Polygon Rendering on a Stream Architecture

John D. Owens William J. Dally Ujval J. Kapasi Scott Rixner Peter Mattson Ben Mowery

Computer Systems Laboratory*
Stanford University

Abstract

The use of a programmable stream architecture in polygon rendering provides a powerful mechanism to address the high performance needs of today's complex scenes as well as the need for flexibility and programmability in the polygon rendering pipeline. We describe how a polygon rendering pipeline maps into data streams and kernels that operate on streams, and how this mapping is used to implement the polygon rendering pipeline on Imagine, a programmable stream processor. We compare our results on a cycle-accurate simulation of Imagine to representative hardware and software renderers.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—Single-instruction-stream, multiple-data-stream processors (SIMD)

Keywords: graphics hardware, stream processing, stream architecture, kernels, streams, polygon rendering, OpenGL, media processors, SIMD

1 Introduction

Modern graphics processors must be able to render tens of millions of triangles per second and rasterize hundreds of millions of pixels per second to render complex scenes at real-time rates. Recently, commodity graphics processors have rendered increasingly more complex scenes at these rates, but they do so with special-purpose hardware that implements a fixed pipeline. Deviation from the specific functionality implemented either incurs a significant performance penalty or is impossible.

Alvy Ray Smith's observation that "Reality begins at eighty million polygons" per frame [1, 32] implies that more than two orders of magnitude of improvements to today's fastest commodity hardware are still required to produce real-world complex scenes at real-time rates. However, as we approach that goal, future graphics hardware must not only increase its geometry and fill rates but also its flexibility.

The demand for more detailed lighting and shading is one of the driving forces behind the desire for more flexibility in the pipeline. Currently, many applications incur the large cost of sophisticated multipass methods to get the effects they need. Unfortunately, multipass methods are still not sufficient for many desirable effects such as an environment mapped onto a bump-mapped object [3, 11]. To enable further improvements, there are nearly 200 proposed extensions to the latest version of OpenGL. There is no question that developers are demanding more functionality, and that hardware manufacturers are attempting to deliver it.

The most flexible, powerful rendering interfaces, such as Pixar's Renderman [34], are currently implemented as software running on general-purpose microprocessors with little, if any, graphics hardware support. While these systems produce high-quality rendered images, they cannot achieve real-time performance on today's general-purpose processors.

High-performance, programmable polygon rendering hardware is needed to bridge the gap between high-performance special-purpose systems and flexible general-purpose systems. We present our implementation of polygon rendering on the programmable Imagine stream processor [26], which is designed to meet this goal. By requiring the application to be cast as a set of computational kernels that operate on homogeneous data streams, Imagine achieves high computational density and efficiency while maintaining the flexibility of a programmable processor.

The remainder of this paper describes and evaluates our implementation of polygon rendering on the Imagine stream architecture. In Section 2 we describe the stream programming model and Imagine, a stream processor that implements this model, and enumerate the benefits of this programming model and machine organization. Section 3 presents our implementation of the polygon rendering pipeline. In Section 4 we describe our experimental setup and in Section 5 we use this setup to compare the performance of our system to representative hardware and software implementations of the OpenGL pipeline to analyze the performance of polygon rendering on our stream architecture.

2 Stream Processing

The characteristics of polygon rendering workloads and the trends in modern VLSI technology together motivate the use of the stream programming model and a stream architecture. Three characteristics of polygon rendering make it poorly suited to general-purpose processors but ideally suited to a stream architecture. First, polygon rendering is an inherently parallel application. Operations on pixels exhibit parallelism, and, subject to ordering constraints, triangles may also be processed in parallel. Except for limited use of SIMD instruction-set extensions, modern microprocessors are unable to fully exploit this parallelism in the same way as stream processors.

Next, the memory traffic generated by polygon renderers has little temporal locality and hence does not take advantage of the caching capabilities of the memory architectures of modern processors. Input primitives have little to no immediate reuse, as they are rendered then discarded. The large size of input datasets (on the order of megabytes) precludes their storage in most caches, and except for explicitly defined display lists, special-purpose processors rarely cache any primitive data at all. (Texture accesses are

*Gates Computer Science Building 4A, Stanford, CA 94305 USA; {jowens, billd, ujk, rixner, pmattson, bmowery}@cva.stanford.edu

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

HWWS 2000, Interlaken, Switzerland
© ACM 2000 1-58113-257-3/00/08 ...$5.00

an exception to this behavior and do benefit from caching, as described by Hakura and Gupta [12].) While renderer memory traffic has little conventional temporal locality, it does have considerable *producer-consumer* locality, which is exploited by the register and memory hierarchy of a stream processor.

Finally, in polygon rendering, the total bandwidth through the pipeline is more important than the latency of any one item within it. Modern special-purpose hardware heavily pipelines the operations on its datasets, and because these datasets are so large, the time for the dataset is much greater than the latency for any primitive. Thus, the time to finish the entire dataset is dependent not on the time for any one primitive but instead on the rate at which primitives can be sent into and through the pipeline. A stream processor is better able to exploit this latency tolerance to increase throughput than a general-purpose processor which is optimized to minimize latency at the expense of throughput.

Meanwhile, modern VLSI technology has enabled designers to place tens to hundreds of ALUs on a chip. Special-purpose machines have been successful at computationally intense tasks such as polygon rendering in large part because they have been able to harness this computing power. Because the bottleneck in modern VLSI systems is shifting from computation to communication, the challenge to designers of modern programmable architectures is to efficiently structure data movement into, out of, and through the chip.

A *stream architecture* is designed to address the needs of graphics and other multimedia applications by exploiting the trends of modern VLSI. This architecture offers the computational horsepower necessary to meet the needs of demanding applications such as polygon rendering. It allows its users to leverage the inherent parallelism in these applications, to take advantage of their producer-consumer data locality, and to exploit their latency tolerance to increase throughput.

2.1 The Stream Programming Model

In the stream programming model, a *stream* is a set of data elements of a single arbitrary datatype. Streams are consumed by popping complex data elements from the front of a stream; streams are produced by appending data elements to the back of a stream. Streams can be constructed from other streams through append, truncate, or extract operations. A stream's elements can also index into another stream.

Programs are structured into *kernels* that operate on streams. Kernels take one or more streams as inputs and produce one or more streams as outputs. Typically, kernels loop over an input stream, performing identical operations on each input element to produce their outputs. Kernels can be chained together, where the output stream from one kernel is fed into the next kernel as an input stream. Producer-consumer locality is exploited by consuming the result of one kernel as soon as it is produced. As an example, Figure 1 shows how the transform and shade stages of a polygon renderer map to streams and kernels.

Stream computation is most efficient when performed on homogeneous data elements. Repeated homogeneous operations on streams permits simple control flow in the computational kernels, allowing a stream architecture to forego the complicated control hardware present in general-purpose processors.

Because of the importance of processing streams of homogeneous elements, a stream programming system provides an efficient mechanism to sort a stream of heterogeneous elements into one or more streams of homogeneous elements. This is accomplished by either removing elements from the input stream or by splitting the input stream into several homogeneous output streams. Each output stream can then be processed by a simple kernel that needs no conditional operations to distinguish between heterogeneous elements.

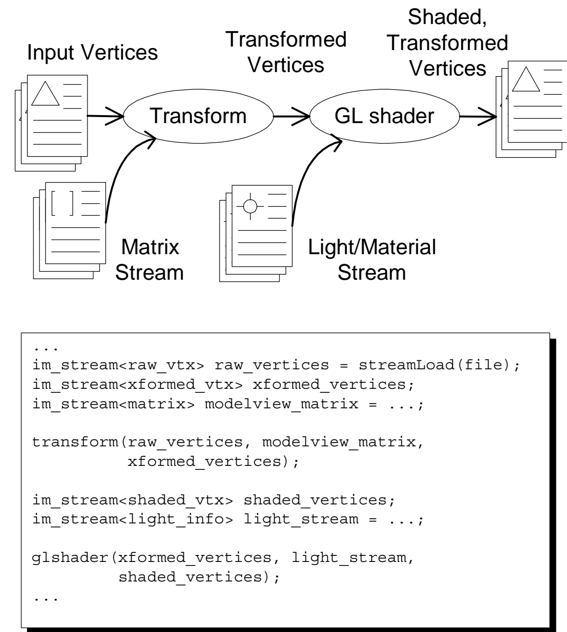


Figure 1: A simple geometry pipeline first transforms then shades its input vertices. This example shows how this pipeline maps into streams (indicated by arrows) and kernels (indicated by ovals). Note the intermediate results (transformed vertices) are immediately fed into the shading kernel. The pseudocode indicates how this pipeline is expressed in a stream programming system; stream-level code is written with a C++ library.

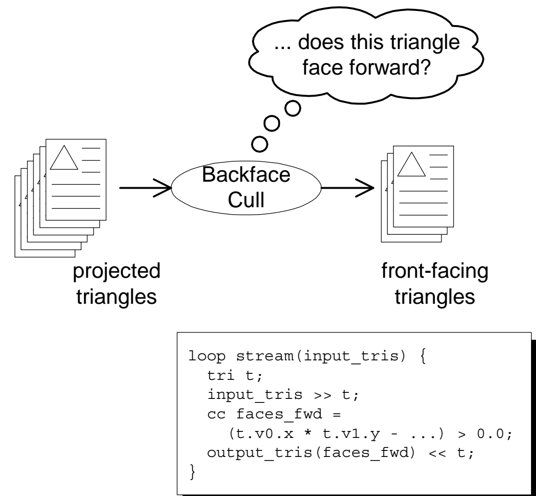


Figure 2: The backface cull kernel reduces a stream of triangles which face both forward and backward into a homogeneous stream of triangles all of which face forward. The $\gg$ and $\ll$ operators indicate stream input and output. The pseudocode indicates how this kernel is expressed in a stream programming system; kernel-level code is written in a subset of C++.

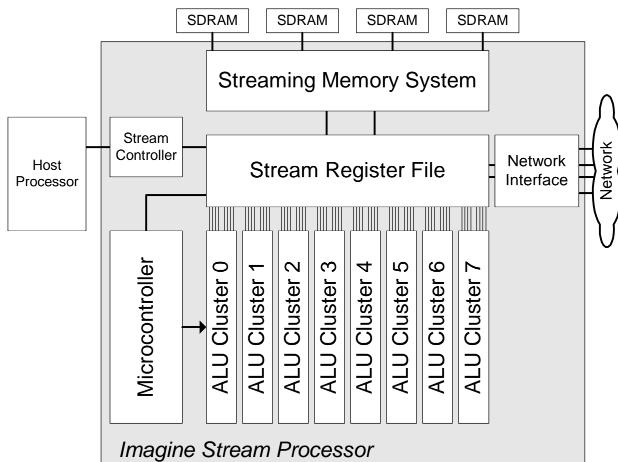


Figure 3: The Imagine stream processor.

Figure 2 contains an example of a backface cull kernel which removes elements from its input stream.

Finally, scalability using the stream programming model is straightforward. Separate kernels can run on multiple nodes of a system while data flows in streams from one node to another. Computation is divided among multiple nodes either at the task level, where separate kernels run on each node, or by running the same kernel on multiple nodes, each working on a subset of the data.

2.2 The Imagine Stream Processor

Figure 3 shows a block diagram of Imagine, a single-chip stream processor that supports the stream programming system. Details on Imagine's architecture, reordering memory system, and register organization can be found in Rixner et al. [26, 27, 28].

Imagine is centered around a 128 KB stream register file (SRF). All data in Imagine is handled as sequential streams passing between the SRF and its clients, which include a streaming memory system supporting two memory streams, a stream controller, a network interface for communication in a multi-node system, a microcontroller, and eight arithmetic clusters. The stream controller communicates with the host and sequences and issues stream operations. The microcontroller contains the microcode store and controls kernel execution in the arithmetic clusters.

Imagine achieves high performance on stream applications for three reasons. First, Imagine efficiently manages data movement through the machine. Imagine's bandwidth hierarchy [26] is tailored to the needs of multimedia applications like polygon rendering. The bandwidth between Imagine and its off-chip memory is 4 GB/s; the SRF's bandwidth to the clusters is 32 GB/s; and the aggregate bandwidth of all of the local register files in the clusters is 544 GB/s.

Each level of the memory hierarchy is optimized to implement the stream programming system. Imagine's streaming memory system [27] is designed to maximize delivered bandwidth. On chip, the local storage and reuse of intermediate streams (producer-consumer locality) is captured in the SRF, allowing Imagine to amortize the cost of memory accesses over all kernels in the pipeline. Finally, Imagine's partitioned register organization [28] captures data locality within kernels and capably feeds operands to the cluster functional units with substantial area, delay, and power savings over a conventional, central register file.

Second, Imagine's SIMD cluster organization takes advantage of the inherent parallelism in polygon rendering while maintaining the efficiencies of processing homogeneous streams characteristic of the stream programming system. The end result of this organization is ample arithmetic throughput to attack the high computational needs of polygon rendering. Each kernel runs on all eight clusters while processing its input streams and completes the processing of its input streams before the next kernel begins. Only one kernel runs at any point in time.

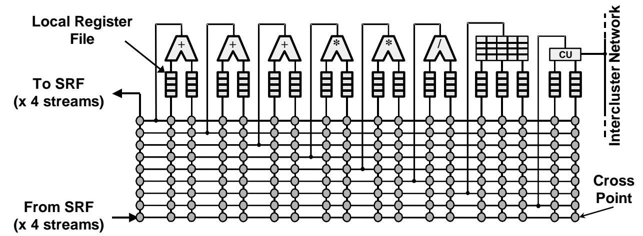


Figure 4: An Imagine cluster. Each functional unit input is fed by a local register file, and the outputs are switched by a cluster-wide switch.

Each of Imagine's eight clusters, shown in Figure 4, contains six arithmetic functional units that operate under VLIW control. The arithmetic units operate on 32-bit integer, single-precision floating point, and 16- and 8-bit packed subword data. The six functional units comprise three adders that execute adds, shifts, and logic operations; two multipliers; and one divide/square root unit. In addition to the arithmetic units, each cluster also contains a 256-word scratchpad register file that allows runtime indexing into small arrays, and a communication unit that transfers data between clusters. Each input of each functional unit is fed by a local two-port register file. In each cluster are 1104 registers spread across 17 local register files.[1] A cluster switch routes functional unit outputs to register file inputs. Across all eight clusters, Imagine has peak arithmetic bandwidth of 20 GOPS on 32-bit floating-point and integer data and 40 GOPS on 16-bit integer data. Imagine achieves about half of this peak bandwidth on typical media applications.

Third, the granularity of operations on a stream architecture is at the stream level. Kernel operations, loads, and stores apply to entire streams and complete when each element has been processed. Thus, the instruction bandwidth in this architecture is less than that of a conventional architecture, since complex operations on streams can be issued with just a single instruction.

Currently, Imagine's architecture and cycle accurate simulator are complete; its kernel scheduler is complete except for register spilling; and its runtime programming system and synthesizable RTL Verilog model are nearly complete. Extensive circuit studies and floorplanning of the layout are also finished. Imagine will be fabricated in a 0.18 μm process with a die size less than 0.7 cm^2. It is expected to operate at 500 MHz and tape out at the end of 2000.

3 Polygon Rendering

Our renderer is implemented as a pipeline of kernels as illustrated in Figure 5. The pipeline's inputs are streams of input data primitives and other input parameters, such as transformation matrices and lighting parameters; its output is an RGB image. We choose to concentrate our effort on an OpenGL-like pipeline: Our system runs in immediate mode, does not require frame semantics, and respects OpenGL ordering semantics. However, it is important to

[1] We use 64-word local register files for the simulations cited in this paper because we have not yet added register spilling to the kernel scheduler, but the final machine will have fewer registers.

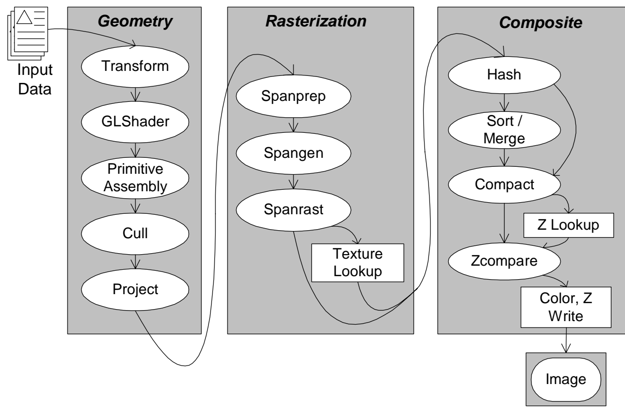


Figure 5: Our general pipeline consists of a series of kernels connected by streams; we divide these kernels into geometry, rasterization, and composite stages.

note that because we have a programmable system, we could extend OpenGL's functionality or implement other APIs.

Our pipeline kernels are grouped into three stages. In the geometry stage, input primitives are transformed from object space to screen space. The rasterization stage converts screen-space primitives to fragments, and the composite stage sorts and assembles the fragments into a final image.

Each stage consists of several kernels connected by data streams. For different rendering contexts, we exploit the programmability of our system and use optimized pipelines and specialized kernels. In general, however, the geometry stage consists of transform, lighting, primitive assembly, backface culling, and projection kernels; the rasterization stage uses triangle setup ("spanprep"), span generation ("spangen"), and span rasterization ("spanrast") kernels; and the composite stage has hash, sort, compact, and depth buffering kernels.

For efficiency, vertices are arranged into input batches; stream operations are applied to an entire input batch at the same time. In a real system, the application or driver would assemble the input into batches, but in our tests we batch the input before beginning processing. This method is modeled after the vector programming technique of strip-mining [14, 17].

3.1 Implementation Challenges

Load balancing Stages of the rendering pipeline such as scan conversion and span rasterization require unequal amounts of work for each input element. Large triangles, for instance, require more work to scan convert than do smaller ones. The challenge of load balancing is to distribute the work across Imagine's clusters in an efficient manner.

Our implementation uses dynamic load balancing to keep all of the clusters busy even when operating on elements that require vastly different amounts of work. With this approach, each cluster works on one input element at a time as the clusters iterate through the input stream. On each loop iteration, any cluster that has completed its old element fetches a new element from the input stream. Clusters that have not completed their previous element continue without fetching a new input.

Because the clusters are controlled in a SIMD fashion, this mechanism incurs the overhead of fetching a new input on each iteration whether or not the fetch actually takes place. However, it has the advantage of giving the functional units useful work at all times. Our system uses this method for scan conversion and span rasterization; in the latter, for example, we process spans by looping over fragments and bringing a new span into each cluster only when the old span is exhausted. As long as each cluster still contains valid input, it outputs a valid fragment on each loop iteration.

Ordering OpenGL's ordering rules require that input primitives must be completed in order. Parallel implementations of OpenGL that work on several primitives at once for efficiency must take special care to meet this requirement. In our implementation, the output of the rasterize stage of our pipeline generates fragments out of order. Consequently, for applications that require ordered fragments, such as alpha-compositing transparency, we must reorder the fragments.

To accomplish this reordering, we first assign a unique, ordered ID to each triangle in the batch. This ID is assigned during primitive assembly and is trivially generated by incrementing a counter each time a new triangle is passed through this kernel. During rasterization, each triangle passes its ID to each of its generated fragments.

Fragments carry color, depth, and an offset into the framebuffer. [2] Ordering requires that only fragments with the same offset must now be processed in ID order. While a sort of all fragments by offset and ID would meet the ordering requirement, the cost of this sort would dominate the runtime. However, many if not most fragments in a batch of tens or hundreds of fragments do not conflict with any other fragments; their depth complexity within the batch, on average, is rarely greater than one.

Instead of a general sort, then, we separate the fragment stream into two streams, a unique stream made of fragments which have no conflicts with other fragments in the batch, and a conflicting stream, consisting of the remaining fragments. This separation is performed by hashing the offsets of each of the fragments in the batch into a conflict table. In our implementation, we use a 12-bit hash value constructed from the low 6 bits of each of the x and y coordinates of the fragment's framebuffer offset. At 2 bits per hash entry, this uses 32 words in each of the 8 Imagine scratchpad register files.

The conflicting stream is then sorted and appended to the unique stream; the resulting fragment stream can then be processed and retired while obeying the ordering constraint.

3.2 Implementation Advantages

Capturing producer-consumer locality Figure 6 demonstrates how we capture the producer-consumer locality of the intermediate streams that are passed between the kernels of our pipeline. Because large data sets do not fit into our SRF, we partition the input primitives into batches and process an entire batch at one time. We choose the size of a batch such that at all times the working set of streams fits into the SRF.[3] Thus, intermediate data is passed from one kernel to the next via the SRF (32 GB/s) and need not be passed through the memory (4 GB/s).

Latency tolerance The design of our pipeline affords significant overlap between computation and memory access, hiding the cost of the memory operations. For example, while the arithmetic clusters process a batch of vertices, the next batch can be fetched from memory and be present in the SRF by the time the current batch's processing is complete. In fact, in our implementation, the depth buffer and texture lookups associated with one batch execute concurrently with geometry processing of the next batch.

To exploit this concurrency, Imagine takes advantage of its ability to overlap the execution of independent instructions. Imagine's

[2] The framebuffer is treated as a linear stream, so the offset is just a function of the x and y coordinates of the fragment.

[3] We discuss methods of processing large triangles, which may generate more fragments than can be stored in the SRF, in Section 5.2.

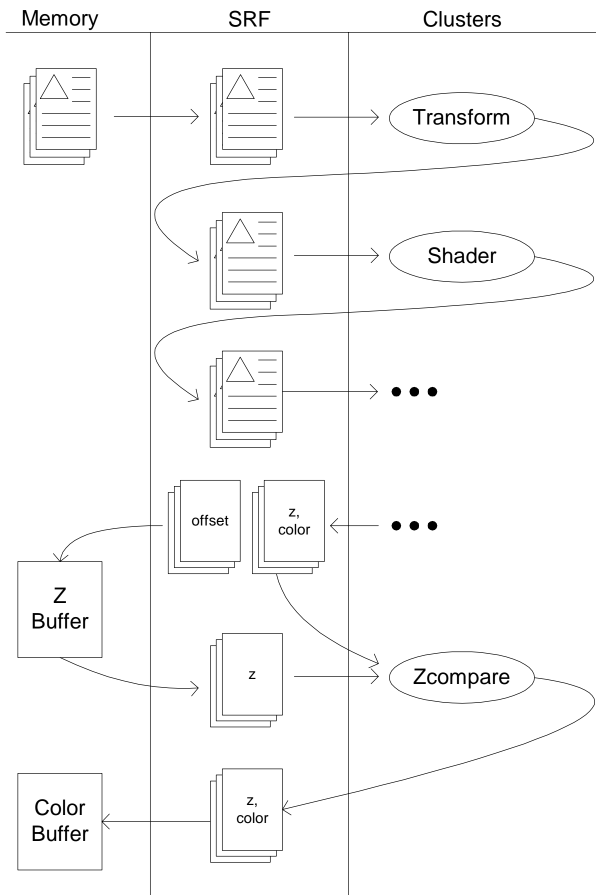


Figure 6: This sample polygon rendering pipeline shows how the locality of the intermediate streams are captured in the stream register file. Memory accesses are only necessary for the initial load of a batch of triangles and for buffer accesses. All other data flows through the SRF and does not leave the chip.

stream controller issues a single instruction to the microcontroller to execute a complex kernel on an entire batch of primitives. While this instruction is executing, the stream controller can issue a separate instruction to the memory system to load the next batch of primitives into memory, and so on. Dependencies between instructions are enforced by the stream controller to ensure that this overlap does not violate sequential semantics.

Pipeline reordering A programmable pipeline offers the opportunity to reorder the steps of the traditional OpenGL pipeline for more efficient execution. Our ADVS-1 test scene, described in the next section, is an excellent example. ADVS-1's rendering context has depth buffering enabled and blending disabled, so the texturing operation can be moved after the depth test. Thus, fragments which fail the depth test do not incur the cost of the texture lookup.

Flexible resource allocation A hardwired pipeline must fix the allocation of resources to different stages of the graphics pipeline at fabrication time. Typically the resources are balanced for some typical input scene, but may become very unbalanced for other scenes. For example, a scene with very large triangles may leave the geometry stages idle while the rasterizers are overloaded and a scene with very small triangles may overload the geometry stages while the rasterizers go idle. In contrast, our pipeline implementation uses the same hardware (SRF, arithmetic clusters, and memory system) for each stage of the pipeline. Thus our pipeline dynamically adapts to the balance of work between the different stages and avoids the inefficiency of fixed resource allocation.

4 Experimental Setup

We implemented our polygon renderer on a simulator of the Imagine stream processor. This simulator models the complete Imagine architecture, including computation, stream and kernel level control, and memory traffic and control,[4] with cycle accuracy and has been validated against our RTL models and circuit studies. At the beginning of each simulation, all input data streams (the list of vertices, transformation matrices, lighting parameters, etc.) were located in Imagine's main (off-chip) memory, and at the end, the complete output image was also in Imagine's main memory.

We compared our results with representative hardware-accelerated and software-only OpenGL systems. The hardware-accelerated system is a 450 MHz Intel Pentium III Xeon workstation with 128 MB of RAM running Microsoft Windows NT 4.0. Its graphics system is an NVIDIA Quadro with DDR SDRAM on an AGP4X bus running NVIDIA's build 363 of their OpenGL 1.15 driver. The software-only system is the same machine and graphics hardware with OpenGL hardware acceleration disabled, using Microsoft's opengl32.dll, GL version 1.1.

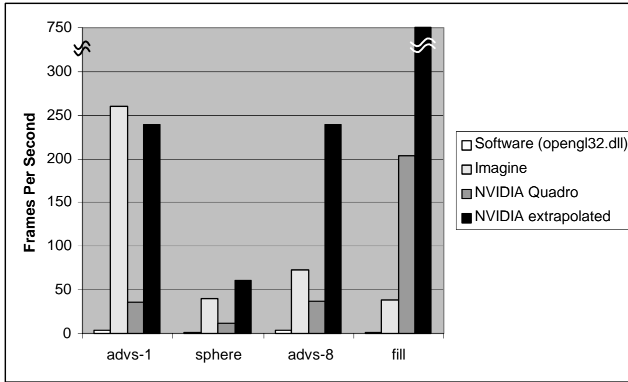
We measured the performance on these scenes with a runtime utility that recorded all of the OpenGL commands made by an application in a trace file. On the PC systems, we played back these traces on a low-overhead trace player to remove the effects of application time on frame rate. We eliminated startup costs by allowing the system to warm up (in particular, to load textures into texture memory) and then averaging frame times over hundreds of frames. Refresh synchronization costs were eliminated by disabling the vertical retrace sync, allowing a new frame to begin immediately after the old frame completed.

With this setup, we accurately model Imagine's chip and memory performance but not its performance in a complete system. A comparison against a commercial system, then, is biased in favor of Imagine, because real systems have other bottlenecks that are not present in the Imagine simulation. In particular, the interaction between the host processor and the graphics subsystem is not modeled, and many hardware-accelerated systems are limited by the bus between the processor and the graphics subsystem. On the scenes tested, we expect the bus communication overhead to be small, but more complex scenes will have a greater cost associated with this communication.

For this reason, we linearly scale the hardware-accelerated system's delivered performance to reach NVIDIA's peak quoted vertex numbers of 15 million vertices per second [10] and trilinear-filtered pixel fill rate numbers of 480 million pixels per second and assume that the causes of the difference are strictly system issues that we do not model in our Imagine simulation as well as departures from the hardware's fast path. The figures cited for this hardware-accelerated extrapolated system could be regarded as performance that could not be exceeded by the hardware system.

The NVIDIA system also uses an older technology: Having shipped in November 1999, it is built in a 0.22 μm process with approximately 23 million transistors; Imagine is targetted for a 0.18 μm process with 16 million transistors. Imagine's clock rate is also significantly higher (500 MHz vs. 120 MHz). While some of this difference can be attributed to the speedup from a smaller feature

[4] We modeled Imagine's main memory with a cycle-accurate simulation of the NEC μPD45D128164 DDR SDRAM [20], clocked at 125 MHz.



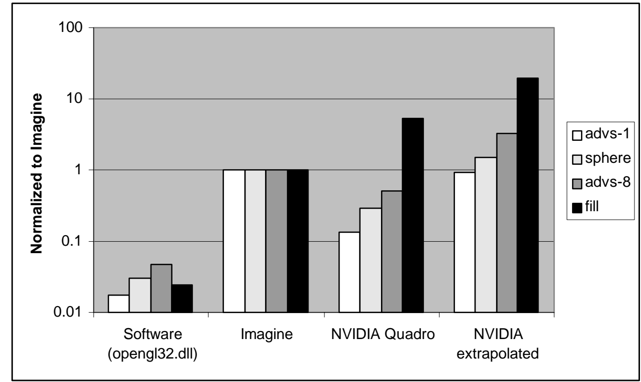


Figure 7: In the left graph, we compare the performance of each system on our four test scenes. On the right graph, we present the performance relative to Imagine = 1 for each scene.

size,[5] Imagine's high clock rate is also the result of our design decision to achieve high arithmetic rates by using deeper pipelines and a faster clock rate instead of more complex control and more arithmetic units, a difference which has nothing to do with the technology. Of course, clock rate in general is by no means the best predictor of overall performance.

Despite all these differences, a comparison between current commercial graphics systems and Imagine is instructive for two main reasons. First, running our test scenes on commercial systems gives a sense of the relative complexity of those scenes. Second, comparing several different scenes against commercial systems indicates on which kinds of graphics tasks Imagine performs relatively well and relatively poorly, providing insight into the bottlenecks of our implementation, both in software and hardware.

4.1 Test Scenes

To facilitate comparison between scenes, each of these scenes was rendered into a 24-bit RGB color framebuffer with a window size of 720 × 720 pixels. All textures are 32 bits, and mipmapping calculations are performed using a square root per fragment. References below to batch sizes apply only to the Imagine implementation.

Scene 1: Sphere Our first scene is an immediate-mode Gouraud-shaded rendering of a finely subdivided sphere lit with three positional lights with diffuse and specular lighting components. The sphere has 81,920 unmeshed triangles and generates 361,816 fragments. Backface culling is disabled so the depth complexity is 2 for each drawn pixel. SPHERE is rendered in 80-triangle batches.

Scene 2: Advs-1 The ADVS dataset is the first frame of the SPECviewperf 6.1.1 Advanced Visualizer benchmark with lighting and blending disabled and all textures point-sampled from a 512 × 512 texture map. The benchmark is run in immediate mode and has 62,576 vertices organized as polygons. It generates 25,704 triangles, of which approximately half are backface culled, and 70,384 fragments, and is rendered with 256-vertex batches.

[5] To first order, logic delays are inversely proportional to feature size, so the speedup for a 0.18 μm process should be roughly 22% over a 0.22 μm process.

Scene 3: Advs-8 This scene uses the same dataset and rendering modes as ADVS-1 except for filtering each texel using a mipmapped texture with base level size of 512 × 512. This scene is rendered in 24-vertex batches.

Scene 4: Fill This artificial benchmark simply fills its entire window with 20,000 mipmapped 25-pixel triangles arranged in long (200-vertex) triangle strips. The texture map has a base level size of 512 × 512. By construction, the texture for each pixel access is locked to mipmap level 0.5 and thus fetches and interpolates 4 texels from each of mipmap levels 0 and 1. FILL is rendered in 16-vertex batches.

5 Discussion

5.1 Imagine's Performance

The results of our experiments are summarized in Figure 7. We present our results in two ways. First, we show the raw performance of each system on each test scene in frames per second. Next, we normalize each system's performance to Imagine's in order to gauge Imagine's relative performance.

The immediate conclusion from a relative comparison between Imagine and the special-purpose hardware system is that Imagine performs relatively better on scenes weighted towards geometry performance and relatively poorer on scenes weighted towards rasterization performance. In the remainder of this section, we will analyze Imagine's performance and bottlenecks and suggest extensions to our implementation and our hardware.

Stream-level performance The time spent in one representative batch for the ADVS-1 dataset is illustrated in Figure 8. All 4 benchmarks achieve high occupancy in the arithmetic clusters: ADVS-8, the lowest, is busy 94.3% of the time and SPHERE, the highest, is busy 98.8% of the time. The remainder of the time can be accounted for by stream dependencies, where either the kernel cannot start because its input stream is not yet ready, or further stream operations cannot be reordered due to control dependencies. Figure 9 depicts the breakdown of kernel time for the entire ADVS-8 run.

This high occupancy in the clusters is achieved by efficiently managing memory bandwidth at all levels of the hierarchy. Except for vertex loads, depth buffer reads and writes, texture reads, and color buffer writes, all data references remain local in the SRF. The

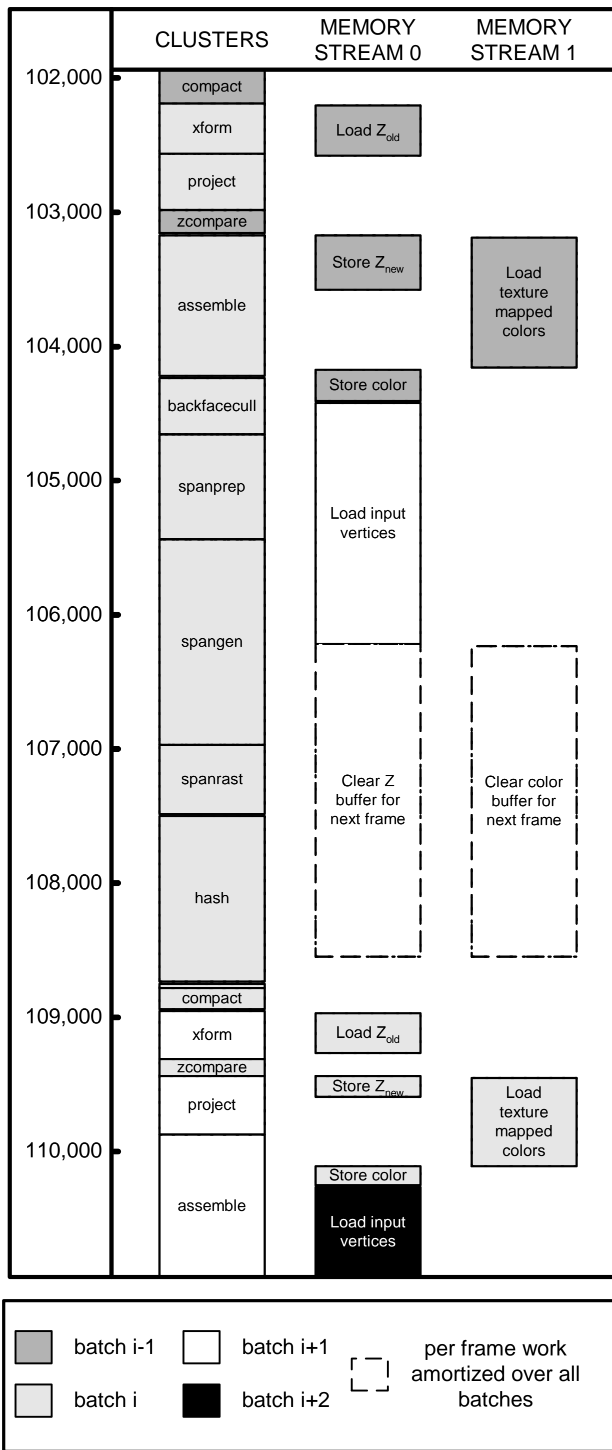


Figure 8: This representative batch of ADVS-1 depicts cluster and memory system occupancy. The leftmost column is cycle number. The CLUSTERS column shows which kernels were running during which cycles, and the MEMORY STREAM columns show the streams passing between Imagine and its off-chip memory. Note that because the memory system is not fully occupied during a batch, we distribute the work of clearing the depth and color buffers across the batches. (If depth and color buffer clears are done at the end of a frame instead, they take roughly 500,000 cycles, or 1 ms, for a 720 $\times$ 720 window.) The batch displayed has 256 vertices.

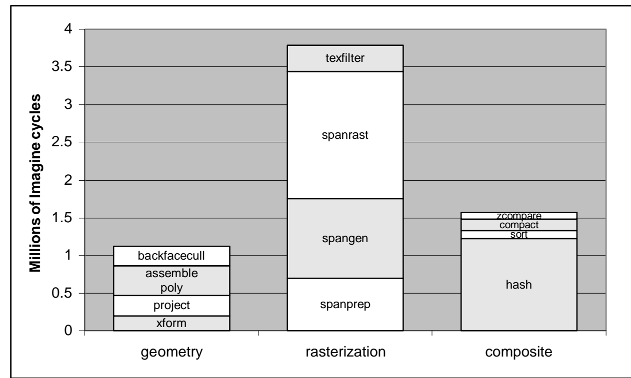


Figure 9: This chart indicates the relative kernel time, organized by stage, for each of the kernels used in the ADVS-8 pipeline for an entire frame.

benchmark sustaining the highest memory bandwidth is ADVS-1; in this benchmark, the average memory store latency is 126 Imagine cycles and the average memory load latency is 42 cycles. The reordered memory system is occupied 58.7% of the time. On this benchmark, the amplification of bandwidth from memory traffic (1.75 GB/s) to the SRF (9.39 GB/s) to the cluster register files (234.8 GB/s) demonstrates that we have adequately captured the locality of streams with our machine organization.

Kernel-level performance The use of data parallelism and software pipelining allows Imagine to sustain a high computation rate. For example, on SPHERE, Imagine averages 11.2 arithmetic operations per cycle across all the clusters for an aggregate rate of 5.58 GOPS. This high computation rate indicates that the stream programming system delivers high computational density on the polygon rendering application.

The $O(n \log n)$ cost of a general fragment sort makes the efficiency of the hash function vital for high performance. For example, the ADVS dataset, when batched in 256-vertex chunks, only has 14.7% of the fragments conflicting with other fragments, providing a frame rate speedup over an implementation with no hashing and a sort for all fragments of nearly 2:1. Still, many of these conflicting fragments have false conflicts with other fragments in the hash table; we plan to explore better hashing functions in future work.

Bottlenecks Imagine's relative performance against the hardware comparison system is poorest on the fill-intensive datasets. Special purpose hardware, with its powerful rasterization hardware, would be expected to excel at these scenes. For several reasons, Imagine falls short of this performance goal.

For mipmapped fragments, where each fragment requires one depth buffer read and write, one color buffer write, and eight texel reads, Imagine's memory system can sustain a peak fill rate of slightly less than 100 million fragments per second. Our hardware-comparison system quotes a fill rate of 480 million fragments per second [10], and even though this fill rate uses 16-bit texture samples and Imagine's uses 32-bit samples, the gap is still substantial. In Section 5.3, we discuss possible remedies to increase fill bandwidth.

The memory, however, is not the primary bottleneck on even a simple scene such as FILL. One bottleneck is simply the amount of computation, particularly in rasterization, as demonstrated in Figure 9: ADVS-8, with the same dataset, takes almost 2.5 times the

number of arithmetic ops to render than does ADVS-1.

Another bottleneck is the loss of efficiency from running shorter streams. Because mipmapped triangles, spans, and especially fragments take up more room than their unmipmapped counterparts, and because the SRF only has a finite amount of space, the number of primitives that can fit in a batch is greatly decreased. For instance, ADVS-1 uses batches with 256 vertices, but ADVS-8 can only fit 24. Shorter streams are less efficient for two reasons: first, each kernel has startup costs such as variable initialization which must be paid per batch, and second, software-pipelined kernels must incur the cost of priming and draining their software-pipelined inner loops. Comparing the runtime of the identical geometry computation in ADVS-8 and ADVS-1 reveals that ADVS-1's smaller batch size causes it to take more than twice the runtime as ADVS-8 in this section.

5.2 Implementation Extensions

Further pipeline developments Currently, the modelview and projection transformation matrices are input as a stream but do not change throughout the scene. Allowing arbitrary matrix changes could be handled in one of two ways. If matrix changes are infrequent, a new matrix could be loaded for each batch of geometry that passes through the pipeline. More frequent changes could be addressed with flags per input primitive (similar to the per-vertex flags used for carrying mesh information to the primitive assembly kernel) indicating a change in the transformation matrix.

We have not yet implemented clipping to the view frustum. This will most easily be handled after the project stage, which would output separate streams of primitives that trivially pass clipping and primitives that must be clipped, while eliminating primitives that trivially fail. The primitives that must be clipped will then be passed into a clip kernel. We will also analyze other algorithms such as those that clip during scan conversion [22]. Amortized over all input primitives, we expect that the cost of clipping will be on the same order as a matrix transformation; the more complex arithmetic in clipping will be offset by the need to clip only a subset of the primitives.

The benchmarks we chose had relatively uniform triangle size, but other datasets with huge triangles will invariably overflow the SRF with fragments. Our stream programming system supports a double-buffering mode that spills streams to memory as they become full at the cost of a degradation in performance. Instead, it may be desirable to split large triangles in screen space before rasterization, then process them in smaller batches which would not incur the cost of spilling. To handle triangle streams with larger variance in screen-space area, another option is to analyze the input size of primitives as they are brought into the clusters for the first time, stopping further transfer when the optimal batch size is reached. This solution imposes an up-front cost to estimate primitive size, but should allow greater efficiency and flexibility.

Until the texture lookup and final color generation, our algorithms use floating-point data throughout the entire rasterization pipeline. We plan to explore the benefits of using Imagine's packed 16- and 8-bit integer datatypes, which should provide gains in bandwidth through the clusters, operation density in the clusters, and SRF space utilization.

Multiple textures, though not supported in the current implementation, could simply be appended onto the stream of texture data. Each triangle would carry an offset into the texture stream (or multiple indices for multitextured pipelines) that would identify the base address of its texture.

Complex scenes Real scenes are more complicated than the scenes benchmarked above. Our implementation is well-suited to handle the larger datasets in real scenes. However, more complex scenes also imply many more rendering modes and options than our benchmarked scenes.

Much of the efficiency of our implementation comes from specialized kernels for specialized tasks. For example, the span rasterization kernel used for mipmapped fragments is not the same kernel as the Gouraud-shaded span rasterizer. While creating several specialized flavors for each kernel is both achievable and desirable, clearly, supporting every permutation of the pipeline is unworkable. Fortunately, the modularity of using kernels allows us to factor the complexity of supporting thousands of pipelines into merely supporting tens of kernels, and compositing those kernels together into specialized pipelines.

Schmidt and Lam describe a similar problem [30] and implement an evolutionary software system to address it. This dynamic method is applicable to our situation, and because kernel specialization information can often be inferred from compile-time analysis of the context, even a static method would work. Fortunately, even the most complex scenes typically use only a subset of the full power of their graphics APIs, and the runtime cost of frequent changes to the rendering context often means complex state changes are infrequent in demanding applications.

Exploiting programmability All of Imagine's kernels are implemented in microcode with no custom graphics specialization. Consequently, the performance of the graphics kernels in the OpenGL pipeline should translate well to kernels which are not in the pipeline. In future work, we intend to explore using the programmability of Imagine to add interesting features to the graphics pipeline, such as programmable shading and flexible multitexturing, the use of more complex input primitives such as subdivision surfaces, and the integration of non-polygon rendering approaches such as image-based rendering.

5.3 Imagine Hardware Extensions

Imagine was designed to run a variety of multimedia applications, including signal- and image-processing workloads. A stream architecture built to run only a polygon rendering pipeline might add additional hardware that would aid its performance. These features would take the form of additional clients to the SRF and additional functional units in a cluster. Adding either additional SRF clients or additional function units is supported by both the architecture and the software tools.

The addition of caching capabilities to the memory system would help texture performance. Modest amounts of texture cache on the order of 16 KB allow significant savings in bandwidth [12]. A bypassable cache attached to the memory system, with streams designated as either cacheable or noncacheable, bears further investigation. Though the scenes tested were not limited by texture bandwidth, more complex, multitextured scenes, or scenes with more sophisticated texture filtering algorithms, may benefit from a cache.

Another design option is to put a small programmable ALU in the memory system (as in Deering et al. [5]) that performs simple operations like a depth test, an alpha blend, or a filtering operation. This would avoid the round-trip from the memory system through the SRF to the clusters and back. Different cluster and functional unit organizations might also be interesting additions or modifications to the Imagine architecture.

The stream programming model is well-suited to multi-node implementations, and Imagine's 750 MB/s network interface provides a high-bandwidth path to pass streams between nodes of a multi-node system. Such a system could implement sort-middle or sort-last parallel machines. It could run one pipeline stage per node and pass intermediate results to the next node, or it could divide its input among multiple nodes, working on them concurrently and compositing their outputs in a final stage. Because of the large amount

of parallelism available in the graphics pipeline, we expect to be able to achieve nearly linear improvements in rendering bandwidth with the addition of Imagine processors to a rendering pipeline.

6 Previous Work

The use of streams as programming elements in media applications is common; Gabriel [2] and its successor Ptolemy [25] are representative examples. Imagine's contribution to the field is its use of streams as architectural primitives in hardware.

SIMD extensions to microprocessor instruction sets are used to exploit data-level parallelism and provide the option of more, lower-precision operations, which are useful in a number of multimedia applications. Sun Microsystems' VIS is described by Tremblay et al. [33] and Intel's MMX is described by Peleg and Weiser [24]. However, adding operations on partitioned data does not address the producer/consumer locality of graphics applications.

Imagine inherits many ideas from vector processor architectures. An early vector machine is the CRAY-1 [29]; a more recent vector architecture is the Berkeley VIRAM [13]. These vector machines lack the local register level of the bandwidth hierarchy and do not have the kernel-level instruction bandwidth savings of Imagine.

VLIW architectures, first described by Josh Fisher [7], have been common in programmable media processors due to their ability to effectively exploit instruction level parallelism. A recent example is the Equator MAP1000A [21], which uses both VLIW and SIMD organizations in its hardware. The use of a bandwidth hierarchy is not addressed by VLIW machines, nor do they simplify the instruction overhead.

The field of computer graphics has seen both architectures of varying programmability and architectures which use custom, special-purpose hardware. For many years, designers have taken advantage of the inherent parallelism in polygon rendering. Pixar's Chap (described by Levinthal and Porter [16]) was one of the earliest processors to explore a programmable SIMD computational organization, on 16-bit integer data; Flap [15], described three years later, extended Chap's integer capabilities with SIMD floating-point pipelines.

The Chromatic Mpact architecture [8] employs a VLIW processor together with a specialized rendering pipeline to add some degree of flexibility to rendering. The Pixel-Planes [9] and PixelFlow [19] family of architectures employ both custom SIMD processors built into enhanced memory chips that evaluate linear equations to accelerate rasterization and more general-purpose processors for other tasks and pipeline enhancements (such as programmable shading [23]). Imagine differs from both of these previous approaches in that it provides no specialization for rendering, but does provide a bandwidth hierarchy that enables much higher arithmetic rates without specialization.

Many systems implement the geometry stages of their pipelines with microcoded computational engines; a representative example is the LeoFloat unit of Deering and Nelson's Leo graphics system [4]. Rasterization is more commonly done with special-purpose hardware.

More recently, the vector units in the Sony Emotion Engine [6] use a single cluster of VLIW-controlled arithmetic units fed with vectors of data. The functional units are connected to either an integer or a floating-point centralized register file.

Recent manufacturers of high-performance consumer-level graphics accelerators have been reluctant to discuss the microarchitectures of their implementations. An exception is Digital's Neon [18], a single-chip graphics accelerator designed with special attention to reducing memory bandwidth requirements.

7 Conclusion

Implementing polygon rendering on a stream processor allows performance approaching that of special-purpose graphics hardware while at the same time providing the flexibility traditionally associated with a software-only implementation. This performance is achieved using a pipeline that is entirely programmed in software. The pipeline can easily be changed by adding, removing, or reordering the kernels, and new kernels can be created by programming in a subset of C++.

The Imagine stream processor achieves this level of performance because it matches the capabilities of VLSI technology to the needs of rendering algorithms in three ways. First, it exploits the producer-consumer locality to reduce the demand for memory bandwidth. Data generated by one pipeline kernel is staged in local storage, the SRF, and then used by the next pipeline kernel without ever being sent to main memory. Second, Imagine provides the large number of arithmetic units (48 total) needed to exploit the parallelism inherent in graphics applications and to provide the arithmetic density required. Finally, by overlapping stream operations, each of which moves or transforms hundreds of data records, Imagine is able to exploit the latency tolerance inherent in graphics applications to further increase throughput. In short, Imagine achieves its performance using the same methods that have traditionally been exploited in special-purpose hardware, but without giving up programmability.

Measurements of a rendering pipeline implemented on Imagine show that the architecture is successful in exploiting producer-consumer locality and in overlapping operations. On all test scenes, the pipeline is entirely compute bound with the arithmetic clusters busy more than 94% of the time while the memory system is busy a maximum of 59% of the time. However, there is considerable room for improvement: Imagine sustains only 5.58 GOPS, 28% of peak performance, on the SPHERE data set. Ideally, further advances in stream algorithms and architectures will make these scenes memory-bound.

There is a continuum of solutions between the general-purpose stream processor we have evaluated in this paper and a specialized graphics chip. Stream processors can easily be extended and specialized by adding either additional stream units, e.g., a texture cache or a scalar unit, or additional functional units. With an Imagine chip measuring only 0.7 cm^2, there is plenty of room for such specialized units, which can be easily supported by the existing programming tools. We expect that considerable performance gains can be realized by such limited specialization without giving up the flexibility and programmability of the stream processor.[6]

As both rendering techniques and models continue to increase in complexity in the quest for even greater realism, we see stream processing as an attractive direction for graphics architecture. A stream processor can easily be configured to handle new rendering methods by coding one or more additional kernels and integrating them into the rendering pipeline. With this approach, new or enhanced rendering methods can be applied with only a gradual degradation in performance. In contrast, using a new rendering method today usually means falling off the performance cliff associated with moving from special-purpose hardware to a software-only solution. Because it manages bandwidth efficiently at all levels of the storage hierarchy, a single stream processor has performance that is competitive with special-purpose solutions. At the point where both are bandwidth-limited, there is no advantage to specialization. Moreover, multiple stream processors can easily be applied to running larger models by exploiting either control parallelism (partitioning the pipeline across processors) or data parallelism.

[6] Of course, specialized features may be of little use in other stream processor application areas such as video compression, radar processing, and signal processing for wireless and wire-line communication.

8 Acknowledgements

The research described in this paper was supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C0037. The authors also acknowledge the support of Stanford Graduate and National Science Foundation Graduate Fellowships. The authors thank Kekoa Proudfoot for both his mipmapping reference implementation and the trace utility used to generate the ADVS trace and Matthew Eldridge for acquiring the PC performance numbers. Russ Brown, Matthew Eldridge, Pat Hanrahan, and Homan Igehy all made useful comments on drafts of the paper, and Magdalen Powers capably copyedited the submission.

References

[1] Chris Anderson. Smooth operator. *The Economist*, 6 March 1999.

[2] Jeffrey C. Bier, Edwin E. Goei, Wai H. Ho, Philip D. Lapsley, Maureen P. O'Reilly, Gilbert C. Sih, and Edward A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, 1990.

[3] James F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78)*, 12(3):286–292, August 1978.

[4] Michael F. Deering and Scott R. Nelson. Leo: A system for cost-effective 3D shaded graphics. *Proceedings of SIGGRAPH 93*, pages 101–108, August 1993.

[5] Michael F. Deering, Stephen A. Schlapp, and Michael G. Lavelle. FBRAM: A new form of memory optimized for 3D graphics. *Proceedings of SIGGRAPH 94*, pages 167–174, July 1994.

[6] Keith Diefendorff. Sony's emotionally charged chip. *Microprocessor Report*, 13(5), 1999.

[7] J. A. Fisher. Very long instruction word architectures and ELI-512. In *Proceedings of the 10th Symposium on Computer Architecture*, pages 478–490, 1983.

[8] Pete Foley. The Mpact(tm) media processor redefines the multimedia PC. In *Proceedings of IEEE COMPCON*, pages 311–318, 1996.

[9] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):79–88, July 1989.

[10] Peter N. Glaskowsky. Most significant bits: NVIDIA GeForce offers acceleration. *Microprocessor Report*, 13(12), 1999.

[11] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics & Applications*, 6(11):21–29, November 1986.

[12] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120, 1997.

[13] Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report CSD-99-1059, University of California, Berkeley, 1999.

[14] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[15] Adam Levinthal, Pat Hanrahan, Mike Paquette, and Jim Lawson. Parallel computers for graphics applications. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–198, 1987.

[16] Adam Levinthal and Thomas Porter. Chap — a SIMD graphics processor. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):77–82, July 1984.

[17] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[18] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon — a single-chip 3D workstation graphics accelerator. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 123–132, August 1998.

[19] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):231–240, July 1992.

[20] NEC Corporation. 128M-bit synchronous DRAM with double data rate 4-bank, SSTL_2 data sheet. Document No. M13852EJ1V1DS00, 1st Edition, December 1998.

[21] John Setel O'Donnell. MAP1000A: A 5W, 230 MHz VLIW mediaprocessor. In *Proceedings of Hot Chips 11*, pages 95–109, 1999.

[22] Marc Olano and Trey Greer. Triangle scan conversion using 2D homogeneous coordinates. *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 89–96, August 1997.

[23] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The PixelFlow shading system. *Proceedings of SIGGRAPH 98*, pages 159–168, July 1998.

[24] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 42–50, August 1996.

[25] José Luis Pino. Software synthesis for single-processor DSP systems using Ptolemy. Master's thesis, University of California, Berkeley, May 1993.

[26] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–13, 1998.

[27] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, to appear, 2000.

[28] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval Kapasi, and John D. Owens. Register organization for media processing. In *Proceedings of the Sixth Annual International Symposium on High-Performance Computer Architecture*, pages 375–386, 2000.

[29] R. M. Russell. The CRAY-1 processor system. *Communications of the ACM*, 21(1):63–72, January 1978.

[30] Brian K. Schmidt and Monica S. Lam. A compiler for creating evolutionary software and application experience. Technical Report CSL-TR-99-782, Stanford University, April 1999.

[31] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 1.2), March 23, 1998.

[32] Alvy Ray Smith. The cameraless movie. *Scientific American*, to appear, 2000.

[33] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 6(4):10–20, August 1996.

[34] Steve Upstill. *The Renderman Companion*. Addison-Wesley, 1990.