

Pomegranate: A Fully Scalable Graphics Architecture

Matthew Eldridge

Homan Igehy

Pat Hanrahan

Stanford University*

Abstract

Pomegranate is a parallel hardware architecture for polygon rendering that provides scalable input bandwidth, triangle rate, pixel rate, texture memory and display bandwidth while maintaining an immediate-mode interface. The basic unit of scalability is a single graphics pipeline, and up to 64 such units may be combined. Pomegranate's scalability is achieved with a novel "sort-everywhere" architecture that distributes work in a balanced fashion at every stage of the pipeline, keeping the amount of work performed by each pipeline uniform as the system scales. Because of the balanced distribution, a scalable network based on high-speed point-to-point links can be used for communicating between the pipelines.

Pomegranate uses the network to load balance triangle and fragment work independently, to provide a shared texture memory and to provide a scalable display system. The architecture provides one interface per pipeline for issuing ordered, immediate-mode rendering commands and supports a parallel API that allows multiprocessor applications to exactly order drawing commands from each interface. A detailed hardware simulation demonstrates performance on next-generation workloads. Pomegranate operates at 87–99% parallel efficiency with 64 pipelines, for a simulated performance of up to 1.10 billion triangles per second and 21.8 billion pixels per second.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing

Keywords: Graphics Hardware, Parallel Computing

1 Introduction

The performance of interactive graphics architectures has been improving at phenomenal rates over the past few decades. Not only have the speed improvements kept up with or exceeded Moore's Law, but each successive generation of graphics architecture has expanded the feature set. Despite these great improvements, many applications cannot run at interactive rates on modern hardware. Examples include scientific visualization of large data sets, photo-realistic rendering, low-latency virtual reality, and large-scale display systems. A primary goal in graphics research is finding ways to push this performance envelope, from the details of the chip architecture to the overall system architecture.

The past few years have also marked a turning point in the history of computer graphics. Two decades ago, interactive 3D graph-

*{eldridge,homan,hanrahan}@graphics.stanford.edu

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

SIGGRAPH 2000, New Orleans, LA USA
© ACM 2000 1-58113-208-5/00/07 ...\$5.00

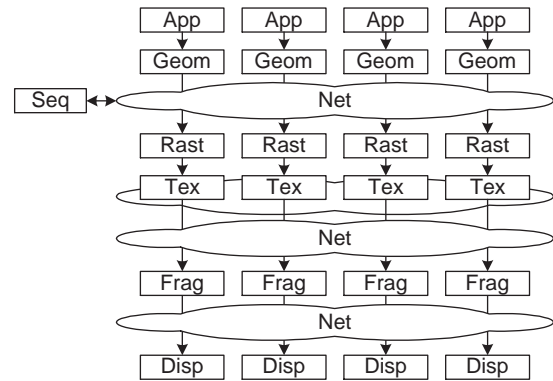


Figure 1: Each Pomegranate pipeline is composed of five stages: geometry (Geom), rasterization (Rast), texture (Tex), fragment (Frag) and display (Disp). A network (Net) connects the pipelines and a sequencer (Seq) orders their execution of multiple graphics streams submitted by the application threads (App).

ics systems were found only at large institutions. As semiconductor technologies improved, graphics architects found innovative ways to place more functionality on fewer chips, and interactive graphics workstations made their way to the desktops of engineers. Today, the entire graphics pipeline can be placed on a single chip and sold at a mass-market price point. Because of the enormous economies of scale afforded by commoditization, this trend has a significant impact on how high-end systems must be built: it is much more cost effective to design a single low-end, high-volume system and replicate it in an efficient manner in order to create high-end, low-volume systems. For example, supercomputers used to be designed with unique, proprietary architectures and esoteric technologies. With the commoditization of microprocessors, these designs were replaced by highly parallel multiprocessor systems that made use of microprocessor technology. The Pomegranate architecture provides a way of scaling the base unit of a single graphics pipeline to create higher performance systems.

Pomegranate is composed of n graphics pipelines interconnected by a scalable point-to-point network, as depicted in figure 1. Each pipeline accepts standard, immediate-mode OpenGL commands from a single context as well as parallel API commands for ordering the drawing commands of the context with the drawing commands of other contexts. As with any parallel system, Pomegranate will only operate efficiently if the load is balanced across its functional units. However, graphics primitives can vary substantially in the amount of processing time they require. Furthermore, the amount of work a primitive will require is not known a priori. Distributing and balancing this workload in a dynamic fashion while minimizing work replication is a key innovation of the Pomegranate architecture and directly contributes to its scalability. A novel serial ordering mechanism is used to maintain the order specified by the OpenGL command stream, and a novel parallel ordering mechanism is used to interleave the work of multiple graphics contexts. Because the use of broadcast communication is minimized in both

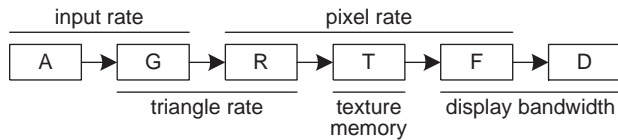


Figure 2: The serial graphics pipeline consists of an application (A), a geometry processor (G), a rasterizer (R), a texture processor (T), a fragment processor (F) and a display processor (D). The units with a direct impact on each scalability measure are underlined.

the data distribution and the ordering, Pomegranate is able to scale to a high degree of parallelism.

In addition to scalability, an equally important characteristic of the Pomegranate architecture is its compatibility with a modern graphics API. OpenGL has strict ordering semantics, meaning that all graphics commands must appear to execute in the order they are specified. For example, two overlapping polygons must appear on the screen in the order they were submitted by the user, and a state change command applies to all subsequent primitives. This constraint forces any parallel OpenGL hardware architecture to be capable of maintaining the serial order specified by the application. This restriction is one of the major obstacles to building a scalable OpenGL hardware renderer. As an analogy, C has become the de facto standard for programming, and as a result microprocessor architects focus the bulk of their efforts addressing the difficulties it introduces — pointer aliasing, limited instruction-level parallelism, strict order of operations, etc. Similarly, we felt it was important to design within the ordering constraints of OpenGL. In addition to specifying ordered semantics, OpenGL is an immediate-mode interface. Commands that are submitted by the application are drawn more or less immediately thereafter. APIs that are built around display lists, scene graphs, or frame semantics all provide the opportunity for the hardware to gather up a large number of commands and partition them among its parallel units. An immediate-mode interface does not enable this approach to extracting parallelism, and thus provides a further challenge.

A fully scalable graphics architecture should provide scalability on the five key metrics depicted in figure 2: input rate, triangle rate, rasterization rate, texture memory and display bandwidth.

- *Input rate* is the rate at which the application can transmit commands (and thus primitives) to the hardware.
- *Triangle rate* is the rate at which geometric primitives are assembled, transformed, lit, clipped and set up for rasterization.
- *Pixel rate* is the rate at which the rasterizer samples primitives into fragments, the texture processor textures the fragments and the fragment processor merges the resultant fragments into the framebuffer.
- *Texture memory* is the amount of memory available to unique textures.
- *Display bandwidth* is the bandwidth available to transmit the framebuffer contents to one or more displays.

Pomegranate provides near-linear scalability in all five metrics while maintaining an ordered, immediate-mode API.

We motivate our discussion of Pomegranate by suggesting two possible implementations: a scalable graphics pipeline and a multi-pipeline chip. A scalable graphics pipeline could be flexibly deployed at many levels of parallelism, from a single pipeline solution with performance comparable to a modern graphics accelerator up to a 64 pipeline accelerator with “supercomputer” performance.

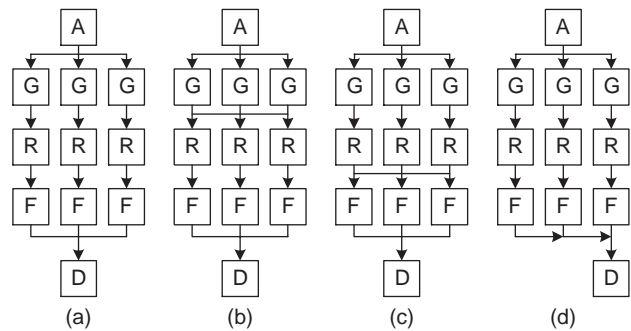


Figure 3: Sort-first (a) sorts triangles before the geometry stage. Sort-middle (b) sorts triangles between geometry and rasterization. Sort-last fragment (c) sorts fragments between rasterization and fragment processing. Sort-last image (d) sorts pixels between fragment processing and the display. The texture stage has been eliminated in this diagram, but in practice will either be located at the end of the rasterization stage or the beginning of the fragment stage.

The incremental cost of the Pomegranate pipeline over a traditional graphics pipeline is the area required for approximately 1MB of buffering, 256KB for supporting 64 contexts, and the area and pins of a high-speed network interface. We estimate that the incremental cost of the Pomegranate pipeline is an additional 200 pins and 50mm² for memory in a modern 0.18μm process. A network chip, replicated as necessary to interconnect all of the pipelines, would weigh in at approximately 1000 pins, which is feasible. Our second possible implementation is a single chip with multiple Pomegranate pipelines. In such a chip the Pomegranate architecture would be leveraged as a practical method for using the hundreds of millions of transistors which will soon be practical in even a consumer-level graphics accelerator. Current graphics accelerators already stretch the capabilities of VLSI tools and engineers with their size and complexity. Pomegranate would enable the design of a comparatively smaller pipeline which could then be replicated to consume the available transistor count, rather than requiring the design of a huge monolithic pipeline.

In this paper, we will first briefly review previous work in parallel graphics architectures. Then, we will give an overview of the Pomegranate architecture and the details of its key components, describing how work is distributed in a balanced way at each stage of the pipeline to give scalable performance in each of the five metrics. Next, we describe the serial ordering algorithm that maintains the serial order mandated by a single OpenGL context as well as a parallel ordering algorithm that interleaves work according to the order specified by a parallel API. Finally, we present results from a detailed hardware simulation that demonstrates Pomegranate’s scalability and compares it to traditional parallel graphics architectures.

2 Background

2.1 Parallel Graphics Architectures

There are a number of published systems that use parallelism to achieve high performance levels. How this parallelism is organized has a direct effect on the scalability of these architectures. Molnar et al. describe a taxonomy for classifying parallel rendering architectures as sort-first, sort-middle or sort-last based on where they transition from object-space parallelism to screen-space parallelism [9]. A variation of this taxonomy is illustrated in figure 3. All of these architectures typically exploit parallelism at each of the geometry, rasterization and fragment stages, either in object-space (assigning work by primitive) or in screen-space (assigning work by screen location). Historically, while addressing scalable triangle

rate and pixel rate, most architectures have used a single host interface, replicated texture memory across the rasterizers, and shared a single bus for display, all of which eventually limit the system's capabilities.

In a *sort-first* architecture, the screen is subdivided so that each graphics pipeline is responsible for a fraction of the pixels. The application processor distributes primitives only to the overlapping pipelines. Because the overlap computation can be time-consuming, it is usually amortized over groups of primitives. The primary advantage of this technique is its ability to use relatively standard graphics pipelines as its building block, with only glue logic for the display, and a straightforward mechanism of providing ordering. A major challenge of sort-first architectures has been the load balancing of both triangle work and pixel work. One scheme is to dynamically subdivide the screen into a small number of large tiles [12]. These schemes typically require a retained-mode interface with frame semantics so that each tile comprises an equal amount of work, and finding an efficient, accurate estimator of work is challenging. Another scheme is to subdivide the screen into a large number of small regions and either statically or dynamically assign the regions. While such schemes work with immediate-mode interfaces, minimizing overlap while balancing the load across a wide variety of workloads is difficult.

As with sort-first architectures, *sort-middle* architectures exploit image parallelism by dividing responsibility for primitive rasterization and fragment processing in image-space. However, any geometry unit is allowed to process any primitive. Thus, a sort must occur between the geometry units and the rasterizers, which are responsible for specific areas of the screen. Generally, the partitioning of the screen has been done on a very fine granularity. For example, 2-pixel wide stripes are used on the SGI Infinite Reality [11] to ensure a good load balance of pixel work across all rasterizers. While providing excellent load balancing of pixel work, these small tiles impose a high cost in redundant triangle work because every triangle is assumed to overlap every tile. This broadcast of triangle work sets an upper limit on the triangle rate the system can sustain. However, this broadcast mechanism does provide a natural point to return the primitives processed by the parallel geometry stage to their specified serial order. Larger tiles have been used to remove this broadcast limitation at the cost of large reorder buffers [7]. Minimizing redundant work due to primitives overlapping multiple tiles while efficiently addressing the temporal load imbalances of an immediate-mode API is a major challenge for these systems.

Unlike the sort-first and sort-middle architectures, *sort-last* architectures exploit object parallelism in both the geometry and rasterization stages. In *fragment sorting* architectures, any primitive may be given to any geometry unit, and each geometry unit distributes its work to a single rasterization unit. The rasterization units then distribute the resultant fragments to the specific fragment processor responsible for the corresponding region of the screen. Because each fragment is communicated only once from a single rasterizer to a single fragment processor, no broadcast is involved. The Evans & Sutherland Freedom 3000 [1] and the Kubota Denali [2] are both examples of fragment sorting architectures. The advantage of these architectures is that they potentially have greater triangle scalability than sort-middle since each triangle is processed by only one geometry and rasterization unit. However, even though a fine image-space interleaving ensures load balancing at the fragment processors, there is little or no flexibility to load balance rasterization work. If a few large primitives are submitted, they may push the system significantly out of balance as one or a few rasterizers are given much more work to do than the other rasterizers. This is problematic since primitive sizes are not known a priori.

A second variation of sort-last architectures are *image composition* architectures such as PixelFlow [10]. Multiple independent graphics pipelines render a fraction of the scene into independent

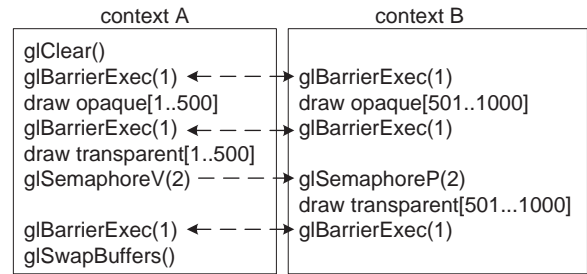


Figure 4: The scene to be rendered consists of 1000 opaque primitives and 1000 transparent primitives. The opaque primitives are rendered with depth buffering enabled, and the transparent primitives are rendered in back to front order. The pseudocode uses two contexts to submit this scene in parallel. The first barrier ensures that the clear performed by context A is complete before context B starts drawing. The second barrier ensures that all the opaque primitives are drawn before any transparent primitives. The semaphore pair ensures that context A's half of the transparent primitives are drawn first. The final barrier ensures that all drawing is done before the swapbuffers occurs.

framebuffers. Then, these framebuffers are composited based on color and depth to form a final image for display. Image composition architectures are a significant departure from the architectures discussed so far because they forfeit ordering altogether in order to scale to higher levels of performance. As with fragment sorting architectures, large primitives can cause significant load imbalance in image composition architectures. Furthermore, while the displays on the previous architectures could be made scalable using approaches similar to Pomegranate, image composition displays are difficult to scale robustly.

2.2 Parallel Interface

While building internally parallel graphics hardware is challenging in its own right, recent graphics accelerators outstrip the ability of the host interface to supply them with data (e.g. NVIDIA's GeForce256). Igehy et al. introduced a parallel API for graphics to address this bandwidth limitation [7]. The parallel API extends OpenGL with synchronization primitives that express ordering relationships between two or more graphics contexts that simultaneously submit commands to the hardware. The significance of these primitives is that they do not execute at the application level, which allows the application threads to execute past the synchronization primitives and continue submitting work. These synchronization commands are then later executed by the graphics system. This allows the programmer to order the execution of the various contexts without being reduced to using a serial interface. The primitives we focus our attention on are barriers and semaphores.

A barrier synchronizes the execution of multiple graphics contexts, ensuring that all of the commands executed by any of the contexts previous to the barrier have completed before any of the commands subsequent to the barrier have any effect. A barrier is defined with `glBarrierCreate(name, count)`, which associates a graphics barrier that has `count` contexts participating in it with `name`. A graphics context enters a barrier by calling `glBarrierExec(name)`. A semaphore provides a point-to-point ordering constraint, and acts as a shared counter. A semaphore "V" (or up) operation atomically increments the counter. A semaphore "P" (or down) operation blocks until the counter is greater than zero, and then atomically decrements the counter. A semaphore is defined with `glSemaphoreCreate(name, initialCount)`, V'd by `glSemaphoreV(name)` and P'd by `glSemaphoreP(name)`. Figure 4 provides an example of the use of these primitives.

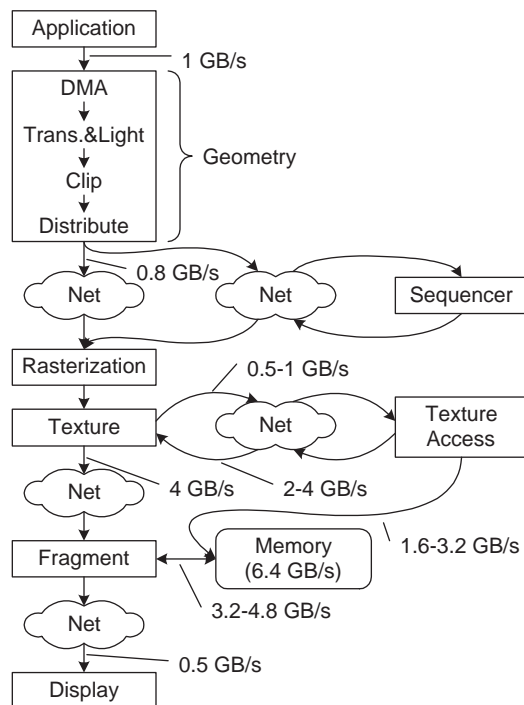


Figure 5: The Pomegranate pipeline. The bandwidth requirements of the communication channels are labeled.

3 Pomegranate Architecture

The Pomegranate architecture is composed of graphics pipelines and a high-speed network which connects them. The pipeline, shown in figure 5, is composed of five stages: geometry, rasterization, texture, fragment and display. The geometry stage receives commands from an application; transforms, lights and clips the primitives; and sends screen-space primitives to the rasterizer. The rasterizer performs rasterization setup on these primitives, and scan converts them into untextured fragments. The texturer applies texture to the resultant fragments. The fragment processor receives textured fragments from the texturer and merges them with the framebuffer. The display processor reads pixels from the fragment processor and sends them to a display. The network allows each pipeline of the architecture to communicate with all the other pipelines at every stage. For example, each geometry processor can distribute its transformed primitives over *all* the rasterizers.

Pomegranate achieves its scalability through a combination of a parallel host interface and multiple types of communication between the functional units.

- Each geometry unit has a host interface that may receive graphics commands simultaneously and independently. Ordering constraints between different graphics contexts may be specified by parallel API commands. This provides scalability of *input rate*. Because the geometry unit is limited by the interface speed, there is no purpose in distributing commands from a single interface across multiple geometry units. The application must therefore provide a balanced number of triangles to each interface. This provides scalability of *triangle rate*.
- A virtual network port allows each geometry unit to transmit screen-space primitives to any rasterizer. There is no constraint on this mapping, thus allowing the geometry units to load balance triangle work among rasterizers. This provides scalability of *triangle rate*.

- The sequencer, shared among all pipelines, determines the interleaving of the execution of the primitives emitted by each geometry unit. It allows multiple contexts to simultaneously submit commands to the hardware and to have their order of execution described by the parallel API. This provides scalability of *input rate*.
- Each rasterizer scan converts screen-space triangles into untextured fragments, and then passes them to the texturer where they are textured. The geometry units may load balance the amount of pixel work sent to each rasterizer in addition to the number of triangles. The geometry units may also subdivide large triangles so that their work is distributed over all the rasterizers. This provides scalability of *pixel rate*.
- Textures are distributed in a shared fashion among the pipeline memories, and each texture processor has a network port for reading and writing of remote textures. This provides scalability of *texture memory*.
- Each texture processor has a network port that enables it to route its resultant fragments to the appropriate fragment processor according to screen-space location. This sorting stage performs the object-space to image-space sort, and allows the unconstrained distribution of triangles between the geometry and rasterization stages that balances object-space parallelism. Fine interleaving of the fragment processors load balances screen-space parallelism and provides scalability in *pixel rate*.
- Each display unit has a network port that allows it to read pixels from all of the fragment processors and output them to its display. This provides scalability of *display bandwidth*.

The Pomegranate architecture faces the same implementation challenges as other parallel graphics hardware: load balancing and ordering. Load balancing issues arise every time that work is distributed. The four main distributions of work are: primitives to rasterizers by the geometry processors; remote texture memory accesses by the texturers; fragments to fragment processors by the texturers; and pixel requests to the fragment processors by the display engine. Additionally a balanced number of primitives must be provided to each geometry processor, but that is the responsibility of the application programmer.

Two distinct ordering issues arise in Pomegranate. First, the primitives of a single graphics context will be distributed twice, first over the rasterizers, and then over the fragment processors. This double distribution results in the work for a single context arriving out of order at the fragment processors, where it must be re-ordered. Second, each serial graphics context will execute its own commands in order, but it must in turn be interleaved with the other graphics contexts to provide parallel execution in accordance with any parallel API commands. In this section, we discuss in detail the different stages of the pipeline and their mechanisms for load balancing, and defer the discussion of maintaining a correct serial and parallel order and the associated sequencer unit until later.

3.1 Network

Central to the Pomegranate architecture is a scalable network that supports the balanced distribution of work necessary for load balancing and the synchronization communication necessary for ordering. We chose to implement the network as a multi-stage butterfly, depicted in figure 6. A discussion of other candidate networks is beyond this paper, and readers are encouraged to see the text by Duato, Yalmanchili and Ni [4] for a deeper discussion of high-performance scalable interconnects.

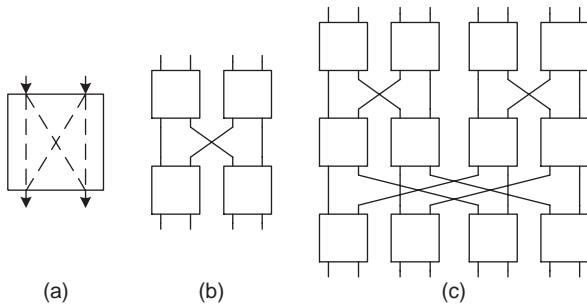


Figure 6: The butterfly network is composed of a single building block (a) which may be cascaded into a multi-stage network to support an arbitrary number of inputs (b & c), with a number of stages that grows logarithmically with the number of inputs.

Networks, and butterflies in particular, are notorious for suffering severe performance penalties under imbalanced loads and increasing latency with increasing utilization. Pomegranate’s network usage is engineered to be both uniform and latency tolerant to avoid these problems. For example, in an n -pipeline system, a geometry unit will send $1/n$ th of its triangles to each rasterizer. This distribution pattern occurs similarly during texture, fragment and display communication, and is balanced over very fine time scales. Furthermore, the algorithms used in Pomegranate are designed to be able to tolerate latency through the use of buffering.

At the heart of a butterfly network is a $k \times k$ switch ($k = 2$ for figure 6). Every cycle, this switch is able to read a single quantum of data (a *flit*) from each of its input channels and write a flit to each of its output channels. Internally, the switch must arbitrate its outputs according to the requests of incoming packets, which are composed of multiple flits. Channels are virtualized to provide multiple virtual channels per physical channel to increase the likelihood that an output channel will have a packet that needs it on every cycle [3]. Virtual channels are critical to large butterfly networks as they increase bandwidth efficiency from the range of 25% to over 75%.

In order to scale a butterfly network beyond the k inputs and outputs of a single switch, an additional stage of switches is introduced. The first stage routes based on the most significant digits of the destination address, and the second stage routes based on the least significant digits. An n -interface network may be constructed using $\log_k n$ stages of n/k switches. As the number of interfaces increases the aggregate bandwidth available increases linearly, while the cost increases as $n \log_k n$.

The multiple networks of figure 5 are actually virtualized ports within a single unified network. Two messages from the same source to the same destination, e.g. geometry processor 0 to rasterizer 3, are guaranteed to arrive in order, but no other ordering guarantees are made by the network. A unified network allows the network to be used efficiently for all types of traffic, rather than having some networks left idle while other networks are overloaded with traffic. In order to support the expected workload and network inefficiencies, each channel runs at 10 GB/sec. Each channel is 32 bits wide and operates at 2.5 GHz. Each 160-bit flit in our system is transferred over 5 32-bit clocks, and thus the switching logic runs at 500 MHz. We use 4×4 switches and 16 virtual channels per physical channel, each capable of buffering 16 flits. Ignoring contention, each hop through a switch imposes 8 flits of latency. Packets are constrained to be an integral number of flits, with a 24 bit header in the first flit, which imposes a small overhead.

3.2 Geometry

The geometry unit consists of a DMA engine, a transform and lighting engine, a clip processor and a distribution processor. Each geometry unit supports a single hardware context, although the context may be virtualized.

- The DMA engine is responsible for transferring blocks of commands across the host interface and transferring them to the transform and lighting engine. In our model the host interface bandwidth is 1 GB/sec. This is representative of AGP 4x, a current graphics interface.
- The transform and lighting (T&L) engine is a vertex parallel vector processor. It transforms, culls and lights the primitives. Clipping is not performed in the T&L engine because it introduces a potentially large number of new vertices and corresponding primitives which must be correctly ordered in the primitive stream. Deferring this generally infrequent operation to a dedicated clip processor greatly simplifies the T&L engine. The T&L engine has a maximum performance of 20 million transformed and lit vertices per second.
- The clip processor performs geometric clipping for any primitives that intersect a clipping plane. Computation of the clip state for each vertex is performed by the T&L engine, so the clip processor’s fast path has no computation. After geometric clipping, the clip processor subdivides large primitives into multiple smaller primitives by specifying the primitives multiple times with different rasterization bounding boxes. This subdivision ensures that the work of rasterizing a large triangle can be distributed over all rasterizers. Large primitives are detected by the signed area computation of back-face culling and subdivided according to a primitive-aligned 64×64 stamp.
- The distribution processor distributes the clipped and subdivided primitives to the rasterizers. This is Pomegranate’s first “sort”. Because the rasterizers are primitive parallel (object-space parallel) rather than fragment parallel (image-space parallel), the distribution processor has the freedom to distribute primitives as it sees fit.

The distribution processors transmit individual vertexes with meshing information over the network to the rasterizers. A vertex with 3D texture coordinates is 228 bits plus 60 bits for a description of the primitive it is associated with and its rasterization bounding box, resulting in 320 bit (2 flit) vertex packets. At 20 Mvert/sec, each distribution processor generates 0.8 GB/sec of network traffic. The distribution processor generates additional network traffic in two cases. First, large primitives are subdivided to ensure that they present a balanced load to all the rasterizers. In such a case the additional network traffic is unimportant, as the system will be rasterization limited. Second, commands that modify rasterizer state (e.g. the texture environment) must be broadcast to all the rasterizers.

The distribution processor governs its distribution of work under conflicting goals. It would like to give the maximum number of sequential triangles to a single rasterizer to minimize the transmission of mesh vertexes multiple times and to maximize the texture cache efficiency of the rasterizer’s associated texture processor. At the same time it must minimize the number of triangles and fragments given to each rasterizer to load balance the network and allow the reordering algorithm, which relies on buffering proportional to the granularity of distribution decisions, to be practical. The distribution processor balances these goals by maintaining a count of the number of primitives and an estimate of the number of fragments sent to the current rasterizer. When either of these counts

exceeds a limit, the distribution processor starts sending primitives to a new rasterizer. While the choice of the next rasterizer to use could be based on feedback from the rasterizers, a simple round-robin mechanism with a triangle limit of 16 and a fragment limit of 4096 has proven effective in practice. When triangles are small, and thus each rasterizer gets very few fragments, performance is geometry limited and the resulting inefficiencies at the texture cache are unimportant. Similarly, when triangles are large, and each rasterizer gets few triangles, or perhaps even only a piece of a very large triangle, the performance is rasterization limited and the inefficiency of transmitting each vertex multiple times is inconsequential.

3.3 Rasterizer

The rasterizer scan converts triangles, as well as points and lines, into a stream of fragments with color, depth and texture coordinates. The rasterizer emits 2×2 fragment “quads” at 100 MHz and requires 3 cycles for triangle setup, for a peak fill rate of 400 Mpixel/sec. Partially covered quads can reduce the rasterizer’s efficiency to 100 Mpixel/sec in the worst case. We achieve 1.34 to 3.95 fragments per quad for the scenes in this paper. Each rasterizer receives primitives from all the geometry processors and receives execution order instructions from the sequencer (see section 4). Each of the geometry units maintains its own context, and thus each rasterizer maintains n contexts, one per geometry processor. The fragment quads emitted by the rasterizer are in turn textured by the texture processor.

3.4 Texture

The texture stage consists of two units, the texture processor which textures the stream of quads generated by the rasterizer, and the texture access unit which handles texture reads and writes. The input to the rasterization stage has already been load balanced by the distribution processors in the geometry stage, so each texture processor will receive a balanced number of fragments to texture.

In order to provide a scalable texture memory, textures are distributed over all the pipeline memories in the system. Igehy et al. have demonstrated a prefetching texture cache architecture that can tolerate the high and variable amount of latency that a system with remote texture accesses, such as Pomegranate, is likely to incur [6]. Igehy et al. subsequently showed that this cache architecture could be used very effectively under many parallel rasterization schemes, including an object-space parallel rasterizer similar to Pomegranate [5]. Based on these results, we distribute our textures according to 4×4 texel blocks. Texture cache misses to a non-local memory are routed over the network to the texture access unit of the appropriate pipeline. The texture access unit reads the requested data and returns it to the texture processor, again over the network. A texture cache miss requires that a 160-bit texture request be sent over the network, which will be followed by a 640-bit reply, for a total of 800 bits of network traffic per 16 texels, or 6.25 bytes per texel. If we assume 1–2 texels of memory bandwidth per fragment, our rasterizer requires 4–8 bytes of texture memory bandwidth and 6.25–12.5 bytes of network bandwidth per fragment. At 400 Mpixel/sec, this becomes 1.6–3.2 GB/sec of memory bandwidth and 2.5–5 GB/sec of network bandwidth.

After texturing the fragments, the texture processor routes the fragment quads to the appropriate fragment processors. The fragment processors finely interleave responsibility for pixel quads on the screen. Thus, while the texture engine has no choice in where it routes fragment quads, the load it presents to the network and all of the fragment processors will be very well balanced. A quad packet contains 4 fragment colors, 4 corresponding sample masks, the depth of the lower-left fragment, the depth slopes in x and y and the location of the quad on the screen. This representation encodes a quad in 241 bits, or 320 bits (2 flits) on the network. Due to network packet size constraints, this is only twice the size of an individually encoded fragment, which is transmitted as 1 flit. At 100

Mquad/sec, the texture processor sends 4 GB/sec of traffic to the fragment processors. Just as the distribution processor broadcasts rasterization state changes to the rasterizers, the texture processor must also broadcast fragment processor state changes.

3.5 Fragment

The fragment stage of the pipeline consists of the fragment processor itself and its attached memory system. The fragment processor receives fragment quads from the texture processor and performs all the per-fragment operations of the OpenGL pipeline, such as depth-buffering and blending. The memory system attached to each fragment processor is used to store the subset of the framebuffer and the texture data owned by this pipeline.

The use of fragment quads, in addition to reducing network bandwidth, allows efficient access to the memory system by grouping reads and writes into 16-byte transactions. Each pixel quad is organized by pixel component rather than by pixel, so, for example, all of the depth components are contiguous and may be accessed in a single transaction. This improves Pomegranate’s efficiency in the peak performance case of fully covered fragment quads, and when fragment quads are only partially covered Pomegranate is already running beneath peak pixel rates, so the loss of memory efficiency is not as important.

The memory system provides 6.4 GB/sec of memory bandwidth. At 400 Mpixel/sec and 8 to 12 bytes per pixel (a depth read, depth write, and color write), fragment processing utilizes 3.2 to 4.8 GB/sec. When combined with texture accesses of 1.6 to 3.2 GB/sec and display accesses of 0.5 GB/sec, the memory system bandwidth is overcommitted. Memory access is given preferentially to the display processor, since it must always be serviced, then to the fragment processor, because it must make forward progress for the texture processor to continue making forward progress, and finally the texture access unit. The majority of our results are not memory access limited.

Pomegranate statically interleaves the framebuffer at a fragment quad granularity across all of the fragment processors. This image-space parallel approach has the advantage of providing a near perfect load balance for most inputs. As with the rasterizers, the fragment processors maintain the state of n hardware contexts. While the rasterizers will see work for a single context from any particular geometry unit, the fragment processor will see work for a single context from all the texture processors because the geometry stage’s distribution processor distributes work for a single context over all the rasterizers.

3.6 Display

The display processor is responsible for retrieving pixels from the distributed framebuffer memory and outputting them to a display. Each pipeline’s display processor is capable of driving a single display. The display processor sends pipelined requests for pixel data to all of the fragment processors, which in turn send back strips of non-adjacent pixels. The display processor reassembles these into horizontal strips for display. Unlike the use of the network everywhere else in Pomegranate, the display system is very sensitive to latency — if pixels arrive late, gaps will appear in the displayed image. We address this issue with a combination of buffering, which enables the display processor to read ahead several scanlines, and a priority channel in the network. Dally has shown that a bounded percentage of the traffic on a network can be made high priority and delivered with guaranteed latency [3]. At a display resolution of 1920×1280 and a 72 Hz refresh rate, the display bandwidth is 0.5 GB/sec, 5% of Pomegranate’s per-pipeline bandwidth.

4 Ordering

Ordered execution of the application command stream must be maintained everywhere its effects are visible to the user. The work distribution algorithms described in the previous section explain

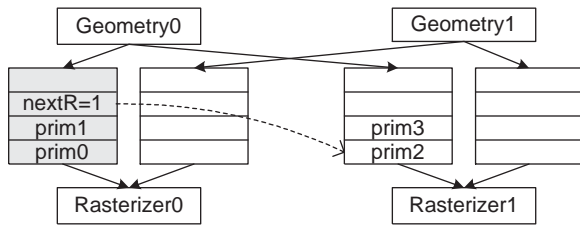


Figure 7: Geometry processor 0 distributes its first 2 primitives to rasterizer 0, and its second two primitives to rasterizer 1. It expresses the ordering constraint between them with a NextR command.

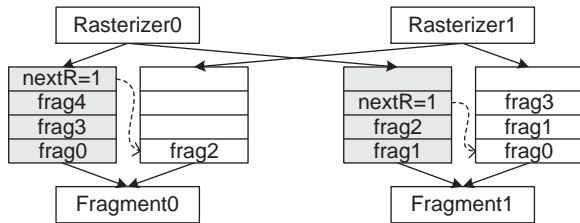


Figure 8: Rasterizer 0 and rasterizer 1 simultaneously process the primitives distributed to them by geometry processor 0. After rasterizing primitive 1, rasterizer 0 broadcast a NextR to all the fragment processors, announcing that they should now process fragments from rasterizer 1. The texture processors have been omitted for clarity.

how the Pomegranate architecture scales performance, but the constraint of ordering was ignored. By far the most prevalent place this constraint is exposed is the OpenGL API itself, which is stateful. For example, a `glBlendFunc` command modifies the blending state for all subsequent primitives and no previous primitives. Second, many commands (i.e. points, lines, triangles) modify the contents of the framebuffer, and these modifications must occur in order at each pixel. Finally, changes to the texture memory must be ordered.

Pomegranate faces two distinct ordering issues. First, the commands for a single context are distributed over all the rasterizers, which in turn distribute their fragments over all the fragment processors. This double sort means that the original order of the command stream must be communicated to the fragment processors to allow them to merge the fragments in the correct order. Second, the operations of different contexts must be interleaved in a manner that observes constraints specified by the parallel API.

4.1 Serial Ordering

The key observation to implementing ordering within a single context is that every place work is distributed, the ordering of that work must be distributed as well. The first distribution of work is performed by the distribution processor, which distributes blocks of primitives over the rasterizers. Every time it stops sending primitives to the current rasterizer and starts sending primitives to a new rasterizer it emits a NextR command to the current rasterizer, announcing where it will send subsequent primitives. Figure 7 shows the operation of this mechanism. These NextR commands provide a linked list of the primitive order across the rasterizers. The rasterizers in turn broadcast the NextR commands to all the fragment processors. Each rasterizer has dedicated command buffering for each geometry unit, so that the commands from different geometry units may be distinguished.

The fragment processors each have dedicated buffering for receiving commands from each of the rasterizers, as illustrated in figure 8. Each fragment processor processes commands from a single

rasterizer at a time. When a fragment processor receives a NextR command, it ceases listening to the current rasterizer and starts listening to the specified next rasterizer. This is analogous to following the linked list of NextR commands emitted by the distribution processor. While a fragment processor will only ever process commands from a single rasterizer at any point in time, all of the rasterizers can continue to make forward progress and transmit fragments to the fragment processors where they will be buffered.

The Pomegranate architecture is designed with the expectation that the same parts which construct the base units are repeated to create larger, more powerful systems. As part of this assumption, the amount of buffering at the input of each fragment processor is fixed. However, this buffering is always divided evenly among all the rasterizers, so as the number of pipelines increases the buffering available per rasterizer at each fragment processor shrinks. However, the increase in the number of pipelines matches this decrease, and the total amount of buffering per rasterizer across all fragment processors remains constant.

The amount of traffic generated by NextR commands from a geometry unit to a rasterizer is limited. When the scene is triangle limited, one single-flit NextR packet is sent to a rasterizer for every 16 two-flit vertex packets sent to a rasterizer. This represents an overhead of approximately 3%, which remains constant as the system scales. The NextR messages from the rasterizers to the fragment processors, on the other hand, represent a potential broadcast in the system because each rasterizer must broadcast each NextR it receives to all the fragment processors. Fortunately, this broadcast may be avoided by employing a lazy algorithm. Because NextR commands take only a few bits to encode, we can include space for a potential NextR command in every fragment quad without increasing its size in network flits. Because the fragment processors have very finely interleaved responsibility for quads on the screen, chances are that a fragment quad will be sent to the fragment processor shortly after the NextR command is observed by the rasterizer. A timeout ensures that a NextR command that is waiting to piggyback on a fragment quad is not excessively delayed, prompting the rasterizer to send as many outstanding NextR commands as possible in a single network packet.

In general, the fragment processors operate independently, each processing fragments at its own rate. The exception is when a command observes or modifies shared state beyond that on a single fragment processor, the fragment processors must be synchronized. Pomegranate uses an internal fragment barrier command, `BarrierF`, to support this synchronization. For example, `glFinish` has an implementation similar to this pseudocode:

```
glFinish( ) {
    BarrierF
    hardware writeback to device driver
}
```

The `BarrierF` ensures that all previous operations by this context are complete before the writeback signaling completion of the `glFinish` occurs.

A similar issue arises at the rasterizers. If a command modifies the current texture state, which is shared among the multiple rasterizers, it must be executed in the correct serial order with respect to the other commands from that context. Pomegranate enforces this constraint with an internal `BarrierR` command which forces all of the rasterizers to synchronize. A texture modification command can be bracketed between `BarrierR` commands and thus be made atomic within the hardware. For example, `glTexImage2D` has an implementation similar to this pseudocode:

```
glTexImage2D( ) {
    BarrierR
    texture download
}
```

```

    BarrierR
}

```

The initial `BarrierR` ensures that all previous commands for this context are complete on all rasterizers before the texture download starts so that the new texture does not appear on any previous primitives. The final `BarrierR` ensures no subsequent commands for this context are executed on any rasterizer before the texture download completes so that the old texture does not appear on any subsequent primitives.

4.2 Parallel Ordering

The internal hardware commands `NextR`, `BarrierR` and `BarrierF` suffice to support serial ordering semantics. The extension of the hardware interface to a parallel API requires additional support. The parallel API requires that some or all of the graphics resources must be virtualized, and more importantly, subject to preemption and context switching. Imagine an application of $n + 1$ graphics contexts running on a system that supports only n simultaneous contexts. If a graphics barrier is executed by these $n + 1$ contexts, at least one of the n running contexts must be swapped out to allow the $n + 1$ th context to run. Furthermore, the parallel API introduces the possibility of deadlock. Imagine an incorrectly written graphics application that executes a `glSemaphoreP` that never receives a corresponding `glSemaphoreV`. At the very least, the system should be able to preempt the deadlocked graphics context and reclaim those resources. Resolving the preemption problem was one of the most difficult challenges of the Pomegranate architecture.

One solution to the preemption problem is the ability to read back all of the state of a hardware context and then restart the context at a later time. Although this may seem straightforward, it is a daunting task. Because a context may block at any time, the preempted state of the hardware is complicated by partially processed commands and large partially-filled FIFOs. As a point of comparison, microprocessor preemption, which has a much more coherent architecture compared to a graphics system, is generally viewed by computer architects as a great complication in high-performance microprocessors.

A second approach to the preemption problem is to resolve the API commands in software, using the preemption resources of the microprocessor. With this approach, even though ordering constraints may be specified to the hardware, every piece of work specified has been guaranteed by the software to eventually execute. Figure 9 illustrates this approach. Each graphics context has an associated submit thread that is responsible for resolving the parallel API primitives. The application thread communicates with the submit thread via a FIFO, passing pointers to blocks of OpenGL commands and directly passing synchronization primitives. If the submit thread sees a pointer to a block of OpenGL commands, it passes this directly to the hardware. If the submit thread sees a parallel API command, it actually executes the command, possibly blocking until the synchronization is resolved. This allows the application thread to continue submitting OpenGL commands to the FIFO beyond a blocked parallel API command. In addition to executing the parallel API command, the submit thread passes the hardware a sequencing command that maintains the order resolved by the execution of the parallel API command. The important part of this hardware sequencing command is that even though an ordering is specified, the commands are guaranteed to be able to drain: the hardware sequencing command for a `glSemaphoreP` will not be submitted until the hardware sequencing command for the corresponding `glSemaphoreV` is submitted. Thus, a blocked context is blocked entirely in software, and software context switching and resource reclamation may occur.

In order to keep hardware from constraining the total number of barriers and semaphores available to a programmer, the inter-

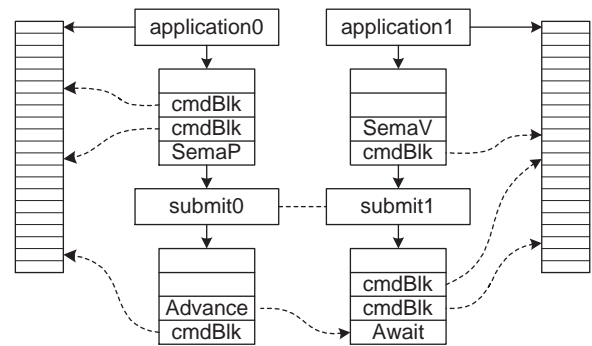


Figure 9: Each graphics context has an associated submit thread which is responsible for resolving the parallel API primitives. In this figure submit thread 0 is blocked waiting to resolve a semaphore P that will be released by context 1. Both application threads are continuing to submit work, and the hardware is continuing to consume work.

nal hardware sequencing mechanism is based on a single sequence number per hardware context. Upon executing a `glSemaphoreV` operation, the submit thread increments the hardware context's sequence number by one to indicate a new ordering boundary, annotates the semaphore with a (ctx, seq) pair and issues an `AdvanceContext` (ctx, seq) command to the hardware. Upon completing the `glSemaphoreP` operation, the signaled submit thread removes the corresponding (ctx, seq) annotation from the semaphore and issues an `AwaitContext` (ctx, seq) command to the hardware. A similar mechanism is used to implement barriers.¹ The sequence numbers are associated with a particular hardware context, not with a virtual graphics context, and when a context switch occurs, it is not reset. This allows us to express dependencies for contexts that have been switched out of the hardware, and thus execute an $n + 1$ context barrier on n context hardware.

Given the `AdvanceContext`/`AwaitContext` commands for expressing ordering constraints among contexts, Pomegranate now needs a way of acting on these constraints. The sequencer unit provides a central point for resolving these ordering constraints and scheduling the hardware. The distribution processors at the end of the geometry stage, each of which is dedicated to a single hardware context, inform the sequencer when they have work available to be run and what ordering constraints apply to that work. The sequencer then chooses a particular order in which to process work from the various contexts and broadcasts this sequence to all of the rasterizers, which, along with all the subsequent stages of the pipeline, are shared among all the contexts.

Whenever a distribution processor starts emitting primitives, it sends a `Start` command to the sequencer to indicate that it has work available to be scheduled. In addition, the distribution processor transmits all `AdvanceContext` and `AwaitContext` commands for its context to the sequencer, which in turn enforces the ordering relationships expressed by these commands when making its scheduling decisions. The counterpart of the `Start` command is the `Yield` command which the distribution processors broadcast to all the rasterizers at the end of a block of work. When a rasterizer encounters a `Yield` it reads the next execute command from the sequencer and starts executing that context. The `Yield` com-

¹The first $n - 1$ submit threads to arrive at the barrier execute an `AdvanceContext` to create a sequence point and block. The last context to arrive at the barrier executes an `AwaitContext` on the previous $n - 1$ contexts, an `AdvanceContext` to create its own sequence point and then unblocks the waiting contexts. The $n - 1$ waiting contexts then each execute an `AwaitContext` on the n th context's just created sequence point, for a total of n `AdvanceContexts` and n `AwaitContexts`.

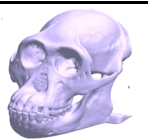


			
scene	March	Nurbs	Tex3D
input	400^3 volume	1632 patches, 8 passes	256^3 volume
output	$2.5K \times 2K$	$2.5K \times 2K$	$1K \times 1K$
triangles	1.53M	6.68M	512
fragments	10.4M	5.81M	92.5M

Table 1: Benchmark scenes.

mand provides context switching points to support ordering and to allow the pipelines to be shared. First, if a context waits on another context, it must then yield to allow the rasterizers to work on other contexts, which will eventually allow this context to run again. Second, a context must occasionally yield voluntarily, to allow the hardware to be shared among all the contexts so that a single context does not unfairly monopolize the machine. The frequency of these yields is determined by the relationship of the triangle rate of the geometry units to the command buffering provided at each rasterizer. In our implementation, a context yields once it has sent one `NextR` command to each rasterizer. Because the sequencer is decoupled from the rasterizers, it will make scheduling decisions as far in advance as it can, limited only by the available information from the distribution processors and the available buffering for execution commands at the rasterizers.

5 Results

We have implemented an OpenGL software device driver and hardware simulator to verify our architecture. Our system supports all of the major functionality of OpenGL.

The Pomegranate hardware is modeled under an event-driven simulator. The simulator is composed of multiple independent threads of execution which communicate with each other via events. Threads advance events and await on events to coordinate their execution. The simulator provides a shared global knowledge of time, so threads may both wait for other threads to complete a task, as well as simply wait for time to pass, to model clock cycles, etc. The simulator is non-preemptive and a particular thread of execution only ceases execution when it explicitly waits for time to pass or waits on a semaphore.

Our simulator masquerades as the system OpenGL dynamic library on Microsoft Windows NT and SGI IRIX operating systems. Application parallelism is supported through additional functions exported by our OpenGL library that allow the creation of user threads within our simulator. This simulation methodology allows us to deterministically simulate Pomegranate, which aids both debugging and analysis of performance. In particular, performance problems can be iteratively analyzed by enabling more and more instrumentation in different areas of the hardware, with the confidence that subsequent runs will behave identically.

We analyzed Pomegranate’s performance with three applications, shown in table 1. The first, *March*, is a parallel implementation of marching cubes [8]. The second, *Nurbs*, is a parallel patch evaluator and renderer. The final, *Tex3D*, is a 3D texture volume renderer.

- *March* extracts and draws an isosurface from a volume data set. The volume is subdivided into 12^3 voxel subcubes that are processed in parallel by multiple application threads. Each subcube is drawn in back to front order, allowing the use of transparency to reveal the internal structure of the volume. The parallel API is used to order the subcubes generated by

	entries	bytes/entry	bytes
Primitive	4096	120	480K
Texture	256	72	18K
Fragment	16384	32	512K

Table 2: Total FIFO sizes for each of functional unit. The FIFO size is listed as the total number of commands it can contain.

each thread in back to front order. Note that while *March* requires a back to front ordering, there are no constraints between cubes which do not occlude each other, so substantial inter-context parallelism remains for the hardware.

- *Nurbs* uses multiple application threads to subdivide a set of patches and submit them to the hardware. We have artificially chosen to make *Nurbs* a totally ordered application in order to stress the parallel API. Such a total order could be used to support transparency. Each patch is preceded by a semaphore P and followed by a semaphore V to totally order it within the work submitted by all the threads. Multiple passes over the data simulate a multipass rendering algorithm.
- *Tex3D* is a 3D texture volume renderer. *Tex3D* draws a set of back to front slices through the volume along the viewing axis. *Tex3D* represents a serial application with very high fill rate demands and low geometry demands, and it is an example of a serial application that can successfully drive the hardware at a high degree of parallelism.

We measure Pomegranate’s performance on these scenes in four ways. First we examine Pomegranate’s raw scalability, the speedup we achieved as a function of the number of pipelines. Next we examine the load imbalance across the functional units, which will determine the best achievable speedup for our parallel system. Then we quantify the network bandwidth demanded by the different stages of the pipeline and analyze the lost performance due to network imbalance. Finally we compare Pomegranate’s performance to simulations of sort-first, sort-middle and sort-last architectures.

All of these simulations are based on the parameters outlined in our description of the architecture, and the FIFO sizes listed in table 2. The primitive FIFO is the FIFO at the input to the rasterizer, and determines how many primitives a geometry unit can buffer before stalling. The texture FIFO is the FIFO that receives texture memory requests and replies and determines how many outstanding texture memory requests the texture system can have. The final major FIFO is the fragment FIFO, which is where the fragment processors receive their commands from the texture processors. The n pipeline architecture uses the same FIFOs as the 1 pipeline architecture, but divides them into n pieces. The FIFO sizes have been empirically determined.

5.1 Scalability

Our first measure of parallel performance is speedup, presented for our scenes in figure 10. *Nurbs* exhibits excellent scalability, despite presenting a totally ordered set of commands to the hardware. At 64 processors the hardware is operating at 99% efficiency, with a triangle rate of 1.10 Gtri/sec and a fill rate of 0.96 Gpixel/sec. The only application tuning necessary to achieve this level of performance is picking an appropriate granularity of synchronization. Because *Nurbs* submits all of its primitives in a total order, the sequencer has no available parallel work to schedule, and is always completely constrained by the API. This results in only 1 geometry unit being schedulable at any point in time, and the other geometry units will only make forward progress as long as there is adequate buffering at the rasterizers and fragment processors to receive their commands. This requirement is somewhat counterintuitive, as the

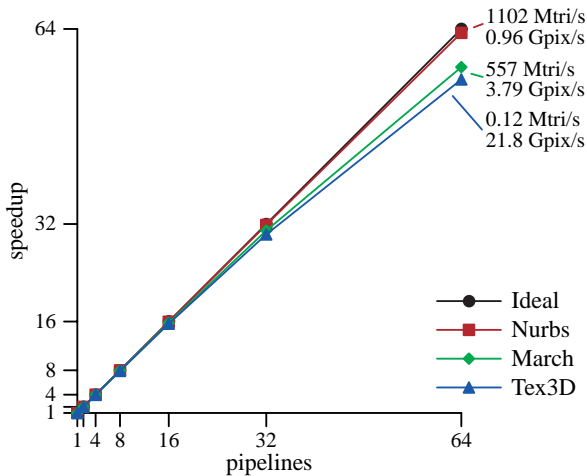


Figure 10: Pomegranate speedup vs. number of pipelines.

usual parallel programming rule is to use the largest possible granularity of work.

March runs at a peak of 557 Mtri/sec and 3.79 Gpixel/sec in a 64-pipeline architecture, a $58\times$ speedup over a single pipeline architecture. While this scalability is excellent, it is substantially less than that of **Nurbs**. If we examine the granularity of synchronization, the problem becomes apparent. **Nurbs** executes a semaphore pair for every patch of the model, which corresponds to every 512 triangles. **March**, on the other hand, executes 3 semaphore pairs for every 12^3 voxel subcube of the volume, and the average subcube only contains 38.8 triangles. Thus, the number of synchronization primitives executed per triangle is more than an order of magnitude greater than that of **Nurbs**. Furthermore, there is high variance in the number of triangles submitted between semaphores. These effects cause **March** to encounter scalability limitations much sooner than **Nurbs** despite its much weaker ordering constraints.

Tex3D runs at 21.8 Gpixel/sec on a 64-pipeline Pomegranate, with a tiny 0.12 Mtri/sec triangle rate, a $56\times$ speedup over a single pipeline architecture. **Tex3D** scales very well, considering that it is a serial application. If **Tex3D**'s input primitives were skewed towards smaller triangles it would rapidly become limited by the geometry rate of a single interface and execution time would cease improving as we add pipelines.

5.2 Load Balance

In order to achieve a high parallel efficiency, the work performed by the hardware must be balanced across the functional units and communication must be balanced across the network. Table 3 presents the load imbalance for **Nurbs** on our architecture with 4, 16, and 64 pipelines. The load balance is within a few percent for all the functional units. This indicates that Pomegranate's methodology for distributing work is providing us with an excellent load balance. By the time **Nurbs** reaches 64 pipelines the network is significantly out of balance. This is an artifact of **Nurbs**'s relatively low network usage, as it is geometry limited, and the asymmetry of the network traffic generated by the sequence processor, as discussed in section 5.3. The results for **March** are not shown, but they are qualitatively similar.

Table 4 shows the load imbalance for **Tex3D**. Despite all of the application commands arriving through a single interface, the subsequent rasterization and fragment stages still receive an extremely balanced load. The texture load imbalance is the ratio of the most texture requests handled by a pipeline to the average. Numbers close to 1 indicate that the shared texture memory is working effectively, because all of the texture requests are well distributed over the pipelines. **Tex3D**'s network imbalance is becoming significant

pipelines	4	16	64
Geometry	1.00/1.00	1.00/1.00	1.00/1.00
Rasterizer	1.00/1.00	1.00/1.00	0.98/1.02
Fragment	1.00/1.00	0.99/1.01	0.99/1.01
Network	0.98/1.04	0.97/1.27	0.95/2.63

Table 3: Load balance for **Nurbs**. Each entry in the table presents the minimum/maximum work done by any functional unit as a fraction of the average work per functional unit. Geometry work is measured in triangles; rasterization and composition work is measured in fragment quads. The network imbalance is measured in bytes of traffic per pipeline.

pipelines	4	16	64
Geometry	0.00/4.00	0.00/16.0	0.00/64.0
Rasterization	1.03/1.00	1.00/1.00	1.00/1.00
Texture	1.00/1.00	1.00/1.00	0.99/1.00
Fragment	1.00/1.00	1.00/1.00	1.00/1.01
Network	1.00/1.01	1.00/1.04	0.99/1.15

Table 4: Load balance for **Tex3D**. Each entry in the table presents the minimum/maximum work done by any functional unit, as a fraction of the average work per functional unit. Geometry work is measured in triangles; rasterization and composition work is measured in fragment quads.

by the time we reach 64 pipelines. This large asymmetry is the result of all of the primitives entering through a single interface and being distributed from a single geometry unit. As Pomegranate is scaled, the total rasterization speed increases, but the entire geometry traffic is borne by a single pipeline.

5.3 Network Utilization

There are five main types of network traffic in Pomegranate: geometry, sequencer, texture, fragment and display. Geometry traffic is comprised of vertexes transmitted from the geometry processor to the rasterizers and the **NextR** ordering commands, as well as any state commands, textures, etc. for the subsequent stages. Sequencer traffic is the communication between the distribution processors and the sequencer as well as the sequencer and the rasterizers, and encapsulates all the traffic which allows the hardware to be shared among multiple contexts and the parallel API commands. Texture traffic is made up of the texture request and texture reply traffic generated by each texture processor. Fragment traffic is composed of the quads emitted by the texture processors and sent to the fragment processors. Display traffic is the pixel read requests and replies between the display processors and the fragment processors. The network bandwidth for each traffic type across our scenes on a 64-pipeline Pomegranate is presented in table 5. The sequencer numbers are extremely skewed because there is a single sequencer in the system, so all sequencing information from the distribution processors flows into a single point, and all sequencing decisions for the rasterizers flow back out of that point, which introduces a broadcast into the system. A future version of Pomegranate will use a low bandwidth broadcast ring connecting all the pipelines specifically for the distribution of the sequencing information.

5.4 Comparison

We compare Pomegranate's performance to 4 other parallel graphics architectures:

Sort-First introduces a communication stage between the DMA units and transform & lighting in the geometry processor. The screen is statically partitioned in 32×32 tiles among the pipelines. The screen-space bounding boxes of blocks of 16 vertexes are used to route primitives to pipelines.

	March	Nurbs	Tex3D
Geometry	0.84/0.85	0.54/0.58	0.01/0.93
Sequence	0.02/1.06	0.05/2.95	0.00/0.14
Texture	0/0	0/0	3.00/3.01
Fragment	1.82/1.84	1.19/1.20	3.31/3.32
Total	2.68/3.71	1.78/4.67	6.33/7.26

Table 5: The network traffic by type for each of our scenes on a 64-pipeline Pomegranate. Each row corresponds to a particular type of traffic and each pair of numbers is the average/maximum amount of traffic per pipeline of that type in gigabytes per second. These simulations do not include a display processor.

Sort-Middle Tiled is a sort-middle architecture with the screen statically partitioned in 32×32 tiles among the rasterizers. Individual primitives are only transmitted to the rasterizers whose tiles they overlap.

Sort-Middle Interleaved partitions the screen in 2×2 tiles to ensure rasterization and fragment load balancing. Each geometry processor broadcasts its primitives to all rasterizers.

Sort-Last Fragment partitions the screen in 2×2 tiles among the fragment processors. Each rasterizer is responsible for all the primitives transformed by its corresponding geometry processor.

All of these architectures are built on top of the Pomegranate simulator, and only differ in how the network is deployed to interconnect the various components. We provide each of these architectures, although not Pomegranate, with an ideal network — zero latency and infinite bandwidth — to illustrate fundamental differences in the work distribution. All of these architectures have been built to support the parallel API and a shared texture memory. The ordering mechanisms necessary to support the parallel API are borrowed from Pomegranate, although they are deployed in different places in the pipeline.

Our simulator requires substantial time to run — over 24 hours for some the 64 pipeline simulations. In order to provide these results across all these architectures we were forced to reduce the size of the benchmarks for the remaining simulations. Point simulations of the full data sets give us confidence that the results presented here are quantitatively very similar to the results for the full scenes used in the previous sections.

Figure 11a shows the performance of all of these architectures for the March data set. As March runs, all of the primitives are clustered along the isosurface, which results in high screen-space temporal locality. Sort-first, which uses coarse-grained screen-space parallelism for both geometry and rasterization, is most severely impacted because *temporal locality causes spatial load imbalances over short periods of time*, the length of which are determined by the amount of FIFOing available. Sort-middle tiled employs object-space parallelism for the geometry stage, and because this scene is not rasterization limited, exhibits substantially more scalability than sort-first, although its limitations are exposed at higher levels of parallelism. Sort-middle interleaved behaves much more poorly than sort-middle tiled because it broadcasts triangle work to every rasterizer, and each rasterizer can process a limited number of triangles per second. Sort-last and Pomegranate both scale very well because they rasterize each triangle only once (eliminating redundant work) and use object-space parallelism for rasterization (eliminating any issues with temporal locality). The main difference between Pomegranate and sort-last, the balancing of fragment work across rasterizers by the geometry processors, does not matter here because the triangles are relatively uniformly sized.

Nurbs, shown in figure 11b, exhibits much worse scalability for sort-first and sort-middle than March, and in fact even slows down at high degrees of parallelism. The granularity of work for Nurbs is a patch, which exhibits a great degree of temporal locality in screen-space, even greater than March, which explains the performance at low degrees of parallelism. However, unlike March, Nurbs is a totally ordered application, and when combined with architectures that use screen-space parallelism for geometry or rasterization, the result is hardware that performs almost no better than the serial case. As the number of pipelines increases, the system is capable of processing more work. However, the amount of FIFOing available from each pipeline to each tile decreases, reducing the window over which temporal load imbalance may be absorbed. The hump in performance at moderate numbers of pipelines is a result of these effects. As with March, sort-last and Pomegranate exhibit excellent scalability.

Unlike March and Nurbs, Tex3D, shown in figure 11c, is a completely rasterization limited application. The speedup for sort-first and sort-middle tiled here is limited purely by the rasterization load balance of the entire frame, illustrating that even scenes which appear very well balanced in screen-space may suffer large load imbalances due to tiling patterns at high degrees of parallelism. Sort-middle interleaved, which was previously limited by its reliance on broadcast communication, is now limited by texture cache performance, which is severely compromised by the use of extremely fine-grained rasterization. Each triangle is so large in this application that it serializes sort-last at the fragment processor stage: the fragment FIFOs provide elasticity for the rasterizers to continue ordered rasterization on subsequent triangles while the current triangle is merged with the framebuffer, but when a single large triangle fills up the entire FIFO this elasticity is lost and the rasterizers are serialized. If we greatly increase the buffering at the fragment processors, shown by the “sort-last big” curve, so that sort-last is no longer serialized by the large primitives, the fundamental problem with sort-last is exposed: imbalances in triangle size cause load imbalances across the rasterizers. In Tex3D at 64 pipelines, the worst rasterizer has almost twice the work of an average rasterizer. Many applications (e.g. architectural walkthroughs) have a few very large polygons and exhibit much more severe imbalance in rasterization work than the relatively innocuous Tex3D. Pomegranate addresses this fundamental problem by load balancing both the number of triangles and the number of fragments across the rasterizers, and exhibits excellent scalability on Tex3D.

6 Discussion

Pomegranate was designed to support an immediate-mode parallel graphics interface and uses high-speed point-to-point communication to load balance work across its pipelines. Our results have demonstrated the quantitative impact of these choices, and we will now revisit their qualitative benefits and costs.

6.1 OpenGL and the Parallel API

The decision to support OpenGL, a strict serial API, has proven somewhat complicated to implement, but has not resulted in a performance impact. In fact, Nurbs, which totally orders the submission of its work across all contexts, achieves almost perfectly linear speedup, despite its very strong ordering constraints. The expense of supporting ordering is FIFOs which allow the various pipeline stages to execute commands in application order.

While it may be necessary for the application programmer to choose an appropriate granularity of parallelism, particularly in a strongly ordered scene, it is not required that the application balance fragment work, only primitive work. This is a desirable feature, as in general application programmers have little knowledge of the amount of fragment work that will be generated by a primitive, but they are well aware of the number of primitives being submitted.

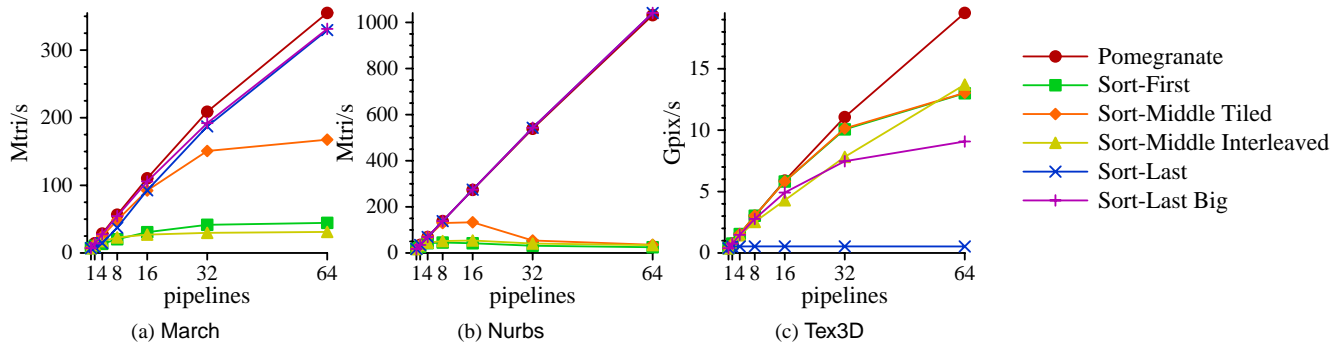


Figure 11: Performance of each architecture on each scene.

6.2 Communication

Pomegranate uses a network to interconnect all the pipeline stages. This approach to interconnecting a graphics system has become much more practical recently due to the advent of high-speed point-to-point signaling, which reduces the cost of providing the multiple high bandwidth links necessary in such a network. Nonetheless, we expect the cost of the network to dominate the cost of implementing an architecture like Pomegranate.

Pomegranate only achieves high scalability when it is able to use the network as a point-to-point communication mechanism. Every time broadcast communication is performed, scalability will be lost. However, some commands must be broadcast. Commands that modify the state of a particular context (e.g. `glBlendFunc`) must be broadcast to all of the units using that state. The command distribution could potentially be implemented lazily, but is still fundamentally a broadcast communication, which will impose a scalability limit. Most high performance applications already try to minimize the frequency of state changes to maximize performance. It remains to be seen how the potentially greater cost of state changes in Pomegranate would impact its scalability.

7 Conclusions

We have introduced Pomegranate, a new fully scalable graphics architecture. Simulated results demonstrate performance of up to 1.10 billion triangles per second and 21.8 billion pixels per second in a 64-way parallel system.

Pomegranate uses a high-speed point-to-point network to interconnect its pipeline stages, allowing each pipeline stage to provide a temporally balanced work load to each subsequent stage, without requiring broadcast communication. A novel ordering mechanism preserves the serial ordered semantics of the API while allowing full parallel execution of the command stream. Hardware support for a parallel host interface allows Pomegranate to scale to previously unreachable levels of performance for immediate-mode applications.

8 Acknowledgments

We would like to acknowledge the numerous helpful comments the reviewers made that have improved this work. We would like to thank Greg Humphreys, John Owens and the rest of the Stanford Graphics Lab for their reviews of this paper and their insights. This work was supported by the Fannie and John Hertz Foundation, Intel and DARPA contract DABT63-95-C-0085-P00006.

References

- [1] Freedom 3000 Technical Overview. Technical report, Evans & Sutherland Computer Corporation, October 1992.
- [2] Denali Technical Overview. Technical report, Kubota Pacific Computer Inc., March 1993.
- [3] William J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, pages 194–205, March 1992.
- [4] José Duato, Sudhakar Yalmanchili, and Lionel Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society Press, 1997.
- [5] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 95–106, August 1999.
- [6] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, August 1998.
- [7] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The Design of a Parallel Graphics Interface. *SIGGRAPH 98 Conference Proceedings*, pages 141–150, July 1998.
- [8] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (SIGGRAPH 87 Conference Proceedings)*, pages 163–169, July 1987.
- [9] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, pages 23–32, July 1994.
- [10] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, pages 231–240, July 1992.
- [11] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. *SIGGRAPH 97 Conference Proceedings*, pages 293–302, August 1997.
- [12] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load Balancing for Multi-Projector Rendering Systems. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 107–116, August 1999.