Microsoft DirectCompute on the 2nd Generation Intel® Core™ Processor

By Wolfgang Engel

This is the second and last part of the Microsoft DirectCompute series. This article covers programming DirectCompute with Microsoft DirectX* 10–class graphics processors, while the previous article focused on Microsoft DirectX 11–class hardware. Each article can exist on its own as each one provides all the information necessary to begin programming DirectX 10– or 11–class hardware.

DirectCompute on DirectX* 10 and 11 Hardware

DirectCompute on DirectX 10.x hardware is exposed as new versions of the compute shader with the cs_4_0 and cs_4_1 profiles in the DirectX 11 application programming interface (API). Here is a list of features with no or restricted support in DirectX 10.x:

- **Device memory.** Only one unordered access view (UAV) is available, and you can't use it to write into a 2D texture (no `RWTexture*` support). Only `RWStructuredBuffer`s and `RWByteAddressBuffer`s are available as UAVs.
- **Thread group shared memory (TGSM).** This memory is limited to 16 KB per group, and a single thread is limited to a 256-byte region of it. TGSM can only be accessed using `SV_GroupIndex` or `SV_DispatchThreadID`.
- **Dispatching Kernels.** For dispatching kernels:
  - `DispatchIndirect()` is not supported.
  - `D3D11_CS_4_X_THREAD_GROUP_MAX_THREADS_PER_GROUP` shows the overall number of threads per group, with `768; D3D11_CS_4_X_THREAD_GROUP_MAX_X / *Y` showing that this maximum number of threads is available for the *x* and *y* direction.
  - Similarly, the number of thread groups is limited in the *z* direction of a dispatch call to 1, as shown in `D3D11_CS_4_X_DISPATCH_MAX_THREAD_GROUPS_IN_Z_DIMENSION`.
- **Atomic operations.** Atomic operations are not available.
- **Double-precision.** Double-precision is not supported.

Device Memory

DirectCompute on DirectX 10.x hardware only supports structured and byte address buffers, also called *raw buffers.* All textures are read only—in other words, there is no support for a UAV to write to textures (no `RWTexture*`).

To read from textures and read or write into buffers, *memory views* are available. DirectX 11 introduced the UAV that allows scattered writes and gathered reads. To read memory in a shader, DirectX 10 introduced a shader resource view (SRV).

Structured Buffers

A *structured buffer* is a buffer that contains elements of a structure. Here's a simple example:

```
// Compute Shader code: structured buffer with Unordered Access View in u0
struct BufferStruct
{
  float4 color;
};
RWStructuredBuffer<BufferStruct> output : register(u0);
```

To fill up a structured buffer in the Compute shader, you can use code like this:

```
uint stride = WindowWidth;

// buffer stride, assumes data stride =
// data width (i.e. no padding)
// DTid is the SV_DispatchThreadID
uint idx = (DTid.x) + (DTid.y) * stride;
output[idx].color = color;
```

The following code creates a structured buffer on the application level:

```
//
// structured buffer
//
struct BufferStruct
{
  float color;
};

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(BufferStruct);

int Height = WindowHeight;
int Width = WindowWidth;
sbDesc.ByteWidth = ((sbDesc.StructureByteStride * gWidth * gHeight  + 63) / 64) * 64;
sbDesc.Usage = D3D11_USAGE_DEFAULT;
pd3dDevice->CreateBuffer(&sbDesc, NULL, &pStructuredBuffer);
```

Byte Address Buffers

Byte address buffers, or *raw buffers,* are a special type of buffer addressed using a byte offset from the beginning of the buffer. The byte offset must be a multiple of 4 so that it is word aligned.

The type of raw buffers is always 32-bit unsigned `int`. Other data types would need to be casted to unsigned `int`. Raw buffers are useful for generating geometry with DirectCompute, because they can be bound as vertex and index buffers. In High-level Shading Language (HLSL), they are declared as follows:

```
ByteAddressBuffer
RWByteAddressBuffer
```

In DirectX 10.x, the base offset for a raw buffer must be aligned to 256-byte boundary.

Shader Resource View and Unordered Access View

Similar to the other shader stages in the DirectX pipeline, an SRV is supported in DirectCompute to allow a shader to read resource memory. In case of a structured buffer, you can create an SRV as follows:

```
//
// shader resource view on structured buffer
//
D3D11_SHADER_RESOURCE_VIEW_DESC sbSRVDesc;
ZeroMemory( &sbSRVDesc, sizeof( sbSRVDesc ) ); sbSRVDesc.Buffer.ElementOffset = 0;
sbSRVDesc.Buffer.ElementWidth = sbDesc.StructureByteStride; sbSRVDesc.Buffer.FirstElement =
sbUAVDesc.Buffer.FirstElement; sbSRVDesc.Buffer.NumElements = sbUAVDesc.Buffer.NumElements;
sbSRVDesc.Format = DXGI_FORMAT_UNKNOWN;
sbSRVDesc.ViewDimension = D3D11_SRV_DIMENSION_BUFFER;
hr = pd3dDevice->CreateShaderResourceView((ID3D11Resource *) pStructuredBuffer, &sbSRVDesc,
&pComputeShaderSRV);
```

A UAV allows you to randomly scatter writes into byte address or raw buffers and structured buffers, and then randomly gather while reading those buffers. In DirectX 10.x, only a single UAV could be bound to the pipeline at any time, while in DirectX 11, eight UAVs can be bound at the same time. DirectX 11.1 seems to allow you to even use a larger number than that.

Code for a UAV to a structured buffer might look like this:

```
// Unordered access view on structured buffer
D3D11_UNORDERED_ACCESS_VIEW_DESC sbUAVDesc;
ZeroMemory( &sbUAVDesc, sizeof(sbUAVDesc) ); sbUAVDesc.Buffer.FirstElement = 0;
sbUAVDesc.Buffer.Flags = 0;
sbUAVDesc.Buffer.NumElements = sbDesc.ByteWidth / sbDesc.StructureByteStride;
sbUAVDesc.Format = DXGI_FORMAT_UNKNOWN;
sbUAVDesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
HRESULT hr = pd3dDevice>CreateUnorderedAccessView((ID3D11Resource *)pStructuredBuffer,
&sbUAVDesc, &pComputeOutputUAV);
```

DirectX 10.x does not support `RWTextures*` and therefore doesn't allow it to write into a render target with a compute shader. To move data from a structured buffer into the back buffer, a pixel shader needs to read the data from the structured buffer via an SRV and write it into the back buffer via a render target view.

Thread Group Shared Memory

TGSM is located in on-chip memory. You can consider it a cache to minimize off-chip bandwidth use. This memory is accessed by all the threads in a thread group. In other words, TGSM allows threads within a given group to cooperate and share data. Reads and writes to shared memory are fast compared to global buffer loads and stores.

A common programming pattern is to have the threads within a group cooperatively load a block of data into shared memory, process the data, and then write out the results to a writable buffer. A typical example is storing all of the neighboring pixels for horizontal or vertical blur kernels for a post-processing pipeline.

TGSM is not persistent between dispatch calls. So, the result of one dispatch call needs to be stored somewhere else. TGSM is indicated in the HLSL shader code using the `groupshared` type qualifier:

```
groupshared float sharedmem[256];
```

In DirectX 10.x, TGSM is limited to 16 KB per group, and a single thread is limited to a 256-byte region of it. TGSM in DirectX 10 can be read from any location in shared memory but can only write to the position indexed by `SV_GroupIndex` or `SV_DispatchThreadID`. On DirectX 10–class hardware, only one shared memory variable can be used in a shader at a time.

DirectCompute Threading Model

The typical multi-threading paradigm used in traditional CPU-based algorithms uses separate processor cores and threads for execution, coupled with a shared memory space and manual synchronization. High-end CPUs like the 2nd generation Intel® Core™ processor have up to six cores, each supporting up to two threads.

DirectCompute uses a different threading model. DirectCompute-capable devices can run thousands of threads, with flexible mapping of threads to data elements, while the same shader or program executes them all—a process called *kernel processing.*

Kernel Processing

A compute shader is considered a processing kernel when executed. A kernel is instantiated for each thread and applied to a set of data. The data is provided through Microsoft Direct3D* resources bound to the DirectCompute stage. In other words, each hardware thread can be tasked with executing one individual invocation of a kernel that is the same for all threads in a dispatch call.

Typical data for a DirectCompute application consists of small parts so that it can be processed separately. In other words, a typical DirectCompute application requires data that consists of a large number of similarly structured pieces of data—the classical domain of graphics hardware.

Dispatching Kernels

Executing a compute shader is also called *dispatching a kernel.* DirectX 10.x only supports one function to dispatch a kernel:

```
Dispatch(UINT ThreadGroupCountX, UINT ThreadGroupCountY, UINT ThreadGroupCountZ);
```

This method expects three values that represent the number of thread groups in three dimensions that should be dispatched. For example, if an application calls the `Dispatch()` method with 4, 8, and 2, a total of 64 thread groups will be launched. The number of threads in each of those thread groups is specified in the compute shader.

The following code snippet shows a typical example of how to call `Dispatch()` and provide the number of threads in a thread group:

```
// C++ application code
pImmediateContext->Dispatch(Width / THREADSX, Height / THREADSY, 1 );

// HLSL compute shader code
[numthreads(THREADSX, THREADSY, 1)]
void CS_QJulia4D( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID, uint GI : SV_GroupIndex )
…
```

In case of the thread group for *x,* the width of the window is divided by the number of threads that should be in the thread group. The number of threads is defined in the HLSL shader code. With 16 threads in each thread group and a window size of 800, the application will use 50 thread groups consisting of 16 threads each. For the *y* direction, if the window has a height of 640, there will be 20 thread groups consisting of 32 threads. This example dispatches 1000 thread groups, each with 512 threads. So, 512,000 threads are in flight.

In the case of DirectX 10.x, the *z* parameter of a thread group in `numthreads` can only take 1 as an input. In other words, DirectX 10.x only supports 2D thread groups, while DirectX 11.x supports 3D groups, as well. Think of the threads in a thread group in DirectX 10.x as a 2D array and those in DirectX 11 as a 3D array.

A thread in a thread group is addressed by using registers that hold the dimensions of the threads and thread groups.

Thread Addressing System

Each of the 512,000 threads executes an instance of a kernel or a compute shader. How does each kernel know which thread is responsible for its execution? Knowing which thread is executing the kernel is important for indexing data, and then reading and writing data from or to Direct3D resources.

The DirectCompute runtime provides system values stored in registers to a kernel. Four registers hold this data:

- `vThreadID.xyz`
- `vThreadGroupID.xyz`
- `vThreadIDInGroup.xyz`
- `vThreadIDInGroupFlattended`

The values those registers hold are accessible in the compute shader via the following semantics:

- **`SV_DispatchThreadID`.** An index of the thread within the entire dispatch in each dimension: *x* - 0..*x* – 1; *y* – 0..*y* – 1; *z* – 0..*z* – 1
- **`SV_GroupID`.** An index of a thread group in the dispatch—for example, calling `Dispatch(2,1,1)` results in possible values of 0,0,0 and 1,0,0, varying from 0 to (`numthreadsX * numthreadsY * numThreadsZ`) – 1
- **`SV_GroupThreadID`.** An index of a thread in a thread group in each dimension—for example, if you specified `numthreads(3,2,1)`, possible values for the `SV_GroupThreadID` input value have the range of values (0–2, 0–1, 0)
- **`SV_GroupIndex`.** A flattened 1D index (`uint`) to show to which thread group a thread belongs

A simple example for writing into a 2D texture is shown in the following source code:

```
RWTexture2D<float4> output : register (u0);
void CS_QJulia4D( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID, uint GI : SV_GroupIndex )
{
…
	output[DTid.xy] = color;
}
```

The following code shows how to access a 1D structured buffer in a compute shader:

```
struct BufferStruct
{
	float4 color;
};
RWStructuredBuffer<BufferStruct> output : register (u0); // UAV 0
void CS_QJulia4D( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID, uint GI : SV_GroupIndex )
{
…
	uint stride = c_width;
```

```
        uint idx = (DTid.x) + (DTid.y) * stride;

        output[idx].color = color;
}
```

Thread Synchronization

As with traditional multi-threaded programming models, many threads can read and write the same memory location, and therefore there is a potential for memory corruption resulting from read-after-write hazards. To synchronize memory access of threads, *memory barriers* are available.

Memory Barriers

In DirectX 10.x, six different HLSL intrinsics, called *memory barriers,* can synchronize thread execution and memory writes:

- AllMemoryBarrier/*WithGroupSync
- DeviceMemoryBarrier/*WithGroupSync
- GroupMemoryBarrier/*WithGroupSync

A *memory barrier* is a method for saying, "wait until the memory operations are complete." You use these barriers to ensure that when threads share data with one another, the desired values written to memory have had a chance to be written before being read by other threads. There is an important distinction here between the shader core executing a Write instruction and that instruction actually being carried out by the GPU's memory system and written to memory. Depending on the underlying hardware, there can be a variable amount of time between writing a value and when it actually ends up at its memory destination.

There are `*MemoryBarriers` for TGSM, device memory, and both memory types. The `*MemoryBarrierWithGroupSync` stalls until outstanding memory operations, which are active at the time of calling, have finished *and* all threads in the group have hit the instruction.

`AllMemoryBarrier` says that no memory access can be moved across this barrier. So until this barrier is hit, there is no guarantee that what was written to memory will be visible to other threads. `AllMemoryBarrierWithGroupSync` adds a threading barrier, as well; all threads must hit this statement before any can continue.

`DeviceMemoryBarrier` says that no UAV access can be moved across this barrier, and `GroupMemoryBarrier` says that no group shared memory access can be moved across this barrier. In both cases, the `GroupSync` version adds a threading barrier.

A typical example for using a memory barrier is shown in the following code:

```
for (uint tile = 0; tile < numTiles; tile++)
{
    sharedPos[threadId] = particles[…];

    GroupMemoryBarrierWithGroupSync();

    // gravitation() uses sharedPos[] as input data
    acceleration = gravitation(…);

    GroupMemoryBarrierWithGroupSync();
}
```

The granularity with which those barriers stall out outstanding memory operations is 4 bytes. Memory barriers are used to synchronize a whole group of threads. They are not an appropriate solution for synchronizing only a few threads in a thread group, which would require atomic instructions. Atomic instructions are not supported on DirectX 10.x–class hardware.

Example

The example program is a DirectX 10 port of the Julia 4D demo that Jan Vlietinck ported to DirectCompute with DirectX 11 functionality. There is a good explanation of the algorithm and the quarternion Julia set on Keenan Crane's website (see "References" at the end of this article). Figure 1 shows a screenshot of the DirectX 10 version of the demo.

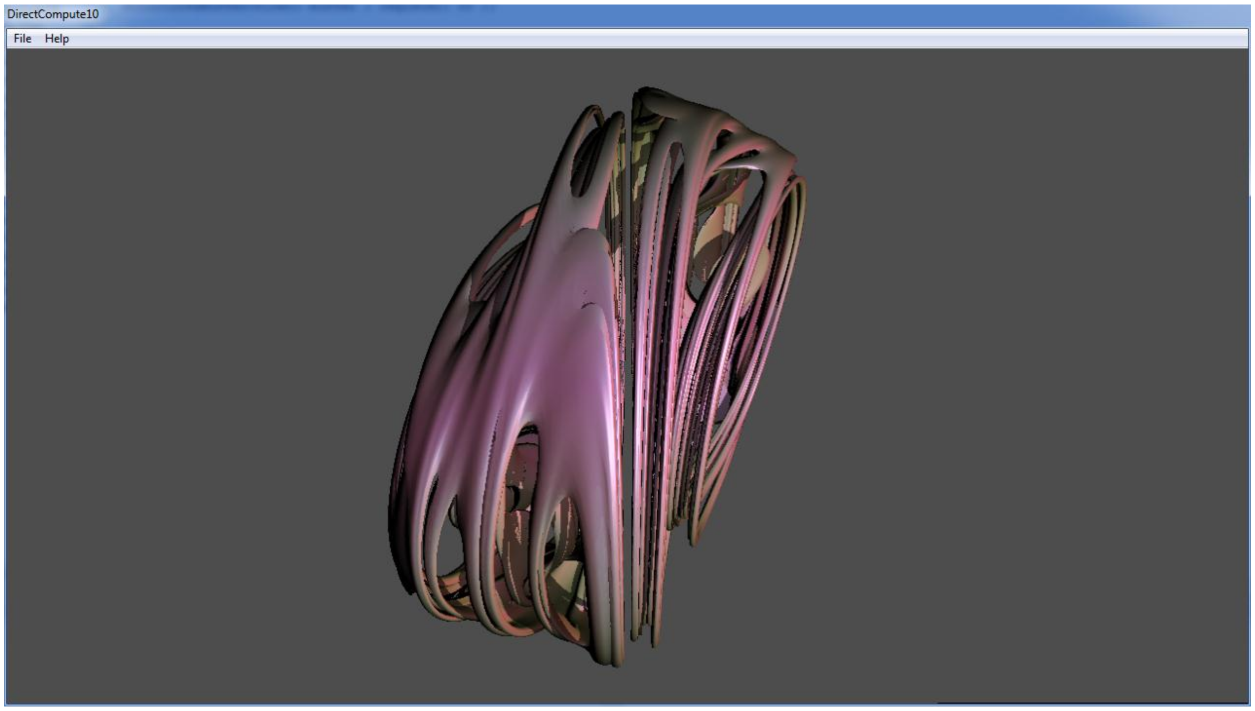

Figure 1. *Julia 4D*

Figure 2 shows another screenshot of the demo application running on the2nd generation Intel Core processor.

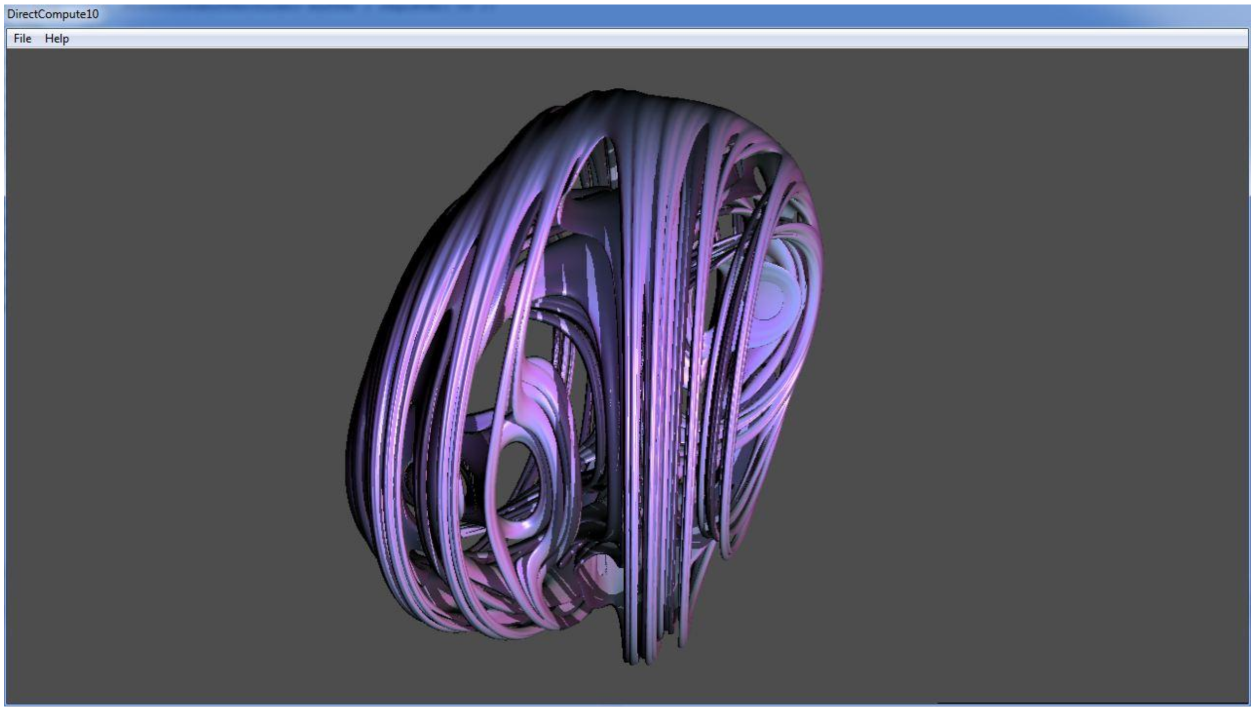

Figure 2. *Julia 4D on a 2nd generation Intel® Core™ processor*

This example is well suited to explain a DirectX 10 implementation because it uses the minimum number of API calls to set up a DirectCompute application for DirectX 10–class hardware and nicely shows the minimum requirements. In general, the example code is not written with production quality in mind to make it easier to read and more instructive. So, there's no checking all return statements, and picking the right device and window size can't be changed without re-compilation.

Setting Up the Device

The simplest way to set up a device is to call `D3D11CreateDeviceAndSwapChain()` with the default values. That means that whatever feature set the first device supports is exposed to the application. For a DirectX 10 application running with the DirectX 11 API, that means that while creating the device, you will discover whether the underlying hardware supports DirectX 10 or 11 and, depending on that support, this feature level will be chosen. If the application asks for DirectX 10 features, it will still run if the underlying hardware supports the DirectX 11 feature level but not vice versa. The simplest way to create a device and a swap chain is as follows:

```
   // return value -> what the hardware supports
   D3D_FEATURE_LEVEL MaxFeatureLevel = D3D_FEATURE_LEVEL_11_0;

   // we are asking for DirectX 10 support here
   D3D_FEATURE_LEVEL FeatureLevel = D3D_FEATURE_LEVEL_10_0;
   HRESULT hr = D3D11CreateDeviceAndSwapChain(
                                                NULL,
                                                D3D_DRIVER_TYPE_HARDWARE,
                                                NULL,
                                                D3D11_CREATE_DEVICE_DEBUG,
                                                &FeatureLevel,
                                                1,
                                                D3D11_SDK_VERSION,
                                                &sd,
                                                &pSwapChain,
                                                &pd3dDevice,
                                                &MaxFeatureLevel,
                                         &pImmediateContext);
```

The code asks whether the hardware supports at least the DirectX 10.0 feature level. If it doesn't, the return value shows an error.

With the swap chain created, a handle to the back buffer and a render target view to write into this back buffer can be retrieved. This back buffer is then set with the help of the handle as the main render target:

```
ID3D11RenderTargetView *pRenderTargetView;

// Create a back buffer render target, get a view on it to clear it later
ID3D11Texture2D *pBackBuffer;
pSwapChain->GetBuffer(0,__uuidof(ID3D11Texture2D),(LPVOID*)&pBackBuffer) ;
pd3dDevice->CreateRenderTargetView( (ID3D11Resource*)pBackBuffer, NULL, &pRenderTargetView );
pImmediateContext->OMSetRenderTargets( 1, &pRenderTargetView, NULL );
```

On DirectX 10.x–capable hardware, a compute shader can only write into a structured buffer. A pixel shader can then read this structured buffer and write its content into the back buffer. Therefore, an application needs to create a structured buffer, a UAV to write into it, and an SRV to read from inside a pixel shader later:

```
//
// structured buffer + shader resource view and unordered access view
//
struct BufferStruct
{
      float color;
};
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(BufferStruct);

sbDesc.ByteWidth = ((sbDesc.StructureByteStride * gWidth * gHeight  + 63) / 64) * 64;
sbDesc.Usage = D3D11_USAGE_DEFAULT;
pd3dDevice->CreateBuffer(&sbDesc, NULL, &pStructuredBuffer);

// UAV
D3D11_UNORDERED_ACCESS_VIEW_DESC sbUAVDesc;
ZeroMemory( &sbUAVDesc, sizeof(sbUAVDesc) );
```

```
sbUAVDesc.Buffer.FirstElement = 0;
sbUAVDesc.Buffer.Flags = 0;
sbUAVDesc.Buffer.NumElements = sbDesc.ByteWidth / sbDesc.StructureByteStride;
sbUAVDesc.Format = DXGI_FORMAT_UNKNOWN;
sbUAVDesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
if(pd3dDevice->CreateUnorderedAccessView((ID3D11Resource *)pStructuredBuffer, &sbUAVDesc,
&pComputeOutputUAV)!= S_OK)
       MessageBoxA(NULL, "UAV error", "Error", MB_OK | MB_ICONERROR);

// SRV on structured buffer
D3D11_SHADER_RESOURCE_VIEW_DESC sbSRVDesc;
ZeroMemory( &sbSRVDesc, sizeof( sbSRVDesc ) );
sbSRVDesc.Buffer.ElementOffset = 0;
sbSRVDesc.Buffer.ElementWidth = sbDesc.StructureByteStride;
sbSRVDesc.Buffer.FirstElement = sbUAVDesc.Buffer.FirstElement;
sbSRVDesc.Buffer.NumElements = sbUAVDesc.Buffer.NumElements;
sbSRVDesc.Format = DXGI_FORMAT_UNKNOWN;
sbSRVDesc.ViewDimension = D3D11_SRV_DIMENSION_BUFFER;
if(pd3dDevice->CreateShaderResourceView((ID3D11Resource *) pStructuredBuffer, &sbSRVDesc,
&pComputeShaderSRV)!= S_OK)
       MessageBoxA(NULL, "SRV error", "Error", MB_OK | MB_ICONERROR);
```

The structured buffer is built with the expectation that a UAV and an SRV will be attached to it. This is expressed in `BindFlags`, which uses a float value to store four 8-bit color values. Those four values will be packed in the shader into this float value and unpacked later when they are blit into the back buffer.

With the structured buffer and the necessary views in place, the actual compute shader, the vertex shader, and the pixel shader are compiled. The vertex and pixel shader combo will blit the content of the structured buffer into the back buffer:

```
//
// compile the compute, vertex and pixel shader
//
// compute shader
if(D3DX11CompileFromFile(L"qjulia4D.hlsl", NULL, NULL, "CS_QJulia4D", "cs_4_0", 0, 0, NULL,
&pByteCodeBlob, &pErrorBlob, NULL)!= S_OK)
   MessageBoxA(…);

if(pd3dDevice->CreateComputeShader(pByteCodeBlob->GetBufferPointer(), pByteCodeBlob-
>GetBufferSize(), NULL, &pCompiledComputeShader)!= S_OK)
   MessageBoxA(…);

// pixel shader
if(D3DX11CompileFromFile(L"blitStructuredBuffer.hlsl", NULL, NULL, "PSBlit", "ps_4_0", 0, 0,
NULL, &pByteCodeBlob, &pErrorBlob, NULL)!= S_OK)
   MessageBoxA(…);
if(pd3dDevice->CreatePixelShader(pByteCodeBlob->GetBufferPointer(), pByteCodeBlob-
>GetBufferSize(), NULL, &pCompiledPixelShader)!= S_OK)
   MessageBoxA(…);

// vertex shader
if(D3DX11CompileFromFile(L"blitStructuredBuffer.hlsl", NULL, NULL, "VSBlit", "vs_4_0", 0, 0,
NULL, &pByteCodeBlob, &pErrorBlob, NULL)!= S_OK)
   MessageBoxA(…);
```

```
if(pd3dDevice->CreateVertexShader(pByteCodeBlob->GetBufferPointer(), pByteCodeBlob-
>GetBufferSize(), NULL, &pCompiledVertexShader)!= S_OK)
   MessageBoxA(…);
```

The application code that runs the compute shader and later blits the content of the structured buffer into the back buffer is rather compact:

```
// Set compute shader
pImmediateContext->CSSetShader(pCompiledComputeShader, NULL, 0 );

// For CS output
pImmediateContext->CSSetUnorderedAccessViews(0, 1, &pComputeOutputUAV, NULL);

// For CS constant buffer
pImmediateContext->CSSetConstantBuffers(0, 1, &pcbFractal );

// Run the CS
pImmediateContext->Dispatch(gWidth / THREADSX, gHeight / THREADSY, 1 );

// D3D11 on D3D10 hW: only a single UAV can be bound to a pipeline at once.
// set to NULL to unbind
ID3D11UnorderedAccessView* pNullUAV = NULL;
pImmediateContext->CSSetUnorderedAccessViews(0, 1, &pNullUAV, NULL);

// set the vertex shader
pImmediateContext->VSSetShader( pCompiledVertexShader, NULL, 0 );

// set the pixel shader
pImmediateContext->PSSetShader( pCompiledPixelShader, NULL, 0 );

// to read the structured buffer a shader resource view is set here
pImmediateContext->PSSetShaderResources( 0, 1, &pComputeShaderSRV );

// set primitive topology
pImmediateContext->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

// draw the results on the screen
pImmediateContext->Draw(3, 0 );

// unbind the compute shader buffer resource
ID3D11ShaderResourceView* pNULLSRV[ 1 ] = { NULL };
pImmediateContext->PSSetShaderResources( 0, 1, pNULLSRV );

// make it visible
pSwapChain->Present( 0, 0 );
```

Before the `Dispatch()` call that invokes the compute shader execution, the compute shader is set, a UAV is set to write into the structured buffer, and a constant buffer is set that holds data for the Julia 4D algorithm. Because DirectX 10.x hardware can only have one UAV set at a time, the available UAV is then set to NULL.

The following draw call in `Draw()` blits the content of the structured buffer into the back buffer. Therefore, it sets the SRV to read the structured buffer and, after the draw call, sets it back to NULL. The back buffer was already set up as the main render target in the initialization code.

Two shaders are involved: the compute shader that writes the Julia 4D data set into the structured buffer and the blit shader that blits the content of this buffer into the back buffer. The compute shader writes into the buffer by packing four color values into a float variable with the following code:

```
float Pack4PNForFP32(float4 channel)
{
	// layout of a 32-bit fp register
	// SEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMM
	// 1 sign bit; 8 bits for the exponent and 23 bits for the mantissa
	uint uValue = 0;

	// just make sure everything is between 0..1
	channel = saturate(channel);

	// pack x
	uValue = ((uint)(channel.x * 255.0 + 0.5));

	// pack y
	uValue |= ((uint)(channel.y * 255.0 + 0.5)) << 8;

	// pack z in EMMMMMMM
	uValue |= ((uint)(channel.z * 255.0 + 0.5)) << 16;

	// pack w in SEEEEEEE
	// the last E will never be 1b because the upper value is 254
	// max value is 11111110 == 254
	// this prevents the bits of the exponents to become all 1
	// range is 1.. 254
	// to prevent an exponent that is 0 we add 1.0
	uValue |= ((uint)(channel.w * 253.0 + 1.5)) << 24;

 return asfloat(uValue);
}

…
  uint stride = c_width;

  // buffer stride, assumes data stride = data width (i.e. no padding)
  uint idx = (DTid.x + (c_width / 2)) + (DTid.y) * stride;
  output[idx].color = Pack4PNForFP32((color));
}
```

The vertex and pixel shader then blit the content of the buffer by unpacking it into the 8:8:8:8 back buffer:

```
StructuredBuffer<float> buffer : register( t0 );

struct PsIn
{
    float4 Pos : SV_POSITION;
};

#define WINDOWWIDTH 800
#define WINDOWHEIGHT 640
```

```
PsIn VSBlit(uint VertexID: SV_VertexID)
{
       PsIn Out = (PsIn)0;

       // Produce a fullscreen triangle with a triangle list
       float4 position;
       position.x = (VertexID == 2)?  3.0 : -1.0;
       position.y = (VertexID == 0)? -3.0 :  1.0;
       position.zw = 1.0;

       Out.Pos = position;

       return Out;
}

// unpack four positive normalized values from a 32-bit float
float4 Unpack4PNFromFP32(float fFloatFromFP32)
{
 float r, g, b, d;
 uint uValue;

 uint uInputFloat = asuint(fFloatFromFP32);

 // unpack a
   r = ((uInputFloat) & 0xFF) / 255.0;

 g = ((uInputFloat >> 8) & 0xFF) / 255.0;

 b = ((uInputFloat >> 16) & 0xFF) / 255.0;

 // extract the 1..254 value range and subtract 1
 // ending up with 0..253
 d = (((uInputFloat >> 24) & 0xFF) - 1.0) / 253.0;

 return float4(r, g, b, d);
}


float4 PSBlit(PsIn In) : SV_TARGET
{
   uint idx = ((In.Pos.x)) + ((In.Pos.y)) * WINDOWWIDTH;

   return Unpack4PNFromFP32(buffer[idx]);
}
```

The vertex shader generates position values by using the vertex counter with the semantic `SV_VertexID`. When `SV_Position` values are used in the pixel shader, they describe the pixel location. Packing four 8-bit values into a 32-bit structured buffer instead of using a 128-bit structured buffer is a more efficient use of memory. Alpha doesn't need to be packed and unpacked. The fourth component is there only to show a complete packing and unpacking routine. You might also consider using a buffer consisting of integer values to simplify the packing routine even further.

Note: The blit shader combo doesn't use a vertex or index buffer.

Summary

DirectCompute allows you to program compute shaders on hardware running on 2nd generation Intel Core processors. Operations that need a more relaxed relationship between threads and data or operations that do not require the rasterizer can therefore be brought over to the graphics hardware. Doing so allows you to balance the load between the CPU and the graphics processor with a fine level of granularity.

References

- Keenan Crane, http://users.cms.caltech.edu/~keenan/project_qjulia.html
- Registers used in cs_5_0 http://msdn.microsoft.com/en-us/library/hh447206(v=VS.85).aspx
- Jan Vlietinck, http://users.skynet.be/fquake
- Jason Zink, Matt Pettineo, Jack Hoxley, "Practical Rendering & Computing with Direct3D 11," CRC Press, 2011; p. 305

About the Author

Wolfgang Engel is the CTO/CEO and Co-founder of Confetti Special Effects Inc., a think tank for advanced real-time graphics research for the video game and movie industry. Previously, he worked for more than four years in Rockstar's core technology group as the lead graphics programmer. Some of his game credits can be found at http://www.mobygames.com/developer/sheet/view/developerId,158706. He is the editor of the *ShaderX Pro* and *GPU Pro* books as well as the author of several other books, and he speaks on graphics programming at conferences worldwide. He has been a DirectX MVP since July 2006 and active in several advisory boards in the industry. He teaches the class "GPU Programming" at the University of California, San Diego. You can find him on Twitter at @wolfgangengel. Confetti's website is www.conffx.com.