Larrabee: A Many-Core x86 Architecture for Visual Computing

Larry Seiler[1], Doug Carmean[1], Eric Sprangle[1], Tom Forsyth[1], Michael Abrash[2],

Pradeep Dubey[1], Stephen Junkins[1], Adam Lake[1], Jeremy Sugerman[3],

Robert Cavin[1], Roger Espasa[1], Ed Grochowski[1], Toni Juan[1], and Pat Hanrahan[3]

Abstract

This paper presents a many-core visual computing architecture code named Larrabee, a new software rendering pipeline, a many-core programming model, and performance analysis for several applications. Larrabee uses multiple in-order x86 CPU cores that are augmented by a wide vector processor unit, as well as some fixed function logic blocks. This provides dramatically higher performance per watt and per unit of area than out-of-order CPUs on highly parallel workloads. It also greatly increases the flexibility and programmability of the architecture as compared to standard GPUs. A coherent on-die 2nd level cache allows efficient inter-processor communication and high-bandwidth local data access by CPU cores. Task scheduling is performed entirely with software in Larrabee, rather than in fixed function logic. The customizable software graphics rendering pipeline for this architecture uses binning in order to reduce required memory bandwidth, minimize lock contention, and increase opportunities for parallelism relative to standard GPUs. The Larrabee native programming model supports a variety of highly parallel applications that use irregular data structures. Performance analysis on those applications demonstrates Larrabee's potential for a broad range of parallel computation.

CCS: I.3.1 [Computer Graphics]: Hardware Architecture--Graphics Processors, Parallel Processing; I.3.3 [Computer Graphics]: Picture/Image Generation--Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism--Color, shading, shadowing, and texture

Keywords: graphics architecture, many-core computing, real-time graphics, software rendering, throughput computing, visual computing, parallel processing, SIMD, GPGPU.

1. Introduction

Modern GPUs are increasingly programmable in order to support advanced graphics algorithms and other parallel applications.

[1] Intel® Corporation: larry.seiler, doug.carmean, eric.sprangle, tom.forsyth, pradeep.dubey, stephen.junkins, adam.t.lake, robert.d.cavin, roger.espasa, edward.grochowski & toni.juan @intel.com

[2] RAD Game Tools: mikea@radgametools.com

[3] Stanford University: yoel & hanrahan @cs.stanford.edu

However, general purpose programmability of the graphics pipeline is restricted by limitations on the memory model and by fixed function blocks that schedule the parallel threads of execution. For example, pixel processing order is controlled by the rasterization logic and other dedicated scheduling logic.

This paper describes a highly parallel architecture that makes the rendering pipeline completely programmable. The Larrabee architecture is based on in-order CPU cores that run an extended version of the x86 instruction set, including wide vector processing operations and some specialized scalar instructions. Figure 1 shows a schematic illustration of the architecture. The cores each access their own subset of a coherent L2 cache to provide high-bandwidth L2 cache access from each core and to simplify data sharing and synchronization.

Larrabee is more flexible than current GPUs. Its CPU-like x86-based architecture supports subroutines and page faulting. Some operations that GPUs traditionally perform with fixed function logic, such as rasterization and post-shader blending, are performed entirely in software in Larrabee. Like GPUs, Larrabee uses fixed function logic for texture filtering, but the cores assist the fixed function logic, e.g. by supporting page faults.
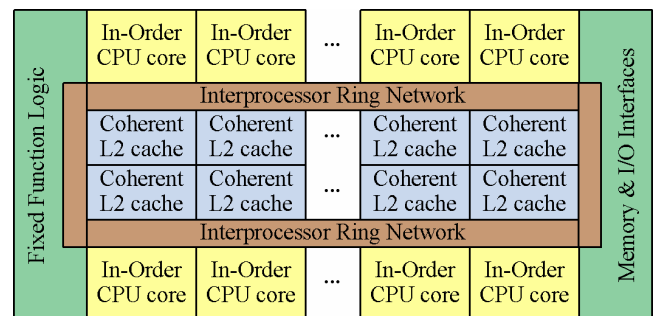

Figure 1: Schematic of the Larrabee many-core architecture: The number of CPU cores and the number and type of co-processors and I/O blocks are implementation-dependent, as are the positions of the CPU and non-CPU blocks on the chip.

This paper also describes a software rendering pipeline that runs efficiently on this architecture. It uses binning to increase parallelism and reduce memory bandwidth, while avoiding the problems of some previous tile-based architectures. Implementing the renderer in software allows existing features to be optimized based on workload and allows new features to be added. For example, programmable blending and order-independent transparency fit easily into the Larrabee software pipeline.

Finally, this paper describes a programming model that supports more general parallel applications, such as image processing, physical simulation, and medical & financial analytics. Larrabee's support for irregular data structures and its scatter-gather capability make it suitable for these throughput applications as demonstrated by our scalability and performance analysis.

2. Previous Work

Recent years have seen the evolution of commodity graphics hardware from fixed function units toward an increasingly programmable graphics pipeline, offering greater flexibility and performance to graphics programmers [Blythe 2006]. Real-time graphics APIs evolved to expose this functionality with high level shading languages such as Cg [Mark et al. 2003], HLSL [Microsoft 2007] and GLSL [Kessenich et al. 2006]. Additionally, a number of arithmetically intensive workloads perform very well on GPU-like architectures [GPGPU 2007; Owens et al. 2007].

2.1 PC Graphics Processor Architectures

Multi-processing graphics hardware has been around for decades. The key ideas behind these architectures are described by Fuchs et al. [1989], Molnar et al. [1992], Foley et al. [1996], and Stoll et al. [2001]. The motivation has always been to leverage the data parallel nature of rendering to gain maximum performance and visual fidelity. Early architectures were complex systems with multiple boards and many specialized chips. The modern graphics architecture is a single chip that fits into the form factor of a PC or other graphics platform [Kelley et al. 1992; Kelley et al. 1994; Torborg & Kajiya 1996]. Recent architectures implement the Microsoft DirectX* 10 API, including the Nvidia GeForce* 8 [Nvidia 2008] and the ATI Radeon* HD 3800 series [AMD 2008].

Figure 2 shows a simplified version of the DirectX 10 pipeline [Blythe 2006]. The programmable OpenGL pipeline is following a similar architectural direction [Rost 2004]. In early implementations, each stage required separate programmable units, but the most recent architectures use a unified shader model. Rasterization and texture filter operations are still largely fixed function in modern GPUs, so changes to the supported features require a new chip design, as well as a new API version.
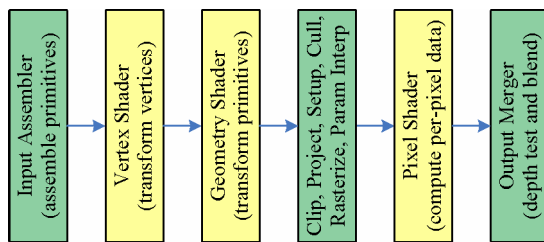

Figure 2: Simplified DirectX10 Pipeline: Yellow components are programmable by the user, green are fixed function. Memory access, stream output, and texture filtering stages are omitted.

Recent research brings the computational capabilities of commodity graphics hardware to bear on a variety of problems including collision detection, financial modeling, and signal processing [Owens et al. 2007]. There are several aspects of GPU architectures that lend themselves to these workloads. Performance increases are most dramatic when the compute to bandwidth ratio is high. That occurs when the application primarily uses regular data structures and requires many arithmetic operations for each data item being processed.

2.2 Taxonomy of Rendering Methods

Molnar et al. [1994] classified graphics architectures by the primary stage in the graphics pipeline where sorting for parallelism occurs. Eldridge [2001] expanded this taxonomy and performed a detailed comparison of parallel rendering systems. Both use *sort-middle* to refer to sorting after geometry processing but before rasterization. Molnar et al. use *sort-last sparse* and

Eldridge uses *sort-last fragment* to refer to sorting before fragment processing, that is, before depth test and alpha blending. Since their work was done before the introduction of pixel shaders, no clear distinction is provided for sorting after computing coverage but before executing the pixel shader. In this paper we treat this alternative as a variant of *sort-middle*.

Current GPUs often use Eldridge's sort-last fragment. This allows pixel shader results to be sorted into a small number of screen-aligned regions just before depth testing and blending. Often these screen aligned regions are associated with individual memory controllers, to allow more efficient memory accesses. Sort-last fragment allows immediate mode rendering with relatively short FIFO buffers for sorting the data. However, if a pixel is accessed multiple times at widely separated intervals, it typically must be read from and written to memory multiple times.

Sort middle algorithms have been called binning, tiling, chunking, bucket, and zone rendering [Hsieh et al. 2001; Chen et al. 1998]. This method processes vertices to produce screen coordinates for primitives, which are sorted into bins based on their location on the screen. Each bin is associated with a tile region on the screen, which can be sized to fit into on-chip cache so that only one access to memory is needed per pixel regardless of the depth complexity. One problem is that primitives that overlap multiple tiles must be stored in multiple bins, which increases the memory bandwidth needed for them. Molnar et al. [1994], Chen et al. [1998], and Eldridge [2001] concluded that the impact of overlap is limited, especially when primitives are small compared to region size, which occurs as the triangle count of a scene increases. The PowerVR* MBX and SGX series [PowerVR 2008], the Intel® Graphics Media Accelerator 900 Series [Lake 2005], the ARM Mali* [Stevens 2006], and Talisman [Torborg & Kajiya 1996] have been generally classified as sort middle architectures.

2.3 General Purpose CPU Architectures

In 1995, Intel introduced the Pentium® Pro processor, which used out-of-order 3-wide instruction execution in response to the demand for increasing single-stream performance [Gwennap 1995]. Out-of-order architectures identify independent instruction streams that can be executed in parallel. The logic to identify these instructions consumes die area as well as power. Later CPU generations used even more elaborate techniques to increase single-stream performance at ever increasing penalties in area and power relative to performance. It has been observed that, within the same process technology, a new microprocessor design with 1.5x to 1.7x the performance consumes 2x to 3x the die area [Pollack 1999] and 2x to 2.5x the power [Grochowski et al. 2004].

For highly parallel algorithms, more performance can be gained by packing multiple cores onto the die instead of increasing single stream performance. The IBM Cell* was designed with these high throughput workloads in mind [Pham et al. 2005]. Cell includes a single Power* Processor core, its L2 cache, and a set of high throughput cores. These cores each contain a local memory store that is incoherent with the rest of the memory system. The local store has a guaranteed latency for data delivery, which allows a simpler execution pipeline than a system with a coherent cache hierarchy. It requires the user to manually manage the data contents through software-programmed DMA operations.

CPUs can also use multi-threading to gain parallelism. Niagara is a multi-core general purpose microprocessor [Kongetira et al. 2005] featuring eight in-order cores, each capable of executing four simultaneous threads, and a shared cache. But given its focus on commercial server workloads, Niagara lacks architectural elements critical for visual computing, such as SIMD floating-point execution, scatter-gather, or fixed function texture support.

3.   Larrabee Hardware Architecture

Figure 1 above shows a block diagram of the basic Larrabee architecture. Larrabee is designed around multiple instantiations of an in-order CPU core that is augmented with a wide vector processor (VPU). Cores communicate through a high-bandwidth interconnect network with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the exact application. For example, an implementation of Larrabee as a stand-alone GPU would typically include a PCIe bus.

The data in Table 1 motivates Larrabee's use of in-order cores with wide VPUs. The middle column shows the peak performance of a modern out-of-order CPU, the Intel® Core™2 Duo processor. The right-hand column shows a test CPU design based on the Pentium® processor, which was introduced in 1992 and used dual-issue in-order instruction execution [Alpert 1993]. The Pentium processor core was modified to support four threads and a 16-wide VPU. The final two rows specify the number of non-vector instructions that can be issued per clock by one CPU and the total number of vector operations that can be issued per clock. The two configurations use roughly the same area and power.

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

Table 1: *Out-of-order vs. in-order CPU comparison: designing the processor for increased throughput can result in ½ the peak single-stream performance, but 20x the peak vector throughput with roughly the same area and power. This difference is 40x in FLOPS, since the wide VPU supports fused multiply-add but SSE doesn't. These in-order cores are not Larrabee, but are similar.*

The test design in Table 1 is not identical to Larrabee. To provide a more direct comparison, the in-order core test design uses the same process and clock rate as the out-of-order cores and includes no fixed function graphics logic. This comparison motivates design decisions for Larrabee since it shows that a wide VPU with a simple in-order core allows CPUs to reach a dramatically higher computational density for parallel applications.

Sections 3.1 to 3.5 below describe the key features of the Larrabee architecture: the CPU core, the scalar unit and cache control instructions, the vector processor, the interprocessor ring network, and the choices for what is implemented in fixed function logic.

3.1 Larrabee Core and Caches

Figure 3 shows a schematic of a single Larrabee CPU core, plus its connection to the on-die interconnect network and the core's local subset of the L2 cache. The instruction decoder supports the standard Pentium processor x86 instruction set, with the addition of new instructions that are described in Sections 3.2 and 3.3. To simplify the design the scalar and vector units use separate register sets. Data transferred between them is written to memory and then read back in from the L1 cache.

Larrabee's L1 cache allows low-latency accesses to cache memory into the scalar and vector units. Together with Larrabee's load-op VPU instructions, this means that the L1 cache can be treated somewhat like an extended register file. This significantly improves the performance of many algorithms, especially with the cache control instructions described Section 3.2. The single-threaded Pentium processor provided an 8KB Icache and 8KB Dcache. We specify a 32KB Icache and 32KB Dcache to support four execution threads per CPU core.
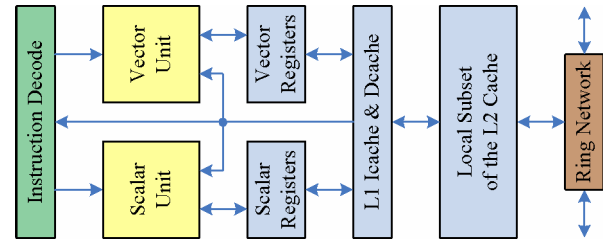

Figure 3: *Larrabee CPU core and associated system blocks: the CPU is derived from the Pentium processor in-order design, plus 64-bit instructions, multi-threading and a wide VPU. Each core has fast access to its 256KB local subset of a coherent 2nd level cache. L1 cache sizes are 32KB for Icache and 32KB for Dcache. Ring network accesses pass through the L2 cache for coherency.*

Larrabee's global 2nd level (L2) cache is divided into separate local subsets, one per CPU core. Each CPU has a fast direct access path to its own local subset of the L2 cache. Data read by a CPU core is stored in its L2 cache subset and can be accessed quickly, in parallel with other CPUs accessing their own local L2 cache subsets. Data written by a CPU core is stored in its own L2 cache subset and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data, as described in Section 3.4. We specify 256KB for each L2 cache subset. This supports large tile sizes for software rendering, as described in Section 4.1.

3.2 Scalar Unit and Cache Control Instructions

Larrabee's scalar pipeline is derived from the dual-issue Pentium processor, which uses a short, inexpensive execution pipeline. Larrabee provides modern additions such as multi-threading, 64-bit extensions, and sophisticated prefetching. The cores support the full Pentium processor x86 instruction set so they can run existing code including operating system kernels and applications. Larrabee adds new scalar instructions such as bit count and bit scan, which finds the next set bit within a register.

Larrabee also adds new instructions and instruction modes for explicit cache control. Examples include instructions to prefetch data into the L1 or L2 caches and instruction modes to reduce the priority of a cache line. For example, streaming data typically sweeps existing data out of a cache. Larrabee is able to mark each streaming cache line for early eviction after it is accessed. These cache control instructions also allow the L2 cache to be used similarly to a scratchpad memory, while remaining fully coherent.

Within a single core, synchronizing access to shared memory by multiple threads is inexpensive. The threads on a single core share the same local L1 cache, so a single atomic semaphore read within the L1 cache is sufficient. Synchronizing access between

multiple cores is more expensive, since it requires inter-processor locks. This is a well known problem in multi-processor design.

Multi-issue CPU cores often lose performance due to the difficulty of finding instructions that can execute together. Larrabee's dual-issue decoder has a high multi-issue rate in code that we've tested. The pairing rules for the primary and secondary instruction pipes are deterministic, which allows compilers to perform offline analysis with a wider scope than a runtime out-of-order instruction picker can. All instructions can issue on the primary pipeline, which minimizes the combinatorial problems for a compiler. The secondary pipeline can execute a large subset of the scalar x86 instruction set, including loads, stores, simple ALU operations, branches, cache manipulation instructions, and vector stores. Because the secondary pipeline is relatively small and cheap, the area and power wasted by failing to dual-issue on every cycle is small. In our analysis, it is relatively easy for compilers to schedule dual-issue instructions.

Finally, Larrabee supports four threads of execution, with separate register sets per thread. Switching threads covers cases where the compiler is unable to schedule code without stalls. Switching threads also covers part of the latency to load from the L2 cache to the L1 cache, for those cases when data cannot be prefetched into the L1 cache in advance. Cache use is more effective when multiple threads running on the same core use the same dataset, e.g. rendering triangles to the same tile.

3.3 Vector Processing Unit

Larrabee gains its computational density from the 16-wide vector processing unit (VPU), which executes integer, single-precision float, and double-precision float instructions. The VPU and its registers are approximately one third the area of the CPU core but provide most of the integer and floating point performance. Figure 4 shows a block diagram of the VPU with the L1 cache.
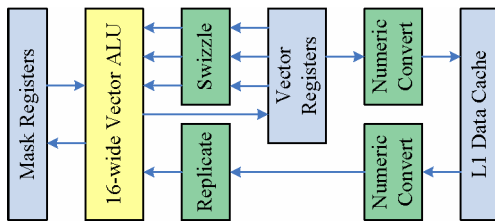

***Figure 4**: Vector unit block diagram: the VPU supports 3-operand instructions. It supports swizzling the register inputs and numeric conversion and replication on the memory input. Mask registers allow predicating the resulting vector writes.*

We chose a 16-wide VPU as a tradeoff between increased computational density and the difficulty of obtaining high utilization for wider VPUs. Early analysis suggested 88% utilization for typical pixel shader workloads if 16 lanes process 16 separate pixels one component at a time, that is, with separate instructions to process red, green, etc., for 16 pixels at a time, instead of processing multiple color channels at once. The Nvidia GeForce 8 operates in a similar fashion, organizing its scalar SIMD processors in groups of 32 that execute the same instruction [Nickolls et al. 2008]. The main difference is that in Larrabee the loop control, cache management, and other such operations are code that runs in parallel with the VPU, instead of being implemented as fixed function logic.

Larrabee VPU instructions allow up to three source operands, one of which can come directly from the L1 cache. If the data has been prefetched into the cache, as described in Section 3.2, then the L1 cache is in effect an extended register file. 8-bit unorm, 8-bit uint, 16-bit sint and 16-bit float data can be read from the cache and converted to 32-bit floats or 32-bit integers with no loss of performance. This significantly increases the amount of data that can be stored in the caches and also reduces the need for separate data conversion instructions.

The next stage is to align the data from registers and memory with the processing lanes in the VPU. Register data can be swizzled in a variety of ways, e.g. to support matrix multiplication. Data from memory can be replicated across the VPU lanes. This is a common operation in both graphics and non-graphics parallel data processing, which significantly increases the cache efficiency.

The VPU supports a wide variety of instructions on both integer and floating point data types. The instruction set provides the standard arithmetic operations, including fused multiply-add, and the standard logical operations, including instructions to extract non-byte-aligned fields from pixels. These are all load-op instructions, which read from registers or memory and write the result to a vector register. Additional load and store instructions support a wider variety of conversions between floating point values and the less common or more complex data formats found on most GPUs. Using separate instructions for these formats saves significant area and power at a small performance cost.

The VPU instruction set also includes gather and scatter support, that is, loads and stores from non-contiguous addresses. Instead of loading a 16-wide vector from a single address, 16 elements are loaded from or stored to up to 16 different addresses that are specified in another vector register. This allows 16 shader instances to be run in parallel, each of which appears to run serially, even when performing array accesses with computed indices. The speed of gather/scatter is limited by the cache, which typically only accesses one cache line per cycle. However, many workloads have highly coherent access patterns, and therefore take much less than 16 cycles to execute.

Finally, Larrabee VPU instructions can be predicated by a mask register, which has one bit per vector lane. The mask controls which parts of a vector register or memory location are written and which are left untouched. For example, a scalar if-then-else control structure can be mapped onto the VPU by using an instruction to set a mask register based on a comparison, and then executing both if and else clauses with opposite polarities of the mask register controlling whether to write results. Clauses can be skipped entirely if the mask register is all zeros or all ones. This reduces branch misprediction penalties for small clauses and gives the compiler's instruction scheduler greater freedom.

The VPU also uses these masks for packed load and store instructions, which access enabled elements from sequential locations in memory. This enables the programmer to bundle sparse strands of execution satisfying complex branch conditions into a format more efficient for vector computation.

3.4 Inter-Processor Ring Network

Larrabee uses a bi-directional ring network to allow agents such as CPU cores, L2 caches and other logic blocks to communicate with each other within the chip. When scaling to more than 16 cores, we use multiple short linked rings.

Each ring data-path is 512-bits wide per direction. All the routing decisions are made before injecting messages into the network. For example, each agent can accept a message from one direction

on even clocks and from the other direction on odd clocks. This simplifies the routing logic and means that no storage is required in the routers once the message is in the network. The result is high bandwidth with minimal contention at a very low cost.

Larrabee's L2 cache is designed to provide each core with high bandwidth access to memory addresses that are not written by other cores, and therefore are stored in the core's local L2 subset. Each core can access its own subset of the L2 cache in parallel, without communicating with other cores. However, before allocating a new line in the L2 cache, the ring network is used to check for data sharing, in order to maintain data coherency.

The inter-processor network also provides a path for the L2 caches to access memory. A typical high-end implementation would include multiple memory interfaces of standard design, spread around the inter-processor network to reduce congestion. Latency around the on-die network increases memory access times, but the extra ring latency is typically very small compared to the latency of DRAM access.

Finally, the on-die inter-processor network allows fixed function logic agents to be accessed by the CPU cores and in turn to access L2 caches and memory. As with memory controllers, these would typically be spread around the ring network to reduce congestion.

3.5 Fixed Function Logic

Modern GPUs contain fixed function logic for a variety of graphics tasks, including texture filtering, display processing, post-shader alpha blending, rasterization, and interpolation. In this paper, *rasterization* refers solely to finding the coverage of a primitive, and *interpolation* refers to finding the values of parameters at covered sample positions in the primitive. Fixed function logic typically requires FIFOs for load balancing. It can be difficult to properly size these logic blocks and their FIFOs to avoid both wasted area and performance bottlenecks.

Larrabee uses software in place of fixed function logic when a software implementation provides sufficient performance. In particular, Larrabee does not include fixed function logic for rasterization, interpolation, or post-shader alpha blending. This allows Larrabee to add new features and optimizations, as well as allowing these tasks to be implemented in different places in the rendering pipeline, depending what is most efficient for a particular application. Implementing them in software also allows Larrabee to allocate to each the performance it requires, instead of designing hardware to meet peak performance requirements. Sections 4.4 and 4.5 describe the software algorithms used and Section 5.5 shows the percentage of processing time required by these operations for three game workloads.

Larrabee includes texture filter logic because this operation cannot be efficiently performed in software on the cores. Our analysis shows that software texture filtering on our cores would take 12x to 40x longer than our fixed function logic, depending on whether decompression is required. There are four basic reasons:

- Texture filtering still most commonly uses 8-bit color components, which can be filtered more efficiently in dedicated logic than in the 32-bit wide VPU lanes.
- Efficiently selecting unaligned 2x2 quads to filter requires a specialized kind of pipelined gather logic.
- Loading texture data into the VPU for filtering requires an impractical amount of register file bandwidth.
- On-the-fly texture decompression is dramatically more efficient in dedicated hardware than in CPU code.

The Larrabee texture filter logic is internally quite similar to typical GPU texture logic. It provides 32KB of texture cache per core and supports all the usual operations, such as DirectX 10 compressed texture formats, mipmapping, anisotropic filtering, etc. Cores pass commands to the texture units through the L2 cache and receive results the same way. The texture units perform virtual to physical page translation and report any page misses to the core, which retries the texture filter command after the page is in memory. Larrabee can also perform texture operations directly on the cores when the performance is fast enough in software.

4.   Larrabee Software Renderer

The key issue for achieving high performance for any parallel rendering algorithm is to divide the rendering task into many subtasks that can be load balanced and executed in parallel with very few synchronization points. Larrabee allows more options for parallelism than typical GPUs due to its flexible memory model and software-controlled scheduling.

This section describes a sort-middle software renderer designed for the Larrabee architecture that uses binning for load balancing. Section 5 provides performance studies for this software renderer.

4.1 Stages of Software Rendering

For simplicity, first we will consider rendering to a single set of render targets, such as a pixel buffer and a depth/stencil buffer. These render targets and the rendering commands that modify them are together called an *RTset*. Section 4.2 discusses more complex cases involving multiple RTsets.

The rendering commands for an RTset are typically specified by graphics APIs as a series of rendering state changes, followed by a batch of triangles rendered using that current device state. Rather than use the concept of a *current* state internally, the Larrabee renderer captures the rendering state in a single fully-specified structure. It then groups the batches of triangles and tags each batch with the state it uses. This batch of triangles and the state it uses is called a *primitive set* or *PrimSet*. This is roughly equivalent to the DirectX DrawPrimitive call, although there is not an exact 1:1 correspondence between the two.

Figure 5 shows the broad structure for rendering the PrimSets of a single RTset. The surface being rendered is split into tiles of pixels. Each tile has a bin that will be filled with the triangles from a PrimSet that intersect that tile. The set of bins for the whole RTset is called a *bin set*. The terms *tile* and *bin* are sometimes used interchangeably. The distinction in this paper is that a *tile* is the actual pixel data, while the *bin* is the set of primitives that affect that tile. In the same way that each tile has a bin, each RTset (set of render target tiles and associated PrimSets) has a single bin set (set of bins that contain the primitives).
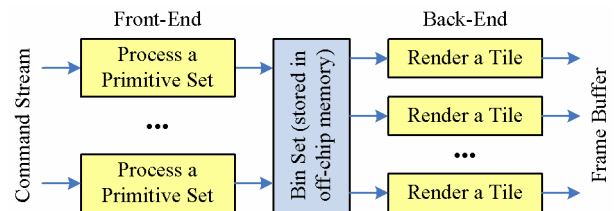

Figure 5: *Larrabee Software Renderer Structure: Multiple sets of primitives (PrimSets) can be processed in parallel to fill per-tile bins, which are later processed in parallel to render screen tiles.*

Tile size is chosen so that the target surfaces in the RTset for that tile will all fit in a core's L2 cache. Thus an RTset with many color channels, or with large high-precision data formats, will use a smaller tile size than one with fewer or low-precision channels. To simplify the code, tiles are usually square and a power-of-two in size, typically ranging in size from 32x32 to 128x128. An application with 32-bit depth and 32-bit color can use a 128x128 tile and fill only half of the core's 256KB L2 cache subset.

As long as a tile fits within the L2 cache, rendering speed does not change substantially for different tile sizes. The main impact of using smaller tiles is that some triangles in the scene will hit more than one tile and require processing in each of those tiles – this is termed *bin spread*. Smaller tiles increase bin spread, but it is not a large increase. Typically we see bin spread of less than 5% in modern workloads. That is, the number of triangles processed across the system is less than 5% higher than the number for a single large bin covering the entire render target.

There are two phases to the processing. In the front-end, each PrimSet is given a sequence ID to identify where in the rendering stream it was submitted. This is used by the back-end to ensure correct ordering, as discussed below. The PrimSet is then assigned to a single core, which performs vertex shading, tessellation, geometry shading, culling and clipping to produce triangles (or other primitives). The core then rasterizes each triangle to determine which tiles it touches and which samples it covers within each of those tiles. The result is a series of X,Y coordinates and sample coverage masks for each triangle. This data is stored in the bins along with indices that reference the vertex data.

Once all front-end processing for the RTset has finished and every triangle has been added to the bin for each tile that it touched, back-end processing is performed. Here, each tile is assigned to a single core, which shades each triangle from the associated bin, including requesting texture sampling from the co-processors. The back-end also performs depth, stencil and blending operations.

Unlike some other tile-based rendering methods, there is no attempt at perfect occlusion culling before shading, reordering of shading, or any other non-standard rendering methods. When taking commands from a DirectX or OpenGL command stream, rendering for a single tile is performed in the order in which the commands are submitted. Using a conventional rendering pipeline within each tile avoids surprises in either functionality or performance and works consistently well across a broad spectrum of existing applications.

4.2 Render Target Dependency Analysis

A single frame consists of a sequence of rendering commands, each sent to a set of rendering surfaces. Modern applications may use multiple pixel targets at once, and may change targets frequently during a single frame in order to render effects such as reflections and shadow maps.

To handle different sets of render targets within a single frame, Larrabee's software renderer starts by creating a graph where each node corresponds to an *RTset*, as defined in Section 4.1. Each node is then assigned the PrimSets that modify that node's set of render targets. When an RTset uses a render target (e.g. a texture) that is used by subsequent rendering operations to a different target, a dependency is set up between the two RTsets. For example, in shadow mapping, the main RTset for a scene (the back buffer and depth/stencil buffer) has a dependency on the RTset for each of the shadow maps used.

Once the dependency graph is created, the nodes can be selected for rendering in any order that satisfies the dependencies. Figure 6 shows a dependency graph for two frames of a scene that requires rendering two shadow maps. For simplicity, the shadow maps for frame 1 are not shown. Frame 2 of the scene cannot be rendered until after frame 2's shadow maps are rendered. Since each frame in this simple example uses the same memory for the back buffer and depth buffer, frame 2 also cannot be rendered until frame 1's scene is rendered and copied to the front buffer (the dotted line dependency). However, rendering the frame 2 shadow maps can overlap with frame 1 rendering, since there are no dependencies. Using a different back buffer for frame 2 would remove the dotted line dependency. This substitution can be done automatically.
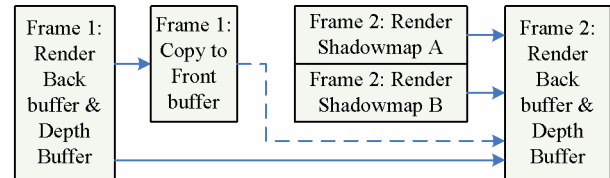

Figure 6: *RTset dependency graph: PrimSets are assigned to an RTset node based on the surfaces (render targets) that they modify. The dependencies ensure that a surface is not used until the PrimSets that modify it have been rendered and is not modified until the PrimSets that use it have been rendered.*

Note that the PrimSets associated with an RTset can be divided into multiple subsets whenever required, so long as ordering is maintained. An RTset can be split if it is too large to be efficiently processed as one unit, e.g. to provide finer scheduling granularity. Not all Larrabee cores need to process PrimSets from the same RTset at the same time. This ability to arbitrarily split and schedule RTsets avoids the limitations of some previous tiling architectures [Lake 2005].

4.3 Front-End Vertex and Geometry Processing

Since graphics rendering commands modify state, the order of execution matters. GPUs process these commands sequentially, so that the commands are started in order and finished in order. When operations within a rendering command are parallelized over the inputs, the outputs must be put back in order. Geometry shaders, where the number of outputs is variable, require particularly large FIFOs to maintain order and minimize stalls.

Larrabee allows front-end processing of multiple PrimSets in parallel. A control processor decides which PrimSets to render at any particular time, according to the dependencies in the RTset graph, and adds those PrimSets to an active list. The Larrabee cores doing front-end work constantly take PrimSets from this active list. Each core works on its own PrimSet independently. When the core is finished, it takes the next from the active list. Each core uses its own subset of the bin for each tile, which eliminates lock contention with the other front-end cores. The PrimSet's sequence ID is written into the bins so that the back-end can restore the original order by always reading primitives from the sub-bin with the smallest sequence ID.

Figure 7 shows the processing stages within a single front-end core. The first step identifies the vertices that form each primitive. This can be complex due to index buffers that allow arbitrary mappings of vertices in a vertex buffer to primitives, e.g. to efficiently store meshes. Next, the required vertices are transformed by running the vertex shader on them if they haven't already been transformed. Transformed vertices are streamed out

to main memory. Values other than the position data are actively evicted from the L2 cache to avoid pollution, as they are not needed again until interpolant setup in the back end. After this, the geometry shader is run, followed by frustum and back-face culling, then clipping.
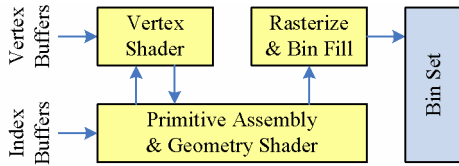

Figure 7: Front-End Rendering Sequence for a PrimSet: the renderer shades vertices when required for a primitive, then puts final primitives into the bins whose tiles the primitive intersects.

We describe a version of the algorithm that computes coverage information in the front-end and puts it into the bins. This ensures good load balancing, even if a small number of bins contain a large number of triangles. Rasterization can occur in either the front-end or the back-end, or can be split between them, since Larrabee uses software rasterization, as described in Section 4.4.

4.4 Software Rasterization and Interpolation

Unlike modern GPUs, Larrabee does not use dedicated logic for rasterization and parameter interpolation. In this paper, *rasterization* refers to finding the coverage of a primitive and *interpolation* refers to finding the values of parameters at covered sample positions. The figures in Section 5.4 show that these operations do not take a significant fraction of the rendering workload, so using software is justifiable. This section describes the algorithms that we use and why software implementations are preferable on Larrabee.

The justification for performing interpolation in software is relatively simple. In older graphics APIs, interpolation produced fixed point numbers, much like the current state for the most common texture filtering operations. In modern graphics APIs such as DirectX 10, the required result is a 32-bit float. Therefore it is efficient to re-use the existing VPU for interpolation.

Rasterization is unquestionably more efficient in dedicated logic than in software when running at peak rates, but using dedicated logic has drawbacks for Larrabee. In a modern GPU, the rasterizer is a fine-grain serialization point: all primitives are put back in order before rasterization. Scaling the renderer over large numbers of cores requires eliminating all but the most coarse-grained serialization points. The rasterizer could be designed to allow multiple cores to send it primitives out of order, but this would impose a significant communication expense and would require software to manage contention for the rasterizer resource. A software rasterizer avoids these costs. It also allows rasterization to be parallelized over many cores or moved to multiple different places in the rendering pipeline. We can optimize the rasterization code for a particular workload or support alternative rasterization equations for special purposes [Lloyd et al. 2007].

Our algorithm is a highly optimized version of the recursive descent algorithm described by Greene [1996]. The basic idea is to convert clipped triangles to screen space, then compute a half-plane equation for each triangle edge [Pineda 1988]. This lets us determine if a rectangular block is entirely inside the triangle, entirely outside the triangle, or partially covered by the triangle. In the latter case, the algorithm subdivides the block recursively until it is reduced to an individual pixel or sample position.

On Larrabee, the first step uses the triangle's bounding box to find the tiles that the triangle overlaps. In the remaining steps, the VPU computes half-plane equations for 16 blocks at a time. For example, if the tile size is 64x64, the first stage processes 16 16x16 blocks that cover the tile. The find first bit instruction makes it efficient to find fully and partially covered blocks. Detecting fully covered blocks early is important for efficiency. The second stage tests the 16 4x4 sub-blocks of each partially covered 16x16 block. The third stage tests the 16 pixels of each partially covered 4x4 block. This stage can be repeated for multiple sample positions in each pixel. About 70% of the instructions run on the VPU and take advantage of Larrabee's computational density. About 10% of the efficiency of the algorithm comes from special instructions such as find first bit.

4.5 Back-End Pixel Processing

Once the front-end processing for an RTset has completed filling the bins with triangle data, the RTset is put into an active list. The cores doing back-end work constantly take the next available tile from the list and render the triangles in the corresponding bin. This software can use many optimizations that are commonly implemented in fixed function logic in modern GPUs, such as fast clear, hierarchical Z, and early Z tests [Morein 2000]. Hierarchical Z tests can be done in the front-end to reduce the number of primitives placed in the bins.

The back-end code starts by prefetching the render target pixels into the L2 cache. All rendering will then be performed to the L2 cache until there are no more primitives to render for the tile, when it will be written back to memory. As a result, the pixels in the RTset for this tile only need to be read and written once to main memory, regardless of how many overlapping primitives are in the bin. Two important optimizations can also be detected to save substantial memory bandwidth. The read can be eliminated if the first command clears the entire tile. The write can also be eliminated or reduced for depth data that is not required after rendering and for MSAA colors that can be resolved to one color per pixel before writing to memory.

Figure 8 shows a back-end implementation that makes effective use of multiple threads that execute on a single core. A setup thread reads primitives for the tile. Next, the setup thread interpolates per-vertex parameters to find their values at each sample. Finally, the setup thread issues pixels to the work threads in groups of 16 that we call a *qquad*. The setup thread uses scoreboarding to ensure that qquads are not passed to the work threads until any overlapping pixels have completed processing.
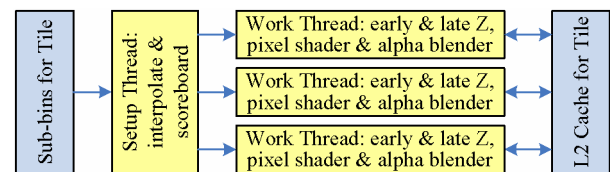

Figure 8: Back-End Rendering Sequence for a Tile: one setup thread processes the primitives and assigns them to one of three work threads that do early Z depth tests, pixel shader processing, late Z depth tests, and alpha blending.

The three work threads perform all remaining pixel processing, including pre-shader early Z tests, the pixel shader, regular late Z tests, and post-shader blending. Modern GPUs use dedicated logic for post-shader blending, but Larrabee uses the VPU. This is particularly efficient since many shaders do not use post-shader blending, so that dedicated blending logic can be unused for some

shaders and may limit performance for other shaders. Section 5.4 provides breakdowns of the total processing time devoted to post-shader blending and parameter interpolation.

One remaining issue is texture co-processor accesses, which can have hundreds of clocks of latency. This is hidden by computing multiple qquads on each hardware thread. Each qquad's shader is called a *fiber*. The different fibers on a thread co-operatively switch between themselves without any OS intervention. A fiber switch is performed after each texture read command, and processing passes to the other fibers running on the thread. Fibers execute in a circular queue. The number of fibers is chosen so that by the time control flows back to a fiber, its texture access has had time to execute and the results are ready for processing.

5. Renderer Performance Studies

This section describes performance and scaling studies for the Larrabee software renderer described in Section 4. Studies include scalability experiments for software rendering, load balancing studies, bandwidth comparisons of binning to immediate mode renderers, performance on several game workloads, and charts showing the how total processing time is divided among different parts of the software renderer.

5.1 Game Workloads and Simulation Method

Performance tests use workloads derived from three well-known games: Gears of War[*], F.E.A.R.[*], and Half Life[*] 2 Episode 2. Table 2 contains information about the tested frames from each game. Since we are scaling out to large numbers of cores we use a high-end screen size with multisampling when supported.

Half Life 2 ep. 2	F.E.A.R.	Gears of War
1600x1200 4 sample	1600x1200 4 sample	1600x1200 1 sample
25 frames (1 in 30)	25 frames (1 in 100)	25 frames (1 in 250)
Valve Corp.	Monolith Productions	Epic Games Inc

Table 2: *Workload summary for the three tested games: the frames are widely separated to catch different scene characteristics as the games progress.*

We captured the frames by intercepting the DirectX 9 command stream being sent to a conventional graphics card while the game was played at a normal speed, along with the contents of textures and surfaces at the start of the frame. We tested them through a functional model to ensure the algorithms were correct and that the right images were produced. Next, we estimated the cost of each section of code in the functional model, being aggressively pessimistic, and built a rough profile of each frame. We wrote assembly code for the highest-cost sections, ran it through cycle-accurate simulators, fed the clock cycle results back into the functional model, and re-ran the traces. This iterative cycle of refinement was repeated until 90% of the clock cycles executed during a frame had been run through the simulators, giving the overall profiles a high degree of confidence. Texture unit throughput, cache performance and memory bandwidth limitations were all included in the various simulations.

In these studies we measure workload performance in terms of *Larrabee units*. A *Larrabee unit* is defined to be one Larrabee core running at 1 GHz. The clock rate is chosen solely for ease of calculation, since real devices would ship with multiple cores and

[*] Other names & brands may be claimed as the property of others.

a variety of clock rates. Using Larrabee units allows us to compare performance of Larrabee implementations with different numbers of cores running at different clock rates. A single Larrabee unit corresponds to a theoretical peak throughput of 32 GFLOPS, counting fused multiply-add as two operations.

5.2 Scalability Studies

The Larrabee software renderer is designed to allow efficient load balancing over a large number of cores. Figure 9 shows the results of testing load balancing for six configurations, each of which scales the memory bandwidth and texture filtering speed relative to the number of cores. This test uses the simulation methodology described in Section 5.1 in combination with a time-based performance model that tracks dependencies and scheduling. This tool is used for multiple graphics products within Intel.
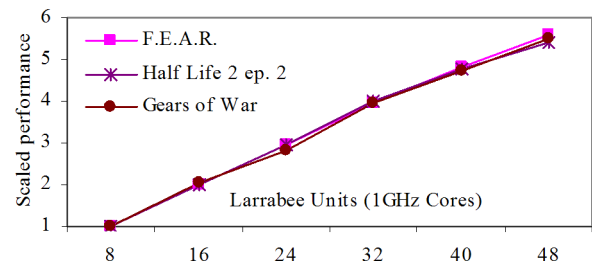

Figure 9: *Relative Scaling as a Function of Core Count: This shows configurations with 8 to 48 cores, with each game's results plotted relative to the performance of an 8-core system.*

The results of the load balancing simulation show a falloff of 7% to 10% from a linear speedup at 48 cores. For these tests, PrimSets are subdivided if they contain more than 1000 primitives, as described in Section 4.2. Additional tests show that F.E.A.R. falls off by only 2% if PrimSets are subdivided into groups of 200 primitives, so code tuning should improve the linearity.

Figure 10 shows the number of Larrabee units required to render sample frames from the three games at 60 frames/second. These results were simulated on a single core with the assumption that performance scales linearly. For Half Life 2 episode 2, roughly 10 Larrabee Units are sufficient to ensure that all frames run at 60 fps or faster. For F.E.A.R. and Gears of War, roughly 25 Larrabee Units suffice.
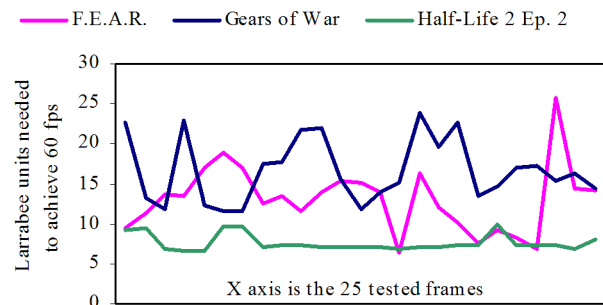

Figure 10: *Overall performance: shows the number of Larrabee Units (cores running at 1 GHz) needed to achieve 60fps for of the series of sample frames in each game.*

The remaining issue that can limit scalability is software locks. Simulating multiple frames of rendering at such a fine level of detail is extremely costly. However, this software rendering pipeline was explicitly designed to minimize the number of locks

and other synchronization events. In general, a lock is obtained and released for each of the following events:

- Twice when a bin set is added to the list of work to do, (once for the front-end queue and once for the back-end queue)
- When a PrimSet is processed by a front-end thread or a tile is processed by a back-end thread
- A few low-frequency locks such as resource creation and deletion, buffer modification by the host CPU, and frame flips or presents.

Modern games usually have significantly less than 10,000 locks per frame. The Larrabee ring network provides relatively good performance for low-contention locks of around 100 clocks per lock per core. Together, these numbers are low enough that lock scaling should be fairly linear with the number of cores, given sufficient memory bandwidth.

5.3 Binning and Bandwidth Studies

We adopted a binning algorithm primarily to minimize software locks, but it also benefits load balancing and memory bandwidth.

Our algorithm assigns back-end tiles to any core that is ready to process one, without attempting to load balance. In theory this could result in significant load imbalance, though cores are free to start processing the next RTset or switch to front-end processing. Bin imbalance is not a problem in the game workloads we have studied. Figure 11 shows a trace of the back-end bin processing time for 16 frames of Gears of War. Each trace records the processing time for each bin on the screen for a frame, sorted from the fastest to slowest bin, and normalized to 1.0 as the mean bin processing time for that frame. Most bins fall in the range ½x to 2x the mean processing time. Few exceed 3x the mean. The other two games produce similar results.
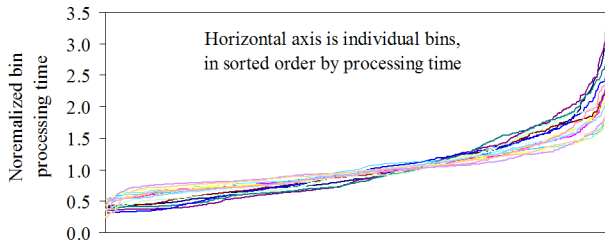
Figure 11: *Bin Balance for Gears of War: each curve shows the time required to process one frame's bins, in sorted order from fastest to slowest, normalized by the mean bin processing time.*

Memory bandwidth is important because the memory subsystem can be one of the more costly and power hungry parts of a GPU, from high end down to low cost designs. It is often a limited resource that can cause bottlenecks if not carefully managed, in part because computational speed scales faster. Our performance studies measure computational speed, unrestricted by memory bandwidth, but it is important to consider how our binning method compares with standard immediate mode rendering algorithms.

Figure 12 compares the total memory bandwidth per frame that we calculated for immediate mode and binned rendering for the three games. The graph presents per-frame data in sorted order from least to most bandwidth for the immediate mode frames. For immediate mode we assume perfect hierarchical depth culling, a 128KB texture cache, and 1MB depth and color caches to represent an ideal implementation. We further assume 2x color and 4x depth compression for single-sampling and 4x color and 8x depth compression for 4-samples per pixel.
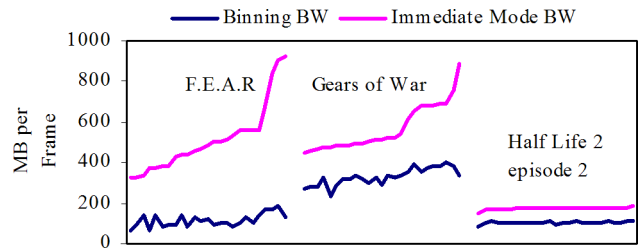
Figure 12: *Bandwidth comparison of binning vs. immediate mode per frame: binning requires bin reads & writes, but eliminates many depth/color accesses that are not detected by hierarchical depth tests. This results in less total bandwidth for binning.*

Immediate mode uses more bandwidth for every tested frame: 2.4x to 7x more for F.E.A.R, , 1.5x to 2.6x more for Gears of War, and 1.6x to 1.8x more for Half Life 2 episode 2. Notably, binning achieves its greatest improvement when the immediate mode bandwidth is highest, most likely because overdraw forces multiple memory accesses in immediate mode. Even with depth culling and frame buffer compression, the 1MB caches are not large enough to catch most pixel overdraw. High resolutions tend to increase the advantage of binning since they increase the impact of pixel access bandwidth on performance.

5.4 Performance Breakdowns

Figure 13 shows the average time spent in each rendering stage for the three games. Pixel shading and interpolant setup is always a major portion of the rendering time, but the balance between different stages can vary markedly in different games. This is illustrated by F.E.A.R, which makes extensive use of stencil-volume shadows. This results in a reduced pixel shading load, but heavy rasterization and depth test loads. This shows the importance of being able to reconfigure the computing resource allocation among different stages, including rasterization, which is 3.2% in two of the games but 20.1% in F.E.A.R.
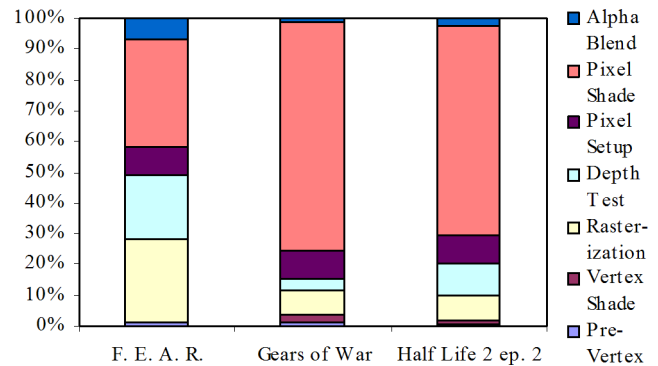
Figure 13: *End-to-End Average Time Breakdowns: shows the average time spent in each rendering stage for the three games*

Figure 14 shows the time spent in each rendering stage for F.E.A.R. For the other two games, the ratios stay very similar across the tested frames, but F.E.A.R. shows significant variation. In addition, considerable variation is observed over the course of a single frame of rendering. Larrabee's cores each process an entire tile at once, then process the next and so on, leading to a reasonably uniform load over the course of the entire frame. By contrast, an immediate-mode renderer doesn't have as many ways to process pixels and primitives out of order. Further, the widely varying loads can cause different units to bottleneck at different times, unless they are over designed for the worst case.
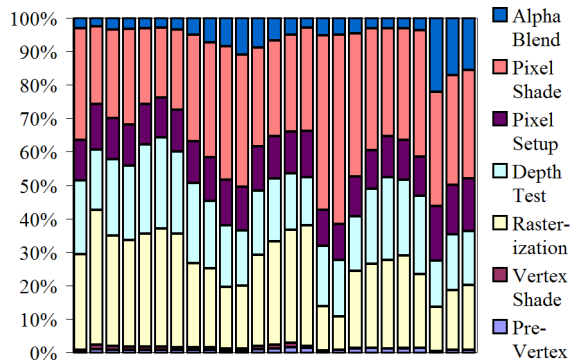
Figure 14: *F.E.A.R. per-frame time breakdowns: this chart shows the time spent in each rendering stage for 25 widely spaced frames of F.E.A.R., which show considerable load variation.*

Our conclusion is that application dependent resource balancing is not sufficient. Instead, dynamic load balancing is likely to be very important to achieving high average performance. Larrabee's entirely software scheduling algorithms provide a great deal of flexibility for adjusting load balancing algorithms.

6. Advanced Applications

Larrabee supports performance implementations of many other parallel applications. Section 6.1 describes how applications can be developed using traditional multi-core high level languages and tools that have been targeted to Larrabee's many-core architecture. Section 6.2 discusses Larrabee support for irregular data structures, which are common in these applications [Pharr 2006]. Sections 6.3 and 6.4 describe results of simulating rendering and other throughput applications on Larrabee.

Scalability and performance analysis in this section uses an in-house simulator that models variable configurations of Larrabee cores, threads, and memory hierarchy. This simulator is derived from proven cycle accurate simulator technology used in the design of CPU cores. Reported data is from hand coded and software threaded kernels running on this simulator.

6.1 Larrabee Many-Core Programming Model

The *Larrabee Native* programming model resembles the well known programming model for x86 multi-core architectures. Central to Larrabee Native programming is a complete C/C++ compiler that statically compiles programs to the Larrabee x86 instruction set. Many C/C++ applications can be recompiled for Larrabee and will execute correctly with no modification. Such application portability alone can be an enormous productivity gain for developers, especially for large legacy x86 code bases like those found in high-performance computing and numeric-intensive computing environments. Two current limitations are that application system call porting is not supported and the current driver architecture requires application recompilation.

We now discuss three important aspects of application programming for Larrabee Native: software threading, SIMD vectorization, and communication between the host and Larrabee.

Larrabee Native presents a flexible software threading capability. The architecture level threading capability is exposed as the well known POSIX Threads API (P-threads) [IEEE 2004]. We have extended the API to also allow developers to specify thread affinity with a particular HW thread or core.

Although P-threads is a powerful thread programming API, its thread creation and thread switching costs may be too high for some application threading. To amortize such costs, Larrabee Native provides a task scheduling API based on a light weight distributed task stealing scheduler [Blumofe et al. 1996]. A production implementation of such a task programming API can be found in Intel Thread Building Blocks [Reinders 2007]. Finally, Larrabee Native provides additional thread programming support via OpenMP [Chandra et al. 2000] pragmas in Larrabee Native's C/C++ compiler.

All Larrabee SIMD vector units are fully programmable by Larrabee Native application programmers. Larrabee Native's C/C++ compiler includes a Larrabee version of Intel's auto-vectorization compiler technology. Developers who need to program Larrabee vector units directly may do so with C++ vector intrinsics or inline Larrabee assembly code.

In a CPU based platform that includes a Larrabee based add-in card, Larrabee will managed by an OS driver for that platform. In such a platform, Larrabee Native binaries are tightly paired with a host binary. Larrabee libraries provide fast message/data passing protocol to manage all memory transfers and communications between the binaries. The API supports both synchronous and asynchronous data transfers. Additionally, execution of some C/C++ standard library functions called from Larrabee application binaries must be shared with the host operating system. Specifically file I/O functions such as read/write/open/close, etc., are proxied from the Larrabee application binary back to a service that executes such functions remotely on the host OS.

Besides high throughput application programming, we anticipate that developers will also use Larrabee Native to implement higher level programming models that may automate some aspects of parallel programming or provide domain focus. Examples include Ct style programming models [Ghuloum et al. 2007], high level library APIs such as Intel® Math Kernel Library (Intel® MKL) [Chuvelev et al. 2007], and physics APIs. Existing GPGPU programming models can also be re-implemented via Larrabee Native if so desired [Buck et al. 2004; Nickolls et al. 2008].

6.2 Irregular Data Structure Support

Larrabee provides excellent support for high throughput applications that use irregular data structures such as complex pointer trees, spatial data structures, or large sparse n-dimensional matrices. They are supported by Larrabee's programming model, memory hierarchy, and VPU instructions.

For Larrabee applications, the multithreaded C++ code to populate, transform, or traverse these data structures follows the familiar programming methodology used on multi-core CPUs. C++ pointers, inheritance, and classes may be used to implement graph nodes. The individual nodes may have significantly different operation execution costs or code branch behavior. Because thread or task scheduling is under programmer control, tasks that operate on these data structures can be dynamically re-bundled to maintain SIMD efficiency. For example, a ray tracer's secondary reflection rays may be re-bundled differently than the primary camera rays that generated them. Finally, data structure techniques such as pre-allocated memory pools can be used to asymmetrically provide only the memory required for a given data structure node. For example, an order-independent transparency implementation may dynamically associate memory based on the number of layers per pixel, rather than pre-allocating a wasteful overestimation of the number of layers per pixel as K-buffer techniques often do [Callahan et al. 2005; Bavoil et al. 2007].

Unlike stream based architectures [Pham et al. 2005; Khailany et al. 2002], Larrabee allows but does not require direct software management to load data into different levels of the memory hierarchy. Software simply reads or writes data addresses, and hardware transparently loads data across the hierarchy. Software complexity is significantly reduced and data structures can employ hard to predict unstructured memory accesses.

In recent Nvidia GPUs, local shared memory support is provided through small (16KB on Nvidia GeForce 8) Per-Block Shared Memories (PBSMs). Each PBSM is shared by 8 scalar processors running up to 768 program instances (which Nvidia calls *threads*) within a SIMD multi-processor [Nickolls et al. 2008]. For high speed local sharing, programmers must explicitly load shared data structures into a PBSM. These are not directly shareable by instances in a different SIMD group. Similarly, order and consistency protection requires software to issue a barrier sync that is visible only within a SIMD group. To facilitate broader sharing across SIMD groups, data must be explicitly written out to higher latency GDDR memory. In contrast, all memory on Larrabee is shared by all processor cores. For Larrabee programmers, local data structure sharing is transparently supported by the coherent cached memory hierarchy regardless of the thread's processor. Protection can be provided by conventional software locks, semaphores, or critical sections.

An important aspect of handling irregular data structures is efficient scatter-gather support, so that the SIMD VPU can work on non-contiguous data. As described in Section 3, Larrabee implements VPU scatter-gather instructions which load a VPU vector register from sixteen non-contiguous memory locations. The non-contiguous data elements can reside anywhere in the large on-die cache, without suffering memory access penalty. This significantly reduces programmer data management overhead. We have observed an average of almost 3x performance gain in Larrabee from hardware support of scatter-gather, compared to software scatter-gather, for basic sparse matrix compute kernels, such as sparse matrix-vector multiply. Algorithms requiring irregular data structures also benefit from Larrabee instructions such as count bits, bit scan, and packed loads and stores.

6.3 Extended Rendering Applications

The Larrabee graphics rendering pipeline is itself a Larrabee Native application. Because it is software written with high level languages and tools, it can easily be extended to add innovative rendering capabilities. Here we briefly discuss three example extensions of the graphics pipeline that we are studying. Future implementations could evolve towards a fully programmable graphics pipeline as outlined by Pharr [2006].

Render Target Read: Because Larrabee's graphics rendering pipeline employs a software frame buffer, we can enable additional programmer access to those data structures. More specifically, a trivial extension to the Larrabee rendering pipeline would be to allow pixel shaders to directly read previously stored values in render targets. Such a capability could serve a variety of rendering applications, including programmer defined blending operations, single-pass tone mapping, and related functions.

Order Independent Transparency: Presently 3D application developers must either depth sort translucent models in their application every frame prior to rendering or else implement multi-pass algorithms such as depth peeling [Wexler et al. 2005] to achieve correct inter-model transparency. Neither method allows the kinds of post-rendering area effects that are possible

with opaque models. Figure 15 illustrates artifacts that occur if such effects are applied after merging the translucent surfaces.

Figure 15: Transparency without and with pre-resolve effects: the left image sorts the geometry and resolves before applying a fog patch. The right image applies the fog patch to the translucent surfaces and then resolves. The fog is visible through the wing in the right image, but not in the left image. (Dragon models designed and created by Jeffery A. Williams and Glen Lewis.)

Larrabee can support order independent transparency (OIT) with no additional dedicated logic by storing multiple translucent surfaces in a per-pixel spatial data structure. After rendering the geometry, we can perform effects on the translucent surfaces, since each surface retains its own depth and color, before sorting and resolving the fragment samples stored per pixel.

Figure 16: Irregular Z-Buffer sample frame: this method uses an irregular spatial data structure to produce alias-free shadowing. Like the transparency example in figure 15, the data structure is tightly integrated with the rendering pipeline. The renderer constructs the shadowmap and then treats it as a special class of frame buffer. (Skeleton model by TurboSquid.)

Irregular Shadow Mapping: Shadow mapping is a popular real-time shadow approximation technique, but most implementations are plagued by visually displeasing aliasing artifacts. A variety of heuristics have been proposed in an attempt to reduce artifacts [Akenine-Möller & Haines 2002; Bookout 2007]. Irregular shadow mapping (ISM) offers an exact solution to this problem and places no additional burden on the application programmer [Aila & Laine 2004; Johnson et al. 2005].

To implement ISM, we dynamically construct a spatial data structure in the light view using depth samples captured in the camera view. We again customize Larrabee's all software graphics pipeline by adding a stage that performs light view ISM rasterization against ISM's spatial data structure. Because the shadow map is computed at exact positions, the resulting shadow map is alias free. This technique can be used to achieve real-time hard shadowing effects, as shown in Figure 16, or as the foundation for real-time soft shadowing effects.

6.4 Other Throughput Computing Applications

Larrabee is also suitable for a wide variety of non-rasterization based throughput applications. The following is a brief discussion of the observed scalability and characteristics of several examples.
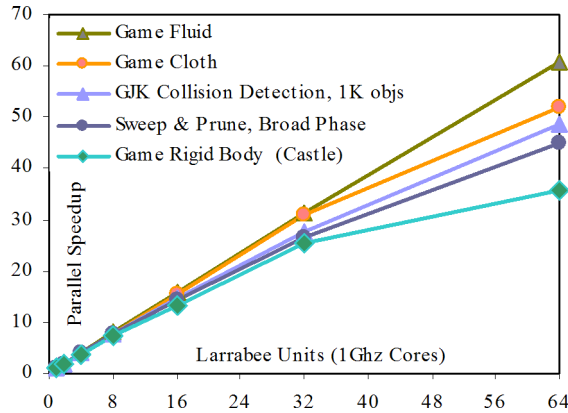

Figure 17: *Game Physics Scalability Performance: this shows that the Larrabee architecture is scalable to meet the growing performance needs of interactive rigid body, fluid, and cloth simulation algorithms and some commonly used collision kernels.*

Game Physics: We have performed detailed scalability simulation analysis of several game physics workloads on various configurations of Larrabee cores. Figure 17 shows scalability of some widely used game physics benchmarks and algorithms for rigid body, fluid, and cloth. We achieve better than 50% resource utilization using up to 64 Larrabee cores, and achieve near-linear parallel speedup is some cases. The game rigid body simulation is based on the popular "castle" destruction scene with 10K objects. Scalability plots for Sweep-and-Prune [Cohen et al. 1995] and *GJK* [Gilbert et al. 1988] distance algorithms are included since they are some of the most commonly used collision detection routines. Game fluid simulation is based on the smoothed particle hydrodynamics (SPH) algorithm. We used a mass spring model and Verlet integration for our game cloth simulation [Jacobsen 2001]. Bader et al. [2008] provide details on the implementation and scalability analysis for these game physics workloads
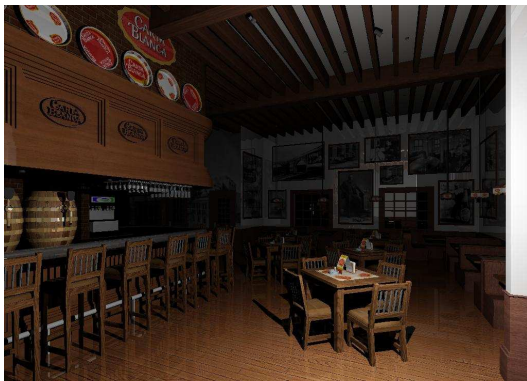

Figure 18: *Real time ray tracing on Larrabee: cropped from a 1Kx1K sample image that requires ~4M rays. The ray tracer was implemented in C++ with some hand-coded assembly code for key routines like ray intersection. Kd-trees are typically 25MB and are built dynamically per frame. Primary and reflection rays are tested in 16 ray bundles. Nearly all 234K triangles are visible to primary or reflection rays. (Bar Carta Blanca model by Guillermo M Leal Llaguno, courtesy of Cornell University.)*

Real Time Ray Tracing: The highly irregular nature of spatial data structures used in Whitted style real-time ray tracers benefit from Larrabee's general purpose memory hierarchy, relatively short pipeline, and VPU instruction set. Here we used SIMD 16 packet ray tracing traversing through a kd-tree. For the complete workload, we observe that a single Intel Core 2 Duo processor requires 4.67x more clock cycles than a single Larrabee core, which shows the effectiveness of the Larrabee instruction set and wide SIMD. Results are even better for small kernels. For example, the intersection test of 16 rays to 1 triangle takes 47 cycles on a single Larrabee core. The same test takes 257 Core 2 Duo processor cycles. Figure 18 shows a 1024x1024 frame of the bar scene with 234K triangles, 1 light source, 1 reflection level, and typically 4M rays per frame. Figure 19 compares performance for Larrabee with an instance of the ray tracer running on an Intel Xeon® processor 2.6GHz with 8 cores total. Shevtsov et al. [2007] and Reshetov et al. [2005] describe details of this implementation.
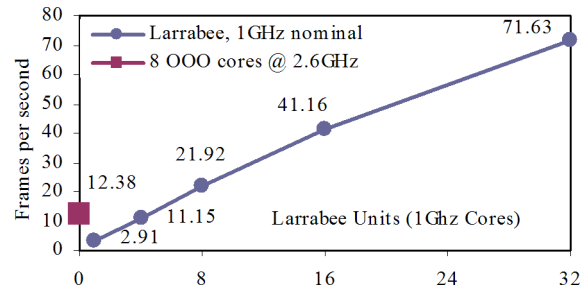

Figure 19: *Real time ray tracing scalability: this graph compares different numbers of Larrabee cores with a nominal 1GHz clock speed to an Intel Xeon processor 2.6GHz with 8 cores total. The latter uses 4.6x more clock cycles than are required by 8 Larrabee cores due to Larrabee's wide VPU and vector instruction set. Figure 18 describes the workload for these tests.*

Image and Video Processing: The Larrabee architecture is suitable for many traditional 2D image and video analysis applications. Native implementations of traditional 2D filtering functions (both linear and non-linear) as well as more advanced functions, like video cast indexing, sports video analysis, human body tracking, and foreground estimation offer significant scalability as shown in Figure 20. Biomedical imaging represents an important subset of this processing type. Medical imaging needs such as back-projection, volume rendering, automated segmentation, and robust deformable registration, are related yet different from those of consumer imaging and graphics. Figure 20 also includes scalability analysis of iso-surface extraction on a 3D volume dataset using the marching cubes algorithm.

Physical Simulation: Physical simulation applications use numerical simulation to model complex natural phenomena in movies and games, such as fire effects, waterfalls in virtual worlds, and collisions between rigid or deformable objects. Large data-sets, unstructured control-flow and data accesses often make these applications more challenging to scale than traditional streaming applications. Looking beyond interactive game physics, we also analyzed applicability of Larrabee architecture for the broader class of entertainment physics including offline movie-industry effects and distributed real-time virtual-world simulation. Specific simulation results based on Stanford's PhysBAM are shown in Figure 20 and illustrate very good scalability for production fluid, production cloth, and production face. Implementation and scalability analysis details are described by Hughes et al. [2007].
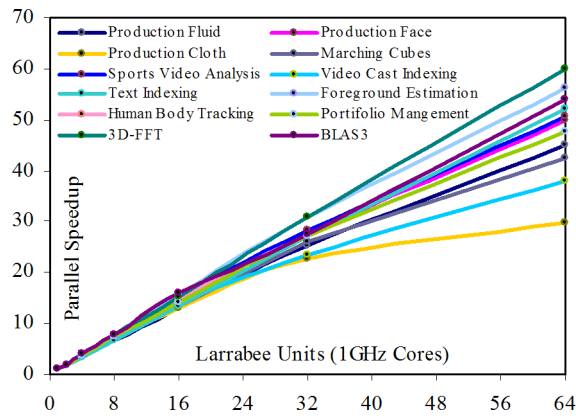
Figure 20: *Scalability of select non-graphics applications and kernels: Larrabee's general-purpose many-core architecture delivers performance scalability for various non-graphics visual and throughput computing workloads and common HPC kernels.*

Larrabee is also highly scalable for non-visual throughput applications, as shown in Figure 20. Larrabee's highly-threaded x86 architecture benefits traditional enterprise throughput computing applications, such as *text indexing*. Its threading, together with its wide-SIMD IEEE-compliant double-precision support, makes it well positioned for financial analytics, such as *portfolio management*. Internal research projects have proven Larrabee architecture scalability for many traditional high performance computing (HPC) workloads and well-known HPC kernels such as *3D-FFT* and *BLAS3* (with dataset larger than on-die cache). More details are described by Chen et al. [2008].

7. Conclusions

We have described the Larrabee architecture, which uses multiple x86-based CPU cores, together with wide vector processor units and some fixed function logic, to achieve high performance and flexibility for interactive graphics and other applications. We have also described a software renderer for the Larrabee architecture and a variety of other throughput applications, with performance and scalability analysis for each. Larrabee is more programmable than current GPUs, with fewer fixed function units, so we believe that Larrabee is an appropriate platform for the convergence of GPU and CPU applications.

We believe that this architecture opens a rich set of opportunities for both graphics rendering and throughput computing. We have observed a great deal of convergence towards a common core of computing primitives across the workloads that we analyzed on Larrabee. This underlying workload convergence [Chen et al. 2008] implies potential for a common programming model, a common run-time, and a native Larrabee implementation of common compute kernels, functions, and data structures.

References

AKENINE-MÖLLER, T., HAINES, E. 2002. *Real-Time Rendering*. 2nd Edition. A. K. Peters.

AILA, T., LAINE, S. 2004. Alias-Free Shadow Maps. In *Proceedings of Eurographics Symposium on Rendering 2004, Eurographics Association*. 161-166.

ALPERT, D., AVNON, D. 1993. Architecture of the Pentium Microprocessor. *IEEE Micro, v.13*, n.3, 11-21. May 1993.

AMD. 2007. Product description web site: ati.amd.com/products/Radeonhd3800/specs.html.

BADER, A., CHHUGANI, J., DUBEY, P., JUNKINS, S., MORRISON T., RAGOZIN, D., SMELYANSKIY. 2008. Game Physics Performance On Larrabee Architecture. Intel whitepaper, available in August, 2008. Web site: techresearch.intel.com.

BAVOIL, L., CALLAHAN, S., LEFOHN, A., COMBA, J. SILVA, C. 2007. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (Seattle, Washington, April 30 - May 02, 2007). I3D 2007. ACM, New York, NY, 97-104.

BLUMOFE, R., JOERG, C., KUSZMAUL, B., LEISERSON, C., RANDALL, K., ZHOU, Y. Aug. 25, 1996. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing, v. 37*, i. 1, 55–69.

BLYTHE, D. 2006. The Direct3D 10 System. *ACM Transactions on Graphics*, 25, 3, 724-734.

BOOKOUT, D. July, 2007. Shadow Map Aliasing. Web site: www.gamedev.net/reference/articles/article2376.asp.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics, v. 23,* n. 3, 777-786.

CALLAHAN, S., IKITS, M., COMBA, J., SILVA, C. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11, 3, 285–295

CHANDRA, R., MENON, R., DAGUM, L., KOHR, D, MAYDAN, D., McDONALD, J. 2000. *Parallel Programming in OpenMP*. Morgan Kaufman.

CHEN, M., STOLL, G., IGEHY, H., PROUDFOOT, K., HANRAHAN P. 1998. Simple models of the impact of overlap in bucket rendering. In *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware* (Lisbon, Portugal, August 31 - September 01, 1998). S. N. Spencer, Ed. HWWS '98. ACM, New York, NY, 105-112.

CHEN, Y., CHHUGANI, J., DUBEY, P., HUGHES, C., KIM, D., KUMAR, S., LEE, V., NGUYEN A., SMELYANSKIY, M. 2008. Convergence of Recognition, Mining, and Synthesis Workloads and its Implications. In *Procedings of IEEE, v. 96*, n. 5, 790-807.

CHUVELEV, M., GREER, B., HENRY, G., KUZNETSOV, S., BURYLOV, I., SABANIN, B. Nov. 2007. Intel Performance Libraries: Multi-core ready Software for Numeric Intensive Computation. *Intel Technology Journal, v. 11*, i. 4, 1-10.

COHEN, J., LIN., M., MANOCHA, D., PONAMGI., D. 1995. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*. SI3D '95. ACM, New York, NY, 189-196.

ELDRIDGE, M. 2001. *Designing Graphics Architectures Around Scalability and Communication*. PhD thesis, Stanford.

FOLEY, J., VAN DAM, A., FEINER, S., HUGHES, J. 1996. *Computer Graphics: Principles and Practice.* Addison Wesley.

FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., ISRAEL, L. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, ACM, 79-88.

GHULOUM, A., SMITH, T., WU, G., ZHOU, X., FANG, J., GUO, P., SO, B., RAJAGOPALAN, M., CHEN, Y., CHEN, B. November 2007. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architectures. *Intel Technology Journal, v. 11*, i. 04, 333-348.

GILBERT, E., JOHNSON, D., KEERTHI, S. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation, 4*, 2, 193–203.

GPGPU. 2007. GPGPU web site: www.gpgpu.org.

GREENE, N. 1996. Hierarchical polygon tiling with coverage masks, In *Proceedings of ACM SIGGRAPH 93*, ACM Press/ACM SIGGRAPH, New York, NY, Computer Graphics Proceedings, Annual Conference Series, ACM, 65-64.

GROCHOWSKI, E., RONEN, R., SHEN, J., WANG, H. 2004. Best of Both Latency and Throughput. *2004 IEEE International Conference on Computer Design (ICCD '04)*, 236-243.

GWENNAP, L. 1995. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report. v. 9*, n. 2, Feb. 16, 1995.

HSIEH, E., PENTKOVSKI, V., PIAZZA, T. 2001. ZR: A 3D API Transparent Technology For Chunk Rendering. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Austin, Texas, December 01 - 05, 2001). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 284-291.

HUGHES, C. J., GRZESZCZUK, R., SIFAKIS, E., KIM, D., KUMAR, S., SELLE, A. P., CHHUGANI, J., HOLLIMAN, M., AND CHEN, Y. 2007. Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors. In *Proceedings of the 34th Annual international Symposium on Computer Architecture* (San Diego, California, USA, June 09 - 13, 2007). ISCA '07. ACM, New York, NY, 220-231.

IEEE Std. 1003.1, 2004 Edition. Standard for Information Technology – Portable Operating System Interface (POSIX) System Interfaces. *The Open Group Technical Standard Base Specifications*. Issue 6.

JACOBSEN, T. 2001. Advanced Character Physics. *Proc. Game Developers Conference 2001*, 1-10.

JOHNSON, G.S., LEE, J., BURNS, C.A., MARK, W.R. 2005. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics. 24*, 4, 1462-1482.

KELLEY, M., GOULD, K., PEASE, B., WINNER, S., YEN, A. 1994. Hardware accelerated rendering of CSG and transparency. In *Proceedings of SIGGRAPH 1994,* ACM Press/ACM SIGGRAPH, New York, NY, Computer Graphics Proceedings, Annual Conference Series, ACM, 177-184.

KELLEY, M., WINNER, S., GOULD, K. 1992. A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm. In *Computer Graphics (Proceedings of ACM SIGGRAPH 1992)*, SIGGRAPH '92. ACM, New York, NY, 241-248.

KESSENICH, J., BALDWIN, D., ROST, R. The OpenGL Shading Language. Version 1.1. Sept. 7, 2006. Web site: www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

KHAILANY, B., DALLY, W., RIXNER, S., KAPASI, U., MATTSON, P., NAMKOONG, J., OWENS, J., TOWLES, B., CHANG, A. 2001. Imagine: Media Processing with Streams. *IEEE Micro, 21*, 2, 35-46.

KONGETIRA, P., AINGARAN, K., OLUKOTUN, K. Mar/Apr 2005. Niagara: A 32-way multithreaded SPARC Processor. *IEEE Micro. v. 25*, i. 2. 21-29.

LAKE, A. 2005. Intel Graphics Media Accelerator Series 900 Developer's Guide. Version 2.0. Web site: download.intel.com/ids/gma/Intel_915G_SDG_Feb05.pdf.

LLOYD, B., GOVINDARAJU, N., MOLNAR, S., MANOCHA, D. 2007. Practical logarithmic rasterization for low-error shadow maps. In *Proceedings of the 22nd ACM SIGGRAPH/ EUROGRAPHICS Symposium on Graphics Hardware*, 17-24.

MARK, W., GLANVILLE, S., AKELEY, K., KILGARD, M. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language, *ACM Transactions on Graphics, v. 22*, n. 3, 896-907.

MICROSOFT. 2007. Microsoft Reference for HLSL. Web site: msdn2.microsoft.com/en-us/library/bb509638.aspx.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications, v.14*, n. 4, July 1994, 23-32.

MOLNAR, S., EYLES, J., POULTON, J. 1992. Pixelflow: High Speed Rendering Using Image Composition. *Computer Graphics (Proceedings of SIGGRAPH 92), v. 26*, n. 2, 231-240.

MOREIN, S. 2000. ATI Radeon HyperZ Technology. Presented at *Graphics Hardware 2000*. Web site: www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf.

NICKOLLS, J., BUCK, I., GARLAND, M. 2008. Scalable Parallel Programming with CUDA. *ACM Queue, 6*, 2, 40-53.

NVIDIA . 2008. Product description web site: www.nvidia.com/object/geforce_family.html.

OWENS, J., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUGER, J., LEFOHN, A., PURCELL, T. 2007. A Survey of General Purpose Computation on Graphics Hardware. *Computer Graphics Forum. v.26*, n. 1, 80-113.

PHAM D., ASANO, S., BOLLIGER,M., DAY, M., HOFSTEE, H., JOHNS.,C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIASK, D., SUZUODI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., YAZAWA, K. 2005. The Design and Implementation of a First Generation CELL Processor. *IEEE International Solid-State Circuits Conference*. 184-186.

PHARR, M. 2006. Interactive Rendering in the Post-GPU Era. Presented at *Graphics Hardware 2006*. Web site: www.pharr.org/matt/.

PINEDA, J. 1988. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88), 22*, 4, 17-20.

POWERVR. 2008. Web site: www.imgtec.com/powervr/products/Graphics/index.asp.

POLLACK, F. 1999. New Microarchitecture Challenges for the Coming Generations of CMOS Process Technologies. *Micro32*.

REINDERS, J., 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reily Media, Inc.

RESHETOV A., SOUPIKOV, A., HURLEY, J. 2005. Multi-level Ray Tracing Algorithm. *ACM Transactions on Graphics, 24*, 3, 1176-1185.

ROST, R. 2004. *The OpenGL Shading Language*. Addison Wesley.

SHEVTSOV, M., SOUPIKOV, A., KAPUSTIN, A. 2007. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Proceedings of GraphiCon 2007,* 33-39.

STEVENS, A. 2006. ARM Mali 3D Graphics System Solution. Web site: www.arm.com/miscPDFs/16514.pdf.

STOLL, G., ELDRIDGE, M., PATTERSON, D., WEBB, A., BERMAN, S., LEVY, R., CAYWOOD, C., TAVEIRA, M., HUNT, S., HANRAHAN, P. 2001. Lightning 2: A High Performance Display Subsystem for PC Clusters. In *Computer Graphics* (*Proceedings of ACM SIGGRAPH 2001)*, ACM, 141-148.

TORBORG, J., KAJIYA, J. 1996. Talisman Commodity Realtime 3D Graphics for the PC. In *Proceedings of ACM SIGGRAPH* 1996, ACM Press/ACM SIGGRAPH, New York. Computer Graphics Proceedings, Annual Conference Series, ACM, 353-363.

WEXLER, D., GRITZ, L., ENDERTON, E., RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Los Angeles, California, July 30 - 31, 2005). HWWS '05, ACM, New York, NY, 7-14.