COMPUTER GRAPHICS ON A STREAM ARCHITECTURE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John Douglas Owens
November 2002

Abstract

Media applications, such as signal processing, image and video processing, and graphics, are an increasing and important part of the way people use computers today. However, modern microprocessors cannot provide the performance necessary to meet the demands of these media applications, and special purpose processors lack the flexibility and programmability necessary to address the wide variety of media applications. For the processors of the future, we must design and implement architectures and programming models that meet the performance and flexibility requirements of these applications.

Streams are a powerful programming abstraction suitable for efficiently implementing complex and high-performance media applications. This dissertation focuses on the design, implementation, and analysis of a computer graphics pipeline on a stream architecture using the stream programming model. Rendering on a stream architecture can sustain high performance on stream hardware while maintaining the programmability necessary to implement complex and varied programmable graphics tasks.

We demonstrate a complete implementation of an OpenGL-like pipeline in the stream programming model, and that the algorithms developed for this implementation are both well-suited for the stream programming model and make efficient and high-performance use of stream hardware. We identify and discuss aspects of our system that impact its performance, including triangle size, rasterization algorithm, batch size, and short stream effects, and discuss the implications of programmability in the pipeline. We demonstrate and analyze the scalability of the algorithms and the implementation in order to anticipate performance on future generations of stream processors. And finally, we describe, implement, and analyze a second rendering pipeline, the Reyes pipeline, and compare it to our OpenGL pipeline in order to show the flexibility of the programming model and to explore alternative directions for future stream-based rendering pipelines.

Foreword

Rendering of photorealistic scenes from geometric models is a pervasive and demanding application. Over a hundred times more arithmetic operations are performed in the graphics processor of a modern PC than in its CPU. As powerful as modern graphics processors are, their flexibility is limited. They perform a fixed function pipeline with just one or two programmable stages. Rendering tasks that fit within this pipeline they perform with blinding speed; rendering tasks that don't conform to the fixed pipeline cannot be handled at all.

In this pioneering work, John Owens shows how a programmable stream processor can achieve performance comparable to modern fixed-function graphics processors while having the flexibility to support arbitrary rendering tasks. John demonstrates the performance and flexibility of stream-based rendering by implementing diverse rendering pipelines: an OpenGL pipeline that renders polygons and a Reyes pipeline that uses subdivision surfaces on the Imagine stream processor. He also demonstrates flexibility by compiling several shaders, described in a programmable shading language, to execute on a stream processor. The techniques John demonstrates go far beyond the capabilities of contemporary graphics hardware and show that programmable hardware can smoothly tradeoff frame rate against rendering features.

A number of novel methods were developed during the implementation of these rendering pipelines. The use of barycentric rasterization was employed to reduce the burden of carrying large numbers of interpolants through the rasterization process. A novel load balancing mechanism using conditional streams was developed to maintain high duty factor in the presence of unequal-sized triangles. Also, a compositor based on hashing and sorting was developed to ensure in-order semantics when triangles are rendered out-of-order.

John deeply analyzes the pipelines he has developed shedding considerable insight onto the behavior of streaming graphics and of stream processing in general. He looks at issues such as short-stream effects to show how performance is gained and lost in a typical stream pipeline. He also looks beyond the Imagine processor to study how well these pipelines would work as stream processors are scaled or are augmented with special-purpose units—such as a rasterizer or texture cache.

Contemporary graphics processors have already adopted some of the technology of stream processors. They incorporate programmable vertex and fragment shading stages that are in effect small stream processors embedded in a fixed-function pipeline. As time goes on, one can imagine more pipeline stages becoming programmable until commercial graphics processors become general purpose stream processors capable of performing a wide range of intensive streaming computations.

William J. Dally
Stanford, California

Acknowledgements

I have had the privilege and pleasure to work with incredibly talented and motivated colleagues during my time at Stanford. The success of the Imagine project is the direct result of the dedication and hard work of the graduate students who worked on it. I would particularly like to acknowledge those with whom I've collaborated most closely—Ujval Kapasi, Brucek Khailany, Peter Mattson, Scott Rixner, and Brian Towles—for their creativity and effort over the past five years. More recently, Jung Ho Ahn, Abhishek Das, and Ben Serebrin have been instrumental in bringing up the Imagine system, and I know the project will be left in good hands with them.

In addition to my colleagues in the Imagine group, the talented graduate students in both the Concurrent VLSI Architecture Group and the Computer Graphics Laboratory helped guide this work through formal talks and informal conversations as well as made my time at Stanford full of intellectual challenge and stimulation. It has been an honor and a privilege to have been part of two such outstanding research groups with their overwhelming excitement and energy.

Working with Kekoa Proudfoot and Bill Mark on the shading language backend to their Real-Time Shading Language was also a treat. Their system was incredibly well designed and I appreciate their time in helping me integrate my changes and additions into it. Kekoa also wrote the first draft of Section 4.1.

The first version of the OpenGL pipeline described in Chapter 4 was published as joint work with Bill Dally, Ujval Kapasi, Peter Mattson, Scott Rixner, and Ben Mowery. Brucek Khailany and Brian Towles did the initial work on the Reyes pipeline of Chapter 8 as part of a class project, and our resulting paper on that pipeline was also coauthored with Bill Dally with substantial input from Pat Hanrahan. Kekoa Proudfoot and Matthew Eldridge aided in acquiring the commercial hardware and software comparisons in Section 5.2. Jung Ho Ahn helped get the texture cache results in Section 6.5 and Ujval Kapasi aided with the single-cluster comparison of Section 5.2.4 and the integration of the hardware rasterizer of Section 6.6. Finally, David Ebert provided the marble shader used in the MARBLE scene, described in Section 5.1.

The ideas and results in this dissertation have been influenced by the candid comments and criticisms of a large number of people besides those named above. Thanks to Kurt Akeley, Russ Brown, Matthew Eldridge, Henry Moreton, Matt Papakipos, Matt Pharr, and the Flash Graphics Group for many productive conversations.

Previous to Imagine, I enjoyed working on (and learned how to do research in!) the Lightning project with Matthew Eldridge under the capable direction of Pat Hanrahan. Gordon Stoll and Homan Igehy also worked in the Flash Graphics group and were helpful resources as I began my research career. Other senior graduate students who provided guidance for me both before and after I arrived at Stanford were John Chapin, David Heine, Mark Heinrich, Robert Jones, Jeff Kuskin, Phil Lacroute, and Dave Ofelt.

My advisor, Bill Dally, has ably led the Imagine project and has provided valuable guidance as I have progressed through my graduate studies. Working for Bill put this graduate student in overdrive—he properly demands much from his students. I have appreciated Bill's intelligence, his experience, and his honesty and ability to take criticism during our conversations, as well as his leadership in fearless descents of too-steep ski slopes and his more general desire to work hard and play hard with his graduate students. Pat Hanrahan was my advisor when I started my graduate career and has served as my second advisor for this work. His constant encouragement and enthusiasm have been an inspiration to me in the work I've done here. Matt Regan, my third reader, is full of energy and great ideas and I have appreciated both working with him at Interval as well as his suggestions on this work.

This research would not have been possible without the support from our funding agencies. I gratefully acknowledge the support of the Defense Advanced Research Projects Agency (under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C0037), the IMTV project and its industrial sponsors Intel, Interval, and Sony, and the DARPA Polymorphous Computing Architectures Project (monitored by the Department of the Air Force under contract F29601-00-2-0085). My first year at Stanford was funded by the College of Engineering Lawrence R. Thielen Memorial Fellowship and I appreciate its financial support as well.

At Stanford I have participated in not only an engaging and fascinating intellectual journey but also had the privilege of sharing it with the many friends I have met here. Hard work was often interrupted by lunch with Ian Buck, Matthew Eldridge, Greg Humphreys, Matt Pharr, Kekoa Proudfoot, and Jeff Solomon. My officemates Matthew, Milton Chen, and Niloy Mitra made Gates 396 an enjoyable place to work. Alex Aravanis, Mohammed Badi, Scott Pegg, and Greg Yap often accompanied me on weekend Games. John Tanner, Susan Ortwein, and Dante Dettamanti allowed me to spend many happy hours in the pool with their water polo squads. I've also had the support and encouragement of many other friends during my time at Stanford, among them Adrian, Brent & Linda, Brian, Diane, Greg, Kari, Kjell & Katie, Rob & Jenn, Summer, and the Men of 1620 Spruce. And my roommates—Alex Aravanis, Yee Lee, Mark Owens, Toby Rossmann, and Ryohei Urata—have put up with me for varying amounts of time, but not nearly as long as Hank Jones and Vijit Sabnis, with whom I've lived for seven years and counting. Thanks to all of them for making our various residences great places to live.

Finally, and most importantly, I've been blessed with the unvarying support of my family in everything I've ever done. My parents, John and DeDe, my Davis and Owens grandparents, my sister Jennie and brother-in-law Bryan, my brother Mark, and my fiancée Shana have always provided perspective, advice, encouragement, and kind words throughout my time at Stanford. To them I am most grateful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is said that “a picture is worth a thousand words”. Computer graphics is the study of constructing computer-generated images from descriptions of scenes, images that are produced both quickly and with realistic detail.

The last decade has brought significant change to the systems through which computers produce graphics. Ten years ago, dedicated hardware for computer graphics was only available in expensive workstations. This hardware was implemented with multiple chips or boards and typically cost thousands of dollars.

Today, the vast majority of personal computers include high-performance graphics hardware as a standard component. Graphics accelerators are typically implemented on a single chip, implement popular programming interfaces such as OpenGL or Direct3D, and cost in the hundreds of dollars. And despite their smaller size and modest cost, they deliver performance several orders of magnitude above their counterparts of ten years ago.

High performance is necessary to meet the demands of many applications: entertainment, visual simulation, and other tasks with real-time interactive requirements. Systems that produce real-time performance must render many images per second to achieve interactive usage. Performance, however, is only half of the story.

Together with the goal of real-time performance has been continued progress toward high visual fidelity and in particular photorealistic images. A major user of such functionality is the entertainment industry, both for special effects and for computer-generated motion pictures.

In these applications, the production of high-quality images typically take on the order of hours. An example is computer-generated movies: frames from Pixar's animated motion pictures usually take several hours to render. These images are not produced by special-purpose graphics hardware but instead by general purpose processors, because special-purpose hardware lacks the flexibility to render all the effects necessary for these images.

Traditionally, the goals of high performance and high fidelity have been considered mutually exclusive. Systems that deliver real-time performance are not used to render scenes of the highest complexity; systems that can render photorealistic images are not used in performance-critical applications. With this work, we describe a system that will deliver both high performance and high flexibility, one that will meet the demands of a wide variety of graphics applications in future computing systems.

To accomplish this goal, we implement our graphics system on a computer system with both software and hardware components. Programs in our system are described in the *stream programming model* and run on a *stream processor*. Specifically, we use the Imagine stream processor and its programming tools as a basis for our implementation.

1.1 Contributions

This dissertation makes several contributions to the areas of stream processing and computer graphics.

- Our system is the first instance of a graphics pipeline developed for a stream processor. We have elegantly expressed the system in the stream programming model, using stream algorithms exploiting the parallelism of the rendering tasks in each stage of the graphics pipeline, and we have developed an efficient, high-performance implementation on stream hardware.

- Our pipelines mirror the functionality of the popular OpenGL and Reyes rendering pipelines. Doing so allows us to both compare our implementation with commercial implementations as well to demonstrate that we can efficiently implement a complex interface. We also develop a backend for our system to the Stanford Real-Time

Shading Language, demonstrating our system's suitability as a target for higher-level shading languages.

- We analyze the performance of our system in detail and pinpoint aspects of the hardware and software that limit our performance.

- We demonstrate that our system is scalable to future generations of stream hardware. As stream hardware becomes more capable, with a greater ability to perform computation, our implementation will continue to increase in performance.

1.2 Outline

We begin by describing the previous work in this area in Chapter 2. The stream programming model and our stream architecture are then described in Chapter 3.

Our implementation is presented in Chapter 4 and evaluated and analyzed in Chapter 5. Chapter 6 explores extensions to our stream architecture to achieve higher performance.

Chapter 7 discusses several points topical to the dissertation: a comparison of our machine organization against commercial processors, a discussion of programmability, and a set of lessons for hardware designers based on our experience. In Chapter 8, our implementation of the Reyes rendering pipeline is compared to our base implementation.

Finally, Chapter 9 offers conclusions, enumerates the contributions of this dissertation, and suggests directions for future work.

Chapter 2

Previous Work

Previous work related to the work in this dissertation falls into four categories. First, hardware and software has previously been optimized for media applications such as graphics. Second, the implementation of this thesis was built using the architecture and tools of the Imagine project, which have been described in other work. Third, high-performance graphics systems are primarily built from special-purpose graphics hardware, and the architectures of those systems have been influential in this work. And finally, previous work involving programmability in graphics has inspired the programmable features of the implementation described in this dissertation.

An earlier version of the core pipeline described in this dissertation was summarized in work published at the Eurographics Hardware Workshop in 2000 [Owens et al., 2000]; the Reyes work of Chapter 8 [Owens et al., 2002] was published at the same conference two years later.

2.1 Media Architectures and Tools

Many architectural features have exploited the parallelism inherent in media applications. First, SIMD extensions to microprocessor instruction sets are used to exploit subword data-level parallelism and provide the option of more, lower-precision operations, which are useful in a number of multimedia applications. Representative examples are Sun Microsystems' VIS [Tremblay et al., 1996] and Intel's MMX [Peleg and Weiser, 1996].

VLIW architectures [Fisher, 1983] have been common in programmable media processors due to their ability to effectively exploit instruction-level parallelism. The Equator MAP1000A [O'Donnell, 1999], for instance, uses both VLIW and SIMD organizations in its hardware.

And finally, data parallelism is used in vector processor architectures. An early vector machine is the CRAY-1 [Russell, 1978]; a more recent vector architecture is the Berkeley VIRAM [Kozyrakis, 1999]. (However, vector machines lack the local register level of the bandwidth hierarchy and do not have the kernel-level instruction bandwidth savings of Imagine.)

All three of these techniques—subword parallelism, instruction-level parallelism, and data-level parallelism–are used in the Imagine architecture and in the implementation described in this dissertation.

The use of streams as programming elements in media applications is common; Gabriel [Bier et al., 1990] and its successor Ptolemy [Pino, 1993] are representative examples. Imagine's contribution to the field is its use of streams as architectural primitives in hardware.

The MIT Raw Architecture Workstation (RAW) [Taylor et al., 2002] and StreamMIT [Gordon et al., 2002] projects have addressed similar application domains as the Imagine project. RAW is a simple, highly parallel VLSI architecture that exposes low-level details to the compiler. It features many connected processors on a single chip and hence divides its tasks in space rather than time like Imagine (discussed further in Section 7.1). StreaMIT is a language for streaming applications; instead of extending C++ like Imagine's StreamC and KernelC, it implements a new, more abstract language. Like Imagine's languages, StreaMIT uses a stream programming model, exposing both the native parallelism and the communication patterns inherent in the computation.

2.2 Imagine Hardware and Software

The Imagine project, used as the platform for the work in this dissertation, includes an architecture and physical implementation as well as a set of software tools used to write and test stream programs on Imagine simulations or hardware. Portions of the Imagine system have been described in previous publications.

An overview of the Imagine project is presented by Khailany et al. [Khailany et al., 2001]. The Imagine Stream Processor was first described by Rixner et al. [Rixner et al., 1998]; this work introduced the *data bandwidth hierarchy* and described Imagine at an architectural level. Imagine's memory access scheduler [Rixner et al., 2000a] and register organization [Rixner et al., 2000b] were described by Rixner et al. in later work. Kapasi et al. describes the conditional stream mechanism used to support data-dependent conditionals in kernels [Kapasi et al., 2000]. Kapasi et al. also discuss SIMD efficiency using in part data from Section 5.2.4 [Kapasi et al., 2002].

Imagine's kernel compiler schedules kernels to run on the Imagine clusters [Mattson et al., 2000]; its stream scheduler compiles stream-level code on the Imagine hardware and host [Kapasi et al., 2001]. Both are described in more detail in Mattson's dissertation [Mattson, 2002].

2.3 Graphics Hardware

The field of computer graphics has seen both architectures of varying programmability and architectures which use custom, special-purpose hardware. Möller and Haines summarize real-time rendering techniques, including graphics hardware, in their 1999 book [Möller and Haines, 1999].

One of the hallmarks of special-purpose graphics hardware is its ability to exploit the parallelism in rendering applications. Pixar's Chap [Levinthal and Porter, 1984] was one of the earliest processors to explore a programmable SIMD computational organization, on 16-bit integer data; Flap [Levinthal et al., 1987], described three years later, extended Chap's integer capabilities with SIMD floating-point pipelines. Our implementation also uses SIMD parallelism across clusters as well as limited subword parallelism in working on multiple stream elements at the same time.

Other approaches to exploiting parallelism include the Pixel-Planes [Fuchs et al., 1989] and PixelFlow [Molnar et al., 1992] family of architectures. They 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 [Olano and Lastra, 1998]). Our implementation has no specialization for rendering but does take advantage of the SIMD nature of the rendering tasks (though not to the degree of Pixel-Planes and PixelFlow) in a similar way.

The Chromatic Mpact architecture [Foley, 1996] employs a VLIW processor together with a specialized rendering pipeline to add some degree of flexibility to rendering. More recently, the vector units in the Sony Emotion Engine [Diefendorff, 1999] 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. Imagine's clusters also feature an internal VLIW organization; however, Imagine's intercluster SIMD organization allows greater computation rates by also utilizing the data parallelism of the rendering tasks.

Historically, special-purpose rendering hardware has used separate hardware to implement each stage in the pipeline with little programmability exposed to the user. The SGI InfiniteReality [Montrym et al., 1997] is a representative example, and a more recent single-chip graphics processor is the Digital Neon [McCormack et al., 1998]. Many of these machines were internally programmable through the use of microcoded computational engines (such as the LeoFloat unit, which implements the geometry stages of the pipeline in the Leo graphics system [Deering and Nelson, 1993]), but did not expose this programmability to the user. Besides the fixed function of the pipeline stages, these machines use a task-parallel machine organization as opposed to Imagine's time-multiplexed organization (discussed further in Section 7.1). More recent machines offer more programmability and are described in the next section.

Humphreys et al. describe a streaming framework for cluster rendering [Humphreys et al., 2002] in which both data and commands are treated as streams on a multi-node compute cluster. Such an approach would be applicable to networks of stream processing nodes, such as the planned 64-Imagine network.

2.4 Programmability and Graphics

Programmable graphics systems allow users to model the complexities of the natural world. One of the first efforts toward formalizing a programmable framework for graphics was Rob Cook's seminal work on shade trees [Cook, 1984], which generalized the wide variety of shading and texturing models at the time into an abstract model. He provided a set of basic operations which could be combined into shaders of arbitrary complexity. His work was the basis of today's shading languages, including Hanrahan and Lawson's 1990 paper [Hanrahan and Lawson, 1990], which in turn contributed ideas to the widely-used RenderMan shading language [Upstill, 1990], which is mostly used for offline rendering. RenderMan shaders are closely tied to the Reyes rendering pipeline; our implementation of Reyes is presented in Chapter 8.

RenderMan is primarily used for non-real-time rendering. However, programmability has also entered the real-time arena. Recently special purpose hardware has added programmability to the rendering pipeline, specifically at the vertex and fragment levels. NVIDIA's GeForce 4 and ATI's Radeon processors are representative examples of current graphics chips with vertex and fragment programmability. NVIDIA's OpenGL extensions `NV_vertex_program`, `NV_texture_shader`, and `NV_register_combiner` [NVIDIA, 2001], and Lindholm et al.'s description of vertex programs [Lindholm et al., 2001], describe some of their most recent features. The programmable features of modern graphics hardware are increasingly used for rendering and scientific tasks (such as raytracing [Purcell et al., 2002]) that are far afield from the traditional polygon rendering algorithms implemented by their non-programmable predecessors. This programmability is accessible through high-level shading languages such as the Stanford RTSL [Proudfoot et al., 2001]. The pipelines in this dissertation use the RTSL to describe their programmable shaders and lights, though Imagine can support a superset of the features provided by this programmable hardware.

Though we write separate pipelines for OpenGL and Reyes in this dissertation, either can be described in terms of the other. RenderMan is much more complex than OpenGL and can easily describe its functionality. But the converse is also true: two years ago, Peercy et al. implemented general RenderMan shaders on OpenGL hardware [Peercy et al., 2000].

They made two key contributions relevant to this work. First, they developed a framework for mapping RenderMan shaders onto multiple passes of OpenGL hardware. Second, they identified extensions to OpenGL, color range and pixel texture, which are necessary and sufficient to support RenderMan and other fully general shading languages.

Other abstractions supporting programmability in the pipeline include those of Olano and McCool. Olano extended the concept of programmability in his dissertation [Olano, 1998], which presented an abstract pipeline that was programmable at each stage and hid the details of the target machine from the programmer. Another programmable API is SMASH [McCool, 2001], which is targetted to extend modern graphics hardware. Both of these pipelines would also be implementable on Imagine using similar techniques and algorithms to those described in this dissertation.

Chapter 3

Stream Architectures

Computer graphics tasks, such as the rendering systems described in this dissertation, are part of a broad class of applications collectively termed “media applications”. These applications comprise a major part of the workload of many modern computer systems.

Typically, media applications exhibit several common characteristics:

High Computation Rate Many media applications require many billions of arithmetic operations per second to achieve real-time performance. Modern graphics processors advertise computation rates of over one trillion operations per second.

High Computation to Memory Ratio Structuring media applications as stream programs exposes their locality, allowing implementations to minimize global memory usage. Thus stream programs, including graphics applications, tend to achieve a high computation to memory ratio: most media applications perform tens to hundreds of arithmetic operations for each necessary memory reference.

Producer-Consumer Locality with Little Global Data Reuse The typical data reference pattern in media applications requires a single read and write per global data element. Little global reuse means that traditional caches are largely ineffective in these applications. Intermediate results are usually produced at the end of a computation stage and consumed at the beginning of the next stage.

Graphics applications exhibit this *producer-consumer locality*: the graphics pipeline is divided into tasks which are assembled into a feed-forward pipeline, with the output of one stage sent directly to the next. Global input data—vertex and connectivity information—is typically read once and only once. Output data exhibits little locality. Texture data, in contrast to most streaming data, does cache well.

Parallelism Media applications exhibit ample parallelism at the instruction, data, and task levels. Efficient implementations of graphics workloads certainly take advantage of the "embarrassingly parallel" nature of the graphics pipeline.

To effectively support these applications with high performance and high programmability, we have developed a hardware/software system called Imagine based on the concept of *stream processing*. *Stream programs* are structured as streams of data passing through computation kernels. Our system consists of a programming model, an architecture, and a set of software tools and is described below.

3.1 The Stream Programming Model

In the stream programming model, the data primitive is a *stream*, an ordered set of data of an arbitrary datatype. Operations in the stream programming model are expressed as operations on entire streams. These operations include stream loads and stores from memory, stream transfers over a multi-node network, and computation in the form of *kernels*.

Kernels perform computation on entire streams, usually by applying a function to each element of the stream in sequence. Kernels operate on one or more streams as inputs and produce one or more streams as outputs.

One goal of the stream model is to exploit data-level parallelism, in particular SIMD (single-instruction, multiple-data) parallelism. To do so requires simple control structures. The main control structure used in specifying a kernel is the loop. Kernels typically loop over all input elements and apply a function to each of them. Other types of loops include looping for a fixed count or looping until a condition code is set or unset. Arbitrary branches within a kernel are not supported. Imagine extends the SIMD model with its conditional streams mechanism [Kapasi et al., 2000] that allows more complex control flow within kernels. Conditional streams are described in more detail in Section 3.3.3.

A second goal of the stream model is fast kernel execution. To accomplish this goal, kernels are restricted to only operate on local data. A kernel's stream outputs are functions only of their stream inputs, and kernels may not make arbitrary memory references (pointers or global arrays). Instead, streams of arbitrary memory references are sent to the memory system as stream operations, and the streams of returned data are then input into kernels.

A stream program is constructed by chaining stream operations together. Programs expressed in this model are specified at two levels: the stream level and the kernel level. A simple stream program that transforms a series of points from one coordinate space to another, for example, might consist of three stream operations specified at the stream level: a stream load to bring the input stream of points onto the Imagine processor, a kernel to transform those points to the new coordinate system, and a stream save to put the output stream of transformed points back into Imagine's memory.

Implementing more complex stream programs requires analyzing the dataflow through the desired algorithm and from it, dividing the data in the program into streams and the computation in the program into kernels. The programmer must then develop a flow of computation that matches the algorithm.

3.1.1 Streams and Media Applications

The stream programming model is an excellent match for the needs of media applications for several reasons. First, the use of streams exposes the parallelism found in media applications. Exploiting this parallelism allows the high computation rates necessary to achieve high performance on these applications. Parallelism is exposed at three levels:

Instruction level parallelism Kernels typically perform a complex computation of tens to hundreds of operations on each element in a data stream. Many of those operations can be evaluated in parallel. In our transformation example above, for instance, the x, y, and z coordinates of each transformed point could be calculated at the same time.

Data level parallelism Kernels that operate on an entire stream of elements can operate on several elements at the same time. In our example, the transformation of each point in the stream could be calculated in parallel. Even calculations that have dependencies between adjacent elements of a stream can often be rewritten to exploit data parallelism.

Task level parallelism Multiple stream processor nodes connected by a network can easily be chained to run successive kernels in a pipeline, or alternatively, to divide the work in one kernel among several nodes. In our example, the stream of points could be divided in half and each half could be transformed on separate stream nodes. Or, if another kernel was necessary after the transformation, it could be run on a second stream node connected over a network to the stream node performing the transformation. This work partitioning between stream nodes is possible and straightforward because the stream programming model makes the communication between kernels explicit.

Next, as communication costs increasingly dominate achievable processor performance, high-performance media applications require judicious management of bandwidth. The gap between deliverable off-chip memory bandwidth and the bandwidth necessary for the computation required by these applications motivates the use of a *data bandwidth hierarchy* to bridge this gap [Rixner et al., 1998]. This hierarchy has three levels: a main memory level for large, infrequently accessed data, an intermediate level for capturing the on-chip locality of data, and a local level for temporary use during calculations. Media applications are an excellent match for this bandwidth hierarchy.

In addition, because kernels in the stream programming model are restricted to operate only on local data, kernel execution is both fast and efficient. Kernel data is always physically close to the kernel's execution units, so kernels do not suffer from a lack of data bandwidth due to remote data accesses.

Finally, kernels typically implement common tasks such as a convolution or a fast Fourier transform. A library of common kernels can be used as modular building blocks to construct new media applications.

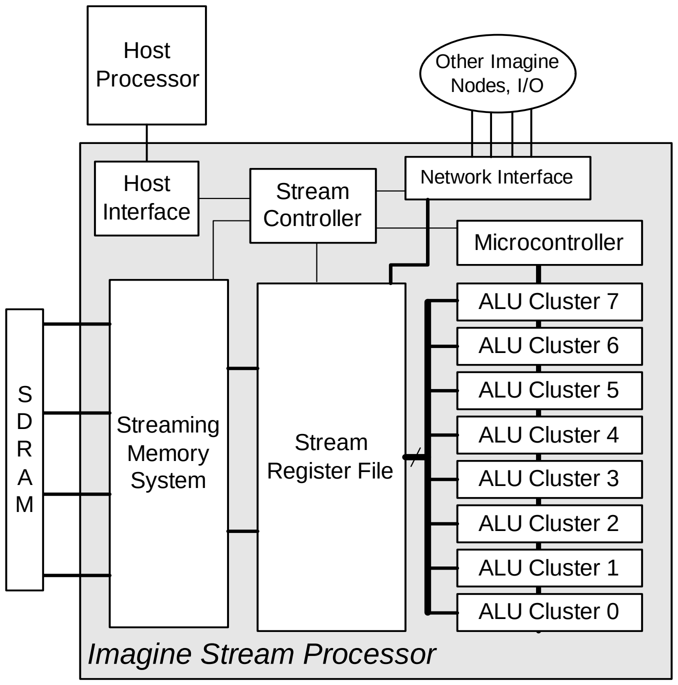


Figure 3.1: The Imagine Stream Processor block diagram.

3.2 The Imagine Stream Processor

The Imagine stream processor is a hardware architecture designed to implement the stream programming model [Khailany et al., 2001]. Imagine's block diagram is shown in Figure 3.1. Imagine is a coprocessor, working in conjunction with a host processor, with streams as its hardware primitive.

The core of Imagine is a 128 KB stream register file (SRF). The SRF is connected to 8 SIMD-controlled VLIW arithmetic clusters controlled by a microcontroller, a memory system interface to off-chip DRAM, and a network interface to connect to other nodes of a multi-Imagine system. All modules are controlled by an on-chip stream controller under the direction of an external host processor.

The working set of streams is located in the SRF. Stream loads and stores occur between the memory system and the SRF; network sends and receives occur between the network interface and the SRF. The SRF also provides the stream inputs to kernels and stores their

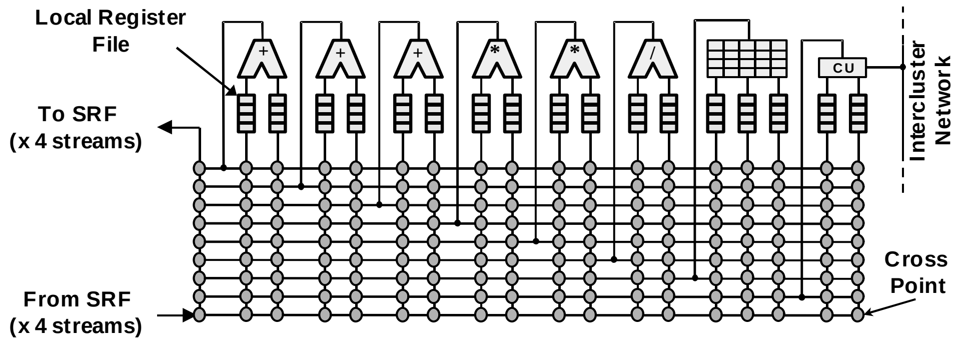


Figure 3.2: Cluster Organization of the Imagine Stream Processor. Each of the 8 arithmetic clusters contains this organization, controlled by a single microcontroller and connected through the intercluster communication unit.

stream outputs.

The kernels are executed in the 8 arithmetic clusters. Each cluster contains several functional units (which can exploit instruction-level parallelism) fed by distributed local register files. The 8 clusters (which can exploit data-level parallelism) are controlled by the microcontroller, which supplies the same instruction stream to each cluster.

Each of Imagine's eight clusters, shown in Figure 3.2, 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. 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.

The three-level memory bandwidth hierarchy characteristic of media application behavior consists of the memory system (2.62 GB/s), the SRF (32 GB/s), and the local register files within the clusters (544 GB/s).

On Imagine, streams are implemented as contiguous blocks of memory in the SRF or in off-chip memory. Kernels are implemented as programs run on the arithmetic clusters. Kernel microcode is stored in the microcontroller. An Imagine application consists of a chain of kernels that process one or more streams. The kernels are run one at a time, processing their input streams and producing output streams. After a kernel finishes, its output is typically input into the next kernel.

3.3 Imagine's Programming System

3.3.1 StreamC and KernelC

The stream programming model specifies programs at two levels, stream and kernel. At the stream level, the programmer specifies program structure: the sequence of kernels that comprise the application, how those kernels are connected together, and the names and sizes of the streams that flow between them.

Stream programs are programmed in C++ and use stream calls through a stream library called StreamC. StreamC programs usually begin with a series of stream declarations then a series of kernels (treated as function calls) that take streams as arguments. StreamC programs are scheduled using a stream scheduler [Kapasi et al., 2001; Mattson, 2002].

StreamC abstracts away the distinction between external memory and the SRF and makes all allocation decisions and handles all transfers for both. The system used in this dissertation first runs the stream programs to extract profile information of application behavior then uses the profile to make allocation decisions. However, a system under development removes the profiling requirement and makes allocation decisions based on compile-time information alone.

The kernel level specifies the function of each kernel. Kernels are written in KernelC, a subset of C++. KernelC lacks most control constructs of C++ (such as `if` and `for` statements), instead supporting a variety of loop constructs and a `select` operator (analogous to C's `?:`). It also does not support subroutine calls. Stream inputs and outputs are specified using the >> and << operators, respectively. KernelC programs are compiled using a kernel scheduler [Mattson et al., 2000] that maps the kernels onto the Imagine clusters, producing microcode that is loaded during program execution into the microcontroller's program store.

3.3.2 Optimizations

Inputs to graphics pipelines—typically lists of vertices—are usually many thousands of elements long. In our system, these inputs are divided into "batches" so that only a subset of them are processed at one time. This common vector technique is called "stripmining" [Lamport, 1974; Loveman, 1977]; it allows the hardware to best take advantage of the producer-consumer locality in the application by keeping the working set of data small enough to fit in the SRF.

The second optimization we perform is used at both the kernel and stream levels. This optimization is applied to loops and is called "software pipelining". Software pipelining raises the instruction-level parallelism in a loop by scheduling more than one loop iteration at the same time. For example, the second half of one loop iteration can be scheduled at the same time as the first half of the next loop iteration.

Software pipelining has a cost: because more than one iteration is run at a time, the loop must run more times overall than if the loop was not software pipelined. The extra cost is due to the need to prime and drain the loop to account for the overlap in execution.

However, software-pipelining a loop often makes the loop faster because the critical path for the loop is reduced. Software pipelining is effective when the loop is run many times because the reduction in critical path saves more time than the cost of the priming and draining.

Software pipelining can be applied to both kernel loops and stream loops. All kernels in this work are software pipelined, as are all stream programs. Software-pipelining stream programs has an additional advantage in that kernel execution for one iteration overlaps memory traffic for the other iteration, allowing latency tolerance for the results of the memory operation.

3.3.3 Conditional Streams

Imagine's 8 SIMD-controlled clusters are ideal for processing long streams of independent data elements. If the same function is to be applied to each element in the stream, and no element in the stream is dependent on another element, then the elements can be evenly divided among the clusters with no necessary inter-cluster communication. However, this simple model is not true in general for stream programs.

The conditional streams mechanism [Kapasi et al., 2000] addresses this problem by introducing two new primitives: conditional input and conditional output. With a conditional input operation, a new data element is fetched from the input stream into a cluster only if a local condition code corresponding to that data element is true. Conditional outputs work in a similar fashion: they append data elements to an output stream only if a local condition code corresponding to that data element is true. These condition codes are typically calculated along with the data elements. The conditional stream primitives are conceptually simple for the programmer but are powerful enough to handle complex data-dependent behavior such as the examples below:

switch This operation is an analog to the C *case* statement. Of n possible types of data elements, each type must be processed differently. The *switch* operator sends a data element to one (or none) of n different output streams. The *switch* operator ensures that all elements in the same output stream are of the same type so that the processing for all elements in the same output stream will be identical. Thus, these streams can be processed efficiently by a processor without the need for a conditional that is evaluated based on the type of the data element.

In the rendering pipeline discussed in Chapter 4, for instance, *switch* is used to separate the fragments with conflicting and unique addresses within a stream and also to cull triangles that face away from the camera.

combine The converse of *switch*, a *combine* operation merges several input streams into a single output stream. In our pipeline, the sort stage merges two sorted streams of fragments into a single sorted stream.

load-balance Each data element in an input stream may require a different amount of computation. The *load-balance* mechanism allows each cluster to receive a new input to process as soon as it is finished processing its previous element. In our pipeline, our triangle rasterization implementation ensures that clusters processing triangles of different size do not have to wait for all clusters to be complete before starting the next triangle.

Traditional mechanisms for implementing these conditional operations on parallel machines result in inefficient use of the processing elements (in this case the clusters). The key to implementing conditional streams on Imagine is to perform the necessary data routing in the clusters by leveraging the inter-cluster communication switch already present in the clusters. The conditional stream mechanism can be implemented either fully in software or accelerated with minimal hardware support. Kapasi et al. show that one scene in our pipeline (a scene similar to ADVS-1) ran 1.8 times faster on a conditional stream architecture than on an equivalent traditional data-parallel architecture.

Chapter 4

The Rendering Pipeline

A renderer produces an image from the description of a scene. Rendering is a computationally intense, complex process and in recent years has been an integral part of workstation and consumer level computer hardware.

In this section, we describe the design and implementation of a rendering pipeline. Our design aims for flexible, high-performance rendering on programmable stream architectures such as Imagine.

Trends The performance of special-purpose rendering hardware has increased markedly in recent years. This increase in performance has resulted in several trends:

- Input primitives are increasing in detail. Because we have more performance with each succeeding generation of graphics hardware, we can make our input models more detailed, with finer geometric resolution.

  The primary geometric primitive is a triangle, and as a result of this trend, screen-space triangles cover an ever-smaller number of pixels.

- More information is used to render each primitive. Instead of using a single color or a single texture address to render a primitive, for example, we now often carry several colors or texture addresses, or a combination of both. Consequently, the amount of processing per primitive is increasing.

- With more information available per primitive, we would like to do more flexible operations on primitives and on their generated fragments. It is this desire which has led to shading languages such as the Stanford Real-Time Shading Language (described in Section 4.1).

Design goals We aimed to create and develop algorithms that:

- Elegantly fit into the stream programming model;
- Make efficient and high-performance use of stream programming hardware;
- Have similar pin bandwidth requirements to today's commercial graphics processors;
- Work with the limitations of our hardware and programming tools;
- Use a programming abstraction similar to established APIs.

Our system has a similar interface to the OpenGL API. Both our system and OpenGL offer immediate mode semantics, carry state, and respect the ordering of input primitives. In a parallel architecture such as ours, the ordering requirement in particular requires special care and will be discussed in later sections.

4.1 Stanford RTSL

As the amount of programmability in graphics hardware increases, the task of programming graphics hardware using existing APIs becomes more difficult. Moreover, current APIs are hardware-centric and do not allow much portability between hardware platforms. Shading languages solve the ease-of-use and portability problems by providing a higher level of abstraction. Languages are inherently easier to use than lower-level APIs, and the process of compilation allows computations to be mapped to multiple hardware platforms.

The Stanford real-time programmable shading system is used to specify programmable lights and shaders, targetting the current generation of programmable graphics hardware [Proudfoot et al., 2001]. This system has several major components: a *hardware abstraction* appropriate for programmable graphics hardware, a *shading language* for describing

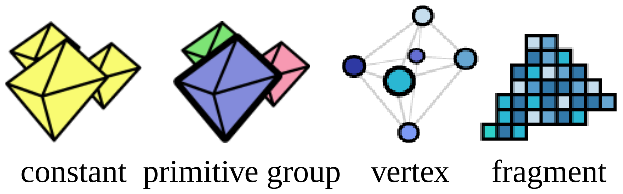


Figure 4.1: RTSL computation frequencies. The Stanford shading system supports four computation frequencies. In the illustrations above, individual elements at each computation frequency are depicted by shade.

shading computations, a *compiler front end* for mapping the language to the hardware abstraction, a *modular retargetable compiler back end* for mapping the abstraction to hardware, and a *shader execution engine* for managing the rendering process.

The shading system organizes its hardware abstraction around multiple *computation frequencies*, illustrated in Figure 4.1. Computation frequencies reflect the natural places to perform programmable computations in the polygon rendering pipeline. In particular, programmable computations may be performed once per primitive group, once per vertex, or once per fragment, where a primitive group is defined as the set of primitives specified by one or more OpenGL Begin/End pairs, a vertex is defined by the OpenGL Vertex command, and a fragment is defined by the screen-space sampling grid.

The shading language provided by the Stanford system is loosely based on RenderMan. Many of the differences between the two languages are accounted for by the features and limitations of real-time hardware. In particular, the language omits support for data-dependent loops and branches and random read/write access to memory, since hardware pipelines do not support these features, and the language includes a number of data types (such as clamped floats) specific to graphics hardware. Like RenderMan, the language provides support for separate surface and light shaders. The language also has features to support the management of computation frequencies.

The shading system used in this paper is a modified version of the Stanford shading system. In particular, we made two changes to the original system:

Removed computation frequency restrictions The Stanford system restricts many operations to a subset of the available computation frequencies to account for limited hardware capabilities. For example, the system does not allow per-fragment divides, per-vertex textures, or fully-general dependent texturing. Since our hardware is flexible enough to support all operations at all computation frequencies, our shading system omits almost all of the computation frequency restrictions present in the original system.

Unlike the original Stanford system, the Imagine backend also supports floating-point computation throughout the pipeline, including the fragment program.

New backend for Imagine We added a new backend to the Stanford shading system to support compilation to our hardware architecture. This backend inputs a parsed representation of the lights and shaders and generates Imagine stream and kernel code.

Because global memory accesses (such as texture lookups) in the stream programming model require stream loads from memory outside of the kernel, the Imagine backend also must split computation frequencies with global memory accesses into multiple kernels with intermediate stream loads. Section 4.4 describes this operation in more detail.

4.2 Pipeline Organization

In designing our pipeline, first we define its inputs and outputs. The inputs to the pipeline are a stream of vertices with connectivity information, a stream consisting of all texture data, and a stream of parameters which apply to the whole input stream (such as transformation matrices, light and material information, and so on).

The vertex and connectivity inputs are stripmined into “batches” so that only a subset of them are processed at one time, as described in Section 3.3.2. The stream scheduler, based on triangle size estimates, suggests an efficient batch size. Batches which are too small suffer from short-stream effects (see Section 5.2.3), while batches which are too large overflow the SRF (requiring spills to memory) and degrade performance. Batches are sized so that in the common case all intermediate streams used in processing a batch will fit into the SRF; this issue is discussed further in Section 4.6.

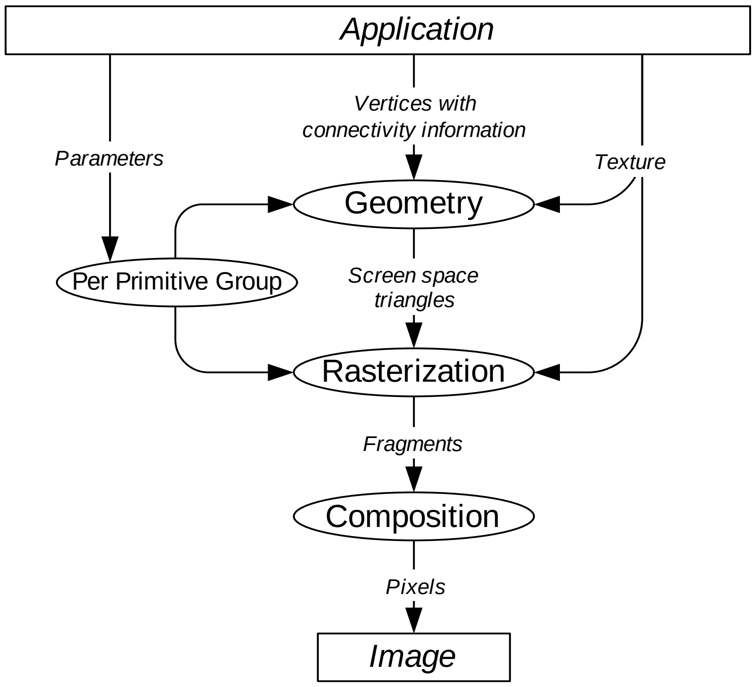


Figure 4.2: Top-Level View of the Rendering Pipeline.

The pipeline's output is an image stored in the color buffer. The color buffer, depth buffer, and all texture maps are stored in Imagine's main memory and are each treated as a single linear stream. To access one of them, we use a stream load of an *index stream* of indexes into the linear buffer, which returns a data stream. To do a depth buffer read, for instance, we generate a stream of depth buffer addresses and send it to the memory system, which returns a stream of depths from the depth buffer. Writes into these buffers have two input streams, an index stream and a data stream.[1]

Next, the pipeline is mapped to streams and kernels to run on a stream architecture such as Imagine. This organization is described below: Section 4.2.1 evaluates the per-primitive-group work, and the geometry, rasterization, and composition steps of the pipeline are detailed in Sections 4.3, 4.4, and 4.5. The top-level view of the pipeline is illustrated in Figure 4.2.

[1]Stream loads, such as depth buffer reads, are vector *gather* operations; stream saves, such as color buffer writes, are vector *scatter* operations. Both are supported in the stream programming model.

4.2.1 Primitive Group Work

Section 4.1 described the RTSL's four computation frequencies. The second computation frequency is "per primitive group" and we call its resulting program the `perbegin` kernel. All shading work that can be performed once per primitive group instead of on each vertex or on each fragment is executed in this kernel.

Consequently, even if the geometry data is complex enough to be stripmined into several batches, this kernel only needs to be run once. Its input usually includes transformation matrices, lighting parameters, and texture identifiers. Its output is divided into several output streams, each of which is an input to a vertex or fragment program kernel.

Unlike the vertex and fragment programs, the `perbegin` kernel does not loop over a large set of primitives (vertices or fragments). It has no data parallelism to exploit in Imagine's SIMD clusters, so as a result, when it is run, it is run redundantly in parallel across all 8 clusters. Because it is executed only once per primitive group, the resulting loss of efficiency is negligible.

4.3 Geometry

The first stage of the rendering pipeline is the geometry stage, which transforms object-space triangles into screen-space and typically performs per-vertex lighting as well as other per-vertex or per-triangle calculations. The operations in this stage are primarily on floating-point values and usually exhibit excellent data parallelism.

Our geometry stage consists of five steps: the vertex program, primitive assembly, clip, viewport, and backface cull. The geometry kernels in our implementation are shown in Figure 4.3.

4.3.1 Vertex Program

The real work in programmable shading is done at the vertex and fragment frequencies. The third computation frequency is "per vertex" and in the resulting vertex kernel, we run a program on each vertex.

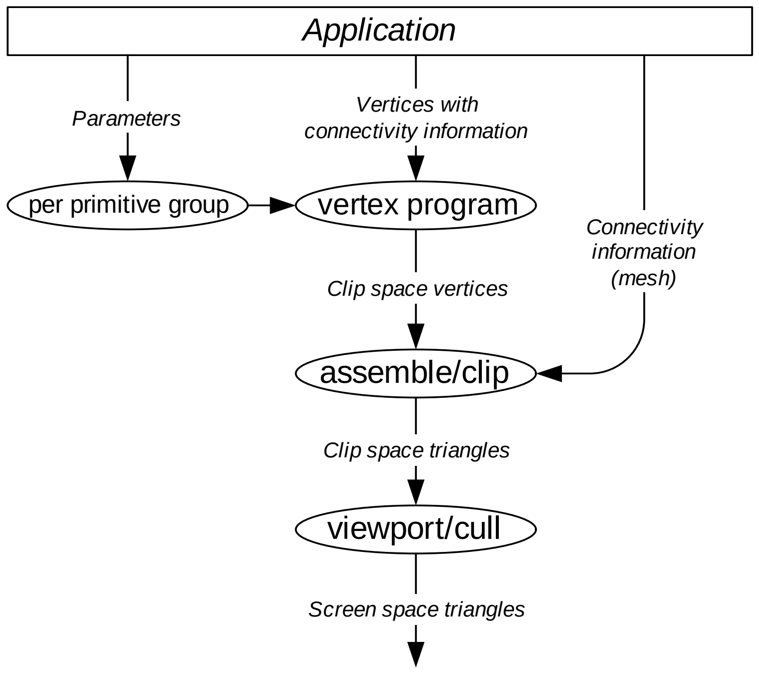


Figure 4.3: Geometry Stage of the Rendering Pipeline.

The main input to this kernel is a stream of vertices; usually each vertex carries its object-space position, normal, and texture coordinate information, and may also contain color, tangent, binormal, or other arbitrary per-vertex information. In practice, the RTSL compiler specifies which information is required for each vertex and we simply specify that information as part of each vertex in the stream. This information is almost always specified in 32-bit floating-point. The vertex program also takes the output of the perbegin kernel as a second input.

When the kernel is run, it first streams in the perbegin information. Then it loops over its stream of input vertices. Because the number of vertices is large (usually on the order of many tens to hundreds), and because the same program must be run on each vertex with no intervertex dependencies, the vertices can be processed in parallel. Imagine's eight SIMD-controlled clusters evaluate the vertex program on eight vertices at the same time.

The vertex program encompasses the traditional OpenGL stages of modelview transform, the application of the GL shading model, and the projection transform. Because it is programmable, however, it can do anything the programmer desires, including more complex lighting models, vertex displacements, vertex textures, and so on. The vertex program is similar in structure to the fragment program, which is described in more detail in Section 4.4.4.

The kernel's output is a stream of transformed vertices. Each vertex includes its position in clip coordinates and an arbitrary number of floating-point interpolants. Typical interpolants include one or more colors and one or more texture coordinates. Each interpolant is then perspective-correct interpolated across the triangle during the rasterization stage of the pipeline.

4.3.2 Primitive Assembly

After vertex processing is complete, the vertices must be assembled into triangles. (This step is skipped if the input vertices are specified in separate triangles.) The most common types of input primitives are triangle meshes and polygons, and to assemble them, we use the `assemble_mesh` and `assemble_poly` kernels.

On each loop, each cluster inputs a vertex and mesh information for that vertex. For the purposes of assembly, each vertex is assumed to be the third vertex in a triangle. The mesh information is encoded (in a single 32-bit word) as two 16-bit integer offsets. The first offset specifies how many vertices must be "counted back" to find the first vertex of the triangle, and the second offset does the same for the second vertex. If either offset is zero, no valid triangle is produced. Only valid triangles are conditionally output into the output stream.

If both offsets are positive, then each cluster fetches its other two vertices from the other clusters via Imagine's intercluster communication mechanism. If the current cluster minus the offset is a negative number (meaning the vertex in question was not input on this cycle), then it is retrieved from a stored vertex kept as state from the previous loop.

For polygons, two vertices must be kept as state: the root of the polygon and the last-seen vertex. For meshes, two vertices are also kept as state, but instead the last-seen and second-to-last-seen vertices are kept.

4.3.3 Clipping

Once the triangles are assembled, they must be clipped. Originally we did a general clip at this point, clipping all triangles to the view frustum. This operation was quite complex on a SIMD machine, put triangles out of order, and presented severe allocation problems for stream-level allocation.

Instead, we do a trivial clip with a `clip` kernel, discarding all triangles which fall fully outside of the view frustum. The remainder of the clip is done during rasterization using homogeneous coordinates [Olano and Greer, 1997].

Assembly and clipping are combined into a single kernel in our implementation.

4.3.4 Viewport

The viewport transform is simple: on each vertex of each input triangle, perform the perspective divide by the homogeneous coordinate w on the x, y, and z values, and apply the viewport transformation from clip coordinates to screen-space coordinates.

With the scanline rasterizer (described in Section 4.4.1), each interpolant is also divided by w to ensure perspective correctness. The barycentric rasterizer (described in Section 4.4.2) does not require this divide.

4.3.5 Backface Cull

Finally, each triangle is tested for backface culling. Triangles which face toward the camera are conditionally output in this kernel, and triangles which have zero area or face away are discarded.

This kernel is completely bound by cluster input-output bandwidth, so (with no increase in kernel loop length), to make the job of the rasterizer slightly easier, we also test to see if the triangle crosses both a x and y scanline. If it does not, it produces no pixels, and we can safely discard it as well.

The viewport transformation and the backface cull are combined into a single kernel in our implementation.

At this point, all triangles are in screen-space coordinates and in input order, and are prepared for the rasterization stage.

4.4 Rasterization

The rasterization stage of the polygon rendering pipeline converts screen-space triangles to fragments. Per-vertex values, such as texture coordinates or a color, must be interpolated across the triangle during this stage.

Rasterization is a difficult task for SIMD architectures because each individual triangle can (and does) produce different numbers of fragment outputs. This behavior makes the design of an efficient rasterizer, and particularly its stream allocation, quite complex.

A second complication is ordering. The implementations described in this section do not output their fragments in strict triangle order, which violates our ordering constraint. The composite stage, described in Section 4.5, must rectify the situation, and the rasterizer must provide the information necessary to do so.

Our rasterization stage has two main parts. The first is the rasterizer itself; it finds the pixel locations for each fragment and calculates the interpolants. The second is the fragment program generated by the RTSL compiler. It runs an arbitrary program on each fragment to produce a final color per fragment, which then enters the composite stage.

Sections 4.4.1 and 4.4.2 describe two rasterization algorithms for stream architectures. Section 4.4.4 describes the fragment program.

4.4.1 Scanline Rasterization

The traditional method to rasterize screen-space triangles is the scanline method. In this method, triangles are first divided into *spans*, which are one-pixel-high strips running from the left side to the right side of the triangle.

Typically, a scanline rasterizer begins at the bottom of the triangle and walks up the left and right edges of the triangle, emitting a span each time it crosses a scanline. The scanline algorithm is well-suited to special-purpose hardware implementations because it can use incremental computation in calculating the linear interpolants.

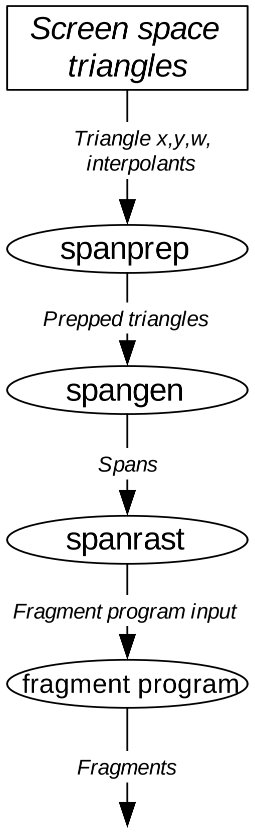


Figure 4.4: Scanline Rasterization in the Rendering Pipeline.

Stream Implementation

A scanline rasterizer maps logically into three kernels: a triangle setup kernel that does all per-triangle prep work; a span generation kernel that produces spans from prepped triangles; and a scan rasterization kernel that rasterizes spans into fragments. In our system, these three kernels are called `spanprep`, `spangen`, and `spanrast` and are shown in Figure 4.4.

spanprep The `spanprep` kernel inputs screen-space triangles and conditionally outputs zero, one, or two valid screen-space half-triangles. Half-triangles are triangles that always have one edge parallel to the x-axis. Using half-triangles allows us to keep track of only two edges at a time instead of three and eases the control complexity without increasing the number of generated spans.

`spanprep` first sorts the triangle's vertices by their y coordinates and assigns an ID to the triangle (which is later passed to all its fragments) for future ordering steps. If mipmapping is enabled, it then calculates the per-triangle coefficients used to generate the screen-space derivatives of the texture coordinates. Next it constructs the left-to-right half-triangle edges, calculates their start and end y-coordinates, and calculates the beginning and delta values for each interpolant at the bottom of each edge. Finally, it conditionally outputs only those half-triangles that span at least one y scanline.

spangen The `spangen` kernel conditionally inputs the prepped half-triangles from the previous stage, then conditionally outputs spans for use in the next span rasterizer kernel.

First, each cluster that has completed its current triangle brings in a new triangle. Clusters that are not yet done keep their current triangles. Each cluster keeps track of its own current scanline, and the kernel next calculates a span on its current scanline. A span consists of integer start and end x-coordinates and for each interpolant, a floating-point start value (at the left side of the span) and the floating-point incremental delta value to be added as the span is traversed in the next kernel. Mipmap calculations are also performed at this point if necessary. At this point, the ordering constraint is violated, as spans are no longer in triangle order.

After the span is calculated, the per-edge interpolant deltas are added for the left and right edges. The span is then output if it will generate at least one valid pixel.

The loop can be unrolled easily; choosing the amount of unrolling involves balancing the added efficiency from processing multiple spans per loop, the loss of efficiency when the number of spans do not divide evenly by the number of spans per loop, and the extra register pressure from processing multiple elements in the same loop iteration. In practice, we usually generate two valid spans per loop.

spanrast The task of generating actual pixels falls to the `spanrast` kernel. `spanrast` loops over its input spans, bringing in a new span per cluster when its current span is exhausted. It conditionally outputs valid fragments with framebuffer address, triangle ID, depth, and interpolant information.

`spanrast` is the simplest of the three rasterization kernels. First, each cluster that has completed its current span conditionally inputs a new span. Clusters that are not yet done keep their current spans. Next it uses the span information to calculate the current fragment's depth, interpolant values, and framebuffer address. If mipmapping is enabled for this fragment, it also computes the mipmap level. It then increments the span for the next fragment. Like `spangen`'s inner loop, its inner loop can also be unrolled, with the number of fragments per unrolled loop determined by similar considerations as in `spangen`; two generated fragments per loop is our typical size.

Advantages of the Stream Implementation

Traditionally, scanline algorithms have been the preferred method of hardware rasterizers. Mapping a scanline algorithm into streams and kernels has several advantages:

Factorization of problem size A renderer on a stream-programmed system would be most efficient if every triangle was the same size and shape. As this is an undue burden for the graphics programmer, our system must handle triangles of varying size. A triangle with many interpolants carries with it a large amount of data, and we must find an efficient way to distill that data to each of its constituent fragments. In effect, we must construct a data funnel to take a complex triangle and map it to simpler, more numerous fragments.

Our experience in attempting to do this operation in one step, from triangles to fragments, has demonstrated that such an approach is difficult. Triangle traversal did not parallelize well, and the huge amount of computation necessary in a one-step algorithm overwhelmed our tools.

Using the intermediate step of spans helps manage this complexity. Roughly speaking, a triangle has two dimensions as it stretches in both the x and y directions. A point has zero dimensions. Using spans provides an intermediate step with one dimension. As we move from triangles to spans to fragments, our dimension decreases, and while the number of primitives increases, the amount of data associated with each primitive decreases.

Efficient computation Interpolation is a linear operation and scanline algorithms exploit this linearity. For example, in span rasterization, all the per-span work is done in `spangen`, including preparing all spans with incremental delta values so that in `spanrast`, moving from pixel to pixel only involves adding the delta to each interpolant. The triangle-to-spans conversion works in the same way.

In addition, triangle traversal is made much simpler by the scanline factorization, so we do little rasterization work which does not contribute to covered pixels.

Problems with the Stream Implementation

Small triangle overhead Factoring so much of the work into setup operations for triangles and spans is expensive for small triangles. If a triangle only covers a single pixel, it is useless to prepare deltas for edges and spans because those deltas will never be used.

Span overhead The intermediate step of spans is not without its costs. Spans are large: each interpolant needs two words of space in the span, one for its initial value, one for its delta. And because spans are conditionally output in `spangen` and conditionally input in `spanrast`, a stream architecture such as Imagine, with only limited conditional stream bandwidth into and out of the cluster, suffers from a bandwidth bottleneck on span input and output.

Mipmaps and derivatives difficult Adding texture mipmap derivative calculations made the size of the prepped triangles and spans much larger. This reduces the number of vertices we can handle per batch, making computation less efficient.

For instance, the ADVS scene, when point-sampled, allows 256 vertices per batch; when mipmapped, only 24 [Owens et al., 2000]. In addition, extracting the necessary information for arbitrary derivatives from the scanline rasterizer would be quite difficult.

Large numbers of interpolants Finally, triangle and span sizes rapidly overwhelm the Imagine scheduler and hardware as the number of interpolants becomes large.

In the original version of PIN, for instance, each vertex carried 30 floating-point interpolants (five texture coordinates with four components per coordinate, two colors with four components per color, and a front and back floating-point value). Imagine ran into a hardware limit in the `spangen` kernel, in which both triangles and spans are conditional inputs/outputs.

Our conditional stream operations require 2 entries per conditional word in the 256-entry scratchpad register file. Each half-triangle has two edges, and each edge a initial and delta value per interpolant. Each span has an initial and delta value per interpolant. Such a kernel would be difficult to schedule using the kernel scheduler, but even if we could, we require $(4 + 2) \times 30 \times 2 = 360$ entries in the scratchpad, which exceeds the limits of the Imagine hardware.

We must consider another method which avoids the difficulties above, in particular the problems associated with the large number of interpolants. Our goal is an algorithm in which the parts of the pipeline that scale with the number of interpolants are easily separable into multiple parts with little to no loss of efficiency.

4.4.2 Barycentric Rasterization

To address the difficulties inherent in scanline rasterization, we moved to another algorithm: barycentric rasterization. While the end result of a barycentric rasterizer is identical to that of a scanline rasterizer, the intermediate calculations are structured differently. We analyze the performance of the two algorithms in Section 5.2.6.

The key idea behind our implementation of the barycentric rasterizer is our separation between *pixel coverage* and *interpolation*. Pixel coverage refers to the calculation of which pixel locations are covered by a given triangle. Interpolation is the calculation of the value of each interpolant at a given pixel location given the values of the interpolant at each vertex.

In a scanline rasterizer, these two processes are intimately tied together. Processing a triangle means processing each of its interpolants at the same time, leading to the problems described above in Section 4.4.1.

In contrast, a barycentric rasterizer first calculates pixel coverage information. The interpolants do not figure into this calculation. It is only after all pixel coverage is complete and the barycentric coordinates for each pixel coordinate are calculated that the interpolants are considered. Moreover, the interpolant rasterization kernel is separable: if the number of interpolants overwhelms the ability of our hardware or tools to handle them, the kernel can be easily and cheaply split into multiple smaller kernels, each of which calculates a subset of the interpolants.

The following explanation of barycentric rasterization follows Russ Brown's treatment [Brown, 1999].

We can describe the value of any interpolant within a triangle at screen location p as the weighted sum of its value at the three vertices:

$$i_p = b_0 i_0 + b_1 i_1 + b_2 i_2 \tag{4.1}$$

The weights b_0 , b_1, and b_2 must add to 1. Geometrically, the weight b_0 for vertex v_0 at screen location $p = (x, y)$ is the area of the triangle formed by p, v_1, and v_2 (A_0) over the area of the triangle formed by v_0, v_1, and v_2 (A):

$$\begin{aligned}
b_0 &= \tfrac{\triangle p v_1 v_2}{\triangle v_0 v_1 v_2} &= \frac{A_0}{A_0 + A_1 + A_2}; \\
b_1 &= \tfrac{\triangle p v_0 v_2}{\triangle v_0 v_1 v_2} &= \frac{A_1}{A_0 + A_1 + A_2}; \\
b_2 &= \tfrac{\triangle p v_0 v_1}{\triangle v_0 v_1 v_2} &= \frac{A_2}{A_0 + A_1 + A_2}.
\end{aligned} \tag{4.2}$$

A_n, in turn, is a linear function of x and y:

$$A_n = \alpha_n x + \beta_n y + \gamma_n. \tag{4.3}$$

Adding correction for perspective distortion to Equation 4.2 is straightforward:

$$\begin{aligned}
b_0 &= \frac{w_1 w_2 A_0}{w_1 w_2 A_0 + w_2 w_0 A_1 + w_0 w_1 A_2}; \\
b_1 &= \frac{w_2 w_0 A_1}{w_1 w_2 A_0 + w_2 w_0 A_1 + w_0 w_1 A_2}; \\
b_2 &= \frac{w_0 w_1 A_2}{w_1 w_2 A_0 + w_2 w_0 A_1 + w_0 w_1 A_2}.
\end{aligned} \tag{4.4}$$

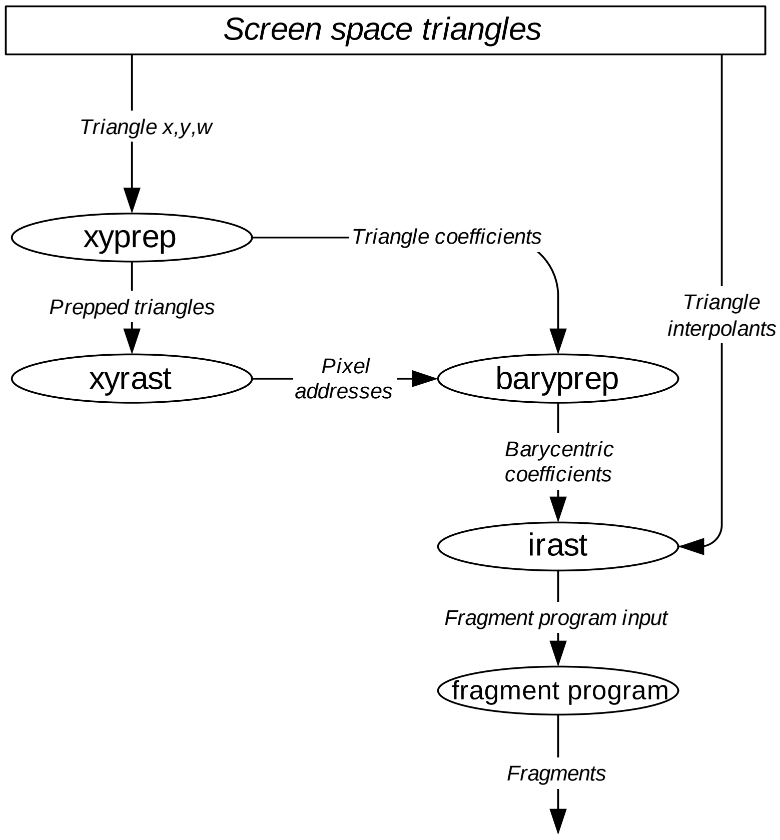


Figure 4.5: Barycentric Rasterization in the Rendering Pipeline.

Stream Implementation

To map a barycentric rasterizer into the stream model, we first divide the work into per-triangle and per-fragment work. The kernel flow is shown in Figure 4.5.

xyprep The per-triangle work is encapsulated in the `xyprep` kernel. Three tasks are necessary in this kernel:

- First we calculate the α, β, and γ coefficients for each A in Equation 4.3 and the premultiplied $w_m w_n$ values needed in Equation 4.4.
- Next we set up the triangle for the `xyrast` kernel, which calculates pixel coverage for the triangle.

- Finally, each triangle receives an ID for use in later ordering steps. This ID is passed to all its fragments.

xyrast Next the triangle must be rasterized into pixels. In the `xyrast` kernel, we conditionally input a setup triangle and on each loop, unconditionally output a fragment covered by that triangle. Along with the x and y pixel coordinate, each output fragment is marked with flags indicating whether that fragment is valid and whether that fragment is the first of its triangle. Fragments are invalid if they fall outside the triangle or if they fall outside the viewport.

Another way to implement rasterization would be to conditionally output only valid fragments. This would compress the output stream with no degradation in performance. However, this would not place the output fragments in a useable order for future steps. No matter how we do the rasterization, the OpenGL ordering constraint is violated, as fragments are no longer in triangle order. But by using unconditional output, the output stream of fragments obeys two important constraints used in later stages:

- All fragments from any given triangle are output by the same cluster. Because future rasterization kernels have unconditional inputs, all the rasterization work for any given triangle will be on the same cluster.

- The first fragments from each triangle are ordered in triangle order. Since those fragments are marked with the "first" triangle flag, this flag can be used to conditionally input corresponding per-triangle information in future kernels.

baryprep The next kernel, `baryprep`, calculates the screen-space areas A_n and the perspective-corrected barycentric coordinates b_n. The clusters unconditionally input a fragment on each loop, and if the fragment is the first of its triangle, conditionally input the triangle setup information calculated by `xyprep`. (It is because of this use of data from two frequencies—the per-triangle frequency and the per-fragment frequency—that the `xyrast` kernel must follow the two ordering constraints above.) On each loop, per fragment, `baryprep` outputs the perspective-corrected barycentric coordinates assocated with the pixel location and triangle.

The three previous kernels, `xyprep`, `xyrast`, and `baryprep`, are all independent of the number or type of interpolants and hence can be shared across all pipelines.

irast The pipeline-specific interpolation is contained in the `irast` kernel. For each interpolant, `irast` simply evaluates the barycentric interpolation of Equation 4.1. If the number of interpolants is too large to be supported by our tools or hardware, `irast` is split into multiple kernels, each of which interpolates a subset of the interpolants.

The `irast` kernel compresses the stream of valid and invalid fragments into a stream of valid fragments by conditionally outputting only the valid fragments.

Extension to mipmapping and derivatives

Producing smooth, non-aliased images requires careful consideration of the problems of filtering, sampling, and reconstruction during the image synthesis process. The process of texture mapping is fundamental to modern graphics pipelines and illustrates the challenges of achieving proper filtering, sampling, and reconstruction.

Applying a texture map to a surface involves sampling the texture map for each pixel location in screen space on the surface. If the texture map's spatial frequency exceeds the Nyquist limit for the screen space pixels, the texture-mapped surface will exhibit aliasing artifacts.

To eliminate this problem, we can prefilter the texture map at a number of lower frequencies ("levels"), then choose the appropriate filtered texture. This approach is called mipmapping [Williams, 1983] and requires accessing 8 texels for each fragment. To choose the appropriate level of the mipmap requires knowledge of the rate of change of texel coordinates with respect to screen coordinates, or in short, a derivative.

With a barycentric rasterizer, obtaining derivatives of interpolants with respect to screen space involves little additional work. Let us call the interpolant of interest u and the screen space distance x. We would thus like to compute $\partial u/\partial x$. u is a function of b_0, b_1, or b_2 (or equivalently, A_0, A_1, and A_2), which are in turn functions of x. By the chain rule,

$$\frac{\partial u}{\partial x} = \sum_{n=0}^{2} \frac{\partial u}{\partial b_n}\frac{\partial b_n}{\partial x} = \sum_{n=0}^{2} \frac{\partial u}{\partial A_n}\frac{\partial A_n}{\partial x}.$$

The three $\partial u/\partial b_n$'s are already computed in Equation 4.1, and the $\partial b_n/\partial x$'s require only a small amount of extra computation (to compute both $\partial u/\partial x$ and $\partial u/\partial y$ together requires an additional 4 adds, 3 subtracts, and 8 multiplies per fragment).

The mipmap calculation requires 4 partial derivatives ($\partial u/\partial x$, $\partial v/\partial x$, $\partial u/\partial y$, and $\partial u/\partial y$). The interpolated value calculated in `irast` might not be the final texture coordinate; the fragment program (described below in Section 4.4.4) could do additional computation on the interpolated value to produce a texture address. If so, we must calculate the partial derivatives for each interpolant that contributes to the texture address. We use the forward mode of automatic differentiation described by Nocedal and Wright [Nocedal and Wright, 1999]; with it, we calculate the partial derivatives of each interpolant and use the chain rule to combine these partial derivatives with their interpolant values to produce the derivative used by the mipmap hardware.

For example, an interpolated texture address ($t' = \{u, v, q\}$) produced by `irast` could then enter the fragment program and there be multiplied by a matrix. The resulting texture address would then be used for the texture lookup. For each fragment, we would do the matrix multiplication ($t = Mt'$) to find the texture address and use the chain rule ($\partial t = M\partial t' \cdot t'\partial M$) to find its derivative. This computation is automatically generated by the RTSL compiler at compile time.

In today's graphics chips, mipmapping hardware typically performs the derivative computation, hiding it from the user. Derivatives are useful in a wide variety of filtering techniques, however. RenderMan, for instance, provides two built-in variables and two functions that aid in computing and using derivatives [Apodaca and Gritz, 2000]. du and dv reflect the change in u and v between adjacent samples; `Du`(x) and `Dv`(x) provide the derivative of x with respect to u and v.

4.4.3 Other Rasterization Alternatives

Handling derivatives has traditionally been done in a different manner, with multi-pixel stamps. A stamp usually covers a 2 $\times$ 2 square of pixels. Derivative information is extracted from the differences between adjacent pixels.

Such an approach is general and amenable to hardware acceleration; we chose not to use it for two reasons. First, it is not as accurate as implicit differentiation, providing a coarser approximation only valid to the first term of the expansion. Second, rasterizing four pixels at a time is inefficient unless all four pixels are used. With triangle screen size decreasing with time, the hit rate of four-pixel stamps becomes smaller and smaller.

4.4.4 Fragment Program

The fragment program, specified by the programmer and compiled by the RTSL compiler, generates a color per fragment. The program typically uses the perbegin information and the per-fragment interpolants, such as lighting colors or texture coordinates, in computing the final color. This program corresponds to the final and finest computation freqency, "per fragment." The code generator for the fragment program is the same as that for the vertex program; their functionality is identical, and the discussion below applies to both the fragment and vertex programs.

Logically, the fragment program corresponds to a single kernel: its inputs are the per-fragment interpolants and the perbegin parameters. Each fragment is independent of each other fragment, and the same operations are applied to each fragment, so it is a perfect match for SIMD execution. However, because of texture accesses, the fragment program is more complex, as described below.

Handling texture In the stream programming model, kernels cannot index into memory directly. Instead, to retrieve a stream of data from memory (such as color, depth, or texture data), the kernel must create a stream of memory addresses. The memory address stream is then sent to Imagine's off-chip memory, which returns a stream of data. This data is then available to other kernels.

While this programming model has the disadvantage of splitting a task that is logically one kernel into two, it allows considerably greater execution efficiency. Kernels are efficient because they only operate on local data. Texels are certainly not local to execution units and are possibly not even on-chip. By separating the texture access from the texture use, we allow latency tolerance in the texture lookup. While the memory system is busy satisfying the stream of texture requests, the clusters are free to execute another kernel and

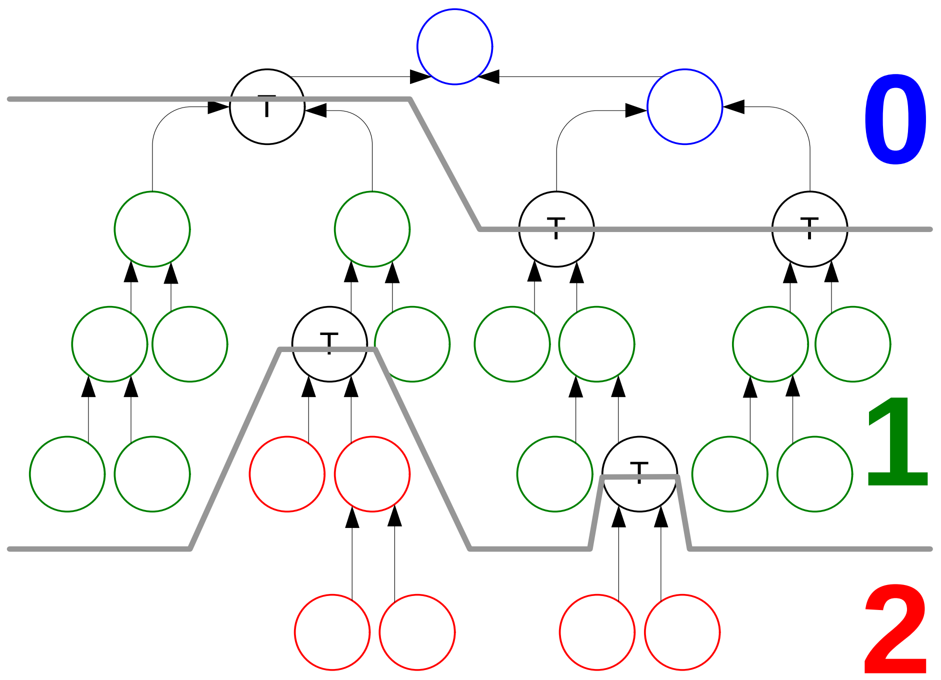
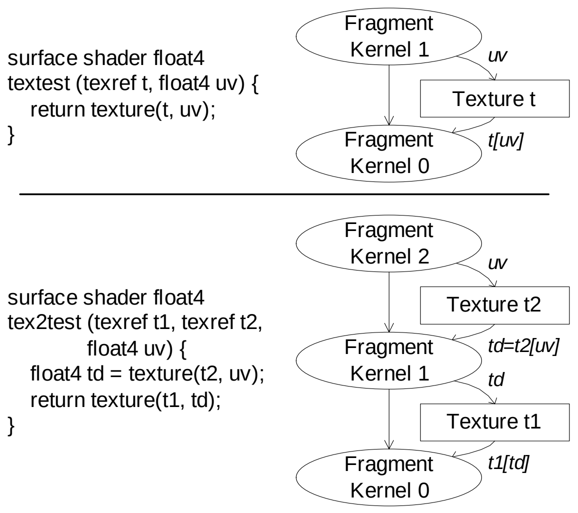


Figure 4.6: Mapping a Complex Fragment Program to Multiple Kernels. Each operation in this sample fragment program is indicated by a circle, with dependencies shown as arrows. Nodes indicating a texture access are marked with a "T". Because of the texture accesses, the program must be split across multiple kernels. This particular program has two levels of texture accesses (as in a dependent texture), meaning it requires 3 kernels (shown as levels 0, 1, and 2 in blue, red, and green) to implement.

not stall waiting for the texels to return from memory. When the texel stream is complete, the kernel which uses the texel data can then be issued and process the texels. This kernel will also have all of its data local to the clusters.

So, in the stream programming model, any fragment program with texture accesses must be divided into multiple kernels. The shading language compiler handles this automatically at compile time by labeling each texture access in the intermediate representation with its "depth" (depth n for texture $\mathcal{T}$ indicates that a string of n dependent textures depends on $\mathcal{T}$). All operations which contribute to calculating textures with depth n are grouped into a single kernel, and operations which contribute to multiple depths simply perform their calculations in the largest depth's kernel and pass their values in a stream



```
surface shader float4
textest (texref t, float4 uv) {
   return texture(t, uv);
}
```

```
surface shader float4
tex2test (texref t1, texref t2,
          float4 uv) {
   float4 td = texture(t2, uv);
   return texture(t1, td);
}
```

Figure 4.7: Mapping Texture Accesses to the Stream Model. The first example is a single texture lookup, which compiles to two kernels with an intermediate memory access. The second example is a dependent texture read, which requires three kernels and two lookups. Note that while these examples perform no calculations on the texture addresses or their returned texels, real shaders often manipulate either or both. In that case, the necessary calculations are performed in the fragment kernels.

directly to lower depths. Figure 4.6 shows an example.

The kernel for depth n generates a single stream of texture accesses for all textures with depth n, which are then sent as indexes into memory. The memory returns a texel stream which is input into the kernel for depth $n - 1$. The final kernel, at depth 0, calculates the final color. Figure 4.7 shows two examples of how the fragment program is divided into kernels.

Vertex textures, when used, are handled in the same way. This scheme can both handle an arbitrary depth of texture dependencies and perform arbitrary functions on the returned texels at each step. While typically texture maps contain colors, in practice a texel can be any datatype: a greyscale intensity, a texture address, a normal vector, and so on.

Procedural noise Vertex and fragment programs can efficiently implement complicated functions such as procedural noise. Our system calculates noise procedurally without using any textures, saving the texture bandwidth usually used in implementing noise in real-time systems. We use Greg Ward's implementation of Ken Perlin's noise function [Ward, 1991] as a basis for our procedure; it uses a pseudorandom number generator, a gradient-value noise function, and Hermite cubic spline interpolation between grid points. This algorithm gives good image quality but is reasonably expensive in terms of computation; more or less complex noise functions could easily be implemented if desired.

4.5 Composition

After the fragment program completes, we have a stream of processed fragments. Each fragment carries an address into the framebuffer, a triangle ID tag for ordering, a depth value, and a final color. The task of the composition stage is to use these fragments to create an image in the color buffer.

Complicating this task is our ordering constraint: if triangle T_1 is specified before triangle T_2, each of T_1's fragments must appear to be processed before any of T_2's. The phrase "appear to be" is significant: fragments can be processed out of order, but the end result must be the same as if they were processed in order.

Thus we begin our composite stage with a sort to reorder the fragments, then proceed to a depth test for hidden surface elimination, and finally write into the color and depth buffer. This kernel flow is shown in Figure 4.8.

4.5.1 Sort

At the beginning of the sort, we have an unordered list of fragments, each with a triangle ID. We can thus sort the fragments by ID and put them back into order.

The sort algorithm is straightforward: first the fragments are divided into chunks of 32 fragments each. Each chunk is sorted locally within the clusters (`sort32`). Then the sorted chunks are recursively merged using a merge sort (`mergefrag`). The cost of this general sort over a typical batch size of 1000 fragments, however, is prohibitive: early tests indicated it took about half the total run time of the batch. Clearly this presented an unacceptable performance degradation.

To reduce the cost of the sort, we analyzed the characteristics of the fragments in a

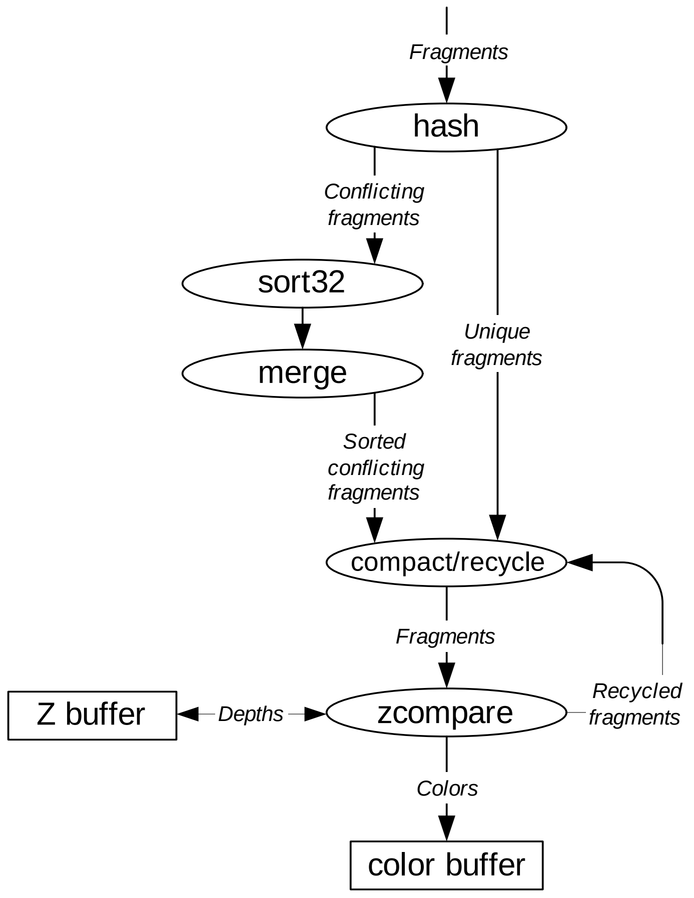


Figure 4.8: Composition in the Rendering Pipeline.

batch. If a fragment's address into a framebuffer is unique within the batch, the fragment can be processed in any order, because it conflicts with no other fragments in the batch. The typical framebuffer has on the order of a million pixels, the typical batch about one thousand, so conflicting fragments should be rare and unique fragments common.

In our implementation, we divide the stream of fragments into two streams, a unique stream and a conflicting stream. We use a `hash` kernel, with an associated hash function, to perform this operation. The bottom 5 bits of each of the x and y coordinates into the framebuffer become a 10-bit index into the hash table (packed into 64 entries of each cluster's 256-entry scratchpad register file). Geometrically, the hash table is a 32×32 stamp in screen space; any two fragments within 32 pixel locations of each other in either the

x or y directions will hash to different values. We chose this function because adjacent triangles often exhibit high spatial locality, particularly when expressed in triangle strips or polygons. Other hash functions could easily be supported in the kernel, or at the cost of more setup time and scratchpad space, the hash table could be made larger.

After the fragments are hashed, the conflicting stream is sorted using the `sort32` and `mergefrag` kernels and appended to the unique stream, resulting in a fragment stream that obeys the ordering constraint.

4.5.2 Compact / Recycle and Depth Compare

We would now like to use the depth buffer to composite the fragment stream into the color buffer. Each fragment must compare its depth against the current depth in the depth buffer and, if closer to the camera, write its color and depth values into the color and depth buffers.

Because the fragments are batched, the natural way to evaluate the depth test is to send the whole stream of addresses to the depth buffer, retrieve a stream of old depth values, compare that stream against the new depth values, and composite only those fragments which passed the depth test. But naively evaluating this sequence leads to a problem.

Consider two fragments, $\mathcal{F}_1$ and $\mathcal{F}_2$, with the same framebuffer address. $\mathcal{F}_1$ is ordered before $\mathcal{F}_2$ and is also closer to the camera. Also suppose that the value currently in the depth buffer at that framebuffer address is D, which is behind both $\mathcal{F}_1$ and $\mathcal{F}_2$.

If the two fragments are processed in a single pass, they will both retrieve the old depth value D from the depth buffer. Both fragments will pass the depth test because both $\mathcal{F}_1$ and $\mathcal{F}_2$ are closer to the camera than D. The color buffer is then updated with a stream of indices including $\mathcal{F}_1$ and $\mathcal{F}_2$. When this stream is sent to the color buffer, it first writes the color value associated with $\mathcal{F}_1$ (because it is ordered first) then the value associated with $\mathcal{F}_2$. At the end of the color store, the fragment of interest has the color from $\mathcal{F}_2$ instead of $\mathcal{F}_1$.

To properly composite the sorted fragments, we must amend the above sequence. The operations remain the same, but we cannot allow fragments with the same framebuffer addresses to participate in the same pass. So instead, we execute multiple passes of this sequence and process only a subset of the fragments on each pass. If the maximum depth complexity of all fragments in the batch is d, we will run the sequence d times. However, given f total fragments, we will only process f fragments in aggregate across all d runs.

To choose the correct fragments for each pass, we run the `compact_recycle` kernel. It takes the sorted fragment stream as an input and outputs two streams: one compacted stream of the first fragment at each unique framebuffer address and one recycled stream of all other fragments, used as the fragment stream for the next pass. Making this determination is simple: because the fragments are sorted, each fragment looks at its immediate predecessor in the stream. If the predecessor has the same address, the fragment is recycled for the next pass. Otherwise, it is appended to the compacted stream and depth-composited.

The compacted fragment stream's addresses are sent to the depth buffer, which returns the old depth value. The `zcompare` kernel then inputs the compacted fragments and the old depth values and outputs those fragments that pass the depth test. Finally, the color and depth values of those fragments are written into the color and depth buffers.

The compact-recycle loop continues until the recycle stream is empty. In practice, in our scenes, the loop usually runs only once because no fragments conflict, and almost never more than twice.

More efficient composition

The compact-recycle loop allows the most general compositing, including blending, at the expense of the loop overhead and the multiple loop iterations. Several common compositing modes lend themselves to an optimization, however.

If fragments that fail the depth test are discarded and blending is disabled, the compact-recycle loop can be entirely avoided. Instead of sorting by framebuffer address then by triangle ID, we sort by framebuffer address then depth. Out of all fragments at any given address, we only care about the one closest to the camera, and the other ones can be discarded with no further processing. In this case `compact_recycle` becomes simply `compact` and the recycled output stream is discarded. Removing this data-dependent loop also makes the job of the stream scheduler easier. However, all scenes described in this dissertation use the full `compact_recycle` loop.

4.6 Challenges

4.6.1 Static vs. Dynamic Allocation

The scenes demonstrated in this dissertation were first profiled. The profile information was then used to make allocation decisions. In particular, we sized batches so that intermediate results fit in the SRF.

This strategy ensures maximum performance for our scenes, but it is based on static information. To perform the same calculation dynamically, the drivers or hardware must know the screen-space area of the primitives they render. While runtime bounding-box estimates may be sufficient for this purpose, we have not yet investigated the ramifications of doing all allocation dynamically.

4.6.2 Large Triangles

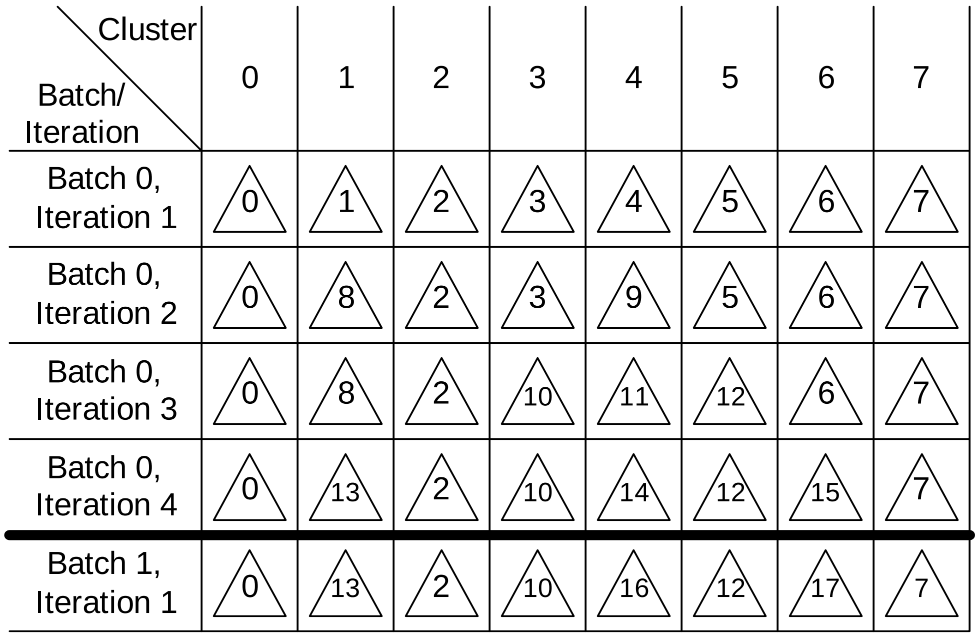
When triangles are dynamic, what do we do with large triangles, or equivalently, what do we do when the allocation is insufficient to hold all the intermediate results? A single large triangle can easily generated enough fragments to overflow the entire SRF.

Subbatching rasterization work

At first glance, we might think to just divide up the rasterization work into multiple subbatches: generate fragments until the allocated stream space is full, finish processing all those fragments, and continue rasterizing more fragments from the same batch.

However, this approach violates our ordering constraint, as shown in Figure 4.9. Since after rasterization, the generated fragments are out of order, and we process each subbatch individually, fragments from a later triangle might land in an earlier subbatch than fragments from an earlier triangle.

To remedy this problem, we begin by determining the ID of the last triangle to finish rasterizing, T_f. We would toss out all fragments for triangles with ID's greater than T_f and rasterize them in the next subbatch, which would begin with triangle T_{f+1}. But what if the first triangle in a subbatch, by itself, exceeds the allocated space? We then rasterize as



Batch/ Iteration \ Cluster	0	1	2	3	4	5	6	7
Batch 0, Iteration 1	0	1	2	3	4	5	6	7
Batch 0, Iteration 2	0	8	2	3	9	5	6	7
Batch 0, Iteration 3	0	8	2	10	11	12	6	7
Batch 0, Iteration 4	0	13	2	10	14	12	15	7
Batch 1, Iteration 1	0	13	2	10	16	12	17	7

Figure 4.9: Subbatching Rasterization Work Violates Ordering. Consider fragment batches that are limited to 32 fragments. Triangle 0 is not complete when the batch is full. Outputting and processing the 32-fragment batch would violate the ordering constraint because not all of Triangle 0's fragments would be processed before the triangles that followed it.

much as we can, and maintain the screen location where we stopped rasterizing as the start position for the next subbatch.

This algorithm is sound, but inefficient. In particular, with very large triangles, it loses all data parallelism. Consider 8 identical triangles, each of which rasterizes into enough fragments to take up slightly more than 1/8 of the allocated space. When these triangles are rasterized, none of them will finish, and we will only be able to send the portion of the first triangle which we have rasterized to the next stage. The (identical amount of) work done on the other seven triangles will be discarded.

Geometry prepass

Another option is a prepass: the geometry is first streamed through the clusters, which calculate triangle areas and dynamically stripmine the geometry into batches. After all the batches have been calculated, the renderer is then run on the batches.

This method has two main advantages: it does not depend on any profile information, and it can size the batches to fill the entire SRF with intermediate results, giving the greatest efficiency.

Unfortunately, it also has two major disadvantages. The first is that the geometry must be sent through the system twice instead of once. Host-to-graphics bandwidth is the bottleneck in many applications, and in sending the geometry twice, we violate our design goal of matching the input-output bandwidth requirements of modern commercial graphics chips.

The second disadvantage is the computation requirement. To calculate triangle areas is relatively straightforward, involving two matrix multiplications to transform each vertex, triangle assembly, and the area computation. However, the area is not sufficient to calculate the number of fragments generated. Pathological cases such as long, thin triangles located on a scanline will foil area calculations. To acquire an accurate fragment count requires actually rasterizing the triangle, which significantly increases the amount of computation required.

Spilling

A third algorithm involves rasterizing the common case with all intermediate results in the SRF, but providing for spilling to memory for the uncommon case where too many fragments are produced. Even though the runtime cost of each spilled batch is large, spilled batches would be infrequent, and the overall runtime would not increase by very much. In fact, by allowing the largest batches to spill instead of sizing the batches to accommodate the largest batch size, we can increase the amount of work per batch and gain an overall runtime benefit from longer streams.

We have a couple of options for how to spill:

Partial double buffering In partial double buffering, a stream is allocated for a certain length. If that length is exceeded, the stream automatically spills to memory. When used, the stream would automatically be loaded on demand from memory. Given sufficient support from hardware, this spilling could be made transparent to the programmer.

In the rasterizer, the fragment streams would be declared as partial double buffered and sized for the common case. The batches with oversized fragment counts would then automatically be spilled to memory if necessary.

Rasterization restart A less aggressive method is rasterization restart, which requires less hardware support. In it, the rasterization kernel would keep track of the number of generated fragments. If this number was less than the allocated number, the rest of the pipeline would proceed normally. But if the number was reached or exceeded, a different stream program would be executed, one which always spilled all intermediate streams to memory. This program would restart the rasterization work from the beginning.

We have now described a rendering pipeline that begins with a geometric description and ends with an image. In the next section, we describe and analyze the performance of this pipeline.

Chapter 5

Results and Analysis

In this chapter we analyze the performance of our implementation on the Imagine Stream Processor. We begin by describing our test scenes and our experimental setup. We then look at the results of running these scenes on a cycle-accurate simulator: frames per second, primitives per second, memory hierarchy performance, cluster occupancy, and kernel breakdown.

Next we discuss some aspects of our implementation that impact its performance: batch size, short stream effects, SIMD efficiency, fill and vertex rates, and our rasterizer choice. Finally, we characterize the kernels by performance limit and scalability and compare the theoretical bandwidth requirements against the achieved performance of our implementation.

5.1 Experimental Setup

For the results in this chapter we used two different simulators. The first simulator, the cycle-accurate simulator called `isim`, models the complete Imagine architecture, including computation, stream and kernel level control, and memory traffic and control, with cycle accuracy and has been validated against our RTL models and circuit studies. The second simulator is a functional simulator called `idebug` that is integrated into our development environment. It gives accurate results of kernel runtime but does not take into account kernel stalls, memory time, or contention between Imagine's resources. Unless otherwise noted, the cycle-accurate simulator `isim` is used for all results; it models a 500 MHz Imagine stream processor with external SDRAM clocked at 167 MHz.

Both simulators are configurable via a machine description file that specifies the components of the processor and the delays in computation and communication between its constituent parts. These delays have been validated against Imagine's RTL description and simulations of its physical implementation.

At the beginning of each simulation, all input data streams (the list of vertices, the input to the perbegin kernel, the texture maps, and the color and depth buffers) were located in Imagine's main (off-chip) memory, and at the end, the complete output image was also in Imagine's main memory.

5.1.1 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 $\times$ 720 pixels.

Scene 1: Sphere Our first scene is a Gouraud-shaded rendering of a finely subdivided sphere, specified as separate triangles and lit with three positional lights with diffuse and specular lighting components.

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 $\times$ 512 texture map.

Scene 3: Advs-8 ADVS-8 is identical to ADVS-1, except all texture lookups are mip-mapped with 8 samples per fragment.

Scene 4: Pin The bowling pin shader (from the UNC Textbook Strike dataset) has 5 fragment textures applied to it as well as a procedurally specified light with diffuse and specular components. Each texture is sampled once per fragment. The pin geometry is a pretessellated NURBS, diced to 200 by 200.

Scene 5: Pin-8 PIN-8 is identical to PIN, except all five texture lookups are mipmapped with 8 samples per fragment.

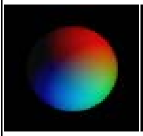
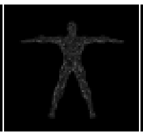
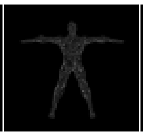
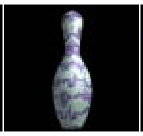
Scene 6: Marble This test uses the same geometry as PIN but uses procedural turbulence involving 4 noise calculations per fragment to generate its appearance. The fragment program applies over 1200 operations to each fragment. The shader's noise function is described in Section 4.4.4; the shader itself uses no texture maps and is courtesy of David Ebert.

Scene 7: Verona VERONA perturbs the positions and normals of each vertex of a 200 $\times$ 200 tessellated sphere using a single combined per-vertex displacement/bump map texture lookup. It displaces the position along the normal, then perturbs the normal in tangent space in directions parallel to the surface tangent and binormal vectors. It then performs a per-fragment bump map lookup to perturb a per-fragment normal given interpolated binormal and tangent vectors. The perturbed normal is used to compute a reflection vector which is used to index into an environment map. This "environment-mapped-bump-map" calculation involves a dependent texture read.

5.2 Results

5.2.1 Performance and Occupancy

We begin by looking at the performance in frames per second of our test scenes. Figure 5.1 shows the results for our scenes. All scenes run well above the 10 frames per second traditionally considered the lower limit for interactivity and our fastest scene, ADVS-1, runs at over 150 frames per second.

	SPHERE	ADVS-1	ADVS-8	PIN	PIN-8	MARBLE	VERONA
							
Vertices	245,760	62,576	62,576	80,000	80,000	80,000	80,000
Triangles	81,920	25,704	25,704	80,000	80,000	80,000	80,000
Fragments	180,909	70,384	70,384	91,591	91,591	91,594	266,810
Frags/tri	4.4	5.5	5.5	2.3	2.3	2.3	6.7
Interpolants	5	4	4	16	16	13	20
Per-V BW[a]	32	28	28	44	44	32	84
Per-F BW[b]	12	16	44	32	172	12	28
Textures	—	1 F	1 F	5 F	5 F	—	1 V, 2 F
Batch size	480	288	48	120	32	112	40
FPS	39.76	150.33	62.88	54.82	23.38	45.64	21.41
Mem BW[c]	399	424	301	354	451	167	304

[a]Per-vertex memory traffic (bytes).
[b]Per-fragment memory traffic (bytes): texture accesses, depth buffer reads and writes, and color buffer reads and writes.
[c]Total memory traffic required to sustain peak Imagine performance (MB/s).

Table 5.1: Scene Characteristics.

Next we compare the performance of a subset of these scenes[1] to representative hardware and software graphics implementations.

Two configurations represent typical configurations of the year 2002: the hardware implementation is a NVIDIA Quadro4 700XGL running on a 930 MHz Pentium 3 processor with 512 MB of Rambus RDRAM under Microsoft Windows 2000 Professional version 5.0.2195. The software implementation is the same system running with no hardware acceleration, using the software NVIDIA drivers. The NVIDIA drivers are version 6.13.10.2942 and the Microsoft opengl32.dll version is 5.0.2195.4709.

[1]Comparing MARBLE to OpenGL implementations is impossible because of the requirement for floating-point computation in the fragment states; comparing VERONA is also impossible because of VERONA's vertex textures. Neither of these features is supported in contemporary hardware OpenGL implementations, though floating-point computation at the fragment level is imminent; even then, however, the large number of operations (1200) on each fragment may make an efficient hardware implementation at least a hardware generation away.

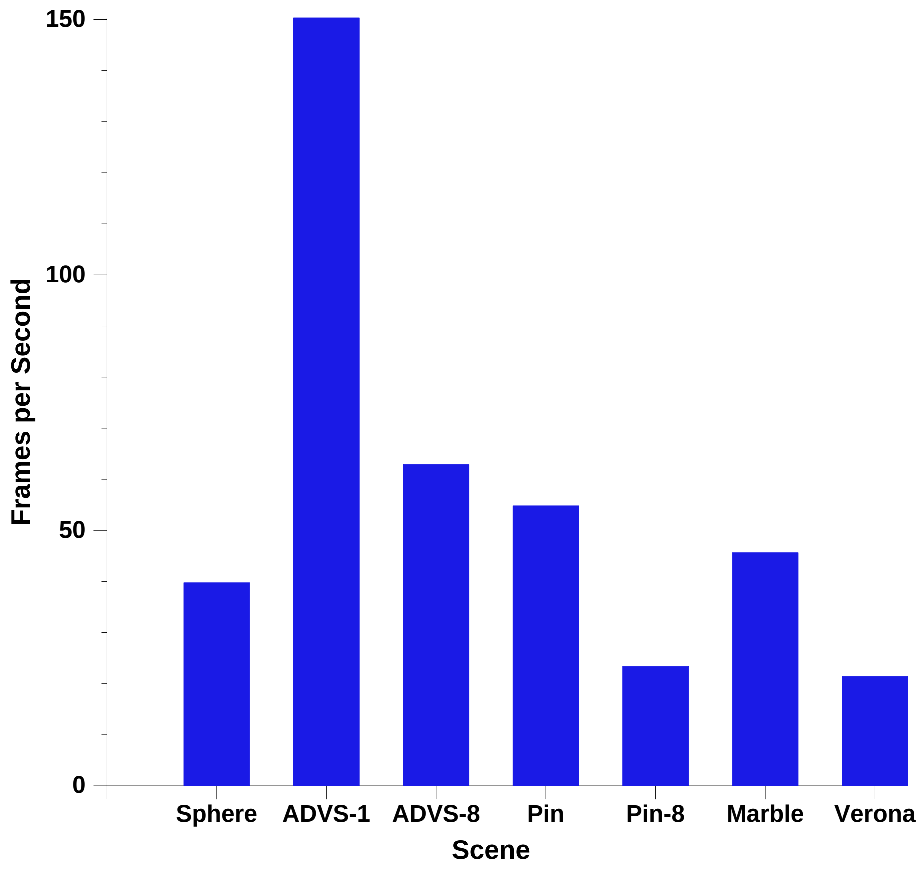


Figure 5.1: Test Scenes: Frames per Second. All data was acquired using the cycle-accurate simulator for an Imagine stream processor running at 500 MHz.

The two other configurations are typical configurations of the year 2000. The hardware-accelerated system for 2000 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. Data from the year 2000 configurations is taken from our previous work [Owens et al., 2000] and does not reflect the PIN scenes.

We measured the performance on these scenes by implementing them in OpenGL. On the PC systems, we ran these from immediate-mode arrays of vertices in system memory

(ADVS and SPHERE) or through Kekoa Proudfoot's lightweight RTSL scene viewer, using GL vertex arrays (PIN), to remove as many effects of application time on frame rate as possible. 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. All windows were single-buffered and neither flushes nor buffer clears were used. The ADVS scenes, because they appeared to be limited by host effects, were also run separately with display lists.

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 in immediate mode, 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 may have a greater cost associated with this communication. Running scenes on the commercial systems with display lists eliminates many of the host effects because the software drivers can more efficiently feed data to the graphics processor. Using display lists in this manner would allow the fairest comparison between Imagine and the commercial systems. However, display lists also perform compile-time optimizations on the data stream that are unavailable to immediate-mode renderers or to Imagine. Thus running with display lists biases the comparison in favor of the commercial systems, but does provide an upper bound on commercial system performance.

Figure 5.2 shows the results of running this subset of our scenes against these software and hardware commercial implementations. Broadly, Imagine is significantly faster than software implementations (over an order of magnitude than the 2000 implementation and on average, several times faster than the 2002 configuration). Imagine's performance is roughly comparable to the 2000 NVIDIA Quadro, a graphics processor with a similar transistor count but lower clock speed (135 MHz) than Imagine. And NVIDIA's latest processor, the Quadro4 700XGL (with three times the number of transistors and a 275 MHz

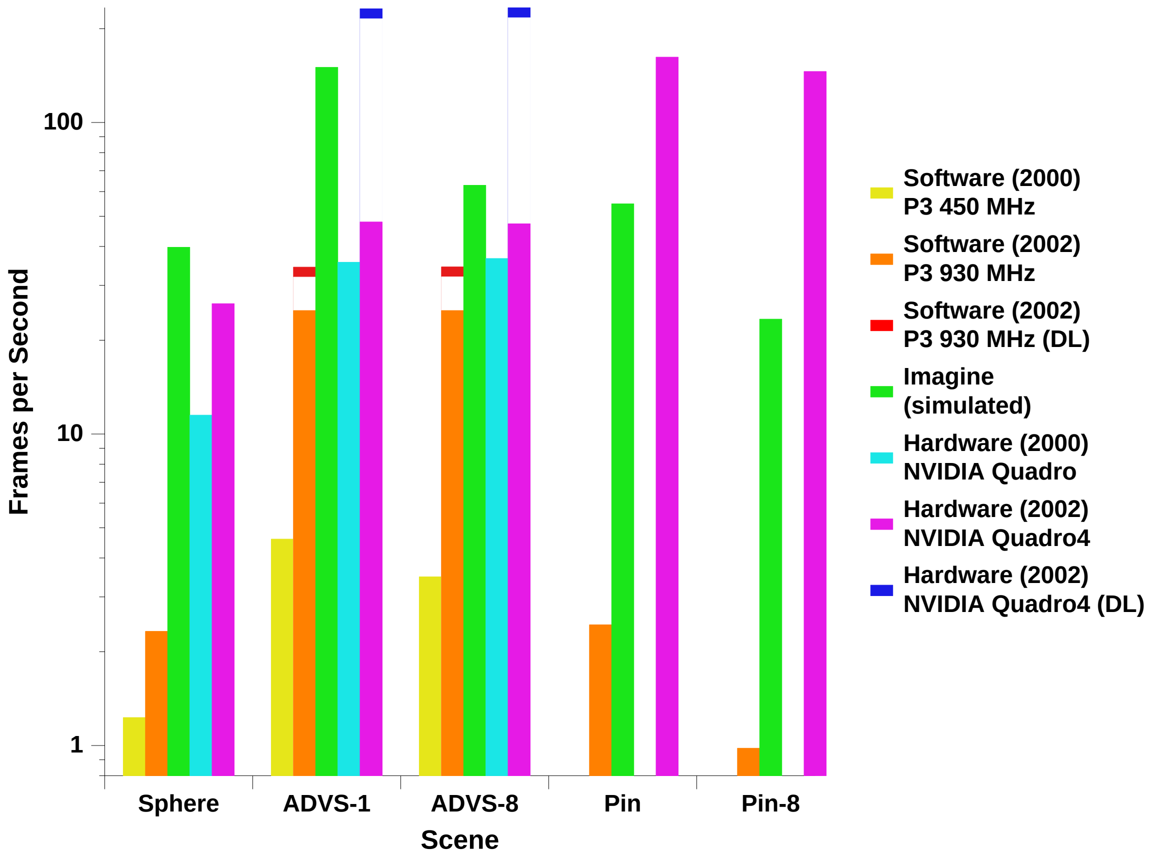


Figure 5.2: Test Scenes: Comparison Against Commercial Systems. Imagine is compared against representative commercial software and hardware implementations of the years 2000 and 2002. The ADVS scene was also measured with display lists (marked DL) for both 2002 commercial configurations.

internal clock), runs at worst 3 times slower than Imagine and at best more than 6 times faster, with a geometric mean of 23.9% faster. The two display list tests (on the ADVS scenes) run on average 139% faster. The reasons for the performance differences with today's special-purpose hardware are discussed more fully in the following sections of this dissertation, particularly in Section 7.1.

Because the different scenes have different primitive counts and amounts of work per primitive, it is difficult to draw many conclusions from only frames-per-second numbers. However, these numbers do allow comparisons between scenes with identical (or similar) input datasets. We can see that adding mipmapping to the ADVS-1 scene, resulting in

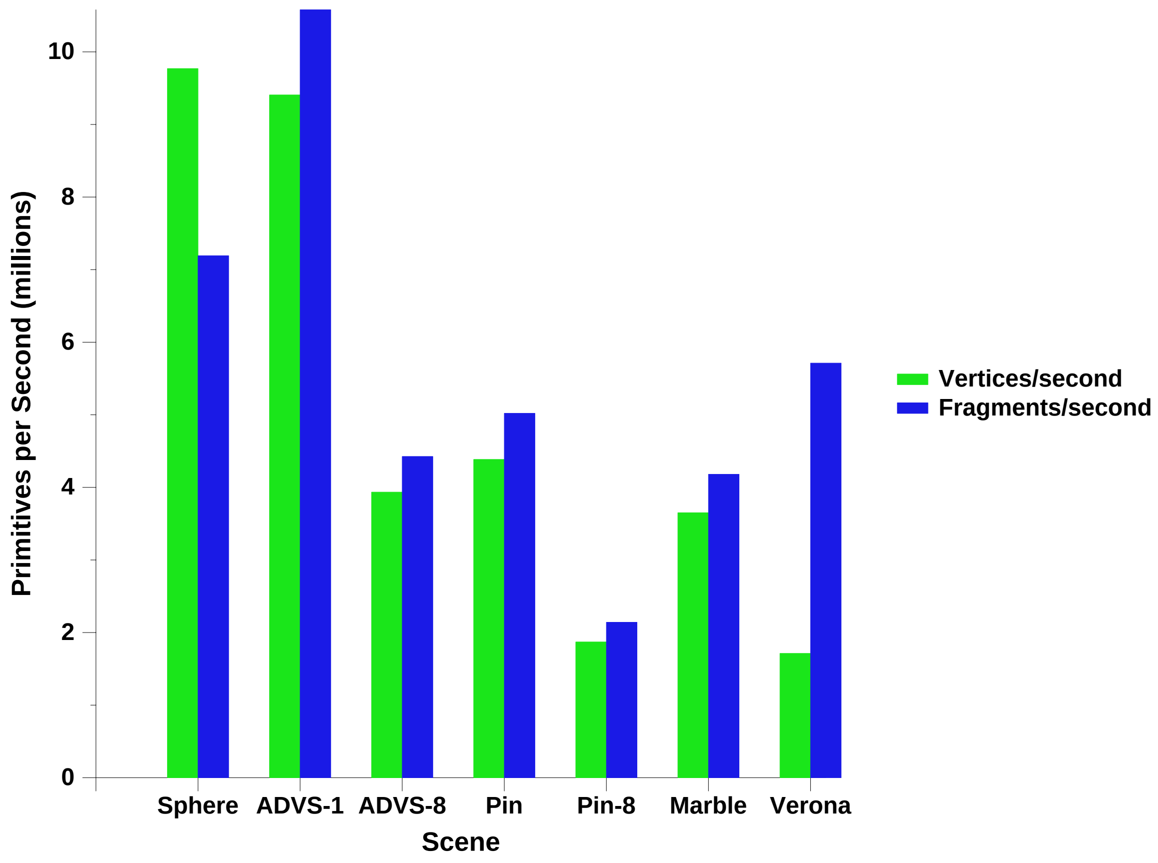


Figure 5.3: Test Scenes: Primitives per Second. Bars show each scene's delivered vertex-per-second (green) and fragment-per-second (blue) rates.

ADVS-8, more than halved the runtime. And we can see that on the pin dataset, the complex noise calculation in MARBLE is slightly more expensive than the 5 point-sampled texture lookups in PIN, but less than half the cost of mipmapping each of those 5 texture lookups in PIN-8.

Another measure of system performance is the achieved vertex and fragment rates of the system, shown in Figure 5.3. Comparing the vertex and fragment rate for any given scene just gives the ratio of vertices to fragments for that scene. More interesting is looking at a specific rate across multiple scenes, which allows a comparison between the amount of work per primitive across those scenes. However, this may be somewhat misleading if one primitive has a disproportionate amount of work. A large amount of work in one primitive

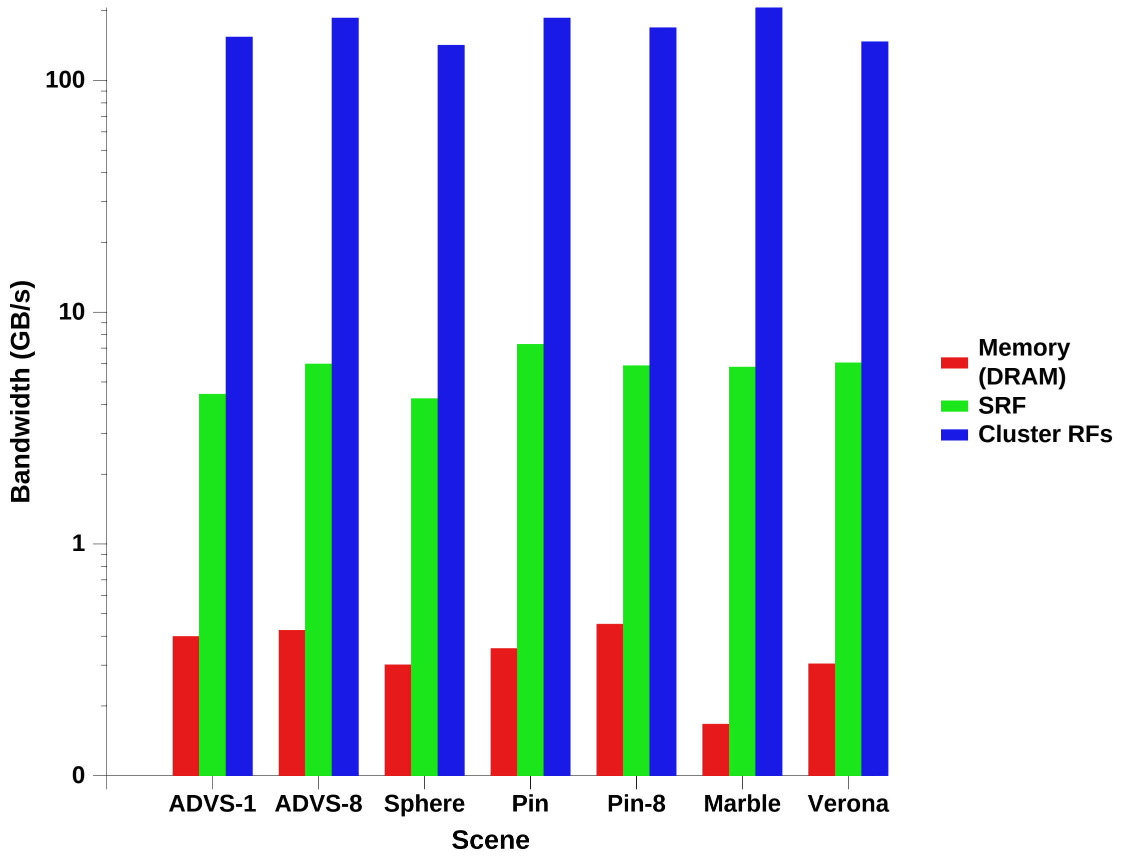


Figure 5.4: Test Scenes: Memory Hierarchy Bandwidth. Bars indicate the memory bandwidth measured for scenes at all three levels of the memory hierarchy. Red indicates main memory bandwidth, green SRF bandwidth, and blue cluster register file bandwidth.

(such as vertices) will push down the rate for the other primitive (fragments) because the large amount of time spent processing vertices will leave less time for fragments[2].

We achieve over 1.7 million vertices per second and over 2.1 million fragments per second on every scene, with a peak rate for vertex rate on SPHERE (9.8 million vertices per second) and for fragment rate on ADVS-1 (10.6 million fragments per second). Section 5.2.5 describes a series of RATE benchmarks that feature peaks of 13.9 million vertices per second and 20.4 million fragments per second.

Chapter 3 argued that a memory bandwidth hierarchy is a cornerstone of the efficiency

[2]In our scenes, for example, SPHERE spends most of its time doing vertex calculations, and MARBLE's runtime is dominated by fragment work. See Figure 5.6.

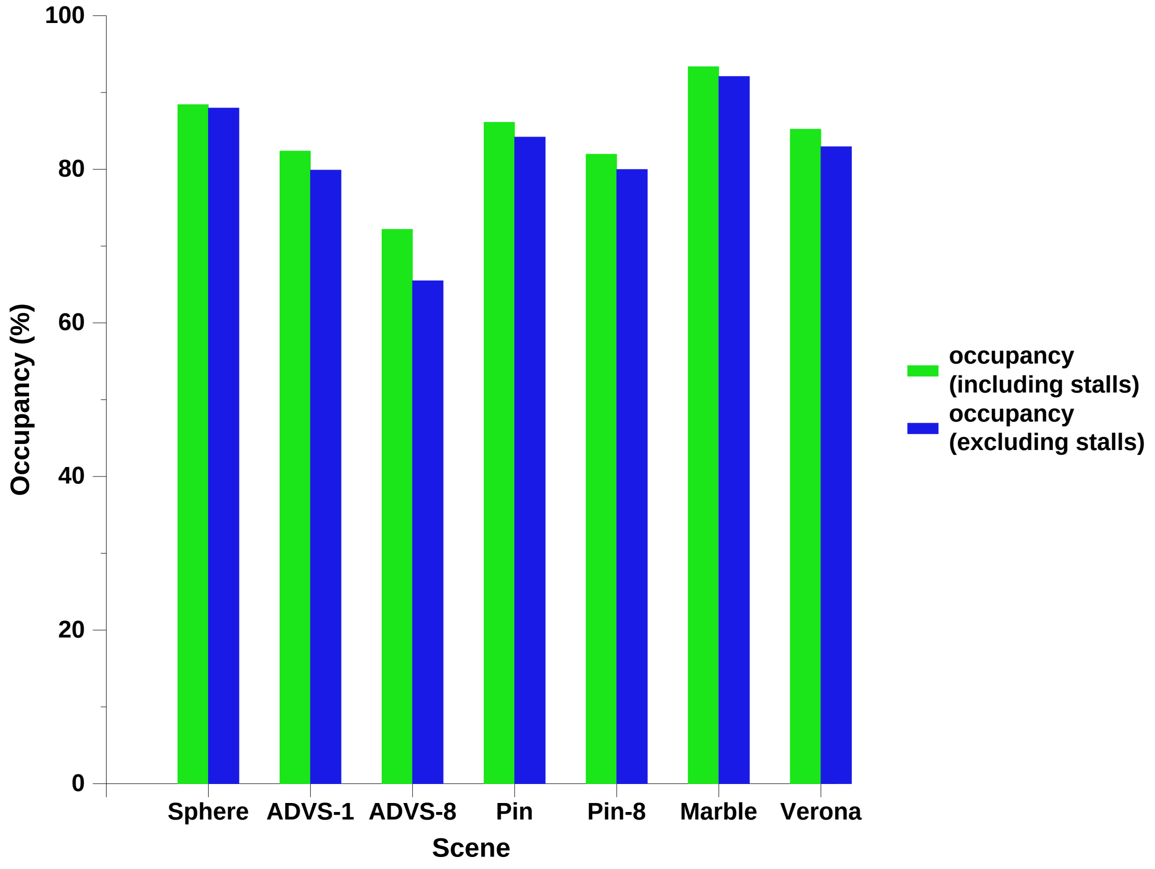


Figure 5.5: Test Scenes: Cluster Occupancy. Each scene's cluster occupancy is measured in two ways, one including cluster stalls as active (green), one excluding cluster stalls (blue).

of stream architectures. Figure 5.4 shows the bandwidths at each level of the hierarchy on Imagine. We see that each level of the memory hierarchy delivers an order of magnitude more bandwidth than the level below it. The bandwidths for each level across all scenes are relatively uniform, although textured scenes exhibit a higher memory bandwidth than non-textured ones, and the heavy per-vertex work in SPHERE and per-fragment work in MARBLE push down their main memory demands.

As we will see in Section 5.4, our scenes are computation bound rather than memory bound. For efficiency, then, we would like to keep the clusters as busy as possible. Figure 5.5 shows the cluster occupancy for each of our scenes. Here, occupancy refers to the percentage of the runtime in which the clusters are busy. Our measured occupancies range from 65% (72% counting cluster stalls as busy) to above 90%.

Ideally, the clusters would be busy 100% of the time. For several reasons, our achieved occupancy is below this ideal figure.

First, the pipeline involves a number of memory transfers. Kernels cannot begin before those transfers are complete. When run straight through, the pipeline is serialized, with the clusters completely idle from the beginning of each memory operation to the end.

We remedy this by software-pipelining the stream program (described in Section 3.3.2). We run two iterations of the loop at the same time, scheduled such that kernel execution for one iteration overlaps memory traffic for the other iteration. Software pipelining is vital to achieving high occupancies on this and other stream applications. All results presented in this work are software-pipelined at the stream level.

However, software pipelining is not perfect. It is difficult to perfectly match memory traffic in one iteration to kernels in the other. Making the job much more difficult is the widely varying amounts of work per iteration. The generated software pipeline is static and while it may be the best static pipeline for all iterations considered in total, for specific iterations it may be a poor match. And when it matches a particular iteration poorly, the cluster occupancy suffers[3].

Several other reasons contribute to a non-ideal occupancy. A minimum kernel execution time (described further in Section 5.2.3) leaves the clusters idle after running very short kernels. Conditionals in the stream program take time to evaluate on the host while the clusters wait. Conflicts in stream allocation in the SRF between the two active iterations also sometimes force kernels to wait for their allocated output SRF space to become free (as a result of a memory write, for instance) even though their inputs are all ready. The stream scheduler optimizes for long streams at the expense of these conflicts, which is generally a win for performance overall but a loss in occupancy.

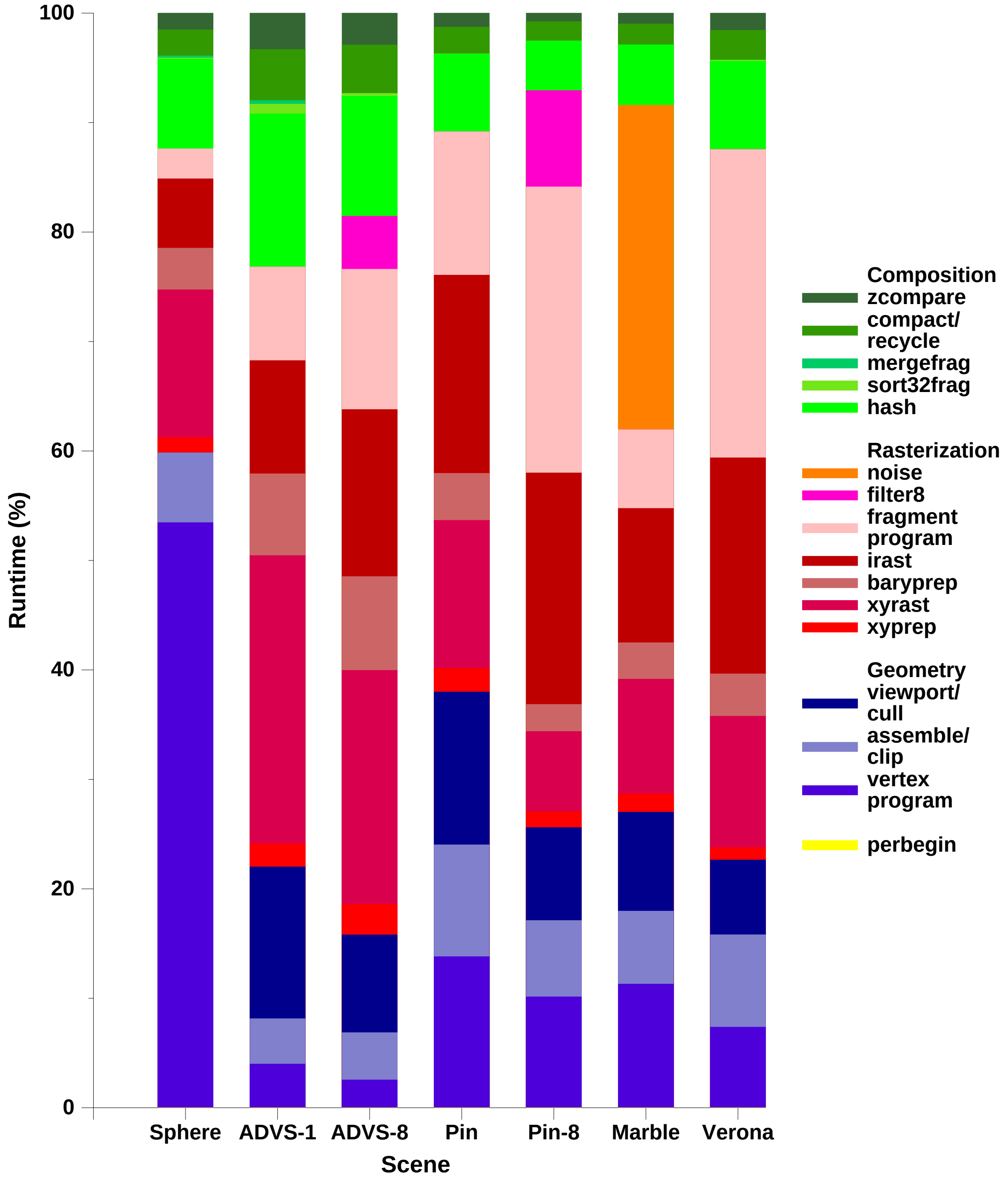


Figure 5.6: Each of the test scenes has its output normalized to 100%, and the respective contributions of each kernel to each scene are displayed as a subset of the 100%. Kernels shaded blue are in the geometry stage, red in rasterization, and green in composition. The perbegin contribution is negligible.

5.2.2 Kernel Breakdown

Figure 5.6 breaks down kernel usage for each of our scenes. In it we can see significant variance in proportional runtime for many of the kernels.

In our scenes, in general, rasterization is the largest contributor to total runtime. Of the rasterization kernels, `xyrast` is the most costly, especially for scenes with a modest number of interpolants. The work done in the fragment program varies greatly across the kernels. Of note is the `noise` kernel in MARBLE, a part of the fragment program, which accounts for the most runtime in that entire scene.

Composition imposes relatively modest costs, with the lion's share in `hash`, which must traverse the entire fragment stream twice. The sort cost across all scenes is quite small, and the `compact` and `zbuffer` kernels also have a small runtime.

The amount of work in the geometry kernels widely varies across scenes. The vertex program is generally small, except for SPHERE, which has a large amount of per-vertex lighting calculations that account for over half the overall runtime. The cost of primitive assembly/clip averages 9% of the runtime over all scenes, with viewport/cull about 6%.

5.2.3 Batch Size Effects

For maximum efficiency, we would like to store all intermediate data produced and consumed by the pipeline on-chip. By capturing this producer-consumer locality on-chip, this intermediate data uses the high bandwidth of the SRF-to-cluster connection rather than the lower bandwidth of external memory.

The entire stream of vertex inputs to the pipeline is much too large to ever fit in on-chip storage. So to keep all intermediate data local and to take advantage of our producer-consumer locality, as described in Section 4.2, the vertex stream is stripmined and processed in batches. This section explores how those batches should be sized for maximum performance. As we will see, the batch size yielding maximum performance is the maximum size before intermediate streams overflow the SRF and spill to main memory.

[3]Scenes with a large variance in work per primitive will suffer the highest losses in occupancy. The ADVS dataset has the most variance of all our scene input data, and ADVS-8's small batch size causes the variance over all batches to be greatest for that scene. Figure 5.5 shows that ADVS-8 has the lowest occupancy for all our scenes.

We take a closer look at two representative scenes, ADVS-1 and VERONA. We ran each scene through the functional simulator and the cycle-accurate simulator over a range of batch sizes.

ADVS-1 has a small amount of information per vertex and fragment and does a small amount of computation on each vertex and fragment. VERONA is the opposite: vertices and fragments are large, and have a large amount of computation performed on them. ADVS-1 can be stripmined without spilling to 288 vertices per batch; VERONA, to 40 vertices per batch.

Figure 5.7 shows the runtime and occupancy of each of these scenes as a function of the batch size in vertices. Despite differing in computation requirements, they exhibit similar behavior with respect to batch size. With small batch sizes, performance is considerably poorer than at large batch sizes with no spilling. In addition, increases in performance are more significant at smaller batch sizes than at large ones.

This behavior is common across a wide range of stream applications, including the two scenes we show here. We can fit a model to the (unspilled) runtime data as a function of batch size (the red dotted line in Figure 5.7). The model is quite simple: if r is runtime and b is batch size,

$$r = c_\infty + c_b/b, \tag{5.1}$$

where c_∞ and c_b are constants.

This model matches the measured data well; it is derived later in this section after our discussion of short stream effects. The coefficient c_∞ is of particular interest as it reflects the performance asymptote as the batch size appoaches infinity. This asymptote indicates the theoretical performance achievable when short stream effects disappear.

The results of the model are summarized in Table 5.2. This table provides the values of c_∞ and c_b for our two scenes of interest as well as the best achieved value (for comparison to c_∞) and $b_{1/2}$ (the “50% Batch Size”), which reflects the batch size at which performance is half of the asymptote c_∞. $b_{1/2}$ is equal to c_b/c_∞.

Ideally, our achieved performance would be close to the asymptote. Our ADVS-1 achieved runtime is 28% larger than the asymptote, and VERONA’s is 49% larger. Both scenes achieve considerably better performance than the performance at $b_{1/2}$. This value

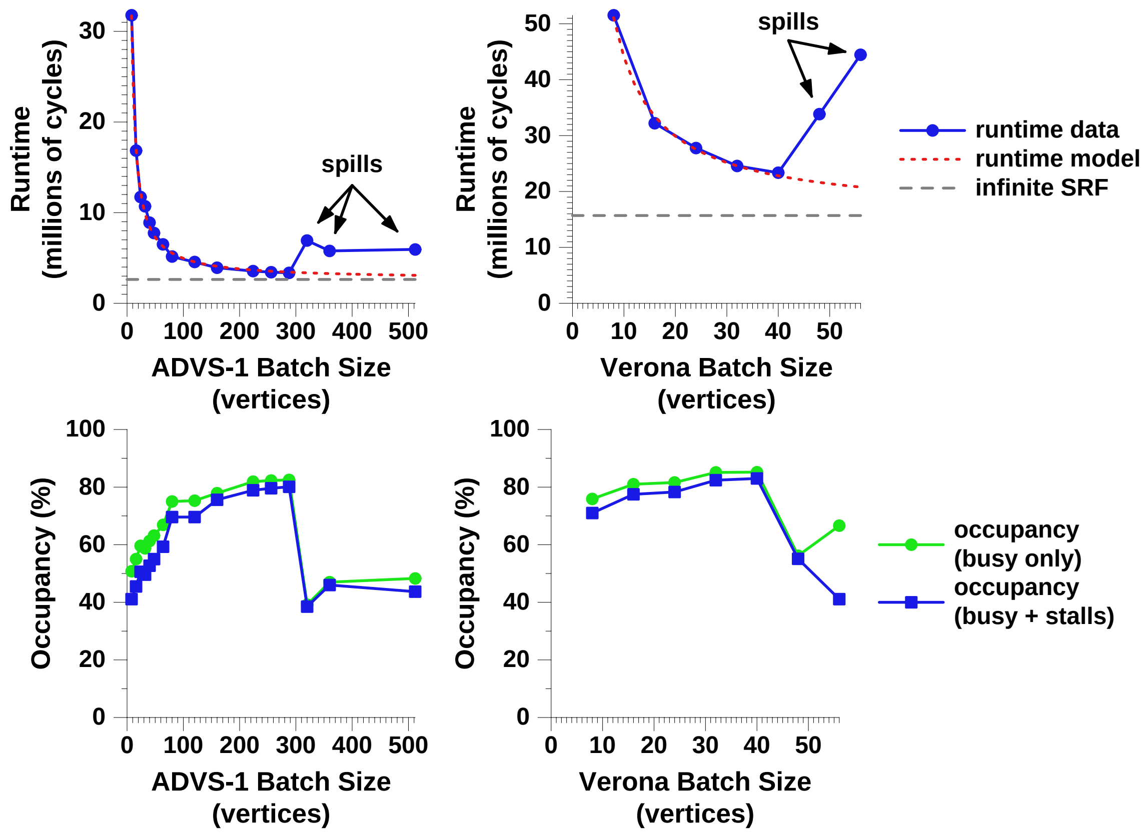


Figure 5.7: Runtime and Occupancy vs. Batch Size for ADVS-1 and VERONA. The runtime (upper) graphs indicate measured runtime as a function of batch size (blue, solid points) and the runtime model fitted to those points (red, dotted line). The dashed grey line reflects the theoretical best performance with an infinite SRF (c_∞ from Equation 5.1). The maximum batch size that fits in Imagine's SRF without spilling for ADVS-1 is 288 vertices; for VERONA, 40. Batch sizes that require spilling intermediate streams to memory are indicated in the runtime graphs. In the occupancy (lower) graphs, the upper line indicates the percentage of time in which the clusters are executing kernels including stalls; the lower line presents the same information but excludes stalls.

	c_∞	c_b	Best Achieved	$b_{1/2}$
ADVS-1	2.63M	232M	3.36M	88
VERONA	15.7M	283M	23.4M	18

Table 5.2: Runtime vs. Batch Size Model. The runtime model used here is $r = c_\infty + c_b/b$, where r is runtime and b is batch size. c_∞ indicates the theoretical performance asymptote as batch size approaches infinity; "best achieved" indicates the best performance measured (at the spilling point). $b_{1/2}$, or the "50% Batch Size," is the batch size at which performance is half of the asymptote c_∞.

indicates the relative amount of computation per primitive; a small $b_{1/2}$ means the amount of computation is relatively large per primitive, which mitigates the effects of small batch sizes. For example, because VERONA has more computation per primitive (and a smaller $b_{1/2}$) than ADVS-1, its "short-stream effects" are less pronounced. These short-stream effects are the key to understanding the loss of performance at small batch sizes and are described below.

Spilling imposes a significant and sudden penalty on runtime for two reasons. First, spilling intermediate streams contributes nothing to the computation and uses the low-bandwidth SRF-to-memory path, which is slow. Second, because of the increased load on the memory system and the resulting increase in memory system occupancy, the clusters must wait for more memory references, making the stream scheduler's efforts to keep the clusters busy much more difficult. Thus the cluster occupancy drops and the runtime increases. The bottom graphs in Figure 5.7 show the dramatic drop in cluster occupancy due to spilling.

Short Stream Effects

Imagine is optimized to operate on long streams of data. When input streams to kernels are short, we see a degradation of performance for many reasons. These short-stream effects are characteristic of both Imagine-style stream architectures as well as vector architectures.

Short-stream effects degrade performance in several ways:

Kernel dispatch Imagine kernels require 200–500 cluster cycles in the host processor to dispatch on Imagine. This length of time is termed the "minimum kernel duration."

If a kernel runs for at least 500 cycles, its successor kernel can issue right away. But if the kernel takes shorter than the minimum kernel duration, its successor still must wait for the host to finish issuing it.

Prologue and epilogue blocks Most kernels involve a main loop that iterates over all elements of the input stream. The length of time spent in this loop scales with the length of the stream.

However, in most kernels, per-kernel code is placed in basic blocks before and after the main loop. This code sets up constants, initializes variables, flushes conditional stream outputs, and so on. These blocks are run once per kernel call. If the main loop is only run a small number of times, these prologue and epilogue blocks can account for a significant proportion of the kernel runtime.

SWP prime and drain For efficiency, most kernel main loops are software pipelined. Using software-pipelined loops has the advantage of increasing functional unit usage and shortening the effective critical path. However, their cost is the time used to prime and drain the software pipeline. If a loop has software pipeline depth of d, running the loop on a c-cluster machine over $c * n$ data elements results in $n + d$ iterations of the loop.

The software pipeline depth for a typical rendering kernel is 2–4 levels. So if a loop with SWP depth of 3 is run on 800 elements on an 8-cluster Imagine, the cost of the SWP setup and teardown is only 2 extra iterations with a total of 2% overhead. But if it's run on a 32-element loop, the 2 extra iterations are a 50% overhead.

Stream pipeline difficulties Kernels operating on short streams are more difficult to pipeline at the stream level than kernels operating on long streams. Software pipelining at the stream level is most useful in covering the latency of a memory operation in part of the pipeline with a kernel call in another part of the pipeline. Kernels which run for a short duration do a poorer job in covering this latency.

Cluster stalls When a kernel is issued in the Imagine hardware, there is a delay between issuing the kernel to the point at which data is ready for reading from the stream buffers. This cost is a fixed once-per-kernel cost and is amortized across all stream elements processed in the kernel. Thus short streams have a larger per-element stall cost than long streams.

Analysis Clearly, smaller batch sizes take longer to run. Why? The occupancy graphs in Figure 5.7 explain part of the reason. They show cluster occupancy as a function of batch size, measured by cycle-accurate simulation. They measure cluster occupancy in two different ways.

The upper lines in the occupancy graphs of Figure 5.7 indicate in what fraction of the runtime the clusters are executing kernels. Software pipelining at the stream level is used primarily to increase this cluster occupancy; note that the smaller the batches (and thus the shorter the streams), the poorer the software-pipelined stream code is able to keep the clusters busy. The minimum kernel duration penalty would also influence this occupancy metric.

The lower lines also show occupancy but do not count the clusters as occupied if they are running a kernel but are stalled. The cost of the stall is the gap between the two lines in the occupancy graphs in Figure 5.7. Like cluster idle time, the cost of stalls also increases as batches become smaller.

So we see that cluster occupancy suffers from the effects of short streams[4]. But occupancy is only half the story. The other important effect is within the kernels themselves: the cost of prologue and epilogue blocks and the software-pipelined prime and drain time in the main loops.

Figure 5.8 demonstrates the performance penalty for kernels due to short-stream effects over a range of batch sizes. It measures the percentage of kernel execution time (ignoring stalls) devoted to priming and draining software-pipelined loops and to prologue and epilogue (non-loop) blocks. These costs are noticeable for even long streams (7% for ADVS-1 at its 288-vertex spilling point, 14% for VERONA at its 40-vertex spilling point) but are crippling for short streams (totaling over half the kernel runtime for VERONA with 8-vertex batches and over 60% for ADVS-1 with 8-vertex batches).

Note that even though the runtime dramatically increases once batches begin to spill

[4]The marked decrease in cluster occupancy after 288-vertex batches for ADVS-1 and after 40-vertex batches for VERONA is due to spilling and is described earlier in this section. Here we are concerned with the short-stream behavior and consider batch sizes that do not spill.

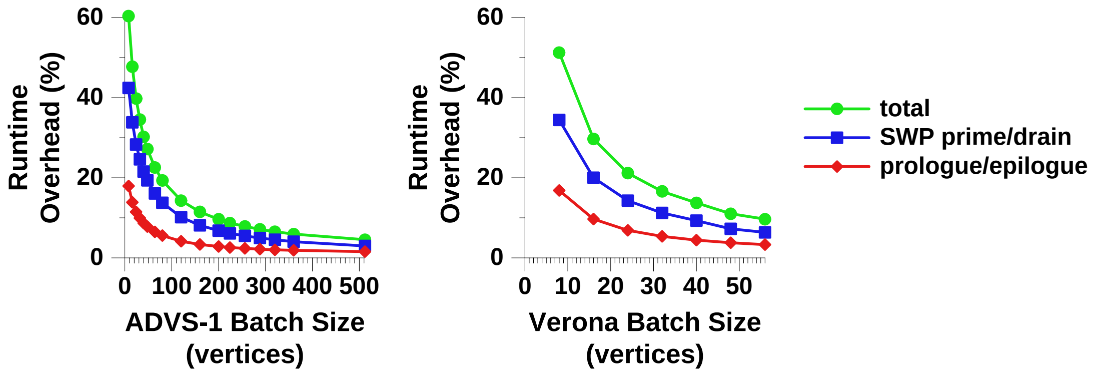


Figure 5.8: Loop Overhead Costs vs. Batch Size for ADVS-1 and VERONA. The lines indicate the percentage of kernel runtime (ignoring stalls) that do not contribute to main-loop computation. The upper line ("total") is the sum of costs due to SWP prime and drain for main loops (the middle line) and prologue and epilogue blocks bracketing the main loops (the lower line).

their intermediate values, short-stream effects are not affected by the spilling and continue to diminish as batches become larger. Figures 5.7 and 5.8 clearly show this behavior.

Derivation of our model Recall that we fit our batch size vs. runtime model as Equation 5.1. We now derive this relationship from the following assumptions:

- The kernel runtime of kernel j can be modeled as

$$r_{kj} = c_{1j} + c_{2j}\ell_j, \tag{5.2}$$

  where c_{1j} and c_{2j} are constants, and ℓ_j is the number of loop iterations required to run kernel j to completion. The constant term c_{1j} for kernel j is the sum of the prologue and epilogue blocks and the setup and teardown of the software pipelined loop. These factors do not change with loop size. The rest of the runtime is proportional to the number of times the loop is executed.

- The number of loop iterations run in kernel j, ℓ_j, is proportional to the batch size[5]:

$$\ell_j = c_{3j}b. \tag{5.3}$$

This relation is generally true of stream computation and is certainly true of our graphics kernels: having twice as many vertices in a batch means the vertex program must loop twice as many times to process them all. And the size of our intermediate data is proportional to the size of the input data—twice as many vertices implies twice as many fragments.

- Finally, the number of batches i necessary to complete the entire scene is inversely proportional to the batch size b:

$$i = c_4/b. \tag{5.4}$$

Since the input data is simply divided evenly among the batches, this assumption is also valid. And the total runtime is the runtime for each batch times the number of batches:

$$r = r_k i. \tag{5.5}$$

Substituting for r_k (with Equation 5.2), ℓ_j (with Equation 5.3), and i (with Equation 5.4) gives the following translation of Equation 5.5:

$$\begin{aligned} r &= (c_{1j} + c_{2j}(c_{3j}b))(c_4/b) \\ &= c_{2j}c_{3j}c_4 + c_{1j}c_4/b \\ &= c_\infty + c_b/b. \end{aligned} \tag{5.6}$$

Extending this analysis to multiple kernels is straightforward:

$$\begin{aligned} r_k &= \sum_j r_k j = \sum_j (c_{1j} + c_{2j}\ell_j) \\ &= \sum_j (c_{1j} + c_{2j}c_{3j}b) \end{aligned}$$

[5]This relation between batch size and loop count does not include SWP setup and teardown; that factor is already accounted for in Equation 5.2 in the c_{1j} term.

$$
\begin{aligned}
&= \sum_j c_{1j} + \sum_j (c_{2j} c_{3j} b) \\
&= c_1 + b \sum_j (c_{2j} c_{3j}) \\
&= c_1 + b c_{23} \qquad (5.7) \\
r &= r_k i = (c_1 + b c_{23})(c_4 / b) \\
&= (c_{23} c_4) + (c_1 c_4 / b) \\
&= c_\infty + c_b / b, \qquad (5.8)
\end{aligned}
$$

which is identical to Equation 5.6.

Discussion We have seen that the performance of our scenes is influenced heavily by the size of the stripmined batches used in their computation. Smaller batches suffer more heavily from short-stream effects. In particular we should strive to make our batch size larger than the 50% batch size, $b_{1/2}$, as batches sized below $b_{1/2}$ have particularly poor performance.

The simplest way to reduce short-stream effects is to have longer streams. Long streams amortize the cost of dispatching the kernel, the cost of non-loop basic blocks preceding and following the main loop across all of the elements in the stream, and the stall cost incurred between starting the kernel and the arrival of the kernel's data. In addition, if the main loop is software pipelined, long streams mitigate the per-element cost of the setup and teardown of the loop.

An alternative to having longer streams is to have more computation per stream element. Spending more time doing useful computation in the main loop reduces the proportional amount of time spent in overhead like setup/teardown loops and kernel dispatch. Combining multiple small kernels into one large kernel is a performance win for the same reasons.

In hardware, the size of the stream register file is the most important factor in having longer streams. (Also, the cost of the minimum kernel duration could be greatly reduced by having the host on the same chip as the stream processor.) In software, longer streams are enabled by good allocation by the stream scheduler.

The programmer can help reduce short-stream effects by making the prologue and epilogue blocks as short as possible and by reducing the software pipeline depth as much as possible without sacrificing loop length. Depending on the application and its stream lengths, it may be preferable to slightly elongate the loop length if such a change reduces the software pipeline depth.

5.2.4 SIMD Efficiency

One of the major goals of our renderer was to design and implement algorithms that are scalable to multiple SIMD-controlled clusters. In this section we look at the SIMD efficiency of our algorithms: in moving from a scalar (1-cluster) to a n-way (n-cluster) SIMD-parallel machine, we hope to achieve a speedup close to n. Chapter 6 explores the impact of increasing the number of clusters.

Much of the pipeline will scale perfectly with the number of clusters: the vertex and fragment programs, for instance, are purely SIMD-parallel. Many other kernels also scale well with the number of clusters. But moving to a SIMD-parallel model imposes several requirements on our algorithms that will hurt their efficiency:

- Primitive assembly is much simpler on a scalar machine. In a SIMD-parallel implementation, assembly requires communication of whole vertices between clusters, because the meshed vertices making up a triangle are processed on separate clusters. A scalar machine processes all vertices on the same cluster and requires no communication.

- Hash, sort, and merge are also considerably simpler in the 1-cluster implementation. With the 8-cluster version, these kernels must perform considerable amounts of communication between clusters because these algorithms are not data-parallel. The 1-cluster versions are more efficient because all of their data is local to the cluster without any communication necessary.

- Several kernels (particularly the interpolant rasterization kernel, `irast`) are bound by conditional stream bandwidth. Imagine implements conditional streams by using the low-bandwidth intercluster communication units. When using conditional streams, scalar implementations do not have to use this low-bandwidth path, because all stream elements must pass through the single cluster and no communication is needed.

- Even identical kernels will not achieve perfect speedup because the input streams to the scalar machine have eight times as many iterations and consequently suffer from fewer short-stream effects.

Due to these differences, as we move from one to many clusters, we should not expect a linear speedup but instead a more modest speedup.[6]

In our tests, we make comparisons between two runs of the ADVS-1 scene. First, we look at kernel statistics for this scene on the cycle-accurate simulator for the 8-cluster Imagine at its maximum batch size. The second run is the same scene with the same batch and SRF sizes on a 1-cluster Imagine. Comparing these two runs allows us to look specifically at how each kernel's runtime changes when moving from 1 cluster to multiple clusters.

Figure 5.9 shows our speedup for each kernel (in green) and each stage (in blue) when moving from the 1-cluster implementation to the 8-cluster implementation. The overall kernel speedup is 5.3, with the geometry stage at 5.1, rasterization at 5.8, and composition at 4.2.

One major cause of the non-ideal speedup is the greater efficiency of the `hash`, `sort`, and `merge` kernels in the scalar implementation. These three kernels account for 14.7% of kernel runtime in the 8-cluster version but only 10.5% in the 1-cluster version. As they account for the greatest share of the composition stage's runtime, composition has the smallest speedup of the three stages.

The other kernels with the poorest speedup were `assemble_clip` and `irast`. Both of these kernels are limited by their conditional stream bandwidth in a multi-cluster Imagine, but can take advantage of the 4-times greater unconditional stream bandwidth in the 1-cluster version.

Overall, a kernel speedup of 5.3 for an 8-cluster machine is quite acceptable. We would

[6]One effect we do not model, a small one, is input batch padding (1-cluster machines do not need their inputs padded to 8-cluster boundaries; we use the same padded inputs for all runs).

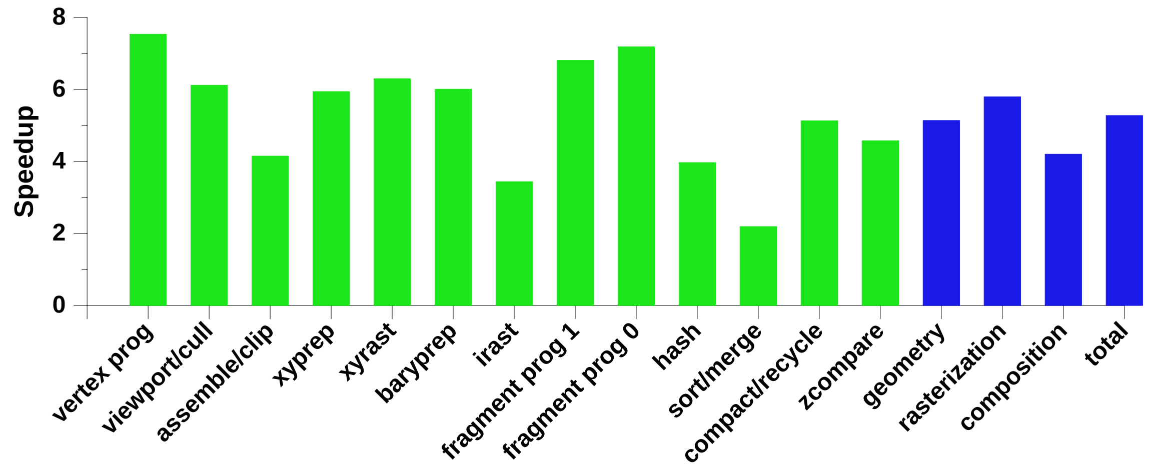


Figure 5.9: SIMD Speedup for ADVS-1. Kernel speedups are in green, stage speedups in blue. The overall kernel speedup from moving from 1 cluster to 8 clusters is 5.3 for kernel runtime only and 4.6 for the entire application. The largest reasons for a non-ideal speedup are the lower SRF-to-cluster bandwidth of conditional streams in a 8-cluster machine and the loss of efficiency in hash/sort/merge in moving to a parallel implementation.

expect that if cluster count was to continue to increase that we would achieve similar increases in speedup. With large cluster counts, this speedup would eventually be limited by short stream effects (described in Section 5.2.3) as the streams are divided among more and more clusters. To continue to achieve larger speedups as the cluster count increases, then, requires that the streams grow at the same rate, which could be achieved by growing the stream register file size at the same rate as the number of clusters. Chapter 6 looks more closely at the performance issues associated with stream architectures with a larger cluster count.

5.2.5 Triangle Rate and Fill Rate

To first order, the total runtime of a scene is the sum of its per-vertex work (roughly the geometry stages of the pipeline) and its per-fragment work (roughly the rasterization and composition stages). Although the exact number of operations for per-vertex and per-fragment work varies significantly with the lighting, shading, and texturing algorithms used in the scene, they are comparable in magnitude: Hanrahan and Akeley cite figures of 246

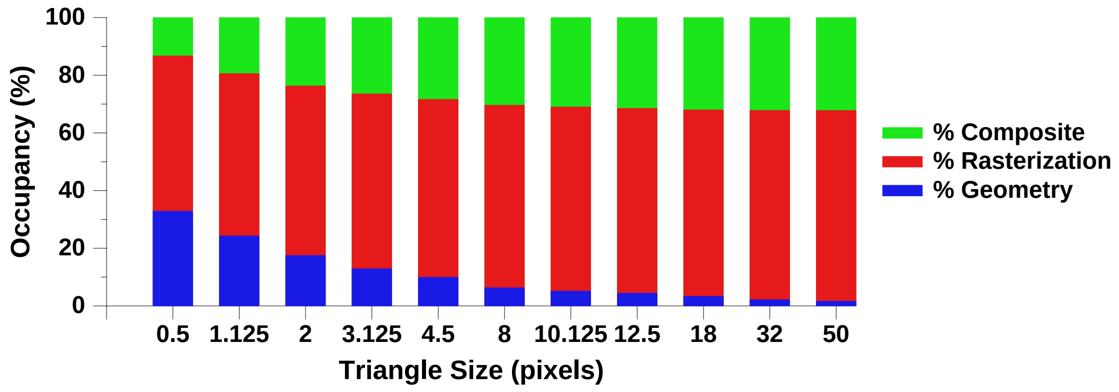


Figure 5.10: RATE Stage Runtime Breakdown. Each stage's total runtime is normalized to 100%. The majority of time for all triangle sizes is spent in rasterization.

operations per vertex and 149 operations per fragment [Hanrahan and Akeley, 2001].

In rendering a scene, then, the division of time between per-vertex and per-fragment work is dependent on the proportion between the number of vertices and the number of fragments. This proportion can be expressed with the average triangle size of the triangles in the scene. Scenes with a large average triangle size have a large number of fragments per vertex and consequently spend most of their time doing fragment work. Scenes with small triangles, on the other hand, spend correspondingly more time doing per-vertex work.

We would like to render scenes with both small and large triangles efficiently. To do this, we will analyze Imagine's performance on scenes with a range of average triangle sizes. As these scenes are designed to measure Imagine's vertex and fragment rates, they are named RATE and are characterized by the base length of their constituent triangles. In each of the measured scenes, we used long strips of identically-sized, unlit, depth-buffered, Gouraud-shaded triangles. The triangles completely fill the 720 $\times$ 720 window with a depth complexity of 1. The same kernels were used for all measurements (although a new software-pipelined StreamC loop was computed for each different scene to ensure that it ran as efficiently as possible).

Figure 5.10 shows the runtime breakdown between the three major stages of the pipeline for the RATE scenes: geometry (roughly, per-vertex work), rasterization, and composite (both per-fragment).

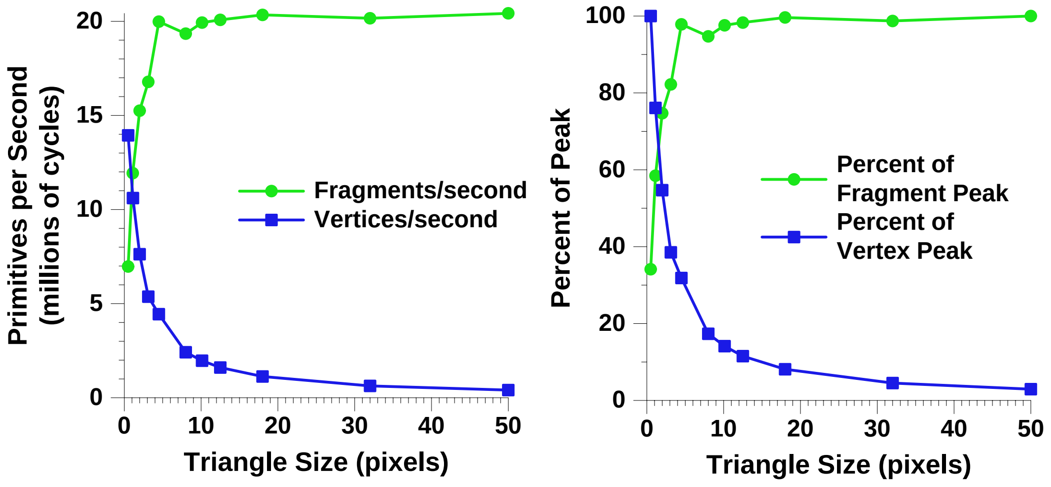


Figure 5.11: Primitives per Second vs. Triangle Size. The left graph shows the fragment and vertex rates as a function of triangle size. The right graph normalizes the peak rates to 100% and plots percentage of peak rates against triangle size.

Performance Effects

The most important component of performance is the sum of the vertex work and the fragment work. The fragment work was identical for all scenes, but the vertex work varies with the size of the triangles. Beginning at scenes with huge triangles and moving to smaller ones, we would expect to see frame rendering time gradually increase as triangles became smaller (corresponding to an increasing cost in per-vertex work), finally increasing sharply as per-vertex work arrives on par with then overtakes per-fragment work.

Indeed, this is what happens. Figure 5.11 shows the details. Our peak fragment rate on RATE is about 20 million fragments per second, and our peak vertex rate is about 14 million vertices per second. One common measure of a graphics system is the triangle size for which it is "balanced," where the fragment rate and the vertex rate make equal contributions to runtime. For our implementation, this point is between 1 and 2 pixel triangles.

All RATE scenes have an identical amount of per-fragment work but a different amount of per-triangle work. Making triangles larger, then, decreases the per-triangle work and increase the overall performance. Figure 5.12 plots frame rate against triangle size.

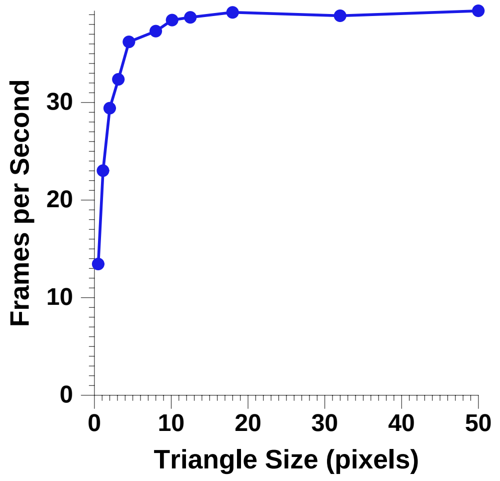


Figure 5.12: Frames per Second vs. Triangle Size. Enlarging triangles past 5 pixels per triangle makes little difference in overall performance.

We also observe a number of second-order effects. The first is the effect of short streams. As explained in Section 5.2.3, processing short streams incurs a loss of efficiency because the fixed costs of running kernels are proportionally costlier with short streams than long ones.

Short stream effects occur in both extremes of triangle sizes: both very small and very large. With small triangles, the data streams associated with the fragment part of the pipeline are short. (This effect is largely hidden by the increasing cost of vertex work, however.) With large triangles, the vertex streams are short, and so we see the largest triangles suffer from some performance degradation when compared to those of medium size.

To achieve a maximum vertex rate, the triangles should be as small as possible; for a maximum fragment rate, as large as possible. However, the combination of inefficiencies with very small and very large average triangle sizes indicate that the ideal triangle size for achieving maximum frame rates on Imagine would lie somewhere between these two extremes. Fortunately, our implementation maintains a relatively constant frame rate over all triangle sizes greater than about 5 pixels.

Figure 5.13 shows that our implementation's sustained operations per second rate is nearly constant across the entire range of RATE triangle sizes. This demonstrates that

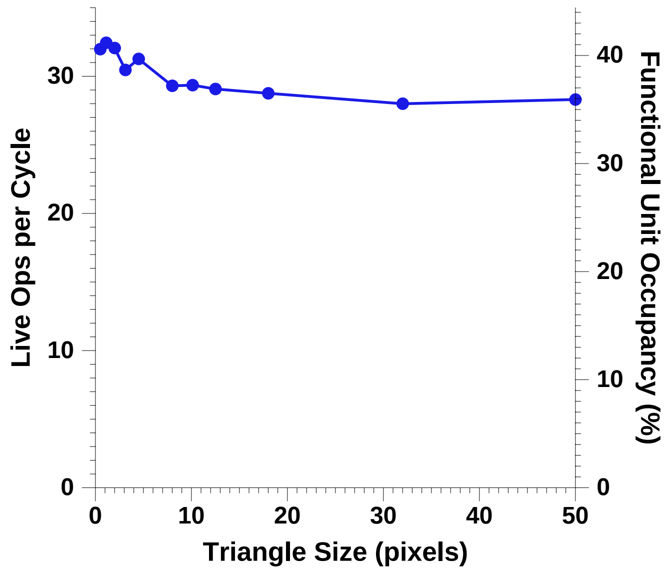


Figure 5.13: Ops per Cycle and Functional Unit Occupancy vs. Triangle Size. Across a range of triangle sizes in the RATE benchmark, our implementation achieves a relatively constant rate of between 28–32 operations per second across all clusters, 35–41% of Imagine's available operation issue slots. These rates translate to 14–16 billion raw operations per second. Of those operations, between 5 and 6 billion are strictly arithmetic or logical operations (25-30% of Imagine's peak of 20 billion 32-bit arithmetic/logical operations per second); the remainder are communication, select, conditional stream, and scratchpad operations.

Imagine's clusters are kept equally busy during the wide range of triangle sizes in these benchmarks. In other words, performance as measured by sustained operation throughput is only barely affected by triangle size. Ultimately, we hope to maintain similarly high operations-per-cycle rates for all scenes, whether or not those scenes achieve high frame rates.

5.2.6 Scanline vs. Barycentric

Our rendering pipeline supports two different rasterizers, a scanline rasterizer (described in Section 4.4.1) and a barycentric rasterizer (described in Section 4.4.2). What are the performance effects of using these rasterizers?

SRF Peak Space Requirements

Section 5.2.3 demonstrated that the best performance is achieved for a given task by making batches as large as possible without spilling intermediate streams from the SRF. The greatest use of SRF space is during rasterization, so in comparing scanline and barycentric rasterizers one of our primary concerns is the amount of SRF space they require during their execution.

To make the comparison, we compute the SRF space required per triangle at each step during the pipeline of each of the two rasterizers. In doing so, we use a simple triangle model. We specify f, the number of fragments per triangle. To derive the number of scanlines for that triangle, we assume the triangle had a square bounding box, making the number of scanlines in that triangle $\sqrt{2f}$. This figure is a conservative estimate for scanlines (tall, thin triangles, for instance, would have many more scanlines), so using this estimate likely underestimates the number of scanlines and favors the scanline rasterizer.

On the other hand, our Imagine implementation of the barycentric rasterizer produces some invalid fragments[7] (as described in Section 4.4.2). These invalid fragments are an artifact of our implementation and are not inherent in the barycentric algorithm. Thus, in our model, we do not account for these invalid fragments, which favors the barycentric rasterizer. Both of these effects are second-order and should not significantly affect the conclusions here.

Our triangle model assumes that each triangle always carries x, y, and homogeneous coordinate w at each vertex. Most triangles carry additional per-vertex information, such as depth, color, and texture; the number of these additional interpolants is parameterized in our model by i. The triangle model does not consider the additional costs of mipmapping, which requires more space in both rasterizers.

The SRF space required for each rasterizer is a function of the number of fragments per triangle f, the number of interpolants i, and the number of scanlines s. Our triangle model relates s to f, and we vary f and i in our experiments.

Figure 5.14 shows the peak SRF space required for both rasterizers over a variety of triangle sizes and interpolant counts. We see that the barycentric rasterizer uses less space

[7]In our scenes with our current implementation, invalid fragments account for roughly 10–20% of all fragments generated by `xyrast`.

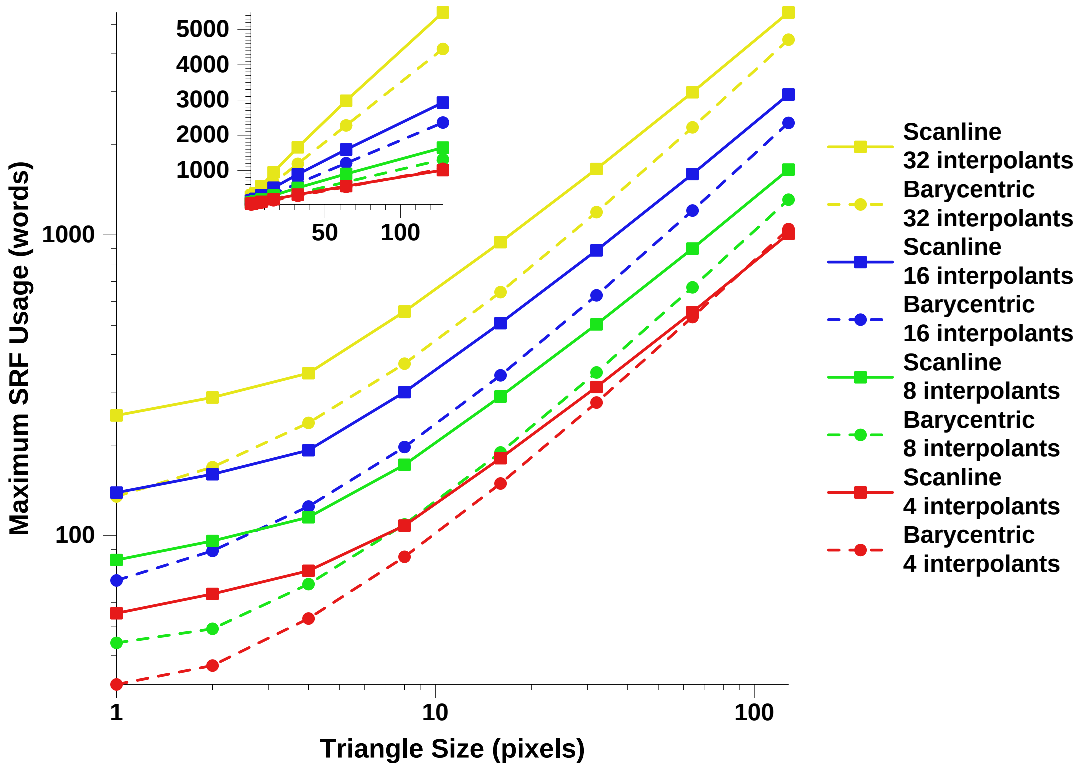


Figure 5.14: Per-Triangle SRF Space Requirements for Scanline and Barycentric Rasterizers. We compare space requirements for a variety of triangle sizes at 4 different interpolant counts for both scanline and barycentric rasterizers. Scanline measurements are marked with boxes and solid lines; barycentric, with circles and dashed lines. The inset graph presents the same data on linear axes.

than the scanline rasterizer at all data points except one[8]. The gap between scanline and barycentric grows with both triangle size and with the number of interpolants.

The peak in SRF demand for each rasterizer is not always in the same place in the pipeline. Table 5.3 presents a simplified model for the peak demand, the condition under which it occurs, and the place in the pipeline where it occurs.

For the scanline rasterizer, the pipeline location where the most SRF space is required depends on triangle size. The usage due to small triangles peaks while the prepped triangle and span streams are both still active. As triangles grow, they have more fragments per triangle, so the point where span and generated fragment streams are active has the highest

[8]This point reflects a small number of interpolants ($i = 4$), and a large triangle ($f = 128$ fragments/triangle).

Rasterizer	Interpolants	Triangle Size	Location	Bound
scanline	—	small	during `spangen`	$(18+4i+(7+2i)s)t$
scanline	—	large	during `spanrast`	$((7+2i)s+(2+i)f)t$
barycentric	small	—	during `baryprep`	$(3i+9+8f)t$
barycentric	large	—	during `irast`	$(3i+5+(2+i)f)t$

Table 5.3: Analytical Model for Peak SRF Usage for Barycentric and Scanline Rasterizers. t indicates the number of triangles; i, the number of interpolants; s, the number of scanlines; and f, the number of fragments.

demand for SRF space.

The pipeline location where the barycentric rasterizer's SRF space requirements is largest is dependent on the number of interpolants. When there are few interpolants, the fixed costs of fragments are more significant, and the peak occurs during barycentric coordinate preparation (the `baryprep` kernel). Barycentric rasterization with many interpolants, on the other hand, uses the most space during interpolant calculation (the `irast` kernel).

Theoretical Op and Cycle Counts

In addition to space requirements, we can also create runtime models for the two rasterizers. We derive these models from the loop lengths and op counts for the scheduled main loops of the rasterizers' kernels. Such a model ignores prologue and epilogue blocks, software-pipeline setup and teardown, and cluster contention, but should provide a fairly accurate estimate of runtime. Both rasterizers have no unrolling (one fragment per loop for the barycentric rasterizer, one scanline per span loop and one fragment per fragment loop for the scanline rasterizer).

Figure 5.15 shows the results of the models for the ADVS-1, SPHERE, and 9I [9] scenes. In practice, the rasterizers' performance is dependent only on the number of interpolants for that scene, so the models for scenes with similar numbers of interpolants differ little. The curves that fit these data are simple: a linear model for the barycentric rasterizer ($c+g(i)f$, where c is constant, f is the number of fragments per triangle and $g(i)$ is a linear function in

[9] The 9I scene is MARBLE with several interpolants removed, leaving 9; its `spanprep` and `spangen` failed register allocation during kernel scheduling but still could provide cycle and op data.

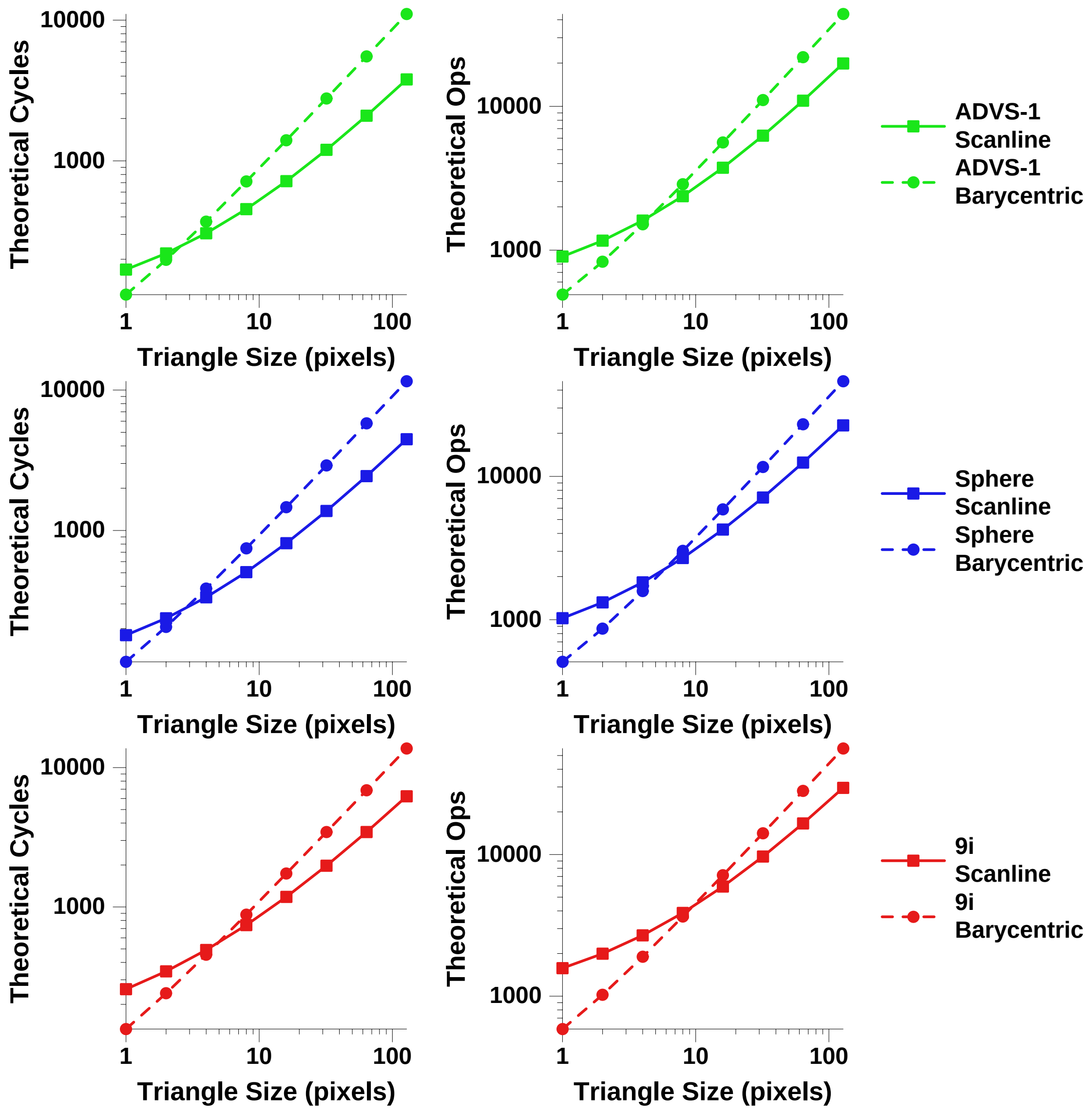


Figure 5.15: Per-Triangle Theoretical Runtime and Op Counts for Scanline and Barycentric Rasterizers. Scanline measurements are marked with boxes and solid lines; barycentric, with circles and dashed lines. Lower lines have fewer cycles/ops. ADVS-1 has 4 interpolants, SPHERE has 5, and 9I has 9.

the number of interpolants) and a linear-plus-square-root model for the scanline rasterizer ($h_1(i) + h_2(i)\sqrt{f} + h_3(i)f$, where $h_n(i)$ is a linear function in the number of interpolants).

This model offers several lessons:

- The barycentric rasterizer has a superior runtime for very small triangles ($\leq$ 2–3 pixels) and the scanline rasterizer is better at larger triangles.

- We model the "cycle count" and the "op count" for each rasterizer. The cycle count is the measure of how many Imagine cycles required by the rasterizer. The op count is how many Imagine operations (per cluster) are required by the rasterizer.

  The point at which both rasterizers have the same cycle count occurs at smaller triangles than the point where both rasterizers have the same op count. This gap indicates that the barycentric rasterizer does not achieve the same operations/cycle efficiency as the scanline rasterizer. The barycentric rasterizer's culprit is `xyrast`, whose 2 raw ops/cycle per cluster is far less than the 5–6 common in other kernels.

- Adding more interpolants pushes the break-even point toward larger triangles, which indicates that the barycentric rasterizer will do comparatively better on scenes with more interpolants than on scenes with fewer.

It would unquestionably be better to compare the models for several other real scenes with more varied numbers of interpolants. Unfortunately, as described in Section 4.4.1, the Imagine kernel scheduler scales poorly to larger numbers of interpolants, primarily because of poor local register allocation. No other Imagine scenes with larger numbers of interpolants than ADVS-1 or SPHERE are scheduleable using the kernel scheduler (and ignoring register allocation [as in 9I] in our model offers increasingly dubious results as the number of interpolants climbs). The barycentric kernels, on the other hand, only depend on the number of interpolants in a single kernel, `irast`, which is easily decomposible into multiple kernels[10]. For this practical reason, our overall scene performance data is generated using the barycentric rasterizer.

[10]The `irast` kernel in PIN-8, for instance, would not schedule using the kernel scheduler, but was easily separable into two `irast` kernels with negligible performance loss.

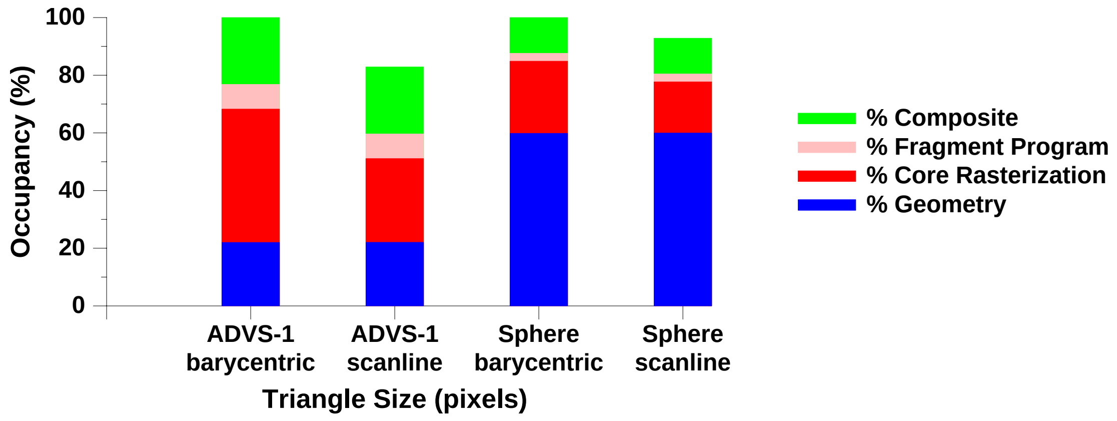


Figure 5.16: Stage Breakdown for Scanline and Barycentric Rasterizers on ADVS-1 and SPHERE. Total cycles are normalized to barycentric = 100%. The rasterization percentage only includes the kernels that are different between the two rasterizers; the fragment program is separate.

Runtime Comparison

Finally, we ran the ADVS-1 and SPHERE scenes through the cycle-accurate Imagine simulator with both scanline and barycentric rasterizers.

Figure 5.16 summarizes the breakdown of runtime for these two scenes. Because the average triangle size for these scenes is larger than the breakeven point for the scene's number of interpolants, we would expect that the scanline rasterizer would demonstrate superior performance. Indeed, this is what happens. The relevant figures are summarized below.

	ADVS-1 Scanline	ADVS-1 Barycentric	SPHERE Scanline	SPHERE Barycentric
Runtime	2,914,615	3,325,975	11,813,332	12,576,589
Rasterizer Runtime	795,768	1,267,194	1,974,065	2,783,481
Triangle Size	5.5	5.5	4.4	4.4

At triangle sizes near the breakpoint, the choice of rasterizer does not make a huge difference in total runtime. ADVS-1, when run with a scanline rasterizer, runs in 83% of the time as with the barycentric rasterizer; SPHERE runs in 93% of the time.

In most of our test scenes, rasterization is the dominant stage in the total runtime. Overall, the rasterization kernels in total maintain similar op rates as other rendering kernels, but their rate still falls far below rasterization rates in special purpose hardware. We explore adding hardware rasterizer support to our stream architecture in Section 6.6.

5.3 Kernel Characterization

Future stream processors will contain more clusters, more functional units per cluster, and more bandwidth into and out of the clusters. They will also have larger stream register files to reduce short stream effects. Chapter 6 explores some of the effects of extending Imagine.

How will these changes affect the kernels in the current pipeline? Table 5.4 summarizes how each kernel scales in performance with input size and what the hardware limit is to its performance. By "scales with input size," we mean that as the input stream length increases as n, how is the runtime of the main loop affected?

Of note is the lack of a single bottleneck for the majority of kernels. They are limited in performance by many factors, none of them dominant. Imagine would seem to be a well-balanced machine for this pipeline because the computational bottlenecks are not concentrated in any specific part of the hardware.

All kernels scale as $O(n)$ or better except for the fragment sorting step in the composition stage (in particular, the `merge` kernel). With current batch sizes, the sort/merge is only a small part of the total runtime, so despite the $n \log n$ dependence of `merge`, the performance of the pipeline is currently practically maximized on Imagine by making the batches as large as possible without spilling.

However, future generations of stream processors might have the ability to handle significantly larger batch sizes. Though a larger hash table in `hash` would mitigate the $n \log n$ cost to some extent by eliminating false matches, it is possible that the cost of the sort would eventually be the dominant performance factor for batches of a sufficiently large size. In this case the best batch size for our pipeline would be found by balancing the short-stream effects of small batches against the cost of the sort for large batches.

The kernels can be grouped by their performance limits:

Kernel	Performance scaling as input size n	Hardware limit to performance
perbegin program	$O(1)$	Does not affect performance
vertex program	$O(n)$	Insufficient functional units
assemble	$O(n)$	Conditional stream bandwidth[a]
clip	$O(n)$	Conditional stream bandwidth[a]
viewport	$O(n)$	Divide latency[b]
backfacecull	$O(n)$	Conditional stream bandwidth[b]
xyprep	$O(n)$	Insufficient functional units
xyrast	$O(n)$	Control complexity of algorithm, also long critical path
baryprep	$O(n)$	Long critical path
irast	$O(n)$	Conditional stream bandwidth
filter8	$O(n)$	Insufficient multipliers
fragment program	$O(n)$	Insufficient functional units
hash	$O(n)$	Scratchpad bandwidth, also insufficient functional units (logical ops)
sort	$O(n)$	Intercluster communication bandwidth
merge	$O(n \log n)$	Intercluster communication bandwidth
compact/recycle	$O(n)$	Long critical path
zcompare	$O(n)$	Long critical path

[a]Because assemble and clip are both bound by conditional stream bandwidth and can be easily composited, we combine them into a single kernel, one which is still conditional-stream bandwidth bound.

[b]Viewport and backfacecull are also easily combined; the resulting kernel is conditional-stream bandwidth bound.

Table 5.4: Kernel Characterization. Each kernel in the pipeline is described by how its performance scales with input size and the Imagine hardware limit to its performance.

Insufficient functional units `vertex_program`, `fragment_program`, and `xyprep` could run faster if given more functional units per cluster. These kernels have a large amount of instruction-level parallelism in their computation. `filter8` would also benefit, particularly from more multipliers. `hash` is balanced between limited scratchpad bandwidth and limited logical operation issue slots and would run faster if both were increased.

Conditional stream bandwidth Few Imagine kernels are limited by the bandwidth of unconditional streams (four words per cluster per cycle), but kernels that input and output large records conditionally suffer from Imagine's one word per cluster per cycle conditional stream bandwidth. In our pipeline, `assemble`, `clip`, `backfacecull`, `assemble_clip`, `viewport_cull`, and `irast` each deal with large records (triangles, except for `irast`, which processes fragments). These kernels are all limited by conditional stream bandwidth related to processing these large records. Given Imagine's conditional stream implementation, increasing conditional stream bandwidth requires increasing both intercluster communication bandwidth and scratchpad bandwidth (by increasing individual scratchpad bandwidth or more likely, scratchpad count).

Divide/square root unit Vertex programs often perform many vertex normalizations (for example, SPHERE), which are limited by divide/square root performance. Our divide/square root unit is not fully pipelined, so only two divide or square root operations can be in flight at the same time. Also, normalizations would benefit from a reciprocal-square-root operation, which would cut the required number of operations in half. The `viewport` kernel, which performs 3 divides, is limited by divide performance—having either a faster divider or one more deeply pipelined would aid this kernel's inner loop time.

Intercluster communication bandwidth The sort kernels, `sort` and `merge`, communicate heavily between clusters and would benefit from increased intercluster communication bandwidth. Imagine implements conditional streams over the intercluster communication paths, so if conditional stream bandwidth were increased, these kernels' performance would also improve.

Long critical path These kernels are particularly interesting, because their performance would not increase with additional hardware. Thus, they are candidates for improved algorithms as stream hardware evolves and improves. We look at them individually:

- `compact_recycle` and `zcompare` are simple and short kernels, merely separating a single stream into two streams. They have little computation per loop and are close to being limited by conditional stream bandwidth, so would benefit little from new algorithms.
- More interesting is `baryprep`. This kernel must conditionally input per-triangle coefficients while inputting next-fragment data. Then it calculates the barycentric coefficients for each fragment and outputs the fragment. The computation is relatively sequential so the critical path is lengthy. Combining this kernel with `irast` would put more strain on the conditional stream bandwidth (and much more strain on the kernel scheduler) but would probably be the best direction for future stream architectures.
- `xyrast` has been the most difficult kernel to optimize. It has three separate threads of execution and a long critical path. Its functional units have a much smaller occupancy than other rasterization kernels. It is the only kernel in the pipeline that would substantially benefit from MIMD clusters as opposed to our SIMD organization. A better triangle-to-fragment generation algorithm would dramatically help this kernel's performance; increasing cluster bandwidth or operations would not. As other cluster bottlenecks are removed with an increase in bandwidth or operations, this kernel will consume an ever greater amount of runtime. Special purpose rasterization hardware would seem to be particularly useful in replacing this kernel.

5.4 Theoretical Bandwidth Requirements

On commercial graphics architectures, applications are becoming increasingly bandwidth-bound [Igehy et al., 1998b]. If our system was constrained only by memory bandwidth, and not by any computation, how fast could it run?

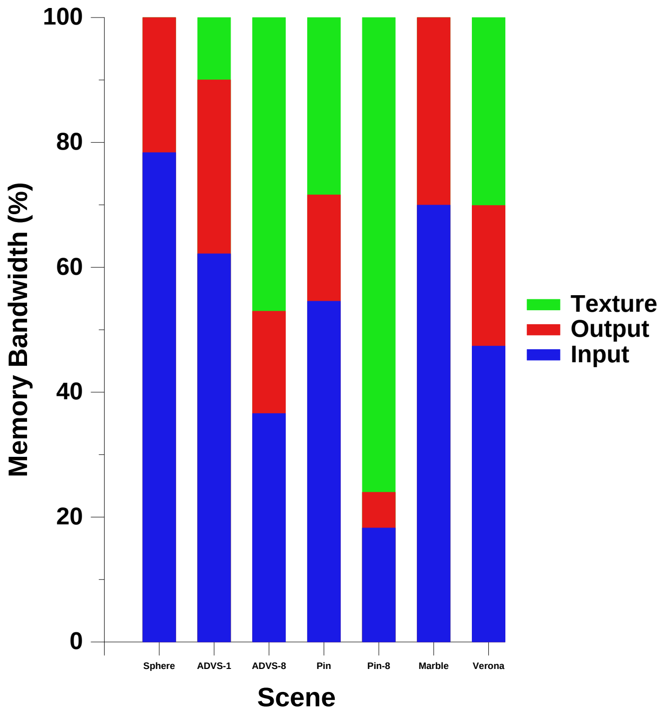


Figure 5.17: Test Scenes: Memory Bandwidth Sources. For each scene, the memory demand is divided into input bandwidth requirements (vertex data), output bandwidth requirements (color and depth buffer accesses), and texture accesses.

5.4.1 Theoretical Analysis

In this analysis we assume that the color buffer and depth buffer as well as textures are located in external (off-chip) memory. Here we are concerned with external memory system bandwidth, not local bandwidth on chip. Local bandwidth has historically scaled faster than off-chip bandwidth (ignoring clock rate and concentrating on data channel count only, the number of tracks on a chip [on-chip bandwidth] increases at 22% per year and the number of pins on a chip [off-chip bandwidth] only increases at 11% per year [Dally and Poulton, 1998]); furthermore, local bandwidth use is highly implementation dependent.

Let us call the memory system bandwidth in bytes per second M, the number of byte traffic required per pixel b, the depth complexity d, and the number of pixels per frame S. Then the maximum frame rate of the system in frames per second is $M/(bdS)$.

In the calculations in this section, we consider a simplified bandwidth model. We will

Work per pixel	bytes/pixel	fps @ 500k pixels	fps @ 1M pixels
Color write	4	500	250
Color write, z compare	12	167	83
Color write, z compare, single-sampled texture	16	125	62
Color write, z compare, mipmapped texture	44	11	22

Table 5.5: Theoretical Performance Limits for Imagine. The task in the first column corresponds to the amount of memory traffic required in bytes per pixel in the second column, which in turn leads to the frames-per-second maxima in the third and fourth columns for 500,000 and 1,000,000 pixel windows. These figures assume a memory system that can deliver 1 GB/s; they scale linearly with memory bandwidth.

assume 1 GB/s of achievable memory bandwidth and a depth complexity of 1. Also, we will neglect the bandwidth required to input the vertex data.

Each pixel in our pipeline is depth buffered, requiring at a minimum 4 bytes of depth read, 4 bytes of depth write, and 4 bytes of color write—a total of 12 bytes per pixel. Thus, we could fill 83 million pixels per second. Using our 720 $\times$ 720 screen size of about 500,000 pixels, we could achieve a frame rate of 167 frames per second. A million-pixel display would run at half that speed, 83 frames per second.

Adding a single 32-bit texture lookup per pixel adds another 4 bytes per pixel, cutting frame rates for the half-million and million pixel displays to 125 and 62 frames per second, respectively. And doing 8 4-byte texture lookups per pixel instead, as in mipmapped textures, yields frame rates of 45 and 22.5 frames per second[13]. Table 5.5 summarizes the theoretical performance limits for several per-pixel workloads.

5.4.2 Achieved Results

The theoretical peak memory bandwidth of any modern memory system is rarely achieved in practice on any real dataset because all accesses cannot be pipelined as the access pattern

[13]All of these calculations neglect the cost of clearing the color and depth buffer for each frame, which adds another 8 bytes per pixel. Special-purpose fast-clear hardware, as is common in commercial graphics systems, could mitigate the clear's bandwidth requirements.

	RATE	ADVS-1	PIN-8
Vertices/second	630k	9.41M	1.87M
Bytes/vertex	32	28	44
Achieved input bandwidth	20.2 MB/s	263 MB/s	82.3 MB/s
Fragments/second	20.16M	10.58M	2.14M
Bytes/fragment	12	16	172
Achieved output bandwidth	242 MB/s	160 MB/s	368 MB/s
Total bandwidth	262 MB/s	424 MB/s	451 MB/s
Measured trace bandwidth	1.48 GB/s	1.26 GB/s	1.80 GB/s
Percent of trace bandwidth achieved	17.7%	33.6%	25.0%

Table 5.6: Achieved vs. Trace Bandwidth. RATE reads in position and color information for each vertex and does a color write and depth read and write per fragment. ADVS-1 has position and texture information for each vertex and performs a single texture lookup, color write, and depth read and write for each fragment. PIN-8 has position, normal, and texture coordinates per vertex and combines 5 mipmapped texture lookups with specular and diffuse light components per fragment.

switches between banks and rows within a bank. On a simulated Imagine 2 GB/s memory system, Rixner et al. studied memory traces of a variety of media applications [Rixner et al., 2000a]. They found sustained memory bandwidths on these traces between 1.4 and 1.8 GB/s. The access patterns of graphics applications are less predictable than most of these applications and so their percentage of sustainable bandwidth would tend to be lower.

Running a memory trace of ADVS-1, in which each pixel is depth-buffered and textured with a single texel, results in a maximum achievable bandwidth of 1.26 GB/s. A RATE trace[14], in which each pixel is only depth-buffered, has a slightly more regular access pattern and hence a higher achievable bandwidth of 1.48 GB/s. The texture locality of PIN-8 allows it to achieve 1.80 GB/s on its trace.

Our implementation achieves considerably less than these peak bandwidth totals. Table 5.6 compares three scenes' achieved bandwidth against the maximum achievable bandwidth of the memory access pattern when run as a memory trace. ADVS-1 only reaches 33.6% of the achievable bandwidth, RATE does not even attain 18%, and PIN-8 manages to achieve 25%.

This behavior is characteristic of all the scenes we have studied: none of them are

[14]This particular RATE trace has 32 pixels/triangle, so input vertex bandwidth is relatively modest.

bandwidth-limited on our rendering pipeline. Short-stream effects are not responsible (RATE at 32 pixels/triangle loses less than 2% of its cycles to short-stream effects, ADVS-1 just above 7%; PIN-8, with 32-vertex batches, loses about 25% of its cycles). The answer is simply that Imagine does not do enough computation per cycle. We must consider extensions to Imagine that increase the computation throughput. This goal is the topic of the next section.

Chapter 6

Extending Imagine

In the previous chapter, we showed that our implementation was not limited by memory bandwidth but instead by computation. Future generations of stream hardware will have more computational capabilities than Imagine: more clusters, more capabilities in each cluster, larger amounts of storage, and perhaps special-purpose hardware dedicated to performing certain tasks. In this chapter we investigate the performance impact of extending Imagine: how can additional hardware aid the performance of the rendering pipeline?

6.1 Adding Instruction-Level Parallelism

The analysis in Chapter 5.3 revealed that many kernels are limited in performance by the limited amount of hardware in each arithmetic cluster and the lack of communication resources between clusters. Those kernels can be grouped into the following categories:

1. Insufficient functional units;

2. Conditional stream bandwidth;

3. Divide/square root performance;

4. Intercluster communication bandwidth.

In this section we explore expanding the clusters along two axes. The first axis is more functional unit performance within a cluster, addressing points (1) and (3) above. The second axis is communication resources between clusters, addressing (2) and (4).

To increase the functional unit performance within a cluster, we make three major changes. First, we double the number of adders (which execute integer and floating point adds and subtracts and also perform all logical operations) from 3 to 6. Second, we double the number of multipliers from 2 to 4. Third, we fully pipeline the divide/square root unit. The outputs of all units remain connected to all register files with a full crossbar switch. All register file sizes remain the same except the condition code register file, which must be doubled in size to cope with the near-doubling in functional unit count. We designate these changes with “`+F`” (for additional functional units).

To increase the communication resources between clusters, each cluster is equipped with three communication units and three scratchpad memory files. (The number three is chosen because increases in conditional stream bandwidth benefit those kernels that deal with triangles most of all. The three vertices of each triangle in our kernels are specified as three separate streams.) These changes triple the intercluster communication bandwidth and the conditional stream bandwidth. The `hash` kernel would likely also benefit from additional scratchpad bandwidth if rewritten, but this was not done for these experiments. Like the +F configuration, all functional units in this configuration remain fully switched. We designate these changes with “`+C`” (for additional communication resources).

Several costs are associated with increasing cluster size. First, the switch that connects functional units and register files must increase in size; the switch size increases with the number of functional units n as $O(n^2)$. It would be possible to use a sparsely connected switch to connect functional units, but the determination of proper connections would require significant investigation, and the kernel scheduler does not handle sparse switches as well as dense ones.

Also, the size of the microcode store increases linearly with the number of units. Larger clusters may require longer delays in sending signals across the clusters. Finally, adding intercluster communication requires additional (or wider) intercluster switches. These hardware costs, which potentially impact cycle time, are not modeled. In our experiments we

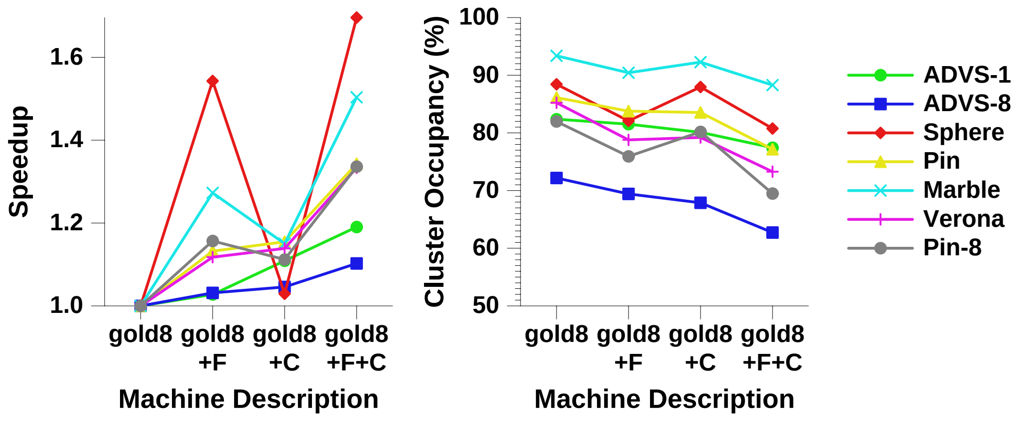


Figure 6.1: Performance for Scenes and Cluster Occupancy on Different Cluster Configurations. Each single line indicates the relative performance for a single scene on each of 4 different machine descriptions. The left graph indicates the overall performance speedup, normalized to `gold8` = 1. The right graph indicates the percentage of time the clusters are busy for each scene and cluster configuration.

assume that larger clusters can be run at the same clock rates as the baseline Imagine clusters.

6.1.1 Results

To test the effects of different cluster configurations, we ran each of our scenes with four different machine descriptions. The base description, mirroring the Imagine hardware, is termed "`gold8`". We alter gold8 in three ways, `gold8+F`, `gold8+C`, and `gold8+F+C`, and run our scenes on each of them.

Performance and Cluster Occupancy First, does adding hardware to the clusters produce gains in performance? Figure 6.1 shows that the new cluster configurations produce performance gains on all scenes. Some scenes benefit more from additional functional units, some from more communication resources. We examine this further below.

We also examine the effects of more complex clusters on cluster occupancy. We see that more complex clusters slightly decreases the achieved cluster occupancy compared to the base case of Imagine's cluster configuration.

As overall runtime decreases because kernels run faster, memory demand remains constant for the whole scene, so the memory system must deliver higher bandwidth. This, in turn, increases memory system occupancy. Software pipelining at the stream level with the goal of achieving high cluster occupancy becomes more difficult with higher memory system occupancy.

Fortunately, the gains associated with more cluster resources are considerably more significant than the losses from reduced cluster occupancy.

Stage Breakdown Figure 6.2 shows the stage breakdown of the kernel runtime for each scene for each cluster configuration. This figure shows kernel runtime only and does not consider the cluster occupancy differences between the different machine descriptions.

We can draw two major conclusions from the stage breakdown. First, adding more functional units (`+F`) and adding more communication resources (`+C`) are orthogonal, so their effects are additive. In other words, the benefits from adding one are different than the benefits from adding the other, and adding both combines the benefits of both.

Second, different scenes benefit in different ways from the cluster configuration changes. For some scenes, adding more functional units produces a larger performance gain; for some scenes, adding more communication resources is a bigger win.

Broadly speaking, scenes that spend a large proportion of their time in the vertex or fragment programs (SPHERE and MARBLE in particular) benefit more from additional functional units. Their complex vertex/fragment programs exhibit ample instruction-level parallelism and can use the additional functional units efficiently.

Scenes with simpler vertex and fragment programs benefit more from additional communication resources, which speed up geometry work (assembly and culling) and the latter half of the core rasterization work (barycentric weight calculation and interpolant rasterization).

As shaders continue to increase in complexity, particularly in computation (as opposed to texture lookups), adding more functional units will continue to increase performance. Even with the `+F` configurations, the vertex program on SPHERE and the noise kernel on MARBLE have even more instruction-level parallelism than can be exploited by the number of functional units and would benefit from even more.

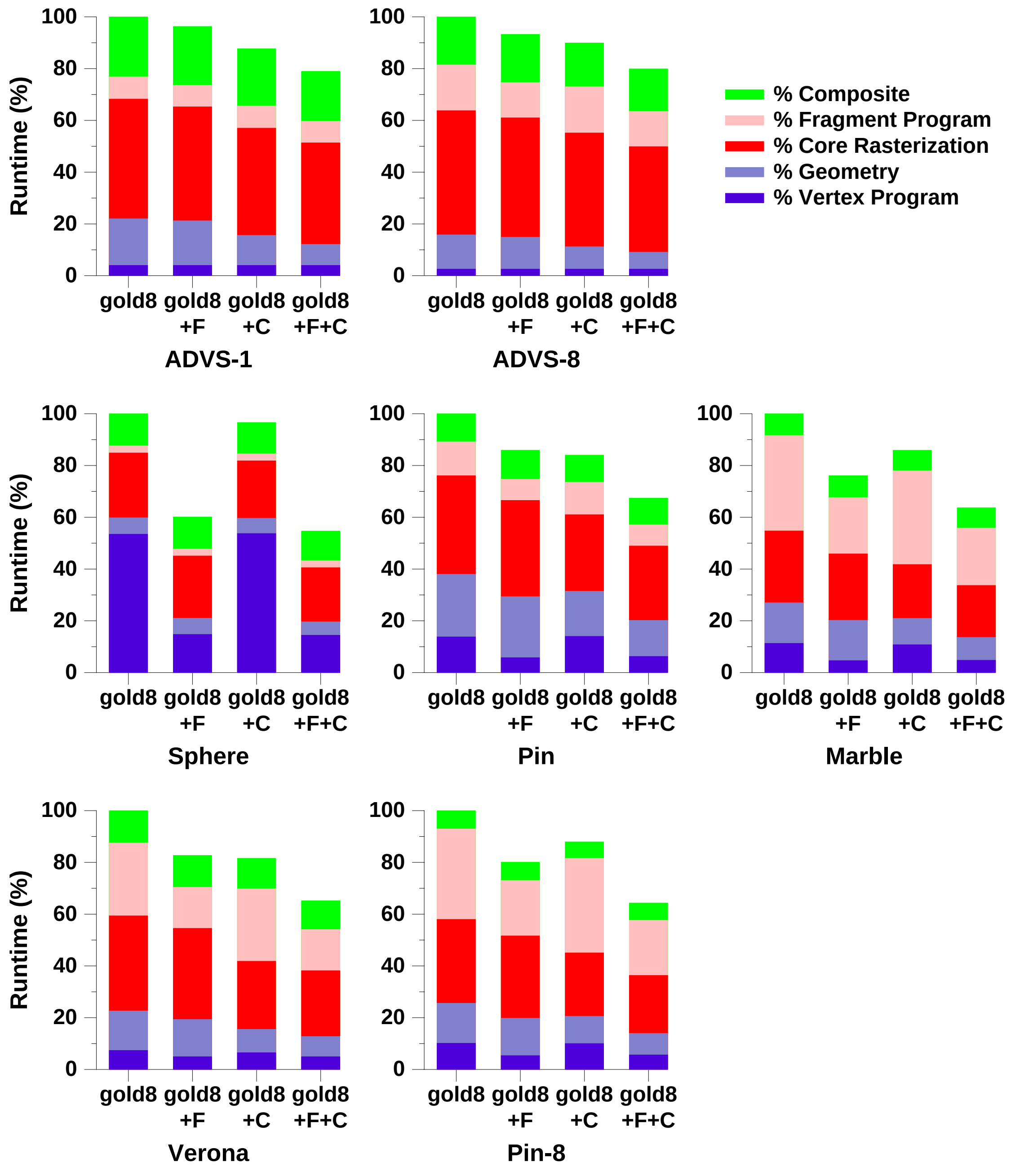


Figure 6.2: Stage Breakdown for Scenes on Different Cluster Configurations. Total cycles are normalized to `gold8` = 100%. Cluster idle time is not considered in these graphs. The vertex program is treated separately from the rest of the geometry stages, and the fragment program is treated separately from the rest of the rasterization stages.

Scene	Batch Size, 32K SRF	Batch Size, 128K SRF
SPHERE	480	1920
ADVS-1	288	2048
ADVS-8	48	800
PIN	120	840
PIN-8	32	400
MARBLE	112	720
VERONA	40	432

Table 6.1: Batch Sizes in Vertices for Larger SRF. Imagine has a 32K SRF; we show batch sizes for both Imagine's SRF and a SRF four times the size of Imagine's.

On the other hand, simple fragment and vertex programs, even with software pipelined loops, are already limited by their critical paths and do not benefit from additional hardware. Other kernels exhibit the same behavior. To speed up these kernels requires rethinking the algorithm, either rewriting it to have a shorter critical path or by running multiple data elements on the same cluster at the same time.

6.2 Increasing Stream Lengths with a Larger Stream Register File

Section 5.2.3 discussed the performance impact of short streams. By making the stream register file larger, intermediate streams can be longer. And with longer streams, we can mitigate the short stream effects, spending more time in main-loop code and less in prologue-epilogue and setup-teardown.

The pipelines run in this section are identical to the ones presented in Section 5.1 except for their batch sizes. All batches are sized so their intermediate results fit in the SRF.

We begin by analyzing the effects of a larger stream register file on batch sizes. Not surprisingly, a larger SRF results in larger batches. Surprisingly, for several scenes, multiplying the SRF size by 4 produces a batch size multiplier considerably larger than 4, as shown in Table 6.1. (For example, ADVS-8 goes from 48 vertices per batch with a 32K SRF to 800 with a 128K SRF.)

This effect is due to the variance in the datasets. Some portions of the datasets produce

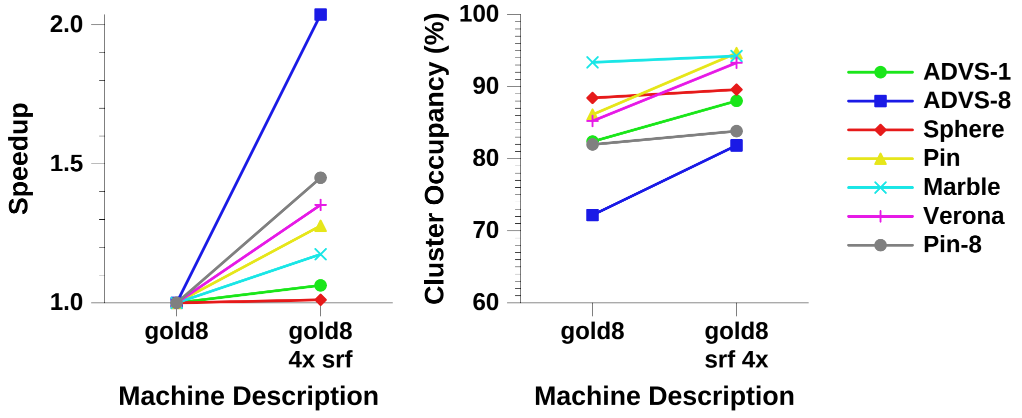


Figure 6.3: Performance for Scenes and Cluster Occupancy with Different SRF Sizes. Each single line indicates the relative performance for a single scene on each of 3 different cluster counts. The left graph indicates the overall performance speedup, normalized to `gold8` = 1. The right graph indicates the percentage of time the clusters are busy for each scene and cluster configuration.

many more fragments than other parts. As the batches become larger, more primitives are sampled and the variance decreases. In other words, the portions of the dataset that produce many fragments are averaged in with portions that produce few and the result produces a fragment count closer to the overall average.

How do the larger batch sizes translate into performance? Figure 6.3 shows a varying range of improvements in all of the scenes. In general, scenes that suffered heavily from short-stream effects (such as ADVS-8 and PIN-8) have substantial speedups—ADVS-8 runs twice as fast with the larger stream register file. On the other hand, scenes that were affected only slightly by short stream effects show only modest performance improvements.

One source of performance improvements is the reduction of short stream effects; closely related is the improvement in cluster occupancy. Figure 6.3 demonstrates that increasing stream size by growing the SRF produces gains in cluster occupancy. Longer streams avoid kernel dispatch penalties due to the minimum kernel duration as well as mitigate the difficulties of pipelining short streams at the stream level.

Overall performance, then, is improved by enlarging the SRF. We now take a closer look at the kernel speedups resulting from this change. Figure 6.4 shows the effects on kernel runtime of increasing SRF size.

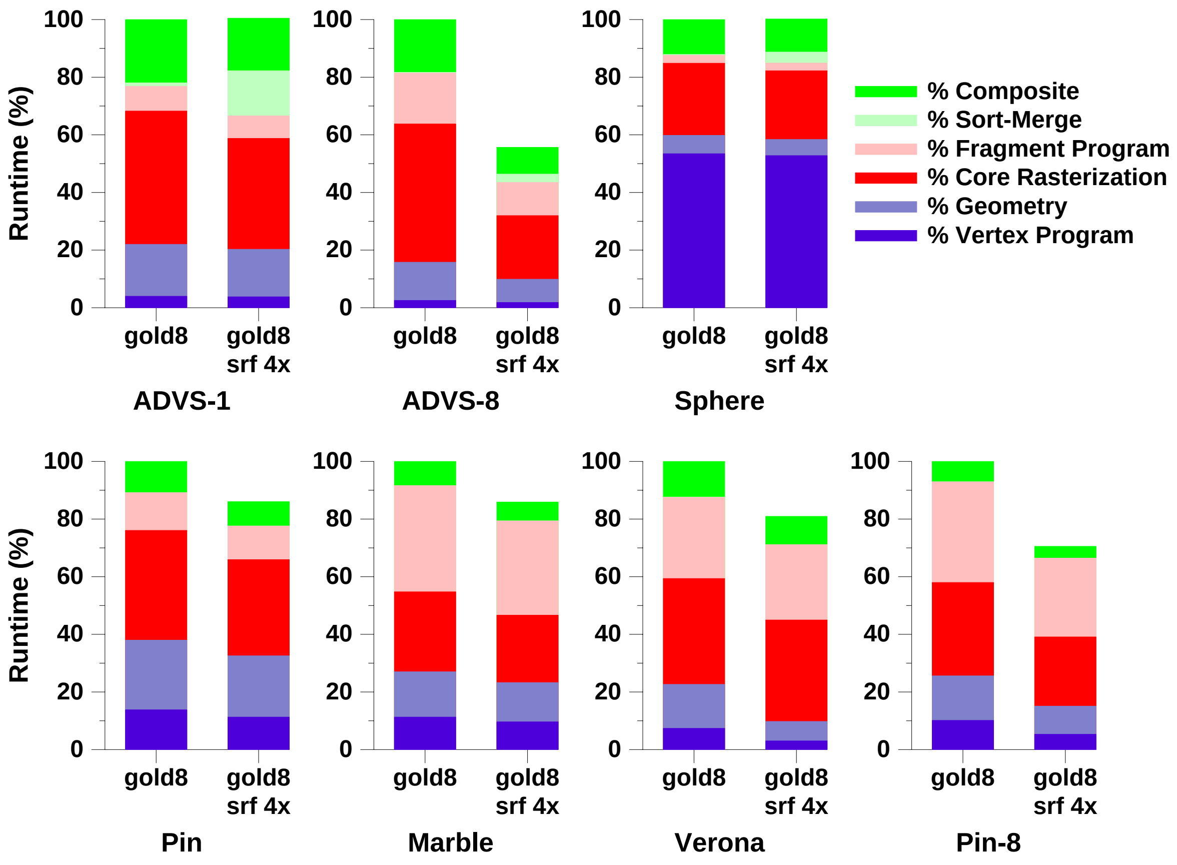


Figure 6.4: Stage Breakdown for Scenes with Different SRF Sizes. Total cycles are normalized to `gold8` = 100%. Cluster idle time is not considered in these graphs. As in the other comparisons, the vertex program is treated separately from the rest of the geometry stages, and the fragment program is treated separately from the rest of the rasterization stages. To emphasize the increasing cost of the support, in this comparison, the `sort` and `merge` kernels are separated from the rest of the composite stage.

The relative kernel runtimes demonstrate the short stream effects of prologue-epilogue and setup-teardown time. Scenes that suffered heavily from short stream effects (such as `Advs-8` and `Pin-8`) show a large speedup in all their kernels. Scenes that did not (such as `Sphere`) show virtually no improvement.

However, we also see a second effect, that of the cost of the sort. Recall from Section 5.3 that though the cost of the sort when running on unmodified Imagine hardware was quite small, unlike the other stages, the sort does not scale linearly with the input size n but instead as $n \log n$. With three of the datasets—ADVS-8, SPHERE, and especially ADVS-1—we see that the cost of the sort is significantly larger with the larger SRF. For ADVS-1 and SPHERE, the total kernel time is actually larger with the larger SRF (though performance is better because of the increase in cluster occupancy). Certainly for at least ADVS-1, the optimal batch size for performance is not the largest batch size for which the intermediate results fit in the larger SRF, but instead the (smaller) batch size for which the sum of the costs of short stream effects and the cost of the sort are minimized.

6.3 Adding Data-Level Parallelism

Chapter 5.2.4 explored the speedup of Imagine on rendering scenes from one to eight clusters. We found that the speedup was quite respectable, and so in this section we explore larger cluster sizes: 16 and 32. If the implementation is truly data-parallel, the resulting performance will approach the ideal speedup as cluster sizes increase.

6.3.1 Algorithm Scalability

To continue to achieve speedups with more clusters, our algorithms must scale with the number of clusters. Most of our kernels (given long streams as inputs) scale trivially with the number of clusters.

The two exceptions are the `hash` and `sort/merge` kernels.

- `Hash` checks each of its hash values against its distributed hash table with a broadcast across the clusters for each value. Thus, as the number of clusters grows, the number of broadcasts does not change, and the kernel does not improve in performance.

At 8 clusters, the cost of the broadcast is similar to the cost of the arithmetic, so the broadcasts do not incur a performance penalty. However, at 16 and higher clusters, the broadcasts dominate the kernel cost.

To remedy this non-scalability, `hash` was rewritten for 16 or more clusters. In this implementation, the hash table is not fully distributed across the clusters but instead replicated across neighborhoods of 8 clusters. Hash values are then broadcast only across the local neighborhoods, with all neighborhoods processed in parallel.

Between the two hash stages, the neighborhoods must combine their hash tables. This cost is proportional to the size of the hash table ($O(h)$), and to the logarithm of the number of clusters ($O(\log c)$) but is constant time with respect to the size of the input stream. In practice, it accounts for only a small portion of `hash`'s runtime for any input stream of reasonable size.

- The `sort` and `merge` kernels grow in complexity with the logarithm of the number of clusters.

  Longer input streams mean more conflicts between fragment addresses, so the sort stage consumes proportionally more time as the number of clusters grows. However, the total amount of time spent on sorting for any of our tests is modest.

Short-stream effects become a greater factor if the number of clusters increases while the stream lengths do not, because the number of iterations required to process a stream of a fixed length is inversely proportional to the number of clusters. Thus, the size of the stream register file must scale with the number of clusters to keep the short stream effects at the same cost. In our experiments, we scale the size of the SRF proportionally to the number of clusters.

6.3.2 Hardware Scalability

Because Imagine's clusters are controlled by a single instruction stream, adding additional clusters is conceptually straightforward. However, several hardware effects must be considered.

Perhaps the most important is the size of the intercluster switch, which grows as $O(n^2)$ with the number of clusters n. With a very large number of clusters, fully interconnected clusters would be impractical, but for smaller numbers (at least up to 128), the switch would be of significant size but still feasible[1].

Another important effect results from the microcontroller driving signals to more clusters and presumably also driving signals longer distances. Finally, 8 clusters are simply laid out in a straight line, but additional clusters may not lend themselves well to such a straightforward layout.

One Imagine-specific effect is the representation of intercluster communication patterns. On an 8-cluster Imagine, each cluster's destination can be encoded in 4 bits (3 bits for which cluster, 1 bit for the microcontroller), so the entire communication pattern can be encoded into a single 32-bit word. As the number of clusters increases, these patterns no longer fit into a single word. In our tests below, on 16 or 32 cluster machines, we avoid the difficulties of representing an entire communication pattern by not using the instruction that uses them (the `commucperm` instruction). Instead we compute each cluster's pattern locally in each cluster (with the `commclperm` instruction). In practice this only requires changes in the `sort32frag` and `mergefrag` kernels, which must be rewritten for each different cluster count anyway.

6.3.3 Experiments

In our experiments to test the effect of greater data-level parallelism, we compared the results with 8 clusters against simulations against 16- and 32-cluster Imagine machines, termed `gold16` and `gold32`. The machine descriptions for these wider machines are identical to the base configuration, `gold8`, with the following changes.

- The SRF size scales with the number of clusters. `gold16` uses a 64k SRF and `gold32` has a 128k SRF.

[1]For Imagine, Khailany et al. show that with C clusters, the intercluster switch grows as $O(C^2)$ with a linear layout and $O(C^2 + C^{3/2})$ with a grid layout. For reasonable numbers of clusters (at least up to 128), the $O(C^{3/2})$ term is dominant and constructing the switch is feasible [Khailany et al., 2003].

Scene	Batch Size, 8 Clusters	Batch Size, 16 Clusters	Batch Size, 32 Clusters
SPHERE	480	960	1920
ADVS-1	288	768	2048
ADVS-8	48	96	296
PIN	120	288	832
PIN-8	32	64	384
MARBLE	112	368	736
VERONA	40	96	416

Table 6.2: Batch Sizes in Vertices for More Clusters. To maintain a comparable number of iterations across cluster count tests, the SRF must grow proportionately to the number of clusters. Consequently, the batch sizes also grow with the number of clusters.

- Due to difficulties with the kernel scheduler register allocator, for the `sort` kernel only, the size of the cluster register files are increased in `gold32`.

First, we must adjust batch sizes to match the new, larger SRF sizes. These new batch sizes, shown in Table 6.2, are similar to those in Section 6.2, and an analysis of them would follow the discussion there[2].

Ideally, increasing the number of clusters would lead to a proportional increase in performance. Most of the scenes approach this speedup; MARBLE and PIN-1 are very close to the ideal.

We look closer at the reasons behind the performance increase by looking at kernel runtimes in Figure 6.6. Again ideally, each kernel should decrease in runtime in inverse proportion to the number of clusters. The geometry and rasterization kernels in each scene mirror this decrease; while the composition stage does decrease in runtime with more clusters, it does not decrease at the same rate because of the $O(n \log n)$ cost of the sort with the increasing number of clusters. Still, the overall kernel runtimes match the expected ideal speedup well.

In these experiments with increased cluster count, batch sizes are increased proportionally to the number of clusters, so the number of iterations should remain constant across

[2]The batch sizes found when increasing cluster count and SRF size are slightly smaller than those found when only increasing SRF size because of quantization effects relating to the requirement that all streams be multiples of the number of clusters.

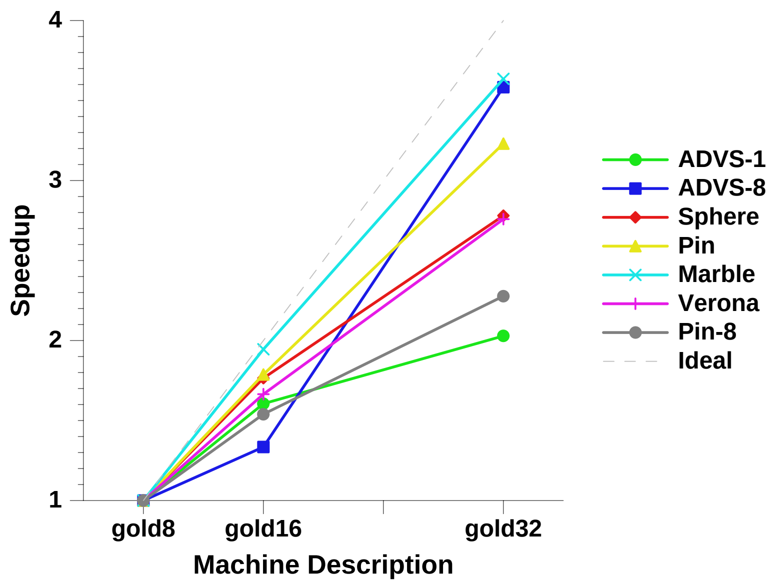


Figure 6.5: Performance for Scenes with Different Cluster Counts. Each single line indicates the overall performance speedup, normalized to `gold8` = 1.

the different cluster counts. Thus the effects of short streams should be constant across the different cluster counts as well, and short stream effects are not responsible for the non-ideality of the performance curves in Figure 6.5.

Instead, the reason the overall performance curves do not track this ideal kernel speedup is shown in Figure 6.7. The cluster occupancy decreases with an increased number of clusters. Again, this is not a result of short stream effects. The culprit is increased memory demand. Because the memory system does not change across the different cluster counts, it is more heavily loaded by more clusters than fewer clusters. And when memory system occupancy increases, cluster occupancy decreases; pipelining at the stream level is a more difficult problem with a more heavily utilized memory system. The increase in memory system demand leads to lower cluster occupancy, which directly leads to losses in performance.

The non-ideality in the performance speedups of Figure 6.5 is largely attributable to the decrease in cluster occupancy. Scenes which are farthest from the ideal (such as ADVS-1 and PIN-8) have the largest decreases in occupancy, while MARBLE has the largest speedup and the smallest occupancy decrease.

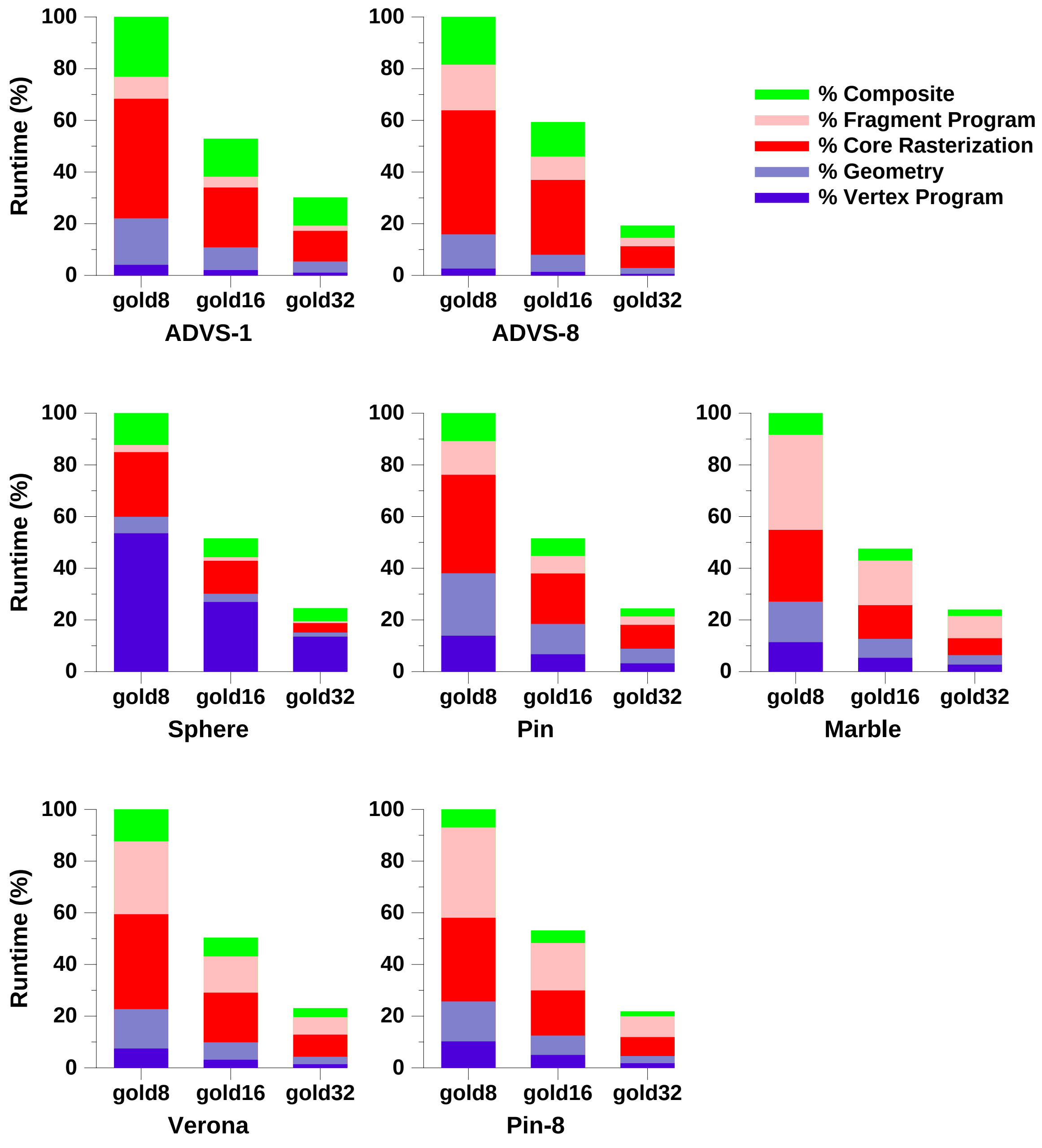


Figure 6.6: Stage Breakdown for Scenes with Different Cluster Counts. Total cycles are normalized to `gold8` = 100%. Cluster idle time is not considered in these graphs. The vertex program is treated separately from the rest of the geometry stages, and the fragment program is treated separately from the rest of the rasterization stages.

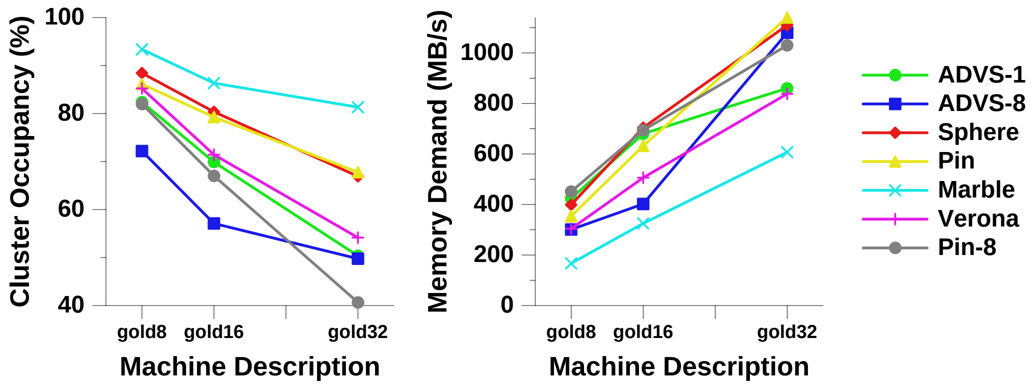


Figure 6.7: Cluster Occupancy and Delivered Memory Bandwidth for Scenes with Different Cluster Counts. Each single line indicates the relative performance for a single scene on each of 3 different cluster counts. The left graph indicates the percentage of time the clusters are busy for each scene and cluster configuration, and the right graph indicates the delivered memory system bandwidth for that scene and cluster configuration.

Still, each scene does realize a significant performance gain from adding clusters, with some scenes approaching the ideal speedup.

6.4 Combinations

Finally, we would like to test the orthogonality of the changes in the previous sections—more functional units per cluster, more clusters, and a larger SRF—by testing them together in a single configuration. The tests in this section use 32 `gold8+F+C` clusters with a SRF 4 times larger than the 32-cluster test in Section 6.3. We call this configuration `gold32combo`; it is equivalent to `gold32+F+C+srf4x`.

The combination of more functional units, more clusters, and a larger SRF produces substantially larger gains than any of those additions alone. Figure 6.8 shows the gains when `gold32combo` is compared to the base `gold8` configuration, to `gold8+F+C`, and to `gold32`. The largest gain overall was for MARBLE, which was 5.8 times faster than `gold8`. MARBLE benefitted from a near-ideal speedup due to more clusters as well as more functional units and the decrease in short stream effects. ADVS-1 had the smallest increase (2.2 times); it had few short stream effects and benefitted little from additional

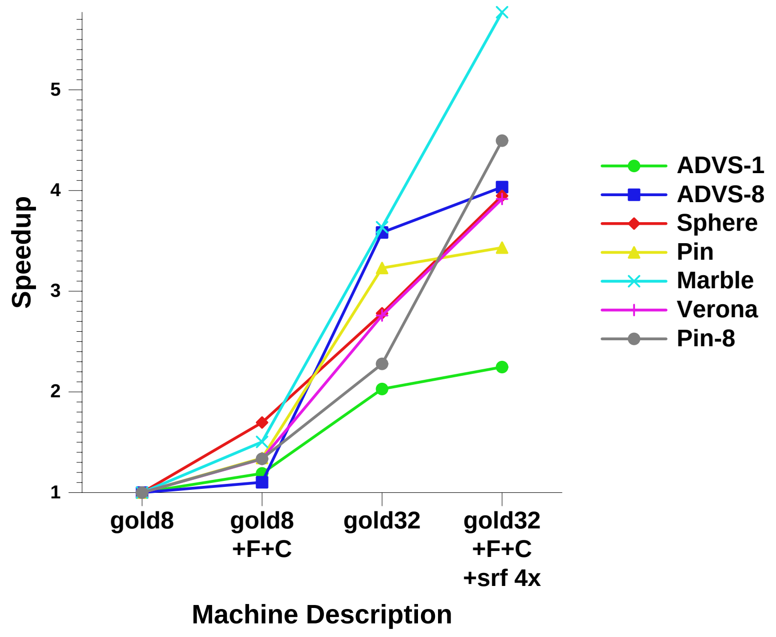


Figure 6.8: Performance for Scenes with Larger SRF and Different Cluster Counts and Configurations. Each single line indicates the overall performance speedup, normalized to `gold8` = 1.

functional units.

Figure 6.9 shows the kernel runtime for each of the 4 cluster configurations (`gold8`, `gold8+F+C`, `gold32`, and `gold32combo`). This comparison ignores occupancy effects and only considers the time spent running kernels. The best speedup was for PIN-8, whose kernels ran over 9 times faster with `gold32combo` than with `gold8`. The worst speedup was ADVS-1, whose kernels ran slightly more than 4 times faster. The speedups are roughly the sums of the speedups from the individual configurations (`gold8+F+C`, `gold8+srf4x`, and `gold32`), modulo occupancy effects, so we can conclude that the effects of adding ILP, DLP, and a larger SRF are orthogonal.

Like the results from additional clusters presented in Section 6.3, the additional hardware and greater demand on the memory system cause the memory demand to rise and the cluster occupancy to fall. The memory demand now approaches its theoretical limit in many of these scenes, causing them to become memory bound rather than computation bound. PIN-8, for instance, sustains 2.03 GB/s of memory bandwidth, whereas a memory

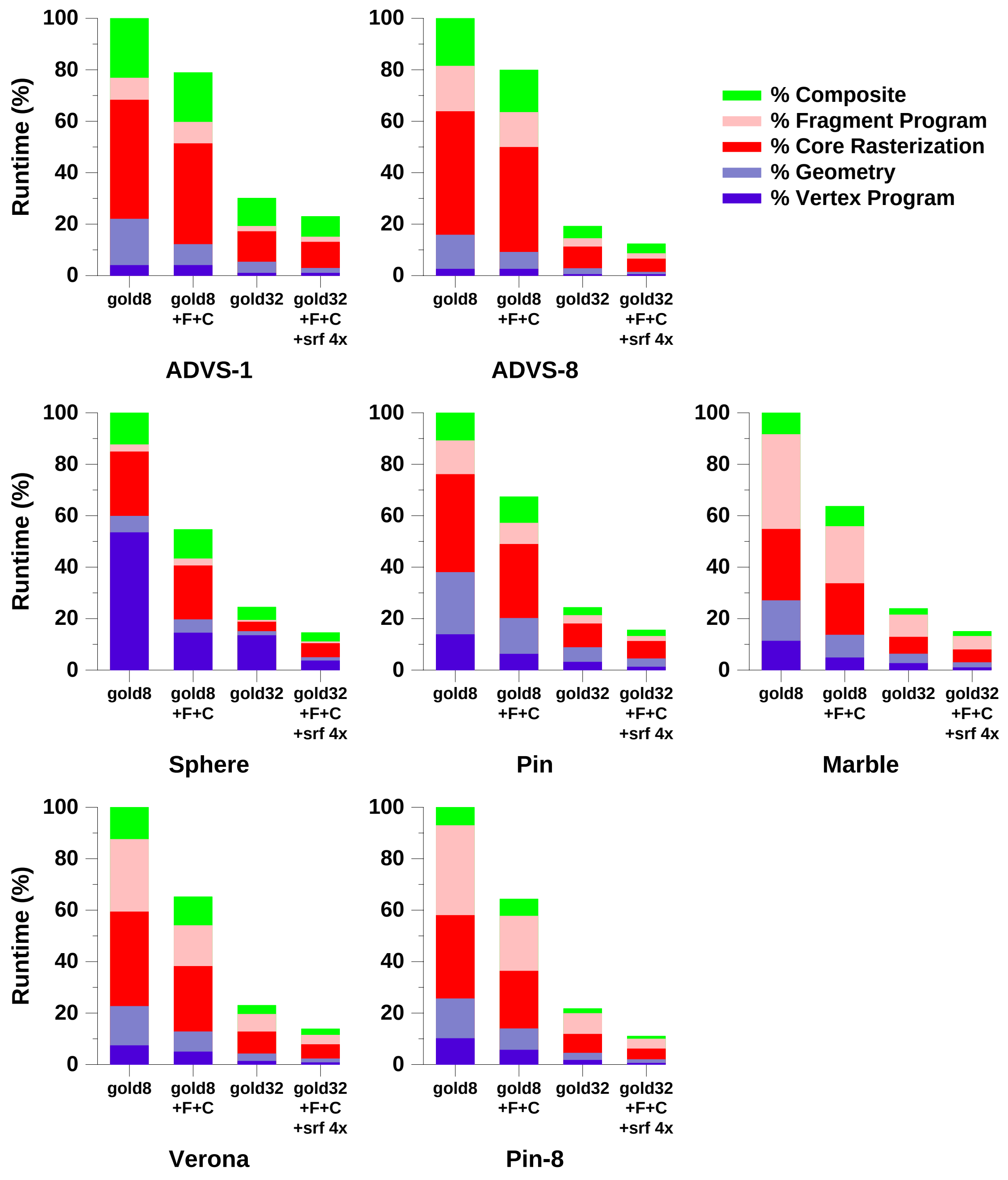


Figure 6.9: Stage Breakdown for Scenes with Larger SRF and Different Cluster Counts and Configurations. Total cycles are normalized to `gold8` = 100%. Cluster idle time is not considered in these graphs. The vertex program is treated separately from the rest of the geometry stages, and the fragment program is treated separately from the rest of the rasterization stages.

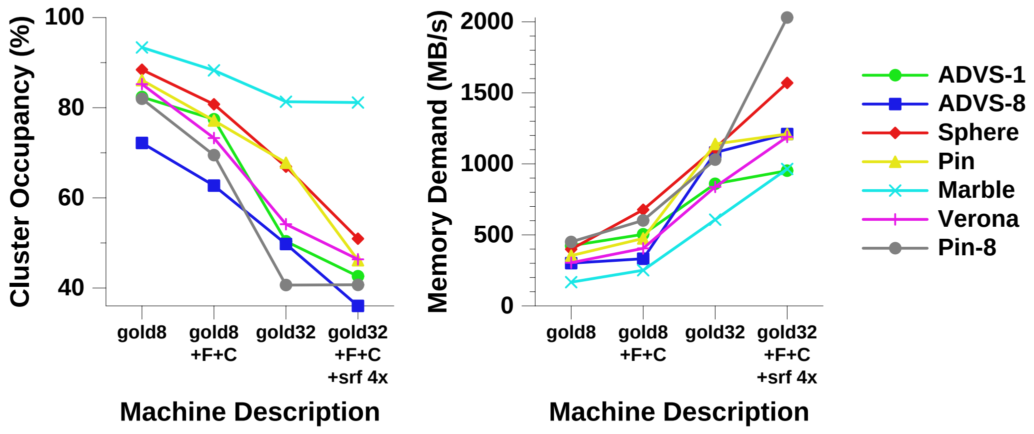


Figure 6.10: Cluster Occupancy and Delivered Memory Bandwidth for Scenes with Larger SRF and Different Cluster Counts and Configurations. Each single line indicates the relative performance for a single scene on each of 3 different cluster counts. The left graph indicates the percentage of time the clusters are busy for each scene and cluster configuration, and the right graph indicates the delivered memory system bandwidth for that scene and cluster configuration.

trace of the same address stream with no computation[3] sustains 2.49 GB/s.

6.5 Texture Cache

In the previous section we see that by adding additional computation hardware, we can convert our scenes from being limited by computation to being limited by memory. Memory traffic in graphics systems consists of three components: input traffic (vertex and mesh information), output traffic (fragment color and depth information), and texture. The average values across our scenes are 52% input, 20% output, and 27% texture. However, this texture statistic is slightly misleading as several of the scenes do not use any textures. Those that do have a higher proportion of texture traffic: 76% of PIN-8's traffic is texture, and 47% of ADVS-8's.

To reduce memory demand, we can make use of the locality of the memory traffic.

[3]PIN-8's memory trace for `gold32combo` differs from the `gold8` trace presented in Section 5.4; the streams are considerably longer in the 32-cluster version and result in a higher bandwidth. The gap between trace and achievable bandwidth for this scene is primarily attributable to a poor software pipeline that unnecessarily serializes computation and memory accesses.

It is possible (though not done in current graphics hardware) to cache input geometry information. And the depth buffer also can benefit from some caching, especially when a hierarchical depth buffer is used, as in some recent hardware systems. However, neither of these components have nearly as much locality as texture data, which caches well [Hakura and Gupta, 1997]. Most modern graphics hardware supports texture caches.

We add a cache to Imagine by intercepting memory requests for streams marked "cacheable" and looking them up in the cache. Our cache contains 16K words and is direct mapped with a cache line size of 2 words[4].

The hardware costs of such an addition are mostly the additional memory requirements. Imagine already has two 1 Mb data structures in the SRF and the microcode instruction store; a 16 KW cache would add another half million bits. Conceptually, the cache fits well between the memory system and the SRF, intercepting memory address requests provided by the address generators and satisfying them through the cache instead of through an external memory reference. Texture accesses are read-only, which greatly simplifies a cache design.

In the tests below, we use `gold32combo` as our base hardware configuration and PIN-8 as our comparison scene; this configuration and scene have the highest memory demand of all of our scenes and configurations. The bandwidth between the memory system and the SRF is multiplied by n (compared to Imagine's memory-to-SRF bandwidth). We run simulations with $n =$ 1, 4, and 32. Small values of n mostly affect the latency of cacheable memory requests; larger values begin to increase the delivered bandwidth as well.

Figure 6.11 shows the result of these two cache configurations. The hit rate in the cache for PIN-8 is 94.1%. The $n =$ 1, 4, and 32 caches produce performance gains of 6%, 28%, and 38% respectively; these gains are a direct result of an increase in cluster occupancy, from 40.7% in the base case to 43.5% for the $n =$ 1 cache, 52.4% for the $n =$ 4 cache, and 56.2% in the $n =$ 32 cache.

The stream-level software pipelining for this scene could be significantly improved,

[4]Modern texture caches have larger line sizes—16 words is typical—and block their textures. Adding a cache with a line size greater than Imagine's 2 words, however, would require a major redesign of the memory system, both in hardware and in simulation. Given the architecture of our simulator, the word size has an insignificant impact on performance anyway.

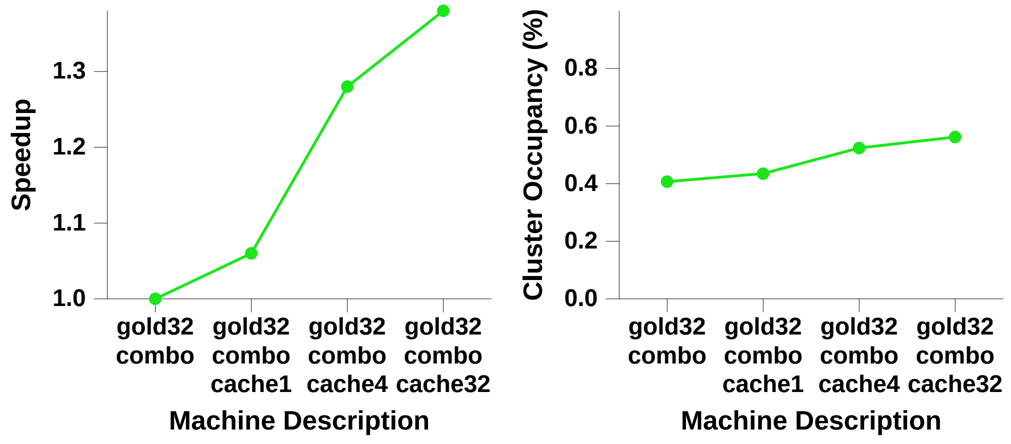


Figure 6.11: Texture Cache Performance and Occupancy.

which would increase cluster occupancy and hence performance for all cache configurations.

6.5.1 Hardware Interpolation

A second possibility for increasing texture performance, one that saves data movement and specializes computation, is a hardware texture interpolation unit located between the memory system and the SRF. Such a unit would intercept 8 mipmapped texture color samples returning from the memory system and, using weights calculated in the fragment program, blend them into a single color.

This unit would not affect memory bandwidth, but it would affect SRF bandwidth. Below is a table that describes the bandwidths necessary per (mipmapped) texture access in both configurations.

Task	`gold8`	`gold8+interp`
texture address, SRF $\longrightarrow$ memory	8 words/access	13 words/access
texture data, memory $\longrightarrow$ SRF	8 words/access	1 word/access
filtering, SRF $\longrightarrow$ clusters	13 words/access	—
filtering, clusters $\longrightarrow$ SRF	1 word/access	—

Thus, for each texture access, 16 words of SRF bandwidth are saved. SRF space requirements do not change[5].

Now, of all graphics applications, only some use mipmapped textures. Of our seven scenes, two use mipmapping (ADVS-8 and PIN-8). ADVS-8 spends 4.8% of its time filtering and PIN-8 spends 8.8% of its time filtering, time that would be eliminated with a hardware interpolation unit.

Overall, then, this unit would save 16 words per access in SRF bandwidth and a few percent in runtime for those applications that use mipmapped textures. Also, we would not have the capability of doing other filtering algorithms (such as anisotropic filtering) with this special-purpose hardware. This unit would likely not provide the performance gains necessary to justify its expense.

6.6 Hardware Rasterizer

The rasterization stage of the polygon rendering pipeline consumes the largest proportion of runtime in our implementation. Special-purpose rasterization hardware is at the core of today's commercial graphics processors, with the most recent chips advertising fill rates of over a billion pixels per second. So it is natural to consider merging the power of special-purpose rasterization hardware with our rendering pipeline in an effort to achieve higher performance.

Such a rasterization unit is easily integrated into a stream architecture such as Imagine. Imagine's SRF already has a large number of clients, including the clusters, network interface, and memory system. A special-purpose rasterizer could be added as another SRF client.

To evaluate this configuration, we simulate a modest hardware rasterizer as an addition to Imagine. The rasterizer is attached to the SRF through five streams: four input streams (triangle data on one stream, interpolant data for each vertex on the other three) and one output stream (which is used as an input to the fragment program). The rasterizer generates one output word every cycle, so the output fragment rate is dependent on the complexity of

[5] In our implementation, we write texels on top of their addresses, so no additional space is needed to store them.

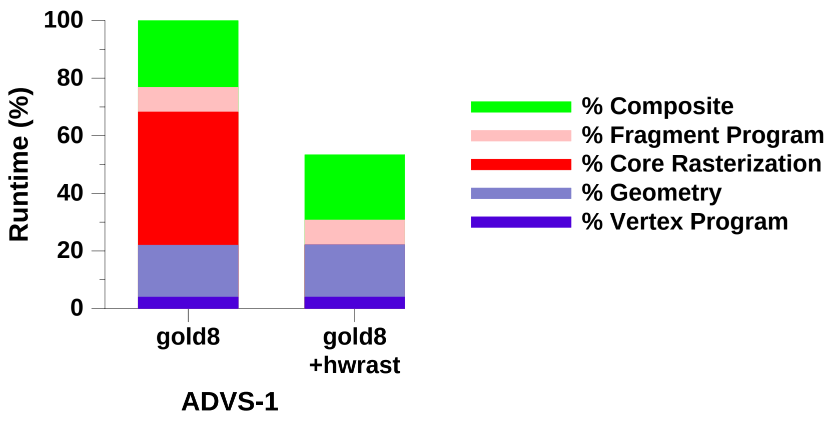


Figure 6.12: Stage Breakdown for ADVS-1 with Hardware Rasterization. Total cycles are normalized to `gold8` = 100%. Cluster idle time is not considered in these graphs. The vertex program is treated separately from the rest of the geometry stages, and the fragment program is treated separately from the rest of the rasterization stages.

the fragments. ADVS-1, for instance, has 6 words per fragment program input (framebuffer address, triangle ID, depth, and texture u/v/q). On this scene, then, this rasterizer could generate 83 million fragments per cycle (a small fraction of the rasterizer performance of modern special-purpose hardware). Our rasterizer also has a fixed latency of 100 cycles from receiving a triangle to outputting fragment data.

In software, the rasterization task is treated as a *hardware kernel*. The API treats a hardware kernel exactly like a programmable kernel in the clusters. When the rasterizer is invoked as a hardware kernel, the input streams are simply routed from the SRF into the hardware rasterizer, and the stream controller in Imagine properly manages the dependencies between streams.

In this chapter we have separated kernel runtime for each hardware configuration into several components: vertex program, fragment program, geometry, rasterization, and composition. One might imagine that adding a special-purpose rasterizer would simply remove the rasterization component from kernel runtime. And in fact, that is exactly what happens: Figure 6.12 shows the result for ADVS-1 with the standard `gold8` configuration and `gold8+hwrast`. The new configuration has a 89% kernel speedup over the base case.

Unfortunately, the overall performance increase is not as encouraging: the 32% application speedup is significantly less than the kernel speedup. The discrepancy is a consequence of poor software pipelining: the cluster occupancy drops from just over 80% to just over 50%. Normally Imagine must pipeline two units, the clusters and the memory system. This pipeline is done automatically by the stream scheduler and results in pipelines that are two stages deep. However, when hardware kernels are pipelined, they are grouped with the cluster kernels. As a result, in the generated schedule, while the hardware rasterizer is active, the clusters are often idle. The gains in overall performance are a consequence of the hardware rasterizer running faster than its cluster equivalent. However, a better profile would result in gains from allowing the clusters to concurrently work on another task as well.

Adding special-purpose hardware such as this rasterizer runs counter to the Imagine philosophy of programmability. Our hardware rasterizer would be of little use in running other media tasks or even other graphics pipelines such as Reyes or a raytracer. Specialization comes with a cost that is examined more closely in Section 7.1.

Chapter 7

Discussion

7.1 Time-Multiplexed vs. Task-Parallel Architectures

The machine organization of Imagine differs markedly from the organizations of today's commercial graphics processors. Imagine features a *time-multiplexed* organization, while contemporary graphics processors use a *task-parallel* organization. In this section we compare the two organizations.

7.1.1 Task-Parallel Architectures

As we have seen in Chapter 4, the graphics pipeline can be divided into many stages. Examples include the vertex program, primitive assembly, triangle setup, and triangle rasterization.

A task-parallel machine implements each of these stages with separate hardware. In a single-chip implementation, then, the processor die area is divided into several modules, each of which implements a separate stage in the pipeline. We can consider the overall task of rendering on a task-parallel machine as divided in *space*: the separate stages are run on different hardware modules.

Advantages and Disadvantages

In a deep, complex pipeline such as the graphics pipeline, this machine organization leads to high-performance implementations for several reasons.

First, each separate stage of hardware can run concurrently. This concurrency exploits the native data parallelism in the graphics pipeline, and with a pipeline many stages deep, a task-parallel architecture can be working on many primitives at the same time. (This advantage relies on the feed-forward structure of the graphics pipeline—if the pipeline allowed results in the latter half of the pipeline to influence those earlier in the pipeline, concurrent execution of multiple primitives might lead to difficulties in satisfying the ordering constraint.)

Second, each module can be specialized to its specific task. Because each module implements a single stage of the graphics pipeline, and because that stage often has fixed or limited functionality, the hardware implementing that stage can be optimized with special-purpose, custom hardware to perform that task both quickly and efficiently.

However, task-parallel organizations suffer from a significant disadvantage in load-balancing work between stages. Because the amount of work in each stage is not fixed, different scenes can cause different loads on the modules. One example is due to triangle size effects: scenes with large triangles will have a much different ratio between geometry work and rasterization work than a scene with small triangles.

In a task-parallel implementation, the balances between the modules must be fixed at the time the machine is designed. When a scene has properties that do not match these design decisions, hardware modules are idle. In our example, if our scene has triangles that are generally larger than the design point, the geometry hardware will be idle while the rasterization hardware is saturated doing rasterization work. In effect, on a given scene, a task-parallel architecture can only run as fast as the slowest module for the scene.

Another danger is overspecialization: not all features of the pipeline are used in any particular scene. Accelerating those features with special purpose hardware means hardware that sits idle for any scenes that do not use that feature. (For instance, hardware to perform fog calculations will not be used in any scenes that do not use fog.)

7.1.2 Time-Multiplexed Architectures

We have seen that in a task-parallel organization, the separate stages of the graphics pipeline are divided in space. A time-multiplexed architecture divides the pipeline not in space but in *time*.

Instead of dividing the stages onto different modules in hardware, a time-multiplexed architecture first evaluates the first stage using the resources of the entire machine, then the next stage, and so on. Thus, different stages are separated by time, not space.

The time-multiplexed organization relies on hardware that can execute any stage of the pipeline. To do so, this hardware must be quite flexible; it is difficult to imagine building a time-multiplexed machine without significant programmability.

This flexibility does not come without cost. Time-multiplexed implementations lose the advantages of hardware specialization featured in task-parallel implementations. While it would be possible to add specialized hardware to a time-multiplexed implementation for certain tasks in the pipeline, that hardware would be idle while the other tasks were running. Algorithms which could be accelerated using special purpose hardware in a task-parallel implementation must instead run on the less efficient, more general computation units of a time-multiplexed machine.

However, the time-multiplexed organization enjoys significant advantages. The first is load balancing. Time-multiplexed machines do not suffer from load imbalances between stages as task-parallel machines do because time-multiplexed machines can dynamically expand or contract the timeslice in which they run each module as needed. Stages that are particularly complex for a given scene simply take longer to run; stages that are simple complete more quickly. No matter what the balance is between stages, the computation hardware remains busy at all times.

The second advantage is centralization of resources. Consider a modern graphics processor that has programmable vertex and fragment processing units. Constructing a programmable module has fixed costs, such as a microcontroller or an instruction store, for example. A task-parallel implementation must pay this cost for each separate module, whereas a time-multiplexed implementation can amortize the fixed costs over all the hardware.

Sharing resources also can lead to better aggregate resource utilization. In our example, consider a task-parallel architecture that allocates space for 1000 instructions each for the vertex and fragment processing units. A centralized unit would have 2000 instructions, allowing scenes that require 1500 fragment instructions and 500 vertex instructions, for instance.

7.1.3 Continuum

The task-parallel and time-multiplexed organizations are not rigid; they may be bridged by hybrid architectures that use elements of each of them. Imagine, for instance, is not a strict time-multiplexed architecture because it separates the functions of computation and memory access with separate hardware. As a result, Imagine suffers from load imbalance between the two (expressed in our results as cluster and memory system occupancy).

An architecture such as Imagine could become more task-parallel by adding units that are specialized to specific functions in the graphics pipeline. Sections 6.5 and 6.6 describe potential Imagine extensions that would move Imagine away from a time-multiplexed organization.

Today's task-parallel graphics processors could become more time-multiplexed by sharing programmable units such as their fragment and vertex processing units. Potentially, different specialized hardware units could be generalized and combined, though this may not produce gains in efficiency.

7.1.4 Toward Time-Multiplexed Architectures

Today, the advantages of task-parallel architectures, particularly those associated with hardware specialization, lead to a performance advantage over time-multiplexed architectures.

Time-multiplexed architectures are intriguing candidates for future graphics processors, however, and will become increasingly compelling with the following trends:

Increased load imbalance If the load imbalances between stages continue to increase as pipeline complexity increases, time-multiplexed architectures are at an advantage.

Increased programmability The costs of adding separate programmable hardware to each stage in an implementation with many modules is significant. As the programmability of the pipeline increases, and in particular the number of stages that feature programmable elements, time-multiplexed architectures become more attractive.

More complex pipelines The concurrency of task-parallel implementations relies on the feed-forward nature of the pipeline. Feeding data back from a later stage of the pipeline to an earlier one in an arbitrary fashion will be a difficult task for two reasons. First, the connections between stages in a task-parallel machine are currently fixed and not fully connected. A much more flexible interconnection structure would be necessary to support arbitrary feedback paths. Second, because different batches of the input data are processed at the same time in different units, feeding back data from a later stage to a previous stage may violate ordering constraints. A time-multiplexed organization, which works on a single batch at a time and uses the SRF as a centralized communication mechanism between stages, suffers from neither of these difficulties.

Finally, in the most complex scenes rendered today (such as those in motion picture production), the large majority of work in the pipeline is programmable shading computation. In pipelines such as Reyes this programmability is implemented in a single stage. If a single stage (such as shading) were to become the dominant component in the rendering pipeline, the distinction between task-parallel and time-multiplexed organizations becomes less important. Implementers of a task-parallel organization would devote most of the chip area toward programmable shading because most of the work would be in programmable shading. At that point, most of the work is concentrated in one stage, so the implementations of the two approaches would be substantially similar in character.

Another trend that may eventually make the distinction meaningless is that off-chip bandwidth scales slower than on-chip capability. If graphics becomes completely bound by off-chip communication, the organization of the computation on the chip is unimportant; with either a time-multiplexed or a task-parallel organization, the performance is still limited by pin bandwidth instead of computation.

Today, however, the advantages of a time-multiplexed organization are augmented when the individual stages are programmable, as described in the next section.

7.2 Why Programmability?

Time-multiplexed implementations must by nature be flexible, as the same computation units are used for all stages in the pipeline. Imagine implements this flexibility with microcoded kernels that implement arbitrary user programs. What are the advantages of programmability, both on Imagine and in other programmable graphics processors?

7.2.1 Shading and Lighting

The first advantage to programmability is the ability to support flexible shading and lighting. With the RTSL, shaders and lights can be expressed in a high-level language and support complex effects that cannot be produced through the normal OpenGL pipeline.

RTSL supports a somewhat limited class of shading and lighting algorithms, however. In it, a single function is applied to every primitive with no large-scale data dependence and no control flow, and each input always produces one output. Such shaders map well to SIMD hardware.

A more complex shading languages such as RenderMan supports considerably more flexibility, and Imagine could easily implement more complex features than those supported by the RTSL.

For example, a natural extension to the current shader functionality would be the addition of data-dependent loops. One example of a shader for which this would be useful is noise/turbulence generation where the complexity was dependent on the rendered surface. Surfaces that required a lot of detail (such as those close to the camera) would use several octaves of noise, whereas surfaces that required less detail might only use a single octave. Imagine could process each element with a data-dependent number of noise loops to implement this effect.

Imagine does not easily permit arbitrary control flow, however. For example, subroutines would not easily map onto kernels because the Imagine implementation does not support them. Fortunately, the combination of inlined function calls, kernel composition, and conditional streams is flexible enough to allow the implementation of shaders with fairly complex control demands.

7.2.2 Alternate Algorithms

Programmability permits the use of multiple algorithms to implement stages in the rendering pipeline. For instance, Section 4.4 describes the implementation of two different rasterization algorithms. The important result from this implementation is not necessarily the superiority of one algorithm to the other but instead the realization that either or both could be used. Not only could the pipeline designer choose one or the other for any given application, but he could also use both in the same application and allow the renderer to dynamically and optimally determine which primitives should be associated with which rasterizers. Other potential algorithmic changes could be dicing in triangle setup to limit triangle size, more sophisticated culling algorithms, or the support of more varied input data formats such as higher-order surfaces.

Knowledge about the compile-time characteristics of the pipeline could allow pipeline optimization. For example, pipelines with disabled blending would benefit from texturing after depth comparison, which would save texture bandwidth. Pipelines with high depth complexity and sophisticated shading might benefit from an early depth kill, which could potentially save shading work.

7.2.3 Hybrid Rendering

A programmable pipeline can feature elements from entirely different pipelines or combine with them to produce hybrid renderers. Besides the Reyes pipeline described in Chapter 8, obvious candidates include a raytracing pipeline and an image-based rendering pipeline. Such pipelines could even include non-graphics elements such as signal processing kernels.

7.3 Lessons for Hardware Designers

7.3.1 Imagine Architecture

Cluster Architecture

The mix of functional units chosen for the clusters has been a good match for a wide variety of media applications, including graphics.

Most rendering tasks are either floating-point or integer with little overlap (the only exceptions would be some of the rasterization code, and perhaps the mipmap calculations); this result suggests that having functional units which can evaluate both floating-point and integer computation is a good idea.

The distributed register files have been successful for a wide variety of media applications and continue to scale with more complex clusters (Section 6.1). Such a cluster architecture should be well-suited for programmable shading modules in current commercial graphics hardware as well.

Cluster register file allocation is a significant problem, and in retrospect, the register files are not large enough for some of our more complex kernels. Informally, cluster complexity is measured by the number of operations in the main loop, and kernels usually will register allocate with main loops of under 600 operations, will rarely register allocate if the main loop has more than 1000 operations, and may or may not register allocate in between that range. These figures could probably improve with better kernel scheduling algorithms, as the current ones do not take into account register pressure. Scheduling kernels on clusters with shared interconnect and distributed register files while considering register pressure is still an open problem.

SRF Architecture

Kernels in our rendering pipeline have a significant amount of computation for each element, so the bandwidth from SRF to clusters was not a limiting factor for performance. Conditional stream bandwidth, however, did limit performance in many of the kernels (as described in Section 5.3). Future SRF designs may consider embedding some conditional stream functionality in the SRF to improve its bandwidth.

One mistake in our design was in not making the SRF's block size (32 words) the same as the cluster count (8 words). As a result, we could only address on block boundaries and could not specify streams at arbitrary starting locations in the SRF. This caused difficulties in implementing the `sort` kernel in particular.

Another design decision was to make all streams multiples of the number of clusters. Thus all streams had to be "flushed" with bogus data to fill out the streams to the correct lengths. This did not cause a significant degradation in performance, but it was tedious to program. Ideally, supporting arbitrary stream lengths would be much easier for the programmer.

More flexible streams in the SRF would probably aid the stream scheduler's allocation of the working set of streams within the SRF. For example, not requiring that the entirety of a stream be allocated in a contiguous portion of the SRF memory (instead allowing multiple blocks to be chained together) might make better use of the space in the SRF. Another interesting feature of a future SRF would be the ability to address both backwards and forwards. This functionality would be useful when a single stream is divided into two streams (for example, the hash kernel divides a single fragment stream into conflicting and unique streams). Then the output streams could be allocated in a block the same size as the index stream, with one output stream starting from the beginning and the other from the end, filling toward each other.

Further SRF functionality is also necessary to support more flexible overflow modes, such as partial double buffering (described in Section 4.6).

Another interesting direction would be the support of "virtual streams" in the stream register file, addressable by kernels. Imagine kernels can address a combination of 8 input or output streams, but some potential kernel algorithms could address many more. Two algorithms that were discussed were implementing Warnock's Algorithm [Warnock, 1969] and implementing a radix sort.

7.3.2 Software Issues

Language Decisions

The decision to use a higher-level language (KernelC) to specify kernels was an excellent one. Of course specifying programs in a higher-level language is more convenient than writing assembly.

But more importantly, the kernel scheduler performs a vital function. Our clusters contain several functional units connected through a switch. We have demonstrated that our media applications in general have made use of this large number of functional units. Scheduling the functional units with their variable latencies, the register files that feed them, and the switch that connects them is quite a difficult task, perhaps one that is impossible to do efficiently by hand. However, the kernel scheduler efficiently handles this complex hardware and produces schedules that are typically close to the theoretical (critical-path-bound) limit.

Thus, scheduling to complex hardware with a high-level language and automated tools is not only possible but also desirable. In addition, writing in the higher-level language allows both portable code and code that migrates easily to new cluster configurations (such as those in Section 6.1).

One promising direction of research is the unification of the kernel and stream levels of code. StreaMIT [Gordon et al., 2002] is one effort toward this goal, and the results of Stanford's Brook programming language for streams are also encouraging.

Software Pipelining

Software pipelining (SWP) is used at both the kernel and stream levels of programming. It has proven to be a valuable technique at both levels.

SWP at the kernel level is supported by hardware to squash inappropriate writes in inactive pipeline stages. For most kernels, particularly those with larger critical-path-to-computation ratios, SWP delivers significant performance gains, often from 50 to 100 percent. On the minus side, it contributes to the prime/drain cost of short streams, and it increases register pressure, but overall, it is well worth the additional hardware cost. As a bonus, SWP for kernels is provided transparently to the user. Future hardware may be able to eliminate the cost of priming software-pipelined conditional streams in the prologue as well, which will reduce their associated short-stream effects.

At the stream level, SWP helps to keep the critical units busy. For Imagine, on the scenes we studied, the critical unit is the clusters, and SWP keeps their occupancy high by overlapping two batches and ensuring memory accesses in one batch are covered by cluster computation in the other.

Producing SWP at the stream level has no hardware support and the software support takes considerable effort from the user; the software pipeliner could be improved as well.

Chapter 8

Reyes on a Stream Architecture

As graphics hardware continues to make remarkable gains in performance, it will render increasingly complex scenes. The efficiency and flexibility of pipelines such as OpenGL are taxed under the current trends of decreasing triangle size and the demand for complex, programmable shaders. This naturally leads to the question of how alternative pipelines lend themselves to high-performance implementations. At one extreme of the performance-realism spectrum is the Reyes rendering pipeline [Cook et al., 1987]. Reyes was designed at Lucasfilm to render extremely complex scenes with total emphasis on the photorealistic, high-fidelity imagery targeted by today's real-time graphics hardware.

In this chapter, we examine the issues associated in implementing the Reyes rendering pipeline with the goal of real-time frame rates. We also identify several key characteristics of a pipeline that contribute to an efficient implementation: arithmetic intensity, data reference locality, predictable memory access patterns, and instruction- and data-level parallelism.

The Reyes pipeline, as well as other graphics pipelines, already contains abundant parallelism as well as high arithmetic intensity (the number of operations per fragment). In addition, the uniform size of the rasterization primitives, called *micropolygons*, produces a predictable number of fragments, which simplifies memory allocation and streamlines the rasterization and fragment-processing steps. We detail our rasterization algorithm, which takes advantage of these uniform primitives.

Another key aspect of the Reyes pipeline is support of high-level primitives, such as subdivision surfaces. While subdivision surfaces significantly reduce the amount of memory bandwidth consumed by loading models, they also introduce additional control complexity. This is especially true for adaptive subdivision schemes because crack prevention traditionally requires elaborate stitching schemes and non-local knowledge of the surface. We address these problems by introducing a novel crack prevention algorithm that stores edge equations of the micropolygons instead of their vertices.

Finally, we compare our implementation of the Reyes pipeline to an OpenGL pipeline on Imagine. Using Imagine as a common substrate for comparison on several scenes, we identify the relative strengths and weakness of both approaches. We show that on several scenes with complex shading, OpenGL delivers superior performance to Reyes. Specifically, the high computational cost of the subdivision and the large number of micropolygons produced that did not contribute to the final image are the primary impediments to making a Reyes implementation competitive with OpenGL.

8.1 Comparing the OpenGL and Reyes Rendering Pipelines

8.1.1 OpenGL

OpenGL is the basis of the pipeline described in Chapter 4. It consists of the following stages:

Transformations and vertex operations Objects are specified in object space and are transformed to eye space, where per-vertex operations, such as lighting and other shading operations, are performed. Recent hardware has added user programmability to this stage [Lindholm et al., 2001].

Assemble/Clip/Project Triangles are assembled from vertices, transformed to clip space, clipped against the view frustum, and projected to the screen.

Rasterize Screen-space triangles are rasterized to fragments, all in screen space. Per-vertex parameters are interpolated across the triangle.

Fragment operations Per-fragment operations, such as texturing and blending, are applied to each fragment. Like the vertex operations stage, this stage supports increasing user programmability.

Visibility/Filter Visibility is resolved in this stage, usually through a depth buffer, and fragments are filtered and composited into a final image.

8.1.2 Reyes

The Reyes image rendering system [Cook et al., 1987] was developed at Lucasfilm and Pixar for high-quality rendering of complex scenes. The system, developed in the mid-1980's, was not designed at the time for real-time rendering but instead for rendering more complex scenes with higher image quality and rendering times from minutes to hours. Reyes is the basis for Pixar's implementation of the RenderMan rendering interface [Upstill, 1990].

The Reyes pipeline has four main stages, described below. The primary rendering primitive used by Reyes is the *micropolygon* or *quad*, a flat-shaded quadrilateral. In the original Reyes implementation, quads were no larger than 1/2 pixel on a side[1], but typical quads in modern Reyes-like implementations are on the order of 1 pixel in area because modern shaders are self-antialiasing.

Dice/Split Inputs to the Reyes pipeline are typically higher-order surfaces such as bicubic patches, but Reyes supports a large number of input primitives. Primitives can split themselves into other primitives, but each primitive must ultimately dice itself into micropolygons. Dicing is performed in eye space[1].

Shade Shading is also done in eye space by procedurally specified shaders. Because micropolygons are small, each micropolygon is flat-shaded.

Sample Micropolygons are projected to screen space, sampled, and clipped to the visible view area. Reyes uses a stochastic sampling method with a variable number of

[1]Our implementation differs from the traditional Reyes approach in quad size, subsampling, and dice space, as described in Section 8.2.

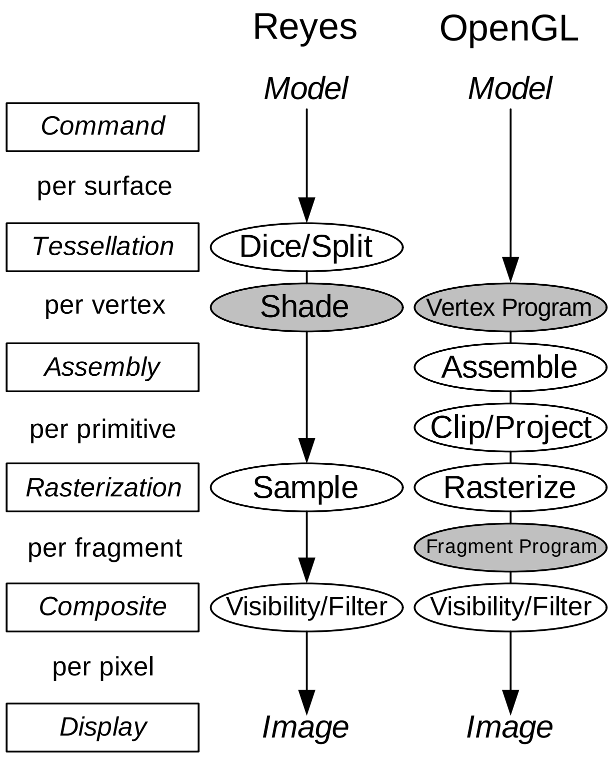


Figure 8.1: Reyes vs. OpenGL. The left column shows stages in a generic pipeline; the middle and right columns show the specific stages in Reyes and OpenGL. Shaded stages are programmable.

subpixels per pixel (16 in the original Reyes description[2]).

Visibility/Filter Visibility is resolved using a depth buffer with one sample per subpixel, then the visible surface subpixel values are filtered to form the final image.

8.1.3 Differences between OpenGL and Reyes

Both pipelines are similar in function, as shown in Figure 8.1. In both, the application produces geometry in object coordinates, which are then shaded, projected into screen space, rasterized into fragments, and composited using a depth buffer. However, the pipelines differ in three important ways: shading space, texture access characteristics, and the method

[2]Our implementation differs from the traditional Reyes approach in quad size, subsampling, and dice space, as described in Section 8.2.

of rasterization.

- The OpenGL pipeline shades at two stages: on vertices in eye coordinates (typically lighting calculations, and more recently programmable vertex shaders), and on fragments in screen coordinates (typically textures and blends, and more recently programmable fragment shaders). The Reyes pipeline supports a single shading stage on micropolygon vertices in eye coordinates.

- In OpenGL, texturing is a per-fragment screen-space operation; to avoid visual artifacts, textures are filtered using mipmapping [Williams, 1983]. Texture accesses are incoherent from texel to texel. In Reyes, texturing is a per-vertex eye-space operation. Reyes supports “coherent access textures” (CATs) for many classes of textures including standard projective textures. CATs require surface dicing at power-of-two resolutions but cause texels in the texture pyramid to align exactly with the vertices of the quads. Therefore, when accessing CATs, texture filtering is unnecessary, and texels in adjacent micropolygons can be accessed sequentially with significant savings in computation and memory bandwidth. Not all textures can be accessed coherently—non-CATs include environment maps, bump maps, and decals.

- OpenGL’s primitive is a triangle. Objects are specified in triangles, and the rasterization stage of OpenGL must be able to rasterize triangles of arbitrary size. The primitive for the Reyes pipeline is the micropolygon, whose size is bounded in screen space to a half-pixel on a side. Micropolygons are created in eye space during the dice stage of the pipeline, so the dicer must make estimates of the screen coverage of the generated micropolygons.

The fundamental differences in Reyes—single shader, coherent texture accesses, and bounded primitives—are all desirable properties for hardware implementation. Supporting only one programmable unit rather than two is a simpler task for hardware designers. Coherent accessed textures both reduce texture demand from the memory system and increase achievable memory bandwidth.

Bounded-size primitives (particularly small ones such as those in Reyes) have several advantages. The rasterizer does not have to handle the complexity of arbitrary sized primitives, so bounded-sized primitives can be rasterized with simpler algorithms and hardware than unbounded ones. Moreover, every sample within a bounded primitive can be computed in parallel because the total number of possible samples is small and bounded. Evaluating several bounded-size primitives in parallel load-balances better than unbounded primitives. And storage requirements for generated samples are much more easily determined with bounded primitives than with unbounded ones.

8.1.4 Graphics Trends and Issues

How do these pipelines cope with the issues facing graphics hardware designers of today?

Decreasing Triangle Size

As graphics hardware has increased in capability, models have become more detailed, and triangle size has decreased. The efficiency of factoring work between the vertex and fragment levels in OpenGL is one of the primary reasons that it is the dominant hardware organization today.

The shading work in OpenGL pipelines is divided between vertices and fragments. Vertex-level shading calculations are performed on each vertex. These results are interpolated during rasterization and then used as inputs to the fragment shader, which evaluates a shading function on each fragment.

Because interpolation is cheaper than evaluating the entire shading function at each fragment, and because the number of fragments in a scene is typically many times the number of triangles, this factorization of shading work into the vertex and fragment levels reduces the overall amount of work for the scene.

However, as triangle size continues to decrease, the benefits of this factorization become less significant. For scenes in which the average triangle size is 1 (the numbers of triangles and fragments are equal), there is no benefit.

Host and Memory Bandwidth

Host and memory bandwidth are both precious in modern graphics processors. Reducing the necessary memory bandwidth is achieved by a variety of techniques as texture caches [Hakura and Gupta, 1997], prefetching [Igehy et al., 1998a], and memory reference reordering [Rixner et al., 2000a]. Parallel interfaces are among the methods used to reduce host bandwidth [Igehy et al., 1998b].

The Reyes pipeline, by supporting higher-order datatypes, can reduce host bandwidth over sending a stream of triangles. And Reyes' coherent access textures can also help reduce the necessary bandwidth to texture memory.

8.2 Imagine Reyes Implementation

Our Reyes implementation follows the description in Section 8.1.2. We begin by projecting the control points of the input B-spline to screen space. We then subdivide in screen space, ensuring that no quad is larger than a fixed size. Because in this implementation we do not support supersampling, we do not subdivide all the way to Reyes' traditional 0.5 pixel area limit. The effects of different subdivision limits are discussed in Section 8.4.2.

After subdivision, we transform the resulting quad positions and normals back into eye space for shading. This differs from the traditional Reyes implementation, which subdivides in eye space with knowledge of screen space. Quads are then shaded in eye space using the RTSL-generated shader. Next, the sampling kernel inputs quads and outputs fragments, which are composited to make the final image.

8.2.1 Subdivision

The subdivision step of the Reyes pipeline is responsible for dividing the high-level primitives into micropolygons. We chose to implement the Catmull-Clark [Catmull and Clark, 1978] subdivision rules to refine these high-level surfaces[3], allowing native support of subdivision, B-Spline, and Bezier surfaces. The subdivision boundary rules are also included, so effects such as sharp creases in the subdivision surfaces are possible.

Subdivision begins with a collection of control points for the surface. In general, these points can be arranged in an arbitrary topology, but for our implementation we assume a quadrilateral mesh. The subdivision kernel begins by selecting an initial quad from the

[3]Other subdivision schemes present similar design issues.

control mesh and computing the side lengths of that quad. If all the side lengths fall below the stopping threshold[4], the quad is considered complete and is output by the kernel. Otherwise, the quad must be subdivided further, producing four, smaller quads. Now, one of these quads is selected and tested for completion. This process continues as a depth-first traversal of the quad-tree associated with the original control mesh with the leaf depth determined by the screen size of the associated quad. Since each stage of the subdivision requires testing the side lengths in pixels of the quads, this step is naturally performed in screen space.

The depth-first traversal has a key storage advantage—the number of live data values needed at any one time to produce a final set of N fragments is $O(\log N)$. This is in contrast to a triangle-based approach, where each rasterization primitive is part of the input set and therefore $O(N)$ live values are needed to produce the N fragments of a complex surface. While the storage efficiency is useful in any Reyes implementation, it is especially important in an efficient hardware implementation: the ability to produce a large number of fragments from a small amount of control information saves critical memory bandwidth.

However, realizing this savings in memory bandwidth presents several implementation challenges. A naive adaptive subdivision algorithm could use a completely local stopping criterion. However, if neighboring quads are subdivided at different levels, cracks in the final surface can appear. Typical algorithms for eliminating cracks involve a stitching pattern to reconnect the surface [Müller and Havemann, 2000], which can be used in concert with a global rule such as limiting the difference in subdivision levels for neighboring quads [Zorin and Schröder, 2000]. However, these approaches are not attractive in stream processors for several reasons. First, the control decisions and number of possible stitching patterns make an efficient data-parallel implementation difficult. Secondly, using global information to determine the stitching pattern can defeat the $O(\log N)$ storage requirement and also increase the complexity of memory access patterns, while also decreasing the efficiency of the stream implementation.

We tackle the problem of surface cracks by implementing a novel and completely localized solution: instead of describing the final micropolygons using their corner vertices,

[4]The stopping threshold in our implementation is 1.5 pixels; the effects of different thresholds are discussed in Section 8.4.2.

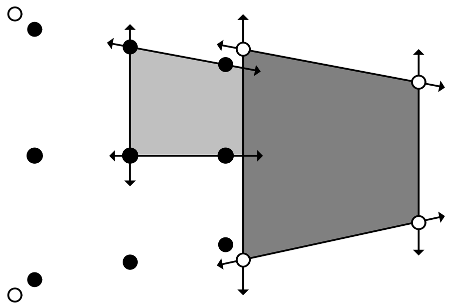


Figure 8.2: An Example of Crack Prevention Using Edge Equations. The dark gray quad is completed at the i^{th} subdivision level, while the light gray quad is completed at the $i+1^{th}$ level. By storing the shared edge as soon as it meets the subdivision criterion, no cracks are created.

they are represented using four edge equations. During subdivision, edge lengths are continually tested to determine if a quadrilateral requires further refinement. Instead of waiting until all four edges meet the length threshold, our approach freezes the final edge equations of a quad *immediately* after they fall below the threshold. Once all four edges have been stored, the final quad is output. This implies that the four edge equations may come from different levels of refinement. However, the edges of a quad are always consistent with its neighbors because the length criterion used on an edge shared between two quads is consistent. This consistency between shared edges is sufficient to prevent cracks in the final surface.

An example of this algorithm in shown in Figure 8.2. First, the control points of the mesh at level i are shown as unfilled circles. At level i of the refinement, all the edges of the right quad have fallen below the stopping threshold. The right quad, whose interior is shown in dark gray, is then complete and output at the i^{th} level of refinement. The right edge of the left quad has fallen below the threshold, so it is stored. However, the other edges of the left quad require further subdivision, so refinement continues. The vertices produced at the $i+1^{th}$ level of refinement are shown as filled circles. The subdivision algorithm perturbs the four vertices of the previous quad and also introduces five new vertices. At this point, the edge lengths of the upper-right sub-quad are tested and found to be below the threshold. This quad, whose interior is shown in light gray, is then output, using three edge equations from the $i + 1^{th}$ subdivision level and one from the i^{th}. Now the importance of storing edge equations becomes clear. The four vertices of the second quad do not necessarily abut the first quad because the refinement can perturb them. However, by using the edge equation from the previous subdivision level to define the right side of the second quad, a crack between the two quads is avoided.

8.2.2 Shading

Shaders are generated from the same RTSL description as in the OpenGL implementation. However, because Reyes has only one stage in the pipeline for shading, RTSL's vertex and fragment frequencies are collapsed into a single frequency. The generated shader kernel projects the screen-space vertices and normals back into eye space, computes the shading function (using coherent access textures if necessary), and outputs a single color per vertex.

8.2.3 Sampling

The sampling stage was implemented as a simple bounding-box rasterizer. Since the subdivider described above guarantees that the micropolygon to be drawn is under a certain size, only a small number of pixel locations need to be tested against the four line equations for the micropolygon. The bounded size of the micropolygons leads to two performance improvements over a rasterizer found in the OpenGL pipeline: good load balancing when run under SIMD control and flat shading within the micropolygon with no necessary interpolation. The original Reyes pipeline used stochastic sampling [Cook, 1986], providing many subsamples per pixel with slightly non-regular sample locations. This scheme was not used for our implementation in order to provide a fair comparison with the OpenGL pipeline. However, it would be straightforward to extend the current sampler to support stochastic sampling.

Reyes is particularly well suited to more complex sampling effects such as depth of field and motion blur. The cost of implementing these effects is merely reprojecting and resampling with no reshading, a much smaller cost than accomplishing the same effects in OpenGL using the accumulation buffer. Our implementation could easily be extended to support these effects.

8.2.4 Composite and Filter

This stage is identical to our OpenGL implementation. Filtering is not currently implemented but could be added in one of two ways. First, subpixels could be composited separately (effectively, a framebuffer with higher resolution) without maintaining a concurrent image at final resolution, and at the end of a frame, subpixels would be filtered as a postpass to create the final image. Second, subpixels are still composited separately but a image at final resolution is concurrently maintained. The second method requires an extra color read and write (for maintaining the image at final resolution) for each sample so potentially uses more memory bandwidth for high-depth complexity scenes. However, it alleviates the massive burst of memory bandwidth necessary if the final image is generated at the end of the scene.

8.3 Experimental Setup

For the results in this paper we use the Imagine cycle-accurate simulator `isim` and functional simulator `idebug`, described in Section 5.1. The OpenGL scenes are measured with `isim`, the Reyes (because of tool issues) with `idebug`. Because of the differences between the two (`idebug` does not model kernel stalls or cluster occupancy effects), `isim` results are on average 20% slower than `idebug`, so all `idebug` results are scaled by 20% to match the more accurate simulator.

Our Reyes implementation also made slight changes to the simulated Imagine hardware. The most significant was increasing the size of the dynamically addressable scratchpad memories in each cluster from 256 to 512 words. These scratchpads are used to implement the working set of quads in the depth-first traversal during adaptive subdivision, and having a larger scratchpad was vital for kernel efficiency. Second, the size of the microcode store and the local cluster register files were increased. We expect future improvements to the Reyes implementation and the software tools will allow us to return the microcode store and cluster register file sizes to the same size as the Imagine hardware.

Scene	Visible Frags	Tris	Avg. Tri Size	Patches	Quads
TEAPOT-20	137k	23.5k	11.6	28	574k
TEAPOT-30	137k	52.1k	5.26		
TEAPOT-40	137k	91.8k	2.99		
TEAPOT-64	137k	233k	1.18		
PIN	80.0k	91.6k	2.29	12	486k
ARMADILLO	48.9k	93.7k	3.83	1632	328k

Table 8.1: Statistics for OpenGL and Reyes Scenes.

8.3.1 Scenes

All scenes are rendered into a 720×720 window. Datasets, textures, cleared depth and color buffers, and kernels are located in Imagine memory at the beginning of the scene. For OpenGL scenes, the input dataset is expressed as subdivided triangle meshes; for Reyes scenes, the input dataset consists of B-spline control points.

We compare three scenes:

- TEAPOT renders the Utah teapot lit by three positional lights with diffuse and specular lighting components. The OpenGL version of the teapot is drawn at several subdivision levels (indicated as TEAPOT-N, where each patch is diced into $2N^2$ triangles) to show performance as a function of triangle size. References to TEAPOT in the context of OpenGL are to TEAPOT-20.

- PIN draws the bowling pin from the UNC Textbook Strike dataset (described in Section 5.1).In OpenGL, we render this scene as PIN-1 and PIN-8, which use point sampled and mipmapped textures, respectively. The Reyes version uses a single coherent and properly filtered access per texture per fragment.

- ARMADILLO renders the Stanford armadillo with a single light and a complex marble procedural shader. The shader calculates a turbulence function identical to the MARBLE scene described in Section 5.1.

Details for the scenes are summarized in Table 8.1.

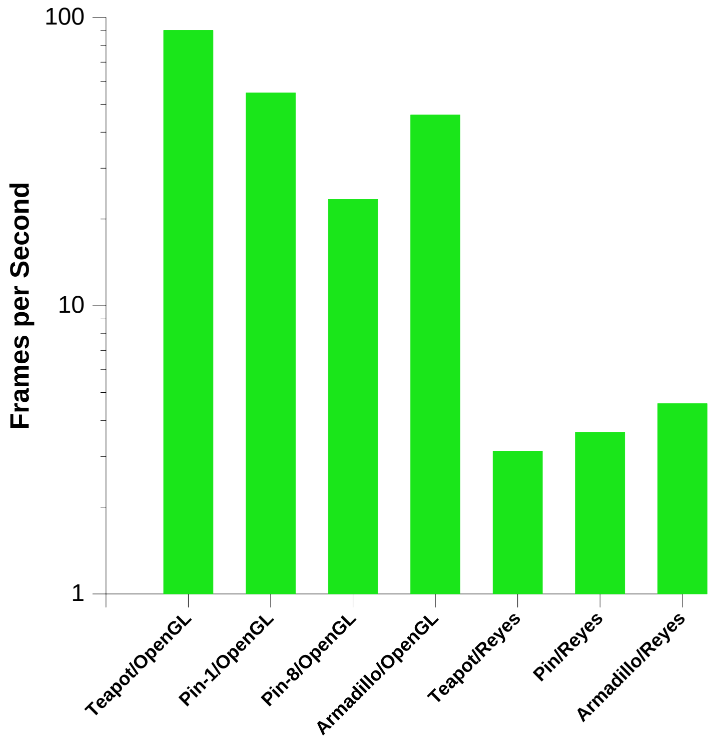


Figure 8.3: Simulated runtime for our scenes. OpenGL scenes run an order of magnitude faster than Reyes scenes.

8.4 Results and Discussion

Figure 8.3 shows our simulated performance for OpenGL and Reyes scenes. We see that OpenGL scenes enjoy a significant performance advantage over their Reyes counterparts. There are two reasons for this. First, in Reyes, scenes spend the majority of their time in subdivision, a stage not present in the OpenGL pipeline. Second, our Reyes implementation produces many quads that cover no pixels and do not contribute to the final image. We discuss these points in more detail below in Sections 8.4.1 and 8.4.2.

8.4.1 Kernel Breakdown

Figure 8.4 classifies the time spent computing each scene into five categories: geometry, rasterization, composition, and the programmable vertex and fragment programs. Subdivision is considered part of the geometry stage, and the vertex program in Reyes encompasses

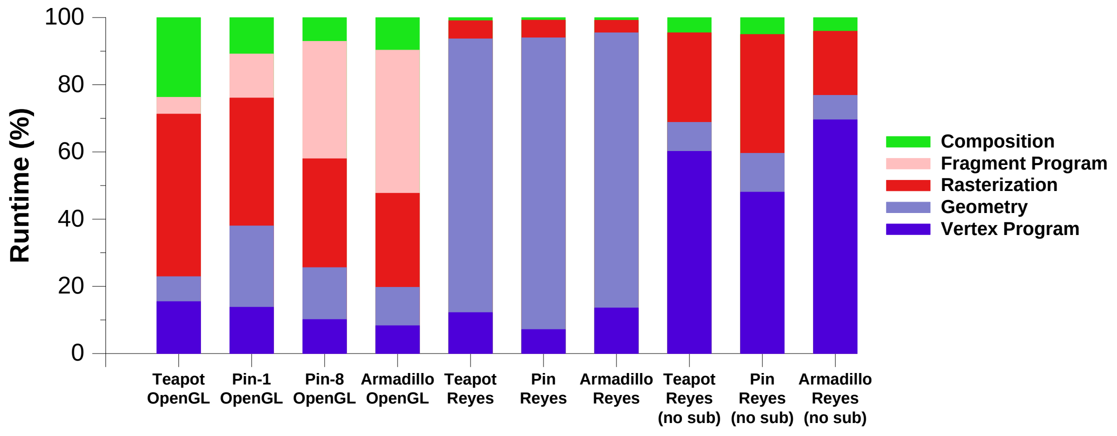


Figure 8.4: Stage breakdown of work in each scene. All scene runtimes are normalized to 100%. The first 4 scenes are OpenGL scenes; the next 3 are Reyes scenes; and the final 3 are Reyes scenes with the subdivision runtime (normally part of the Geometry stage) removed.

all the shading work for the scene because Reyes does not have a fragment program.

We see that Reyes runtime is dominated by geometry processing, in particular the subdivision kernel. On average, this kernel takes 82% of the runtime.

OpenGL does not have a subdivision stage because its primitives are subdivided either at compile time or by the host. When subdivision is removed from the Reyes accounting, the two pipelines have more similar stage breakdowns. The Reyes pipelines spend comparatively more time in the shading stages than do the OpenGL pipelines, with the exception of the OpenGL PIN-8 scene. Mipmapping is an expensive operation computationally (simply adding mipmapping to PIN-1 cut the resulting OpenGL performance in half), so the amount of time spent in shading in this scene was greater than its Reyes counterpart, which did not require texture filtering.

Rasterization is considerably simpler in the Reyes scenes for two reasons. First, determining pixel coverage of bounded primitives is computationally easier and more parallelizable than unbounded primitives. Second, the primitives in Reyes are smaller and have less computation. OpenGL implementations must carry all their interpolants through their rasterizers, while a Reyes sampler must only carry a single color.

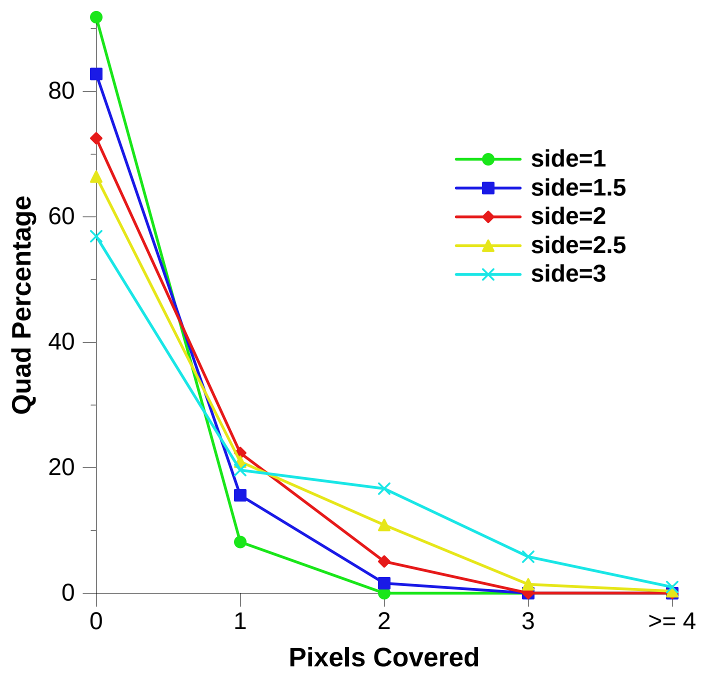


Figure 8.5: Quad Size for PIN. Other scenes have similar characteristics. Each data point represents the percentage of quads in PIN that cover a certain number of pixels given a certain stop length. Lines indicate points associated with the same subdivision stop length. Our implementation has a stop length of 1.5 pixels.

8.4.2 Reyes: Subdivision Effects

Even with a zero-cost subdivision, the Reyes scenes are still about half the performance of their OpenGL equivalents. This cost is largely due to shading and rasterization work performed on quads that cover no pixels. Ideally, each quad would cover a single pixel. Quads that cover more than one pixel introduce artifacts, while quads that cover zero pixels do not contribute to the final image.

Figure 8.5 shows the distribution of pixels covered by quads for PIN for several different subdivision stopping criteria (no quad side greater than a certain length). Our implementation stops subdividing when all quad sides are less than 1.5 pixels in length.

In practice, the majority of quads cover no pixels at all, even for larger stop lengths. On our three scenes, with a stop length of 1.5 pixels, zero-pixel quads comprise 73–83% of all generated quads. Even when we double the stop length to 3.0 pixels, we still find that over half of all quads generated do not cover any pixels. As a result, our implementation spends

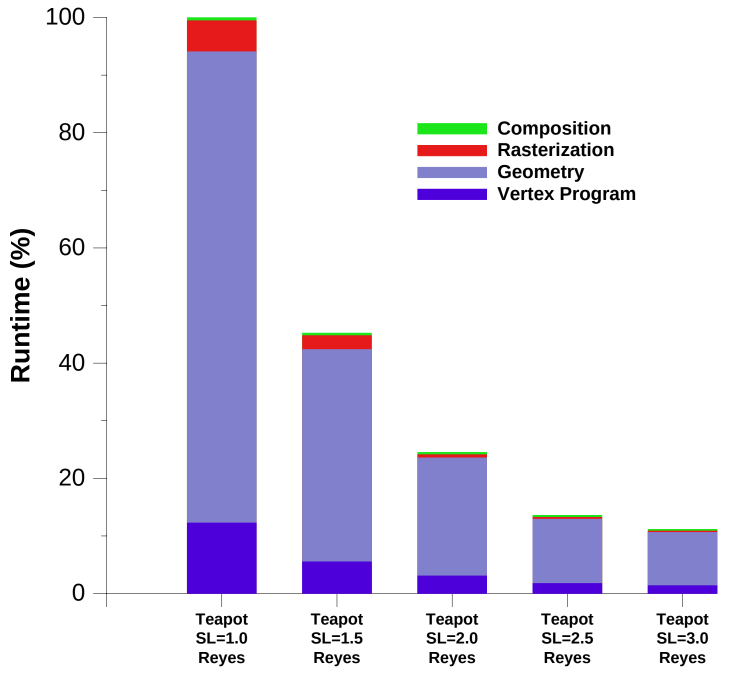


Figure 8.6: TEAPOT performance at several stop lengths, normalized to a stop length of 1.0 pixels. Our implementation has a stop length of 1.5 pixels.

a significant amount of time shading and sampling quads that produce no fragments.

To improve performance, we could consider reducing the number of zero-pixel quads that are passed through the latter half of the pipeline. At the cost of additional computation, a test could be performed before shading and sampling that tested whether the quad covered any fragments. Alternatively, at the cost of slightly more artifacts due to multiple fragments covered by the same quad, the stop length could be increased, resulting in a significant decrease in the number of quads and hence a decrease in runtime. Increasing the stop length from 1 to 1.5 for PIN, for instance, cuts the number of quads produced by more than a factor of 2 (1,121k to 486k). Doubling the stop length to 3 further decreases the number of quads (to 121k).

Figure 8.6 shows the performance impact of varying the stop length. Subdivision and geometry/vertex operations decrease with an increased stop length. Because the number of quads decreases (though the total number of fragments covered does not), rasterization work also declines, although not at the same rate. Composition is unaffected.

Also important are datasets that are well-behaved under subdivision. Many of our

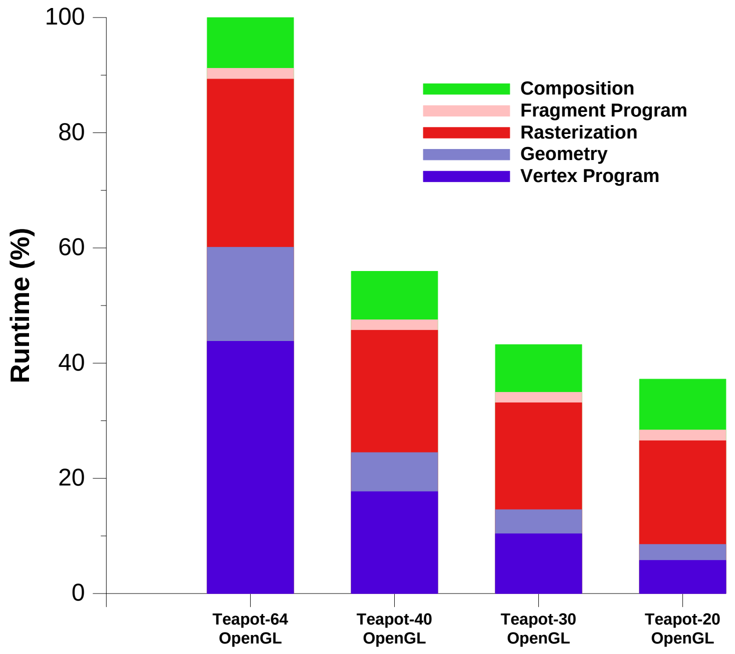


Figure 8.7: TEAPOT performance at several subdivision levels, normalized to TEAPOT-64 = 100%. The benchmark described in Section 5.1 is TEAPOT-20.

patches, when subdivided, generated quads with irregular aspect ratios that covered no pixels. Partially this is because when we subdivide a quad, we always generate 4 quads; at the cost of additional computation, subdividing in only one direction instead of both would significantly aid the quality of the generated quads. Choosing both a subdivision scheme that produces well-behaved data and a dataset that conforms well to the subdivision scheme is vital for achieving high efficiency.

8.4.3 OpenGL: Triangle Size Effects

Similarly, OpenGL performance degrades as triangles become smaller. Figure 8.7 shows TEAPOT's performance at different subdivision levels. As triangles shrink, the vertex program, geometry, and rasterization cost grows rapidly. TEAPOT-64, with Reyes-sized primitives (an average triangle size of 1.18 pixels), has more than twice the runtime of TEAPOT-20, our benchmark scene.

Small triangles make OpenGL's performance suffer for the same reasons that Reyes' performance is poor. The shading work increases with the number of triangles, and much of the rasterization work is also per-triangle.

8.4.4 Toward Reyes in Hardware

Many parts of our pipeline are well-suited for programmable stream hardware such as Imagine. The vertex programs for our three Reyes scenes, for instance, sustain an average of 24.5 operations per cycle in their main loops. The sampling algorithm is also efficient, and both would benefit in future stream hardware from more functional units to exploit further levels of instruction-level parallelism.

But subdivision cost dominates the runtime of our Reyes scenes, so continued investigation of subdivision algorithms and hardware is vital. The ideal subdivider has several properties:

Adaptive Uniform subdivision, while simple to implement, is inappropriate for a general subdivider. Uniformly subdividing a patch with part of that patch requiring a fine subdivision means that the entire patch will also be divided finely. This could lead to a huge number of produced quads, most of which would not contribute to the final image.

High performance Ideally, the subdivider would not dominate the runtime of the entire scene.

Artifact free Subdividers must take care that neighboring quads at different subdivision levels do not allow cracks to form as a result of the different levels. Our algorithm, with its use of line equations to represent quad boundaries, guarantees cracks will not occur.

Efficient The ideal subdivider would not output any quads that did not contribute to the final image. Our subdivider does poorly on this point, but could potentially improve at the cost of more computation by testing for pixel coverage before outputting quads or by improving quad quality by allowing subdivision in only one direction.

In the future, we hope to explore other subdivision algorithms that might better address some of the above points. As well, other subdivision schemes and algorithms may be better candidates for our hardware and programming system. For example, Pulli and Segal explore a Loop subdivision scheme that is amenable to hardware acceleration [Pulli and Segal, 1996]; Bischoff et al. exploit the polynomial characteristics of the Loop scheme with another algorithm for efficient subdivision [Bischoff et al., 2000].

Investigating what functional units and operations would allow stream hardware to better perform subdivision would be an interesting topic for future research. Alternatively, our pipelines are implemented in programmable hardware, but due to its large computational costs and regular computation, subdivision may be better suited for special purpose hardware. Hybrid stream-graphics architectures, with high-performance programmable stream hardware evaluating programmable elements such as shading and special-purpose hardware performing fixed tasks such as subdivision, may be attractive organizations for future graphics hardware.

8.5 Conclusion

We have shown that although Reyes has several desirable characteristics—bounded-size primitives, a single shader stage, and coherent access textures—the cost of subdivision in the Reyes pipeline allows the OpenGL pipelines to demonstrate superior performance. Continued work in the area of efficient and powerful subdivision algorithms is necessary to allow a Reyes pipeline to demonstrate comparable performance to its OpenGL counterpart.

As triangle size continues to decrease, Reyes pipelines will look more attractive. And though the shaders we have implemented are relatively sophisticated for today's real-time hardware, they are much less complex than the shaders of many thousands of lines of code used in movie production. When graphics hardware is able to run such complex shaders in real time, and the cost of rendering is largely determined by the time spent shading, we must consider pipelines such as Reyes that are designed for efficient shading.

Furthermore, as graphics hardware becomes more flexible, multiple pipelines could be supported on the same hardware, as we have done with our implementation on Imagine. Both the OpenGL and Reyes pipelines in our implementation use the same API, the Stanford Real-Time Shading Language, for their programmable elements. Such flexibility will allow graphics hardware of the future to support multiple pipelines with the same interface or multiple pipelines with multiple interfaces, giving graphics programmers and users a wide range of options in both performance and visual fidelity.

Chapter 9

Conclusions

Computer graphics, a complex task with high computational demand, is an increasingly important component in modern workloads. Current systems for high-performance rendering typically use special purpose hardware. This hardware began as a fixed rendering pipeline and, over time, has added programmable elements to the pipeline.

In this dissertation, we take a different direction in the design and implementation of a rendering system. We begin with a programming abstraction, the stream programming model, and a programmable processor, the Imagine stream processor, and investigate their performance and suitability for the rapidly evolving field of computer graphics.

We find that the stream programming model is well suited to implement the rendering pipeline, that it both efficiently and effectively utilizes the stream hardware and is flexible enough to implement the variety of algorithms used in constructing the pipeline. Because of its programmability, such an architecture and programming system is also well positioned to handle future rendering tasks as we move toward graphics solutions that have the real-time performance of today's special-purpose hardware and the photorealism and complexity of the best of today's computer-generated motion pictures.

9.1 Contributions

The contributions of this dissertation are in several areas.

The Rendering Pipeline First, the development of the stream framework for the rendering pipeline and the algorithms used in the kernels that comprised it are significant advances in the field of stream processing. In particular, the rasterization pipeline and the mechanism for preserving ordering were difficult problems that are solved with efficient and novel algorithms. The framework is suited for not only a single pipeline but also alternate pipelines as well as hybrid pipelines; in this work we describe two complete pipelines, an OpenGL-like pipeline and a Reyes-like pipeline.

The algorithms employed also exploit the native concurrency of the rendering tasks, effectively utilizing both the instruction-level and data-level parallelism inherent in the tasks. Finally, the shading language backend that generates the vertex and fragment programs produces efficient Imagine kernel and stream code while still allowing shader descriptions in a high-level language.

Performance Analysis As a result of this dissertation work, we identify several important factors in achieving high performance on a stream implementation. One of the primary goals of any stream implementation must be to make the internal streams as long as possible while not spilling to memory. We see that, absent special purpose hardware, stream implementations on stream hardware are likely to remain fill-limited for OpenGL-like pipelines. We compare the barycentric and scanline rasterizers and conclude that while scanline rasterizers have superior performance today, the trends of more interpolants and smaller triangles lead to barycentric implementations becoming more attractive as those trends progress. And we observe that, with computation-to-bandwidth ratios like Imagine's, computation, rather than memory bandwidth, is the limit to achieving higher performance. Finally, the task of adaptive subdivision is a significant challenge to efficient implementations of pipelines such as Reyes that must perform tessellation as part of their pipelines.

Scalability The implementation is well-suited to future generations of stream hardware that feature more capable hardware: more functional units per cluster, more communication between clusters, more clusters, and larger stream register files. We see that increasing any of these metrics increases the performance of our pipeline, and moreover, that these increases are orthogonal. In particular, the addition of more clusters and with them, data-level parallelism, offers near-linear speedups, limited only by the increasing demand on memory bandwidth.

9.2 Imagine Status

The work described in this dissertation has been validated using cycle-accurate simulations of the Imagine Stream Processor. However, a prototype of Imagine has been developed concurrently with Imagine's tool and applications and with this dissertation. A silicon implementation of Imagine was delivered on 1 April 2002. At the time of submission of this dissertation (October 2002), the Imagine prototype was running several complete stream applications with a clock speed of 288 MHz. Continued board and application development and debugging, in addition to efforts to increase Imagine's clock speed and host bandwidth, are actively under way. The rendering pipelines described in this dissertation have not yet been tested on the Imagine prototype.

9.3 Future Research

This work naturally leads to a number of future directions that may lead to further advances in stream algorithms and implementations and lead to graphics systems that deliver even higher performance with even more flexibility.

Higher order surfaces OpenGL systems today typically use triangle meshes or vertex arrays as their primary input primitive. However, we have demonstrated that vertices account for the major portion of bandwidth in our scenes.

Reducing bandwidth is an important goal. Reyes uses higher-order surfaces to achieve a bandwidth reduction in input primitives, but an efficient implementation on stream hardware has been elusive. Future work could identify particular surface representations and algorithms that are best suited for stream hardware and the stream programming model.

More complex shaders Our shaders have simple control flow and are perfectly parallel. Future shaders will surely have more complex control flow, and rethinking and designing the necessary control constructs to support them is a vital direction for future hardware. Should the ideal shader architecture support arbitrary programmability, with pointers, procedure calls, and a high-level abstraction? The rapid improvement in shader programmability will bring these issues to the forefront in the near future.

Imagine tools and hardware support In the course of this work, Imagine's tools were constantly improved to support more complicated tasks, but both the software and the hardware to support it could continue to improve in several ways. Better register allocation algorithms would greatly aid kernel development; automatic software pipelining and the compile-time allocation decisions would help at the stream level. Dynamic allocation of stream resources (such as partial double buffering or efficient spilling) requires significant change in both the tools and the hardware that supports them. Finally, integration of more host capabilities onto the stream processor die will remove many of the host effects we see in our results.

Cluster architecture Imagine's clusters were optimized for very different tasks than graphics. Though the instruction set and functional unit mix have both proven to be applicable to rendering tasks, designing a stream machine optimized for graphics may well have a very different functional unit and instruction mix. What primitives could improve the efficiency of rendering tasks and still be applicable across the entire pipeline? Would increasing the cluster word width to 128 bits (as in NVIDIA vertex hardware) cause a corresponding increase in performance?

System organization We discussed the relevant differences between the task-parallel and time-multiplexed organizations. Further analysis of these differences, and the ramifications of combining the two organizations, may well lead to superior architectures in the next generation of graphics hardware. As well, investigation of the best way to apply special-purpose components to future graphics hardware, particularly in the context of these two organizations, will be a fruitful research direction.

Multi-node stream systems Finally, we have not explored the next axis of parallelism: the task level. The stream programming model is well-suited to identifying the necessary communication paths in a multi-node implementation; adapting this work to such an implementation will allow ever higher levels of performance and spur the continued development of suitable algorithms and tools for stream architectures.

Bibliography

[Apodaca and Gritz, 2000] Anthony A. Apodaca and Larry Gritz. *Advanced Renderman*, chapter 11 (Shader Antialiasing). Morgan Kaufmann, 2000.

[Bier et al., 1990] 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, October 1990.

[Bischoff et al., 2000] Stephan Bischoff, Leif P. Kobbelt, and Hans-Peter Seidel. Towards Hardware Implementation Of Loop Subdivision. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 41–50, August 2000.

[Brown, 1999] Russ Brown. Barycentric Coordinates as Interpolants. 1999.

[Catmull and Clark, 1978] E. Catmull and J. Clark. Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes. *Computer-Aided Design*, 10(6):350–355, September 1978.

[Cook et al., 1987] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 95–102, July 1987.

[Cook, 1984] Robert L. Cook. Shade Trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 223–231, July 1984.

[Cook, 1986] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[Dally and Poulton, 1998] William J. Dally and John W. Poulton. *Digital Systems Engineering*, chapter 1. Cambridge University Press, 1998.

[Deering and Nelson, 1993] Michael F. Deering and Scott R. Nelson. Leo: A System for Cost-effective 3D Shaded Graphics. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108, August 1993.

[Diefendorff, 1999] Keith Diefendorff. Sony's Emotionally Charged Chip. *Microprocessor Report*, 13(5), 1999.

[Fisher, 1983] J. A. Fisher. Very Long Instruction Word Architectures and ELI-512. In *Proceedings of the 10th Symposium on Computer Architecture*, pages 478–490, 1983.

[Foley, 1996] Pete Foley. The Mpact$^{\text{TM}}$ Media Processor Redefines the Multimedia PC. In *Proceedings of IEEE COMPCON*, pages 311–318, 1996.

[Fuchs et al., 1989] 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. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 79–88, July 1989.

[Gordon et al., 2002] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[Hakura and Gupta, 1997] 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.

[Hanrahan and Akeley, 2001] Pat Hanrahan and Kurt Akeley. CS 448: Real Time Graphics Architecture. Fall Quarter, 2001.

[Hanrahan and Lawson, 1990] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 289–298, August 1990.

[Humphreys et al., 2002] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693–702, July 2002.

[Igehy et al., 1998a] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, August 1998.

[Igehy et al., 1998b] Homan Igehy, Gordon Stoll, and Patrick M. Hanrahan. The Design of a Parallel Graphics Interface. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 141–150, July 1998.

[Kapasi et al., 2000] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 159–170, 2000.

[Kapasi et al., 2001] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, December 2001.

[Kapasi et al., 2002] Ujval J. Kapasi, William J. Dally, Brucek Khailany, John D. Owens, and Scott Rixner. The Imagine Stream Processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[Khailany et al., 2001] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

[Khailany et al., 2003] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, John D. Owens, and Brian Towles. Exploring the VLSI Scalability of Stream Processors. In *Proceedings of the Ninth Annual International Symposium on High-Performance Computer Architecture*, 2003.

[Kozyrakis, 1999] Christoforos Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems. Technical Report CSD-99-1059, University of California, Berkeley, 1999.

[Lamport, 1974] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Levinthal and Porter, 1984] Adam Levinthal and Thomas Porter. Chap – A SIMD Graphics Processor. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 77–82, Minneapolis, Minnesota, July 1984.

[Levinthal et al., 1987] 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.

[Lindholm et al., 2001] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A User-Programmable Vertex Engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158, August 2001.

[Loveman, 1977] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[Mattson et al., 2000] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication Scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–92, 2000.

[Mattson, 2002] Peter Mattson. *Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[McCool, 2001] Michael McCool. SMASH: A Next-Generation API for Programmable Graphics Accelerators. Technical Report CS-2000-14, Department of Computer Science, University of Waterloo, 20 April 2001. API Version 0.2.

[McCormack et al., 1998] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon — A Single-Chip 3D Workstation Graphics Accelerator. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 123–132, August 1998.

[Möller and Haines, 1999] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, 1999.

[Molnar et al., 1992] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 231–240, July 1992.

[Montrym et al., 1997] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 293–302, August 1997.

[Müller and Havemann, 2000] Kerstin Müller and Sven Havemann. Subdivision Surface Tesselation on the Fly using a versatile Mesh Data Structure. *Computer Graphics Forum*, 19(3), August 2000.

[Nocedal and Wright, 1999] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 7 (Calculating Derivatives), pages 164–191. Springer Verlag, 1999.

[NVIDIA, 2001] NVIDIA Corporation. NVIDIA OpenGL Extension Specifications, 6 March 2001.

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

[Olano and Greer, 1997] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 89–96, August 1997.

[Olano and Lastra, 1998] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 159–168, July 1998.

[Olano, 1998] Marc Olano. *A Programmable Pipeline for Graphics Hardware*. PhD thesis, University of North Carolina, Chapel Hill, 1998.

[Owens et al., 2000] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[Owens et al., 2002] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a Stream Architecture. In *2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, September 2002.

[Peercy et al., 2000] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive Multi-Pass Programmable Shading. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics

Proceedings, Annual Conference Series, pages 425–432, July 2000.

[Peleg and Weiser, 1996] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture — Improving multimedia and communications application performance by 1.5 to 2 times. *IEEE Micro*, 16(4):42–50, August 1996.

[Pino, 1993] José Luis Pino. Software Synthesis for Single-Processor DSP Systems Using Ptolemy. Master's thesis, University of California, Berkeley, May 1993.

[Proudfoot et al., 2001] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001.

[Pulli and Segal, 1996] Kari Pulli and Mark Segal. Fast Rendering of Subdivision Surfaces. In *Rendering Techniques '96 (Proceedings of the 7th Eurographics Workshop on Rendering)*, pages 61–70, 1996.

[Purcell et al., 2002] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):703–712, July 2002.

[Rixner et al., 1998] 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.

[Rixner et al., 2000a] 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*, pages 128–138, June 2000.

[Rixner et al., 2000b] 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.

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

[Taylor et al., 2002] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, March/April 2002.

[Tremblay et al., 1996] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing — Enhancing conventional RISC instruction sets to significantly accelerate media-processing algorithms. *IEEE Micro*, 16(4):10–20, August 1996.

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

[Ward, 1991] Greg Ward. *Graphics Gems II*, chapter VIII. 10 (A Recursive Implementation of the Perlin Noise Function), pages 396–401. AP Professional, 1991.

[Warnock, 1969] John Warnock. *A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures*. PhD thesis, University of Utah, 1969. Computer Science Department TR 4-15, NTIS AD-753 671.

[Williams, 1983] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, volume 17, pages 1–11, July 1983.

[Zorin and Schröder, 2000] Denis Zorin and Peter Schröder. *Subdivision for Modeling and Animation: Implementing Subdivision and MultiResolution Surfaces*, chapter 5, pages 105–115. Siggraph 2000 Course Notes, 2000.