International Journal of Innovative Technology & Adaptive Management (IJITAM)
ISSN: 2347-3622, Volume-2, Issue-1, October, 2014

IO Stream Architecture Of Graphics In Remote Sensor Device

[1]Bipasha Mallick
Assistant Professor,
Haldia Institute Of Technology
bipasm@gmail.com

[2]Dipankar Pramanik
CSE,AIET
prdipu@gmail.com

Abstract- This article related to the graphical design of remote sensor device in terms of application areas in computer Science. It is concerned with the graphical design of remote sensor devices. Remote device applications are the most demanding applications in Gaming technology, Mobile Technology, Micro technology and graphics. This article describes a system that will deliver both high performance and high flexibility, one that will meet the demands of a wide variety of graphical applications in future computing systems. To accomplish this goal, we implement our graphical system on a computer system with both software and hardware components.

Keywords- *Remote device application, Stream architecture of stream programming model, Imagines' programming system*

1. Introduction

Remote device applications are the most demanding applications in Gaming technology, Mobile Technology, Micro technology and graphics. That is the study of constructing computer-generated images by remote sensor device 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. 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. 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 takes on the order of hours. An example is computer-generated games controlled by joy-stick These stick 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, I 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, I use the Imagine stream processor and its programming tools as a basis for implementation.

2. Stream Architectures

The Stream Programming Model

In the stream programming model, the data primitive is a *stream*, an ordered set of data of an arbitrary data type. 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 that allows more complex control flow within kernels. 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.

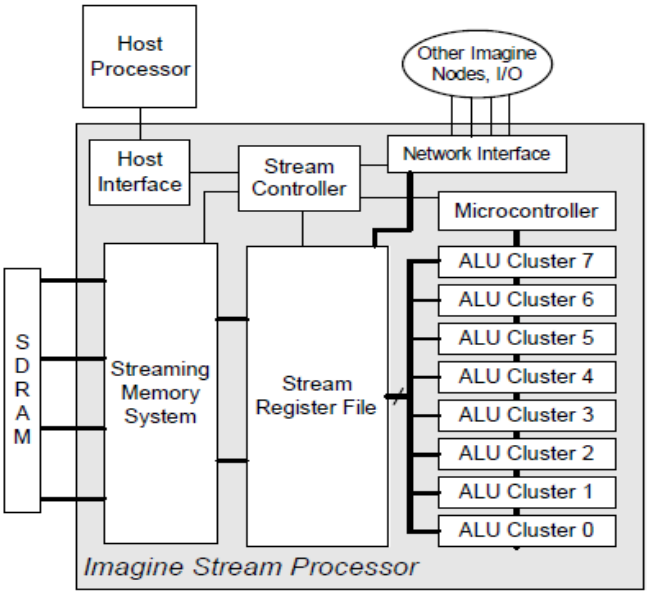
▣ Streams and Media Applications

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. 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.

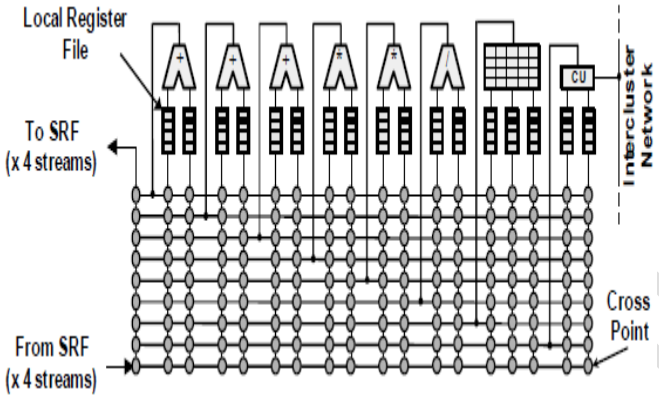


The Imagine Stream Processor block diagram.

▣ The Imagine Stream Processor

The Imagine stream processor is a hardware architecture designed to implement the stream programming model.

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 stream outputs.



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 inter cluster communication unit.

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, 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 sub word 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.

▣ Imagine's Programming System

■ 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. 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 statement. It also does not support subroutine calls. Stream inputs and outputs are specified using the >> and << operators, espectively. KernelC programs are compiled using a kernel scheduler that maps the kernels onto the Imagine clusters, producing microcode that is loaded during program execution into the microcontroller's program store.

■ 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 "strip-mining"; 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.

■ **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 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.

▫ **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. 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.

3. Results and Analysis

In this chapter we analyze the performance of our implementation on the Imagine Stream Processor. I begin by describing our test scenes and our experimental setup. I 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, I characterize the kernels by performance limit and scalability and compare the theoretical bandwidth requirements against the achieved performance of our implementation.

◙ **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.

Test Scenes

To facilitate comparison between scenes, each of these scenes was rendered into a 24-bit
RGB color frame buffer with a window size of 720 × 720 pixels.

Remote 1: Joystick 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.

Remote 2: Mouse The Mouse is the first frame of the SPECviewperf 6.1.1 Advanced Visualizer benchmark with lighting and blending disabled and all frequency pointsampled from a 512 × 512 texture map.

Remote 3: Keyboard The Keyboard is identical to mouse, except all frequency lookups are mipmapped with 8 samples per fragment.

Remote 4: Pin-pen The bowling pin shader has 5 fragment frequency applied to it as well as a procedurally specified light with diffuse and specular components. Each frequency is sampled once per fragment.

Remote 5: Pin-pen8 is identical to Pin-pen, except all five frequency lookups are mipmapped with 8 samples per fragment.

Remote 6: Joy board uses the same geometry as Pin-pen but uses procedural turbulence involving 4 noise calculations per fragment to generate its appearance. The fragment program applies over 1200 operations to each fragment.

Remote 7: Projector-remote perturbs the positions and normals of each vertex of a 200 × 200 tessellated sphere using a single combined per-vertex displacement/bump map frequency 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.

Remote 1: Joystick	Remote 2: Mouse	Scene 3: Keyboard	Remote4: Pin pen	Remote 5: Pin pen8	Remote 6: Joy board	Remote 7: Projector remote
						

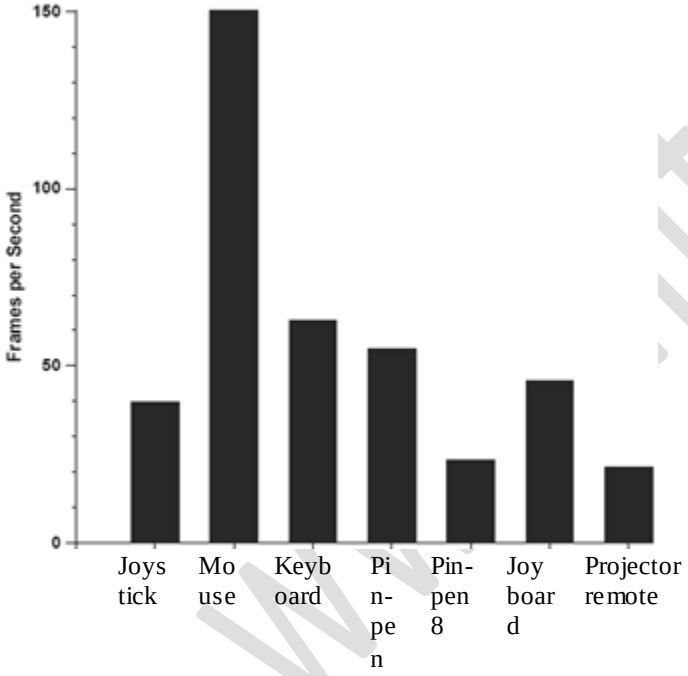
Vertices	245,760	62,576	62,576	80,000	80,000	80,000	80,000
Triangles	81,920	25,704	25,704	80,000	80,000	80,000	80,000
Fragments	180,909	70,384	70,384	91,591	91,591	91,594	266,810
Frags/tri	4.4	5.5	5.5	2.3	2.3	2.3	6.7
Interpolants	5	4	4	16	16	13	20
Per-V BW[a]	32	28	28	44	44	32	84
Per-F BW[b]	12	16	44	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

Remote Characteristics.

▣ Results

▣ Performance and Occupancy

We begin by looking at the performance in frames per second of our test scenes. Above figure 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, Mouse, runs at over 150 frames per second. Next I compare the performance of a subset of these scenes1 to representative hardware and software graphics implementations. 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.

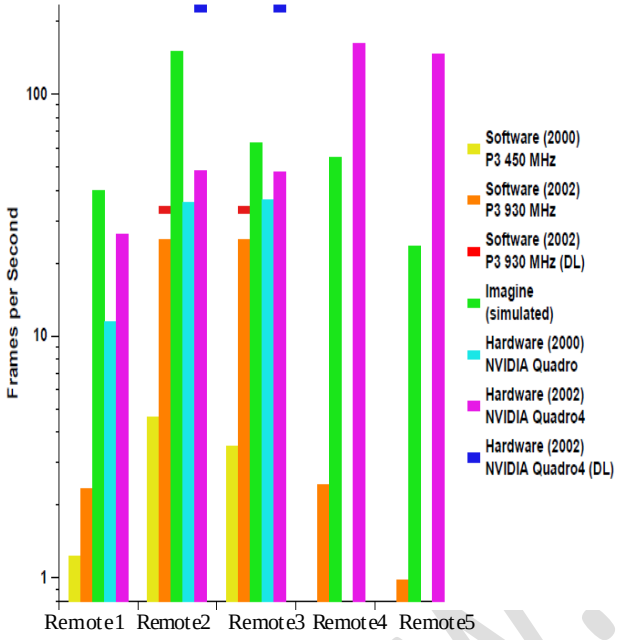


Test Remote: Frames per Second. All data was acquired using the cycle-accurate simulator for an Imagine stream processor running at 500 MHz.

The two other configurations are typical configurations of the year 2000. The hardwareaccelerated 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 I measured the performance on these scenes by implementing them in OpenGL. On the PC systems, I ran these from immediate-mode arrays of vertices in system memory or through Kekoa Proudfoot's lightweight RTSL scene viewer, using GL vertex arrays (Pin-pen), 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.

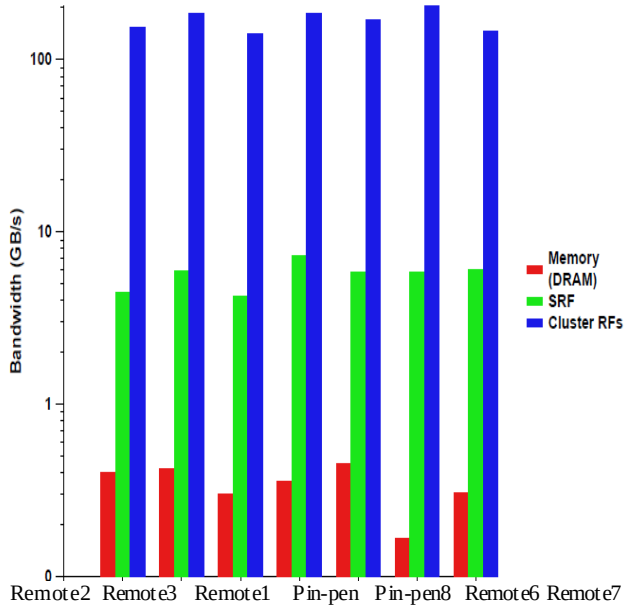
Refresh synchronization costs were eliminated by disabling the vertical retrace sync, allowing a new frame to begin immediately after the old frame completed. All windows were single-buffered and neither flushes nor buffer clears were used. The Mouse scenes, because they appeared to be limited by host 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 bellow 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). Imagines' 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 internal clock), runs at worst 3 times slower than Imagine and at best more than 6 timesfaster, with a geometric mean of 23.9% faster. The

two display list tests (on the Remote device 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. 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 Mouse scene, resulting in Keyboard, more than halved the runtime.

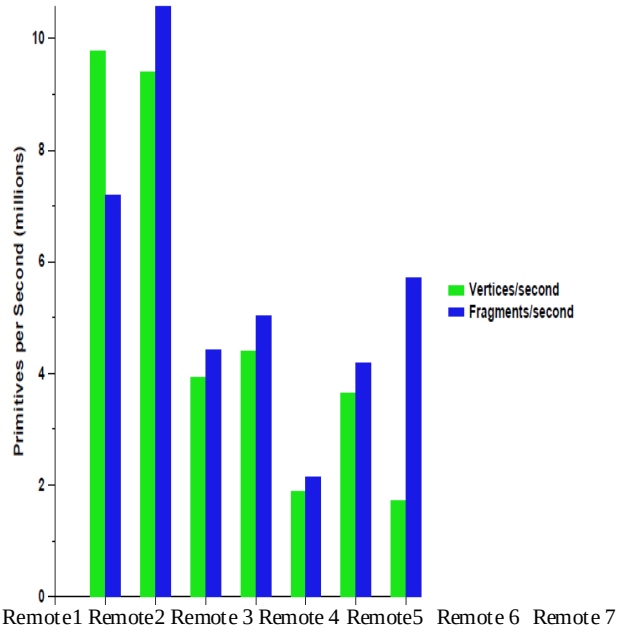


Test Remote: Comparison against Commercial Systems. Imagine is compared against representative commercial software and hardware implementations of the years 2000 and 2002. The Remote device scene was also measured with display lists (marked DL) for both 2002 commercial configurations.

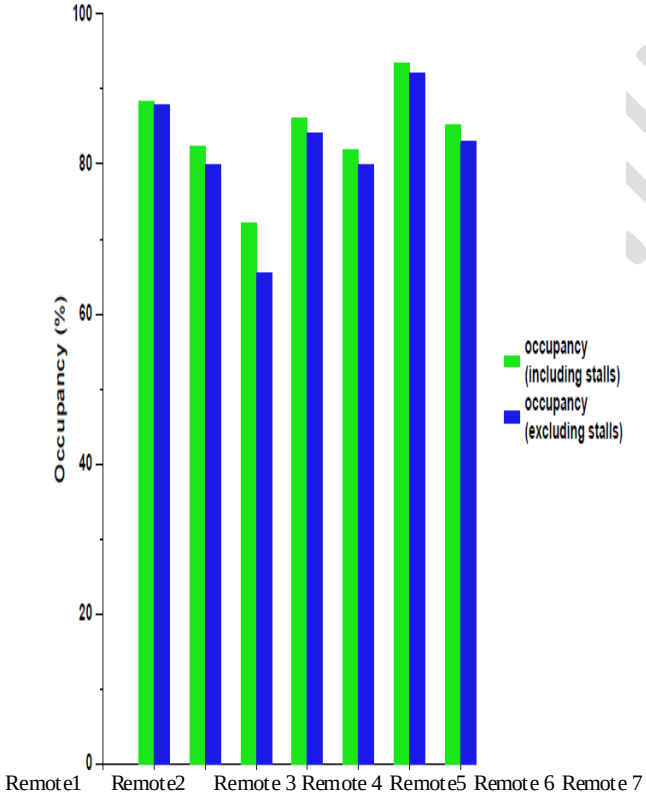
And we can see that on the pin dataset, the complex noise calculation in Projector-remote is slightly more expensive than the 5 point-sampled frequency lookups in Pin-pen, but less than half the cost of mapping each of those 5 texture lookups in Pin-pen8. Another measure of system performance is the achieved vertex and fragment rates of the system, shown in above figure. 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 (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 fragments2. I 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 Joystick (9.8 million vertices per second) and for fragment rate on Mouse (10.6 million fragments per second).



Test Remote: 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.



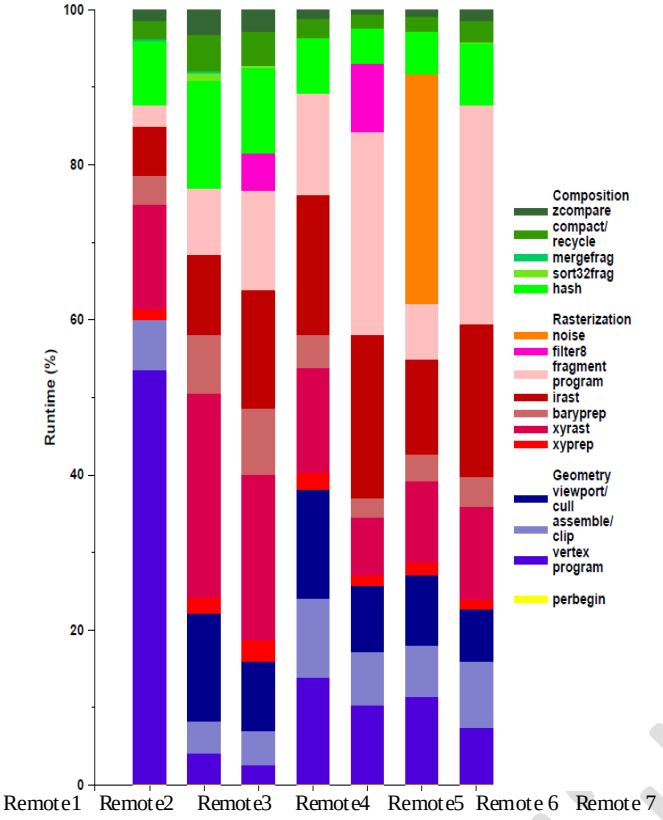
Test Remote: Primitives per Second. Bars show each scene's delivered vertexper-second (green) and fragment-per-second (blue) rates.



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

Memory bandwidth hierarchy is a cornerstone of the efficiency of stream architectures. 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 frequency scenes exhibit a higher memory bandwidth than non-textured ones, and the heavy per-vertex work in Joystick and per-fragment work in Joy-board push down their main memory demands. For efficiency, then, we would like to keep the clusters as busy as possible. Above figure 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 suffers3. Several other reasons contribute to a non-ideal occupancy. A minimum kernel execution time 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.



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.

4. 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.

Computer architecture of sensor device, 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, I take a different direction in the design and implementation of a rendering system. I 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 remote sensor device 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.

◙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 of 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, I 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 OpenGLlike 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. I 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, datalevel parallelism, offers near-linear speedups, limited only by the increasing demand on memory bandwidth.

5. References

[1] Apodaca and Gritz, 2002
Anthony A. Apodaca and Larry Gritz. Advanced Renderman, chapter 11 (Shader Antialiasing). Morgan Kaufmann.

[2] Bier et al., 1999
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.

[3] Bischoff et al., 2005
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.

[4] Brown, 1998
Russ Brown. Barycentric Coordinates as Interpolants.

[5] Catmull and Clark, 1996
E. Catmull and J. Clark. Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes. Computer-Aided Design, 10(6):350–355.

[6] Cook et al., 2007
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.

[7] Cook, 1994
Robert L. Cook. Shade Trees. In Computer Graphics (Proceedings of SIGGRAPH 84), volume 18, pages 223–231.

[8] Cook, 1996
Robert L. Cook. Stochastic Sampling in Computer Graphics. ACM Transactions on Graphics, 5(1):51–72.

[9] Dally and Poulton, 2008 William J. Dally and John W. Poulton. Digital Systems Engineering, chapter 1. Cambridge University Press.

[10] Deering and Nelson, 2003
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.