
SIGGRAPHASIA2008
NEW HORIZONS

Parallel Programming on Larrabee

Tim Foley
Intel Corp

SIGGRAPHASIA2008
NEW HORIZONS

Motivation

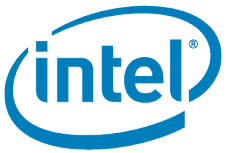
- This morning we talked about abstractions
  - A mental model for GPU architectures
  - Parallel programming models
  - Particular tools and APIs
- This talk is about implementation
  - How these abstractions map to Larrabee

Outline

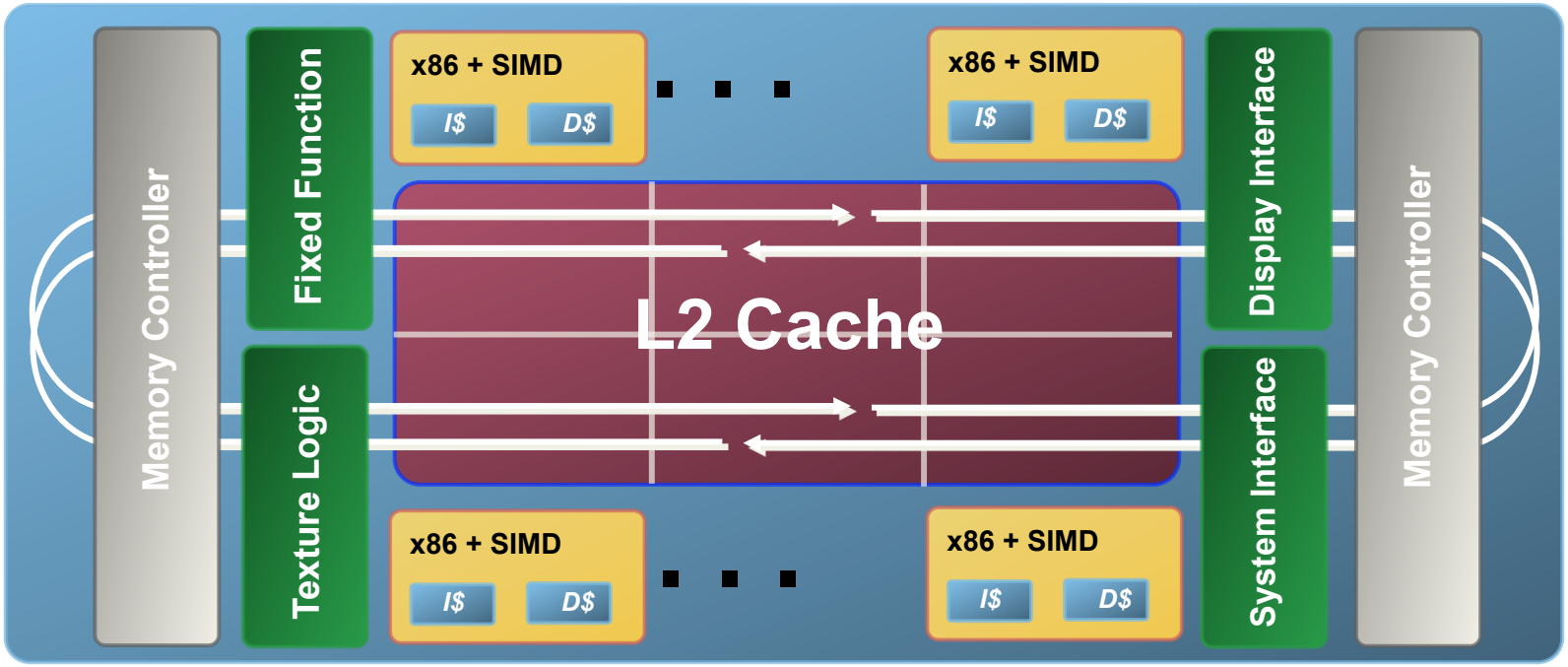
- Whirlwind tour of
  - Larrabee architecture
  - Low-level programming models
- Case studies
  - Efficient execution of data-parallel kernels
  - Scalable work distribution with user-space tasks

- Next talk: using these for graphical innovation

Larrabee Architecture



Obligatory block diagram



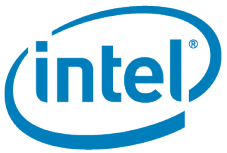
- Lots of x86 cores with 16-wide SIMD
- Fully coherent caches
- Wide ring bus
- Fixed-function texture hardware

Larrabee cores

- Fully coherent L1 and L2 caches
- Fast bidirectional ring
  - L2 caches can share data with each other
- Short in-order pipeline
  - No latency on scalar ops, low latency on vector
  - Cheap branch mispredict and cache miss
- 4-way SMT
  - Designed to hide L1 miss

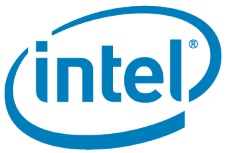
SIMD extensions

- 16-wide float32/int32, 8-wide float64
  - Ternary ops
  - Multiply-add
  - One source from memory
- All math at 32- or 64-bit
  - Almost free conversion to float16, {int,norm}{8,16}
  - Cheap conversion to other graphics formats
    - sRGB, float11:11:10, unorm10:10:10:2



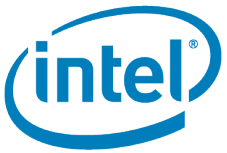
SIMD extensions, continued

- Fast predication on every instruction
  - 16-bit predication registers – per-lane enable
- Gather/scatter instructions
  - Read/write 16 values from/to 16 different offset
- Enable high utilization for shader-style code
  - We will see more on this later

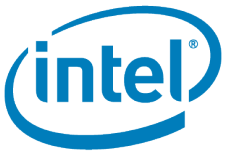


For more info…

- “Larrabee: A Many-Core x86 Architecture for Visual Computing”
  - Seiler et al.
  - SIGGRAPH 2008



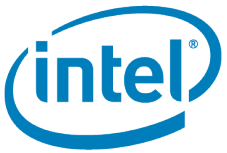
Low-Level Programming Models



Larrabee is just x86

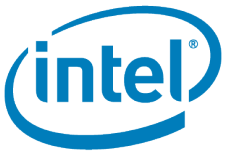
- Cores can run C/C++
- The **real** C/C++
  - Recursion, function pointers, virtual functions
  - printf(), malloc(), pthreads
- Plus intrinsics for the SIMD extensions

- Have fun! You are on your own!



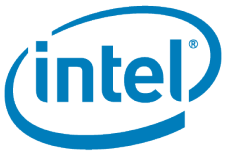
Just kidding

- Larrabee is still a GPU architecture
  - Will support your favorite graphics APIs
- Some users need easy-to-program solution
  - We know this – and we are ready
- High-level models can shield you from:
  - SIMD, threads, prefetch, caches
- This isn't what I'm talking about today…



Peeking under the covers

- When the high-level model doesn't fit
  - Larrabee lets you talk to the lower levels
- Don't want data-parallel?
  - We don't force you
- Know how to code for SIMD, prefetch?
  - Hand-tune select inner loops
  - Use high-level tools for the rest



“Once a month” code

- Nobody would printf() in an inner loop
  - Unless they are debugging
- Nobody would fread() on a GPU
  - Unless they are demand-loading compressed geometry
- Sometimes ease-of-use trumps performance

Mix and match

- Larrabee can support industry-standard APIs
  - DirectX Graphics, Compute Shader
  - OpenGL, OpenCL
- Plus Larrabee Native applications
  - Access the full power and flexibility of the architecture
  - Use standard C/C++ compilers and tools

Larrabee Native programming

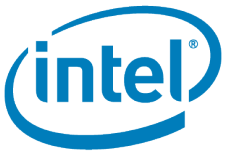
- Two tightly paired binaries
  - Host CPU and Larrabee
- Host CPU code
  - Load Larrabee binary
  - Data transfer, message passing to/from Larrabee
- Larrabee code
  - Full C/C++ compiler + parallel languages / APIs
  - Data transfer, message passing to/from host
  - Access fixed-function HW units

The rest of this talk

- Case studies
  - How we implement some high-level idioms
  - Making scalable and efficient Larrabee apps

- Nothing up our sleeves
  - We use these to build efficient graphics pipelines
  - These techniques may help as you build your own native graphics algorithms

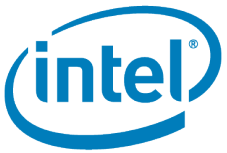
Data Parallelism on Larrabee



Quick review

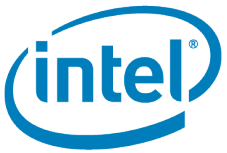
- Data-parallel basics
  - Launch N independent work items
    - N determined by domain of computation (NDRange, …)
  - Each running the same kernel
    - Scalar per-item program
  - Groups of work items may share storage

- Design goals
  - High SIMD utilization
  - Hide long-latency memory operations (texture)
  - Exploit locality of group-local storage

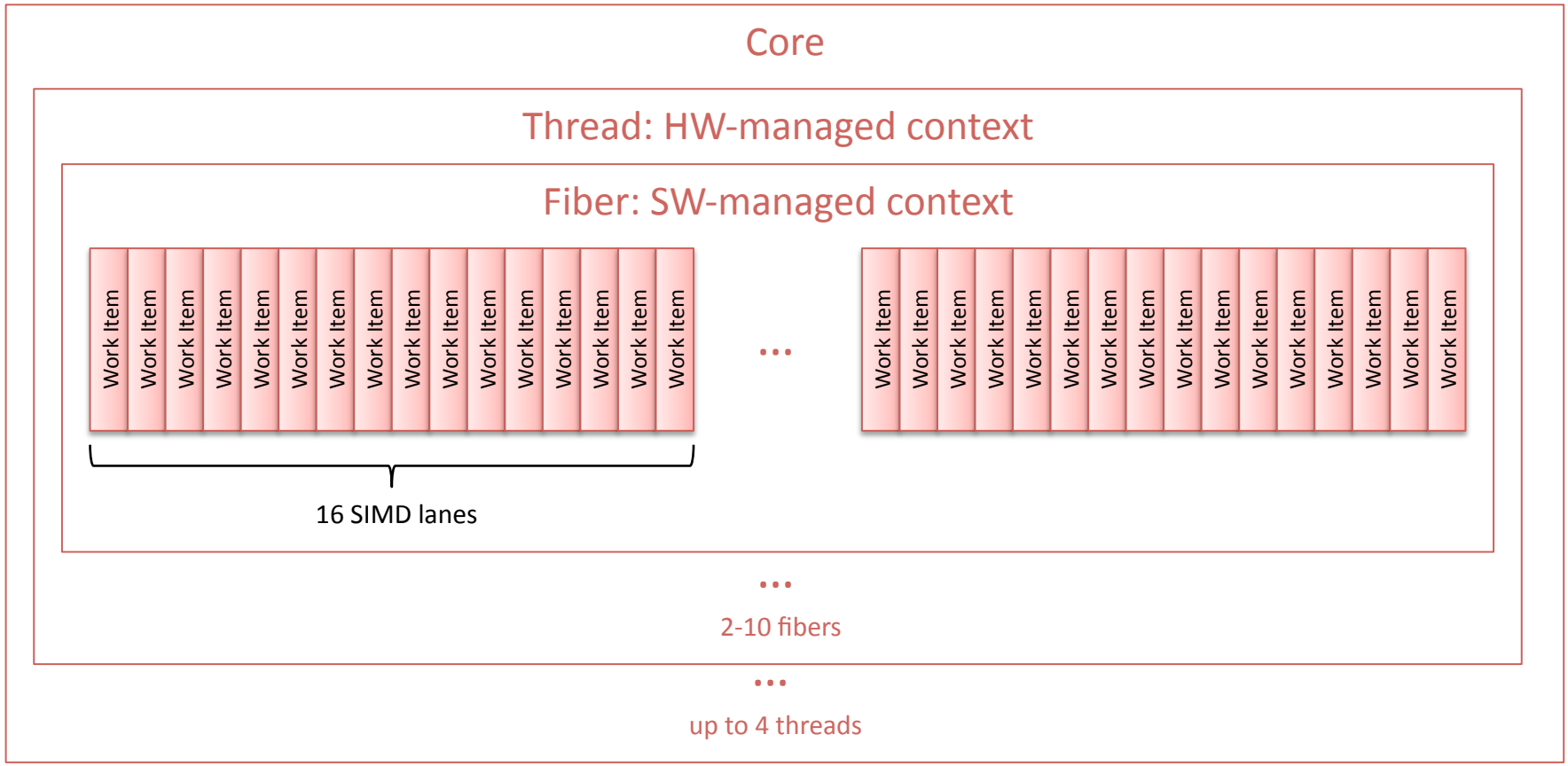


Mapping to Larrabee

- Work Item
  - Logical invocation of kernel program
  - Runs in one SIMD lane
- Fiber
  - SW-managed context. Runs 16-64 work items
- Thread
  - HW-managed context
  - Switches between 2-10 fibers to cover latencies
- Core
  - Independent processor that runs 1-4 threads

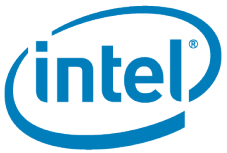


Data-parallel mapping



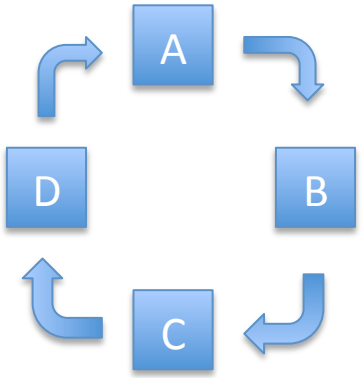
Generating fiber code

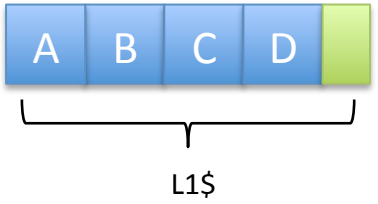
- Map 16 work items to the 16 SIMD lanes
  - Most ops map to vector equivalent
    - Add becomes 16-wide vector add, …
  - Array reference becomes scatter/gather
- Control flow uses predication
  - "Divergent" branch – execute both paths
  - Use masks registers to predicate execution
- Larrabee extremely friendly to compilers
  - Easy to create "shader"-style data-parallel languages



Hiding texture latency

- Fiber starts asynchronous texture fetch
  - Then switches to another fiber
  - Ultra-lightweight user-space context switch
- 2-10 fibers per HW thread
  - Each fiber logically assigned a slice of L1
  - State still hot when fiber resumes
- Groups of work items easy to support
  - Run whole group on same core
  - Group-shared data stays warm in cache



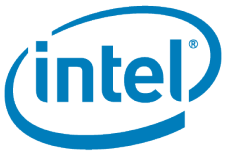


Workload distribution

- Want to load-balance data-parallel kernels
  - … and rendering
  - … and geometry synthesis
  - … and physics
  - … and scene traversal / culling
  - All at the same time!

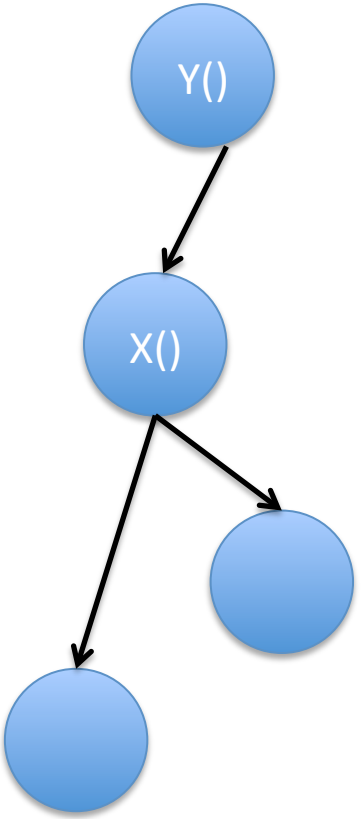
- The solution: task-parallel programming

Task Parallelism on Larrabee



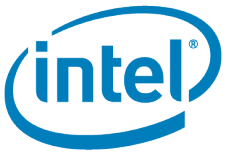
What is a Task?

- Think of it as an asynchronous function call
  - "Do X at some point in the future"
  - Optionally "… after Y is done"
- Often implemented in user space
  - One worker thread per HW thread
  - Pull from incoming queue of requests
    - Per-thread queues decrease contention
    - Use "work stealing" for load balancing [Cilk]



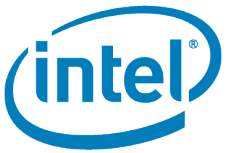
Why Tasks?

- Proven, scalable parallel programming model
  - Xbox 360, PlayStation 3, multicore CPUs, Larrabee
- Scales well with increasing core count
- Copes with heterogeneous workloads
  - Animation, rendering, physics, AI, GI, …



Example task systems

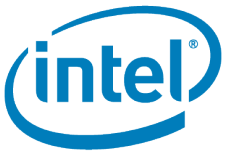
- Apple's Grand Central Dispatch
- Microsoft's Task Parallel Library
- Intel's Thread Building Blocks
- OpenMP 3.0
- OpenCL 1.0



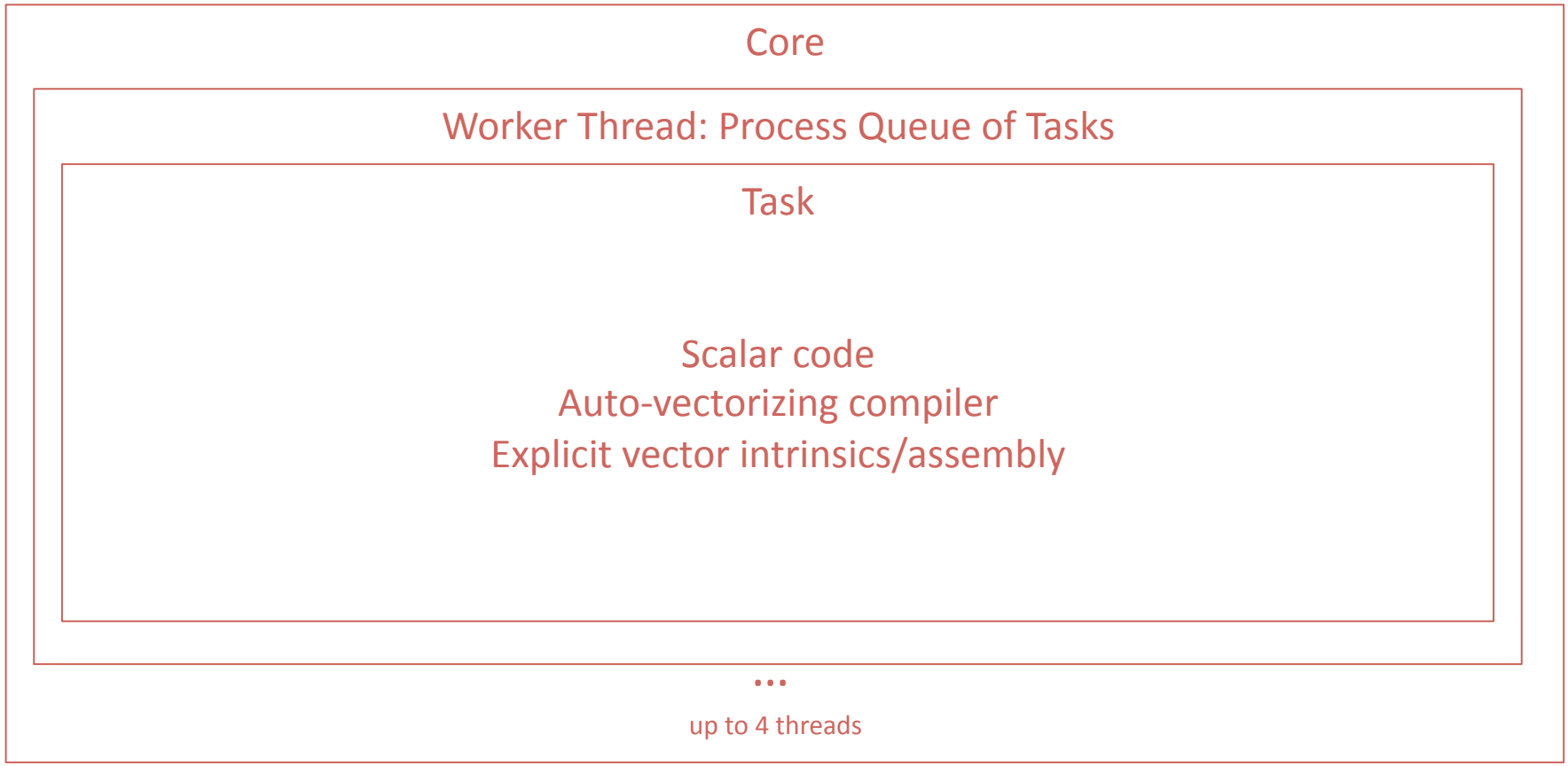
When to use Tasks

- For efficiency, some GPU programming models require 1000s of work items
  - Some problems aren't that fine-grained

- Larrabee can run efficiently with 1-4 tasks per core
  - While a task runs it "owns" the HW thread
    - Get > 64K of scratch space instead of < 1K
  - In return, take responsibility for
    - Memory locality / prefetch
    - SIMD utilization



Task-parallel mapping



Examples of tasks

- Scene traversal and culling
- Procedural geometry synthesis
- Physics contact group solve

- Data-parallel work group
  - Distribute across cores using task system
  - Exploit core resources with fibering/SIMD

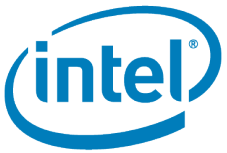
Key point

- From the perspective of Larrabee, a data-parallel work-group is “just another task”
- Task “owns” the resources of a core when it runs
- Fibering is one way to use those resources
  - Automatic and implicit SIMD utilization
  - Automatic hiding of long latencies
  - Efficient sharing data through cache hierarchy

Nested tasks

- Can tasks launch other tasks?
  - OpenCL 1.0 – no
  - TBB – yes
  - …
- Larrabee can submit work to itself!
  - Important benefit of Larrabee Native programming
  - Culling tasks can spawn drawing tasks
  - Fragment-shading tasks can spawn ray-tracing tasks

Summary

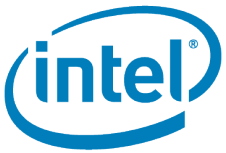


Larrabee architecture

- Larrabee is a manycore x86 architecture
  - Optimized for graphics workloads
- When using C/C++, the sky is the limit
  - Real x86 architecture
  - Modern memory system, caches
  - Powerful SIMD extensions

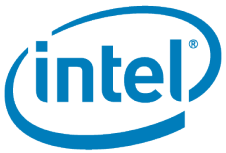
Proven Programming Models

- Data parallelism
  - Easy (implicit) way to get high SIMD utilization
  - Effectively exploit asynchronous texture HW
- Task parallelism
  - Load-balance heterogeneous workloads
  - Scalable approach to “whole program” parallelism
- Next: Using these tools for next-gen graphics!



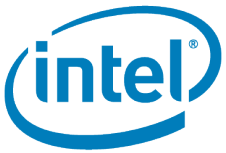
Acknowledgements

- Tom Forsyth
- Aaron Lefohn
- Matt Pharr



Questions?

tim.foley@intel.com




SIGGRAPHASIA2008
NEW HORIZONS