A gamer wanders through a virtual world rendered in near-cinematic detail. Seconds later, the screen fills with a 3D explosion, the result of unseen enemies hiding in physically accurate shadows. Disappointed, the user exits the game and returns to a computer desktop that exhibits the stylish 3D look-and-feel of a modern window manager. Both of these visual experiences require hundreds of gigaflops of computing performance, a demand met by the GPU (graphics processing unit) present in every consumer PC.

GPUs a closer look

As the line between GPUs and CPUs begins to blur, it's important to understand what makes GPUs tick.

KAYVON FATAHALIAN
and MIKE HOUSTON,
STANFORD UNIVERSITY

GPUs a closer look

The modern GPU is a versatile processor that constitutes an extreme but compelling point in the growing space of multicore parallel computing architectures. These platforms, which include GPUs, the STI Cell Broadband Engine, the Sun UltraSPARC T2, and, increasingly, multicore x86 systems from Intel and AMD, differentiate themselves from traditional CPU designs by prioritizing high-throughput processing of many parallel operations over the low-latency execution of a single task.

GPUs assemble a large collection of fixed-function and software-programmable processing resources. Impressive statistics, such as ALU (arithmetic logic unit) counts and peak floating-point rates often emerge during discussions of GPU design. Despite the inherently parallel nature of graphics, however, efficiently mapping common rendering algorithms onto GPU resources is extremely challenging.

The key to high performance lies in strategies that hardware components and their corresponding software interfaces use to keep GPU processing resources busy. GPU designs go to great lengths to obtain high efficiency, conveniently reducing the difficulty programmers face when programming graphics applications. As a result, GPUs deliver high performance and expose an expressive but simple programming interface. This interface remains largely devoid of explicit parallelism or asynchronous execution and has proven to be portable across vendor implementations and generations of GPU designs.

At a time when the shift toward throughput-oriented CPU platforms is prompting alarm about the complexity of parallel programming, understanding key ideas behind the success of GPU computing is valuable not only for developers targeting software for GPU execution, but also for informing the design of new architectures and programming systems for other domains. In this article, we dive under the hood of a modern GPU to look at why

A Simplified Graphics Pipeline

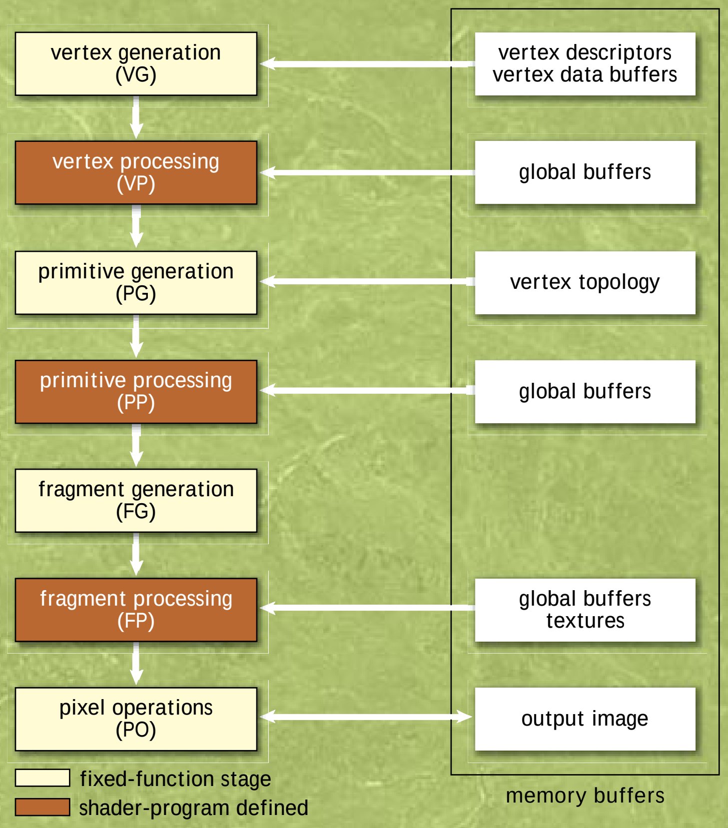


FIG 1

interactive rendering is challenging and to explore the solutions GPU architects have devised to meet these challenges.

THE GRAPHICS PIPELINE

A graphics system generates images that represent views of a virtual scene. This scene is defined by the geometry, orientation, and material properties of object surfaces and the position and characteristics of light sources. A scene view is described by the location of a virtual camera. Graphics systems seek to find the appropriate balance between conflicting goals of enabling maximum performance and maintaining an expressive but simple interface for describing graphics computations.

Realtime graphics APIs such as Direct3D and OpenGL strike this balance by representing the rendering computation as a *graphics processing pipeline* that performs operations on four fundamental entities: vertices, primitives, fragments, and pixels. Figure 1 provides a block diagram of a simplified seven-stage graphics pipeline. Data flows between stages in streams of entities. This pipeline contains fixed-function stages (tan) implementing API-specified operations and three programmable stages (brown) whose behavior is defined by application code. Figure 2 illustrates the operation of key pipeline stages.

VG (vertex generation). Realtime graphics APIs represent surfaces as collections of simple geometric primitives (points, lines, or triangles). Each primitive is defined by a set of vertices. To initiate rendering, the application provides the pipeline's VG stage with a list of vertex descriptors. From this list, VG prefetches vertex data from memory and constructs a stream of vertex data records for subsequent processing. In practice, each record contains the 3D (*x,y,z*) scene position of the vertex plus additional application-defined parameters such as surface color and normal vector orientation.

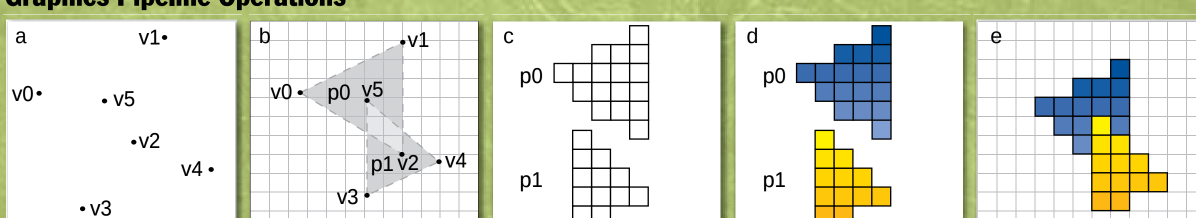
VP (vertex processing). The behavior of VP is application programmable. VP operates on each vertex independently and produces exactly one output vertex record from each input record. One of the most important operations of VP execution is computing the 2D output image (screen) projection of the 3D vertex position.

PG (primitive generation). PG uses vertex topology data provided by the application to group vertices from VP into an ordered stream of primitives (each primitive record is the concatenation of several VP output vertex records). Vertex topology also defines the order of primitives in the output stream.

PP (primitive processing). PP operates independently on each input primitive to produce zero or more output primitives. Thus, the output of PP is a new (potentially longer or shorter) ordered stream of primitives. Like VP, PP operation is application programmable.

FG (fragment generation). FG samples each primitive densely in screen space (this process is called *rasterization*). Each sample is manifest as a fragment record in the FG output stream. Fragment records contain the output image position of the surface sample, its distance from

Graphics Pipeline Operations



(A) Six vertices from the VG output stream define the scene position and orientation of two triangles.

(B) Following VP and PG, the vertices have been transformed into their screen-space positions and grouped into two triangle primitives, *p0* and *p1*.

(C) FG samples the two primitives, producing a set of fragments corresponding to *p0* and *p1*.

(D) FP computes the appearance of the surface at each sample location.

(E) PO updates the output image with contributions from the fragments, accounting for surface visibility. In this example, *p1* is nearer to the camera than *p0*. As a result *p0* is occluded by *p1*.

FIG 2

GPUs a closer look

the virtual camera, as well as values computed via interpolation of the source primitive's vertex parameters.

FP (fragment processing). FP simulates the interaction of light with scene surfaces to determine surface color and opacity at each fragment's sample point. To give surfaces realistic appearances, FP computations make heavy use of filtered lookups into large, parameterized 1D, 2D, or 3D arrays called *textures*. FP is an application-programmable stage.

PO (pixel operations). PO uses each fragment's screen position to calculate and apply the fragment's contribution to output image pixel values. PO accounts for a sample's distance from the virtual camera and discards fragments that are blocked from view by surfaces closer to the camera. When fragments from multiple primitives contribute to the value of a single pixel, as is often the case when semi-transparent surfaces overlap, many rendering techniques rely on PO to perform pixel updates in the order defined by the primitives' positions in the PP output stream. All graphics APIs guarantee this behavior, and PO is the only stage where the order of entity processing is specified by the pipeline's definition.

SHADER PROGRAMMING

The behavior of application-programmable pipeline stages (VP, PP, FP) is defined by *shader functions* (or shaders). Graphics programmers express vertex, primitive, and fragment shader functions in high-level *shading languages* such as NVIDIA's Cg, OpenGL's GLSL, or Microsoft's HLSL. Shader source is compiled into bytecode offline, then transformed into a GPU-specific binary by the graphics driver at runtime.

Shading languages support complex data types and a rich set of control-flow constructs, but they do *not* contain primitives related to explicit parallel execution. Thus, a shader definition is a C-like function that serially computes output-entity data records from a single input entity. Each function invocation is abstracted as an independent sequence of control that executes in complete isolation from the processing of other stream entities.

As a convenience, in addition to data records from stage input and output streams, shader functions may access (but not modify) large, globally shared data buffers. Prior to pipeline execution, these buffers are initialized to contain shader-specific parameters and textures by the application.

CHARACTERISTICS AND CHALLENGES

Graphics pipeline execution is characterized by the following key properties.

Opportunities for parallel processing. Graphics presents opportunities for both task (across pipeline stages) and data (stages operate independently on stream entities) parallelism, making parallel processing a viable strategy for increasing throughput. Despite abundant potential parallelism, however, constraints on the order of PO stage processing introduce dynamic, fine-grained dependencies that complicate parallel implementation throughout the pipeline. Although output image contributions from *most* fragments can be applied in parallel, those that contribute to the same pixel cannot.

Fixed-function stages encapsulate difficult-to-parallelize work. Each shader function invocation executes serially; programmable stages, however, are trivially parallelizable by executing shader functions simultaneously on multiple stream entities. In contrast, the pipeline's non-programmable stages involve multiple entity interactions (such as ordering dependencies in PO or vertex grouping in PG) and stateful processing. Isolating this non-data-parallel work into fixed stages keeps the shader programming model simple and allows the GPU's programmable processing components to be highly specialized for data-parallel execution. In addition, the separation enables difficult aspects of the graphics computation to be encapsulated in optimized, fixed-function hardware components.

Extreme variations in pipeline load. Although the number of stages and data flows of the graphics pipeline is fixed, the computational and bandwidth requirements of all stages vary significantly depending on the behavior of shader functions and properties of scenes. For example, primitives that cover large regions of the screen generate many more fragments than vertices. In contrast, many small primitives result in high vertex-processing demands. Applications frequently reconfigure the pipeline to use different shader functions that vary from tens of instructions to a few hundred. For these reasons, over the duration of processing for a single frame, different stages will dominate overall execution, often resulting in bandwidth- and compute-intensive phases of execution. Maintaining an efficient mapping of the graphics pipeline to a GPU's resources in the face of this variability is a significant challenge, as it requires processing and on-chip storage resources to be dynamically reallocated to pipeline stages, depending on current load.

Mixture of predictable and unpredictable data access. The graphics pipeline rigidly defines inter-stage data flows using streams of entities. This predictability presents opportunities for aggregate prefetching of stream data records and highly specialized hardware management on-chip storage resources. In contrast, buffer and texture accesses performed by shaders are fine-grained memory operations on dynamically computed addresses, making prefetch difficult. As both forms of data access are critical to maintaining high throughput, shader programming models explicitly differentiate stream from buffer/texture memory accesses, permitting specialized hardware solutions for both types of accesses.

Opportunities for instruction stream sharing. While the shader programming model permits each shader invocation to follow a unique stream of control, in practice, shader execution on nearby stream elements often results in the same dynamic control-flow decisions. As a result, multiple shader invocations can likely share an instruction stream. Although GPUs must accommodate situations where this is not the case, instruction stream sharing across multiple shader invocations is a key optimization in the design of GPU processing cores and is accounted for in algorithms for pipeline scheduling.

PROGRAMMABLE PROCESSING RESOURCES

A large fraction of a GPU's resources exist within programmable processing cores responsible for executing shader functions. While substantial implementation differences exist across vendors and product lines, all modern GPUs maintain high efficiency through the use of multicore designs that employ both hardware multithreading and SIMD (single instruction, multiple data) processing. As shown in table 1, these throughput-computing techniques are not unique to GPUs (top two rows). In comparison with CPUs, however, GPU designs push these ideas to extreme scales.

Multicore + SIMD Processing = Lots of ALUs. A thread of control is realized by a stream of processor instructions that execute within a processor-managed environment, called an execution (or thread) context. This context consists of states such as a program counter, a stack pointer, general-purpose registers, and virtual memory mappings. A multicore processor replicates processing resources (both ALUs and execution contexts) and organizes them into independent cores. When an application features multiple threads of control, multicore architectures provide increased throughput by executing these instruction streams on each core in parallel. For example, an Intel Core 2 Quad contains four cores and can execute four instruction streams simultaneously. As significant parallelism exists across shader invocations, GPU designs easily push core counts higher. High-end models contain up to 16 cores per chip.

Even higher performance is possible by populating each core with multiple floating-point ALUs. This is done efficiently with SIMD processing, which uses each ALU to perform the same operation on a different piece of data. The most common implementation of SIMD processing is via *explicit* short-vector instructions, similar to those provided by the x86 SSE or PowerPC Altivec ISA extensions. These extensions provide a SIMD width of four, with instructions that control the operation of four ALUs. Alternative implementations, such as NVIDIA's 8-series architecture, perform SIMD execution by *implicitly* shar-

TABLE 1 Tale of the Tape: Throughput Architectures

Type	Processor	Cores/Chip	ALUs/Core[3]	SIMD width	MaxT[4]
GPUs	AMD Radeon HD 2900	4	80	64	48
	NVIDIA GeForce 8800	16	8	32	96
CPUs	Intel Core 2 Quad[1]	4	8	4	1
	STI Cell BE[2]	8	4	4	1
	Sun UltraSPARC T2	8	1	1	4

[1] *SSE processing only, does not account for x86 FPU.*
[2] *Stream processing (SPE) cores only, does not account for PPU cores.*
[3] *32-bit, floating point (all ALUs are multiply-add except the Intel Core 2 Quad)*
[4] *The ratio of core thread contexts to simultaneously executable threads. We use the ratio T (rather than the total number of per-core thread contexts) to describe the extent to which processor cores automatically hide thread stalls via hardware multithreading.*

GPUs a closer look

ing an instruction across multiple threads with identical PCs. In either SIMD implementation, the complexity of processing an instruction stream and the cost of circuits and structures to control ALUs are amortized across multiple ALUs. The result is both power- and area-efficient chip execution.

CPU designs have converged on a SIMD width of four as a balance between providing increased throughput and retaining high single-threaded performance. Characteristics of the shading workload make it beneficial for GPUs to employ significantly wider SIMD processing (widths ranging from 32 to 64) and to support a rich set of operations. It is common for GPUs to support SIMD implementations of reciprocal square root, trigonometric functions, and memory gather/scatter operations.

The efficiency of wide SIMD processing allows GPUs to pack many cores densely with ALUs. For example, the NVIDIA GeForce 8800 Ultra GPU contains 128 single-precision ALUs operating at 1.5 GHz. These ALUs are organized into 16 processing cores and yield a peak rate of 384 Gflops (each ALU retires one 32-bit multiply-add per clock). In comparison, a high-end 3-GHz Intel Core 2 CPU contains four cores, each with eight SIMD floating-point ALUs (two 4-width vector instructions per clock), and is capable of, at most, 96 Gflops of peak performance.

GPUs execute groups of shader invocations in parallel to take advantage of SIMD processing. Dynamic per-entity control flow is implemented by executing all control paths taken by the shader invocations. SIMD operations that do not apply to all invocations, such as those within shader code conditional or loop blocks, are partially nullified using write-masks. In this implementation, when shader control flow diverges, fewer SIMD ALUs do useful work. Thus, on a chip with width-S SIMD processing, worst-case behavior yields performance equaling $1/S$ the chip's peak rate. Fortunately, shader workloads exhibit sufficient levels of instruction stream sharing to justify wide SIMD implementations. Additionally, GPU ISAs contain special instructions that make it possible for shader compilers to transform per-entity control flow into efficient sequences of SIMD operations.

Hardware Multithreading = High ALU Utilization. Thread stalls pose an additional challenge to high-performance shader execution. Threads *stall* (or block) when the processor cannot dispatch the next instruction in an instruction stream because of a dependency on an outstanding instruction. High-latency off-chip memory accesses, most notably those generated by fragment shader texturing operations, cause thread stalls lasting hundreds of cycles (recall that while shader input and output records lend themselves to streaming prefetch, texture accesses do not).

Allowing ALUs to remain idle during the period while a thread is stalled is inefficient. Instead, GPUs maintain more execution contexts on chip than they can simultaneously execute, and they perform instructions from runnable threads when others are stalled. Hardware scheduling logic determines which context(s) to execute in each processor cycle. This technique of overprovisioning cores with thread contexts to hide the latency of thread stalls is called *hardware multithreading*. GPUs use multithreading to hide both memory access and instruction pipeline latencies.

The latency-hiding ability of GPU multithreading is dependent on the ratio of hardware thread contexts to the number of threads that can be simultaneously executed in a clock (value T from table 1). Support for more thread contexts allows the GPU to hide longer or more frequent stalls. All modern GPUs maintain large numbers of execution contexts on chip to provide maximal memory latency-hiding ability (T ranges from 16 to 96). This represents a significant departure from CPU designs, which attempt to avoid or minimize stalls using large, low-latency data caches and complicated out-of-order execution logic. Current Intel Core 2 and AMD Phenom processors maintain one thread per core, and even high-end models of Sun's multithreaded UltraSPARC T2 processor manage only four times the number of threads they can simultaneously execute.

Note that in the absence of stalls, the throughput of single- and multithreaded processors is equivalent. Multithreading does not increase the number of processing resources on a chip. Rather, it is a strategy that interleaves execution of multiple threads in order to use existing resources more efficiently (improve throughput). On average, a multithreaded core operating at its peak rate runs each thread $1/T$ of the time.

Large-scale multithreading requires execution contexts to be compact in order to fit many contexts within on-chip memories. The number of thread contexts supported by a GPU core is shader-program dependent and typically limited by the size of on-chip storage. GPUs require compiled shader binaries to declare input and output entity sizes, as well as bounds on temporary storage and scratch registers required for execution. At runtime, GPUs use these bounds to partition unspillable on-chip storage (including data registers) dynamically among execution contexts. Thus, GPUs support many thread contexts (up to an architecture-specific bound) and, correspondingly, provide maximal latency-hiding ability when shaders use fewer resources. When shaders require large amounts of storage, the number of execution contexts provided by a GPU drops. (The accompanying sidebar details an example of the efficient execution of a fragment shader on a GPU core.)

FIXED-FUNCTION PROCESSING RESOURCES

A GPU's programmable cores interoperate with a collection of specialized fixed-function processing units that provide high-performance, power-efficient implementations of nonshader stages. These components do not simply augment programmable processing; they perform sophisticated operations and constitute an additional hundreds of gigaflops of processing power. Two of the most important operations performed via fixed-function hardware are texture filtering and rasterization (fragment generation).

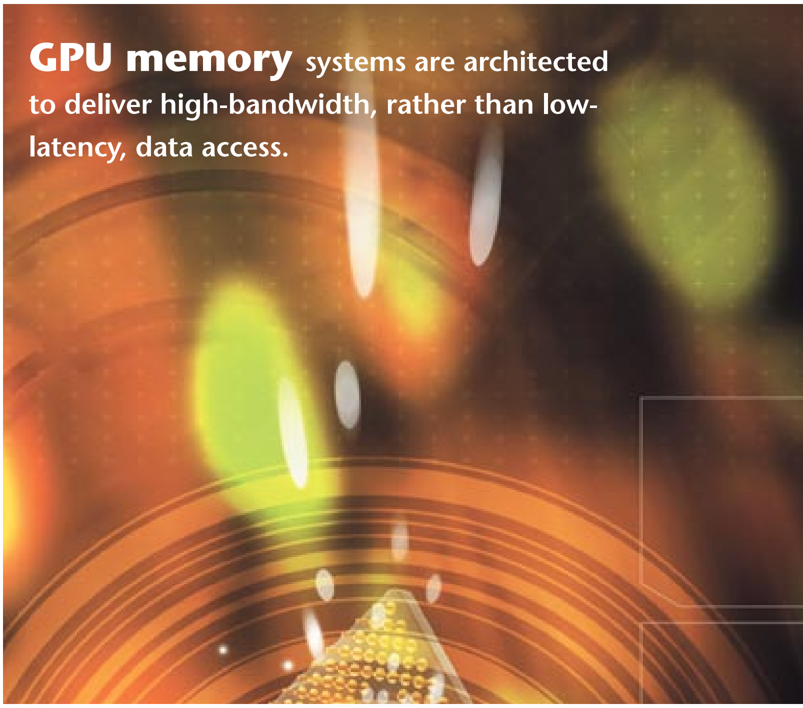
Texturing is handled almost entirely by fixed-function logic. A texturing operation samples a contiguous 1D, 2D, or 3D signal (a texture) that is discretely represented by a multidimensional array of color values (2D texture data is simply an image). A GPU texture-filtering unit accepts a point within the texture's parameterization (represented by a floating-point tuple, such as {.5,.75}) and loads array values surrounding the coordinate from memory. The values are then filtered to yield a single result that represents the texture's value at the specified coordinate. This value is returned to the calling shader function. Sophisticated texture filtering is required for generating high-quality images. As graphics APIs provide a finite set of filtering kernels, and because filtering kernels are computationally expensive, texture filtering is well suited for fixed-function processing.

Primitive rasterization in the FG stage is another key pipeline operation implemented by fixed-function components. Rasterization involves densely sampling a primitive (at least once per output image pixel) to determine which pixels the primitive overlaps. This process involves interpolating the location of the surface at each sample point and then generating fragments for all sample points covered by the primitive. Bounding-box computations and hierarchical techniques optimize the rasterization process. Nonetheless, rasterization involves significant computation.

In addition to the components for texturing and rasterization, GPUs contain dedicated hardware components for operations such as surface visibility determination, output pixel compositing, and data compression/decompression.

THE MEMORY SYSTEM

Parallel-processing resources place extreme load on a GPU's memory system, which services memory requests from both fixed-function and programmable compo-



nents. These requests include a mixture of fine-granularity and bulk prefetch operations and may even require realtime guarantees (such as display scan out).

Recall that a GPU's programmable cores tolerate large memory latencies via hardware multithreading and that interstage stream data accesses can be prefetched. As a result, GPU memory systems are architected to deliver high-bandwidth, rather than low-latency, data access. High throughput is obtained through the use of wide

GPUs a closer look

memory buses and specialized GDDR (graphics double data rate) memories that operate most efficiently when memory access granularities are large. Thus, GPU memory controllers must buffer, reorder, and then coalesce large numbers of memory requests to synthesize large operations that make efficient use of the memory system. As an example, the ATI HD 2700XT memory controller manipulates thousands of outstanding requests to deliver 105 GB per second of bandwidth from GDDR3 memories attached to a 512-bit bus.

GPU data caches meet different needs from CPU caches. GPUs employ relatively small, read-only caches (no cache coherence) that filter requests destined for the memory controller and reduce bandwidth requirements placed on main memory. Thus, GPU caches typically serve to amplify total bandwidth to processing units rather than decrease latency of memory accesses. Interleaved execution of many threads renders large read-write caches inefficient because of severe cache thrashing. GPUs benefit from small caches that capture spatial locality across simultaneously executed shader invocations. This situation is common, as texture accesses performed while processing fragments in close screen proximity are likely to have overlapping texture-filter support regions.

Although most GPU caches are small, this does not imply that GPUs contain little on-chip storage. Significant amounts of on-chip storage are used to hold entity streams, execution contexts, and thread scratch data.

PIPELINE SCHEDULING AND CONTROL

Mapping the entire graphics pipeline efficiently onto GPU resources is a challenging problem that requires dynamic and adaptive techniques. A unique aspect of GPU computing is that hardware logic assumes a major role in mapping and scheduling computation onto chip resources. GPU hardware "scheduling" logic extends beyond the thread-scheduling responsibilities discussed in previous sections. GPUs automatically assign computations to threads, clean up after threads complete, size and manage buffers that hold stream data, guarantee ordered processing when needed, and identify and discard unnecessary pipeline work. This logic relies heavily on specific upfront knowledge of graphics workload characteristics.

Conventional thread programming uses operating-system or threading API mechanisms for thread creation, completion, and synchronization on shared structures. Large-scale multithreading coupled with the brevity of shader function execution (at most a few hundred instructions), however, means GPU thread management must be performed entirely by hardware logic.

GPUs minimize thread launch costs by preconfiguring execution contexts to run one of the pipeline's three types of shader functions and reusing the configuration multiple times for shaders of the same type. GPUs launch threads when a shader stage's input stream contains a sufficient number of entities, and then they automatically provide threads access to shader input records. Similar hardware logic commits records to the output stream buffer upon thread completion. The distribution of execution contexts to shader stages is reprovisioned periodically as pipeline needs change and stream buffers drain or approach capacity.

GPUs leverage upfront knowledge of pipeline entities to identify and skip unnecessary computation. For example, vertices shared by multiple primitives are identified and VP results cached to avoid duplicate vertex processing. GPUs also discard fragments prior to FP when the fragment will not alter the value of any image pixel. Early fragment discard is triggered when a fragment's sample point is occluded by a previously processed surface located closer to the camera.

Another class of hardware optimizations reorganizes fine-grained operations for more efficient processing. For example, rasterization orders fragment generation to maximize screen proximity of samples. This ordering improves texture cache hit rates, as well as instruction stream sharing across shader invocations. The GPU memory controller also performs automatic reorganization when it reorders memory requests to optimize memory bus and DRAM utilization.

GPUs ensure inter-fragment PO ordering dependencies using hardware logic. Implementations use structures such as post-FP reorder buffers or scoreboards that delay fragment thread launch until the processing of overlapping fragments is complete.

GPU hardware can take responsibility for sophisticated scheduling decisions because semantics and invariants of

Running a Fragment Shader on a GPU Core

Shader compilation to SIMD (single instruction, multiple data) instruction sequences coupled with dynamic hardware thread scheduling leads to efficient execution of a fragment shader on the simplified single-core GPU shown in figure A.

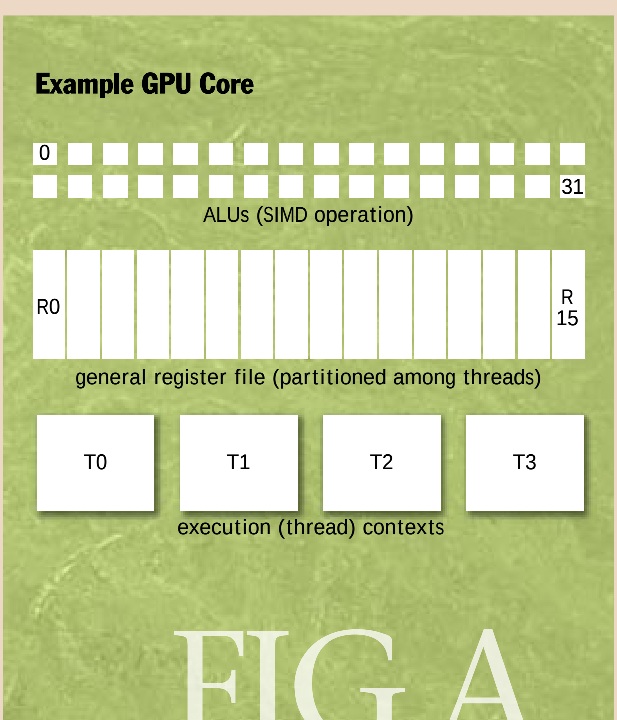
- The core executes an instruction from at most one thread each processor clock, but maintains state for four threads on-chip simultaneously (*T*=4).
- Core threads issue explicit width-32 SIMD vector instructions; 32 ALUs simultaneously execute a vector instruction in a single clock.
- The core has a pool of 16 general-purpose vector registers (R0 to R15) that are partitioned among thread contexts. The elements of each length-32 vector are 32-bit values.
- The only source of thread stalls is texture access; they have a maximum latency of 50 cycles.

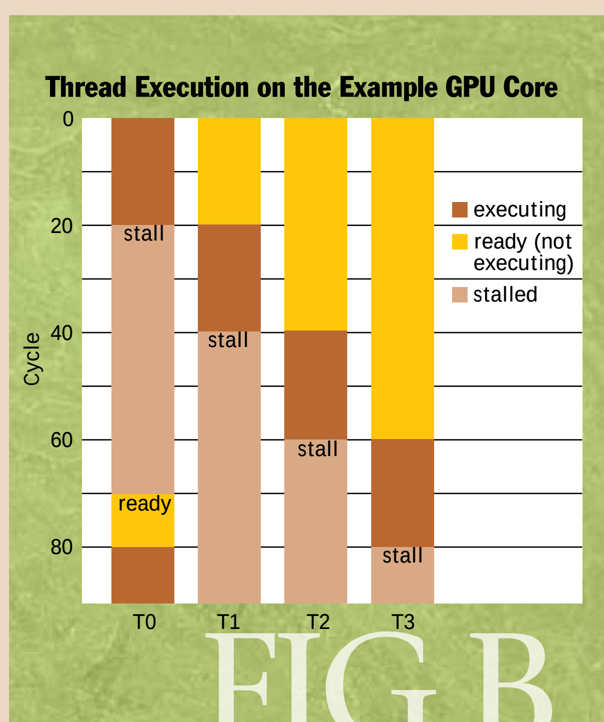
Shader compilation by the graphics driver produces a GPU binary from a high-level fragment shader source. The resulting vector instruction sequence performs 32 invocations of the fragment shader simultaneously by carrying out each invocation in a single lane of the width-32 vectors. The compiled binary requires four vector registers for temporary results and contains 20 arithmetic instructions between each texture access operation.

At runtime, the GPU executes a copy of the shader binary on each of its four thread contexts, as illustrated in figure B. The core executes T0 (thread 0) until it detects a stall resulting from texture access in cycle 20. While T0 waits for the result of the texturing operation, the core continues to execute its remaining three threads. The result of T0's texture access becomes available in cycle 70. Upon T3's stall in cycle 80, the core immediately resumes T0. Thus, at no point during execution are ALUs left idle.

When executing the shader program for this example, a minimum of four threads is needed to keep core ALUs busy. Each thread operates simultaneously on 32 fragments; thus, 4*32=128 fragments are required for the chip to achieve peak performance.

As memory latencies on real GPUs involve hundreds of cycles, modern GPUs must contain support for significantly more threads to sustain high utilization. If we extend our simple GPU to a more realistic size of eight processing cores and provision each core with storage for 16 execution contexts, then simultaneous processing of 4,096 fragments is needed to approach peak processing rates. Clearly, GPU performance relies heavily on the abundance of parallel shading work.





GPUs a closer look

the graphics pipeline are known a priori. Hardware implementation enables fine-granularity logic that is informed by precise knowledge of both the graphics pipeline and the underlying GPU implementation. As a result, GPUs are highly efficient at using all available resources. The drawback of this approach is that GPUs execute only those computations for which these invariants and structures are known.

Graphics programming is becoming increasingly versatile. Developers constantly seek to incorporate more sophisticated algorithms and leverage more configurable graphics pipelines. Simultaneously, the growing popularity of GPGPU (general-purpose computing using GPU platforms) has led to new interfaces for accessing GPU resources. Given both of these trends, the extent to which GPU designers can embed a priori knowledge of computations into hardware scheduling logic will inevitably decrease over time.

A major challenge in the evolution of GPU programming involves preserving GPU performance levels while increasing the generality and expressiveness of application interfaces. The designs of GPGPU interfaces, such as NVIDIA's CUDA and AMD's CAL, are evidence of how difficult this challenge is. These frameworks abstract computation as large batch operations that involve many invocations of a kernel function operating in parallel. The resulting computations execute on GPUs efficiently only under conditions of massive data parallelism. Programs that attempt to implement non-data-parallel algorithms perform poorly.

GPGPU programming models are simple to use and permit well-written programs to make good use of both GPU programmable cores and (if needed) texturing resources. Programs using these interfaces, however, cannot use powerful fixed-function components of the chip, such as those related to compression, image compositing, or rasterization. Also, when these interfaces are enabled, much of the logic specific to graphics-pipeline scheduling is simply turned off. Thus, current GPGPU programming frameworks restrict computations so that their structure, as well as their use of chip resources, remains sufficiently simple for GPUs to run these programs in parallel.

GPU AND CPU CONVERGENCE

The modern graphics processor is a powerful computing platform that resides at the extreme end of the design space of throughput-oriented architectures. A GPU's processing resources and accompanying memory system are heavily optimized to execute large numbers of operations in parallel. In addition, specialization to the graphics domain has enabled the use of fixed-function processing and allowed hardware scheduling of a parallel computation to be practical. With this design, GPUs deliver unsurpassed levels of performance to challenging workloads while maintaining a simple and convenient programming interface for developers.

Today, commodity CPU designs are adopting features common in GPU computing, such as increased core counts and hardware multithreading. At the same time, each generation of GPU evolution adds flexibility to previous high-throughput GPU designs. Given these trends, software developers in many fields are likely to take interest in the extent to which CPU and GPU architectures and, correspondingly, CPU and GPU programming systems, ultimately converge. Q

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

KAYVON FATAHALIAN is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming systems for commodity parallel architectures and computer graphics/animation systems for the interactive and film domains. His thesis research seeks to enable execution of more flexible rendering pipelines on future GPUs and multicore PCs. He will soon be looking for a job.
MIKE HOUSTON is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming models, algorithms, and runtime systems for parallel architectures including GPUs, Cell, multicore CPUs, and clusters. His dissertation includes the Sequoia runtime system, a system for programming hierarchical memory machines. He received his B.S. in computer science from UCSD in 2001 and is a recipient of the Intel Graduate Fellowship.