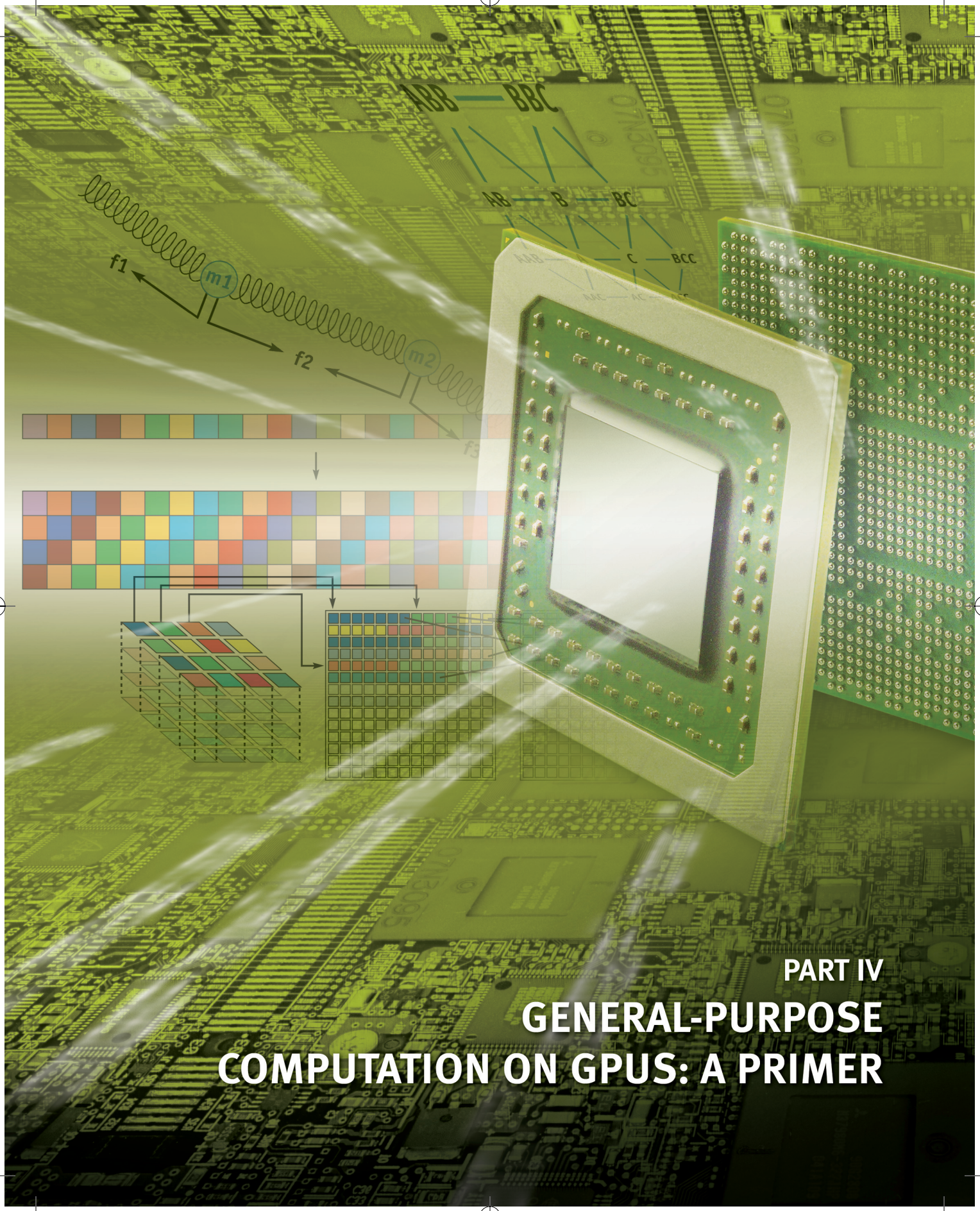


PART IV
GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER

This part of the book aims to provide a gentle introduction to the world of general-purpose computation on graphics processing units, or "GPGPU," as it has come to be known. The text is intended to be understandable to programmers with no graphics experience, as well as to those who have been programming graphics for years but have little knowledge of parallel computing for other applications.

Since the publication of *GPU Gems*, GPGPU has grown from something of a curiosity to a well-respected active new area of graphics and systems research.

Why would you want to go to the trouble of converting your computational problems to run on the GPU? There are two reasons: price and performance. Economics and the rise of video games as mass-market entertainment have driven down prices to the point where you can now buy a graphics processor capable of several hundred billion floating-point operations per second for just a few hundred dollars.

The GPU is not well suited to all types of problems, but there are many examples of applications that have achieved significant speedups from using graphics hardware. The applications that achieve the best performance are typically those with high "arithmetic intensity"; that is, those with a large ratio of mathematical operations to memory accesses. These applications range all the way from audio processing and physics simulation to bioinformatics and computational finance.

Anybody with any exposure to modern computing cannot fail to notice the rapid pace of technological change in our industry. The first chapter in this part, **Chapter 29, "Streaming Architectures and Technology Trends,"** by **John Owens** of the University of California, Davis, sets the stage for the chapters to come by describing the trends in semiconductor design and manufacturing that are driving the evolution of both the CPU and the GPU. One of the important factors driving these changes is the memory "gap"—the fact that computation speeds are increasing at a much faster rate than memory access speeds. This chapter also introduces the "streaming" computational model, which is a reasonably close match to the characteristics of modern GPU hardware. By using this style of programming, application programmers can take advantage of the GPU's massive computation and memory bandwidth resources, and the resulting programs can achieve large performance gains over equivalent CPU implementations.

Chapter 30, "The GeForce 6 Series GPU Architecture," by **Emmett Kilgariff** and **Randima Fernando** of NVIDIA, describes in detail the design of a current state-of-the-art graphics processor, the GeForce 6800. Cowritten by one of the lead architects of the chip, this chapter includes many low-level details of the hardware that are not available anywhere else. This information is invaluable for anyone writing high-performance GPU applications.

The remainder of this part of the book then moves on to several tutorial-style chapters that explain the details of how to solve general-purpose problems using the GPU.

Chapter 31, "Mapping Computational Concepts to GPUs," by **Mark Harris** of NVIDIA, discusses the issues involved with converting computational problems to run efficiently on the parallel hardware of the GPU. The GPU is actually made up of several programmable processors plus a selection of fixed-function hardware, and this chapter describes how to make the best use of these resources.

Chapter 32, "Taking the Plunge into GPU Computing," by **Ian Buck** of Stanford University, provides more details on the differences between the CPU and the GPU in terms of memory bandwidth, floating-point number representation, and memory access models. As Ian mentions in his introduction, the GPU was not really designed for general-purpose computation, and getting it to operate efficiently requires some care.

One of the most difficult areas of GPU programming is general-purpose data structures. Data structures such as lists and trees that are routinely used by CPU programmers are not trivial to implement on the GPU. The GPU doesn't allow arbitrary memory access and mainly operates on four-vectors designed to represent positions and colors. Particularly difficult are sparse data structures that do not have a regular layout in memory and where the size of the structure may vary from element to element.

Chapter 33, "Implementing Efficient Parallel Data Structures on GPUs," by **Aaron Lefohn** of the University of California, Davis; **Joe Kniss** of the University of Utah; and **John Owens** gives an overview of the stream programming model and goes on to explain the details of implementing data structures such as multi-dimensional arrays and sparse data structures on the GPU.

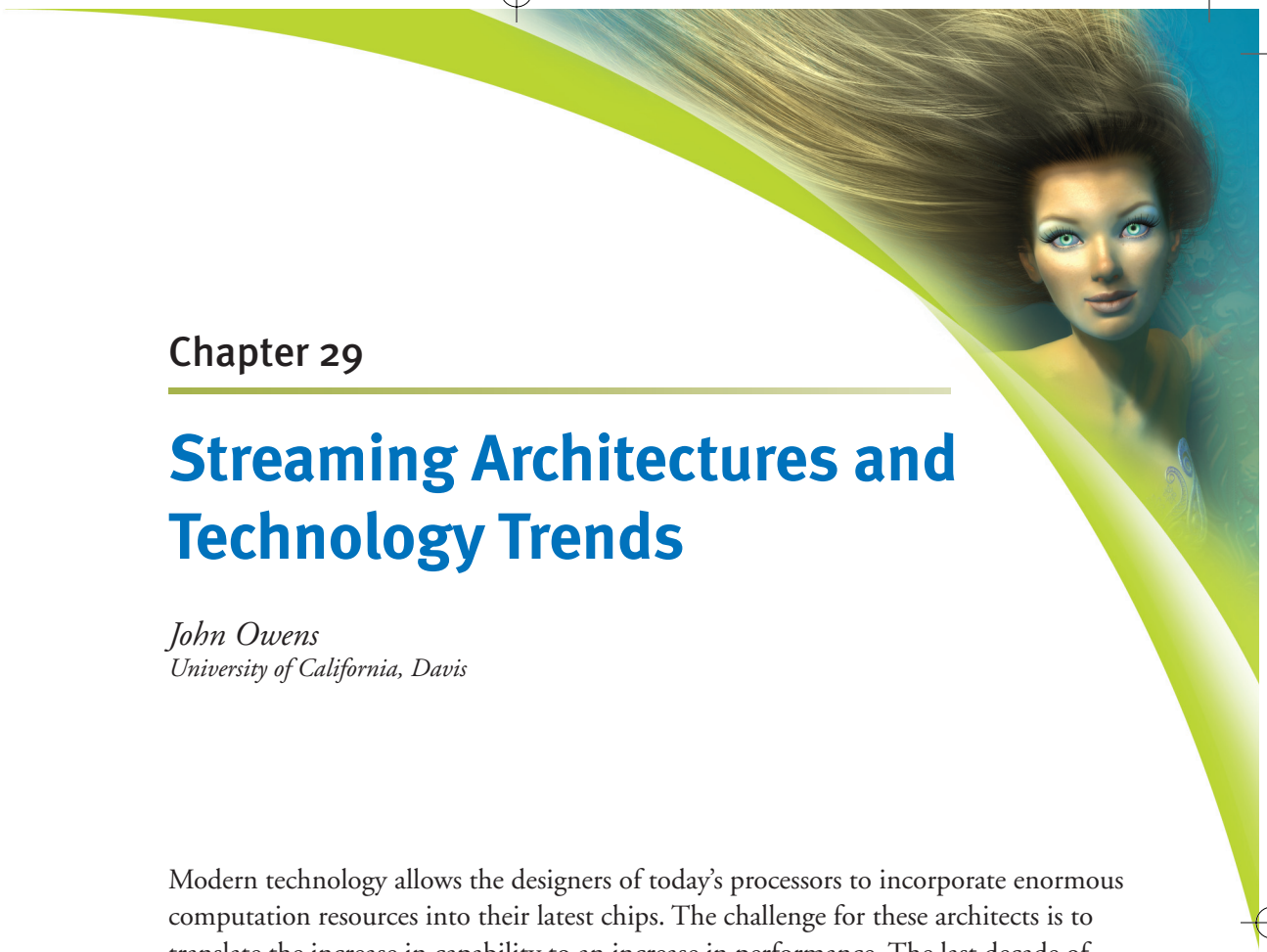
Traditionally, GPUs have not been very good at executing code with branches. Because they are parallel machines, they achieve best performance when the the same operation can be applied to every data element. **Chapter 34, "GPU Flow-Control Idioms,"** by **Mark Harris** and **Ian Buck**, explains different ways in which flow-control structures such as loops and if statements can be efficiently implemented on the GPU. This includes using the depth-test and z-culling capabilities of modern GPUs, as well as the branching instructions available in the latest versions of the pixel shader hardware.

Cliff Woolley of the University of Virginia has spent many hours writing GPGPU applications, and (like many of our other authors) he has published several papers based on his research. In **Chapter 35, "GPU Program Optimization,"** he passes on his experience on the best ways to optimize GPU code, and how to avoid the common mistakes made by novice GPU programmers. It is often said that premature optimization is the root of all evil, but it has to be done at some point.

On the CPU, it is easy to write programs that have variable amounts of output per input data element. Unfortunately, this is much more difficult on a parallel machine like the GPU. **Chapter 36, "Stream Reduction Operations for GPGPU Applications,"** by **Daniel Horn** of Stanford University, illustrates several ways in which the GPU can be programmed to perform filtering operations that remove elements from a data stream in order to generate variable amounts of output. He demonstrates how this technique can be used to efficiently implement collision detection and subdivision surfaces.

Only time will tell what the final division of labor between the CPU, the GPU, and other processors in the PC ecosystem will be. One thing is sure: the realities of semiconductor design and the memory gap mean that data-parallel programming is here to stay. By learning how to express your problems in this style today, you can ensure that your code will continue to execute at the maximum possible speed on all future hardware.

Simon Green, NVIDIA Corporation



Chapter 29

Streaming Architectures and Technology Trends

John Owens
University of California, Davis

Modern technology allows the designers of today's processors to incorporate enormous computation resources into their latest chips. The challenge for these architects is to translate the increase in capability to an increase in performance. The last decade of graphics processor development shows that GPU designers have succeeded spectacularly at this task. In this chapter, we analyze the technology and architectural trends that motivate the way GPUs are built today and what we might expect in the future.

29.1 Technology Trends

As computer users, we have become accustomed to each new generation of computer hardware running faster, with more capabilities—and often a lower price—than the last. This remarkable pace of development is made possible by continued advances in the underlying technologies, allowing more processing power to be placed on each chip. Each year in the *International Technology Roadmap for Semiconductors* (ITRS), the semiconductor industry forecasts how a number of chip metrics of interest, such as the size of transistors, the number of transistors per chip, and overall power consumption, will change in the coming years (ITRS 2003). These projections have an enormous impact on the companies that make chips and chip-making equipment, as well as the designers of next-generation chips. In this section we explain some of the trends we can expect in the future, as well as what they will mean for the development of next-generation graphics processors.

29.1.1 Core Technology Trends

Today's processors are constructed from millions of connected switching devices called transistors. As process technologies advance, these transistors, and the connections between them, can be fabricated in a smaller area. In 1965, Gordon Moore noted that the number of transistors that could be economically fabricated on a single processor die was doubling every year (Moore 1965). Moore projected that such an increase was likely to continue in the future. This oft-quoted prediction, termed "Moore's Law," today means that each year, approximately 50 percent more components can be placed on a single die.[1] In the forty years since Moore made his prediction, the number of transistors per die has gone from fifty (in 1965) to hundreds of millions (in 2005), and we can expect this rate of growth to continue for at least the next decade.

New chip generations not only increase the number of transistors, they also decrease transistor size. Because of their smaller size, these transistors can operate faster than their predecessors, allowing the chip to run faster overall. Historically, transistor speeds have increased by 15 percent per year (Dally and Poulton 1998). In modern processors, a global signal called a *clock* synchronizes computation that occurs throughout the processor, and so processor users see the increase in transistor speed reflected in a faster clock. Together, the increase in transistor count and clock speed combine to increase the *capability* of a processor at 71 percent per year. This yearly increase in capability means that each year, we can expect 71 percent more computation on a chip compared to the year before.

Semiconductor computer memory, which uses slightly different fabrication methods than processor logic, also benefits from similar advances in fabrication technology. The ITRS forecasts that commodity dynamic random-access memory (DRAM) will continue to double in capacity every three years. DRAM performance can be measured in two ways: by *bandwidth*, which measures the amount of data it can transfer each second, and by *latency*, which measures the length of time between the time data is requested and the time it is returned. DRAM performance does not increase as quickly as processor capability. DRAM bandwidth increases by 25 percent each year (ITRS 2003, Tables 4c, 4d), and DRAM latency improves by only 5 percent per year.

29.1.2 Consequences

In general, most of the trends just described are positive ones: with each new generation of fabrication technology, processor capability, memory bandwidth, and memory latency

1. Though references to "Moore's Law" in the popular press often refer to *performance* increases, Moore's actual prediction referred only to the number of devices that could fit on a single die.

all improve. For example, the yearly capability increase has led to an enormous degree of integration on a single die. Fifteen years ago, designers were only just beginning to integrate floating-point arithmetic units onto a processor die; today, such a unit occupies less than a square millimeter of die area, and hundreds can be placed onto the same die.

However, the most important consequences of these technology trends are the *differences* between them. When one metric changes at a different rate than another, it requires rethinking the assumptions behind processor and system design. We can identify three major issues that will help drive GPU architectures of the future: compute versus communicate, latency versus bandwidth, and power.

Compute vs. Communicate

As both clock speeds and chip sizes increase, the amount of time it takes for a signal to travel across an entire chip, measured in clock cycles, is also increasing. On today's fastest processors, sending a signal from one side of a chip to another typically requires multiple clock cycles, and this amount of time increases with each new process generation. We can characterize this trend as an increase in the cost of communication when compared to the cost of computation. As a consequence, designers of the future will increasingly use computation via cheap transistors to replace the need for expensive communication. Another likely impact will be an increase in the amount of computation available per word of memory bandwidth. As an example, let us compare NVIDIA's last three flagship GPUs (2002's GeForce FX 5800, 2003's GeForce FX 5950, and 2004's GeForce 6800) in measured peak programmable floating-point performance against peak off-chip bandwidth to memory. The GeForce FX 5800 could sustain 2 floating-point operations for every word of off-chip bandwidth, while the GeForce FX 5950 could sustain 2.66 operations, and the GeForce 6800 could sustain nearly 6. We expect this trend to continue in future chip generations. Figure 29-1 shows historical data for observed floating-point operations per second and available memory bandwidth for a series of GPU architectures.

Latency vs. Bandwidth

The gap between the trends of bandwidth and latency will also be an important driver of future architectures. Because latency will continue to improve more slowly than bandwidth (Patterson 2004), designers must implement solutions that can tolerate larger and larger amounts of latency by continuing to do useful work while waiting for data to return from operations that take a long time.

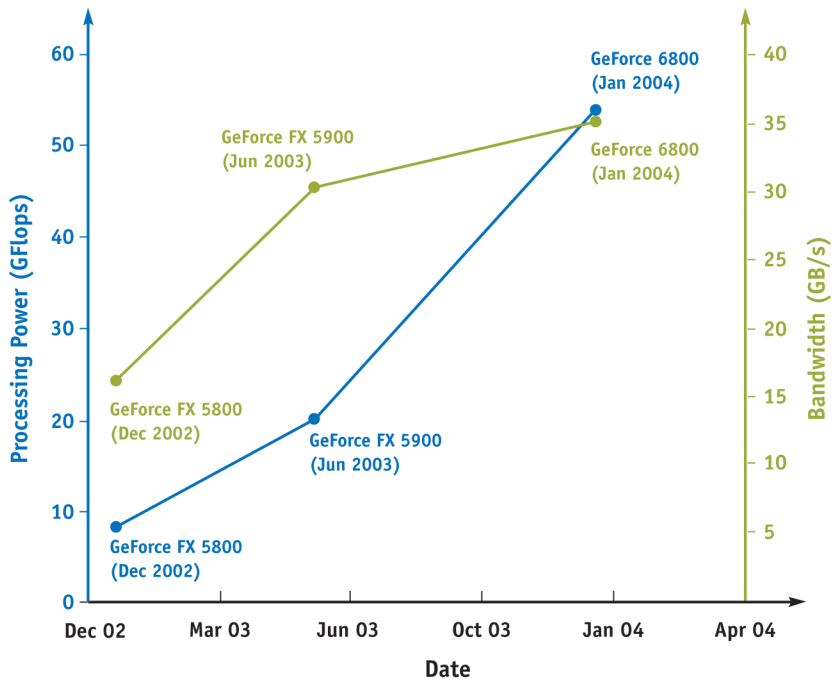


Figure 29-1. Rapidly Changing GPU Capabilities
The number of observed floating-point operations per second on the GeForce FX 5800, 5950, and the GeForce 6800 has been growing at a rapid pace, while off-chip memory bandwidth has been increasing much more slowly. (Data courtesy of Ian Buck, Stanford University)

Power

Although smaller transistors require less power than larger ones, the number of transistors on a single processor die is rising faster than the amount at which power per transistor is falling. Consequently, each generation of processors requires more power: the ITRS estimates that the maximum power allowed for 2004 chips with a heat sink is 158 watts and will gradually rise to a ceiling of 198 watts by 2008. This power constraint will be one of the primary limitations of future processors; the future figure of merit may no longer be the number of operations per second but instead the number of operations per second per watt.

Figure 29-2 summarizes the forecasted change in capability, DRAM bandwidth, DRAM latency, and power over the next decade.

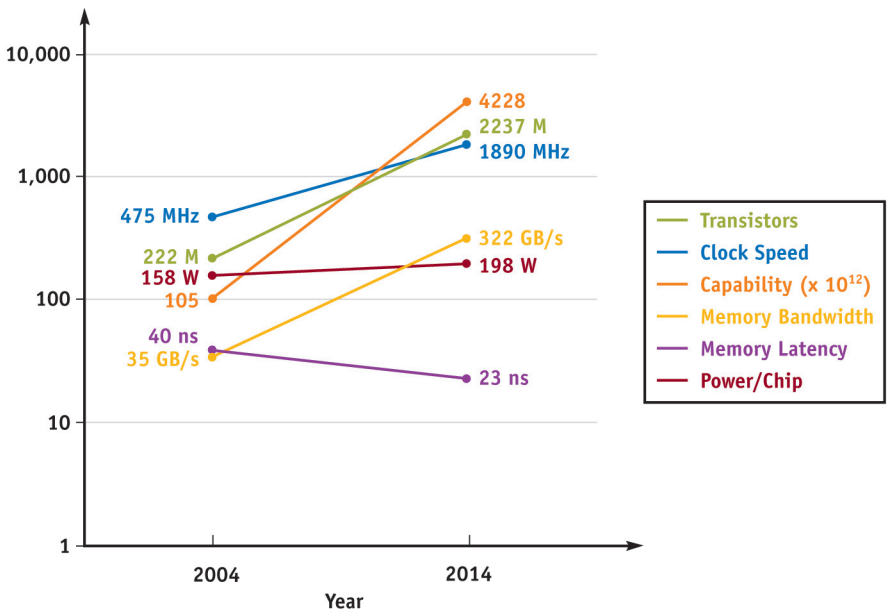


Figure 29-2. Changes in Key GPU Properties over Time

29.2 Keys to High-Performance Computing

In the previous section, we have seen that modern technology allows each new generation of hardware a substantial increase in capability. Effectively utilizing this wealth of computational resources requires addressing two important goals. First, we must organize our computational resources to allow high performance on our applications of interest. Simply providing large amounts of computation is not sufficient, however; efficient management of communication is necessary to feed the computation resources on the chip. In this section we describe techniques that allow us to achieve efficient computation and efficient communication, and we discuss why modern microprocessors (CPUs) are a poor match for these goals.

29.2.1 Methods for Efficient Computation

In Section 29.1 we saw that it is now possible to place hundreds to thousands of computation units on a single die. The keys to making the best use of transistors for computation are to maximize the hardware devoted to computation, to allow multiple computation units to operate at the same time through parallelism, and to ensure that each computation unit operates at maximum efficiency.

Though modern technologies allow us a large number of transistors on each die, our resources are not infinite. Our use of transistors can be broadly divided into three categories: *control*, the hardware used to direct the computation; *datapath*, the hardware used to perform the computation; and *storage*, the hardware used to store data. If our goal is to maximize performance, we must make hardware and software decisions that allow us to maximize the transistors in the datapath that are devoted to performing computation.

Within the datapath, we can allow simultaneous operations at the same time. This technique is termed *parallelism*. We can envision several ways to exploit parallelism and permit simultaneous execution. Complex tasks such as graphics processing are typically composed of several sequential tasks. When running these applications, we may be able to run several of these tasks on different data at the same time (*task parallelism*). Within a stage, if we are running a task on several data elements, we may be able to exploit *data parallelism* in evaluating them at the same time. And within the complex evaluation of a single data element, we may be able to evaluate several simple operations at the same time (*instruction parallelism*). To effectively use hundreds of computation units, we may take advantage of any or all of these types of parallelism.

Within each of the tasks, we could make each of our arithmetic units fully programmable and thus able to perform any task. However, we can gain more efficiency from our transistors by taking advantage of *specialization*. If a specific arithmetic unit performs only one kind of computation, that unit can be specialized to that computation with a considerable gain in efficiency. For example, triangle rasterization, in which a screen-space triangle is transformed into the fragments that cover that triangle, realizes an enormous efficiency benefit when the rasterization task is implemented in special-purpose hardware instead of programmable hardware.

29.2.2 Methods for Efficient Communication

As we saw in Section 29.1.2, off-chip bandwidth is growing more slowly than on-chip arithmetic capability, so high-performance processors must minimize off-chip communication. The easiest way to reduce this cost is to eliminate it: modern processors attempt to keep as much required data communication on-chip as possible, requiring off-chip communication only to fetch or store truly global data.

Another common way to mitigate the increasing cost of communication is through caching: a copy of recently used data memory is stored on-chip, removing the need for a fetch from off-chip if that data is needed again. Such data caching will be more com-

mon in future architectures, extending to local caches or user-controlled memories that relieve the need for on-chip communication. These caches effectively trade transistors in the form of cache memory for bandwidth. Another powerful technique is compression; only the compressed form of data is transmitted and stored off-chip. Compression also trades transistors (compression and decompression hardware) and computation (the compression/decompression operation) for off-chip bandwidth.

29.2.3 Contrast to CPUs

Today's high-performance microprocessors target general-purpose applications with different goals from a computer graphics pipeline. In general, these general-purpose programs have less parallelism, more complex control requirements, and lower performance goals than the rendering pipeline. Consequently, the design goals we enumerated previously are not those addressed by CPUs, and CPUs have made different design choices that result in a poor mapping to the graphics pipeline and many other applications with similar attributes.

CPU programming models are generally serial ones that do not adequately expose data parallelism in their applications. CPU hardware reflects this programming model: in the common case, CPUs process one piece of data at a time and do not exploit data parallelism. They do an admirable job of taking advantage of instruction parallelism, and recent CPU additions to the instruction set such as Intel's SSE and the PowerPC's AltiVec allow some data parallel execution, but the degree of parallelism exploited by a CPU is much less than that of a GPU.

One reason parallel hardware is less prevalent in CPU datapaths is the designers' decision to devote more transistors to control hardware. CPU programs have more complex control requirements than GPU programs, so a large fraction of a CPU's transistors and wires implements complex control functionality such as branch prediction and out-of-order execution. Consequently, only a modest fraction of a CPU's die is devoted to computation.

Because CPUs target general-purpose programs, they do not contain specialized hardware for particular functions. GPUs, however, can implement special-purpose hardware for particular tasks, which is far more efficient than a general-purpose programmable solution could ever provide.

Finally, CPU memory systems are optimized for minimum latency rather than the maximum throughput targeted by GPU memory systems. Lacking parallelism, CPU programs must return memory references as quickly as possible to continue to make

progress. Consequently, CPU memory systems contain several levels of cache memory (making up a substantial fraction of the chip's transistors) to minimize this latency. However, caches are ineffective for many types of data, including graphics inputs and data that is accessed only once. For the graphics pipeline, maximizing throughput for all elements rather than minimizing latency for each element results in better utilization of the memory system and a higher-performance implementation overall.

29.3 Stream Computation

In the previous section, we have seen that building high-performance processors today requires both efficient computation and efficient communication. Part of the reason that CPUs are poorly suited to many of these high-performance applications is their serial programming model, which does not expose the parallelism and communication patterns in the application. In this section, we describe the *stream* programming model, which structures programs in a way that allows high efficiency in computation and communication (Kapasi et al. 2003). This programming model is the basis for programming GPUs today.

29.3.1 The Stream Programming Model

In the stream programming model, all data is represented as a *stream*, which we define as an ordered set of data of the same data type. That data type can be simple (a stream of integers or floating-point numbers) or complex (a stream of points or triangles or transformation matrices). While a stream can be any length, we will see that operations on streams are most efficient if streams are long (hundreds or more elements in a stream). Allowed operations on streams include copying them, deriving substreams from them, indexing into them with a separate index stream, and performing computation on them with *kernels*.

A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. The defining characteristic of a kernel is that it operates on entire *streams* of elements as opposed to individual elements. The most typical use of a kernel is to evaluate a function on each element of an input stream (a "map" operation); for example, a transformation kernel may project each element of a stream of points into a different coordinate system. Other desirable kernel operations include expansions (in which more than one output element is produced for each input element), reductions (in which more than one element is combined into a single output element), or filters (in which a subset of input elements are output).

Kernel outputs are functions only of their kernel inputs, and within a kernel, computations on one stream element are never dependent on computations on another element. These restrictions have two major advantages. First, the data required for kernel execution is completely known when the kernel is written (or compiled). Kernels can thus be highly efficient when their input elements and their intermediate computed data are stored locally or are carefully controlled global references. Second, requiring independence of computation on separate stream elements within a single kernel allows mapping what appears to be a serial kernel calculation onto data-parallel hardware.

In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. Figure 29-3 shows how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline.

The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels.

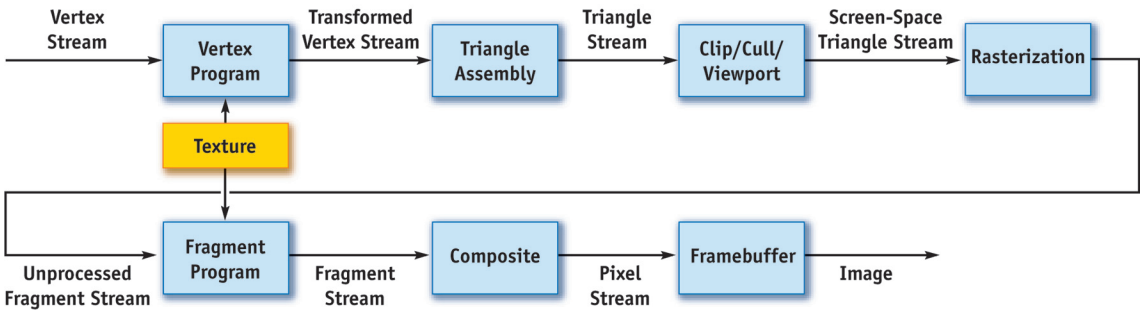


Figure 29-3. Mapping the Graphics Pipeline to the Stream Model
The stream formulation of the graphics pipeline expresses all data as streams (indicated by arrows) and all computation as kernels (indicated by blue boxes). Both user-programmable and nonprogrammable stages in the graphics pipeline can be expressed as kernels.

Efficient Computation

The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with many elements allow this data-level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction-level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism.

Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model.

Finally, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on each input element) permits hardware implementations to devote most of their transistors to datapath hardware rather than control hardware.

Efficient Communication

Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempt to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data references within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency.

29.3.2 Building a Stream Processor

The stream programming model structures programs in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High-performance graphics hardware must effec-

tively exploit the high arithmetic performance and the efficient computation exposed by the stream model. How do we structure a hardware implementation of a GPU to ensure the highest overall performance?

The first step to building a high-performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as *task parallel*, which permits not only task-level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task-parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they can communicate effectively without requiring global memory access.

Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task-level and data-level parallelism allows GPUs to profitably use dozens of functional units simultaneously.

Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high-latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high-latency operations.

For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed-function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today's GPUs feature high-performance data-parallel processors that implement two kernels in the graphics pipeline: a *vertex program* that allows users to run a program on each vertex that passes through the pipeline, and a *fragment program* that allows users to run a program on each fragment. Both of these stages permit single-precision floating-point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates on user-specified programs with sufficient precision to address general-purpose computing problems has effectively made them *programmable stream processors*—that is, processors that are attractive for a much wider variety of applications than simply the graphics pipeline.

29.4 The Future and Challenges

The migration of GPUs into programmable stream processors reflects the culmination of several historical trends. The first trend is the ability to concentrate large amounts of computation on a single processor die. Equally important has been the ability and talent of GPU designers in effectively using these computation resources. The economies of scale that are associated with building tens of millions of processors per year have allowed the cost of a GPU to fall enough to make a GPU a standard part of today's desktop computer. And the addition of reasonably high-precision programmability to the pipeline has completed the transition from a hard-wired, special-purpose processor to a powerful programmable processor that can address a wide variety of tasks.

What, then, can we expect from future GPU development?

29.4.1 Challenge: Technology Trends

Each new generation of hardware will present a challenge to GPU vendors: How can they effectively use additional hardware resources to increase performance and functionality? New transistors will be devoted to increased performance, in large part through greater amounts of parallelism, and to new functionality in the pipeline. We will also see these architectures evolve with changes in technology.

As we described in Section 29.1.2, future architectures will increasingly use transistors to replace the need for communication. We can expect more aggressive caching techniques that not only alleviate off-chip communication but also mitigate the need for some on-chip communication. We will also see computation increasingly replace communication when appropriate. For example, the use of texture memory as a lookup table may be replaced by calculating the values in that lookup table dynamically. And instead of sending data to a distant on-chip computation resource and then sending the result back, we may simply replicate the resource and compute our result locally. In the trade-off between *communicate* and *recompute/cache*, we will increasingly choose the latter.

The increasing cost of communication will also influence the microarchitecture of future chips. Designers must now explicitly plan for the time required to send data across a chip; even local communication times are becoming significant in a timing budget.

29.4.2 Challenge: Power Management

Ideas for how to use future GPU transistors must be tempered by the realities of their costs. Power management has become a critical piece of today's GPU designs as each

generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge.

29.4.3 Challenge: Supporting More Programmability and Functionality

While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels.

As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality.

Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs.

29.4.4 Challenge: GPU Functionality Subsumed by CPU (or Vice Versa)?

We can be confident that CPU vendors will not stand still as GPUs incorporate more processing power and more capability onto their future chips. The ever-increasing number of transistors with each process generation may eventually lead to conflict between CPU and GPU manufacturers. Is the core of future computer systems the CPU, one that may eventually incorporate GPU or stream functionality on the CPU itself? Or will future systems contain a GPU at their heart with CPU functionality incorporated into the GPU? Such weighty questions will challenge the next generation of processor architects as we look toward an exciting future.

29.5 References

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

ITRS. 2003. *International Technology Roadmap for Semiconductors*. **http://public.itrs.net**

Kapasi, Ujval J., Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. 2003. "Programmable Stream Processors." *IEEE Computer*, pp. 54–62.

Moore, Gordon. 1965. "Cramming More Components onto Integrated Circuits." *Electronics* 38(8). More information is available at **http://www.intel.com/research/silicon/mooreslaw.htm**

Owens, John D. 2002. "Computer Graphics on a Stream Architecture." Ph.D. Thesis, Stanford University, November 2002.

Patterson, David A. 2004. "Latency Lags Bandwidth." *Communications of the ACM* 47(10), pp. 71–75.