High Performance Post-Processing

Nathan Hoobler, NVIDIA

(nhoobler@nvidia.com)
Image-Based Effects are Great!
Beautiful, But Costly

- Workload differs greatly from 3D rendering
- Bandwidth-hungry shaders
- Algorithms shoe-horned into Graphics API

Direct3D 11 provides great ways around this
New Resource Types

• Buffers / Structured Buffers
• Unordered Access Views (UAV)
  – RWTexture/RWBuffer
  – Allows arbitrary reads and writes from PS and CS
    • Ability to “scatter” provides new opportunities
    • Have to be aware of hazards and access patterns
New Intrinsic Operations

• Interlocked/Append operations
  – Allow parallel workloads to combine results easily
  – Useful for collapsing information across the image
  – Not free! Cost increases if a value is hit often
    • NV: This may be more efficient from Compute shaders
    • NV: Append is more efficient than IncrementCounter for dynamically growing UAVs
    • AMD: No special cases for performance
DirectCompute

- New shader mode that operates on arbitrary threads
- Frees processing from restrictions of gfx pipeline
- Full access to conventional Direct3D resources
DIRECTCOMPUTE REVIEW
Why use Compute?

- Allows much finer-level control of workload
- Inter-thread communication
- Hardware has more freedom for optimizations
- Just plain easier to write!
Threads, Groups, and Dispatches

• Shaders run as a set of several thread blocks that execute in parallel
  – “Threads” runs the code given in the shader
  – “Groups” are sets of threads that can communicate using on-chip memory
  – “Dispatches” are sets of groups
Dispatch Example

// CPU Code
pContext->CSSetUnorderedAccessViews(
    0, 1, &pOutputUAV, NULL);
pContext->CSSetShader(pSimpleCS);
pContext->Dispatch(4,4,1);

// HLSL Code
RWTexture<float3> uavOut : register(u0);
[numthreads(8,8,1)]
void SimpleCS(uint3 tID : SV_DispatchThreadID)
{
    uavOut[tID] = float3(0,1,0);
}
DispatchIndirect

• Fetch Dispatch parameters from device buffer instead of from the CPU
• Let Compute work drive Compute!
  – Still bound by CPU to issue DispatchIndirect call
• Great when combined with Append buffers for dynamic workload generation
## Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Speed</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory (Buffers, Textures, Constants)</td>
<td>Longest Latency</td>
<td>All Threads</td>
</tr>
<tr>
<td>Shared Memory (groupshared)</td>
<td>Fast</td>
<td>Single Group</td>
</tr>
<tr>
<td>Local Memory (Registers)</td>
<td>Very Fast</td>
<td>Single Thread</td>
</tr>
</tbody>
</table>
Inter-Thread Communication

• Threads in a group can communicate via shared memory.
• Thread execution cannot depend on other groups!
  – Not all groups execute simultaneously.
  – Groups can execute in any order within a Dispatch.
  – *Inter-group dependency could lead to deadlocks.*
• If a group relies on results from another group, split shader into multiple dispatches.
Data Hazards and Stalls

• Re-binding a resource used as UAV may stall HW to avoid data hazards
  – Have to make sure all writes complete so they are visible to next dispatch
  – Driver may re-order unrelated Dispatch calls to hide this latency
Context Switch Overhead

• Have to be aware of context switch cost
  – Penalty switching between Graphics and Compute
  – Usually minimal unless repeatedly hit
  – Back-to-Back Dispatches avoid this, so group calls
OPTIMIZATION CONCEPTS
Common Pitfalls

Memory Limited
- Inefficient access pattern
- Inefficient formats
- Just too much data!

Computation Limited
- Divergent threads
- Bad Instruction Mix
- Poor Hardware Utilization
Memory Architecture

- D3D11 introduces complex memory systems
- Caching behavior depends on access mode
  - Buffers may hit cache better for linear accesses
  - Textures work better for less predictable/more 2D access within a group
Stratified Sampling

Bad – Poor Sample Locality

Good – Better for Cache
“Going with the Grain”

• Unlike textures, buffers are linear memory

• Be sure to read along the pitch of a 2D array mapped to a buffer where possible!

```cpp
Buffer<float> srvInput;
[numthreads(128,1,1)]
void ReadCS(
    uint3 gID : SV_GroupID
    uint3 tID : SV_DispatchThreadID)
{
    float val;

    // Good: Reading along pitch
    val = srvRead[128*gID.x+tID.x];
    // ... Use data ...

    // Bad: Reading against pitch
    val = srvRead[128*tID.x+gID.x];
    // ... Use data ...
}
```
Divergence

- Theoretically, threads execute independently
- Practically, they execute in parallel wavefronts
  - Threads are “masked” for instructions in untaken branches while wavefront executes
  - Different wavefronts can diverge without cost
  - Wavefronts size is hardware specific
    - NV:32  AMD:64
Wavefront Divergence

```c
// Assume WAVE_SIZE = wavefront size for
// the hardware (NV:32,AMD:64)
[numthreads(WAVE_SIZE,2,1)]
void DivergeCS(uint3 tID : SV_DispatchThreadID) {
    float val;
    // Divergent
    if (tID.x%2)
        val = ComplexFuncA(tID);
    else
        val = ComplexFuncB(tID);
    // Not divergent
    if (tID.y%2)
        val += ComplexFuncC(tID);
    else
        val += ComplexFuncD(tID);
    Output(tID, val);
}
```
Divergent Pixels

- In PS, threads are grouped into 2D clusters of samples
- Branches are OK if they’re coherent across the image
  - Especially if it saves work!
Utilization

- Create enough work to saturate hardware
  - Dozens of groups is about the sweet spot

- Maximize number of threads per group
  - Need enough to hide latency in the hardware
  - 256-512 is a good target

- Experiment with Shared memory usage
  - More shared memory = fewer groups/processor
  - Try making groups smaller when shared memory/thread increases
Group-Level Coordination

• Compute threads can communicate and share data via “groupshared” memory
  – Pre-load data used by every thread in a group
    • Unpacked values, Dynamic programming
    • Save bandwidth and computation
  – Share workload for common tasks
    • Compute the sum/max/etc of a set
    • More efficient than shared atomics
Pre-Loading into Shared Memory

Separable Convolution:

• Read entire footprint of kernel into shared memory
• Fetch values from shared buffer and multiply by kernel for each pixel
• Read less often, and more efficiently!
Naïve Sum

Buffer<float> srvIn;
groupshared float sSum;
[numthreads(GROUP_SIZE,1,1)]
void SimpleSumCS(...) {
  if (gtID.x == 0) {
    sSum = 0;
    for(int t=1; t<8; ++t) {
      sSums += srvIn[tID+t];
    }
  }
  GroupMemoryBarrierWithGroupSync();
  // Use total
}
Parallel Sum

Buffer<float> srvIn;
groupshared float sSums[GROUP_SIZE];
[numthreads(GROUP_SIZE,1,1)]
void ParallelSumCS(...)
{
    sSums[gtID.x] = srvIn[tID];
    GroupMemoryBarrierWithGroupSync();
    for(int t=GROUP_SIZE/2; t>0; t=t>>1)
    {
        if (gtID.x < t)
            sSums[gtID.x] += sSums[gtId.x+t];
        GroupMemoryBarrier();
    }
    // Use result; total is in sSums[0]
}

O(N) ops, but in parallel!
Buffer<float> srvIn;
groupshared float sSums[GROUP_SIZE];
[numthreads(GROUP_SIZE,1,1)]
void ParallelSumCS(...) {
    sSums[gtID.x] = srvIn[tID];
    GroupMemoryBarrierWithGroupSync();
    for(int t=1; t>GROUP_SIZE; t=t<<1) {
        if (gtID.x >= t)
            sSums[gtID.x] += sSums[gtId.x-t];
        GroupMemoryBarrier();
    }
    // Use results
    // sSums[N] = total of samples [0...N]
Case Study: Summed Area Table
SAT Approach

• Create texture where each value is the sum of all pixels before it in the image.

• Sample corners of region and use difference to compute average value.

$$\text{Avg} = \frac{S(X_2,Y_2) - S(X_2,Y_1) - S(X_1,Y_2) + S(X_1,Y_1)}{(X_2-X_1)(Y_2-Y_1)}$$
SAT Sampling

- Overlap multiple regions to better approximate filters

```cpp
float3 BetterFilter(float2 center, float fSize) {
    float3 value = float3(0,0,0);
    value += BoxFilter(center, 0.5*fSize, fSize);
    value += BoxFilter(center, fSize, 0.5*fSize);
    value += BoxFilter(center, 0.75*fSize, 0.75*fSize);
    value /= 3; // to account for overlap
    return value;
}
```
SAT Advantages

- Great technique blurring with varying kernels
  - Constant work per sample, regardless of kernel
  - Applications: DOF, sampling environment maps
- A textbook case for parallel prefix sum!
SAT Computation

Step 1: Sum Subsections

• Dispatch groups that sum segments of each row
SAT Computation

Step 2: Offset Segments

• Sum the last element of every group prior to this in parallel
• Add the total to each px in the row segment
• Output to new buffer, transposing coords
SAT Computation

Step 3: Repeat 1 (for columns)

Step 4: Repeat 2, transpose
Segment Sum Shader

1. Fetch values for the pixels covered by the group to shared memory
2. Perform a parallel prefix sum
3. Output results

```cpp
#define GROUP_SIZE 16
#define WARP_SIZE 32
Texture2D<float3> texInput : register(t0);
RWTexture2D<float3> texOutput : register(u0);
groupshared float3 sSums[WARP_SIZE*GROUP_SIZE];

[numthreads( WARP_SIZE, GROUP_SIZE, 1 )]
void SumSegments_CS(uint3 groupId : SV_GroupID, uint3 threadIdx : SV_GroupThreadID, uint3 dispatchId : SV_DispatchThreadID)
{
    uint2 pixelID = uint2(dispatchId.x, dispatchId.y);

    // 1. Fetch Values
    sSums[threadIdx.x + threadIdx.y*WARP_SIZE] = texInput[pixelID];
    GroupMemoryBarrierWithGroupSync();

    // 2. Parallel Prefix-Sum
    for (int t=1; t<WARP_SIZE; t=t*2) {
        if (threadIdx.x >= t) {
            sSums[threadIdx.x+threadIdx.y*WARP_SIZE] += sSums[(threadIdx.x-t)+threadIdx.y*WARP_SIZE];
        }
        GroupMemoryBarrierWithGroupSync();
    }

    // 3. Output Results
    texOutput[pixelID] = sSums[threadIdx.x + threadIdx.y*WARP_SIZE];
}
```
Segment Offset Shader

1. Fetch the totals of all previous segments
2. Perform a parallel sum of the segment totals
3. Add the final total to the value of each pixel in the segment to offset it, and transpose the coordinates for the next pass

In practice, do 1-2 multiple times based on image width (not shown for brevity)
SAT Depth of Field

Scene + Filter Size + SAT → Image
Case Study: Scattered Bokeh
Scattered Bokeh

• “Hot-Spots” make up most of the visual impact

• Sparse samples can be computed more efficiently with a scattered approach
Scattered Bokeh

1: Gather Hot-Spots

• Identify pixels over threshold
• Append positions to UAV
Scattered Bokeh

1: Gather Hot-Spots
2: Expand Samples
   • Render buffer as point data
   • Use GS to expand based on CoC
Scattered Bokeh

1: Gather Hot-Spots
2: Expand Samples
3: Accumulate Splats
   • Mask quads with Bokeh pattern
   • Accumulate masked splats
   • Opt: To save fill rate, use a pyramid of textures and assign splats based on size
Scattered Bokeh

1: Gather Hot-Spots
2: Expand Samples
3: Accumulate Splats
4: Combine Results
   • Splat straight to DOF’d image
   • OR blend DOF with splat pyramid
QUESTIONS?