Neon: A Single-Chip 3D Workstation Graphics Accelerator

Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi & Ken Correll
Digital Equipment Corporation

Abstract

High-performance 3D graphics accelerators traditionally require multiple chips on multiple boards, including geometry, rasterizing, pixel processing, and texture mapping chips. These designs are often scalable: they can increase performance by using more chips. Scalability has obvious costs: a minimal configuration needs several chips, and some configurations must replicate texture maps. A less obvious cost is the almost irresistible temptation to replicate chips to increase performance, rather than to design individual chips for higher performance in the first place.

In contrast, Neon is a single chip that performs like a multichip design. Neon accelerates OpenGL [19] 3D rendering, as well as X11 [20] and Windows/NT 2D rendering. Since our pin budget limited peak memory bandwidth, we designed Neon from the memory system upward in order to reduce bandwidth requirements. Neon has no special-purpose memories; its eight independent 32-bit memory controllers can access color buffers, Z depth buffers, stencil buffers, and texture data. To fit our gate budget, we shared logic among different operations with similar implementation requirements, and left floating point calculations to Digital's Alpha CPUs. Neon's performance is between HP's Visualize fx^4 and fx^6, and is well above SGI's MXE for most operations. Neon-based boards cost much less than these competitors, due to a small part count and use of commodity SDRAMs.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture – Graphics processors; I.3.3 [Computer Graphics]: Picture/Image Generation – Line and curve generation; I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism – Color, shading, shadowing, and texture; B.3.2 [Memory Structures]: Design Style – Cache memories

Additional Keywords: graphics pipeline, rasterization, chunk rendering, tile rendering, texture cache, level of detail, direct rendering

1. INTRODUCTION

Neon borrows much from Digital's Smart Frame Buffer [14] family of chips, in that it extracts a large proportion of the peak memory bandwidth from a unified frame buffer, accelerates only rendering operations, and efficiently uses a general I/O bus.

Neon makes efficient use of memory bandwidth by reducing page crossings, prefetching pages, and batching operations to reduce memory bus turnaround. It uses about 70% of its 3.2 gigabytes/second of bandwidth for rendering and screen refresh, wasting 30% in overhead cycles that don't transfer data. Neon supports 32 to 128 megabytes of SDRAM; the 128 megabyte configuration has over 100 megabytes available for textures, and can store a 512 x 512 x 256 3D 8-bit intensity texture.

Unlike most fast workstation accelerators, Neon does not accelerate floating-point operations. Digital's 500 MHz 21164A Alpha CPU [5] processes 1.5 to 4 million lit vertices per second. We expect the 600 MHz 21264 Alpha [7] [10] to process 2.5 to 6 million vertices per second, and faster Alpha CPUs are coming.

Since Neon accepts vertex data after lighting computations, it requires as little as 12 bytes/vertex for (x, y, z, r, g, b, a) information. A 32-bit, 33 MHz PCI supports over 8 million such vertices/second; a 64-bit PCI should support 15 million vertices/second. The 64-bit PCI allows texture downloads at 200 megabytes/second, and the large-memory configurations should allow many textures to stay in the frame buffer across several frames. We thus saw no need for a special-purpose graphics bus.

Neon targets many of the capabilities of the SGI RealityEngine [1], with the exception of polygon antialiasing. Neon accelerates rendering of Z-buffered Gouraud shaded, trilinear perspective-correct texture-mapped triangles and lines. Neon supports antialiased lines, Microsoft Windows lines, and X11 wide lines.

Performance goals were 4 million 25-pixel, shaded, Z-buffered triangles/second, 2.5 million 50-pixel triangles/second, and 600,000 to 800,000 50-pixel textured triangles/second. Very early in the design, we traded increased gate count for reduced design time, increasing the Gouraud shaded, Z-buffered triangle setup rate from a planned 3.1 million per second to over 7 million per second. This decision proved fortunate—applications are using ever smaller triangles, and the software team doubled their original estimates of vertex processing rates.

2. WHY A SINGLE CHIP?

A single chip's pin count constrains peak memory bandwidth, while its die size constrains gate count. But there are compensating implementation, cost, and performance advantages over a multichip accelerator.

A single-chip accelerator is easier to design. Partitioning the frame buffer across multiple chips forces copy operations to move data from chip to chip, increasing complexity, logic duplication, and pin count. In contrast, internal wires switch faster than pins and allow wider interfaces (our Fragment Generator ships nearly 600 bits downstream). And changing physical pin interfaces is harder than changing internal interfaces.

A single-chip accelerator uses fewer gates, as operations with similar functionality can share generalized logic. For example, copying pixel data requires computing source addresses, reading data, converting it to the correct format, shifting, and writing to a group of destination addresses. Texture mapping requires computing source addresses, reading data, converting it, filtering, and writing to a destination address. In Neon, pixel copying and texture mapping share source address computation, a small cache for texel and pixel reads, read request queues, format conversion, and destination steering. In addition, pixel copies, texture mapping, and pixel fill operations use the same destination queues and source/destination blending logic. And unlike some PC accelera-

Joel McCormack and Norman Jouppi are at Digital's Western Research Lab, Robert McNamara is at Digital's Systems Research Center, and Christopher Gianos is in Digital's Semiconductor group, e-mail at [First.Last@digital.com]. Larry Seiler and Ken Correll, formerly of Digital, now at Mitsubishi Electric Research Laboratory, e-mail at {seiler,correll}@merl.com.

1998 Workshop on Graphics Hardware Lisbon Portugal

tors, both 2D and 3D operations share the same paths through the chip.

This sharing amplifies design optimization efforts. For example, the chunking fragment generation described below in Section 5.2.5 decreases SDRAM page crossings. By making the chunk size programmable, we also increased the hit rate of the texture cache. The texture cache, in turn, was added to decrease texture bandwidth requirements—but also improves the performance of 2D tiling and copying overlay pixels.

A single-chip accelerator can provide more memory for texture maps at lower cost. For example, a fully configured RealityEngine replicates the texture map 20 times for the 20 rasterizing chips; you pay for 320 megabytes of texture memory, but applications see only 16 megabytes. A fully configured InfiniteReality [17] replicates the texture four times—but each rasterizing board uses a redistribution network to fully connect 32 texture RAMs to 80 memory controllers. In contrast, Neon doesn't replicate texture maps, and uses a simple 8 x 8 crossbar to redistribute texture data internally. The 64 megabyte configuration has over 40 megabytes available for textures after allocating 20 megabytes to a 1280 x 1024 display.

Neon is a large chip. Its die size is 17.3 x 17.3 mm, using IBM's 0.35 μm CMOS 5S standard cell process with 5 metal layers [9]. (Their 0.25 μm 6S technology would reduce this to about 12.5 x 12.5 mm, and 0.18 μm 7S would further reduce this to about 9 x 9 mm.) The design uses 6.8 million transistors and runs at 100 MHz. The chip has 609 signal pins, packaged in an 824-pin ceramic column grid array. The 8 memory controllers each use 32 data pins and 23 address/control pins, for a total of 440 signal pins to memory. The direct interface to a 64-bit PCI uses 90 pins, and the RAMDAC interface uses 68 pins.

3. WHY A UNIFIED MEMORY SYSTEM?

Neon differs from many workstation accelerators in that it has a single, general graphics memory system to store colors, Z depths, textures, and off-screen buffers.

The biggest advantage of a single memory system is the dynamic reallocation of memory bandwidth. Dedicated memories imply a dedicated partitioning of memory bandwidth—and wasting of bandwidth dedicated to functionality currently not in use. If Z buffering or texture mapping is not enabled, Neon has more bandwidth for the operations that are enabled. Further, partitioning of bandwidth changes instantaneously at a fine grain. If texel fetches overlap substantially in a portion of a scene, so that the texture cache's hit rate is high, more bandwidth becomes available for color and Z accesses. If many Z buffer tests fail, and so color and Z data writes occur infrequently, more bandwidth becomes available for Z reads. This automatic allocation of memory bandwidth enables us to design closer to average memory bandwidth requirements than to the worst case.

A unified memory system offers flexibility in memory allocation. For example, users can specify 16-bit colors rather than 32-bit colors; this gains 7.5 megabytes for textures when using a 1280 x 1024 screen.

A unified memory system also offers greater potential for sharing logic. For example, the sharing of copy and texture map logic described above in Section 2 is possible only if textures and pixels are stored in the same memory.

A unified memory system has one major drawback—texture mapping may cause page thrashing as memory accesses alternate between texture data and color/Z data. Neon reduces such thrashing in several ways. Neon's deep memory request and reply queues fetch large batches of texels and pixels, so that switching occurs infrequently. The texel cache and fragment generation chunking ensure that the texel request queues contain few duplicate requests, so that they fill up slowly and can be serviced infrequently. The memory controllers prefetch texel and pixel pages when possible to minimize switching overhead. Finally, the four SDRAM banks available on the 64 and 128 megabyte configurations usually eliminate thrashing, as texture data is stored in different banks from color/Z data. These techniques are discussed in greater detail in Section 4 below.

SGI's O2 [13] carries unification one step further, by using the CPU's system memory for graphics data. But roughly speaking, CPU performance is usually limited by memory latency, while graphics performance is usually limited by memory bandwidth, and different techniques must be used to address these limits. We believe that the substantial degradation in both graphics and CPU performance caused by a completely unified memory isn't worth the minor cost savings.

4. IS NEON JUST ANOTHER PC ACCELERATOR?

A single chip connected to a single frame buffer memory with no floating point acceleration may lead some readers to conclude "Neon is like a PC accelerator." The dearth of hard data on PC accelerators makes it hard to compare Neon to these architectures, but we feel a few points are important to make.

Neon is in a different performance class from PC accelerators. Without floating point acceleration, PC accelerators are limited by the slow vertex transformation rates of Pentium CPUs. Many PC accelerators also burden the CPU with computing and sending slope and gradient information for each triangle; Neon uses an efficient packet format that supports strips, and computes triangle setup information directly from vertex data. Neon does not require the CPU to sort objects into different chunks like Talisman [3][21], nor does it suffer the overhead of constantly reloading texture map state for the different objects in each chunk.

Neon directly supports much of the OpenGL rendering pipeline, and this support is general and orthogonal. Enabling one features does not disable other features, and does not affect performance unless the feature requires more memory bandwidth. For example, Neon can render OpenGL lines that are both wide and dashed. Neon supports all OpenGL 1.2 source/destination blending modes, and both exponential and exponential squared fog modes. All pixel and texel data are accurately computed, and do not use gross approximations such as a single fog or mip-map level per object, or a mip-map level interpolated across the object. And all three 3D texture coordinates are perspective correct.

5. ARCHITECTURE

Neon's performance isn't the result of any one great idea, but rather many good ideas—some old, some new—working synergistically. Some key components to Neon's performance are:

- a unified memory to reduce idle memory cycles,
- a large peak memory bandwidth (3.2 gigabytes/second with 100 MHz SDRAM),
- the partitioning of memory among 8 memory controllers, with fine-grained load balancing,
- the batching of fragments to amortize read latencies and bus turnaround cycles, and to allow prefetching of pages to hide precharge and row activate overhead,
- chunked mappings of screen coordinates to physical addresses, and chunked fragment generation, which reduce page crossings and increase page prefetching,
- a screen refresh policy in which memory controllers overlap refresh page crossings with other work,
- a small texel cache and chunked fragment generation to increase the cache's hit rate,
- deeply pipelined triangle setup logic and a high-level interface with minimal software overhead,

- multiple formats for vertex data, which allow software to trade CPU cycles for I/O bus cycles,
- the ability for applications to map OpenGL calls to Neon commands, without the inefficiencies usually associated with such direct rendering.

Section 5.1 below briefly describes Neon's major functional blocks in the order that it processes commands, from the bus interface on down. Sections 5.2 to 5.6, however, provide more detail in roughly the order we designed Neon, from the memory system on up. We hope that this better conveys how we first made the memory system efficient, then constantly strove to increase that efficiency as we moved up the rendering pipeline.

5.1. Architectural Overview

Figure 1 shows a block diagram of the major functional units of Neon.

The PCI logic supports 64-bit transfers at 33 MHz. Neon can act as a bus master, and so can initiate DMA requests to read or write main memory.

The PCI logic forwards command packets and DMA data to the Command Parser. The CPU can write commands directly to Neon via Programmed I/O (PIO), or Neon can DMA commands from main memory. The parser accepts nearly all OpenGL [19] object types, including line, triangle, and quad strips, so that CPU cycles and I/O bus bandwidth aren't wasted by duplication of vertex data. Finally, the parser oversees DMA operations from the frame buffer to main memory, hence the connection from Texel Central.

The Fragment Generator performs object setup and traversal. The Fragment Generator uses half-plane edge functions [18] to determine object boundaries, and moves a fragment stamp inside each object in an order that enhances the efficiency of the memory system. Each cycle, the stamp generates a single textured fragment, a 2 x 2 square of 64-bit Z-buffered fragments, or up to 8 32-bit or 32 8-bit fragments along a scanline. (A fragment contains the information required to paint one pixel.) When generating a 2 x 2 block of fragments, the stamp interpolates six channels for each fragment: red, green, blue, alpha transparency, Z depth, and fog intensity. When generating a single texture-mapped fragment, the stamp interpolates eight additional channels: three texture coordinates, the perspective correction term, and the four derivatives needed to compute the mip-mapping level of detail. Setup time depends upon the number of channels, ranging from over 7 million RGBZ triangles/second to just over 2 million triangles/second for all 14 channels. The Fragment Generator tests fragments against four clipping rectangles (which may be inclusive or exclusive), and sends visible fragments to Texel Central.

Texel Central was named after Grand Central Station, as it provides a crossbar between memory controllers. Any data that is read from the frame buffer in order to derive data that is written to a different location goes through Texel Central. This includes texture mapping, copies within the frame buffer, and DMA transfers to main memory. Texel Central also expands an internal 32 x 32 stipple pattern or an externally supplied stipple into 256 bits of color information for 2D fill operations, generating 800 million 32-bit RGBA fragments/second or 3.2 billion 8-bit Index fragments/second.

Texture mapping is performed at a peak rate of one fragment per clock cycle *before* a Pixel Processor tests the Z value. This wastes bandwidth by fetching texels that are obscured, but pre-textured fragments require about 350 bits and post-textured fragments need about 100 bits. We didn't have enough real estate for more and wider fragment queues to texture map after the Z depth test. Further, the Z value of a textured fragment cannot be written until after the textured color has passed OpenGL's Alpha test. Such a wide separation between reading and writing Z values

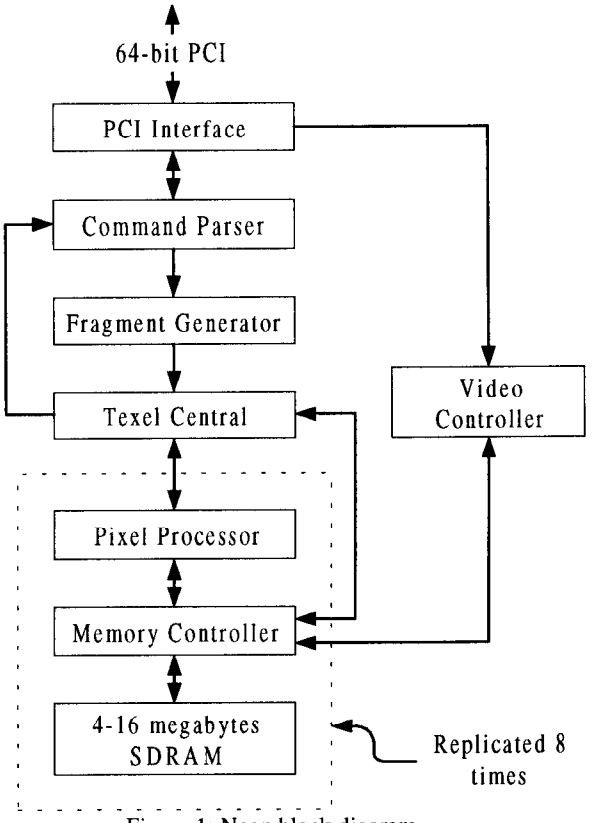

Figure 1: Neon block diagram

would significantly complicate maintaining frame buffer consistency, as described in Section 5.2.2 below.

Texel Central feeds fragments to the eight Pixel Processors, each of which has a corresponding Memory Controller. The Pixel Processors handle the back end of the OpenGL rendering pipeline: alpha, stencil, and Z depth tests; fog; source and destination blending (including raster ops); and dithering.

The Video Controller refreshes the screen, which can be up to 1600 x 1200 pixels at 76 Hz. It requests pixel data for specific screen locations from each Memory Controller, and yanks a "do it now" flag if a Memory Controller is too nonchalant about satisfying the request. Each Memory Controller autonomously reads and interprets overlay and display format bytes, and if the overlay is transparent it reads data from the front, back, left, or right color buffer. The Video Controller sends low color depth pixels (5/5/5 and 4/4/4) through "inverse dithering" logic, which uses an adaptive digital filter to restore much of the original color information. Finally, the controller sends the filtered pixels to an external RAMDAC for conversion to an analog video signal.

Frame buffer memory is partitioned equally among the eight Memory Controllers. Each controller has five request queues: Source Read Request from Texel Central, Pixel Read and Pixel Write Request from its Pixel Processor, and two Refresh Read Requests (one for each SDRAM bank) from the Video Controller. Each cycle, a Memory Controller services a request queue using heuristics intended to minimize wasted memory cycles.

A single Memory Controller owns all data associated with a pixel, so that it can process rendering and screen refresh requests independently of the other controllers. Neon stores the front/back/left/right buffers, Z, and stencil buffers for a pixel in a group of 64 bits or 128 bits, depending upon the number of buffers and the color depth. To improve 8-bit 2D rendering speeds and to decrease screen refresh overhead, a controller stores a pixel's overlay and display format bytes in a packed format on a different page.

5.2. Pixel Processors and Memory Controllers

Neon's design began with the Pixel Processors and Memory Controllers. We wanted to make effective use of a large peak bandwidth by maximizing the number of controllers, and by reducing read/write turnaround overhead, page crossing overhead, and pipeline stalls due to unbalanced loading of the controllers.

5.2.1. Memory Technology

We evaluated several memory technologies. We quickly rejected extended data out (EDO) DRAM and RAMBUS RDRAM due to inadequate performance (the pre-Intel RAMBUS protocol is inefficient for the short transfers we expected), EDO VRAM due to high cost, and synchronous graphic RAM (SGRAM) due to high cost and limited availability. This left synchronous DRAM (SDRAM) and 3DRAM.

3D-RAM [4], developed by Sun and Mitsubishi, turns read/modify/write operations into write-only operations by performing Z tests and color blending inside the memory chips. Deering [4] claims that this feature gives it a "3-4x performance advantage" over conventional DRAM technology at the same clock rate, and that its internal caches further increase performance to "several times faster" than conventional DRAM.

We disagree. A good SDRAM design is quite competitive with 3D-RAM's performance. Batching eight fragments reduces read latency and bus turnaround overhead to ½ cycle per fragment. While 3D-RAM requires color data when the Z test fails, obscured fragment writes never occur to SDRAM. In a scene with a depth complexity of three (each pixel is covered on average by three objects), about 7/18 of fragments fail the Z test. Factoring in batching and Z failures, we estimated 3D-RAM's rendering advantage to be a modest 30 to 35%. 3D-RAM's support for screen refresh via a serial read port gives it a total performance advantage of about 1.8-2x SDRAM. 3D-RAM's caches didn't seem superior to intelligently organizing SDRAM pages and prefetching pages into SDRAM's multiple banks; subsequent measurement of a 3D-RAM-based design confirmed this conclusion.

3D-RAM has several weaknesses when compared to SDRAM: incomplete support for OpenGL stencils and color blending, a slow 20 nsec read cycle time, inflexible addressing targeted at 1280 x 1024 screens, dedicated Z data pins that sit idle when not Z buffering, and high cost (currently 6 to 10 times more expensive per megabyte than SDRAM). Further, we'd need a different memory system for texture data. The performance advantage during Z buffering didn't outweigh these problems.

5.2.2. Fragment Batching and Overlaps

Processing fragments one at a time is inefficient, as each fragment incurs the full read latency and high impedance bus turnaround cycle overhead. Batch processing several fragments reduces this overhead to a reasonable level. Neon reads all Z values for a batch of fragments from the frame buffer, compares each to the corresponding fragment's Z value, then writes each visible fragment's Z and color values to the frame buffer.

Batching introduces a read/write consistency problem. If two fragments have the same pixel address, the second fragment must not use stale Z data. Either the first Z write must complete before the second Z read occurs, or the second Z "read" must use an internal bypass. Since it is rare for overlaps to occur closely in time, we considered it acceptable to stop reading pixel data until the first fragment's write completes.

We evaluated several schemes to create batches with no overlapping fragments, such as limiting a batch to a single object; all resulted in average batch lengths that were too short. We finally designed an eight-entry fully associative overlap detector per memory controller, which normally creates batches of eight fragments. The overlap detector terminates a batch and starts a new batch if an incoming fragment overlaps an existing fragment in the batch, or if the overlap detector is full. In both cases, it marks the first fragment in the new batch, and "forgets" about the old batch by clearing the associative memory. When a memory controller sees a marked fragment, it writes all data associated with the previous batch before reading new data for the marked fragment. Thus, the overlap detector need not keep track of all unretired fragments further down the pixel processing pipeline.

To reduce chip real estate for tags, we match against only the two bank bits and the column address bits of a physical address. If two fragments are in the same position on different pages in the same SDRAM bank, the detector falsely flags an overlap. This "mistake" can actually *increase* performance. In this case, it is usually faster to terminate the batch, and so turn the bus around twice to complete all work on the first page and then complete all work on the second page, than it is to bounce twice between two pages in the same bank (see Section 5.2.4 below).

5.2.3. Memory Controller Interleaving

Most graphics accelerators load balance memory controllers by finely interleaving them in one or two dimensions, favoring either rendering or screen refresh operations. An accelerator may cycle through all controllers across a scanline, so that screen refresh reads are load balanced. This one-dimensional interleaving pattern creates vertical strips of ownership, as shown in Figure 2. Each square represents a pixel on the screen; the number inside indicates which memory controller owns the pixel.

The SGI RealityEngine [1] has as many as 320 memory controllers. To improve load balancing during rendering, the RealityEngine horizontally and vertically tiles a 2D interleave

0	1	2	3	4	5	6	7	0	1
0	1	2	3	4	5	6	7	0	1
0	1	2	3	4	5	6	7	0	1

Figure 2: Typical 1D pixel interleaving

0	1	2	3	4	5	6	7	0	1
8	9	10	11	12	13	14	15	8	9
0	1	2	3	4	5	6	7	0	1

Figure 3: Typical 2D pixel interleaving

0	1	2	3	4	5	6	7	0	1
2	3	4	5	6	7	0	1	2	3
4	5	6	7	0	1	2	3	4	5
6	7	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	0	1

Figure 4: Neon's rotated pixel interleaving

pattern, as shown in Figure 3. Even a two-dimensional pattern may have problems load balancing the controllers. For example, if a scene has been tessellated into vertical triangle strips, and the 3D viewpoint maintains this orientation (as in an architectural walk-through), a subset of the controllers get overworked.

Neon load balances controllers for both rendering and screen refresh operations by rotating a one-dimensional interleaving pattern from one scanline to the next. Since half the controllers can handle the bandwidth requirements of a one-pixel wide vertical line, we interleave only half the controllers vertically, by rotating two pixels each scanline, as shown in Figure 4. This is also a nice pattern for texture maps, as any 2 x 2 block of texel resides in different memory controllers. (The SGI InfiniteReality [17] uses a rotated pattern like Neon within a single rasterizing board, but does not rotate the 2-pixel wide vertical strips owned by each of the four rasterizing boards, and so has the same load balancing problems as an 8-pixel wide non-rotated interleave.)

5.2.4. SDRAM Page Organization

Like other types of dynamic RAM, a page of SDRAM data must be loaded into a bank using a row activate command before reading from the page. Since this load is destructive, a bank must be written back using a precharge command before loading another page into the bank. The precharge and row activate take several cycles, so it is desirable to access as much data as possible within a page before moving to a new page.

Neon reduces the frequency of page crossings by allocating a rectangle of pixels to an SDRAM page. Object rendering favors square pages, while screen refresh favors wide pages. In order to keep screen refresh overhead low, Neon allocates screen pages with at worst an 8 x 1 aspect ratio, and at best a 2 x 1 aspect ratio, depending upon pixel size, number of color buffers, and SDRAM page size. Texture maps and off-screen buffers have no screen refresh requirements, and use pages that are at worst twice as wide as they are high. (Three-dimensional textures use pages that are as close to a cube of texels as possible.)

A 16 megabit SDRAM has two banks, called A and B. A 64 megabit SDRAM has four banks, called A, B, C, and D. These banks act as a two or four entry direct mapped page cache. It is possible to prefetch a page to a bank—that is, precharge one page and row activate a new page—in the midst of reading or writing data to another bank. Prefetching a page early enough hides the prefetch latency; in such cases it costs at most one overhead cycle to switch from accessing one page to accessing another page in a different bank. Note that a page belonging to the same bank as the current page cannot be prefetched; all accesses to the current page must complete before precharge and row activate can occur.

In the 32 megabyte configuration, each memory controller has two banks. To maximize the chance that a page crossing switches from one bank to another, Neon checkerboards pages between two banks, as shown in Figure 5. Any horizontal or vertical page crossings move from one bank to a different bank.

In the 64 and 128 megabyte configurations, each controller has four banks. Checkerboarding all four banks slightly improves performance at page corners, but not enough to warrant the complication of prefetching more than one bank ahead. Instead, these configurations assign the A and B banks to the bottom half of memory, and the C and D banks to the top half. Software preferentially allocates pixel buffers to the A and B banks, and texture maps to the C and D banks, to eliminate page thrashing between drawing buffer and texture map accesses.

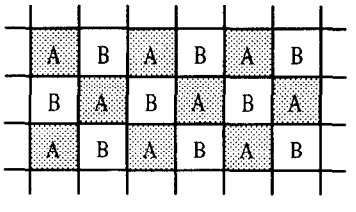

Figure 5: Page interleaving with two banks

5.2.5. Fragment Generation Chunking

To further increase locality of reference, the Fragment Stamp generates an object in rectangular "chunks." Normally, the chunk size matches the page size, so that the stamp generates all fragments of an object on one page before generating fragments for another page. Figure 6 shows the order in which a scanline-based algorithm generates fragments for a triangle that touches four pages. The shaded pixels belong to bank A. Note how only the four fragments numbered 0 through 3 access the first A page before fragment 4 accesses the B page, which means that the precharge and row activate overhead to open the first B page may not be completely hidden. Note also that fragment 24 is on the first B page, while fragment 25 is on the second B page. In this case the page transition cannot be hidden at all.

Figure 7 shows the order in which our chunking algorithm generates fragments. Note that generating all fragments on a page gives us the maximum possible time to prefetch the next page. Note further how the "serpentine" chunk ordering generates fragments first for the upper left page, then the upper right page, then the lower right page, and finally lower left page. This generally increases the number of page crossings that can use prefetching.

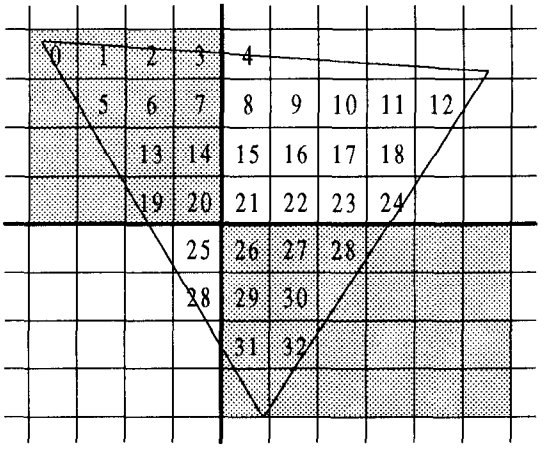

Figure 6: Scanline fragment generation order

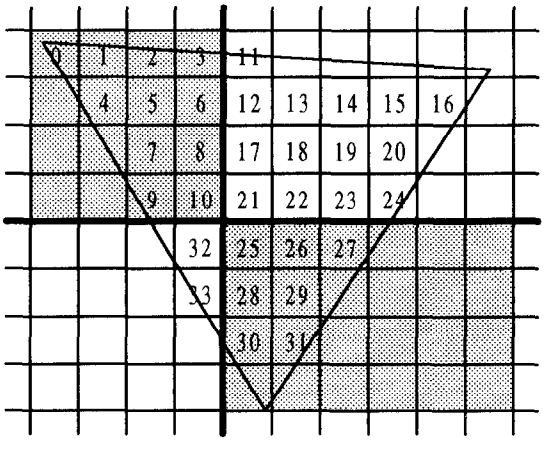

Figure 7: Neon's chunking fragment generation order

5.3. Texel Central

Texel Central is the kitchen sink of Neon. Since it is the only crossbar between memory controllers, it handles texturing and frame buffer copies. Since it has full connectivity to the Pixel Processors, it expands the internal 32 x 32 stipple pattern or an externally supplied stipple to foreground and background colors.

The subsections below describe a texture cache that reduces memory bandwidth requirements with fewer gates than a traditional cache, and a method of computing OpenGL's mip-mapping level of detail with greater accuracy than is typically achieved.

5.3.1. Texel Cache Overview

Texel Central has eight fully associative texel caches, one per memory controller. These are vital to texture mapping performance, since texel reads steal bandwidth from other memory transactions. Without caching, the 8 texel fetches per cycle for trilinear filtering require the entire peak bandwidth of memory. Fortunately, many texel fetches are redundant. When trilinear filtering, a fragment requires a 2 x 2 block of texels from two different texture maps. Adjacent fragments usually require overlapping 2 x 2 texel blocks; Hakura & Gupta [8] found that each texel is used by an average of four fragments. Each cache stores 32 bytes of data, so can hold 8 32-bit texels, 16 16-bit texels, or 32 8-bit texels. The total cache size is a mere 256 bytes, compared to the 16 to 128 *kilo*byte texel caches described in [8]. Our small cache size works well because chunking fragment generation improves the hit rate, the caches allow many more outstanding misses than cache lines, the small cache line size of 32 bits avoids fetching of unused data, and we never speculatively fetch cache lines that will not be used.

The texel cache also improves rendering of small X11 tiles. An 8 x 8 tile completely fits in the caches, so once the caches are loaded, Texel Central generates tiled fragments at the maximum fill rate of 3.2 gigabytes per second. The cache helps larger tiles, too, as long as one scanline of the tile fits into the cache.

5.3.2. Improving the Texel Cache Hit Rate

Scanline-oriented fragment generation creates locality of reference problems not just for SDRAM pages, but also for a multi-line texel cache. If the texel requirements of one scanline of a wide object exceed the capacity of the cache, texel overlaps across adjacent scanlines are not captured by the cache, and performance degrades to that of a single-line cache. Scanline generators can alleviate this problem, but not eliminate it. For example, fragment generation may proceed in a serpentine order, going left to right on one scanline, then right to left on the next. This always captures *some* overlap between texel fetches on different scanlines at the edges of a triangle, but also halves the width of triangles at which cache capacity miss problems appear.

Neon attacks the locality problem directly, by exploiting the chunking fragment generation described in Section 5.2.5 above. When texturing is enabled, Neon matches the chunk size to the capacity of the texel caches, rather than to the page size. This ensures that most fragments that are physically close in space are also generated closely in time. Redundant fetches still occur, but usually only for fragments along two edges of a chunk. Neon further reduces redundant fetches by making chunks very tall but only one pixel wide, so that usually only the top fragment of each chunk refetches texels that were already fetched for the chunk above it. If each texel is fetched on behalf of four fragments, chunking reduces redundant fetches in large triangles by about a factor of 8, and texel read bandwidth by about 35%, when compared to a scanline fragment generator.

5.3.3. Texel Cache Operation

A texel cache must not stall requests after a miss, and must track a large number of outstanding misses—since several other request queues are vying for the memory controller's attention, a miss might not be serviced for tens of cycles.

A typical CPU cache requires too much associative logic per outstanding miss. By noting that a texel cache should always return texels in the same order that they were requested, we eliminated most of the associative bookkeeping. Neon instead uses a queue between the address tags and the data portion of the texel cache to maintain hit/miss and cache line information. This approach appears to be similar to the texel cache described in [24].

Figure 8 shows a block diagram of the texel cache. If an incoming request address matches an Address Cache entry, the hardware appends an entry to the Probe Result Queue recording a hit at the cache line index of the matched address. If the request doesn't match a cached address, the hardware appends an entry to the Probe Result queue recording a miss, and that the new data will replace the data at the Least Recently Written Counter's (LRWC) value. It appends the requested address to the Address Queue, writes the address into the Address Cache line at the location specified by the LRWC, and increments the LRWC. The memory controller eventually services the entry in the Address Queue, reads the texel data from memory, and deposits the corresponding texel data at the tail of the Data Queue.

At the reply end of things, the hardware examines the head entry of the Probe Result Queue each cycle. A "hit" entry means that the requested data is available in the Data Cache at the location specified by the cache index. When the requested data is consumed, the head entry of the Probe Result Queue is removed.

If the head entry indicates a "miss" and the Data Queue is non-empty, the requested data is in the head entry of the Data

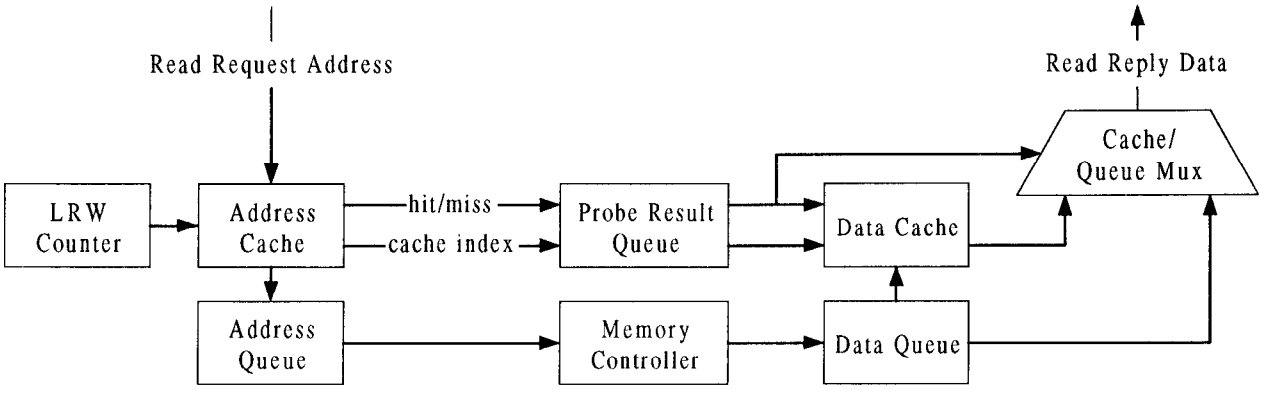

Figure 8: Texel cache block diagram

Queue. When the data is consumed, it is written into the Data Cache at the location specified by the cache index. The head entries of the Probe Result and Data Queues are then removed.

5.3.4. Accurate Level of Detail Computation

Neon implements a more accurate computation of the mip-mapping [23] level of detail (LOD) than most hardware. The LOD is used to bound, for a given fragment, the instantaneous ratio of movement in the texture map coordinate space *(u, v)* to movement in screen coordinate space *(x, y)*.

OpenGL's desired LOD computation first determines the distances moved in the texture map as a function of moving in the *x* or *y* direction on the screen, takes the maximum, and then takes the base 2 logarithm:

$$LOD = \log_2(\max(\mathrm{sqrt}((\partial u/\partial x)^2 + (\partial v/\partial x)^2), \mathrm{sqrt}((\partial u/\partial y)^2 + (\partial v/\partial y)^2)))$$

Software implementations usually do four multiplies, and convert the square root to a divide by 2 after the $\log_2$.

OpenGL allows implementations to compute the LOD using gross approximations to the desired computation. Hardware commonly takes the maximum of the partial derivative magnitudes:

$$LOD = \log_2(\max(\mathrm{abs}(\partial u/\partial x), \mathrm{abs}(\partial v/\partial x), \mathrm{abs}(\partial u/\partial y), \mathrm{abs}(\partial v/\partial y)))$$

This implementation can result in an LOD that is too low by half a mipmap level, which reintroduces the aliasing artifacts that mip-mapping was designed to avoid.

Neon uses a two-part linear function to directly approximate the desired distances. Without loss of generality, assume that $a > b$. The function:

if (b < a/2) return a + b/8 else return 7a/8 + b/4

is within ± 3% of $\mathrm{sqrt}(a^2 + b^2)$. This reduces the maximum error to about ±0.05 mipmap levels—a ten-fold increase in accuracy over typical implementations, for little extra hardware. The graph in Figure 9 shows three methods of computing the level of detail as a texture mapped square on the screen rotates from 0° through 45°. In this example, the texture map is being reduced by 50% in each direction, and so the desired LOD is 1.0. Note how closely Neon's implementation tracks the desired LOD, and how poorly the typical implementation does.

5.4. Fragment Generator

The Fragment Generator determines which fragments are within an object, generates them in an order that reduces memory bandwidth requirements, and interpolates the channel data provided at vertices.

The fragment generator uses half-plane edge functions

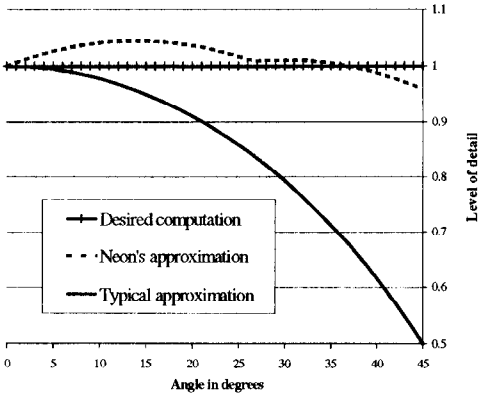

Figure 9: Various level of detail approximations

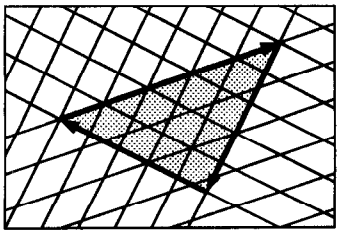

Figure 10: Triangle described by three edge functions

[6][18] to determine if a fragment is within an object. The three directed edges of a triangle, or the four edges of a line, are represented by planar (affine) functions that are negative to the left of the edge, positive to the right, and zero on the edge. A fragment is inside an object if it is to the right of all edges in a clockwise series, or to the left of all the edges in a counterclockwise series. (Fragments exactly on an edge of the object use special inclusion rules.) Figure 10 shows a triangle described by three clockwise edges, which are shown with bold arrows. The half-plane where each edge function is positive is shown by several thin "shadow" lines with the same slope as the edge. The shaded portion shows the area where all edge functions are positive.

For most 3D operations, a 2 x 2 fragment stamp evaluates the four edge equations for each of the four positions in the stamp. Texture mapped objects use a 1 x 1 stamp, and 2D objects use an 8 x 1 or 32 x 1 stamp. The stamp bristles with several probes that evaluate the four edge equations outside the stamp boundaries; the results are combined to determine in which direction the stamp should move next. Probes are cheap, as they only compute a sign bit. We use enough probes so that the stamp moves to locations where it does not generate any fragments only when it is following a very thin sliver triangle, or when it must move diagonally by moving first horizontally, then vertically. (Moving the stamp diagonally would have decreased performance due to an increased cycle time.) When the stamp is one pixel high or wide, several different probes may evaluate the edge functions at the same point. The stamp movement algorithm handles coincident probes without special code for the myriad stamp sizes. Stamp movement logic cannot be pipelined, so simplifications like this avoid making a critical path even slower.

The stamp may also be constrained to generate all fragments in a 2^n by 2^m rectangular "chunk" before moving to the next chunk, at the cost of three additional save states and associated multiplexors. Chunking is not cheap.[1] Each save state requires over 600 bits to record the four edge values and all 14 interpolated channels. But chunking improves the texture cache hit rate and decreases page crossings, especially non-prefetchable crossings. We found the cost well worth the benefits.

The fragment generator contains several capabilities specific to lines. The setup logic can adjust endpoints to render Microsoft Windows "cosmetic" lines. Lines can be dashed with a pattern that is internally generated for OpenGL lines and some X11 lines, or externally supplied by software for the general X11 dashed line case. We paint OpenGL wide dashed lines by sweeping the stamp horizontally across scanlines for y-major lines, and vertically across columns for x-major lines. Again, to avoid slowing the movement logic, we don't change the movement algorithm. Instead, the stamp always moves across what it thinks are scanlines, and we lie to it by exchanging the x and y coordinate information on the way in and out of the stamp movement logic.

Software can provide a scaling factor to the edge equations to paint the rectangular portion of X11 wide lines. (This led us to discover a bug in the X11 server's wide line code.) Software can provide a similar scaling factor for antialiased lines. Neon nicely

[1] It could be cheaper. We recently discovered that we could have implemented chunking with only two additional save states.

rounds the tips of antialiased lines and provides a programmable filter radius [16]. The OpenGL implementation exploits these features to paint antialiased square points up to six pixels in diameter that look like the desired circular points.

5.5. Command Parser

The Command Parser decodes packets, detects packet errors, converts incoming data to internal fixed-point formats, and decomposes complex objects into triangle fans for the fragment generator. Neon's command format is sufficiently compact that we found it unnecessary to use a high-speed proprietary bus between the CPU and the graphics device.

Neon's command set is designed for high bandwidth sequential streaming data modes. We don't initiate activity with out-of-order writes to registers or frame buffer locations, but use low-overhead, variable-length sequential commands. The processor can write commands directly to Neon, or can write to a ring buffer in main memory, which Neon accesses using DMA.

Neon supports multiple ring buffers at different levels of the memory hierarchy. Under normal conditions, the CPU uses a small ring buffer that fits in the on-chip cache, which allows the CPU to generate vertex information at the maximum possible speed. If Neon falls behind the CPU, which then fills the small ring buffer, the CPU can switch to a larger ring buffer. This larger buffer mostly resides further down in the memory system, so that the CPU then generates commands with somewhat less efficiency. Once Neon catches up, the CPU switches back to the smaller, more efficient ring buffer again.

5.5.1. Vertex Data Formats

Neon supports multiple representations for some data. For example, RGBA color and transparency can be supplied as four 32-bit floating point values, four packed 16-bit integers, or four packed 8 bit integers. The x and y coordinates can be supplied as two 32-bit floating point values, or as signed 12.4 fixed-point numbers. Using floating point, the six values (x, y, z, r, g, b) require 24 bytes per vertex. Using Neon's most compact representation, they require only 12 bytes per vertex. These translate into about 4 million and 8 million vertices/second on a 32-bit PCI.

If the CPU is the bottleneck, as with lit triangles, the CPU uses floating-point values and avoids clamping, conversion, and packing overhead. If the CPU can avoid lighting computations, and the PCI is the bottleneck, as with wireframe drawings, the CPU uses the packed formats. Future Alpha chips may saturate even a 64-bit PCI or an AGP-2 bus with floating point triangle vertex data, but may also be able to hide clamping and packing overhead using new instructions and more integer functional units. Packed formats on a 64-bit PCI allows transferring about 12 to 16 million (x, y, z, r, g, b) vertices per second.

5.5.2. Better Than Direct Rendering

Many vendors have implemented some form of *direct rendering*, in which applications get direct control of a graphics device in order to avoid the overhead of encoding, copying, and decoding an OpenGL command stream [11]. (X11 command streams are generally not directly rendered, as X11 semantics are harder to satisfy than OpenGL's.) We were unhappy with some of the consequences of direct rendering. To avoid locking and unlocking overhead, CPU context switches must save and restore both the architectural and internal implementation state of the graphics device on demand, including in the middle of a command. Direct rendering applications make new kernel calls to obtain information about the window hierarchy, or to accomplish tasks that should not or cannot be directly rendered. These synchronous kernel calls may in turn run the X11 server before returning. Applications that don't use direct rendering use more efficient asynchronous requests to the X11 server.

Neon uses a technique we called "Better Than Direct Rendering" (BTDR) to provide the benefits of direct rendering without these disadvantages. Like direct rendering, BTDR allows client applications to create hardware-specific rendering commands. Unlike direct rendering, BTDR leaves dispatching of these commands to the X11 server. In effect, the application creates a sequence of hardware rendering commands, then asks the X11 server to call them as a subroutine. To avoid copying client-generated commands, Neon supports a single-level call to a command stream stored anywhere in main memory. Since only the X11 server communicates directly with the accelerator, the accelerator state is never context switched preemptively, and we don't need state save/restore logic. Since hardware commands are dispatched in the correct sequence by the server, there is no need for new kernel calls. Since BTDR maintains atomicity and ordering of commands, we believe (without an existence proof) that BTDR could provide direct rendering benefits to X11, with much less work and overhead than Mark Kilgard's D11 proposal [12].

5.6. Video Controller

The Video Controller refreshes the display, but delegates much of the work to the memory controllers in order to increase opportunities for page prefetching. It periodically requests pixels from the memory controllers, "inverse dithers" this data to restore color fidelity lost in the frame buffer, then sends the results to an IBM RGB640 RAMDAC for color table lookup, gamma correction, and conversion to an analog video signal.

5.6.1. Opportunistic Refresh Request Servicing

Each screen refresh request to a memory controller asks for data from a pair of A and B bank pages. The memory controller can usually finish rendering in the current bank, ping-pong between banks to satisfy the refresh request and return to rendering in a different bank, and use prefetching on all page crossings.

Screen refresh reads cannot be postponed indefinitely in an attempt to increase prefetching. If a memory controller is too slow in satisfying the request, the Video Controller forces it to fetch the requested refresh data immediately. When the controller returns to the page it was rendering, it cannot prefetch it, as this page is in the same bank as the second screen refresh page.

The Video Controller delegates to each Memory Controller the interpretation of overlay and display format bytes, and the reading of pixels from the front, back, left, or right buffers. This allows the memory controller to immediately follow overlay and display format reads with color data reads, further increasing prefetching efficiency.

5.6.2. Inverse Dithering

Dithering is commonly used with 16 and 8-bit color pixels. Dithering decreases spatial resolution in order to increase color resolution. In theory, the human eye integrates pixels to approximate the original color. In practice, dithered images are at worst annoyingly patterned with small or recursive tessellation dither matrices, and at best slightly grainy with the large void and cluster dither matrices we have used in the past [22].

Hewlett-Packard introduced "Color Recovery™" [2] to perform this integration digitally and thus improve the quality of dithered images. Color Recovery applies a 16 pixel wide by 2 pixel high filter at each 8-bit pixel on the path out to the RAMDAC. In order to avoid blurring, the filter is not applied to pixels that are on the opposite side of an "edge," which is defined as a large change in color.

HP's implementation has two problems. Their dithering is non-mean preserving, and so creates an image that is too dark and too blue. And the 2 pixel high filter requires storage for the previous scanline's pixels, which takes a lot of real estate for Neon's worst case scanlines of 1920 16-bit pixels. The alternative—fetching pixels twice—requires too much bandwidth.

Neon implements an "inverse dithering" process somewhat similar to Color Recovery, but dynamically chooses between several higher quality filters, all of which are only one pixel high. We used both mathematical analysis of dithering functions and filters, as well as empirical measurements of images, to choose a dither matrix, the coefficients for each filter, and the selection criteria to determine which filter to apply to each pixel in an image. We use small asymmetrical filters near edges, and up to a 9-pixel wide filter for the interior of objects. Even when used on Neon's lowest color resolution pixels, which have 4 bits for each color channel, inverse dithering results are nearly indistinguishable from the original 8 bits per channel data.

5.7. Performance Counters

Modern CPUs include performance counters in order to increase the efficiency of the code that compilers generate, to provide measurements that allow programmers to tune their code, and to help the design of the next CPU.

Neon includes the same sort of capability with greater flexibility. Neon includes two 64-bit counters, each fully programmable as to how conditions should be combined before being counted. We can count multiple occurrences of some events per cycle (e.g., events related to the eight memory controllers or pixel processors). This allows us to directly measure, in a single run, statistics that are ratios or differences of different conditions.

6. PERFORMANCE

In this section, we present some incomplete performance results, based on cycle-accurate simulations of a 100 MHz part. (The publication deadline was the day after power-on.)

We achieved our goal of using memory efficiently. When painting 50-pixel triangles to a 1280 x 1024 screen refreshed at 76 Hz, screen refresh consumes about 25% of memory bandwidth, rendering consumes another 45%, and overhead cycles that do not transfer data (read latencies, high-impedance cycles, and page precharging and row addressing) consume the remaining 30%. When filling large areas, rendering consumes 60% of bandwidth.

As a worst-case acid test, we painted randomly placed triangles with screen refresh as described above. Each object requires at least one page fetch. Half of these page fetches cannot be prefetched at all, and there is often insufficient work to completely hide the prefetching in the other half. The results are shown in the "Random triangles" column of Table 1. (Texels are 32 bits.)

We also painted random strips of 10 objects; each list begins in a random location. This test more closely resembles the locality of rendering found in actual applications, though probably suffers more non-prefetchable page transitions than a well-written application. Triangle results are shown in the "Random strips" column of Table 1, line results are shown in Table 2.

The only fill rates we've measured are not Z-tested, in which case Neon achieves 240 million 64-bit fragments/second. However, the "Aligned strip" column in Table 1 shows triangle strips that were aligned to paint mostly on one page or a pair of pages, which should provide a lower bound on Z-tested fill rates. Note that 50-pixel triangles paint 140 million Z-buffered, shaded pixels/second, and 70 million trilinear textured, Z-buffered, shaded pixels/second. In the special case of bilinearly magnifying an image, such as scaling video frames, we believe Neon will run extremely close to the peak texture fill rate of 100 million textured pixels/second.

Triangle size	Random triangles	Random strips	Aligned strips	Peak generation
10-pixel	N/A	N/A	7.8	7.8
25-pixel	2.6	4.2	5.4	7.5
50-pixel	1.6	2.3	2.8	4.5
25-pixel, trilinear textured	N/A	2.0	2.3	4.0
50-pixel, trilinear textured	0.75	1.3	1.4	2.0

Table 1: Shaded, Z-buffered triangles, millions of triangles/second

Type of line	Random strips
10-pixel, constant color, no Z	11.0
10-pixel, shaded, no Z	10.6
10-pixel, shaded, Z-buffered	7.8
10-pixel, shaded, Z-buffered, antialiased	4.7

Table 2: Random line strips, millions of lines/second

The "Peak generation" column in Table 1 shows the maximum rate at which fragments can be delivered to the memory controllers. For 10-pixel triangles, the limiting factor is setup. The 2 x 2 stamp generates on average 1.9 fragments/cycle for 25-pixel triangles, and 2.3 fragments/cycle for 50-pixel triangles. For textured triangles, the stamp generates one fragment/cycle.

Neon's efficiency is impressive, especially when compared to other systems for which we have enough data to compute bandwidth numbers. For example, we estimate that the SGI Octane MXE, using RAMBUS RDRAM, has over twice the peak bandwidth of Neon—yet paints 50-pixel Z-buffered triangles about as fast as Neon. Even accounting for the MXE's 48-bit colors, Neon extracts about twice the performance per unit of bandwidth. The MXE uses special texture-mapping RAMs, and quotes a "texture fill rate" 38% higher than Neon's peak texture fill rate, while Neon uses SDRAM and steals texture mapping bandwidth from other rendering operations—yet their actual texture mapped performance is equivalent. Tuning of the memory controller heuristics might improve efficiency even further.

Good data on PC accelerators is hard to come by (many PC vendors tend to quote peak numbers without supporting details, others quote performance for small screens using 16-bit pixels and texels, etc.). Nonetheless, when compared to PC accelerators in the same price range, Neon has a clear performance advantage. It appears to be about twice as fast, in general, as Evans & Sutherland's REALimage technology (as embodied in the Mitsubishi 3DPro chip set), and the 3Dlabs GLINT chips.

7. CONCLUSIONS

Historically, fast workstation graphics accelerators have used multiple chips and multiple memory systems to deliver high levels of graphics performance. Low-end workstation and PC accelerators use single chips connected to a single memory system to reduce costs, but their performance consequently suffers.

The advent of 0.35 µm technology coupled with ball or column grid arrays means that a single ASIC can contain enough logic and connect to enough memory bandwidth to compete with multichip 3D graphics accelerators. Neon extracts competitive

performance from a limited memory bandwidth by using a greater percentage of peak memory bandwidth than competing chip sets, and by reducing bandwidth requirements wherever possible. Neon fits on one die, because we extensively share real estate among similar functions—which had the nice side effect of making performance tuning efforts more effective. Newer 0.25 µm technology would reduce the die size to about 160 mm^2 and increase performance by 20-30%. Emerging 0.18 µm technology would reduce the die to about 80 mm^2 and increase performance another 20-30%. This small die size, coupled with the availability of SDRAM at under $2/megabyte, would make a very low-cost, high-performance accelerator.

8. ACKNOWLEDGEMENTS

Hardware Design & Implementation: Bart Berkowitz, Shiufun Cheung, Jim Claffey, Ken Correll, Todd Dutton, Dan Eggleston, Chris Gianos, Tracey Gustafson, Tom Hart, Frank Hering, Andy Hoar, Giri Iyengar, Jim Knittel, Norm Jouppi, Joel McCormack, Bob McNamara, Laura Mendyke, Jay Nair, Larry Seiler, Manoo Vohra, Robert Ulichney, Larry Wasko, Jay Wilkinson.

Hardware Verification: Chris Brennan, John Eppling, Tyrone Hallums, Thom Harp, Peter Morrison, Julianne Romero, Ben Sum, George Valaitis, Rajesh Viswanathan, Michael Wright, John Zurawski.

CAD Tools: Paul Janson, Canh Le, Ben Marshall, Rajen Ramchandani.

Software: Monty Brandenberg, Martin Buckley, Dick Coulter, Ben Crocker, Peter Doyle, Al Gallotta, Ed Gregg, Teresa Hughey, Faith Lin, Mary Narbutavicius, Pete Nishimoto, Ron Perry, Mark Quinlan, Jim Rees, Shobana Sampath, Shuhua Shen, Martine Silbermann, Andy Vesper, Bing Xu, Mark Yeager.

Keith Farkas commented extensively on far too many drafts of this paper. A more detailed description of Neon is available in Reference [15].

References

[1] Kurt Akeley. RealityEngine Graphics. *Proceedings of SIGGRAPH 1993*, pp. 109-116.

[2] Anthony C. Barkans. Color Recovery: True-Color 8-Bit Interactive Graphics. *IEEE Computeer Graphics and Applications*, volume 17, number 1, pp. 193-198, January/February 1997.

[3] Anthony C. Barkans. High Quality Rendering Using the Talisman Architecture. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 79-88.

[4] Michael F. Deering, Stephen A. Schlapp, Michael G. Lavelle. FBRAM: A New Form of Memory Optimized for 3D Graphics. *Proceedings of SIGGRAPH 1994*, pp. 167-174.

[5] John H Edmondson, et. al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, volume 7, number 1, 1995

[6] Henry Fuchs, et. al. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *Proceedings of SIGGRAPH 1985*, pp. 111-120.

[7] Lynn Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, volume 10, issue 14, October 28, 1996.

[8] Siyad S. Hakura & Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th ISCA*, pp. 108-120, June 1997.

[9] *IBM CMOS 5S ASIC Products Databook*, IBM Microelectronics Division, 1995.

[10] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Presentation at *Microprocessor Forum*, October 22-23 1996, slides available at www.digital.com/info/semiconductor/a264up1/index.html.

[11] Mark J. Kilgard, David Blythe & Deanna Hohn. System Support for OpenGL Direct Rendering. *Proceedings of Graphics Interface 1995*.

[12] Mark J. Kilgard. D11: A High-Performance, Protocol-Optional, Transport-Optional Window System with X11 Compatibility and Semantics. *The X Resource*, issue 13, Proceedings of the 9th Annual X Technical Conference, 1995.

[13] Mark J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 45-55.

[14] Joel McCormack & Robert McNamara. *A Smart Frame Buffer*, Digital Equipment Corporation's Western Research Laboratory Research Report 93/1, January 1993, available at www.research.digital.com/wrl/techreports/pubslist.html.

[15] McCormack, Joel, Robert McNamara, Chris Gianos, Larry Seiler, Norman Jouppi & Ken Correll. *Neon: A (Big) (Fast) Single-chip Workstation Graphics Accelerator*, Digital Equipment Corporation's Western Research Laboratory Research Report 98/1, August 1998, available at www.research.digital.com/wrl/techreports.pubslist.html.

[16] Robert McNamara, Joel McCormack & Norman Jouppi. *Prefiltered Antialiased Lines Using Distance Functions*, Digital Equipment Corporation's Western Research Laboratory Research Report 98/2, August 1998, available at www.research.digital.com/wrl/techreports/pubslist.html.

[17] John S. Montrym, Daniel R. Baum, David L. Dignam & Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. *Proceedings of SIGGRAPH 1997*, pp. 293-302.

[18] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. *Proceedings of SIGGRAPH 1988*, pp. 17-20.

[19] Mark Segal & Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.1)*, 1995, available at http://www.sgi.com/Technology/OpenGL/spec.html.

[20] Robert W. Scheifler & James Gettys. *X Window System, Second Edition*, Digital Press, 1990.

[21] Jay Torborg & James Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. *Proceedings of SIGGRAPH 1996*, pp. 353-363.

[22] Robert Ulichney. The Void-and-Cluster Method for Dither Array Generation. *IS&T/SPIE Symposium on Electronic Imaging Science & Technology*, volume 1913, pp. 332-343, 1993.

[23] Lance Williams. Pyramidal Parametrics. *Proceedings of SIGGRAPH 1983*, pp 1-11.

[24] Stephanie Winner, Mike Kelley, Brent Pease, Bill Rivard & Alex Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer Algorithm. *Proceedings of SIGGRAPH 1997*, pp. 307-316.