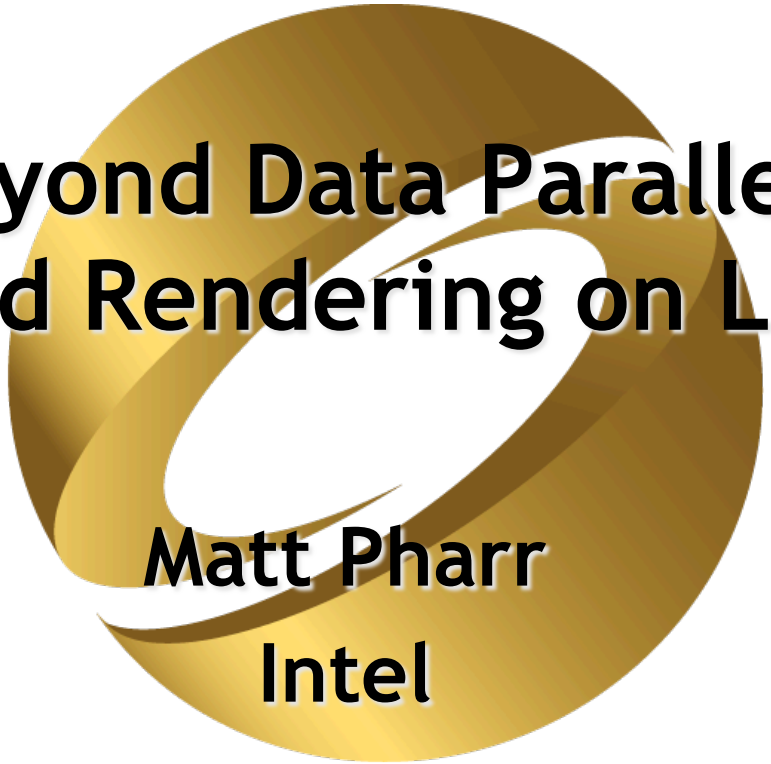
Beyond Data Parallel: Advanced Rendering on Larrabee



Matt Pharr
Intel

SIGGRAPH2008

Beyond Programmable Shading: In Action

The big picture

- **Graphics hardware continues to make huge computational resources available to developers**
  - With increasingly flexible programming models
  - From fixed-function to data-parallel to task-parallel and fully general
- **The center of rendering innovation has moved from h/w designers to s/w developers**
  - The coming years will see great innovation in interactive rendering

Larrabee flexibility

- Peak performance is achievable for both data-parallel and thread-parallel computation
- Large coherent caches minimize impact of irregularity when present
- Flexibility (with performance) is sufficient to implement the DX/OGL graphics pipeline entirely in software
  - (Modulo texture sampling)

Implications of LRB flexibility

- **Big win for innovators in rendering (both developers and researchers)**
  - Many fewer limitations than (current) GPUs
- **Great new ideas that depend on h/w modifications for new graphics will be far less common**
  - Stochastic rasterization, logarithmic rasterization, programmable culling unit, a-buffer, k-buffer, programmable blending, irregular z buffer, new compressed texture formats, REYES, …

Three main themes for today

- Dynamic data structures
- Dynamic computation
- Customized graphics pipelines

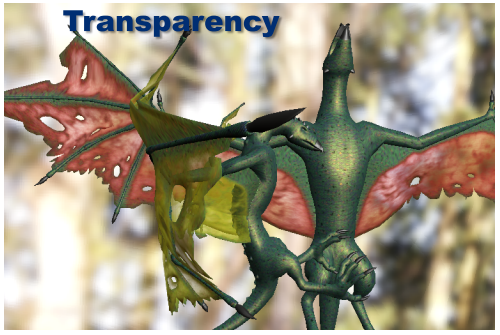
Dynamic data structures

Dynamic data structures

- **Data structures and algorithms—to the pixel level—are the frontier of advanced interactive rendering:**
  - Irregular z-buffer: linked list of samples per pixel
  - A-buffer: linked list of (transparent) fragments
  - Disk tree hierarchy for dynamic ambient occlusion
  - kd-trees for fast geometric query and ray tracing
  - RMSMs: sparse quadtree of shadowmap pages

Dynamic data structures

- **In complex dynamic environments, these data structures must be built on the fly**
  - Effects are either impossible or too inefficient without them
- **Data-parallel can be used to build some (but not all) of them efficiently**



Dynamic data structure taxonomy

- **Fine-grained adaptive data structures**
  - Small per pixel: a-buffer, multi-layer z buffer, …
- **Demand-generated data structures**
  - Sparse textures, procedural geometry, …
- **Data structures that encode analysis of large amounts of dynamic data**
  - Sparse shadowmap quadtree
  - BVH of visible samples in deep framebuffer

Fine-grained adaptive data structures

- **Approach 1: static pre-allocation**
  - Pre-allocate memory per pixel
  - Build data structure in pixel shader
    - Requires framebuffer read-modify-write in pixel shader
    - LRB software graphics pipeline can support this
      - Fine-grained execution (16 pixel qquad) greatly reduces impact of different computation in different pixels
      - All memory access is out of L2$, thanks to tiling
  - Problems if not enough memory in a pixel…

Fine-grained adaptive data structures

- **Approach 2: two-pass static allocation**
  - Leverage flexibility of s/w graphics pipeline
  - First pass over qquads: determine storage needs from qquads coming into tile
  - Allocate memory, once per tile
  - Second pass over qquads: rasterize and shade, fill allocated memory
  - Like multi-pass rendering, but with only one geometry submission
    - Save front-end processing
    - Reduce driver draw-call overhead

Fine-grained adaptive data structures

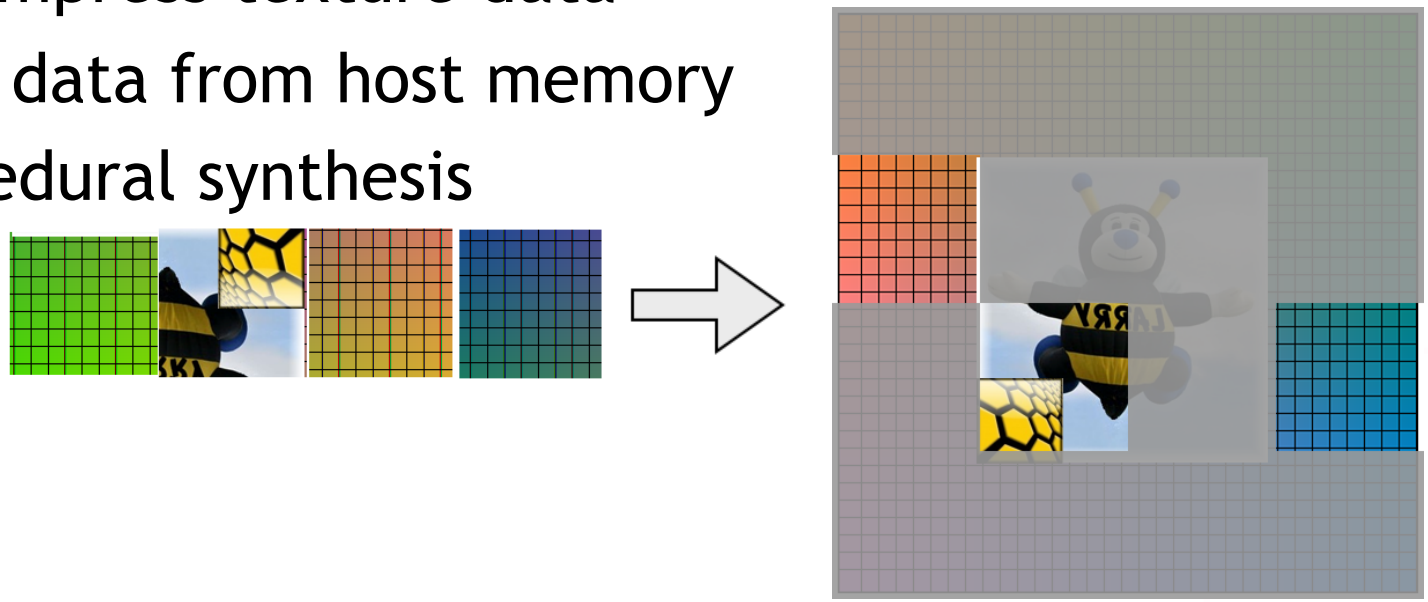
- **Approach 3: dynamic allocation**
  - Dynamically allocate memory per pixel (or qquad)
    - Not malloc()!
    - Small per-tile pool can be accessed with minimal locking from a single core
      - Very efficient locks through L1$ in LRB
    - Or, per-h/w thread pool accessed with no locking
- **Which is best?**
  - No single answer; different apps will want to make different trade-offs
  - Freedom of choice is a good thing

Demand-generated data structures

- **A small subset of scene data will typically be needed per frame**
  - Not all texture is visible, not all MIP levels used
  - Geometry occluded by closer geometry
- **It's wasteful to submit too much excess scene data to rendering pipeline**
  - Memory and computation are both precious
- **Classic issue in offline rendering**
- **Increasingly important in interactive**
  - id Megatexture, …

Demand-generated data structures

- Allocate unmapped large virtual memory region
- Service faults as rendering pipeline progresses
  - Decompress texture data
  - Load data from host memory
  - Procedural synthesis



Demand-generated data structures

- **LRB provides two key mechanisms**
  - General page faults from MMU
  - Fixed-function texture sampler with “soft” faults
- **LRB execution model flexibility is also key**
  - Procedural geometry, decompression algorithms, etc., are generally better expressed in a “narrow” task parallel, not “wide” data parallel manner
    - Easier to decode a JPEG with a few strands than with 1000s of strands
    - More on this in the next section…

Boiling down big data

- **Resolution-matched shadow maps**
  - Render out per-pixel “request buffer” of shadow map sample needed at each pixel
  - Build data structure to record required pages
  - Render shadow pages, use them
- **Reduction is inelegantly expressed in data-parallel style**
  - Easier mapping to lower degrees of required parallelism on LRB
  - Data structure can live in cache on LRB
    - Substantial bandwidth savings

Boiling down big data

- **Deep framebuffer bounding volume hierarchy (BVH)**
  - Given deep framebuffer, build BVH of camera-space positions to be shaded
  - With traditional deep fb, rasterize based on light shapes to kick off shading
  - Can search more efficiently, pack computation into 16-wide vector blocks with a better data structure
- **LRB architecture advantages**
  - Ease of building data structure at runtime
  - Scatter/gather instructions tightly pack work into LRB vectors

Dynamic Computation

Dynamic Computation

- **Efficient dynamic computation is a prerequisite to rendering of rich and irregular scenes**
  - Different execution paths at different pixels, …
- **Amount of available (or desired) parallelism varies greatly in the course of rendering**
- **Brute force burns compute *and* bandwidth**
  - LRB can achieve peak performance with **both** wide data-parallel and narrow CPU-style computation
  - Much less downside to irregularity vs. today's GPUs

Computation styles (wide / GPU)

- **Many strands per execution unit (100s+)**
- **When strand reads memory, lightweight switch to another strand**
- **Given enough strands, memory latency is fully hidden**
  - Advantage: memory latency is a bottleneck for many computations on CPU
    - Hiding it keeps execution units busy

Computation styles (wide / GPU)

- **Disadvantages (sometimes):**
  - Caches have limited utility
    - Useful for horizontal reuse (across strands)
    - Not useful for vertical reuse (within single strands)
    - Large thread count causes useful data to be evicted (from perspective of a single strand)
  - Programmer must provide many strands
    - Easy to do for vertex, pixel processing
      - Many other rendering tasks, too
    - Less so for more general computation
      - Many-core is hard, now 100s of threads per many-core?

Computation mix: wide vs. narrow

- **Narrow (traditional CPU)**
  - Few (1-2) threads per execution unit
  - Cache is your only help with main memory latency
  - If you miss L1/L2, other threads may help
  - If you miss to main memory, you probably stall
    - Not enough threads to hide latency
    - Main memory latency is increasing

Spanning both styles on LRB

- **Narrow (~CPU)**
  - Do the usual thing, run a few threads/tasks per core
  - Can achieve peak h/w performance
    - As long as you don't have too many cache misses
    - Don't have automatic vector-unit filling
- **Wide (~GPU)**
  - Run many fibers, fine-grained switching
  - Can achieve peak h/w performance
    - As long as you have enough threads to hide latency
    - Vector unit utilization very good (for free)

Spanning both styles on LRB

- **Dynamic variation on Larrabee**
  - Launching a new fiber is an efficient user-space operation (10s of cycles)
  - As computation progresses, can ramp up fibers when memory latency hiding is most important, ramp down when cache efficiency is preferable
- **Example: spawning rays in pixel shader, Stanford GRAMPS**

Variable parallelism as you go

- **Traversing tree data structures is important**
  - e.g. ambient occlusion disk trees, kd-trees, lightcuts, shadow quad-trees, …
- **If many points traverse the same top level nodes, computation and b/w are wasted**
- **Solution: start traversal with few fibers representing many points until divergence, then launch additional fibers**
  - Clean mental model for developer

Adaptive processing is critical

- **Consider e.g. image-space global illumination**
  - In some tiles, a few smooth surfaces
    - Compute sparse GI solution, propagate to visible points
  - In other tiles, much more geometric complexity
    - Compute denser solution, at more points

Adaptive processing and Larrabee

- **Dynamic processing is a good fit**
  - Data-parallel is a good fit to compute visible points and their properties
  - Analysis better expressed in task-parallel fashion
  - GI computation may map best to data or task parallel, depending on algorithm used
    - Memory access and locality characteristics can drive the choice
- **Ability to launch new computation without CPU overhead is key**
  - Especially when new computation is based on results from immediately previous stage

Customized Graphics Pipelines

Software graphics pipelines

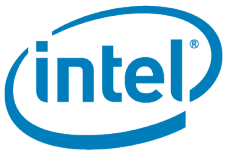
- **The (re)-advent of the software graphics pipeline opens up the GPU "black box"**
- **Pervasive implications**
  - Debugging
  - Performance measurement and tuning
  - Customization and extensibility
  - Testing a new idea doesn't require designing and simulating hardware; exploration in software is faster and cheaper

What is the value of a pipeline?

- **Producer/consumer computation**
  - Intermediate data stays on chip
  - Increasingly important given disconnect between off-chip bandwidth vs available computation
- **Implicit task parallelism**
  - Can easily reason about data-dependencies between stages
  - And thus can run multiple stages concurrently
- **Extending the "pipeline" deeper into the application broadens these benefits**

Customizing the pipeline

- **What is the cost of implementing n different algorithms to do {rasterization, culling, …}?**
- **For fixed-function, cost is very high**
  - Each one consumes additional chip area
    - Must build enough of it so that not (too often) a bottleneck
  - Area = $ + opportunity cost
  - Design time, testing, verification is expensive
  - More fixed-function means less area for more general compute



Customizing the pipeline

- **Cost of software implementations is very low**
  - ~100s of bytes of memory for instructions
  - Much less development/debugging time
  - Doesn't consume additional chip area
- **Software upgrades can continually improve performance**

Graphics pipeline innovation

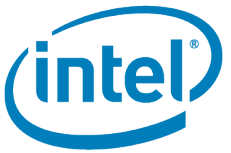
- **Innovating in the graphics pipeline is a matter of writing high-performance software**
  - This is easier than designing new high-performance hardware!
- **Low-hanging opportunities:**
  - Tessellation and displacement mapping
  - Custom culling stages
  - Compression / decompression
  - A-buffer / order-independent transparency
  - High-quality z-buffer, stochastic sampling
  - F-buffers / generalized stream-out

Example: tiled image processing

- **New pipeline stage**
  - After the backend has finished, tile FB data is in L2$
  - Run additional computation before writing results to memory
  - Adding such a stage is just a software modification
- **Great opportunity for efficient post-processing**
  - “Draw quad to do image processing” model wastes bandwidth
  - Tone mapping, color space conversion, MSAA resolve, transparency resolve, …

What differentiates “the pipeline”?

- **Developers can blur the lines between the graphics pipeline and the application**
  - Rich graphics data can live in LRB memory
    - Not just textures and vertex buffers
    - Scene graphs, data structures, …
    - Data structures easily updated in-place from LRB
      - And updates aren’t limited to wide data-parallel code
- **Wins from tighter coupling between application scene data and pipeline:**
  - Finer-grained/more effective culling
  - On-demand data generation



Stanford GRAMPS Project

- **Defined a set of pipeline construction primitives:**
  - Shaders (programmable stages), threads, work queues connecting stages, …
  - Scheduler to execute pipeline stages
  - These primitives map well to the LRB architecture
- **Built D3D pipeline, ray tracing, hybrid D3D+ray tracing on top of these primitives**
- **See upcoming TOG paper by Sugerman, Fatahalian, Boulos, Akeley, and Hanrahan**

Summary

- **Dynamic data structures and computation are the future of interactive rendering**
  - Fully software graphics pipelines are one culmination of this trend
- **New graphics architectures like LRB have become increasingly flexible to the point of CPU-level generality**
  - Data-parallel, thread-parallel, and points in between all at high performance
  - Great freedom to developers will lead to great innovations in rendering

Thanks

- Doug Carmean, Eric Sprangle, Tom Forsyth, Mike Abrash, Larry Seiler, Stephen Junkins, …
- Paul Lalonde, Craig Kolb, Aaron Lefohn, Jean-Luc Duprat
- Kayvon Fatahalian, Jeremy Sugerman