Shader Model 5.0 and Compute Shader

Nick Thibieroz, AMD
DX11 Basics

» New API from Microsoft
» Will be released alongside Windows 7
  » Runs on Vista as well
» Supports downlevel hardware
  » DX9, DX10, DX11-class HW supported
  » Exposed features depend on GPU
» Allows the use of the same API for multiple generations of GPUs
  » However Vista/Windows7 required
» Lots of new features...
Shader Model 5.0
SM5.0 Basics

» All shader types support Shader Model 5.0
  » Vertex Shader
  » Hull Shader
  » Domain Shader
  » Geometry Shader
  » Pixel Shader

» Some instructions/declarations/system values are shader-specific

» Pull Model

» Shader subroutines
Uniform Indexing

» Can now index resource inputs
  » Buffer and Texture resources
  » Constant buffers
  » Texture samplers

» Indexing occurs on the *slot number*
  » E.g. Indexing of multiple texture arrays
  » E.g. indexing across constant buffer slots

» Index must be a constant expression

```
Texture2D txDiffuse[2] : register(t0);
Texture2D txDiffuse1   : register(t1);
static uint Indices[4] = { 4, 3, 2, 1  };
float4 PS(PS_INPUT i) : SV_Target
{
  float4 color=txDiffuse[Indices[3]].Sample(sam, i.Tex);
  // float4 color=txDiffuse1.Sample(sam, i.Tex);
}
```
SV_Coverage

» System value available to PS stage only
» Bit field indicating the samples covered by the current primitive
  » E.g. a value of 0x09 (1001b) indicates that sample 0 and 3 are covered by the primitive

» Easy way to detect MSAA edges for per-pixel/per-sample processing optimizations
  » E.g. for MSAA 4x:
  » `bIsEdge=(uCovMask!=0x0F && uCovMask!=0);`
Double Precision

» Double precision optionally supported
  » IEEE 754 format with full precision (0.5 ULP)
  » Mostly used for applications requiring a high amount of precision
  » Denormalized values support

» Slower performance than single precision!

» Check for support:

```c
D3D11_FEATURE_DATA.DOUBLES fdDoubleSupport;
pDev->CheckFeatureSupport( D3D11_FEATURE.DOUBLES,
        &fdDoubleSupport,
        sizeof(fdDoubleSupport) );

if (fdDoubleSupport.DoublePrecisionFloatShaderOps)
{
    // Double precision floating-point supported!
}
```
Gather(
» Fetches 4 point-sampled values in a single texture instruction
» Allows reduction of texture processing
  » Better/faster shadow kernels
  » Optimized SSAO implementations
» SM 5.0 Gather() more flexible
  » Channel selection now supported
  » Offset support (-32..31 range) for Texture2D
  » Depth compare version e.g. for shadow mapping
    
    Gather[Cmp]Red()
    Gather[Cmp]Green()
    Gather[Cmp]Blue()
    Gather[Cmp]Alpha()
Coarse Partial Derivatives

» \( \text{ddx}() / \text{ddy}() \) supplemented by coarse version
  » \( \text{ddx}_\text{coarse}() \)
  » \( \text{ddy}_\text{coarse}() \)

» Return *same* derivatives for whole 2x2 quad
  » Actual derivatives used are IHV-specific

» Faster than “fine” version
  » Trading quality for performance

Same principle applies to \( \text{ddy}_\text{coarse}() \)
Other Instructions

» FP32 to/from FP16 conversion
  » uint f32tof16(float value);
  » float f16tof32(uint value);
  » fp16 stored in low 16 bits of uint

» Bit manipulation
  » Returns the first occurrence of a set bit
    » int firstbithigh(int value);
    » int firstbitlow(int value);
  » Reverse bit ordering
    » uint reversebits(uint value);
  » Useful for packing/compression code
  » And more...
Unordered Access Views

» New view available in Shader Model 5.0
» UAVs allow binding of resources for arbitrary (unordered) read or write operations
  » Supported in PS 5.0 and CS 5.0

» Applications
  » Scatter operations
  » Order-Independent Transparency
  » Data binning operations

» Pixel Shader limited to 8 RTVs+UAVs total
  » `OMSetRenderTargetsAndUnorderedAccessViews()`

» Compute Shader limited to 8 UAVs
  » `CSSetUnorderedAccessViews()`
Raw Buffer Views

» New Buffer and View creation flag in SM 5.0
  » Allows a buffer to be viewed as array of typeless 32-bit aligned values
    » Exception: Structured Buffers
  » Buffer must be created with flag
    D3D11RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS
  » Can be bound as SRV or UAV
    » SRV: need D3D11BUFFEREX_SRV_FLAG_RAW flag
    » UAV: need D3D11BUFFER_UAV_FLAG_RAW flag

```c
ByteAddressBuffer MyInputRawBuffer;    // SRV
RWByteAddressBuffer MyOutputRawBuffer;   // UAV

float4 MyPS(PSINPUT input) : COLOR
{
  uint u32BitData;
  u32BitData = MyInputRawBuffer.Load(input.index);  // Read from SRV
  MyOutputRawBuffer.Store(input.index, u32BitData);  // Write to UAV
  // Rest of code ...
}
```
Structured Buffers

» New Buffer creation flag in SM 5.0
  » Ability to read or write a data structure at a specified index in a Buffer
  » Resource must be created with flag D3D11_Resource_MISC_BUFFER_STRUCTURED
  » Can be bound as SRV or UAV

```c
struct MyStruct
{
    float4 vValue1;
    uint uBitField;
};
StructuredBuffer<MyStruct> MyInputBuffer; // SRV
RWStructuredBuffer<MyStruct> MyOutputBuffer; // UAV

float4 MyPS(PSINPUT input) : COLOR
{
    MyStruct StructElement;
    StructElement = MyInputBuffer[input.index]; // Read from SRV
    MyOutputBuffer[input.index] = StructElement; // Write to UAV
    // Rest of code ...
}
```
Buffer Append/Consume

» Append Buffer enables *global* write counter
  » Can be used to append() new data at the end of the buffer – useful for building lists

» Declaration
  ```
  Append[ByteAddress/Structured]Buffer MyAppendBuf;
  ```

» Access to write counter
  ```
  uint uWriteCounter = MyAppendBuf.IncrementCounter();
  ```

» Append data to buffer using counter
  ```
  MyAppendBuf.Store(uWriteCounter, value);
  ```

» Same rules for Consume with read counter
  ```
  Consume[ByteAddress/Structured]Buffer MyConsumeBuf;
  uint uReadCounter = MyConsumeBuf.DecrementCounter();
  value = MyConsumeBuf.Load(uReadCounter);
  ```
Atomic Operations

» CS supports atomic operations
  » Can be used when multiple threads try to modify the same data location (UAV or TLS)
  » Avoid contention
    InterlockedAdd
    InterlockedAnd/InterlockedOr/InterlockedXor
    InterlockedCompareExchange
    InterlockedCompareStore
    InterlockedExchange
    InterlockedMax/InterlockedMin
  » Can optionally return original value
  » Potential cost in performance
    » Especially if original value is required
    » More latency hiding required
Compute Shader
Compute Shader Intro

» A new programmable shader stage in DX11
  » Independent of the graphic pipeline
» New industry standard for GPGPU applications
» CS enables general processing operations
  » Post-processing
  » Video filtering
  » Sorting/Binning
  » Setting up resources for rendering
  » Etc.
» Not limited to graphic applications
  » E.g. AI, pathfinding, physics, compression...
CS 5.0 Features

- Supports Shader Model 5.0 instructions
- Texture sampling and filtering instructions
  - Explicit derivatives required
- Execution not limited to fixed input/output
- Thread model execution
  - Full control on the number of times the CS runs
- Read/write access to “on-cache” memory
  - Thread Local Storage (TLS)
  - Shared between threads
  - Synchronization support
- Random access writes
  - At last! 😊 Enables new possibilities (scattering)
CS Threads

» A thread is the basic CS processing element
» CS declares the number of threads to operate on (the “thread group”)
  » \([\text{numthreads}(X, Y, Z)]\)
  » \texttt{void MyCS(…)}
» To kick off CS execution:
  » \texttt{pDev11->Dispatch( nX, nY, nZ );}
  » nX, nY, nZ: number of thread groups to execute
» Number of thread groups can be written out to a Buffer as pre-pass
  » \texttt{pDev11->DispatchIndirect(LPRESOURCE *hBGroupDimensions, DWORD dwOffsetBytes);}  
  » Useful for conditional execution

CS 5.0
X*Y*Z\leq1024  
Z\leq64
CS Threads & Groups

- `pDev11->Dispatch(3, 2, 1);`
- `[numthreads(4, 4, 1)]
  `void MyCS(...)`
- Total threads = $3 \times 2 \times 4 \times 4 = 96$

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Thread Grid](image)
CS Parameter Inputs

» pDev11->Dispatch(nX, nY, nZ);
» [numthreads(X, Y, Z)]
void MyCS(
    uint3 groupID:          SV_GroupID,
    uint3 groupThreadID:    SV_GroupThreadID,
    uint3 dispatchThreadID: SV_DispatchThreadID,
    uint  groupIndex:       SV_GroupIndex);

» groupID.xyz: group offsets from Dispatch()
    » groupID.xyz ∈ (0..nX-1, 0..nY-1, 0..nZ-1);
    » Constant within a CS thread group invocation

» groupThreadID.xyz: thread ID in group
    » groupThreadID.xyz ∈ (0..X-1, 0..Y-1, 0..Z-1);
    » Independent of Dispatch() parameters

» dispatchThreadID.xyz: global thread offset
    » = groupID.xyz*(X,Y,Z) + groupThreadID.xyz

» groupIndex: flattened version of groupThreadID
CS Bandwidth Advantage

» Memory bandwidth often still a bottleneck
  » Post-processing, compression, etc.

» Fullscreen filters often require input pixels to be fetched multiple times!
  » Depth of Field, SSAO, Blur, etc.
  » BW usage depends on TEX cache and kernel size

» TLS allows reduction in BW requirements

» Typical usage model
  » Each thread reads data from input resource
  » ...and write it into TLS group data
  » Synchronize threads
  » Read back and process TLS group data
Thread Local Storage

» Shared between threads
» Read/write access at any location
» Declared in the shader
  » `groupshared float4 vCacheMemory[1024];`
» Limited to 32 KB
» Need synchronization before reading back data written by other threads
  » To ensure all threads have finished writing
    » `GroupMemoryBarrier();`
    » `GroupMemoryBarrierWithGroupSync();`
CS 4.X

» Compute Shader supported on DX10(.1) HW
   » CS 4.0 on DX10 HW, CS 4.1 on DX10.1 HW
» Useful for prototyping CS on HW device before DX11 GPUs become available
» Drivers available from ATI and NVIDIA
» Major differences compared to CS5.0
   » Max number of threads is 768 total
   » Dispatch Zn==1 & no DispatchIndirect() support
   » TLS size is 16 KB
   » Thread can only write to its own offset in TLS
   » Atomic operations *not* supported
   » Only one UAV can be bound
   » Only writable resource is Buffer type
PS 5.0 vs CS 5.0
Example: Gaussian Blur

» Comparison between a PS 5.0 and CS5.0 implementation of Gaussian Blur

» Two-pass Gaussian Blur
  » High cost in texture instructions and bandwidth

» Can the compute shader perform better?
Gaussian Blur PS

» Separable filter Horizontal/Vertical pass
  » Using kernel size of x*y

» For each pixel of each line:
  » Fetch x texels in a horizontal segment
  » Write H-blurred output pixel in RT: $B_H = \sum_{i=1}^{x} G_i P_i$

» For each pixel of each column:
  » Fetch y texels in a vertical segment from RT
  » Write fully blurred output pixel: $B = \sum_{i=1}^{y} G_i P_i$

» Problems:
  » Texels of source texture are read multiple times
  » This will lead to cache trashing if kernel is large
  » Also leads to many texture instructions used!
Gaussian Blur PS
Horizontal Pass

Source texture

Temp RT
Gaussian Blur PS
Vertical Pass

Source texture (temp RT)

Destination RT
Gaussian Blur CS – HP(1)

groupshared float4 HorizontalLine[WIDTH]; // TLS
Texture2D txInput; // Input texture to read from
RWTexture2D<float4> OutputTexture; // Tmp output

[numthreads(WIDTH,1,1)]
void GausBlurHoriz(uint3 groupID: SV_GroupID,
uint3 groupThreadID: SV_GroupThreadID)
{
  // Fetch color from input texture
  float4 vColor=txInput[int2(groupThreadID.x,groupID.y)];
  // Store it into TLS
  HorizontalLine[groupThreadID.x]=vColor;
  // Synchronize threads
  GroupMemoryBarrierWithGroupSync();
  // Continued on next slide
}

pDevContext->Dispatch(1,HEIGHT,1);
Gaussian Blur CS – HP(2)

// Compute horizontal Gaussian blur for each pixel
vColor = float4(0,0,0,0);

[unroll]for (int i=-GS2; i<=GS2; i++)
{
    // Determine offset of pixel
t    int nOffset = groupThreadID.x + i;
    // Clamp offset
    nOffset = clamp(nOffset, 0,
    // Add color for pixels within horizontal filter
    vColor += G[GS2+i] * HorizontalLine[nOffset];
}

// Store result
OutputTexture[int2(groupThreadID.x,groupID.y)]=vColor;
Gaussian Blur BW:PS vs CS

» Pixel Shader
  » # of reads per source pixel: 7 (H) + 7 (V) = 14
  » # of writes per source pixel: 1 (H) + 1 (V) = 2
  » Total number of memory operations per pixel: 16
  » For a 1024x1024 RGBA8 source texture this is 64 MBytes worth of data transfer
    » Texture cache will reduce this number
    » But become less effective as the kernel gets larger

» Compute Shader
  » # of reads per source pixel: 1 (H) + 1 (V) = 2
  » # of writes per source pixel: 1 (H) + 1 (V) = 2
  » Total number of memory operations per pixel: 4
  » For a 1024x1024 RGBA8 source texture this is 16 MBytes worth of data transfer
Conclusion

» New Shader Model 5.0 feature set extensively powerful
  » New instructions
  » Double precision support
  » Scattering support through UAVs

» Compute Shader
  » No longer limited to graphic applications
  » TLS memory allows considerable performance savings

» DX11 SDK available for prototyping
  » Ask your IHV for a CS4.X-enabled driver
  » REF driver for full SM 5.0 support
Questions?

nicolas.thibieroz@amd.com