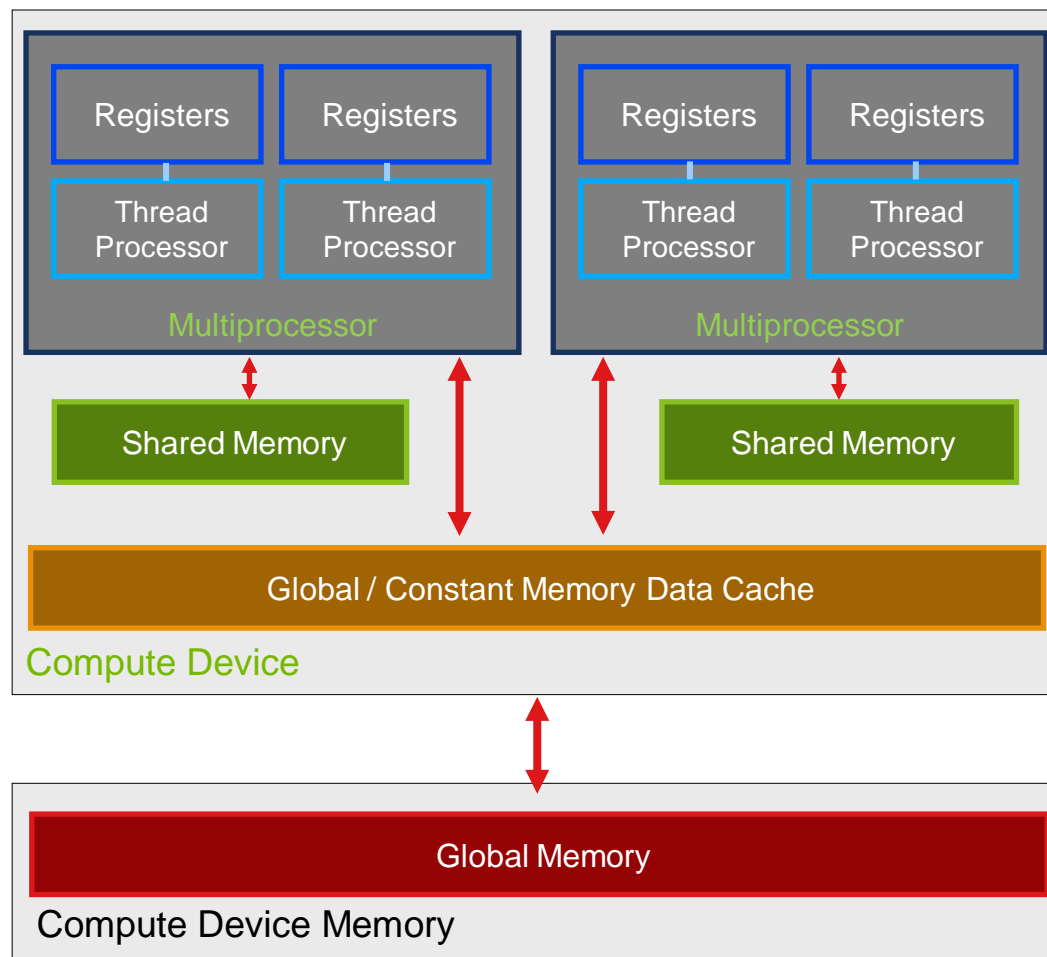
Memory Access

- Minimize the number of temporary variables (registers) used in your compute shader.
- If your algorithm requires unpredictable random read accesses, use textures.
- Try to group structure members that will be accessed together close to each other in the structure.
 - This minimizes the number of memory transactions that will be necessary to load the data

Outline

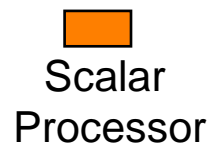
- Introduction
- Case study: Rolling Box Blur
- Applications
- Optimizations
 - Memory optimization
 - Execution configuration optimization

CUDA Architecture

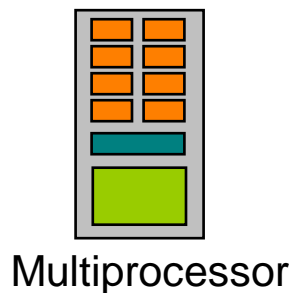
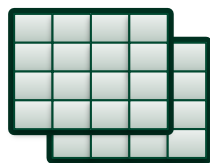


Execution Model

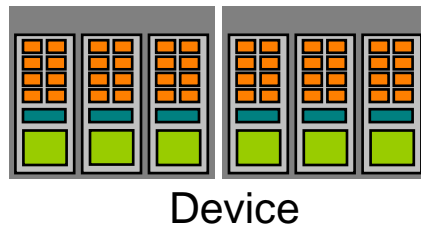
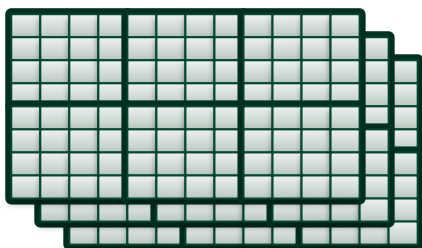
DirectCompute Hardware



A thread is executed by a scalar processors



A thread group is executed on a multiprocessor



A compute shader kernel is launched as a grid of thread-groups

Only one grid of thread groups can execute on a device at one time

Thread groups

- Thread groups should be multiples of 32 threads in size
- # of thread groups > # of multiprocessors
- More than one thread group per multiprocessor
- Amount of allocated shared memory per thread group should be at most half the total shared memory per multiprocessor

Thread Heuristics

- The number of threads per thread group should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies

Occupancy Considerations

- Occupancy is
 - a metric to determine how effectively the hardware is kept busy
 - the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps
- Increase occupancy to achieve latency hiding

Occupancy Considerations

- After some point (e.g. 50%), further increase in occupancy won't lead to performance increase
 - Limited number of registers allowed
 - Minimize the number of registers used in CS
 - Maximum number of warps


Control Flow

- No branching within a warp
 - different execution paths within a warp must be serialized, increasing the total number of instructions.
- No penalty if different warps diverge
 - No divergence if controlling condition depends only on `local_id/warp_size`

Control Flow

- Optimization by compiler
 - Unroll loops
 - Branch prediction
- Optimization by programmer
 - Use **[unroll]** in shader code

Final Remarks

- Compute enables efficient implementation of new rendering algorithms
- Hybrid algorithms are the future
 - Leverage the power of rasterization with the flexibility of compute
- Try out NVIDIA Nexus 
 - October 2, 10:30-11:30 am, in Piedmont Room

Questions?

Tianyun Ni
tni@nvidia.com