Building Real-Time Rendering Pipelines in Software

Aaron Lefohn
Recap/Overview

- Recap of this course so far
  - GPU architecture
  - Parallel programming for graphics
  - Parallel graphics algorithms plus the current rendering pipeline
    - Post/image processing
    - Order-independent transparency
    - Shadows

- This lecture: building new rendering pipelines: CPU/GPU
- Next two lectures: ray tracing and micropolygon pipelines
“The Graphics Pipeline”

- Central to the rise of 3D hardware and software.
- A stable and universal abstraction
- Shaped the evolution of the field…
- … while leaving enormous room for innovation.
The Graphics Pipeline is evolving

Direct3D 11 Shading

Input Assembler → Vertex Shader → Hull Shader → Pixel Shader → Tessellator → Output Merger

Domain Shader → Geometry Shader → Stream Output

Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009
“GPU” is evolving

- Continued drive for algorithmic innovation and advanced rendering techniques

- First class programming models for compute:
  - OpenCL, compute shaders, vendor specific, …

- New / different hardware implementations:
  - E.g., Larrabee, CPU-GPU combinations / hybrids
  - Even NVIDIA and AMD GPUs are very different

*Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009*
CPU is evolving

- CPUs getting more parallel
- Can buy machine today with 32 CPU cores
  - 4-socket, 8-cores-per-socket
  - 64 HW threads

- Intel and AMD will ship CPU-GPU integrated dies soon
  - Imagine a 4-socket, N-CPU-M-GPU system?
What Kinds of Pipelines Will Be Built?

- Custom front-end (PS3)
- Special-purpose: e.g., particle with render-target-read
- Ray tracing
- Micropolygon
- Better sampling
- Hybrids
- ...

5/14/2010 CS448S - Beyond Programmable Shading
What Makes up a Graphics Pipeline?

- Recap foley’s “what is a pipeline” slides
- GRAMPS overview
Task Parallelism
Mental Model

- Think of task as asynchronous function call
  - “Do F() at some point in the future…”
  - Optionally “… after G() is done”

- Can be implemented in HW or SW

- Usually cooperative, not preemptive
  - Task runs to completion – no blocking
  - Can create new tasks, but not wait
Mental Model

- Execute $N$ independent work items
  - aka: elements, fragments, threads

- All work items run the same program: kernel

- Work item uses data determined by $0 \leq i < N$
  - $[0, N)$ is the domain of computation
Pipeline Parallelism
Key Idea

- Algorithm is ordered sequence of stages
  - Each stage emits zero or more items for next

- Buffer items between stages
  - Increase throughput by running stages in parallel
  - Re-balance load between stages

- Exploit producer/consumer locality
  - On-chip FIFOs
  - Efficient bus between cores
  - ...
Why pipelines?

- **Variable-rate amplification**
  - Rasterization: 1 triangle in, 0-N fragments out
  - Ray tracer: 1 hit in, 0-N secondary/shadow rays out
  - Load imbalance
Key Pipeline Design Challenges

- Scheduling, scheduling, scheduling

- Load balancing between stages
  - Buffer up results for next stage
  - Prioritization: favor early stages? favor late stages?
  - Where place boundaries between stages?

- Capturing producer-consumer locality between stages
  - Keep buffers small, local
  - On-chip FIFOs, busses, caches, scratchpads
- Programming model and run-time for parallel hardware
- **Graphs** of stages and queues
- GRAMPS handles scheduling, parallelism, data-flow

**Example: Simple GRAMPS Graph**

Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009
The Graphics Pipeline becomes an app!

- Structure/setup is (application) software
  - Customized or completely novel renderers
- Reuses current hardware: FIFOs, shader cores, rast, …
- Analogous to the transition to programmable shading
  - Proliferation of new use cases and parameters
  - Not (unthinkably) radical

Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009
Writing a GRAMPS application

Design the execution graph:

- Input
- Vertex
- Rast
- Pixel
- Merge
- Frame Buffer

Design the stages:

- Shaders
- Threads (and Fixed Function stages)

Instantiate and launch.

Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009
More Detail – Queues

- Queues operate at a “packet” granularity
  - “Large bundles of coherent work”

- GRAMPS can optionally enforce ordering
  - Required for some workloads, adds overhead

Slide from Jeremy Sugerman
GRAMPS presentation @ SIGGRAPH 2009
More Detail – Shaders

- Shaders: Like pixel (or compute) shaders, stateless
  - Automatic instancing, pre-reserve/post-commit
- “Collection” packets: shared header and N elements
- New: “Push” operation to coalesce variable outputs
More Detail – Thread/Fixed Function

- Threads: Like POSIX threads, stateful
  - Explicit reserve/commit on queues
- Fixed Function: Effectively non-programmable Threads
Queue sets enable binning-style algorithms

- One logical queue with multiple lanes (or bins)
  - One consumer at a time per lane
  - Many lanes with data allows many parallel consumers
Software Pipeline Examples

- Hybrid CPU-GPU
  - Custom geometry front-end: PlayStation3
- CPU
  - Intel CPU real-time ray tracer
- OpenCL/CUDA/DX11 ComputeShader
  - Simple particle pipeline in DX11 CS
  - SW rasterizer, REYES, ray tracer

Note: Spot the parallelism constructs we just discussed in these examples
Custom Geometry Front-End: PlayStation3

- Generate/process geometry on Cell SPUs
- GPU consumes output of Cell pipeline
  - Tight producer-consumer synchronization SPUs ↔ GPU
  - 1 ring-buffer per SPU thread

From Jon Olick (id Software), SIGGRAPH 2008 Beyond Programmable Shading course
Custom geometry processing

Software control opens up great flexibility and programmability!

Simple custom culling/processing that we’ve added:
- Partition bounding box culling
- Mesh part culling
- Clip plane triangle trivial accept & reject
- Triangle cull volumes (inverse clip planes)

Others are doing: Full skinning, morph targets, CLOD, cloth

Future wish: No forced/fixed vertex & geometry shaders
- DIY compute shaders with fixed-func stages (tessellation and rasterization)
- Software-controlled queuing of data between stages
  - To avoid always spilling out to memory
“Old” Multi-Core CPU Ray Tracer (Does Not Scale)

- “Render” and “Acceleration Structure Build” done by two different sets of pthreads
  - Use explicit pthreads that subscribe all HW threads (no task system)
  - All tasks use SSE where applicable
  - Mix of task and data parallelism
  - Synchronously switch between render and build pthreads

- Render system
  - Pick from queue of tiles in task-stealing way

- Acceleration structure build system
  - Build two-level bounding volume hierarchy (BVH) with one BVH per object and a “BVH of per-object BVHs”
“Old” CPU Ray Tracer Movies

- **Caveat**
  - This is not the state-of-the-art system. The newer system scales much better and supports far richer content

- **Demo details**
  - Quake Wars
  - 24-CPU system (4 sockets, 6 cores per socket)
  - 300K triangle scenery + ~500 dynamic characters
  - 1 sun light
  - ~2.5 rays per pixel
  - [http://visual-computing.intel-research.net/](http://visual-computing.intel-research.net/)
Problems with “Old” CPU Ray Tracer

- Explicit pthreads a bad idea
  - Full subscription of machine
  - Switching between build and render threads is expensive
  - Thread barriers are bad
  - Want cooperative task system to schedule all work

- Serialization of build-render-build also a problem
  - Does not scale to many-core and prevents overlapping frames
New Many/Multi-CPU Ray Tracer (Scalable)

- Use one cooperative tasking system that “does it all”
  - Task system uses a threadpool shared by all parallel systems
  - All work in renderer is a task
    - Task system synchronization, scheduling, dependency tracking, etc.
    - Tasks spawn task-parallel or data-parallel work
      - E.g., Object can trigger new transform task, update tasks, etc

- Overlap two frames
  - Parallelize build and render from different frames to fill in unavoidable gaps in schedule

- Use SIMD throughout all tasks
Recent research and product demonstrate it is increasingly possible to write SW pipelines on the GPU

- Simple (naïve, no dynamic load balancing) pipelines relatively easy to write
- High performance and scalability requires task system (just like the many-core CPU solution)
- Building task system with current GPU programming models requires “turning programming model inside out”

References
- Rasterizer: “FreePipe”, I3D 2010 (1 triangle per SIMD lane)
- REYES: “RenderAnts,” SIGGRAPH Asia 2009
- Task systems on the GPU
  - “Understanding the Efficiency of Ray Traversal on GPU,” Aila et al, HPG 2009
  - “Task Management for a Programmable Real-time Reyes Renderer”, Tzeng et al, HPG 2010
- Look for more SW rendering papers at High Performance Graphics 2010
Basic DX11 SW Pipeline: Particle “Rasterizer”

- Mock-up particle rendering pipeline with render-target-read
  - Written by 2 people over ~1 week (Marco Salvi and Aaron Lefohn)
  - Embarrassingly simple: Only ~300 lines of DX11 CS code
  - Runs as fast as 2x slower than D3D rendering pipeline (but has glass jaws)
Tiled Particle “Rasterizer” in DX11 CS

```c
[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();
    }
}
```
// 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;
}
Basic DX11 SW Pipeline: Particle “Rasterizer”

- **Caveat**
  - This is NOT an optimized SW pipeline!!

- **Lesson learned**
  - 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
SW Pipelines on the GPU: Task Systems

- Limitation of current GPU programming models
  - Kernels cannot spawn new work
  - Scheduling of workgroups is hidden/opaque
  - No function pointers

- Solution
  - Defeat HW scheduler and put all scheduling in SW
  - Build SW task system inside of GPU prog. model
  - Sometimes called “persistent threads” usage model
SW Pipelines on the GPU: Task Systems

- **How?**
  - Abuse current GPU prog. models to look like (CPU) threads
  - Launch “as many warps/wavefronts as there are execution contexts”
  - Combine all code for all possible “tasks” into big switch statement
  - Spawn work by pushing a “call” to one of the case statements and its arguments onto global queue
  - Use local queues for optimization
  - Implement CPU-like task system scheduling such as work stealing, etc.
SW Pipelines on the GPU: Task Systems

- **Pros**
  - Support nested and braided parallelism on current GPUs
  - Dynamic SW load balancing
  - Enables SW rendering pipelines on current GPUs

- **Cons**
  - Fragile, arch-specific code not portable or safe
  - Register pressure of big switch can destroy performance
NVIDIA OptiX

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

- Custom rendering pipeline
  - Implemented entirely in C-for-CUDA/PTX
  - Users define geometry, materials, lights in C-for-CUDA
  - PTX intermediate layer used to synthesize optimized code

- Steve Parker (NVIDIA Research) guest lecturer in this class on Thursday talking about OptiX
SW Pipeline System Design Considerations

- Runtime code generation
  - Need JIT for ‘specialization’ to efficiently support user-defined code in inner-most loops of system

- Concurrent execution of multiple invocations of a pipeline
  - Implicit or explicit synchronization (buffer tracking)?
  - Multi-frame rendering?

- Scheduling must be “robust”
  - Very difficult aspect of SW pipelines
Open Research Questions (Many!)

- There are many open research questions
- Here are a few:
  - Scheduling
    - Should scheduling run on the same processors as the pipeline stages?
    - Good scheduling is the key/hardest problem in building real-time SW rendering pipelines
  - Programming model
    - Abstractions for non-expert users to create pipelines?
    - What about cross-stage optimizations such as early-z?
    - What is the base programming model on top of which pipelines are built?
  - ...And all of the graphics research to be done with SW pipelines
Conclusions

- **Software real-time pipelines are here**
  - SPU-GPU pipelines shipping in PS3 games
  - SW ray tracing pipeline running interactively on multi-core CPUs and GPUs. Shipping product from NVIDIA (interactive but not real-time)
  - Real-time SW rendering research is quickly growing research area

- **Many unsolved research problems in:**
  - Programming models, HW architectures, rendering, …
  - Key design challenges: scheduling, buffering, capturing locality

- **Contact point between CPUs and GPUs**
  - Especially interesting on upcoming hybrid CPU-GPU chips from Intel and AMD
References


Acknowledgements

- Tim Foley, Intel/Stanford
- Jeremy Sugerman, Stanford
- Jonathan Ragan-Kelley, MIT
- Ingo Wald, Bill Mark, Marco Salvi, Matt Pharr, Craig Kolb, Intel
- Mike Houston, AMD