Parallel Programming for Graphics

Aaron Lefohn
Advanced Rendering Technology (ART)
Intel

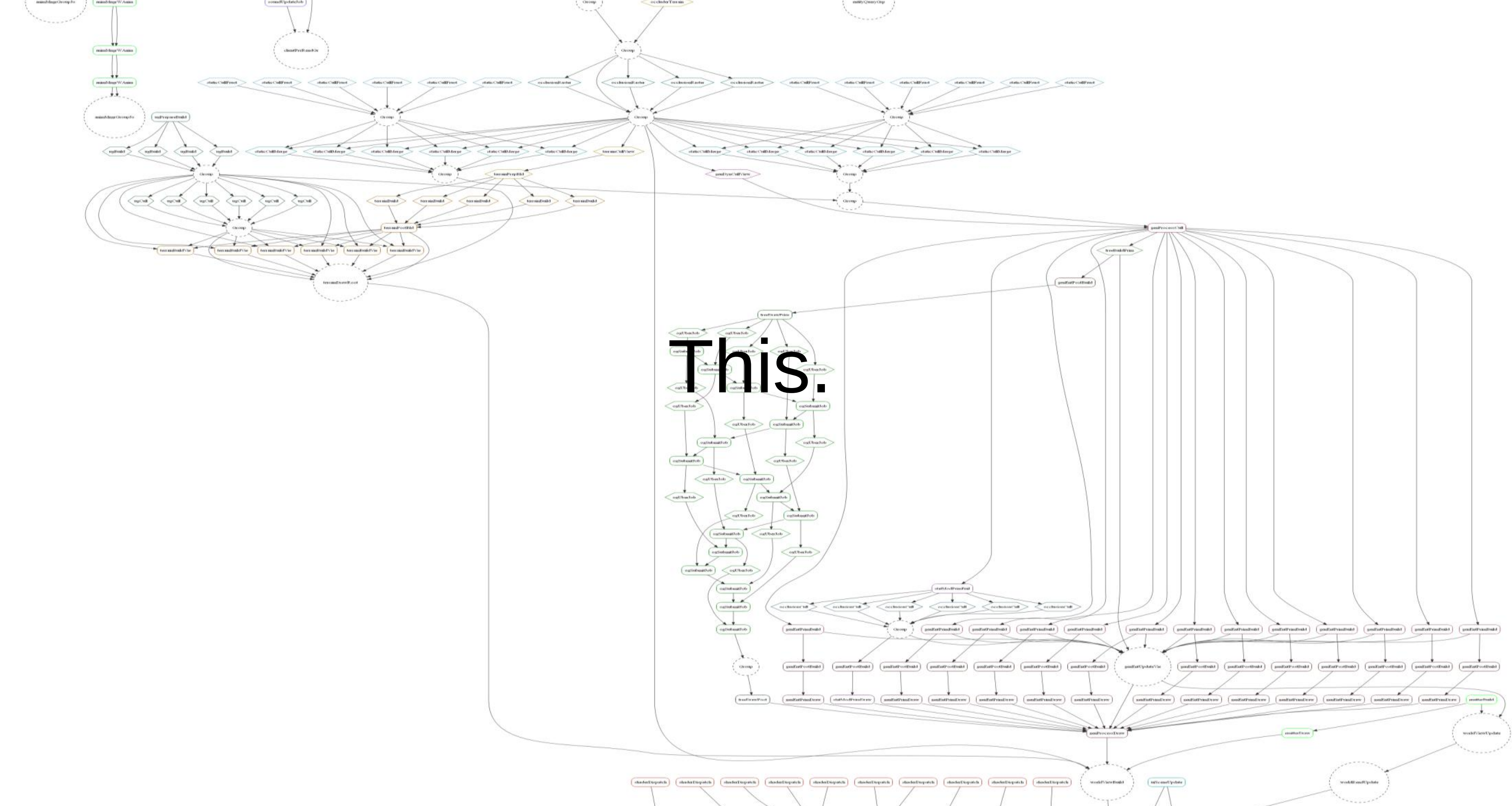
What's In This Talk?

- 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

- Discussion of strengths/weaknesses between the models

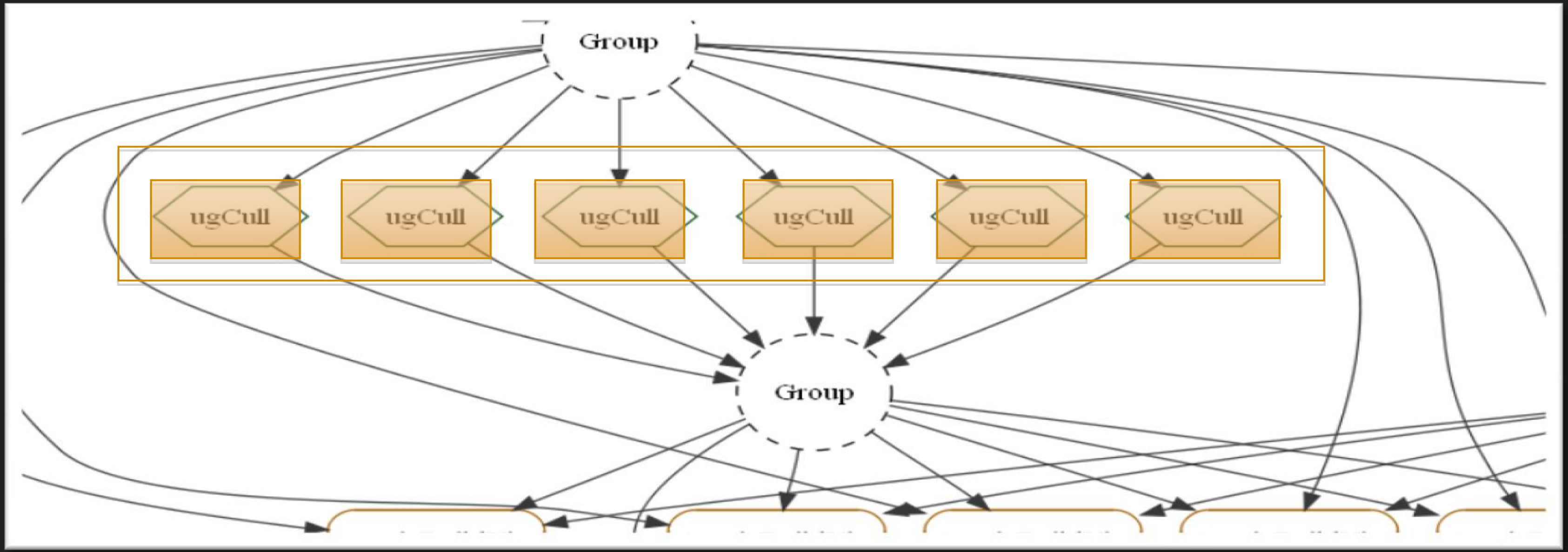
What Goes into a Game Frame? (2 years ago)

This.

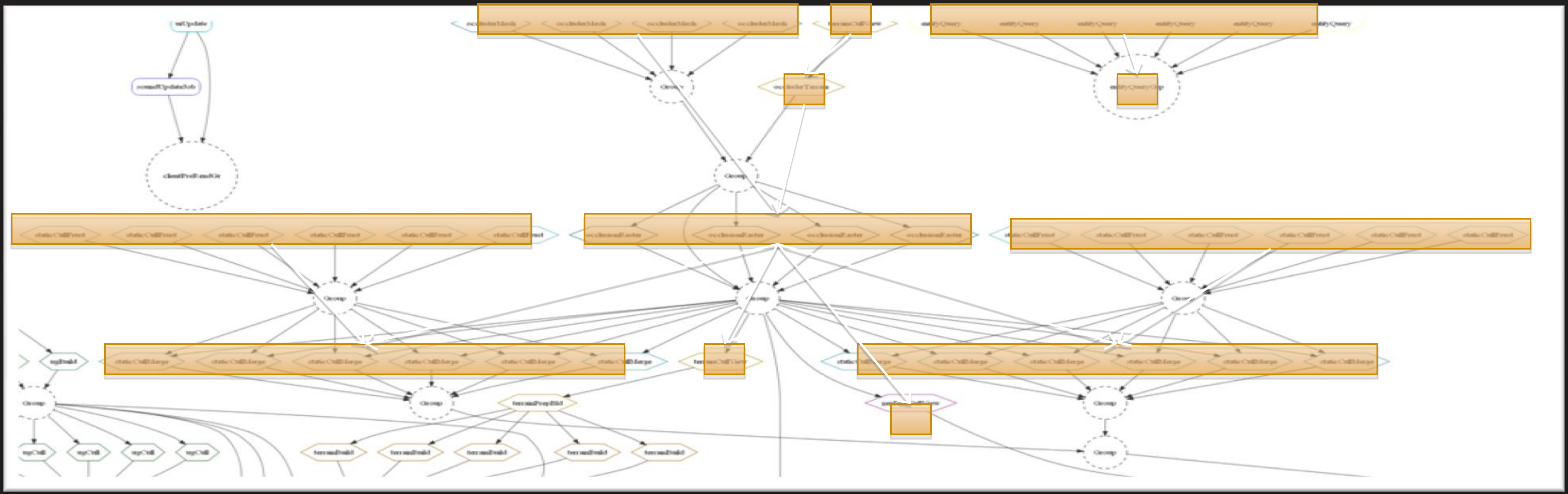


Computation graph for *Battlefied: Bad Company* provided by DICE

Data Parallelism



Task Parallelism



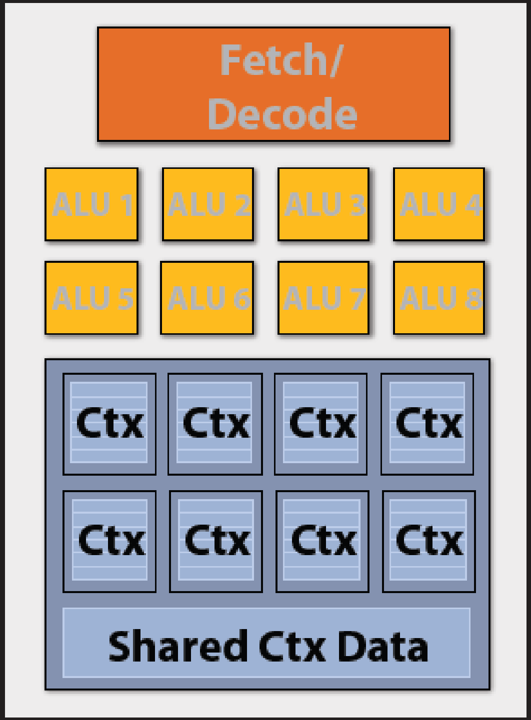
Graphics Pipelines

- Input Assembly
- Vertex Shading
- Primitive Setup
- Geometry Shading
- Rasterization
- Pixel Shading
- Output Merging

Pipeline Flow

Hardware Resources (from Kayvon's Talk)

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



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
- 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

- ***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

Pixel Shaders

- Execution
  - Concurrent execution of identical per-pixel tasks
  - Parallelism between four pixels in a 2x2 quad

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

- What synchronization is allowed?
  - Between draw calls

“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

Conventional Thread Parallelism (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

DirectX/OpenGL Rendering Pipeline

- 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?
  - 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

- DX11 DirectCompute
- OpenCL
- CUDA

- There are multiple possible usage models. We'll start with the "text book" hierarchical data-parallel usage model

Terminology Decoder Ring

Direct Compute	CUDA	OpenCL	Pthreads+SSE	This talk
thread	thread	work-item	SIMD lane	work-item
-	warp	-	thread	execution context
threadgroup	threadblock	Work-group	-	work-group
-	streaming multiprocessor	compute unit	core	core
-	grid	N-D range	-	Set of work-groups

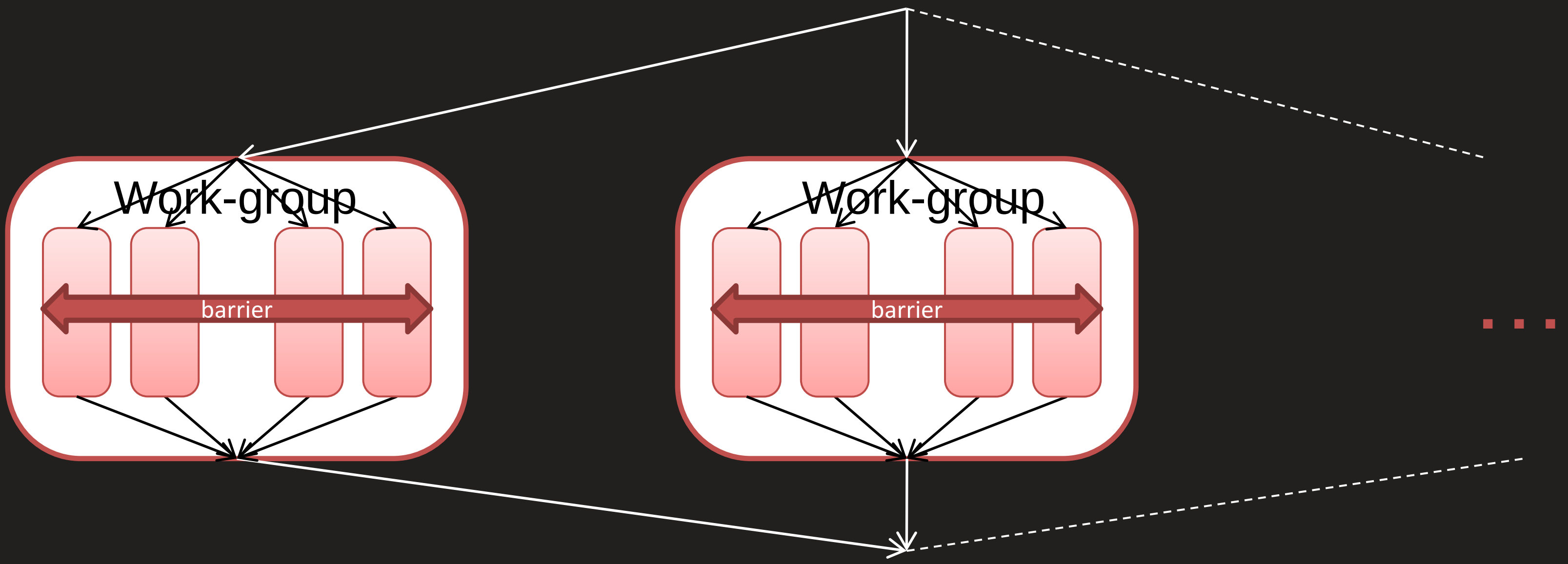
GPU Compute Languages

- 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)

GPU Compute Models



User Responsibilities: GPU Compute

- User manually maps work-item index to problem domain

- User controls size and number of work-groups
  - Selecting these parameters is complex combination of architecture- and task-specific considerations

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)

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?

Conclusions

- 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
- Look for uses of these different models throughout the rest of the course

Acknowledgements

- Tim Foley, Intel
- Kayvon Fatahalian, Stanford
- Mike Houston, AMD
- Tim Mattson and Andrew Lauritzen, Intel
- Craig Kolb and Matt Pharr

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)
  - Aila et al, "Understanding the Efficiency of Ray Traversal on GPUs," High Performance Graphics 2009
  - Tzeng et al, "Task Management for Irregular-Parallel Workloads on the GPU," High Performance Graphics 2010
  - Parker et al, "OptiX: A General Purpose Ray Tracing Engine," SIGGRAPH 2010

- Additional input (concepts, terminology, patterns, etc)
  - Foley, "Parallel Programming for Graphics,"
    - Beyond Programmable Shading SIGGRAPH 2009
    - Beyond Programmable Shading CS448s Stanford course
  - Fatahalian, "Running Code at a Teraflop: How a GPU Shader Core Works," Beyond Programmable Shading SIGGRAPH 2009-2010
  - Keutzer et al, "A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL projects, " ParaPLoP 2010

Questions?

Course web page and slides:

http://bps10.idav.ucdavis.edu