Microsoft* DirectCompute on Intel® Ivy Bridge Processor Graphics

By Wolfgang Engel

Microsoft* DirectCompute exposes the compute functionality of graphics hardware as a new shader type: the compute shader. A compute shader is similar to a vertex, geometry, or pixel shader and offers a programming interface that makes the massively parallel computation power of graphics hardware available to tasks outside of the normal raster-based graphics pipeline exposed with Microsoft* Direct3D* or OpenGL*.

Note: Although DirectCompute was introduced with Microsoft* DirectX* 11, it is possible to run a compute shader on Microsoft* DirectX* 10-, 10.1-, and 11-class hardware.

DirectCompute has several advantages over application programming interfaces (APIs) that also offer a compute solution. First, it integrates well with Direct3D*, which means that it has not only efficient interoperability with the Direct3D* resources like textures and buffers, but many of the concepts and the language syntax are already well known to Direct3D* programmers. In addition, the interface is more generalized and simplified when compared to other computing APIs. Like Direct3D*, DirectCompute guarantees consistent results across different hardware.

DirectCompute Applications

DirectCompute does not have a fixed mapping between the data it is processing and the threads doing the processing, like the vertex or pixel shader. One thread can process one or many data elements, and the application can control directly how many threads are used to perform the computation.

Applications for DirectCompute are algorithms that do not map into the fixed mapping between data and processing or threads, which the vertex or pixel shader requires and which do not require involving the rasterizer. Typical use cases include:

- Game physics and artificial intelligence;
- Applying algorithms in image space that require kernels that are more flexible when it comes to the relationship between data used and the threads applied on this data; and
- The many advanced rendering effects that use the Read/Write abilities, such as order-independent transparency, ray tracing, and global illumination effects.

DirectCompute Memory Model

Graphics hardware is expected to have different memory types, each of which favors certain access patterns. In DirectCompute, you can differentiate between *register-based memory, device memory,* and *group shared memory.*

Register-based Memory

DirectCompute uses the same set of registers as the other programmable stages of the Direct3D* pipeline (*DirectCompute registers*). Because you can only program shaders in the High-level Shader Language (HLSL), those registers are not accessible to you. Nevertheless, the compiler can generate intermediate assembly. It is difficult to tell how much value the assembly code has, because it is only intermediate and might be changed by the driver of the underlying hardware.

The most interesting register-based memory is the temporary registers. Overall, 4096 temporary and indexed temporary registers—which are array like—are available. The regular temporary registers are named *r#,* while the indexed temporary array registers are named *x#[n].* Those registers might be shared among all threads in flight in a processing core or among several processing cores. The driver compiler selects and allocates these registers automatically, without any direct influence from the shader programmer. There is a strong chance that more complex shaders will consume more temporary registers, thereby "starving" the hardware of temporary registers. A good graphics hardware profiling tool will flag the fact that the hardware is running out of temporary registers. You can then attempt to rewrite parts of the shader to reduce the number of temporary registers used.

Device Memory

Although data in temporary registers only persists during execution of a shader program, you also need data that is kept over a longer period of time and offers more storage. DirectCompute can store data in generic Direct3D* resources such as buffers and textures. Read/Write buffers and textures, structured buffers, and byte address (raw) buffers are available. To write into and read from textures and buffers, so-called *memory views* are available. DirectX* 11 introduced the Unordered Access view (UAV), which allows scattered writes and gathered reads. To read memory in a shader, DirectX* 10 introduced a Shader Resource view (SRV). A special category of device memory is the *constant buffer,* which favors access patterns were 16 consecutive memory reads are done. An example for this access pattern is reading a 4×4 matrix from constant memory.

Read/Write Buffers and Textures

Read-only texture memory is supposed to favor memory access patterns that are spatially close to each other. For example, *bilinear filtering* requires access to texels that are close to each other. DirectX* 11 added a new set of Read and Write textures:

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

Here's an example of how to define an `RWTexture2D` texture in a shader:

```
// Compute Shader code: RWTexture with Unordered Access View in u0
```

```
RWTexture2D<float4> output : register (u0);
```

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[4];
};

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 * Width * Height; 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 cast to unsigned `int`. Raw buffers are useful for generating geometry with DirectCompute, because they can be bound as vertex and index buffers. In HLSL, they are declared as follows:

```
ByteAddressBuffer
RWByteAddressBuffer
```

DirectX* 11 aligns raw buffers to 16 bit.

Constant Buffers

Constant buffers provide read-only access to data that is expected to be accessed as 16 consecutive float values. As long as they are accessed in order, the cost is similar to reading only one value.

A shader has access to 4096 32-bit, four-component constants: 64 KB. Although DirectX* 10.x and DirectX* 11 define this as the upper limit for the size of a constant buffer, DirectX* 11.1 allows you to store many more constants in the constant buffer and to access a subrange of this buffer in a shader:

```
// Create constant buffer
typedef struct
{
	float diffuse[4]; // diffuse shading color
	float mu[4];    // quaternion julia parameter
	float epsilon;  // detail julia
	int c_width;      // view port size
	int c_height;
	int selfShadow;  // selfshadowing on or off
	float orientation[4*4]; // rotation matrix
	float zoom;
} QJulia4DConstants;

D3D11_BUFFER_DESC Desc;
	Desc.Usage = D3D11_USAGE_DYNAMIC;
	Desc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
	Desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
	Desc.MiscFlags = 0;

	// must be multiple of 16 bytes
Desc.ByteWidth = ((sizeof( QJulia4DConstants ) + 15)/16)*16;
pd3dDevice->CreateBuffer(&Desc, NULL, &pcbFractal);
```

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. DirectX* 11 can bind eight UAVs at the same time.

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);
```

In DirectX* 11.x, a UAV also allows a new access pattern called *append and consume.* This pattern allows for building and accessing data in list or stack form. An append and consume buffer is a structured or a raw buffer with a specially created UAV:

```
// Unordered access view on structured buffer
D3D11_UNORDERED_ACCESS_VIEW_DESC sbUAVDesc;
ZeroMemory( &sbUAVDesc, sizeof(sbUAVDesc) ); sbUAVDesc.Buffer.FirstElement = 0;
sbUAVDesc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_APPEND;
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);
```

In HLSL, the `AppendStructuredBuffer<T>` provides the `append(T)` method;
the `ConsumeStructuredBuffer<T>` provides the `T .consume()` method.

Thread Group Shared Memory

Thread group shared memory (TGSM) is located in on-chip memory. You can consider it a cache to minimize off-chip bandwidth use. All the threads in a thread group access this memory. 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—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, 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];
```

DirectCompute Threading Model

The typical multithreading 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 Intel® Ivy Bridge 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 in parallel.*

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 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.

That means that you can split the typical data for a DirectCompute application into small enough parts that it can be processed in separation. 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.* Two functions in DirectX* 11.x dispatch a kernel:

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

DispatchIndirect(ID3D11Buffer *pBufferForArgs, UINT AlignedOffsetForArgs);
```

The first 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 second method—`DispatchIndirect()`—indirects the dispatch call by allowing it to fill a buffer with the parameters that the `Dispatch()` call expects, and then dispatch a new job based on the buffer data. 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.

DirectX* 11.x supports 3D groups and threads. Think of the threads in a thread group in DirectX* 11 as a 3D array. Each thread “array” is part of a thread group as a 2D or 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 in the example above execute 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 into data, and then reading data from 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` - 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` - 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` - 3D version of SV_GroupIndex - 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` - index of a thread within a thread group

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 multithreaded 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* and *atomic functions* are available.

Memory Barriers

In DirectX* 11.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 such a barrier to ensure that when threads share data with one another in a device or TGSM, the desired values written to the memory have had a chance to be written before being read by other threads. In other words, 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 `*MemoryBarrier`s 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. 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. This is where atomic functions come in handy.

Atomic Functions

DirectX* 11.x supports atomic functions, or *interlocked functions,* in the compute and pixel shaders. They are guaranteed to operate atomically—in other words, they are guaranteed to occur in the order programmed. Here is a list of the atomic functions:

- `InterlockedAdd`
- `InterlockedMin`
- `InterlockedMax`
- `InterlockedOr`
- `InterlockedAnd`
- `InterlockedXor`
- `InterlockedCompareStore`
- `InterlockedCompareExchange`
- `InterlockedExchange`

With the exception of `InterlockedExchange`, all functions accept only input values that are integer or unsigned integer values in TGSM. For example, if the compute shader wants to keep a count of the number of threads that encounter a particular value, `InterlockedAdd()` can be called. `InterlockedCompareExchange()` compares the value of a destination to a reference value; if the two match, the third argument is written to the destination (`Zink`).

Note that all of those integer atomics except `Add`, `Min`, and `Max` work as-is on floating-point numbers if they are passed in `asInteger()`. However, `Min` and `Max` work as-is on `asInteger()` floats as long as all floats are positive.

Example

The example program shows the Mandelbrot set. (There are good explanations of the algorithm on Wikipedia.) Jan Vlietinck's website and other websites show implementations with source code. Figure 1 shows a screenshot of the example program.

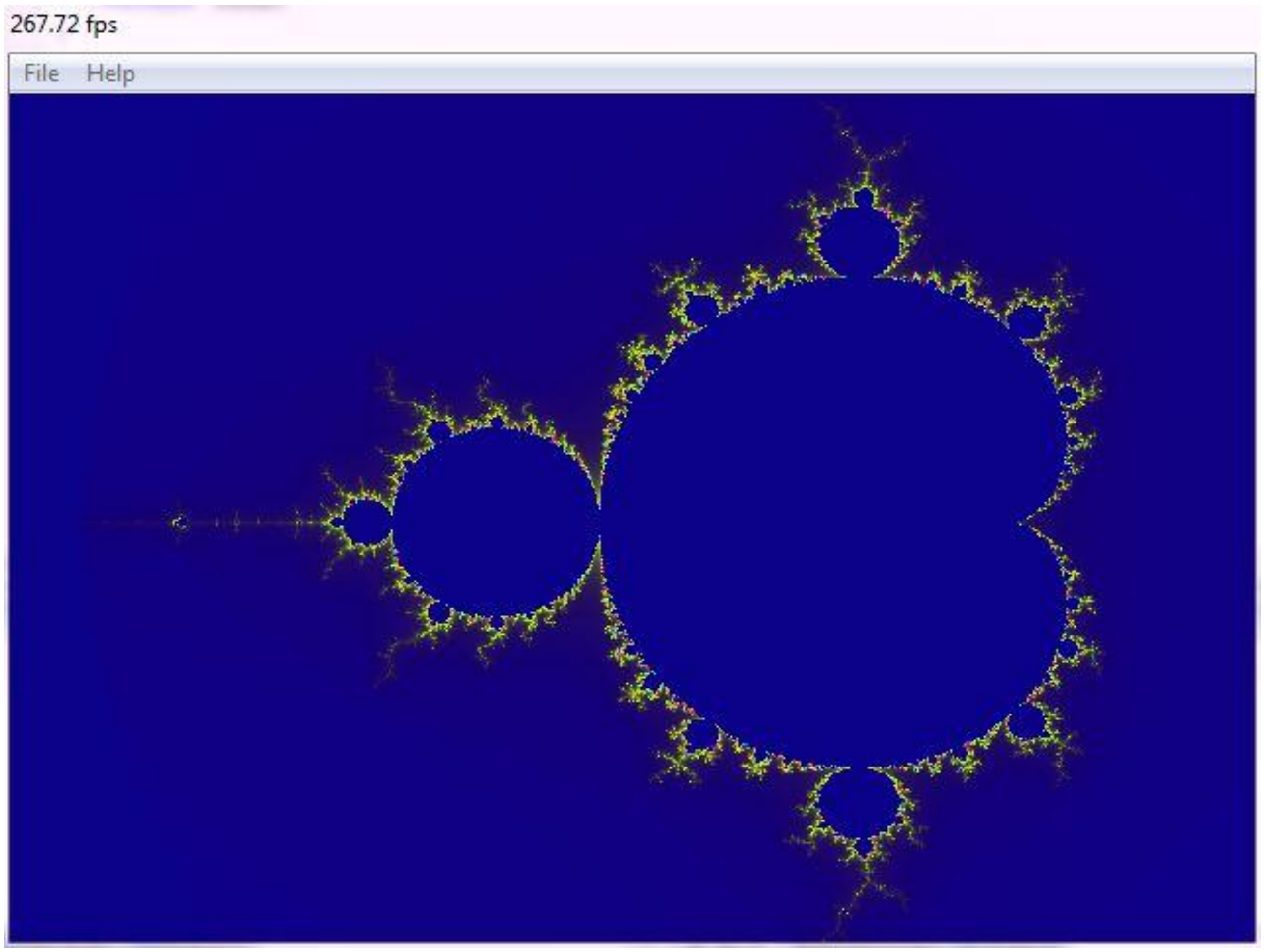

Figure 1. Mandelbrot

Figure 2 shows another screenshot of the demo application running on Intel® Ivy Bridge.

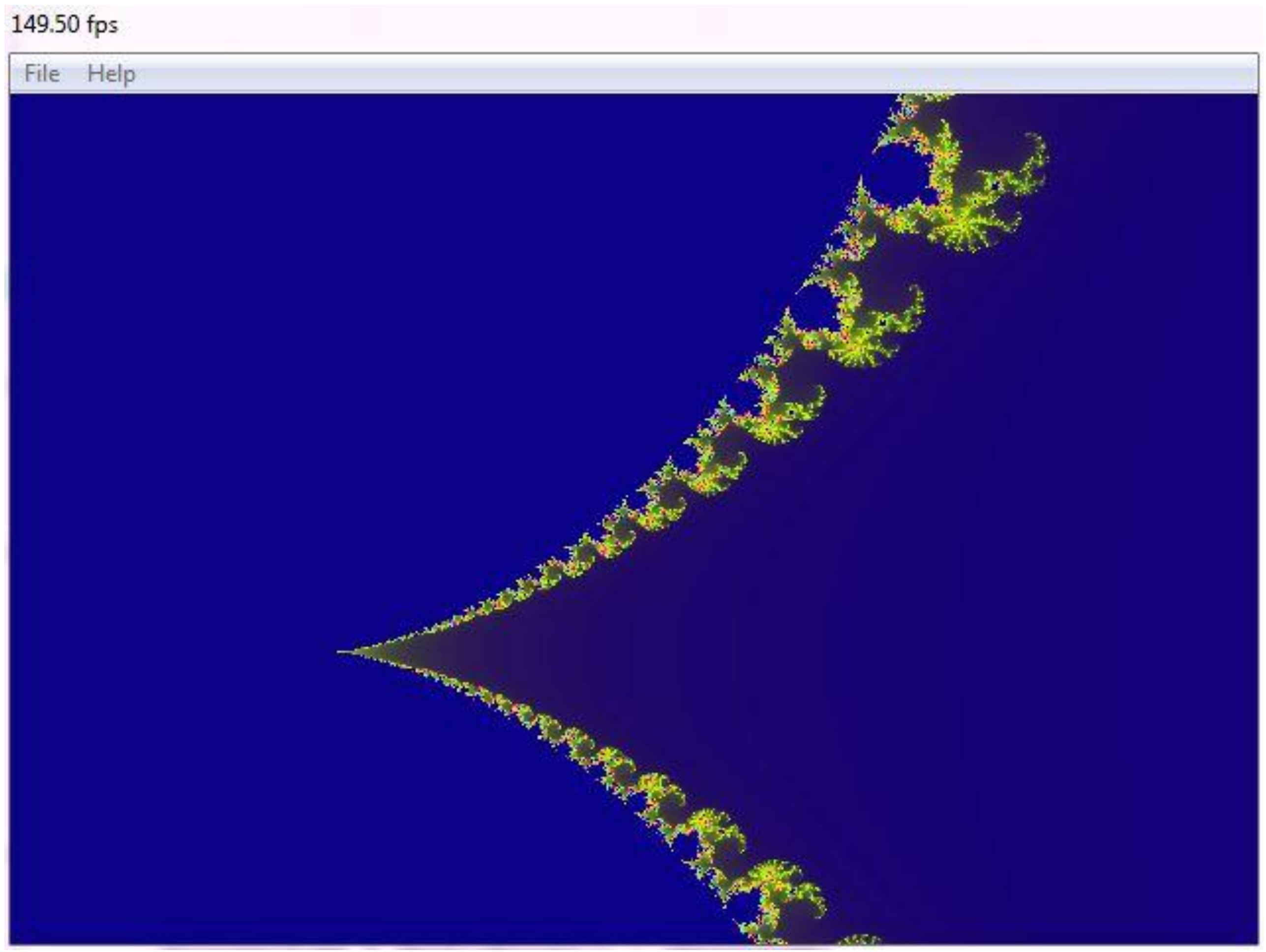

Figure 2. Mandelbrot

This example is well suited to explaining a DirectX* 11 implementation, because it uses the minimum number of API calls to set up a DirectCompute application for DirectX* 11 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 the return statements and spending some time on picking the right device, and the window size can't be changed without re-compilation. Let's go through the implementation following the order of the DirectX*11 calls.

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* 11 application running with the DirectX* 11 feature level, that means that while creating the device, it needs to be verified that the underlying hardware supports the DirectX* 11 feature level.

```
   //
   // Initialize Direct3D device and swap chain
   //
   DXGI_SWAP_CHAIN_DESC sd;
// …
   // this qualifies the back buffer for being the target of compute shader writes
   sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT | DXGI_USAGE_UNORDERED_ACCESS |
DXGI_USAGE_SHADER_INPUT;
// …

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

   // we are asking for DirectX 11 feature level support here
   D3D_FEATURE_LEVEL FeatureLevel = D3D_FEATURE_LEVEL_11_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* 11.0 feature level. If it doesn't, the return value shows an error.

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

```
DXGI_SWAP_CHAIN_DESC sdtemp;
pSwapChain->GetDesc(&sdtemp);

// get access to the back buffer via a texture
ID3D11Texture2D* pTexture;
pSwapChain->GetBuffer(0, __uuidof( ID3D11Texture2D ), ( LPVOID* )&pTexture );

// create shader unordered access view on back buffer for compute shader to write into
texture
pd3dDevice->CreateUnorderedAccessView(pTexture, NULL, &pComputeOutput );
```

DirectX*11-capable hardware supports `RWTexture` and allows you to use a UAV into this texture. The code above retrieves the back buffer from the swap chain and creates a UAV that points to the buffer. This way, writing into a structured buffer and reading this buffer later are not necessary.

Constant Memory

The only buffer that needs to be allocated for this example is a constant buffer. To make use of the read optimizations that this buffer offers, the buffer needs to be aligned to 16 bytes:

```
//
// Create constant buffer
//
D3D11_BUFFER_DESC Desc;
Desc.Usage = D3D11_USAGE_DYNAMIC;
Desc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
Desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
Desc.MiscFlags = 0;
Desc.ByteWidth = ((sizeof( MandelbrotConstants ) + 15)/16)*16; // must be multiple of 16
bytes

pd3dDevice->CreateBuffer(&Desc, NULL, &pcbFractal);
```

Later in the `Render()` functions, this constant buffer is filled by mapping the system memory copy to GPU memory:

```
// Fill constant buffer
D3D11_MAPPED_SUBRESOURCE msr;
pImmediateContext->Map(pcbFractal, 0, D3D11_MAP_WRITE_DISCARD, 0,  &msr);
 *(MandelbrotConstants *)msr.pData = MandelC;
pImmediateContext->Unmap(pcbFractal,0);
```

Compiling the Compute Shader Kernel

A compute shader is compiled like any other shader in DirectX*11:

```
//
// compile the compute shader
//
if(D3DX11CompileFromFile(L"Mandelbrot.hlsl", NULL, NULL, "Mandelbrot", "cs_5_0", 0,
                           0, NULL, &pByteCodeBlob, &pErrorBlob, NULL)!= S_OK)
  MessageBoxA(NULL, (char *)pErrorBlob->GetBufferPointer(), "Error", MB_OK | MB_ICONERROR);
if(pd3dDevice->CreateComputeShader(pByteCodeBlob->GetBufferPointer(),
                    pByteCodeBlob->GetBufferSize(), NULL, &pCompiledComputeShader)!= S_OK)
  MessageBoxA(NULL, "CreateComputerShader() failed", "Error", MB_OK | MB_ICONERROR);
```

The main difference is the specification of the `cs_5_0` target.

Dispatching the Kernel

The application code that runs the compute shader is rather compact:

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

```
// UAV for CS output into back-buffer
pImmediateContext->CSSetUnorderedAccessViews(0, 1, &pComputeOutput, NULL);

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

// Run the CS
pImmediateContext->Dispatch((gWidth) / 16, (gHeight) / 16, 1 );

// 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 back buffer, and a constant buffer is set that holds data for the Mandelbrot algorithm.

Thread Addressing System

The `Dispatch()` call shown above creates a grid that consists of the windows width and height, each divided by a thread group of 16 threads. In other words, with a 640×480 window, the grid consists of 40 thread groups in the *x* and 30 thread groups in the *y* direction, while the *z* direction is one. Because the compute shader directly writes into the back buffer, you might think of this as tiled-based rendering, where each tile is rendered with a group of threads consisting of 256 threads.

The compute shader then defines with the `numthreads` keyword the thread group size of 16×16×1.

```
[numthreads(16, 16, 1)]
void Mandelbrot( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid :
SV_GroupThreadID, uint GI : SV_GroupIndex )
{
…
        output[ DTid.xy ] = color;
}
```

The `SV_DispatchThreadID` system value is used to address the right thread. This runtime-generated value provides the index into a thread within the entire dispatch. In the case of the Mandelbrot compute shader, the `output` variable outputs to the 2D texture that is the back buffer. It is indexed so that all tiles can be filled simultaneously by different instances of the compute shader.

Summary

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

References

- Wikipedia on the Mandelbrot set, http://en.wikipedia.org/wiki/Mandelbrot_set
- 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 & Co-Founder of Confetti*, a think-tank for advanced real-time graphics research for the video game and movie industry. Previously, he worked for more than 4 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* and *GPU Pro* books, the author of several other books, and speaks on graphics programming on conferences worldwide. He has been a DirectX* MVP since July 2006 and active in several advisory boards in the industry. He also teaches the class “GPU Programming” at University of California, San Diego. You can find him on Twitter at @wolfgangengel.