Lightning-2: A High-Performance Display Subsystem for PC Clusters

Gordon Stoll* Matthew Eldridge† Dan Patterson* Art Webb* Steven Berman‡ Richard Levy‡
Chris Caywood* Milton Taveira* Stephen Hunt* Pat Hanrahan†

*Intel Corporation †Stanford University ‡Cornell University

Abstract

Clusters of PCs are increasingly popular as cost-effective platforms for supercomputer-class applications. Given recent performance improvements in graphics accelerators, clusters are similarly attractive for demanding graphics applications. We describe the design and implementation of Lightning-2, a display subsystem for such a cluster. The system scales in both the number of rendering nodes and the number of displays supported, and allows any pixel data generated from any node to be dynamically mapped to any location on any display. A number of image-compositing functions are supported, including color-keying and depth-compositing. A distinguishing feature of the system is its platform independence: it connects to graphics accelerators via an industry-standard digital video port and requires no modifications to accelerator hardware or device drivers. As a result, rendering clusters that utilize Lightning-2 can be upgraded across multiple generations of graphics accelerators with little effort. We demonstrate a renderer that achieves 106 Mtri/s on an 8-node cluster using Lightning-2 to perform sort-last depth compositing.

CR Categories: I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing; I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations; C.2.4 [Computer-communication Networks]: Distributed Systems—Client/Server;

Keywords: Graphics Hardware, Graphics Systems, Parallel Computing, Rendering Hardware, Rendering Systems

1 Introduction

Our goal in the design of Lightning-2 is to enable the display of high-performance 3D graphics from clusters of PCs. The prototype system we have developed is an interconnection fabric that connects the video outputs of multiple graphics accelerators to the video inputs of multiple display devices. Any pixel data rendered on any graphics accelerator can be routed in real-time to any location on any display. Three major requirements drove the design of the system: scalability, host system independence, and performance.

The need for scalability in both the number of video inputs and the number of video outputs was a direct result of our target application requirements. The groups involved in the initial design use PC clusters ranging from two machines under a desk to hundreds of machines in a national laboratory. The range of the display systems of interest is also large, from a single monitor to a display wall installation with dozens of projectors. These requirements obviated any design that could not scale along both of these axes. For this reason, the Lightning-2 system is built up from modules that can be tiled in two dimensions. Each individual module supports up to four video inputs and up to eight video outputs.

Our requirement for host system independence is a result of the dynamics of the PC and graphics accelerator markets. It would not be feasible to redesign and reimplement Lightning-2 on the design cycle of commodity graphics accelerators. Instead, it is designed independently from any specific accelerator and its compatibility requirements are kept to a minimum. As a result, clusters built using Lightning-2 can leverage the tremendous resources dedicated to the design and implementation of mass-market PCs and accelerators. In order to maintain system independence, Lightning-2 needs to capture data rendered by a modern PC graphics accelerator in a device-independent, standardized way. There are currently two choices: data can be read into system memory via the graphics I/O interface (AGP4x) or data can be captured as it is transmitted out of the display interface (now feasible due to the development of the industry-standard digital display interface, DVI). The first option has significant performance drawbacks because pixel data must be transmitted over the AGP and system memory buses, perhaps repeatedly. In contrast, the display interface has high and predictable performance and is optimized for minimal interference with ongoing rendering operations. Therefore, the Lightning-2 inputs connect directly to the DVI digital video outputs of commodity graphics accelerators.

The final requirement is performance. Even in its prototype form, the system is intended to serve as a research platform for the development of cluster-based interactive rendering libraries and applications. In order to be useful in this regard, the performance characteristics of the system must be good enough to support these applications. Lightning-2 supports workstation-class resolutions, refresh rates, update rates, and frame latencies, while making this level of performance straightforward for a programmer to attain.

2 Previous Work

Image reassembly and composition from multiple sources is a well-established area of parallel rendering architecture. One of the earliest such systems is the NASA II flight simulator, an image composition renderer based on multiple polygon “face” cards and a priority-based visibility scheme [10]. The Hewlett-Packard Visualize Ascend architecture [2] uses a custom network to flexibly composite the results of multiple graphics accelerators. Sony’s GScube, demonstrated at SIGGRAPH 2000, supports tiled and depth-based image composition of the outputs of multiple Playstation2 graphics

*{gordon.stoll,dan.w.patterson,art.d.webb}@intel.com
*{chris.caywood,milton.o.taveira,stephen.h.hunt}@intel.com
†{eldridge,hanrahan}@graphics.stanford.edu
‡{stb5,rdl6}@cornell.edu

ACM SIGGRAPH 2001, 12-17 August 2001, Los Angeles, CA, USA

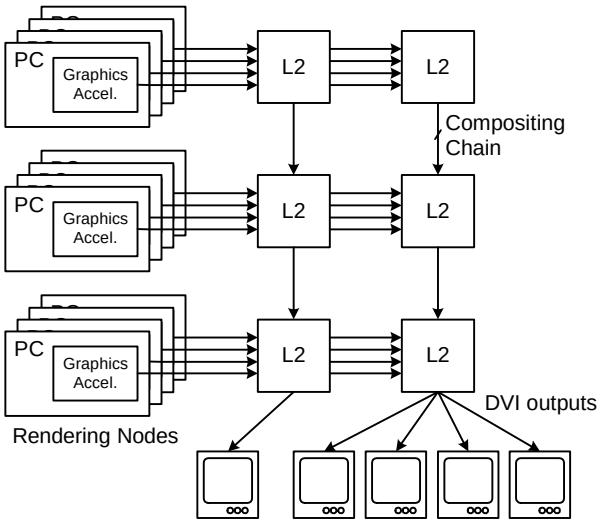


Figure 1: A 3×2 Lightning-2 matrix. A basic building block cascades in two dimensions to support widely varying PC clusters and display systems.

systems over a custom network.

One of the most ambitious image composition architectures is the PixelFlow [9] depth-compositing system. PixelFlow is an entire rendering system, with several custom chip designs. By contrast, Lightning-2 is a system for combining imagery from commodity rendering accelerators. The most similar element is the pipelined linear compositing chain used in both systems. However, in PixelFlow, the chain operates on a single tile of the output image at a time and the final output is reassembled in a framebuffer for display. The Lightning-2 compositing chain operates in raster order and at video rates; the results are sent directly out the video outputs with no further buffering.

Lightning [4], the direct precursor to this work, and the Sepia system [6] are more closely related to Lightning-2. Both systems serve as display subsystems for clusters. Unlike Lightning-2, neither system includes support for high-performance transfer of frames from graphics accelerators. Instead, it is necessary to use software-based rendering or to copy the framebuffer out of the accelerator into system memory and then back out into the compositing system. Either of these options limits the resolutions and frame rates that can be achieved. The MetaBuffer design [1], like Lightning-2, is intended to connect commodity graphics accelerators by capturing frames from the DVI interface. The MetaBuffer architecture includes novel multi-resolution capabilities that have not been explored in Lightning-2.

Molnar et al. define a taxonomy of graphics architectures based on where in the graphics pipeline they sort the input object parallelism to the output image parallelism [8]. Under their taxonomy, Lightning-2 can function as a part of a sort-first or sort-middle system, in which case it reassembles the tiled framebuffer for display, or it can function as a sort-last system, in which case it implements the depth-composition of the multiple framebuffers into a final output.

3 Architecture

A block diagram of an example Lightning-2 rendering cluster is shown in figure 1. The cluster consists of 12 PCs with graphics accelerators, 6 Lightning-2 modules arranged in a 3×2 matrix, and 5 displays, 1 driven by the first column of Lightning-2 modules and the other 4 displays driven by the second column. Digital video

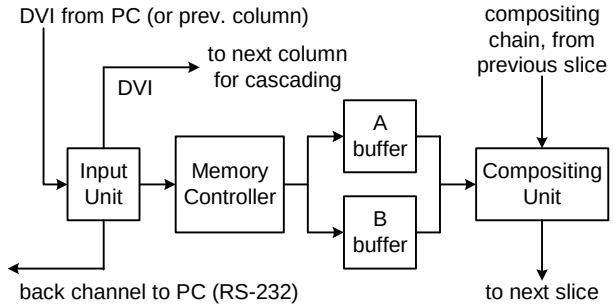


Figure 2: One "slice" of Lightning-2. A slice processes the output of a single rendering node.

from the graphics accelerators is captured by the Lightning-2 modules in the first column, which in turn repeat the video signal to the second column. Within each column, data from each of the PCs are combined via a linear compositing chain to produce the final output video. The chain is pipelined and constructed with point-to-point communication links, providing effectively unlimited scalability in this dimension.

A single Lightning-2 module has four DVI inputs, four DVI repeaters, and eight DVI outputs. In addition, there are communication channels back to the rendering nodes for use in the frame transfer protocol, described in section 3.3.2, and a control port for programming and configuration. In the prototype, the programming and configuration port is a JTAG hardware access port, and the back-channel to the PC is an RS-232 serial connection. The components necessary for processing the four DVI inputs are packaged together to minimize the per-board overhead of the compositing chain and video outputs; however, each input is essentially an independent slice of the board.

Figure 2 shows the components that comprise a single input slice. The input unit captures video from a single graphics accelerator via a digital interface, interprets control information embedded in the video, and writes pixel data to the framebuffer memory controller accordingly. The input unit repeats its DVI input to drive subsequent Lightning-2 columns. The compositing unit reads the slice's framebuffer and composites it with incoming framebuffer information from the previous slice, passing the composited result to the next slice. The framebuffer is double-buffered, allowing one frame to be stored while another is composited together with frames captured by other input inputs.

There are three important advantages to combining the multiple inputs via a pipeline of image compositing units and to cascading the input DVI signals across multiple Lightning-2 columns. First, this makes Lightning-2 a full crossbar router: each rendering node can composite its entire framebuffer contents into the output image and can do so with no layout restrictions. Second, it allows for powerful compositing operations beyond simple screen tiling, such as depth compositing. Third, this approach supports arbitrary scalability in the number of inputs and outputs while being relatively straightforward to design and implement. The disadvantage of this approach is that the cost of a full crossbar is proportional to the product of the number of inputs and the number of outputs.

The following sections describe the design and operation of the compositing chain, the memory system, and the input processing unit.

3.1 Compositing Chain

A pipeline of compositing units combines the pixel data from the rendering nodes to produce output video. A number of compositing operations are implemented in the prototype and are described in section 4. A single chain of Lightning-2 modules supports up to

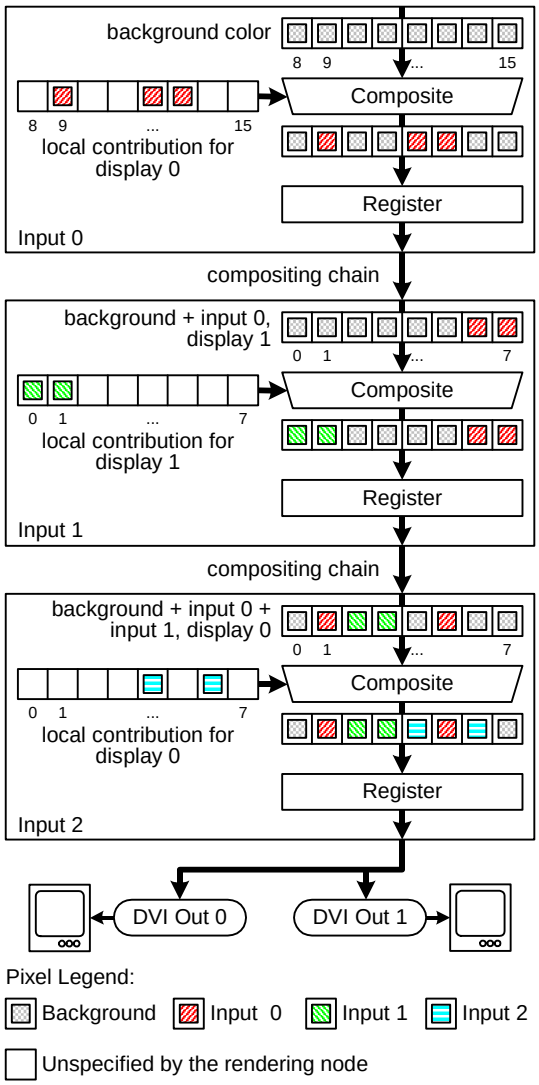


Figure 3: Snapshot of the pipelined compositing chain in operation. The background color pixels are injected at the top of the chain and are overwritten by data from the rendering nodes that are targeted at the same output pixel locations. The pixel data coming from a rendering node (the local contribution for a particular display) are composited via a programmable compositing operator with pixel data from rendering nodes earlier in the chain. Each compositing unit operates on 8 pixels at a time. The compositing chain is pipelined, so at any point in time many groups of pixels are in flight. Pixels destined for different displays are time-multiplexed on the chain. In this example, Lightning-2 is driving two displays and the compositing logic implements a simple priority scheme, in which valid pixels from the local host override the incoming pixels on the compositing chain.

533 Mpix/s of output bandwidth, enough to support three 1600×1200 workstation-class displays or eight 1024×768 projectors. For scalability beyond this point in the number of outputs, up to 15 chains can be cascaded together via the repeated DVI inputs of each node.

Pixel data for each output frame is shifted down the chain in raster order. At each compositing unit, the contributions of the associated rendering node are read from the Lightning-2 display buffer in raster order and composited into the output. Pixel data for different displays are time-multiplexed on the chain on a clock-by-clock basis. Each pixel is 32-bits and consists of a 24-bit RGB color and an 8-bit opcode that can be used to control compositing operations. The chain is eight pixels wide (256 bits) and runs at a maximum clock rate of 66MHz. The time-multiplexed pixel data are demultiplexed at the bottom of the chain to produce up to eight DVI output signals. Note that there is no buffering of the video data either within or at the bottom of the chain; the chain operates in output-space raster order and at the output video clock rate and thus can produce DVI output signals directly. An example of the chain in operation is shown in figure 3.

3.2 Memory System

The frames of pixel data rendered by the PCs arrive in input-space raster order, that is, the rendering node's framebuffer raster order. The compositing units operate on pixel data in output-space raster order. There are two basic approaches to reorganizing the pixel data between the input-space order and the output-space order. Pixel data can be reorganized before being stored in the framebuffer using a "forward" map (from input locations to output locations), or it can be reorganized as it is read out of the framebuffer using a "reverse" map (from output locations to input locations).

The demands on the framebuffer memory system differ significantly in these two mapping approaches. In a forward-mapping scheme, the input (draw) side accesses are irregular while the output (display) side accesses are perfectly regular. In a reverse-mapping scheme, the opposite is true. Given the characteristics of DRAM memories, the irregularity magnifies the input-side bandwidth demands in a forward-mapping scheme and the output-side demands in a reverse-mapping scheme. Additionally, the capacity required for the two approaches differs. The forward mapping approach requires storage per slice proportional to the size of the output space, while the reverse mapping approach requires storage proportional to the size of the input space. However, a reverse mapping framebuffer requires an additional data structure to allow the compositing unit to access pixel data in output space raster order. Such a structure, while buildable, complicates both the input unit and the compositing unit. Consequently, we have chosen a forward mapping (output space) framebuffer.

The pixel data must be double-buffered to allow for the simultaneous display of one frame of data while receiving the next frame of data. To address the high bandwidth requirements of simultaneous read and write access to the memory at video rates, we partition the memory into two physical banks corresponding to the two logical buffers. At its peak, DVI can drive the input with 165 Mpix/s. Under a forward-mapping scheme, the write access pattern of the input data to the framebuffer can consist of small, irregular writes that suffer from poor memory efficiency. In order to support the full DVI input bandwidth even in this case, we have constructed each bank of the framebuffer with 533 Mpix/s of memory bandwidth (2.1 GB/s). Because the output-space accesses are completely regular, and thus have excellent memory efficiency, the compositing unit has the full 533 Mpix/s of read bandwidth available. This bandwidth, as well as the total framebuffer memory, can be partitioned among up to 8 displays. For example, a single Lightning-2 chain can support 8 1024×768 displays or 3 1600×1200 displays.

3.3 Input Processing

There are two device-independent, standardized ways to capture data rendered by a modern PC graphics accelerator. Either data can be read into system memory via the graphics I/O interface, generally AGP4x, or data can be captured as it is transmitted out the display port. The first option, data readback, has significant performance drawbacks. Although the theoretical peak for such an operation over an AGP4x port is 88 Mpix/s, the best performance we have been able to achieve is 36.7 Mpix/s. Moreover, using readback

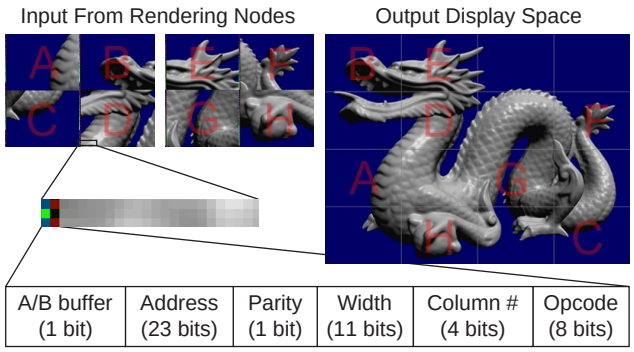


Figure 4: Mapping of pixel data from input space to output space using embedded strip headers. The images at the left are the contributions of two rendering nodes out of four to the output image on the right. The barely visible light lines in the final image indicate the tiling and are not present in the output. Two-pixel-wide strip headers are drawn into the framebuffers of the rendering nodes to provide Lightning-2 with mapping information. Each header controls the mapping of a one-pixel-high strip of arbitrary width. In addition to the width, the header specifies the target buffer (A/B), the target address in that buffer, the target column within the Lightning-2 matrix, an 8-bit "opcode" for controlling compositing operations, and a parity bit for detection of bit errors in the header itself.

directly impacts rendering performance, as it occupies the interface to the graphics accelerator.

Until recently, capturing pixel data from the display port was untenable due to the analog nature of standard display interfaces. This hurdle has been removed by the development of an industry-standard digital display port: the Digital Visual Interface, or DVI [5]. Currently available DVI interfaces support a pixel clock rate of 165 MHz, transferring active video data at up to 150 Mpix/s, depending on the video blanking intervals specified. A dual-channel version with twice this bandwidth is expected to be available in the near future. In contrast to the pixel readback path, the display refresh path in an accelerator has high and predictable performance and is optimized for minimal interference with ongoing rendering operations.

For these reasons, Lightning-2 captures display data via the DVI interface. Given this interface, we must now address two further issues. First, the mapping of pixel data from input space to output space must be specified. Second, the rendering nodes and Lightning-2 must be able to coordinate the transfer of frames.

3.3.1 Pixel Mapping

The forward mapping from input locations to output locations is specified by the user application or by the underlying rendering library. Mapping information is specified by drawing a 48-bit (two-pixel) wide "strip header" directly into two adjacent pixels in the framebuffer. These two pixels specify the mapping information for a run of pixels that immediately follow on the same scanline. The first two pixels on a scanline always contain a strip header. A width field in the header specifies the number of pixels being mapped by the header, and implicitly, the location of the next header on the scanline. Figure 4 shows this technique in use for assembly of screen tiles in an image-space subdivision rendering of a dragon.

This method allows the mapping of input locations to output locations to be arbitrarily fine-grained and completely dynamic. There is no excessive overhead for applications that do not require fine granularity, and synchronization between mapping data and pixel data is automatic as they are transferred on the same channel. This method also distributes the overhead of preparing and transmitting the mapping information across all of the rendering nodes. This is especially important for image-space subdivision, because the granularity of mapping decreases (and thus the overhead increases) quickly as the number of rendering nodes increases.

3.3.2 Frame Transfer Protocol

The rendering nodes transfer frames to Lightning-2 by displaying them on their DVI outputs. Each Lightning-2 input must capture a complete frame from its corresponding rendering node before the node swaps and displays the next frame. Until every input unit has received a complete frame, Lightning-2 cannot swap and start accepting the next frame from the rendering nodes. Thus, the earliest Lightning-2 will swap is one frame time after the last rendering node swaps. Furthermore, Lightning-2 is always receiving pixel data for the same frame from all the inputs. Consequently, Lightning-2's maximum update rate is limited by the time to transfer a frame to Lightning-2 plus the difference between when the first rendering node and the last rendering node swap.

This limit would not be a problem if all rendering nodes swapped simultaneously; however, it is not possible to synchronize the video refresh cycles of most commodity accelerators. As a result, there may be a delay of up to one refresh cycle between the swaps of different rendering nodes, even for a perfectly balanced renderer. Such a delay will limit Lightning's maximum update rate to half the refresh rate. We solve this problem by disabling the synchronization of buffer swapping on the accelerators to their vertical blanking intervals, instead allowing swapping to occur during horizontal blanking intervals.

Swapping during horizontal rather than vertical blanking is widely supported by commodity graphics accelerators. This allows buffer swapping to occur without significant delay, but with two repercussions. First, in order for this change to be useful, Lightning-2 must detect when the buffer swap occurs within the frame. This is done by embedding an extra bit in the strip headers that is toggled on alternately rendered frames. Second, Lightning-2 must be able to accept the scanlines of a frame out of order. The structure of the mapping information in Lightning-2 makes this relatively straightforward since each strip header only describes subsequent pixels on the same scanline – there are no inter-scanline decoding dependencies.

Lightning-2 is pipelined in its operation with the rendering nodes, so while it is accepting frame n from the rendering nodes, they may be rendering frame $n+1$. In order to insure that a Lightning-2 input has received all of frame n from its attached rendering node before the node swaps and displays frame $n+1$, we provide a per-input back-channel notification port that tells the attached rendering node when the frame it is currently displaying has been completely received. The node needs to wait for this notification only when it is ready to swap, not before, so in general, the transmission of the notification from Lightning-2 is overlapped with the application's rendering of the next frame.

The operation of a typical parallel rendering application running with Lightning-2 support is shown in figure 5. Each rendering node, shown on the left, first synchronizes with all of the other rendering nodes and determines viewing parameters for the next frame to be drawn. Each node then renders its portion of the scene to its local framebuffer and draws the strip headers that will route its pixels to the appropriate places on the output displays. Before swapping, each node must wait for notification from its corresponding Lightning-2 input unit that the previous frame has been completely received. Each rendering node then swaps and waits for the other rendering nodes to complete the frame.

Each Lightning-2 input unit follows a similar protocol. Starting at the rightmost state in the protocol diagram, each input unit first waits for the next frame to arrive from its attached rendering node, as recognized by the A/B bit in the strip headers. The input unit

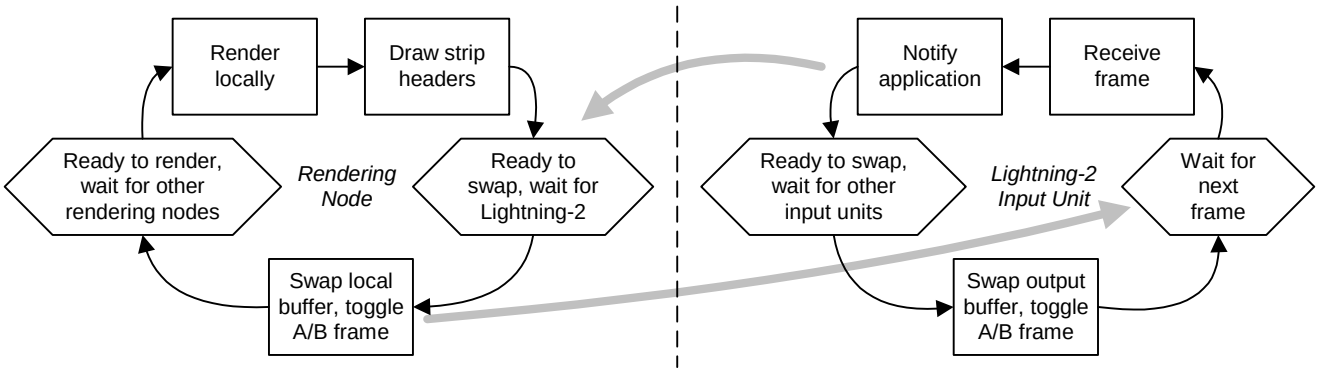


Figure 5: Operation of a typical application driving Lightning-2. The application communicates its progress around the state diagram to Lightning-2 by toggling the A/B bits of its strip headers on successive cycles. Similarly, Lightning-2 indicates the completion of an input cycle by signaling each rendering node over its back-channel. Both the application and Lightning-2 also require internal barriers to coordinate their internal parallelism.

then receives an entire frame of data, and at the end of the frame notifies the rendering node over the back-channel that its frame of data has been received. Each input unit then waits for each other input unit to have received a full frame of data, at which point they all swap simultaneously.[1] The input unit then resumes waiting for the next frame from its attached rendering node.

The Lightning-2 communication protocol has two performance implications. First, applications are restricted to run slightly slower than the local framebuffer refresh rate, because the back-channel notification is not sent until after the entire frame is received. To avoid impacting the final Lightning-2 output frame rate, the inputs may be run at a slightly higher refresh rate to provide adequate margin for the per-frame back-channel notification. Section 5.1 quantifies this overhead. Second, the use of Lightning-2 will always introduce an additional frame of latency in the display system, because an entire frame must be received from each rendering node (requiring a frame time at the application node refresh rate) before the inputs can swap. This latency can be reduced by increasing the application's local framebuffer refresh rate.[2]

4 Image Composition

Image composition encompasses a broad range of possible operations, from simple assembly of disjoint screen tiles to the combined transparency and depth operations described by Duff [3]. For the prototype implementation of Lightning-2, three basic operations were chosen as necessary for usability with interactive applications: simple screen-tile assembly, priority-based window layering, and color-keying. Additionally, a depth compositing function was added to allow experimentation with sort-last rendering algorithms.

4.1 Basic Operations

One of the most straightforward uses of Lightning-2 is to assemble tiled images that have been rendered on multiple nodes. In this case, each pixel in the output image is owned by a single rendering node, and the compositing function emits all pixels specified by its input slice and only passes through the pixels of the previous slices when its input has not specified a pixel for an output location.

Priority-based window layering allows overlapping windows drawn on different rendering nodes to correctly occlude each other, independent of the order of the nodes in the compositing chain. Lightning-2 implements priority-based window layering by encoding pixel data priority in the opcode field of the strip headers. This 8-bit opcode field is carried along with all of the RGB pixel data associated with that strip header.

Some applications require detailed, dynamic mapping geometry that can be difficult to express using the two-pixel strip headers. One example is the composition of dynamic GUI elements, such as cursors and menus, that are rendered on one node with content rendered on another node (or nodes). Using strip headers in this case would require extensive modification of the GUI library. For this reason, Lightning-2 supports color-keying, allowing pixels that match a programmable key color to be be ignored on a per input basis. In the GUI example, the content area of a window can be filled with a key color to create a "hole" into which content rendered on other nodes can be composited. Cursors, menus, and other GUI elements will properly occlude the remotely rendered content as long as they do not contain the key color.

4.2 Depth Compositing

Depth compositing merges color images into a single output image based on comparisons of corresponding depth images. Unlike the compositing operations previously described, depth compositing requires two values per pixel: color and depth.

The first difficulty in implementing any compositing operation that utilizes data other than simple RGB color is the potential difficulty in getting such data out of the DVI output port. For our experiments with depth-compositing, we currently use a read of the accelerator's depth buffer into system memory followed by a write from system memory back to the color buffer to achieve the required copy of 24 bits of depth information into the 24 bits of the RGB framebuffer. It would be preferable to perform a direct copy of the depth buffer to the color buffer, or even directly refresh the depth buffer out the DVI port; however, we have so far been unable to find such functionality in a mass-market accelerator.

The second difficulty is that the depth value for each pixel must be available at the compositing unit at the same time as the corresponding color value. We take advantage of Lightning-2's flexible pixel mapping to place the depth image on display 0, and the corresponding color image on display 1. Due to the time-multiplexing of the output display video on the compositing chain, the depth and color information for any given pixel will arrive at a given compositing unit in quick succession. The compositing unit compares the display 0 (depth) values and retains the result to choose between

[1] Our current Lightning-2 implementation waits for a vertical blanking interval before swapping, although this could be modified to swap during the horizontal blanking interval.

[2] There is additional latency introduced by the pipelining of the compositing chain. Each pipeline register in the compositing unit introduces 8 pixel clocks of latency per input slice. This latency is insignificant except in extremely large configurations.

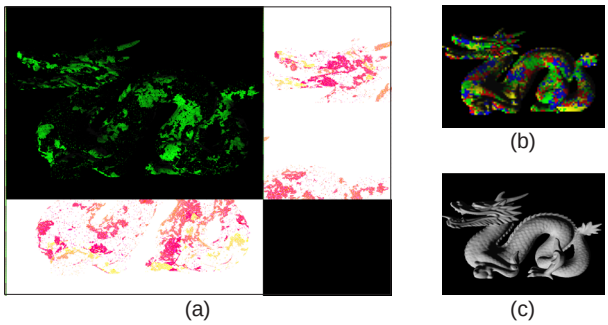


Figure 6: Depth compositing with four rendering nodes. Each node renders one quarter of the model into the upper-left region of its framebuffer. This leaves room around the edges for the depth information, which is read out of the depth buffer and written back into the color buffer in pieces to the right and bottom of the color information. Image (a) shows the contribution of one rendering node. Image (b) shows the final color image as reconstructed by Lightning-2, color-coded by rendering node. Image (c) shows the final seamless rendering.

the display 1 (color) values. Figure 6 shows images from the operation of this technique with four rendering nodes.

5 Results

We have constructed four prototype Lightning-2 boards. A completely populated board, shown in figure 7, consists of 23 Xilinx FPGAs, 512MB of SDRAM, and a small number of support chips. We expect that the architecture could be implemented relatively inexpensively, with the cost probably dominated by memory. The 512MB of memory consists of 128MB of memory per input slice, which is comparable in size and cost to the memory already included in the graphics accelerator.

The cluster used in our experiments consists of nine 1.5GHz Pentium-4 processor based PCs. Eight PCs are connected to a Lightning-2 matrix. The ninth is used as a "master" node by applications that are organized in a master/slave fashion. A Myrinet network using PCI-64/66 network adapters connects all nine PCs, allowing the master to synchronize the operation of the slaves and distribute control information to the applications. Each workstation is equipped with 256 MB of RDRAM and an NVIDIA GeForce2 Ultra / Quadro2 Pro AGP4x graphics accelerator.

5.1 Frame Transfer Protocol Performance

The frame transfer protocol of Lightning-2 involves a number of communication steps that might impose overhead that could reduce rendering performance, reduce the frame update rate, or increase the frame update latency. Our goal for the protocol is to achieve a consistent 60Hz update to a 60Hz output display. As the transfer of a frame from the rendering node into Lightning-2 necessarily includes a complete display refresh cycle on the rendering node, the refresh cycle must be set faster than 60Hz in order to allow time for the rest of the operations involved in the frame transfer.

There are two possible sources of overhead in the frame transfer protocol that are on the critical path for achieving a 60Hz update rate. These operations cannot be overlapped within the required rendering node refresh cycle time and must instead be completed within the small gap between that time and the 16.67ms output display refresh time at 60Hz. The first is the communication from Lightning-2 to the PC via the RS-232 back-channel when a frame transfer has completed. The second is the internal synchronization that Lightning-2 must perform before performing an internal buffer swap. Due to the relative speeds of the RS-232 back-channel and Lightning-2's internal communication paths, the back-channel communication path dominates for any but extremely large Lightning-2 configurations. The raw communication path from the PC over DVI to the Lightning-2 input unit and back to the PC over RS-232 was measured in isolation during the design phase of Lightning-2 and found to be under 0.1ms. The worst-case time that Lightning-2 must wait for a horizontal blanking interval is on the order of 0.16ms. Given these measurements, even a slight increase in the rendering node refresh rate should allow for the necessary operations. We are currently unable to set our rendering node graphics accelerators to arbitrary DVI refresh rates, so we chose 70Hz, the first available refresh rate faster than 60Hz. This provides a gap of approximately 2.3ms in which to perform the necessary operations. Note that the time available to the application for rendering and other application tasks remains approximately 16.67ms; application processing is fully overlapped with the frame transfer operation so that the 2.3ms is not lost. Operating the graphics accelerators at this higher refresh rate does impose a small extra bandwidth burden on the memory system of the graphics accelerator.

Using the above configuration, we measured a minimal Lightning-2 application that performs no rendering work, but only synchronizes with Lightning-2 and swaps its local framebuffer repeatedly. The application meets 60Hz timing greater than 90% of

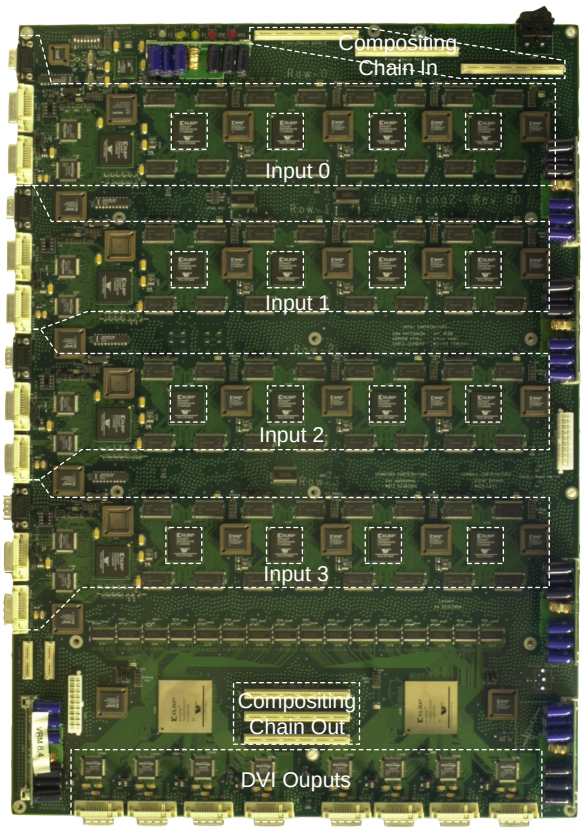
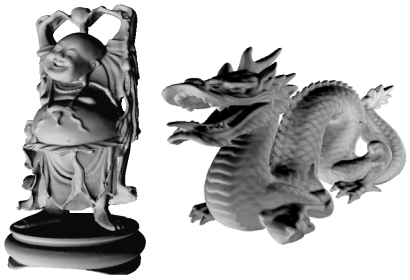


Figure 7: Lightning-2 module. The 4 inputs are the pairs of white connectors down the left-hand edge of the board. The upper connector of each pair is the DVI input and the lower connector is the repeated output. Visible above each pair is the serial port for the back-channel. Each input slice stretches out horizontally from its associated connectors and consists of an input FPGA and 4 FPGAs that map the input's framebuffer and implement the compositing logic. The 8 DVI outputs are distributed across the bottom of the board. The 3 wide white connectors staggered at the top of the board and in a column at the bottom of the board carry the compositing chain.



	Buddha	Dragon
Vertices	543,652	4,376,450
Triangles	1,155,523	9,263,100
Strips	254,955	2,033,100
Average Strip Length	6.53	6.56
Serial Frames/s	13.0	1.69
Serial Mtri/s	15.2	15.7

Figure 8: The Buddha and Dragon models.

the time. This is the same result observed for a standalone renderer that does not use Lightning-2. We have verified that the majority of the missed frames correspond to interference by the operating system and other background processes. The results are identical for synchronized operation across our cluster of 8 rendering nodes.

5.2 Depth Compositing Performance

We have implemented an interactive model viewer that utilizes the depth compositing technique described in section 4.2. This renderer is intended for the demonstration and measurement of Lightning-2's capabilities. Issues in distributing and maintaining scene data and in managing distributed texture data are not addressed. The WireGL image-subdivision renderer described in section 5.3 is more fully developed and addresses more of these issues.

Our two test models, Buddha and Dragon, are shown in figure 8. Both models are from the Stanford 3D Scanning Repository. The models consist of 1.16 million and 9.26 million stripped triangles respectively, and both include per-vertex normals. In order to experiment with a large model that could utilize the full rendering power of our 8-node cluster while still comparing our performance to that of the uniprocessor case, Dragon uses instancing to draw the same model 10 times. This increases our geometry work 10-fold while leaving our memory footprint unchanged so that we can still measure the renderer's serial performance. In all of our experiments, the strips are assigned in round-robin order to the rendering nodes at startup time and each node builds a display list for its subset of the model. The color and depth information is organized as shown in figure 6. For these experiments, the rendering node framebuffer is 1280×1024 and the output image size is 800×600.

The speedup of our parallel renderer on Dragon is shown in figure 9. Dragon has a 6.8x speedup at 8 rendering nodes, achieving a framerate of 11.4Hz and a triangle rate of 106 Mtri/s. The speedup is computed relative to the serial rendering time. In each case, including the one rendering node case, the parallel renderer incurs the overhead of master-slave communication, the frame transfer protocol, and the copy of the depth buffer to the color buffer.

We investigate the overhead of this copy operation by rendering Buddha, which has significantly less geometry than Dragon and will more readily expose the Lightning-2 overhead. Figure 10a shows the proportion of time spent copying the depth buffer, rendering, and synchronizing with Lightning-2 as a function of the number of rendering nodes for Buddha. The execution time shows continual improvements to 8 rendering nodes, although the large fixed cost of the depth copy limits the achieved performance. For the 800×600 window used in our experiments, this copy operation takes approximately 17ms. To examine the renderer's performance without this overhead, we measured Buddha with the depth copy disabled. The output images are incorrect in this case, but this does not affect the performance of Lightning-2. The execution time without the depth copy, shown in figure 10b, shows performance improvements up to 5 rendering nodes. At this point, Buddha is rendering at 60Hz and further decreases in the rendering time will have no impact since the rendering nodes cannot swap more frequently than the Lightning-2 output rate. This shows up in the graph as an increase in time spent waiting for back-channel notification from Lightning-2.

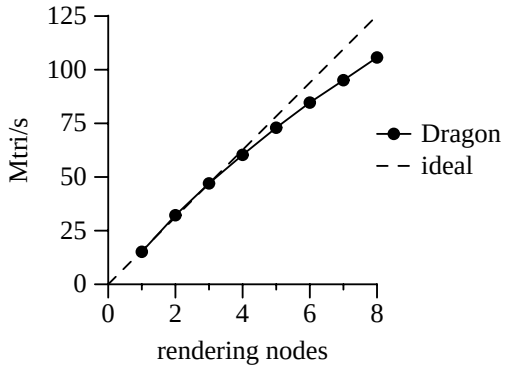


Figure 9: Speedup for Dragon, as a function of the number of rendering nodes. The speedup is relative to the serial execution time in figure 8.

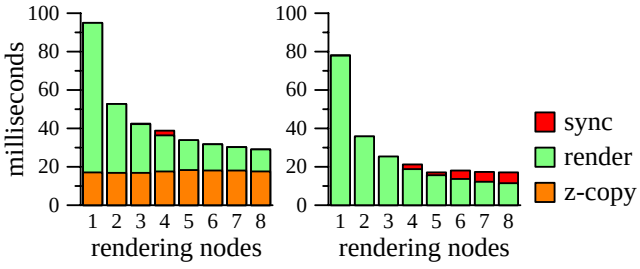


Figure 10: Frame time for Buddha, as a function of the number of rendering nodes. The frame time is partitioned into "z-copy," the amount of time spent copying the depth buffer to the color buffer, "render," the amount of time spent in the application code, and "sync," the amount of time spent waiting for back-channel notification from Lightning-2.

5.3 Image-Space Subdivision Performance

The experiments described here have focused on using Lightning-2 for sort-last rendering using depth-compositing. Humphreys et al. describe a sort-first tiled rendering system called "WireGL" [7] that uses Lightning-2 to assemble the output image. WireGL partitions the work of rendering a frame in screen space by dividing the screen into a number of tiles and assigning the tiles to rendering nodes. Lightning-2 allows WireGL to perform this screen-space subdivision of work and then, at no cost, reassemble the outputs of the rendering nodes into a single image. Using Lightning-2 in this fashion, WireGL can operate at over 70 million vertices per second with 16 client nodes submitting commands and 16 server nodes rendering them.

WireGL can support some functionality that would be difficult to support with a depth-compositing renderer, such as alpha-blended transparency and other operations which depend on primitive sub-

mission order. However, depth compositing has the significant advantage that each node of a parallel application can interface directly with a graphics accelerator rather than performing the comparatively expensive operation of transmitting commands over a network to the appropriate rendering node. The WireGL authors cite rendering rates of over 21 million vertices per second on the local host as compared to only 7.5 million over the network to a remote rendering node.

Screen-partitioning approaches can support very large output display spaces. We have used the WireGL/Lightning-2 system to render 3D imagery to multiple workstation displays, large multi-projector display walls, and next-generation LCD panels such as the IBM 9.2 megapixel "Bertha" prototype [12]. As described previously, this type of application is difficult when using depth-compositing since each application framebuffer must match the resolution of the output display.

6 Discussion

Lightning-2 is designed as a digital video crossbar: it allows any rendering node to place any pixel at any location on any output display. A crossbar design is resource-intensive since its cost is proportional to the product of the number of inputs and the number of outputs. This is most noticeable in the memory system because the memory capacity and bandwidth are increasingly sparsely utilized as a Lightning-2 system is scaled up. The advantage of the crossbar design is that it provides high and predictable performance across all possible usage models, an important property for a research platform. When the usage models and performance tradeoffs are better understood in this design space, a more efficient design could be possible.

Lightning-2 receives input from the rendering nodes via the digital display interface, DVI. Using DVI has the advantage of high and predictable bandwidth, but the disadvantage of limiting the communication to the color raster information DVI is designed to transport. Because Lightning-2 requires more general communication with the rendering nodes, we chose to create our own packet-based protocol by embedding two-pixel headers within the color information, rather than introducing an additional communication channel. However, this embedding can complicate application and graphics library software. In order to support depth-compositing, we also transport non-RGB data in the form of depth values. Unfortunately, current graphics hardware does not allow for direct copies from the depth buffer into the color buffer. This impacts the performance of depth-compositing on Lightning-2 severely, as described in section 5.2. Fortunately, digital display interfaces are beginning to move towards more general functionality. Packet-based display interfaces have been implemented in systems such as PV Link [11] and are being discussed for DVI 2.0. These interfaces include support for selective (non-raster) refresh and more general datatypes. It is likely that future designs in this space could leverage this greater generality.

We have not implemented anti-aliasing support in the current Lightning-2 system. In an image-space subdivision renderer such as WireGL, anti-aliasing could be implemented on the current Lightning-2 hardware and could leverage anti-aliasing support in the graphics accelerators. Anti-aliasing in a depth-compositing renderer is more difficult. Supersampling, the approach used in PixelFlow, is the most straightforward approach, but greatly increases the memory system and compositing chain bandwidth requirements. This approach could be implemented in Lightning-2 with minor changes to the FPGA programming; the bandwidth of the memory system and compositing chain are sufficient to support a single 1024×768 depth-composited output display with 4x supersampling.

7 Conclusion

We have described Lightning-2, a hardware image composition architecture for clusters of graphics-accelerated PCs. The key aspects of the design are its scalability, its independence from any specific graphics accelerator, and its high performance in terms of display resolution, refresh rate, update rate, and update latency. Scalability makes it possible to achieve much higher performance than is possible with a single PC, and device independence makes it possible to maintain this advantage over multiple generations of graphics accelerators. The prototype implementation has been demonstrated to achieve 60Hz update rates to 60Hz output displays from our cluster of eight PCs. We have demonstrated performance with 8 rendering nodes of up to 106 Mtri/s. The system is flexible enough to support both this sort-last rendering approach and WireGL, a sort-first parallel implementation of OpenGL.

Acknowledgements

Kai Li made significant contributions to the early stages of the design and advocated including support for multiple displays. John Owens helped design and build Lightning, Lightning-2's predecessor. Ian Buck integrated Lightning-2 support into WireGL. The Dragon and Buddha models are from the Stanford 3D Scanning Repository. Diane Tang provided helpful feedback on the organization and writing. The reviewers made numerous helpful comments that improved this work. This work was supported at Stanford by the Fannie and John Hertz Foundation, Intel Corporation, and the DARPA DIS program (contract DABT63-95-C-0085-P00006).

References

[1] William Blank, Chandrajit Bajaj, Donald Fussel, and Xiaoyu Zhang. The MetaBuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System Using Commodity Rendering Engines. Technical Report TR2000-16, Department of Computer Science, University of Texas at Austin, 2000.

[2] Ross Cunniff. Visualize fx Graphics Scalable Architecture. In *Proceedings of Eurographics Hardware/SIGGRAPH Hot3D*, pages 29–38, August 2000.

[3] Tom Duff. Compositing 3-D Rendered Images. *Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 41–44, July 1985.

[4] Matthew Eldridge and John D. Owens. Lightning: A Scalable, Distributed, Virtual Framebuffer. Technical Report (Unpublished), Department of Electrical Engineering, Stanford University, 1998.

[5] Digital Display Working Group. Digital Visual Interface 1.0 Specification, 1999. `http://www.ddwg.org`.

[6] Alan Heirich and Laurent Moll. Scalable Distributed Visualization Using Off-the-Shelf Components. *Symposium on Parallel Visualization and Graphics*, pages 55–60, October 1999.

[7] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A Scalable Graphics System for Clusters. *Computer Graphics (Proceedings of SIGGRAPH 01)*, August 2001.

[8] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, pages 23–32, July 1994.

[9] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed Rendering Using Image Composition. *Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 231–240, July 1992.

[10] Steven Molnar and Henry Fuchs. *Advanced Raster Graphics Architecture*, chapter 18, pages 899–900. Addison–Wesley, second edition, 1990.

[11] K. R. Schleupen. Driving and Interface Technology for High Resolution AM-LCDs. *Seventh International Display Workshops (Proceedings of IDW 00)*, November 2000.

[12] T. Ueki. Requirements for Large Size and High Resolution TFT-LCDs. *Proceedings of the International Display Manufacturing Conference*, 2000.