The Interactive Graphics Toolbox

• Current GPUs
  - 3D rendering pipeline
  - Data-parallelism

• Larrabee adds new tools to the toolbox
  - Task parallelism
  - Nested task- and data-parallelism
  - Sequential code
  - ...

Beyond Programmable Shading: Fundamentals
New Parallel Tools for Graphics?

- Next-generation rendering algorithms and pipelines
  - Curved surfaces
  - Physically-inspired light transport
  - High-detail textures,
  - Improved sampling strategies, ...

- Require **combining many styles of parallel algorithms**, interspersed with efficient sequential code

- Current GPUs hard-code all but the data-parallel execution units in fixed-function hardware

- **Larrabee makes (nearly) all of interactive graphics programmable** by efficiently executing a wide range of parallel and sequential programming paradigms
• **Current GPUs**
  - Efficiently execute only wide data-parallel computation
  - Cannot submit work to self (reliant on CPU)
• Larrabee
  - Efficiently executes *braided parallelism* that intermixes data- and task-parallelism with sequential code
  - Can submit work to itself
Programming Larrabee

• Larrabee can support industry-standard GPU programming models such as
  - Direct3D, DX ComputeShader
  - OpenGL, OpenCL

• Larrabee Native
  - Access full power and flexibility of LRB architecture
Larrabee Native Programming

- Two tightly paired binaries
  - Host CPU and Larrabee

- Host CPU code
  - Load Larrabee binary
  - Data transfers & message passing to/from Larrabee

- Larrabee code
  - Full C/C++ compiler + parallel languages/libraries
  - Data transfers & message passing to/from Host
  - Access fixed-function graphics HW
Data Parallelism on Larrabee
**Data-Parallelism on Larrabee**

- **Data-parallelism is powerful tool for right problem**
  - Run same program over large number of elements
  - Easy explanation
    - “Write code looking out from within a parallel ‘for’ loop”

- **Pros**
  - Lots of independent work to hide long-latency memory access
  - Implicit (magic) use of SIMD

- **Cons**
  - Small amount of on-chip memory per-program invocation
  - Difficult and bandwidth-intensive to build data structures
  - Not all algorithms a good fit
Data-Parallelism on Larrabee

• Key ideas of data parallel programming
  - Define an array (grid) of parallel program invocations
  - Define groups within the grid that can share on-chip memory

• Mapping program invocations on Larrabee
  - Strand: a program invocation that runs in one SIMD lane
  - Fiber: a SW-managed context that runs 16-64 strands
  - Thread: a HW-managed context that switches among 2-10 fibers in order to cover long latencies (e.g. texture filtering)
  - Core: independent processor that runs 1-4 threads, executing all strands in a group (their shared memory is in the core’s L2 cache)
Data-Parallelism on Larrabee

16-wide vector unit

...
Data-Parallelism on Larrabee

**Fiber**: SW-managed context (hides long predictable latencies)

16-wide vector unit

... More Fibers running (typically 2-10, depending on latency to cover)
**Thread**: HW-managed context (hide short unpredictable latencies)

**Fiber**: SW-managed context (hides long predictable latencies)

16-wide vector unit

... More Fibers running (typically 2-10, depending on latency to cover)

... More Threads (up to 4 per core, share memory via L1 & L2 caches)
Data-Parallelism on Larrabee

**Cores:** Each runs multiple threads

**Thread:** HW-managed context (hide short unpredictable latencies)

**Fiber:** SW-managed context (hides long predictable latencies)

More Fibers running (typically 2-10, depending on latency to cover)

More Threads (up to 4 per core, share memory via L1 & L2 caches)
Beyond Data Parallelism

• Larrabee executes current GPU workloads

  and

• Larrabee is a many-core x86 architecture...
Task Parallelism on Larrabee
Task Parallelism on Larrabee

• To run efficiently, current GPUs require executing 100s-1000s of program invocations per core

• **Larrabee can run efficiently with four or fewer program invocations per core**
  - If algorithm hides latency using r/w caches and asynchronous memory operations
  - And explicitly uses SIMD units
    • Auto-vectorizing compiler, C++ classes, vector intrinsics, ...

• **Benefit**
  - Much larger on-chip storage per program invocation
    • E.g., > 64 kB instead of 1/8th kB
Task: SW-managed context

Scalar code
Auto-vectorizing compiler, C++ vector classes, or explicit vector code
16-wide vector unit

... More tasks enqueued, waiting to run...
Beyond Programmable Shading: Fundamentals

Task-Parallelism on Larrabee

Thread: HW-managed context (hide short unpredictable latencies)

Task: SW-managed context

Scalar code
Auto-vectorizing compiler, C++ vector classes, or explicit vector code
16-wide vector unit

More tasks enqueued, waiting to run

More Threads (up to 4 per core, share memory via L1 & L2 caches)
Task-Parallelism on Larrabee

Cores: Each runs multiple threads

Thread: HW-managed context (hide short unpredictable latencies)

Task: SW-managed context

Scalar code

Auto-vectorizing compiler, C++ vector classes, or explicit vector code

16-wide vector unit

More tasks enqueued, waiting to run

More Threads (up to 4 per core, share memory via L1 & L2 caches)
What is a Task?

• A task:
  - Think of task as an asynchronous function call
  - Implemented as “lightweight, user-space cooperative execution entity that virtualizes HW threads”
  - Successful scalable parallel programming model for Xbox 360*, PlayStation 3*, multi-core CPUs, and Larrabee
    • Tasks automatically scale with increasing core count
    • Spawn many more tasks than cores
    • 100s - 1000s of tasks per frame is common

• Example task systems
  - Apple’s Grand Central Dispatch*
  - Microsoft’s Task Parallel Library*
  - Intel’s Thread Building Blocks
A Task is not a POSIX thread

• Threads are HW resource that execute tasks
  - Tasks scheduled via per-thread workqueue
  - Cooperative programming model

• Task is much finer-grain than thread
  - Function call granularity
  - Ideal task size is “as small as possible while keeping scheduling overhead low % of runtime”

• Dependencies between tasks clearer than between threads
  - “Launch this task when these N tasks complete”
  - Small scope limits amount of shared data accesses
  - Natural to express data locks at task scope
When to Use Tasks

• Design algorithms that combine
  - Asynchronous function calls (tasks) and
  - Parallel loops (data-parallelism)

• Example use cases for tasks
  - Implementing graphics pipelines requires both task- and data-parallelism
    • Traditional GPUs support only fixed-function tasking
  - Algorithms that require large on-chip storage and have known memory access patterns
  - Algorithms where data-parallel formulation is painful and/or burns bandwidth
  - “When you just don’t have 1000s of identical things to do”
When Not to Use Tasks

- When data-parallelism is natural fit for problem
- Have 1000s of identical things to do and algorithm has small per-computation storage requirements
- Want implicit SIMD utilization from strands
Task + Data Parallel Example

- **Resolution-Matched Shadow Maps**
  - ACM Transactions on Graphics 2007 paper
  - Build quadtree shadow map every frame

- **Original algorithm (data-parallel only)**
  - Build quadtree with data-parallel algorithms
  - Scan, sort, compact, scan, ...

- **Optimized algorithm (task + data-parallel)**
  - Build quadtree with task-parallel algorithm
  - Replace data-parallel sort/scan/compact with tasks that build meta-structure entirely in on-chip memory
  - Greater than 20x bandwidth reduction over original algorithm

- **General theme**
  - Task-parallel algorithm captured on-chip locality that data-parallel algorithm could not
Larrabee Can Submit Work to Itself
• Larrabee can submit draw calls to itself

• Larrabee can implement graphics pipelines in software (current GPUs use fixed-function HW to spawn fixed-form work)
  - Ex 1: Draw call spawns vertex processing
  - Ex 2: Rasterization spawns fragment processing
  - Ex 3: Fragment processing spawns rays

• Larrabee supports nested parallelism
  - Varying amounts of parallelism
  - Efficient tree traversals, fork/join, reductions, ...

• Larrabee communicates with CPUs at higher abstraction level than current GPUs
Larrabee Can Execute Sequential Code
Sequential Code

• What!? This is graphics!
  - Sequential code ties together parallel execution
  - Sequential code makes decisions
  - “Parallel algorithms are spawned from sequential code”

• Executing sequential code on Larrabee
  - Enables users to write more complex algorithms and complex data structures without involving host CPUs
  - Enables host and Larrabee to communicate at a higher level of abstraction
Tying it All Together

- Imagine doing the following *entirely on Larrabee*
  - Receive scene update from host CPUs
  - Traverse scene database
  - Spawn tasks to cull, animate, ..., draw models
  - Compute dynamic global illumination approximation using braided parallel algorithm (task, data, nested, sequential)
  - Spawn final draw calls

- This is just one example—see Matt Pharr’s talk this afternoon for more details, examples, and ideas
“Choose the Right Tool”

- Use the 3D rendering pipeline
  - How you always have (or create your own)

- Use data parallelism
  - Parallel ‘for’ loop
  - Hide latency by switching to work from other elements

- Use task parallelism
  - Asynchronous function call
  - Capture on-chip locality with large per-task memory
  - Hide latency with in-task reuse and asynchronous memory ops

- Use nested parallelism
  - Dynamically determine amount of work/parallelism

- Use sequential code
  - Tie it all together
  - Communicate with host at higher level of abstraction
Conclusions

• Larrabee adds new tools to the interactive graphics programmer’s toolbox

• Building graphics pipelines and rendering algorithms that deliver realistic imagery requires a combining many styles of parallel algorithms with efficient sequential code

• Larrabee makes (nearly) all of interactive graphics programmable by efficiently executing a wide range of parallel and sequential programming paradigms

• “Pick the right tool for the right job”
Acknowledgements

• Doug Carmean, Eric Sprangle, Tom Forsyth, Mike Abrash, Larry Seiler, Stephen Junkins, ...

• Matt Pharr, Craig Kolb, Tim Foley, Andrew Lauritzen, Tim Mattson, ...

• Kayvon Fatahalian
Reference