**Intel Extension Framework DirectX 12 64-bit Typed Atomics**

**Introduction**

Intel offers a subset of the HLSL Shader Model 6.6 64-bit Typed Atomic support through the Intel Extension Framework on our latest Alchemist A-Series GPUs. This paper will detail how to use the Intel Extension Framework for both DirectX 12 API Extensions as well as how to use the extension framework to use a shader (HLSL) extension. Initializing the Extension Framework is an easy task.

**Limitations**

It is important to understand that there are limitations to the 64-bit typed atomic support on DG2/Alchemist. Because of these limitations we are exposing what we can support via an extension until we are able to more fully support Shader Model 6.6.

* D3D12\_RESOURCE\_DIMENSION\_TEXTURE2D only
* No mip-map surfaces, no 3D surfaces or 2D arrays
* We are not supporting subregions of a surface

**The Sample**

We wrote a basic sample to demonstrate the instantiation and use of the extension in an easy to follow project. First import and link the extension framework igdex64.lib along with its dependencies: shlwapi.lib, setupapi.lib, and cfgmgr32.lib to your project. Next, add the igdext.h and IntelExtensions12.hlsl header files to the project. Once this is completed the framework can be initialized during application startup.

**Loading The Extension Framework**

First load the Intel Extension Library using INTC\_LoadExtensionsLibrary function. We can pass true to have it look for the igdext64.dll in the working directory, or false to look in the driver installation directory.  
  
INTC\_LoadExtensionsLibrary(true);

Next, query the supported extension framework version to determine what level of support can be expected with the currently library and hardware configuration. This function first needs to be called to get the supported version count and then called a second time to populate an INTCExtensionVersion array. The first call return value is used to allocate an array for the second call to populate with version details as can be seen in the same code.

//first, fill in the supportedExtVersionCount. This does not populate pSupportedExtVersions  
INTC\_D3D12\_GetSupportedVersions(m\_device.Get(), nullptr, &supportedExtVersionCount);

//allocate space for pSupportedExtVersions here using supportedExtVersionCount

//Next, use returned value for supportedExtVersionCount to allocate space for the supported extensions  
pSupportedExtVersions = new INTCExtensionVersion[supportedExtVersionCount]{};  
INTC\_D3D12\_GetSupportedVersions(m\_device.Get(), pSupportedExtVersions, &supportedExtVersionCount);

Finally, create the device extension context using the current device. The m\_pINTCExtensionContext object will be used to call the Extension API functions.

INTC\_D3D12\_CreateDeviceExtensionContext(m\_device.Get(), &m\_pINTCExtensionContext, &intcExtensionInfo, nullptr);

It is important to note that the examples above need error handling code, please refer to the project for initialization source code.

**Intel Extension 64-Bit Committed Resource**

To perform 64-bit typed atomic operations a UAV needs created for the surface we intend to apply the atomic operations. Create a resource descriptor using the INTC\_D3D12\_RESOURCE\_DESC\_0001 type. Also, we need to define a resource with a DXGI\_FORMAT\_R32G32\_UINT format for the UAV.

D3D12\_RESOURCE\_DESC texture2D = {};

texture2D.Dimension = D3D12\_RESOURCE\_DIMENSION\_TEXTURE2D;

texture2D.Alignment = 0;

texture2D.Width = TEX\_WIDTH;

texture2D.Height = TEX\_HEIGHT;

texture2D.DepthOrArraySize = 1;

texture2D.MipLevels = 1;

texture2D.Format = DXGI\_FORMAT\_R32G32\_UINT;

texture2D.SampleDesc.Count = 1;

texture2D.SampleDesc.Quality = 0;

texture2D.Layout = D3D12\_TEXTURE\_LAYOUT\_UNKNOWN;

texture2D.Flags = D3D12\_RESOURCE\_FLAG\_ALLOW\_UNORDERED\_ACCESS;

Next, use the Extension Framework API create an INTC\_D3D12\_RESOURCE\_DESC\_0001 with EmulatedTyped64bitAtomics set to TRUE.

INTC\_D3D12\_RESOURCE\_DESC\_0001 dsResDesc1;

texture2D.Flags = D3D12\_RESOURCE\_FLAG\_ALLOW\_UNORDERED\_ACCESS;

dsResDesc1.pD3D12Desc = &texture2D;

dsResDesc1.Texture2DArrayMipPack = FALSE;

dsResDesc1.EmulatedTyped64bitAtomics = TRUE;

Use the INTC\_D3D12\_CreateCommittedResource API to create the resource with D3D12\_HEAP\_FLAGS::D3D12\_HEAP\_FLAG\_ALLOW\_SHADER\_ATOMICS flag set.

INTC\_D3D12\_CreateCommittedResource(

m\_pINTCExtensionContext,

&heap\_props,

D3D12\_HEAP\_FLAGS::D3D12\_HEAP\_FLAG\_ALLOW\_SHADER\_ATOMICS,

&dsResDesc1,

D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS,

nullptr,

IID\_PPV\_ARGS(&m\_computeBuffer));

**64-bit Typed Atomic Root Signature**

Using the Intel Extension requires us to populate our root signature slightly differently since the compiler leverages a UAV slot to be aware an extension will be used.

CD3DX12\_DESCRIPTOR\_RANGE1 descRange[2] = {};

// UAV Slot for the 64-bit Typed Atomic UAV

descRange[0].Init(D3D12\_DESCRIPTOR\_RANGE\_TYPE\_UAV, 1, 0, 0, D3D12\_DESCRIPTOR\_RANGE\_FLAG\_DATA\_VOLATILE, 0);

A second UAV slot is required for extension support. This null descriptor tells the compiler an extension is going to be used. In this case we use slot u7 which is the default slot used for HLSL extensions.

// UAV Slot for Intel Extension Framework

descRange[1].Init(D3D12\_DESCRIPTOR\_RANGE\_TYPE\_UAV, 1, 7, 0, D3D12\_DESCRIPTOR\_RANGE\_FLAG\_DATA\_VOLATILE, 0);

CD3DX12\_ROOT\_PARAMETER1 rootParameters[1] = {};

rootParameters[0].InitAsDescriptorTable(\_countof(descRange), descRange, D3D12\_SHADER\_VISIBILITY\_ALL);

**Creating a 64bit Typed Unordered Access View**

Finally, to finish setup, create a UAV with a format of DXGI\_FORMAT\_R32G32\_UINT and call CreateUnorderedAccessView to create the UAV.

// Create a UAV resource to write to.

CD3DX12\_CPU\_DESCRIPTOR\_HANDLE uavHandle(m\_computeUAVHeap->GetCPUDescriptorHandleForHeapStart());

//texture2D

D3D12\_UNORDERED\_ACCESS\_VIEW\_DESC uavDesc2 = {};

uavDesc2.Format = DXGI\_FORMAT\_R32G32\_UINT;

uavDesc2.ViewDimension = D3D12\_UAV\_DIMENSION\_TEXTURE2D;

uavDesc2.Texture2D.MipSlice = 0;

uavDesc2.Texture2D.PlaneSlice = 0;

m\_device->CreateUnorderedAccessView(m\_computeBuffer.Get(), nullptr, &uavDesc2, uavHandle);

**Passing a 64-bit UAV To The Compute Shader**

Once the 64-bit Typed UAV is created we can pass it into a compute shader we can execute it like a normal compute shader by passing in the 64-bit Typed UAV and calling Dispatch.

ID3D12DescriptorHeap\* ppHeapsCompute[] = { m\_computeUAVHeap.Get() };

m\_commandList->SetDescriptorHeaps(\_countof(ppHeapsCompute), ppHeapsCompute);

m\_commandList->SetPipelineState(m\_computePipelineState.Get());

m\_commandList->SetComputeRootSignature(m\_computeRootSignature.Get());

m\_commandList->SetComputeRootDescriptorTable(0, m\_computeUAVHeap->GetGPUDescriptorHandleForHeapStart());

m\_commandList->Dispatch(TEX\_WIDTH / 32, TEX\_HEIGHT / 32, 1);

**Intel Shader Extensions**

Now that the API side of the Extension Framework steps have been completed and the program can dispatch calls to a compute shader the extension framework can be used to call atomic instructions from HLSL.

Tell the Extension Framework to use UAV register u7 to match the root parameter setup using the following macro

#define INTEL\_SHADER\_EXT\_UAV\_SLOT u7

Next import the IntelExtension12.hlsl include file

#include "IntelExtensions12.hlsl"

Now initialize u0 with a RWTexture2D for the 64-bit Typed Atomic UAV that was initialized previously on the CPU.

RWTexture2D<uint2> typedUAV64Bit : register(u0);

In the function the buffer can be used to store values but must be stored as a two part 32-bit integer vector (uint2).

// Store Low Bytes in first position and High bytes in second position.

typedUAV64Bit[DTid.xy] = uint2(0x00000009, 0x00000000); // 9

uint2 value2 = uint2(0x00000001, 0x00000000); // 1

The Extension Framework does need to be initialized in the shader before using any atomic functions by calling:

IntelExt\_Init();

Finally, the IntelExt\_InterlockedMaxUint64 function can be called to execute the atomic operation on the typed 64-bit UAV buffer.

IntelExt\_InterlockedMaxUint64(typedUAV64Bit, DTid.xy, value2);

The IntelExt\_InterlockedMaxUint64 function takes the typed 64-bit buffer as the first parameter, a 2D UV coordinate as the second parameter and a third value to compare to the value at the address in the typed UAV.

**Shutting Down The Extension Framework**

After the application is finished, the Extension framework context must be destroyed and the library must be unloaded. To Destroy the context call INTC\_DestroyDeviceExtensionContext with a pointer to the m\_pINTCExtensionContext extension framework context.

INTC\_DestroyDeviceExtensionContext(&m\_pINTCExtensionContext);

Once the extension framework context has been destroyed the library can be unloaded with call to INTC\_UnloadExtensionsLibrary.

INTC\_UnloadExtensionsLibrary();

At this point the application can be safely shutdown.