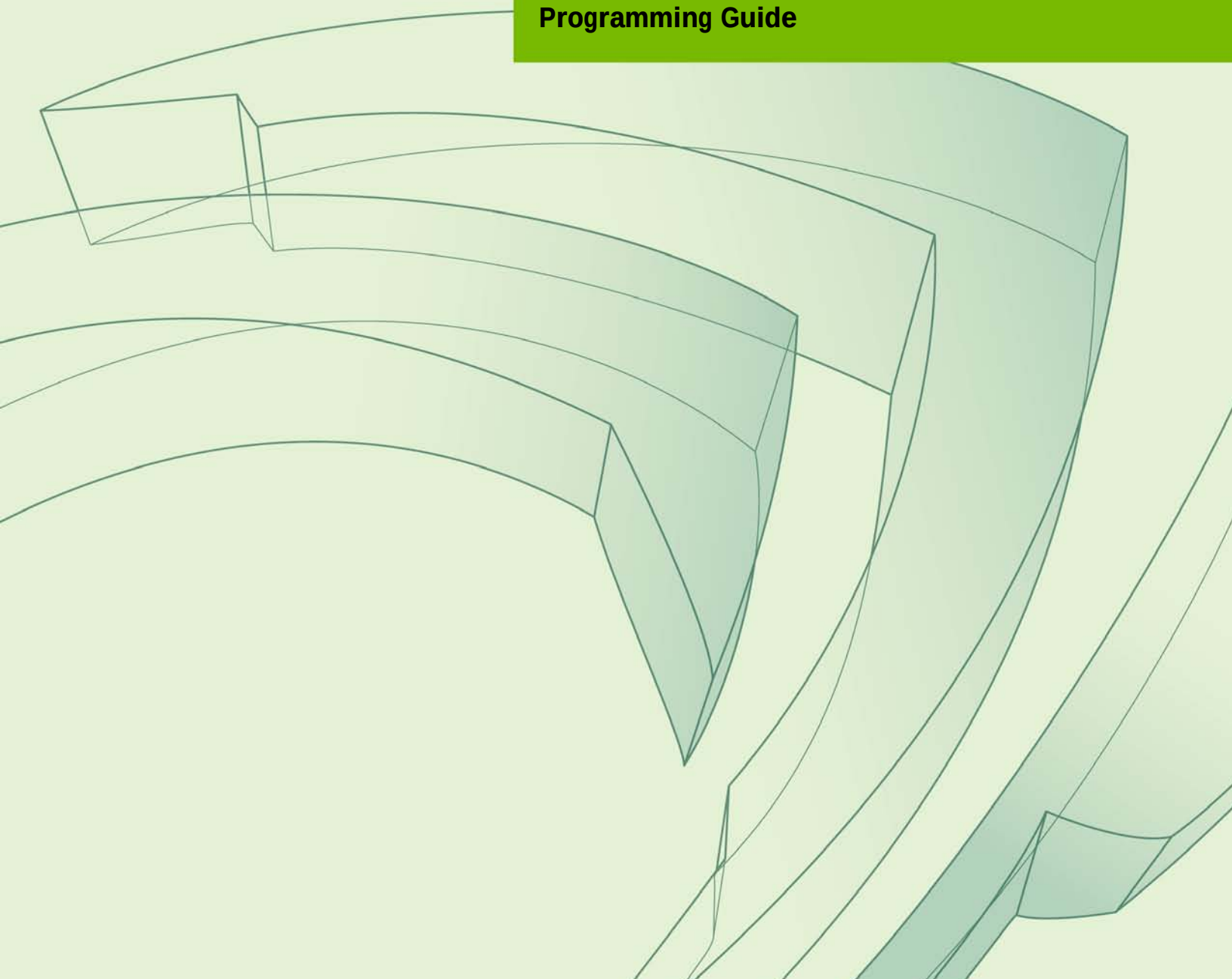
DirectCompute PROGRAMMING GUIDE

PG-05629-001_v3.2 | December 2010

Programming Guide



DOCUMENT CHANGE HISTORY

PG-05629-001_v3.2

Version	Date	Authors	Description of Change
1.0	15 April 2009	Simon Green	Initial release
2.3	31 August 2009	Simon Green	Public Release
3.2	December 15, 2010	TechPubs	Assigned new template.

TABLE OF CONTENTS

INTRODUCTION

It is now widely accepted that the GPU has evolved into a highly capable general purpose processor capable of improving the performance of a wide variety of parallel applications beyond graphics. NVIDIA's CUDA architecture has led the way in proving the compute capabilities of the GPU, and provides the infrastructure that DirectCompute is built on.

At the same time, Microsoft's DirectX APIs have matured into the standard interface for utilizing graphics hardware on Windows platforms, both for video games and consumer graphics applications such as photo and video editing.

The introduction of DirectCompute allows developers to take advantage of the massive parallel computation abilities of today's GPUs directly from within DirectX applications, without the need to use a separate compute API.

It is supported on both current DirectX 10 hardware (NVIDIA GeForce 8 series and later) and forthcoming DirectX 11 GPU hardware.

THE COMPUTE SHADER

DirectCompute exposes the compute functionality of the GPU as a new type of shader - the compute shader, which is very similar to the existing vertex, pixel and geometry shaders, but with much more general purpose processing capabilities.

The compute shader is not attached specifically to any stage of the graphics pipeline, but interacts with the other stages via graphics resources such as render targets, buffers and textures.

Unlike a vertex shader (which is executed once for each input vertex), or a pixel shader (which is executed once for each pixel), the compute shader doesn't have to have a fixed

mapping between the data it is processing and the threads that are doing the processing. One thread can process one or many data elements, and the application can control directly how many threads are used to perform the computation.

The compute shader also allows unordered memory access, in particular the ability to perform writes to any location in a buffer (also known as scattered writes). This was possible in a limited way previously by rendering point primitives, but this method was not efficient.

The last major feature of DirectCompute is thread group shared memory (referred to from now on as simply shared memory). This allows groups of threads to share data, and can reduce bandwidth requirements significantly.

Together, these features allow more complex data structures and algorithms to be implemented that were not previously possible in Direct3D, and can improve application performance considerably.

It is worth noting that, as with other Compute APIs, Compute Shaders do not directly support any fixed-function graphics features with the exception of texturing. This includes rasterization, depth and stencil test, blending and derivatives. Future versions of the compute shader will likely offer tighter integration with the fixed function hardware.

DirectCompute ADVANTAGES

DirectCompute has several advantages over other GPU computing solutions:

- It is integrated with Direct3D – this means it has efficient interoperability with D3D graphics resources (textures, buffers etc.).
- It includes all texture features (including cube maps and mip-mapping). LOD must be specified explicitly.
- It uses the familiar HLSL shading language.
- It provides a single API across all graphics hardware vendors on Windows platforms.
- It gives some level of guarantee of consistent results across different hardware and over time.
- Out of bounds memory checking?

DirectCompute APPLICATIONS

DirectCompute has an almost unlimited number of possible applications, but its main strength lies in applications that are closely coupled with graphics. The main applications it was designed for are:

- Photo and video processing for consumer applications.
- Image post-processing for games.
- Game physics and artificial intelligence.
- Advanced rendering effects – order independent transparency, ray tracing and global illumination.

COMPUTE SHADER FEATURES

DirectCompute is part of the DirectX 11 API, but it is possible to create a Direct3D 11 device on current DirectX 10 hardware (see below).

Since DirectCompute is based on the existing Direct3D shader infrastructure and the HLSL language, if you are used to DirectX 10 shader programming it is very simple to learn.

The DirectX 11 API is very logically designed and simple to use even if you are used to other compute APIs.

READ/WRITE BUFFERS AND TEXTURES

In DirectX 10, buffers and textures are read-only. DirectX 11 adds a new set of resources that can be both read from and written to in the same shader:

- RWBuffer
- RWTexture1D, RWTexture1DArray
- RWTexture2D, RWTexture2DArray
- RWTexture3D

Note that DirectX 10 hardware does also not support typed UAVs, so it is not possible to write to textures directly.

STRUCTURED BUFFERS

A structured buffer is a buffer that contains elements of a structure type. Here is a simple example of a structured buffer in HLSL:

```
struct Particle
{
    float4 position;
    float4 velocity;
};
RWStructuredBuffer<Particle> particles;
```

To access the buffer from the compute shader code we simply use the array indexing notation to read from or write to the buffer, for example:

```
float4 vel = particles[i].velocity;
particles[i].position += vel;
```

Note that on DirectX 10 hardware, we generally recommend using structure of arrays (SOA) layout instead of the array of structures (AOS) layout above, so that the memory accesses by each thread are contiguous. Unfortunately, on DirectX 10 hardware, DirectCompute only permits a single UAV, so the only way to do this is in this case is to create a single buffer of twice the size containing all the particle positions followed by all the velocities:

```
RWStructuredBuffer<float> particlePositionAndVelocity;
```

To create a structured buffer from the API, we can use something like the following code, making sure to set D3D11_BIND_UNORDERED_ACCESS in the bind flags so that it can later be bound as an unordered access view:

```
ID3D11Buffer *pStructuredBuffer;

// Create Structured Buffer
D3D11_BUFFER_DESC sbDesc;
sbDesc.BindFlags            = D3D11_BIND_UNORDERED_ACCESS |
D3D11_BIND_SHADER_RESOURCE;
sbDesc.CPUAccessFlags       = 0;
sbDesc.MiscFlags            = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
sbDesc.StructureByteStride  = sizeof(D3DXVECTOR4);
sbDesc.ByteWidth            = sizeof(D3DXVECTOR4) * numParticles * 2;
sbDesc.Usage                = D3D11_USAGE_DEFAULT;
pd3dDevice->CreateBuffer(&sbDesc, 0, &pStructuredBuffer);
```

Note that the `D3D11_BIND_SHADER_RESOURCE` flag has also been set, so this buffer could also be bound as a shader resource for reading in pixel shader, for example.

BYTE ADDRESS (RAW) BUFFERS

Byte address buffers (also known as raw buffers) are a special type of buffer which are addressed using a byte offset (from the beginning of the buffer). This can be contrasted with regular Direct3D buffers which are indexed using element indices and so automatically take into account the size of the type. The offset must be a multiple of 4 so that it is word aligned.

The contents of raw buffers are always 32-bit unsigned ints, but it is possible to store other data types by casting them using HLSL functions such as asfloat().

Raw buffers can be bound as vertex buffers and index buffers and so are useful for generating geometry from compute shaders.

They are declared in HLSL like this:

```
ByteAddressBuffer
RWByteAddressBuffer
```

UNORDERED ACCESS VIEWS

Unordered Access Views (UAVs) are a new type of view introduced in Direct3D 11. UAVs allow unordered read/write access from multiple threads.

On DirectX 10 hardware, UAV access is only possible from compute shaders, but on Direct3D 11 hardware it is possible from pixel shaders as well, which will open up interesting possibilities such as A-buffer algorithms that store a list of fragments per pixel.

To create a resource that can be used with a UAV, we need to add the D3D11_BIND_UNORDERED_ACCESS flag at resource creation time.

For example:

```
ID3D11UnorderedAccessView *pStructuredBufferUAV;

// Create the UAV for the structured buffer
D3D11_UNORDERED_ACCESS_VIEW_DESC sbUAVDesc;
sbUAVDesc.Buffer.FirstElement       = 0;
sbUAVDesc.Buffer.Flags              = 0;
sbUAVDesc.Buffer.NumElements        = numParticles * 2;
sbUAVDesc.Format                    = DXGI_FORMAT_UNKNOWN;
sbUAVDesc.ViewDimension             = D3D11_UAV_DIMENSION_BUFFER;
pd3dDevice->CreateUnorderedAccessView(pStructuredBuffer,
                 &sbUAVDesc, &pStructuredBufferUAV) );
```

To bind the UAV to the compute shader we use the following code:

```
UINT initCounts = 0;
m_pd3dImmediateContext->CSSetUnorderedAccessViews(0, 1,
                              &m_pStructuredBufferUAV, &initCounts);
```

COMPUTE SHADER INVOCATION

Compute Shaders are executed using the Dispatch call. This is the equivalent of the Draw call in graphics shaders, and specifies directly how many thread groups to create in each dimension:

```
pD3D11Device->Dispatch(UINT nX, UINT nY, UINT nZ);
```

Note that in DirectCompute, the number of threads within each group is specified in the shader source code (see below).

On DirectX 11 hardware, the maximum number of thread groups is 65535 in each dimension. On DirectX 10 hardware the Z dimension must be 1, as described later.

In some cases it is useful to be able to specify the number of thread groups based on a previous compute shader, which avoids having to read this information back to the host CPU. The DispatchIndirect function executes a compute shader taking the number of thread groups directly from a buffer resource:

```
pD3D11Device->DispatchIndirect(ID3D11Buffer *pBufferForArgs,
 UINT AlignedByteOffsetForArgs);
```

Note that this function is not supported on DirectX 10 hardware (compute shader 4.0), and will be ignored (see below).

API STATE

The compute shader has its own set of state, just as pixel, vertex and geometry shaders do. Like the other shader types, there are methods on the D3DDevice to set the additional state:

```
pD3D11Device->CSSetShaderResources()
```

-bind memory resources of buffer or texture type

```
pD3D11Device->CSSetConstantBuffers()
```

-bind read-only buffers that store data that does not change during shader execution

```
pD3D11Device->CSSetSamplers()
```

-bind sampler state that controls how any texture resources are sampled

```
pD3D11Device->CSSetUnorderedAccessViews()
```

- bind unordered access views

```
pD3D11Device->CSSetShader()
```

-bind the compute shader object

The syntax for these methods is the same as the corresponding calls for other Direct3D11 shaders.

HLSL SYNTAX

Unlike other compute APIs, in DirectCompute the number of threads in the thread group is specified as an attribute in the HLSL source code. This allows the compiler to make optimization decisions based on the number of threads, and verify that the code will run correctly. The syntax is as follows:

```
[numthreads(16,16,1)]
void CS(…)
{
     // shader code
}
```

The Dispatch call specifies the number of thread groups to create:

```
pD3D11Device->Dispatch(10, 10, 1);
```

The code above will launch a grid of 10 x 10 thread groups, each group made of 16 x 16 (256 total) threads. This means the GPU will process a total of 10 x 10 x 256 = 25600 threads.

The compute shader adds some new system generated values that allow the shader to determine which thread group, and which thread within that group it is processing:

```
uint3 groupID : SV_GroupID
```

- the index of the group within the dispatch (for each dimension).

```
uint3 groupThreadID : SV_GroupThreadID
```

- the index of the thread within the group.

```
uint groupIndex : SV_GroupIndex
```

- a flattened 1D thread index within the group. This is provided for convenience and is computed as:

```
groupIndex = groupThreadID.x*dimx*dimy +
             groupThreadID.y*dimx + groupThreadID.x;
```

Where dimx and dimy are the dimensions of the group specified by the numthreads attribute in the shader.

Finally, for convenience there is a global thread index whithin the dispatch:

```
uint3 dispatchThreadID  : SV_DispatchThreadID
```

THREAD GROUP SHARED MEMORY

Many of today's applications are memory bandwidth limited, and shared memory has proven to be a powerful tool in improving the performance of such applications.

Shared memory is the main reason for the existence of thread groups.

Shared memory allows threads within a given group to cooperate and share data. Reads and writes to shared memory are very fast compared to global (buffer) loads and stores, close to the speed of register reads and writes.

A common programming pattern is to have the threads within a group cooperatively load a block of data into shared memory, process the data in the fast shared memory, and then finally write the results back to a writable buffer. The performance improvement obtained from using shared memory depends largely on how much the data is re-used in shared memory.

In DirectCompute, shared memory is indicated using the "groupshared" type qualifier, for example:

```
groupshared float smem[256];
```

The compiler checks at compile time that the total amount of shared memory does not exceed the limit defined for the shader model. Note that there is no way of specifying the size of shared memory at runtime in DirectCompute.

On DirectX 10 hardware you are also limited to only one shared memory array.

Synchronization between threads is achieved using the HLSL function:

```
GroupMemoryBarrierWithGroupSync();
```

This function ensures that all threads within the thread group have reached this point before execution continues. This function cannot appear within dynamic flow control.

DirectCompute on DirectX 10 HARDWARE

DirectCompute includes a sub-set of the full DirectX 11 functionality that runs on existing DirectX 10 hardware (known as "downlevel" hardware in the DirectX documentation). There are a number of limitations, but they are not insurmountable and this feature allows developers to start developing DirectCompute software today.

DirectCompute on DirectX 10 hardware is exposed as a new version of the compute shader, CS 4.0, which is based on the vertex shader 4.0 instruction set.

It is necessary to use the DirectX 11 API, but it is possible to create a DirectX 11 device with a Direct3D 10.0 (shader model 4.0) "feature level" on DirectX 10 hardware.

To check if your hardware supports this you can use the caps bit:

ComputeShaders_Plus_RawAndStructuredBuffers_Via_Shader_4_x

which indicates support for both compute shader 4.0 and raw and structured buffers. See the Microsoft DirectX 11 SDK documentation for more details.

The main features that are missing from CS 4.0 compared to CS 5.0 are:

- Atomic operations
- Append/consume
- Typed unordered access views (UAVs).
  Note that this means that texture UAVs are also not supported, so it is not possible to write directly to textures on DirectX 10 hardware. It is possible to work around this limitation by using a pixel shader that reads from a buffer resource and writes to a render target texture.
- Double precision
- DispatchIndirect()

There are a number of additional limitations:

- Only a single UAV can be bound to the pipeline at once. This is not a terrible restriction in practice, since you can store multiple arrays together in a single buffer. It is still possible to bind other read-only shader resources (for example texture views) at the same time.
- The Z dimension of the thread group grid must be 1 (i.e. there are no 3D grids).
- The total number of threads in a group is limited to maximum of 768 (X*Y*Z), compared to 1024 on D3D11 hardware.
- The thread group shared memory is limited to 16KB total, compared to 32KB on D3D11 hardware.

Many of these restrictions are enforced by the HLSL compiler, and will cause a compile error if you violate them.

THREAD GROUP SHARED MEMORY RESTRICTIONS

Compute Shader 4.0 has some additional restrictions on how thread group shared memory can be used. It is worth noting that these restrictions are for cross-vendor compatibility and are not present when using shared memory from other compute APIs on NVIDIA hardware.

Each thread can read from any location in shared memory, but can only write to the position indexed by SV_GroupIndex.

Only one shared memory variable can be used in a shader at a time.

Shared memory bank conflicts exist as in other compute APIs.

OPTIMIZING DirectCompute ON NVIDIA HARDWARE

Much of our standard advice about optimizing for CUDA also applies to DirectCompute.

Context Switching

Each time your program switches between running a compute shader and a graphics shader causes a hardware context switch. You should try to avoid switching too often, and ideally only once per frame.

Thread Groups

Thread groups should be multiples of 32 threads in size. In order to make full use of the resources of the GPU, there should be at least as many thread groups as there are multiprocessors on the GPU, and ideally two or more.

You should try and minimize the number of temporary variables (registers) used in your compute shaders.

Use shared memory to save bandwidth where possible. You can think of it as a small (but fast) user-managed cache.

Memory Access

For optimal memory access performance on the NVIDIA GeForce 8/9 series there are some restrictions if you want to achieve maximum memory bandwidth. This is less of an issue on GeForce GTS series, which has more flexible memory coalescing hardware.

Reads and writes to structured buffers should be linear and aligned, so that thread *i* always reads or writes to location *i*. This allows hardware to "coalesce" memory accesses into a minimum number of transactions.

If your algorithm requires unpredictable random read accesses, use textures.

CONCLUSION

DirectCompute offers a simple way of adding general purpose algorithms to existing DirectX applications and accelerating operations that would traditionally be performed on the CPU. It includes a functional sub-set that will run on current DirectX 10 hardware and allows developers to start programming the next generation of interactive applications today

Notice

OpenCL

Trademarks

Copyright