Introduction to Parallel Programming For Real-Time Graphics (CPU + GPU)

Mike Houston, AMD / Stanford
Aaron Lefohn, Intel / University of Washington
What’s In This Talk?

• Overview of parallel programming models used in real-time graphics products and research
  – Abstraction, execution, synchronization
  – Shaders, task systems, conventional threads, graphics pipeline, “GPU” compute languages

• Parallel programming models
  – Vertex shaders
  – Conventional thread programming
  – Task systems
  – Graphics pipeline
  – GPU compute languages (ComputeShader, OpenCL, CUDA)

• Discuss
  – Strengths/weaknesses of different models
  – How each model maps to the architectures
What Goes into a Game Frame? (2 years ago)
Computation graph for *Battlefield: Bad Company* provided by DICE

Spring 2011 – Beyond Programmable Shading
Data Parallelism

Spring 2011 – Beyond Programmable Shading
Task Parallelism
Graphics Pipelines

- Input Assembly
- Vertex Shading
- Primitive Setup
- Geometry Shading
- Rasterization
- Pixel Shading
- Output Merging
Remember GPU Architecture talk

Figure by Kayvon Fatahalian
Hardware Resources (from Arch Talk)

- Core
- Execution Context
- SIMD functional units
- On-chip memory

Figure by Kayvon Fatahalian
Abstraction

• Abstraction enables portability and system optimization
  – E.g., dynamic load balancing, producer-consumer, SIMD utilization

• Lack of abstraction enables arch-specific user optimization
  – E.g., multiple execution contexts jointly building on-chip data structure

• Remember:
  – When a parallel programming model abstracts a HW resource, code written in that programming model scales across architectures with varying amounts of that resource
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
Vertex Shaders: "Pure Data Parallelism"

• Execution
  – Concurrent execution of identical per-vertex tasks

• What is abstracted?
  – Cores, execution contexts, SIMD functional units, memory hierarchy

• What synchronization is allowed?
  – Between draw calls
Shader (Data-parallel) Pseudocode

```c
concurrent_for( i = 0 to numVertices - 1 )
{
    ... execute vertex shader ... 
}
```

- SPMD = Single Program Multiple Data
  - This type of programming is sometimes called SPMD
  - Instance the same program multiple times and run on different data
  - Many names: shader-style, kernel-style, SPMD
Conventional Thread Parallelism (e.g., pthreads)

• Execution
  – Parallel execution of N tasks with N execution contexts

• What is abstracted?
  – Nothing (ignoring preemption)

• Where is synchronization allowed?
  – Between any execution context at various granularities
Conventional Thread Parallelism

• Directly program:
  – N execution contexts
  – N SIMD ALUs / execution context
  – ...

• To use SIMD ALUs:

  __m128 a_line, b_line, r_line;
  r_line = _mm_mul_ps(a_line, b_line);

• Powerful, but dangerous...

Figure by Kayvon Fatahalian
Game Workload Example
Typical Game Workload

• Subsystems given % of overall time “budget”
• Input, Miscellaneous: 5%
• Physics: 30%
• AI, Game Logic: 10%
• Graphics: 50%
• Audio: 5%

GPU Workload:

I | Physics | AI | Graphics | A

“Rendering”
Parallelism Anti-Pattern #1

- Assign each subsystems to a SW thread

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Physics</td>
<td>AI</td>
<td>Graphics</td>
</tr>
<tr>
<td>I</td>
<td>Physics</td>
<td>AI</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

- Problems
  - Communication/synchronization
  - Load imbalance
  - Preemption leads to thrashing

- Don’t do this!
Parallelism Anti-Pattern #2

• Group subsystems into HW threads

• Problems
  – Communication/synchronization
  – Load imbalance
  – Poor scalability (4, 8, ... HW threads)

• Don’t do this either!

Slide by Tim Foley
Better Solution: Find Concurrency...

- Identify where ordering constraints are needed and run concurrently between constraints.

- Visualize as a graph.
...And Distribute Work to Threads

- Dynamically distribute medium-grained concurrent tasks to hardware threads
- (Virtualize/abstract the threads’

Slide by Tim Foley
“Task Systems” (Cilk, TBB, ConcRT, GCD, …)

• Execution
  – Concurrent execution of many (likely different) tasks

• What is abstracted?
  – Cores and execution contexts
  – Does not abstract: SIMD functional units or memory hierarchy

• Where is synchronization allowed?
  – Between tasks
Mental Model: Task Systems

• 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
  – Launching/spawning task is nearly as fast as function call
  – Usage model: “Spawn 1000s of tasks and let scheduler map tasks to execution contexts”

• Usually cooperative, not preemptive
  – Task runs to completion – no blocking
  – Can create new tasks, but not wait
void myTask(...some arguments...)
{
    ...
}

void main()
{
    for( i = 0 to NumTasks - 1 )
    {
        cilk_spawn myTask(...);
    }
    cilk_sync;
}
Task Parallel Code (Cilk)

```c
void myTask(...some arguments...)
{
  ...
}

void main()
{
  cilk_for( i = 0 to NumTasks - 1 )
  {
    myTask(...);
  }
  cilk_sync;
}
```
Nested Task Parallel Code (Cilk)

```c
void barTask(...some parameters...)
{
    ...
}

void fooTask(...some parameters...)
{
    if (someCondition) {
        cilk_spawn barTask(...);
    }
    else {
        cilk_spawn fooTask(...);
    }
    // Implicit cilk_sync at end of function
}

void main()
{
    cilk_for( i = 0 to NumTasks - 1 ) {
        fooTask(...);
    }
    cilk_sync;
    ...
    ... More code ...
}
```
“Task Systems” Review

- **Execution**
  - Concurrent execution of many (likely different) tasks

- **What is abstracted?**
  - Cores and execution contexts
  - *Does not abstract: SIMD functional units or memory hierarchy*

- **Where is synchronization allowed?**
  - Between tasks

*Figure by Kayvon Fatahalian*
DirectX/OpenGL Rendering Pipeline (Combination of multiple models)

• Execution
  – Data-parallel concurrent execution of identical task within each shading stage
  – Task-parallel concurrent execution of different shading stages
  – No parallelism exposed to user

• What is abstracted?
  – (just about everything)
  – Cores, execution contexts, SIMD functional units, memory hierarchy, and fixed-function graphics units (tessellator, rasterizer, ROPs, etc)

• Where is synchronization allowed?
  – Between draw calls
GPU Compute Languages (Combination of Multiple Models)

- DX11 DirectCompute
- OpenCL
- CUDA

- There are multiple possible usage models. We’ll start with the “text book” hierarchical data-parallel usage model
GPU Compute Languages

• Execution
  – Hierarchical model
  – Lower level is parallel execution of identical tasks (work-items) within work-group
  – Upper level is concurrent execution of identical work-groups

• What is abstracted?
  – Work-group abstracts a core’s execution contexts, SIMD functional units
  – Set of work-groups abstracts cores
  – Does not abstract core-local memory

• Where is synchronization allowed?
  – Between work-items in a work-group
  – Between “passes” (set of work-groups)
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

Spring 2011 – Beyond Programmable Shading
GPU Compute Models

Work-group

barrier

Work-group

barrier

…”
GPU Compute Use Cases

• 1:1 Mapping
• Simple Fork/Join
• Switching Axes of Parallelism

Increasing Sophistication
1:1 Mapping

• One work item per ray / per pixel / per matrix element
• Every work item executes the same kernel
• Often first, most obvious solution to a problem
• “Pure data parallelism”

```c
void saxpy( int i,  
    float a,  
    const float* x,  
    const float* y,  
    float* result )
{
    result[i] = a * x[i] + y[i];
}
```
Simple Fork/Join

• Some code must run at work-group granularity
  – Example: work items cooperate to compute output structure size
  – Atomic operation to allocate output must execute once

• Idiomatic solution
  – Barrier, then make work item #0 do the group-wide operation

```c
void subdividePolygon(...) {
    shared int numOutputPolygons = 0;

    // in parallel, every work item does
    atomic_add( numOutputPolygons, 1);
    barrier();

    Polygon* output = NULL;
    if( workItemID == 0 ) {
        output = allocateMemory( numOutputPolygons );
    }
    barrier();
    ...
}
```
Multiple Axes of Parallelism

• Deferred rendering with DX11 Compute Shader
  – Example from Johan Andersson (DICE)
  – 1000+ dynamic lights

• Multiple phases of execution
  – Work group responsible for a screen-space tile
  – Each phase exploits work items differently:
    – Phase 1: pixel-parallel computation of tile depth bounds
    – Phase 2: light-parallel test for intersection with tile
    – Phase 3: pixel-parallel accumulation of lighting

• Exploits producer-consumer locality between phases
## Terminology Decoder Ring

<table>
<thead>
<tr>
<th>Direct Compute</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>Pthreads+SSE</th>
<th>This talk</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>thread</td>
<td>work-item</td>
<td>SIMD lane</td>
<td>work-item</td>
</tr>
<tr>
<td>-</td>
<td>warp</td>
<td>-</td>
<td>thread</td>
<td>execution context</td>
</tr>
<tr>
<td>threadgroup</td>
<td>threadblock</td>
<td>Work-group</td>
<td>-</td>
<td>work-group</td>
</tr>
<tr>
<td>-</td>
<td>streaming multiprocessor</td>
<td>compute unit</td>
<td>core</td>
<td>core</td>
</tr>
<tr>
<td>-</td>
<td>grid</td>
<td>N-D range</td>
<td>-</td>
<td>Set of work-groups</td>
</tr>
</tbody>
</table>

*Slide 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)
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
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)
Summary of Concepts

• Abstraction
  – When a parallel programming model abstracts a HW resource, code written in that programming model scales across architectures with varying amounts of that resource

• Execution
  – Concurrency versus parallelism

• Synchronization
  – Where is user allowed to control scheduling?
“Ideal Parallel Programming Model”

• Combine the best of CPU and GPU programming models
  – Task systems are great for scheduling (from CPUs)
    – “Asynchronous function call” is easy to understand and use
    – Great load balancing and scalability (with cores, execution contexts)
  – SPMD programming is great for utilizing SIMD (from GPUs)
    – “Write sequential code that is instanced N times across N-wide SIMD”
    – Intuitive: only slightly different from sequential programming

• Why not just “launch tasks that run fine-grain SPMD code?”
  – The future on CPU and GPU?
Conclusions

• Task-, data- and pipeline-parallelism
  – Three proven approaches to scalability
  – Plentiful of concurrency with little exposed parallelism
  – Applicable to many problems in visual computing

• Current real-time rendering programming uses a mix of data-, task-, and pipeline-parallel programming (and conventional threads as means to an end)

• Current GPU compute models designed for data-parallelism but can be abused to implement all of these other models


References

• GPU-inspired compute languages
  – DX11 DirectCompute, OpenCL (CPU+GPU+...), CUDA

• Task systems (CPU and CPU+GPU+...)
  – Cilk, Thread Building Blocks (TBB), Grand Central Dispatch (GCD), ConcRT, Task Parallel Library, OpenCL (limited in 1.0)

• Conventional CPU thread programming
  – Pthreads

• GPU task systems and “persistent threads” (i.e., conventional thread programming on GPU)

• Additional input (concepts, terminology, patterns, etc)
  – Foley, “Parallel Programming for Graphics,”
    – Beyond Programmable Shading SIGGRAPH 2009
    – Beyond Programmable Shading CS448s Stanford course