GDC

Wave Programming in D3D12 and Vulkan

David Lively Developer Technology Engineer AMD

Holger Gruen Developer Technology Engineer NVIDIA

GAME DEVELOPERS CONFERENCE" | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17

Terms

- Lane [DX] / Invocation [VK]
 - A single shader invocation (thread) within the wave
- Wave [DX] / Subgroup [VK]
 - Collection of shader invocations where amount varies per vendor
 - Wave [DX] execution model : all lanes execute simultaneously and in lock-step
 - Subgroup [VK] execution model : subgroup operations include implicit barrier
- Dynamically Uniform = same across all active lanes







Terms



Be aware that **some** intrinsics only work on active lanes/invocations!



Benefits of Wave Programming

- Faster synchronization of threads within a wave
 - Reduced use of barrier-/interlocked-intrinsics
 - For some cases
 - Simpler shader code
 - Easier to maintain
 - Easier to write
- More control over DFC coherency
 - Helps to improve flow coherency
 - Helps to improve memory access coherency



Wave Operations

- Available in Compute and Pixel shaders [DX], all stages* [VK]
- Share data between threads
- Categories (details to follow)
 - Query Get data about a single thread
 - Vote Compare values across a wave
 - Broadcast Share with all threads in the wave
 - Reduce Wave-level sum, product, bitwise, min/max, etc.
 - Scan & Prefix
 - Global ordered append
 - Quad Read & swap with neighbors in a quad DX Pixel Shader Only

* Quad-shuffle and helper lane operations available only in DirectX pixel shaders

Wave Operations

- Portable D3D12 [DX]
 - Expected to be provided via SM6 (talk covers details from preview spec online)
 - Operations supported in CS and PS
- Portable Vulkan [VK]
 - Currently provided by a collection of portable KHR extensions
 - Operations supported in all shader stages*
- Vendor-Specific Extensions
 - Vendors provide increased functionality (not covered in this talk)
 - AMD supports D3D11/D3D12 via AGS, and AMD extensions in Vulkan
 - NV supports D3D11/D3D12 via NVAPI, and NV extensions in Vulkan

* Quad-shuffle and helper lane operations available only in DirectX pixel shaders

Core Functionality

- Next slides highlight some of the core wave-programming constructs
 - Not enough time to include everything
 - See detailed specs for the full list. The links below are a good starting point
- D3D12 Shader Model 6 Preview
 - <u>https://msdn.microsoft.com/en-</u> <u>us/library/windows/desktop/mt733232%28v=vs.85%29.aspx</u>
- Vulkan Supports GLSL ARB Extensions
 - <u>https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_group_vote.txt</u>
 - <u>https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_ballot.txt</u>



Queries

- WaveGetLaneCount() [DX] / gl_SubGroupSizeARB [VK]
 - The number of lanes in a wave (typically 32 [NV] and 64 [AMD])
- WaveLaneIndex() [DX] / gl_SubGroupInvocationARB [VK]
 - Returns the index of the shader invocation's lane in the wave / subgroup
- WaveIsHelperLane() [DX] / gl_HelperInvocation [VK]
 - In a pixel / fragment shader, returns true if shader invocation is used only for derivatives (ddx, ddy) for the quad
- WaveOnce() [DX Only] Only run code for one lane



Broadcast

- WaveReadFirstLane() [DX] / readFirstInvocationARB() [VK]
 - Fetch value at first active lane in wave
 - Safe in divergent control flow and safe for non-full waves
- WaveReadLaneAt() [DX] / readInvocationARB() [VK]
 - Fetch value at specific lane in wave (lane index needs to be uniform across the wave HG)
 - Safe only when shader knows a specific lane will be active



Vote

- WaveAnyTrue() [DX] / anyInvocationARB() [VK]
 - Returns true if any active lane has a true value
- WaveAllTrue() [DX] / allInvocationsARB() [VK]
 - Returns true only if all active lanes have a true value
- WaveAllEqual() [DX] / allInvocationsEqualARB() [VK]
 - Returns true if all active lanes have the same value



Ballot

- WaveBallot() [DX] / ballotARB() [VK]
 - Returns bit array packed into an unsigned 64-bit integer
 - Bit array representing a bool value across the wave, one bit per lane
 - Bit is 0 when lane is inactive or value=false or lane > wave size
 - Bit is 1 when lane is active and value=true
 - LSB of return starts with lane 0



Parallel Reductions

- WaveAll<Op>() [DX]
 - Applies an associative operation across all active lanes in the wave
 - All active lanes have same final result
 - <Op>=Sum computes the sum of the value across all active lanes
 - For floating point, precision of the result can vary across implementations
 - Ordering of operations is implementation dependent
 - Operation may be computed on hardware by a parallel reduction including multiple passes
- Available in Vulkan via extensions





Parallel Prefix Operations

- WavePrefixSum() [DX]
 - Returns sum of value across all active lanes with a lower lane index
 - Result is unique per lane stored in VGPR
 - Operation also known as "SCAN"
- Available in Vulkan via extensions



WavePrefixSum() returns 0,1,4,4,6 ... one output/lane starting with lane index=0

Don't Code for Fixed Lane Count

- Can't assume same lane count on every GPU
 - D3D12_FEATURE_DATA_D3D12_OPTIONS1.WaveLaneCountMin [DX]
 - gl_SubGroupSizeARB [VK]
 - WaveGetLaneCount() [HLSL]
- Write flexible shaders that deal with differing lane counts
 - Minimize instruction divergence and data divergence
- Waves are 32 wide on NVIDIA GPUs
 - Recast problem to use 32 lanes whenever possible!
- Waves are 64 wide on AMD GPUs
 - Recast problem to use 64 lanes whenever possible!



Wave Programming Use Cases I

• Wave programming can reduce DFC divergence or memory divergence

Check these references for details:

- Reducing divergence can increase memory locality on NVIDIA
 - NVIDIA Optix uses wave ops for prioritized scheduling for a 1.25x perf improvement
 - <u>http://www.highperformancegraphics.org/previous/www_2009/presentations/nvidia-rt.pdf</u>
- Dynamically uniform values and loads leverage scalar hardware on AMD
 - Hardware has a separate scalar {ALU pipe, registers, cache} for dynamically uniform data
 - DOOM uses wave ops for a 1.43x performance improvement
 - <u>http://advances.realtimerendering.com/s2016/Siggraph2016_idTech6.pdf</u>

Wave Programming Use Cases II

- Choose optimized paths if in divergent and non-divergent control flow
 - Make sure to test performance/register pressure ramifications on all target GPUs
 - May leverage scalar hardware on AMD GCN

```
bool usePathA = ...
if(WaveAllTrue(usePathA)) {
    /* optimized for only path A */
} else if(WaveAllTrue(~usePathA)) {
    /* optimized for only path B */
} else {
    /* optimized for doing both paths A and B at same time */
}
```



Wave Programming Use Cases II

- Choose path if it is important for enough lanes in the wave
 - Make sure to test performance/register pressure ramifications on all target GPUs
 - May leverage scalar hardware on AMD GCN

```
bool wantPath;
// Will execute if more than 1/8 of the wave wants the path.
if(countbits(WaveBallot()) > (WaveGetLaneCount() >> 3)) {
```



Wave Programming Use Cases III

- Parallel prefix operations can be used to reduce Global Atomics
 - WavePrefixSum [DX], future extension [VK]
 - WavePrefixProduct [DX], future extension [VK]
- Example for fully-active, wave-granularity compute shader

```
uint count; // number of items to allocate
uint position = WavePrefixSum(count); // position in bulk allocation
uint base; // one global atomic allocates space for all lanes (bulk allocation)
```

```
if (WaveGetLaneIndex() == WaveGetLaneCount() - 1) {
    counters.InterlockedAdd(counterIndex, position + count, base);
}
```

position += WaveReadLane(base, WaveGetLaneCount() - 1); // final allocation

Wave Programming Use Cases IV - Reductions

- Just use WaveAll*() right?
- Sort of
 - Remember, wave size varies between device and vendor!

A 16x16 Reduction - Option 1

```
groupshared float3 g_center[4*(64/LANE_COUNT)]; // AMD 4 / NVIDIA 8
[numthreads(4/(64/LANE_COUNT),16,4*(64/LANE_COUNT)] // AMD 4x16x4 / NVIDIA 2x16x8
void main(...)
```

```
... // compute position
float3 center = (0.0f).xxx, total = WaveAllSum( position );
```

```
if( WaveGetLaneIndex() == 0 )
g_center[GroupThreadID.z]=total;
```

ł

}

```
GroupMemoryBarrierWithGroupSync();
```

```
[unrol1]for(int i=0; i< 4*( 64/LANE_COUNT ); ++i) // // AMD 4 / NVIDIA 8
    center += (1.0f/(16.0f*16.0f)) * g_center[i];</pre>
```



A 16x16 Reduction - Option 2

```
[numthreads(4/(64/LANE_COUNT),16,1] // AMD 4x16x1 / NVIDIA 2x16x1
void main(...)
{
```

```
float3 position[4*(64/LANE_COUNT)]; // AMD 4 / NVIDIA 8
```

```
... // compute each position[i]
```

```
float3 center = (0.0f).xxx;
```

}

```
[unrol1] for( int i = 0; i < 4*(64/LANE_COUNT); ++i ) // AMD 4 / NVIDIA 8
    center += (1.0f/(16.0f*16.0f)) * WaveAllSum( position[i] );</pre>
```



Which Option is fastest?

- Test and benchmark to find out
 - Benchmark on all relevant GPUs
 - Test for correctness on all relevant GPUs
- Limiters/Mileage will vary
 - Register pressure
 - Occupancy



Usecases V - Scans

•Just use WavePrefix*() right?

•Sort of, remember that max lane counts vary



8 wide Scans => by 2x4 wide Scans







8 wide Scans => by 2x4 wide Scans







64 wide Scan

```
[numthreads(LANE_COUNT,X,Z)] // 64 AMD, 32 NVIDIA
void main(...)
\{ // 64/LANE_COUNT => 1 AMD, 2 NVIDIA \}
   float scanRes[64/LANE_COUNT], value[64/LANE_COUNT];
   . . .
   for(int i = 0; i < 64/LANE_COUNT; ++i ) value[i] = ...;
   for(int i = 0; i < 64/LANE_COUNT; ++i ) scanRes[i] = WavePrefixSum(value);</pre>
   if( LANE_COUNT < 64 ) {
      ScanRes[1] += WaveReadLaneAt( scanRes[1], LANE_COUNT-1 )+
                    WaveReadLaneAt( value[0], LANE_COUNT-1 ); }
```

And now imagine a 128 wide Scan !







Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2017 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.