What’s In This Talk?

• Brief review of last lecture

• Advanced usage patterns of GPU compute languages

• Rendering uses cases for GPU Computing Languages
  – Histograms (for shadows, tone mapping, etc)
  – Deferred rendering
  – Writing new graphics pipelines (sort of 😊)
Remember:

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

*Figure by Kayvon Fatahalian*
Definitions: Execution

• **Task**
  – A logically related set of instructions executed in a single execution context (aka shader, instance of a kernel, task)

• **Concurrent execution**
  – Multiple tasks that may execute simultaneously (because they are logically independent)

• **Parallel execution**
  – Multiple tasks whose execution contexts are guaranteed to be live simultaneously (because you want them to be for locality, synchronization, etc)
Synchronization

• Synchronization
  – Restricting when tasks are permitted to execute

• Granularity of permitted synchronization determines at which granularity system allows user to control scheduling
GPU Compute Languages Review

• “Write code from within two nested concurrent/parallel loops”

• Abstracts
  – Cores, execution contexts, and SIMD ALUs

• Exposes
  – Parallel execution contexts on same core
  – Fast R/W on-core memory shared by the execution contexts on same core

• Synchronization
  – Fine grain: between execution contexts on same core
  – Very coarse: between large sets of concurrent work
  – No medium-grain synchronization “between function calls” like task systems provide
GPU Compute Pseudocode

void myWorkGroup()
{
    parallel_for(i = 0 to NumWorkItems - 1)
    {
        ... GPU Kernel Code ... (This is where you write GPU compute code)
    }
}

void main()
{
    concurrent_for( i = 0 to NumWorkGroups - 1)
    {
        myWorkGroup();
    }
    sync;
}
DX CS/OCL/CUDA Execution Model

• Fundamental unit is **work-item**
  – Single instance of “kernel” program (i.e., “task” using the definitions in this talk)
  – Each work-item executes in single SIMD lane

• Work items collected in **work-groups**
  – Work-group scheduled on single core
  – Work-items in a work-group
    – Execute *in parallel*
    – Can share R/W on-chip scratchpad memory
    – Can wait for other work-items in work-group

• Users launch a **grid** of work-groups
  – Spawn many *concurrent* work-groups

Figure by Tim Foley
When Use GPU Compute vs Pixel Shader?

• Use GPU compute language if your algorithm needs on-chip memory
  – Reduce bandwidth by building local data structures

• Otherwise, use pixel shader
  – All mapping, decomposition, and scheduling decisions automatic
  – (Easier to reach peak performance)
Conventional Thread Parallelism on GPUs

• Also called “persistent threads”
• “Expert” usage model for GPU compute
  – Defeat abstractions over cores, execution contexts, and SIMD functional units
  – Defeat system scheduler, load balancing, etc.
  – Code not portable between architectures
Conventional Thread Parallelism on GPUs

• Execution
  – Two-level parallel execution model
  – Lower level: parallel execution of M identical tasks on M-wide SIMD functional unit
  – Higher level: parallel execution of N different tasks on N execution contexts

• What is abstracted?
  – Nothing (other than automatic mapping to SIMD lanes)

• Where is synchronization allowed?
  – Lower-level: between any task running on same SIMD functional unit
  – Higher-level: between any execution context
Why Persistent Threads?

• Enable alternate programming models that require different scheduling and synchronization rules than the default model provides

• Example alternate programming models
  – Task systems (esp. nested task parallelism)
  – Producer-consumer rendering pipelines
  – (See references at end of this slide deck for more details)
Sample Distribution Shadow Maps

Without ComputeShader

With ComputeShader

“Sample Distribution Shadow Maps,” Lauritzen et al., SIGGRAPH 2010
Sample Distribution Shadow Maps

• Algorithm Overview
  – Render depth buffer from eye
  – Analyze depth buffer with Compute Shader:
    – Build histogram of pixel positions
    – Analyze histogram to optimize shadow rendering
  – Renders shadow maps

• Compute Shader to build histogram (from pixel data)
  – Sequential and parallel code in single kernel (many fork-joins)
  – Liberal use of atomic to shared (on-chip) and global (off-chip) memory
  – Gather and scatter to arbitrary locations in off-chip and on-chip memory

• Compute Shader to analyze histogram
  – Sequential and parallel code in single kernel (many fork-joins)
Building Histogram in DX11 CS

[numthreads(BLOCK_DIM, BLOCK_DIM, 1)]

void ScatterHistogram(uint3 groupId : SV_GroupID,
                       uint3 groupThreadId : SV_GroupThreadID,
                       uint groupIndex : SV_GroupIndex)
{

    // Initialize local histogram in parallel
    // Parallelism:
    // - Within threadgroup: SIMD lanes map to histogram bins
    // - Between threadgroups: Each threadgroup has own histogram
    localHistogram[groupIndex] = emptyBin();

    GroupMemoryBarrierWithGroupSync();

    . . .
Building Histogram in DX11 CS

// Build histogram in parallel
// Parallelism:
// - Within threadgroup: SIMD lanes map to pixels in image tile
// - Between threadgroups: Each threadgroup maps to image tile
// Read and compute surface data
uint2 globalCoords = groupId.xy * TILE_DIM + groupThreadId.xy;
SurfaceData data = ComputeSurfaceDataFromGBuffer(globalCoords);

// Bin based on view space Z
// Scatter data to the right bin in our local (on-chip) histogram
int bin = int(ZToBin(data.positionView.z));
InterlockedAdd(localHistogram[bin].count, 1U);
InterlockedMin(localHistogram[bin].bounds.minTexCoordX, data.texCoordX);
InterlockedMax(localHistogram[bin].bounds.maxTexCoordX, data.texCoordX);
//... (more atomic min/max operations for other values in histogram bin) ...

GroupMemoryBarrierWithGroupSync();

...
Building Histogram in DX11 CS

// Use per-threadgroup scalar code to atomically merge all on-chip histograms into
// single histogram in global memory.

// Parallelism
// - Within threadgroup:  SIMD lanes map to histogram elements
// - Between threadgroups: Each threadgroup writing to single global histogram

uint i = groupIndex;
if (localHistogram[i].count > 0) {
    InterlockedAdd(gHistogram[i].count, histogram[i].count);
    InterlockedMin(gHistogram[i].bounds.minTexCoordX,   histogram[i].bounds.minTexCoordX    );
    InterlockedMin(gHistogram[i].bounds.minTexCoordY,   histogram[i].bounds.minTexCoordY    );
    InterlockedMin(gHistogram[i].bounds.minLightSpaceZ, histogram[i].bounds.minLightSpaceZ    );
    InterlockedMax(gHistogram[i].bounds.maxTexCoordX,   histogram[i].bounds.maxTexCoordX    );
    InterlockedMax(gHistogram[i].bounds.maxTexCoordY,   histogram[i].bounds.maxTexCoordY    );
    InterlockedMax(gHistogram[i].bounds.maxLightSpaceZ, histogram[i].bounds.maxLightSpaceZ    );
}
}
Optimization: Moving farther away from basic data-parallelism

• Problem---1:1 mapping between workgroups and image tiles
  – Flushes local memory to global memory more times than necessary
  – Would like larger workgroups but limited to 1024 workitems per group

• Solution
  – Use the largest workgroups possible (1024 workitems)
  – Launch fewer workgroups. Find sweet spot that fills all threads on all cores to maximize latency hiding but minimizes the writes to global memory
  – Loop over multiple image tiles within a single compute shader

• Take-away
  – “Invoke just enough parallel work to fill the SIMD lanes, threads, and cores of the machine to achieve sufficient latency hiding”
  – The abstraction is broken because this optimization exposes the number of hardware resources 😞
Building Histogram in DX11 CS

// Build histogram in parallel
// Parallelism:
//   - Within threadgroup: SIMD lanes map to pixels in image tile
//   - Between threadgroups: Each threadgroup maps to image tile
uint2 tileStart = groupId.xy * TILE_DIM + groupThreadId.xy;
for (uint tileY = 0; tileY < TILE_DIM; tileY += BLOCK_DIM) {
  for (uint tileX = 0; tileX < TILE_DIM; tileX += BLOCK_DIM) {
    // Read and compute surface data
    uint2 globalCoords = groupId.xy * TILE_DIM + groupThreadId.xy;
    SurfaceData data = ComputeSurfaceDataFromGBuffer(globalCoords);

    // Bin based on view space Z
    // Scatter data to the right bin in our local (on-chip) histogram
    int bin = int(ZToBin(data.positionView.z));
    InterlockedAdd(localHistogram[bin].count, 1U);
    InterlockedMin(localHistogram[bin].bounds.minTexCoordX,
                   data.texCoordX);
    ... (more atomic min/max ops for other values in histogram bin) ...
  }
}
GroupMemoryBarrierWithGroupSync();

...
SW Pipeline 1: Particle Rasterizer

• Mock-up particle rendering pipeline with render-target-read
  – Written by 2 people over the course of 1 week
  – Runs ~2x slower than D3D rendering pipeline (but has glass jaws)

Without Volumetric Shadow

With Volumetric Shadow
Tiled Particle Rasterizer in DX11 CS

[numthreads(RAST_THREADS_X, RAST_THREADS_Y, 1)]
void RasterizeParticleCS(uint3 groupId : SV_GroupID,
                        uint3 groupThreadId : SV_GroupThreadID,
                        uint groupIndex : SV_GroupIndex)
{
    uint i = 0;  // For all particles..
    while (i < mParticleCount) {
        GroupMemoryBarrierWithGroupSync();
        const uint particlePerIter = min(mParticleCount - i, NT_X * NT_Y);

        // Vertex shader and primitive assembly
        // Parallelism: SIMD lanes map over particles.
        if (groupIndex < particlePerIter) {
            const uint particleIndex = i + groupIndex;

            // ... read vertex data for this particle from memory,
            // construct screen-facing quad, test if particle intersects tile,
            // use atomics to on-chip memory to append to list of particles
        }
    }
    GroupMemoryBarrierWithGroupSync();

    . . .
Tiled Particle Rasterizer in DX11 CS

// Find all particles that intersect this pixel
// Parallelism: SIMD lanes map over pixels in image tile
for (n = 0; n < gVisibleParticlePerIter; n++) {
    if (ParticleIntersectsPixel(gParticles[n], fragmentPos)) {
        float dx, dy;
        ComputeInterpolants(gParticles[n], fragmentPos, dx, dy);
        float3 viewPos = BilinearInterp3(gParticles[n].viewPos, dx, dy);
        float3 entry, exit, t;
        if (IntersectParticle(viewPos, gParticles[n], entry, exit, t)) {
            // Run pixel shader on this particle
            // Read-modify-write framebuffer held in global off-chip memory
        }
    }
}

i += particlePerIter;
SW Pipeline 1: Particle Rasterizer

• Usage
  – Atomics to on-chip memory
  – Gather/scatter to on-chip and off-chip memory
  – Latency hiding of off-chip memory accesses

• Lesson learned
  – The programmer productivity of these programming models is impressive
  – This pipeline is statically scheduled (from a SW perspective) but underlying hardware scheduler is dynamically scheduling threadgroups
  – Needs to be doing dynamic SW scheduling to achieve more stable / higher performance
Software Pipeline 2: OptiX

• NVIDIA interactive ray-tracing library
  – Started as research project, product announced 2009

• Custom rendering pipeline
  – Implemented entirely with CUDA using “persistent threads” usage pattern
  – Users define geometry, materials, lights in C-for-CUDA
  – PTX intermediate layer used to synthesize optimized code

• Other custom pipelines in the research world
  – Stochastic rasterization, particles, conservative rasterization, ...
(Possibly the most important use of ComputeShader)

Deferred Rendering

(Slides by Andrew Lauritzen)
Overview

• Forward shading
• Deferred shading and lighting
• Tile-based deferred shading
Forward Shading

• Do everything we need to shade a pixel
  – for each light
    – Shadow attenuation (sampling shadow maps)
    – Distance attenuation
    – Evaluate lighting and accumulate

• Multi-pass requires resubmitting scene geometry
  – Not a scalable solution
Forward Shading Problems

• Ineffective light culling
  – Object space at best
  – Trade-off with shader permutations/batching

• Memory footprint of all inputs
  – Everything must be resident at the same time (!)

• Shading small triangles is inefficient
  – Covered earlier in this course: [Fatahalian 2010]
Conventional Deferred Shading

• Store lighting inputs in memory (G-buffer)
  – for each light
    – Use rasterizer to scatter light volume and cull
    – Read lighting inputs from G-buffer
    – Compute lighting
    – Accumulate lighting with additive blending

• Reorders computation to extract coherence
Modern Implementation

• Cull with screen-aligned quads
  – Cover light extents with axis-aligned bounding box
    – Full light meshes (spheres, cones) are generally overkill
    – Can use oriented bounding box for narrow spot lights
  – Use conservative single-direction depth test
    – Two-pass stencil is more expensive than it is worth
    – Depth bounds test on some hardware, but not batch-friendly
Lit Scene (256 Point Lights)
Deferred Shading Problems

• Bandwidth overhead when lights overlap
  – for each light
    – Use rasterizer to scatter light volume and cull
    – Read lighting inputs from G-buffer $\Leftarrow$ overhead
  – Compute lighting
    – Accumulate lighting with additive blending $\Leftarrow$ overhead

• Not doing enough work to amortize overhead
Improving Deferred Shading

• Reduce G-buffer overhead
  – Access fewer things inside the light loop
  – Deferred lighting / light pre-pass

• Amortize overhead
  – Group overlapping lights and process them together
  – Tile-based deferred shading
Tile-Based Deferred Rendering

Parallel_for over lights
   Atomically append lights that affect tile to shared list

Barrier

Parallel_for over pixels in tile
   Evaluate all selected lights at each pixel
Tile-Based Deferred Shading

• **Goal**: amortize overhead
  - Large reduction in bandwidth requirements

• **Use screen tiles to group lights**
  - Use tight tile frusta to cull non-intersecting lights
    - Reduces number of lights to consider
  - Read G-buffer once and evaluate all relevant lights
    - Reduces bandwidth of overlapping lights

• See [Andersson 2009] for more details
Lit Scene (1024 Point Lights)
Tile-Based Light Culling
Quad-Based Lighting Culling
Light Culling Only at 1080p

Frame Time (ms)

Number of Point Lights

- Slope ~ 7 μs / light
- Slope ~ 0.5 μs / light

Tile setup dominates

- Quad (ATI 5870)
- Quad (NVIDIA 480)
- Tiled (NVIDIA 480)
- Tiled (ATI 5870)
Deferred lighting slightly faster, but trends similarly.

Slope ~ 20 µs / light

Slope ~ 4 µs / light

Few lights overlap

Deferred Shading (NVIDIA 480)
Deferred Shading (ATI 5870)
Deferred Lighting (ATI 5870)
Deferred Lighting (NVIDIA 480)
Tiled (NVIDIA 480)
Tiled (ATI 5870)
Anti-aliasing

• Multi-sampling with deferred rendering requires some work
  – Regular G-buffer couples visibility and shading

• Handle multi-frequency shading in user space
  – Store G-buffer at sample frequency
  – Only apply per-sample shading where necessary
  – Offers additional flexibility over forward rendering
Identifying Edges

• Forward MSAA causes redundant work
  – It applies to all triangle edges, even for continuous, tessellated surfaces

• Want to find *surface* discontinuities
  – Compare sample depths to depth derivatives
  – Compare (shading) normal deviation over samples
Per-Sample Shading Visualization
Deferred Rendering Conclusions

• Deferred shading is a useful rendering tool
  – Decouples shading from visibility
  – Allows efficient user-space scheduling and culling

• Tile-based methods win going forward
  – ComputeShader/OpenCL/CUDA implementations save a lot of bandwidth
  – Fastest and most flexible
  – Enable efficient MSAA
Summary for GPU Compute Languages

• GPU compute languages
  – “Easy” way to exploit compute capability of GPUs (easier than 3D APIs)
  – The performance benefit over pixel shaders comes when using on-core R/W memory to save off-chip bandwidth
  – Increasingly used as “just another tool in the real-time graphics programmer’s toolkit”
    – Deferred rendering
    – Shadows
    – Post-processing
    – ...
  – The current languages have a lot of rough edges and limitations.
Future Work

• Hierarchical light culling
  – Straightforward but would need lots of small lights

• Improve MSAA memory usage
  – Irregular/compressed sample storage?
  – Revisit binning pipelines?
  – Sacrifice higher resolutions for better AA?
Acknowledgements

• Microsoft and Crytek for the scene assets
• Johan Andersson from DICE
• Craig Kolb, Matt Pharr, and others in the Advanced Rendering Technology team at Intel
• Nico Galoppo, Anupreet Kalra and Mike Burrows from Intel
References


Questions?

• Full source and demo available at:
  – http://visual-computing.intel-research.net/art/publications/deferred_rendering/
Quad-Based Light Culling
Deferred Lighting / Light Pre-Pass

• Goal: reduce G-buffer overhead
• Split diffuse and specular terms
  – Common concession is monochromatic specular
• Factor out constant terms from summation
  – Albedo, specular amount, etc.
• Sum inner terms over all lights
Deferred Lighting / Light Pre-Pass

• Resolve pass combines factored components
  – Still best to store all terms in G-buffer up front
  – Better SIMD efficiency

• Incremental improvement for some hardware
  – Relies on pre-factoring lighting functions
  – Ability to vary resolve pass is not particularly useful

• See [Hoffman 2009] and [Stone 2009]
MSAA with Quad-Based Methods

• Mark pixels for per-sample shading
  – Stencil still faster than branching on most hardware
  – Probably gets scheduled better

• Shade in two passes: per-pixel and per-sample
  – Unfortunately, duplicates culling work
  – Scheduling is still a problem
Per-Sample Scheduling

• Lack of spatial locality causes hardware scheduling inefficiency
MSAA with Tile-Based Methods

• Handle per-pixel and per-sample in one pass
  – Avoids duplicate culling work
  – Can use branching, but incurs scheduling problems
  – Instead, reschedule per-sample pixels
    – Shade sample 0 for the whole tile
    – Pack a list of pixels that require per-sample shading
    – Redistribute threads to process additional samples
    – Scatter per-sample shaded results
Tile-Based MSAA at 1080p, 1024 Lights

Frame Time (ms)

Crytek Sponza (ATI 5870)  2009 Game (ATI 5870)  Crytek Sponza (NVIDIA 480)  2009 Game (NVIDIA 480)

- No MSAA
- 4x MSAA (Branching)
- 4x MSAA (Packed)
4x MSAA Performance at 1080p

Deferred lighting even less compelling

Frame Time (ms)

Deferred Shading (ATI 5870)
Deferred Lighting (ATI 5870)
Deferred Shading (NVIDIA 480)
Deferred Lighting (NVIDIA 480)
Tiled (ATI 5870)
Tiled (NVIDIA 480)

Slope ~ 35 µs / light
Slope ~ 5 µs / light

Tiled takes less of a hit from MSAA

Number of Point Lights