Intel’s Larrabee

Gary Jackson
garyj@cs.umd.edu
Outline

• Resources
• Introduction
• Hardware
• Software
• Potential Problems
• Conclusion
Disclaimer

• Larrabee doesn’t exist yet

• So we have to rely on what Intel says
Resources

• SIGGRAPH 2008 paper (Seiler, et al)
• Articles in Dr. Dobb’s Journal by Michael Abrash
• Article on AnandTech
What is Larrabee?

- Larrabee is a coupled hardware and software platform
- Hardware design is influenced by graphics developers and vice versa (Abrash)
Software

- Renderer
  - Consumes DirectX & OpenGL commands
- More than a typical driver
- Compiler for Larrabee
  - More than just x86
Hardware: Cores

- Many small cores based on the Pentium
- 16-way vector unit with 32 vector registers
- 256k coherent L2 cache
- In-order, dual issue
- 4 hardware threads

Adapted from Seiler, et al
Hardware: High Level Architecture

- Many cores (24-32 in initial release)
- 512 byte bidirectional ring bus
- Memory controller(s)
- Hardware texture filtering

Adapted from Seiler, et al
Vector Instructions

- With few exceptions, arithmetic is done entirely between registers
  - Some allow a load
  - 16x floats, 16x int32, 8x double, or 8x int64
  - Fused multiply-add
Vector Register Access

- Mask Registers
- Scatter/Gather/Broadcast to/from memory
- Swizzle
What’s a Swizzle?

- Convolution of vector values
- On Larrabee, swizzles are done in groups of four
- One swizzle for free on a vector operation
- In this example, we use the \{cdab\} swizzle to swap values in vector register $v0$
Cross Product: an Example

- Suppose we have two sets of four vectors in v0 and v1
- We can compute the cross product in two instructions
- We will need to “un-swizzle” v0 when reading the cross products

\[
\begin{align*}
&\text{vmulp v2 v0\{dacb\} v1} \\
&\text{vmsub132p v0 v2 v1\{dacb\}}
\end{align*}
\]
Comparison: Cell

- Similar: many cores arranged around a bus

- Different
  - Each core must use DMA
  - Controller is attached to bus

Source: M. Gochwind et al., Hot Chips-17, August 2005
Comparison: NVIDIA

- Hundreds of very simple cores
- Hardware threads
- Lots of fixed function silicon
  - Maybe not so much in Tesla
Software Renderer

- Tile based
  - Appears in low power and mobile devices
  - But not much in the high end, until now
Front End

• Break down rendering commands into PrimSets
  • PrimSets are geometric primitives and the associated rendering state

• Divide PrimSets into bins
  • Bins are the primitives and state that correspond to a tile in the frame
Binning

- Vertex shading
- Geometry shading
- Frustum and back plane culling
- Clipping
- Rasterizing
Back End: Pixel Shading, etc.

• Four threads per core work on a tile
  • One to set up and feed groups of sixteen pixels called qquads
  • Three to consume pixels and render the qquads in to the frame buffer
Back End: Fibers

- Each quad is associated with a fiber
- When texture filtering is requested, quad processing will block for many cycles
- Making a texture filter request causes the current fiber to yield to another one on the current work thread
Software Rasterizer

- Rasterize a triangle: Find the pixels in the tile that correspond to the triangle.
- Typically done in hardware, and hardware can do a better job.
- But the software rasterizer is “good enough”
Software Rasterizer

- Use triangle’s bounding box to find high-level coverage
- Then use a recursive descent algorithm to find coverage
Software Rasterizer

- Trivial exclusion corners
- If a particular corner does not lie in the half plane of a triangle side
- Then we don’t render the block, and we don’t have to recurse
Software Rasterizer

- Trivial inclusion corners
  - If the corner is not an exclusion corner, then it is inclusion
  - If all corners of a given block are trivial inclusion, then we just render the entire block
  - And no need to recurse
Finally ...

- We have 16x16 blocks
- We can just go through and process them in 4x4 chunks to identify individual pixels
Important Hardware Help

• We are always checking 16 corners for exclusion

• Size of our vector registers

• So we can compute the implicit line equation in two ops for all sixteen corners

• Bit operations help speed up the bottom
Performance: Methodology

- From Seiler, et al
- Getting the input data
  - Intercept the DirectX command stream
  - Extract 25 widely separated frames
- Three games
  - F.E.A.R., Gears of War, Half-Life 2 Ep. 2
Performance: Methodology

• Run the frames through a functional model of the renderer for verification and analysis

• Re-write portions in Larrabee assembly

• Re-test and re-cycle until 90% of the clock cycles are running in the cycle-accurate simulator
Performance: Relative Scaling

- Compare scaling on the three games as the number of cores goes up
- Scaling drops off at 48 cores

*Figure 9: Relative Scaling as a Function of Core Count: This shows configurations with 8 to 48 cores, with each game’s results plotted relative to the performance of an 8-core system.*
Performance: # of Cores

Figure 10: Overall performance: shows the number of Larrabee Units (cores running at 1 GHz) needed to achieve 60fps for of the series of sample frames in each game.

- 25 cores for F.E.A.R. and Gears of War
- 10 cores for Half-Life 2 Ep. 2
Performance: Bin Balance

Figure 11: Bin Balance for Gears of War: each curve shows the time required to process one frame’s bins, in sorted order from fastest to slowest, normalized by the mean bin processing time.

- The question: are we going to end up with idle cores at the end of processing?
Performance: Memory Bandwidth

![Graph showing memory bandwidth comparison between Binning BW and Immediate Mode BW for different games.]

**Figure 12:** Bandwidth comparison of binning vs. immediate mode per frame: binning requires bin reads & writes, but eliminates many depth/color accesses that are not detected by hierarchical depth tests. This results in less total bandwidth for binning.

- Sorted in least to most needs for memory bandwidth, per each frame
- No overdraw, so less bandwidth needed for binning on frames with greatest improvement
Performance: Breakdowns

- The point: software rendering can more easily adapt to varying workloads
- Though this isn’t actually presented here
Larrabee Native

• Statically compile C/C++ programs
• High-end parallelizing compiler
• Existing code will compile
Rendering Applications

- Render Target Read: reading out of a frame buffer
- Order independent transparency
- Irregular shadow mapping: alias free shadows
Other Examples

• Game Physics: cloth, collision detection, rigid body, etc.
• Real-time ray tracing
• Image & video processing
• Physical simulation
Potential Problems

• Is it fast enough for the money and power?
• Manufacturing obstacles
  • Estimated to be Intel’s largest die for the 45 nm process
Potential Problems

- Magical Compiler Syndrome
  - Intel is very good at writing compilers
  - But what about ...?
    - i860
    - Itanium
Conclusion

- Larrabee is exciting. They promise:
  - CPU ease
  - GPU performance
- They have novel software to support their hardware
- May release renderer
- It’s not out yet, so stay tuned ...
Questions?