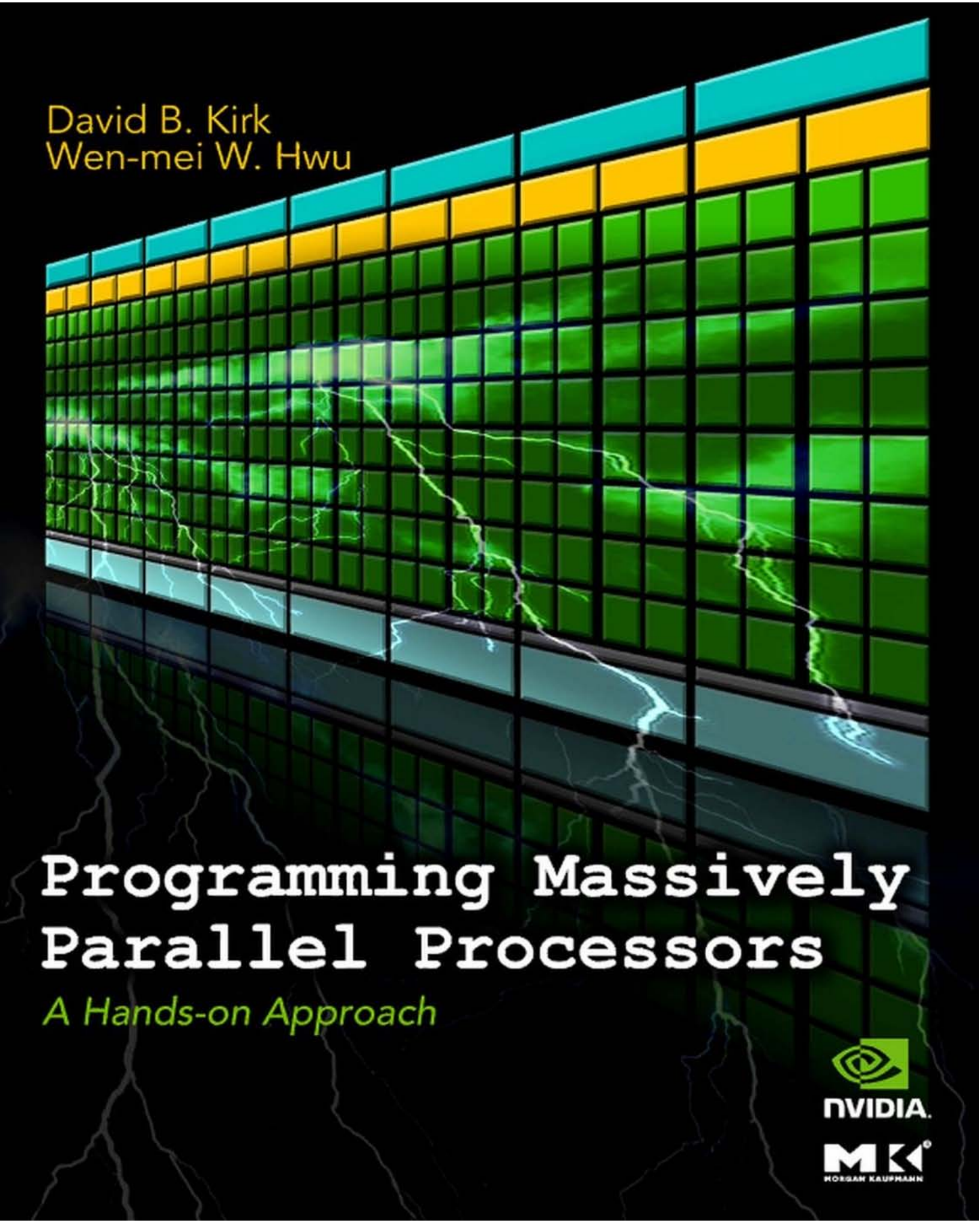
David B. Kirk
Wen-mei W. Hwu
Programming Massively Parallel Processors
A Hands-on Approach
NVIDIA
MK
MORGAN KAUFMANN

CHAPTER

History of GPU Computing 2

CHAPTER CONTENTS

INTRODUCTION

To CUDA™ and OpenCL™ programmers, graphics processing units (GPUs) are massively parallel numeric computing processors programmed in C with extensions. One needs not understand graphics algorithms or terminology in order to be able to program these processors. However, understanding the graphics heritage of these processors illuminates the strengths and weaknesses of these processors with respect to major computational patterns. In particular, the history helps to clarify the rationale behind major architectural design decisions of modern programmable GPUs: massive multithreading, relatively small cache memories compared to central processing units (CPUs), and bandwidth-centric memory interface design. Insights into the historical developments will also likely give the reader the context needed to project the future evolution of GPUs as computing devices.

2.1 EVOLUTION OF GRAPHICS PIPELINES

Three-dimensional (3D) graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid- to late 1990s. During this period, the performance-

leading graphics subsystems decreased in price from $50,000 to $200. During the same period, the performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second. Although these advancements have much to do with the relentlessly shrinking feature sizes of semiconductor devices, they also have resulted from innovations in graphics algorithms and hardware design that have shaped the native hardware capabilities of modern GPUs.

The remarkable advancement of graphics hardware performance has been driven by the market demand for high-quality, real-time graphics in computer applications. In an electronic gaming application, for example, one needs to render ever more complex scenes at an ever-increasing resolution at a rate of 60 frames per second. The net result is that over the last 30 years graphics architecture has evolved from being a simple pipeline for drawing wire-frame diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes. Concurrently, many of the hardware functionalities involved became far more sophisticated and user programmable.

2.1.1 The Era of Fixed-Function Graphics Pipelines

From the early 1980s to the late 1990s, the leading performance graphics hardware was fixed-function pipelines that were configurable but not programmable. In that same era, major graphics application programming interface (API) libraries became popular. An API is a standardized layer of software (i.e., a collection of library functions) that allows applications (such as games) to use software or hardware services and functionality. An API, for example, can allow a game to send commands to a graphics processing unit to draw objects on a display. One such API is DirectX™, Microsoft's proprietary API for media functionality. The Direct3D® component of DirectX™ provides interface functions to graphics processors. The other major API is OpenGL®, an open standard API supported by multiple vendors and popular in professional workstation applications. This era of fixed-function graphics pipeline roughly corresponds to the first seven generations of DirectX™.

Figure 2.1 shows an example of fixed-function graphics pipeline in early NVIDIA® GeForce® GPUs. The host interface receives graphics commands and data from the CPU. The commands are typically given by application programs by calling an API function. The host interface typically contains a specialized direct memory access (DMA) hardware to efficiently transfer

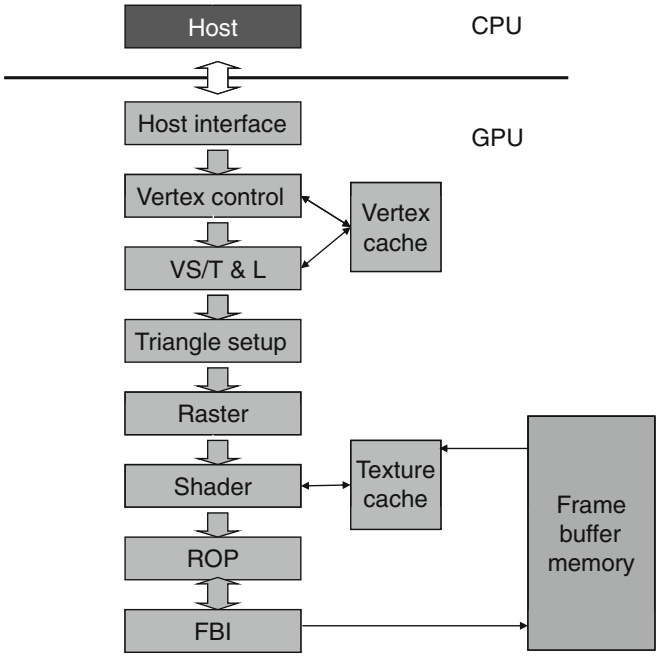

FIGURE 2.1

A fixed-function NVIDIA GeForce graphics pipeline.

bulk data to and from the host system memory to the graphics pipeline. The host interface also communicates back the status and result data of executing the commands.

Before we describe the other stages of the pipeline, we should clarify that the term *vertex* usually refers to the corner of a polygon. The GeForce graphics pipeline is designed to render triangles, so the term *vertex* is typically used in this case to refer to the corners of a triangle. The surface of an object is drawn as a collection of triangles. The finer the sizes of the triangles are, the better the quality of the picture typically becomes. The vertex control stage in Figure 2.1 receives parameterized triangle data from the CPU. The vertex control stage then converts the triangle data into a form that the hardware understands and places the prepared data into the vertex cache.

The vertex shading, transform, and lighting (VS/T&L) stage in Figure 2.1 transforms vertices and assigns per-vertex values (e.g., colors, normals, texture coordinates, tangents). The shading is done by the pixel shader hardware. The vertex shader can assign a color to each vertex, but color is not applied to triangle pixels until later. The triangle setup stage further creates

edge equations that are used to interpolate colors and other per-vertex data (such as texture coordinates) across the pixels touched by the triangle. The raster stage determines which pixels are contained in each triangle. For each of these pixels, the raster stage interpolates per-vertex values necessary for shading the pixel, including the color, position, and texture position that will be shaded (painted) on the pixel.

The shader stage in Figure 2.1 determines the final color of each pixel. This can be generated as a combined effect of many techniques: interpolation of vertex colors, texture mapping, per-pixel lighting mathematics, reflections, and more. Many effects that make the rendered images more realistic are incorporated in the shader stage. Figure 2.2 illustrates texture mapping, one of the shader stage functionalities. It shows an example in which a world map texture is mapped onto a sphere object. Note that the sphere object is described as a large collection of triangles. Although the shader stage must perform only a small number of coordinate transform

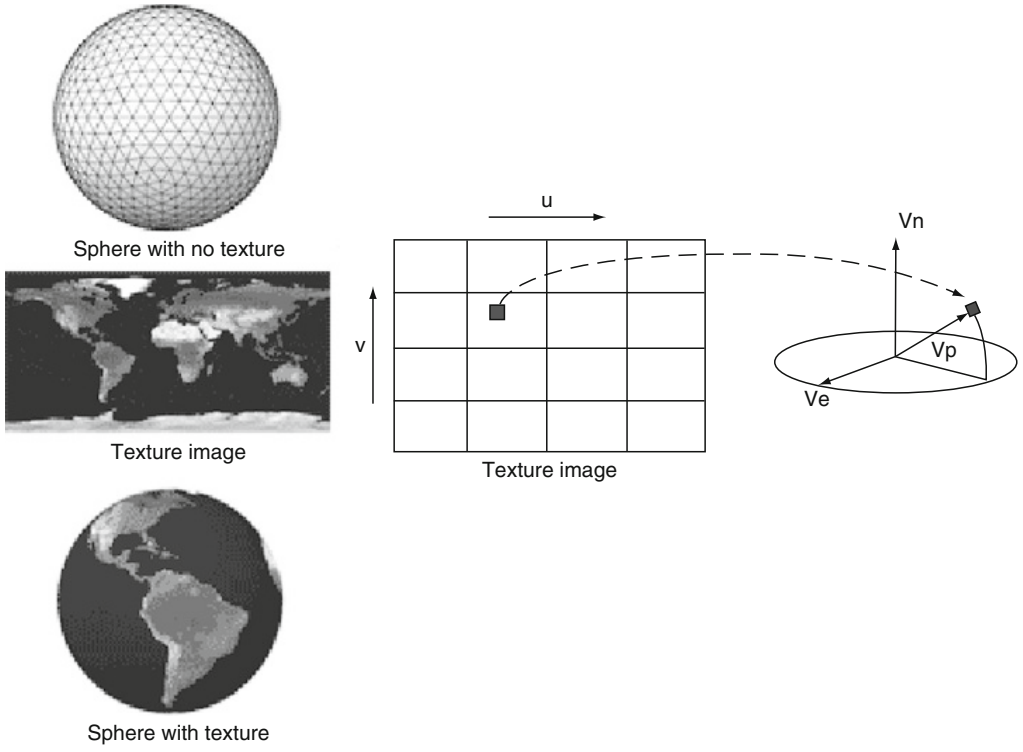

FIGURE 2.2

Texture mapping example: painting a world map texture image onto a globe object.

calculations to identify the exact coordinates of the texture point that will be painted on a point in one of the triangles that describes the sphere object, the sheer number of pixels covered by the image requires the shader stage to perform a very large number of coordinate transforms for each frame.

The raster operation (ROP) stage in Figure 2.2 performs the final raster operations on the pixels. It performs color raster operations that blend the color of overlapping/adjacent objects for transparency and antialiasing effects. It also determines the visible objects for a given viewpoint and discards the occluded pixels. A pixel becomes occluded when it is blocked by pixels from other objects according to the given view point.

Figure 2.3 illustrates antialiasing, one of the ROP stage operations. Notice the three adjacent triangles with a black background. In the aliased output, each pixel assumes the color of one of the objects or the background. The limited resolution makes the edges look crooked and the shapes of the objects distorted. The problem is that many pixels are partly in one object and partly in another object or the background. Forcing these pixels to assume the color of one of the objects introduces distortion into the edges of the objects. The antialiasing operation gives each pixel a color that is blended, or linearly combined, from the colors of all the objects and background that partially overlap the pixel. The contribution of each object to the color of the pixel is the amount of the pixel that the object overlaps.

Finally, the frame buffer interface (FBI) stage in Figure 2.1 manages memory reads from and writes to the display frame buffer memory. For high-resolution displays, there is a very high bandwidth requirement in

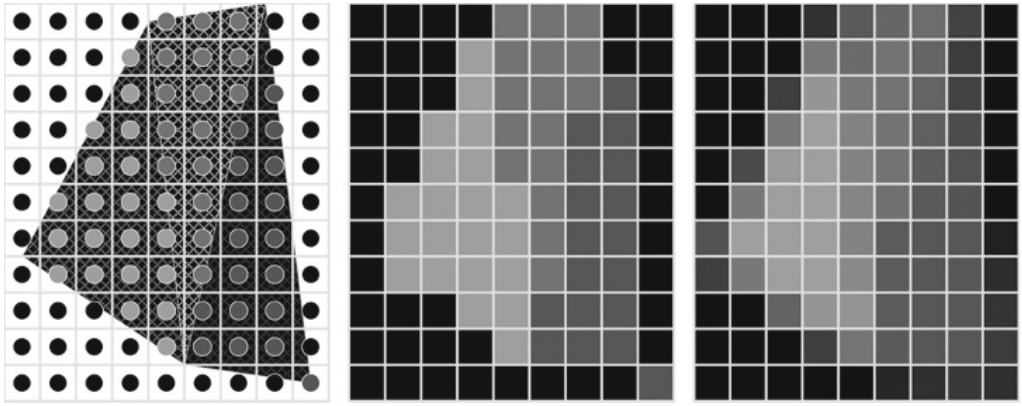

FIGURE 2.3

Example of antialiasing operations.

accessing the frame buffer. Such bandwidth is achieved by two strategies. One is that graphics pipelines typically use special memory designs that provide higher bandwidth than the system memories. Second, the FBI simultaneously manages multiple memory channels that connect to multiple memory banks. The combined bandwidth improvement of multiple channels and special memory structures gives the frame buffers much higher bandwidth than their contemporaneous system memories. Such high memory bandwidth has continued to this day and has become a distinguishing feature of modern GPU design.

For two decades, each generation of hardware and its corresponding generation of API brought incremental improvements to the various stages of the graphics pipeline. Although each generation introduced additional hardware resources and configurability to the pipeline stages, developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The obvious next step was to make some of these graphics pipeline stages into programmable processors.

2.1.2 Evolution of Programmable Real-Time Graphics

In 2001, the NVIDIA GeForce 3 took the first step toward achieving true general shader programmability. It exposed the application developer to what had been the private internal instruction set of the floating-point vertex engine (VS/T&L stage). This coincided with the release of Microsoft's DirectX 8 and OpenGL vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating-point capability to the pixel shader stage and made texture accessible from the vertex shader stage. The ATI Radeon™ 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel shader processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. These programmable pixel shader processors were part of a general trend toward unifying the functionality of the different stages as seen by the application programmer. The GeForce 6800 and 7800 series were built with separate processor designs dedicated to vertex and pixel processing. The XBox® 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

In graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the positions of triangle vertices or generating pixel colors. This *data independence* as the dominating application characteristic is a key difference between the design

assumption for GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have a million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms. Such variation has motivated the hardware designers to make those pipeline stages programmable. Two particular programmable stages stand out: the vertex shader and the pixel shader. Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation. Typically, a vertex shader thread reads a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Vertex shader programs and geometry shader programs execute on the vertex shader (VS/T&L) stage of the graphics pipeline.

A shader program calculates the floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. These programs execute on the shader stage of the graphics pipeline. For all three types of graphics shader programs, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects. This property has motivated the design of the programmable pipeline stages into massively parallel processors.

Figure 2.4 shows an example of a programmable pipeline that employs a vertex processor and a fragment (pixel) processor. The programmable vertex processor executes the programs designated to the vertex shader stage, and the programmable fragment processor executes the programs designated to the (pixel) shader stage. Between these programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability. For example, between the vertex processing stage and the pixel (fragment) processing stage is a *rasterizer* (rasterization and interpolation), a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. Together, the mix of programmable and fixed-function stages is engineered to balance extreme performance with user control over the rendering algorithms.

Common rendering algorithms perform a single pass over input primitives and access other memory resources in a highly coherent manner. That is, these algorithms tend to simultaneously access contiguous memory locations, such as all triangles or all pixels in a neighborhood. As a result, these algorithms exhibit excellent efficiency in memory bandwidth utilization

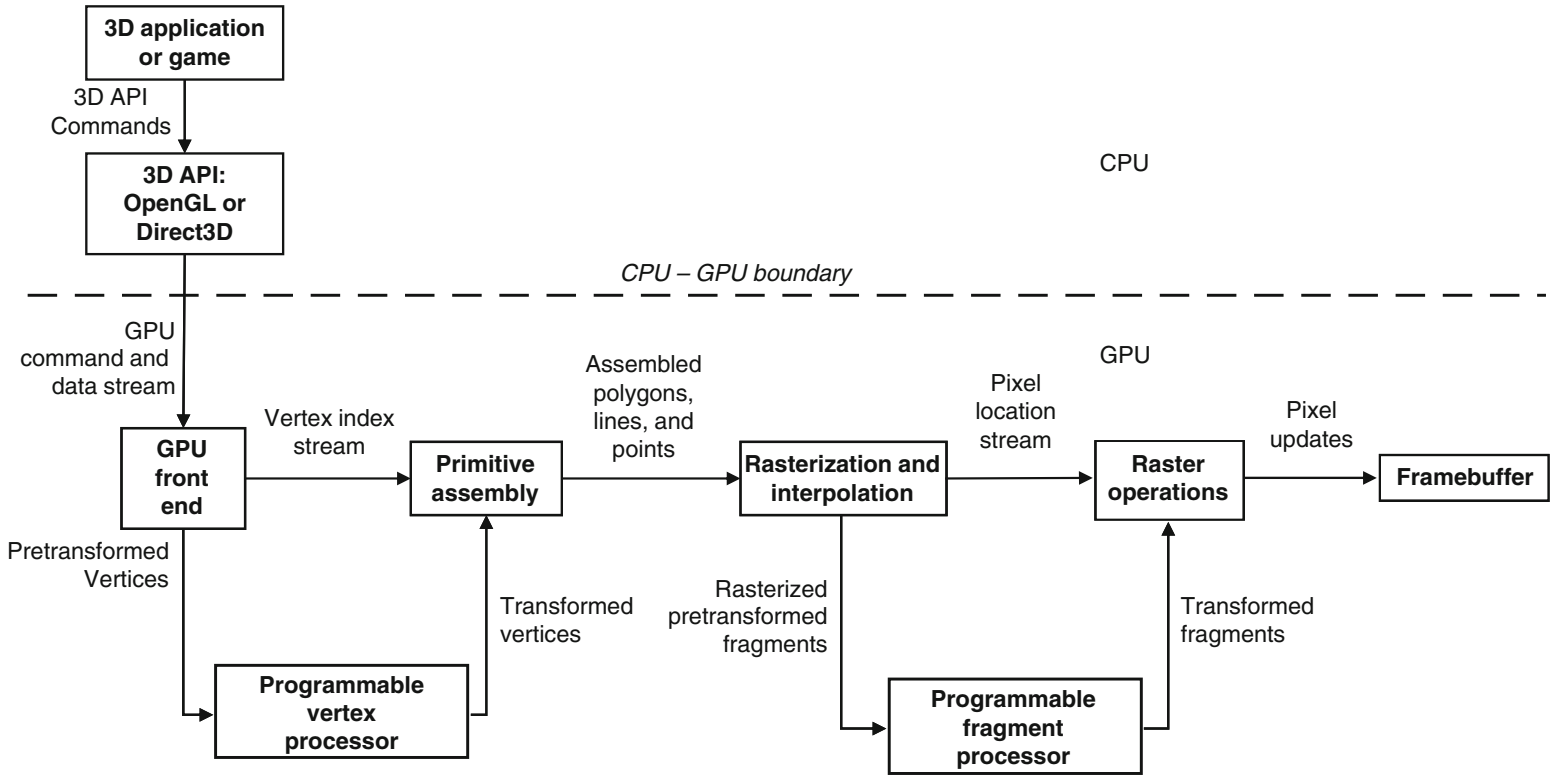

FIGURE 2.4

Example of a separate vertex processor and fragment processor in a programmable graphics pipeline.

and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. In particular, whereas the CPU die area is dominated by cache memories, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (as latency can be readily hidden by massively parallel execution); indeed, bandwidth is typically many times higher than that for a CPU, exceeding 100 GB/s in more recent designs.

2.1.3 Unified Graphics and Computing Processors

Introduced in 2006, the GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors; the logical graphics pipeline is physically a recirculating path that visits these processors three times, with much fixed-function graphics logic between visits. This is illustrated in Figure 2.5. The unified processor array allows dynamic partitioning of the array to vertex shading, geometry processing, and pixel processing. Because different rendering algorithms present wildly different loads among the three programmable stages, this unification allows the same pool of execution resources to be dynamically allocated to different pipeline stages and achieve better load balance.

The GeForce 8800 hardware corresponds to the DirectX 10 API generation. By the DirectX 10 generation, the functionality of vertex and pixel shaders had been made identical to the programmer, and a new logical stage was introduced, the geometry shader, to process all the vertices of a primitive rather than vertices in isolation. The GeForce 8800 was designed with DirectX 10 in mind. Developers were coming up with more sophisticated shading algorithms, and this motivated a sharp increase in the available shader operation rate, particularly floating-point operations. NVIDIA pursued a processor design with higher operating clock frequency than what was allowed by standard-cell methodologies in order to deliver the desired operation throughput as area efficiently as possible. High-clock-speed design requires substantially greater engineering effort, thus favoring the design of one processor array rather than two (or three, given the new geometry stage). It became worthwhile to take on the engineering challenges of a unified processor—load balancing and recirculation of a logical pipeline onto threads of the processor array—while seeking the benefits of one processor design. Such design paved the way for using the programmable GPU processor array for general numeric computing.

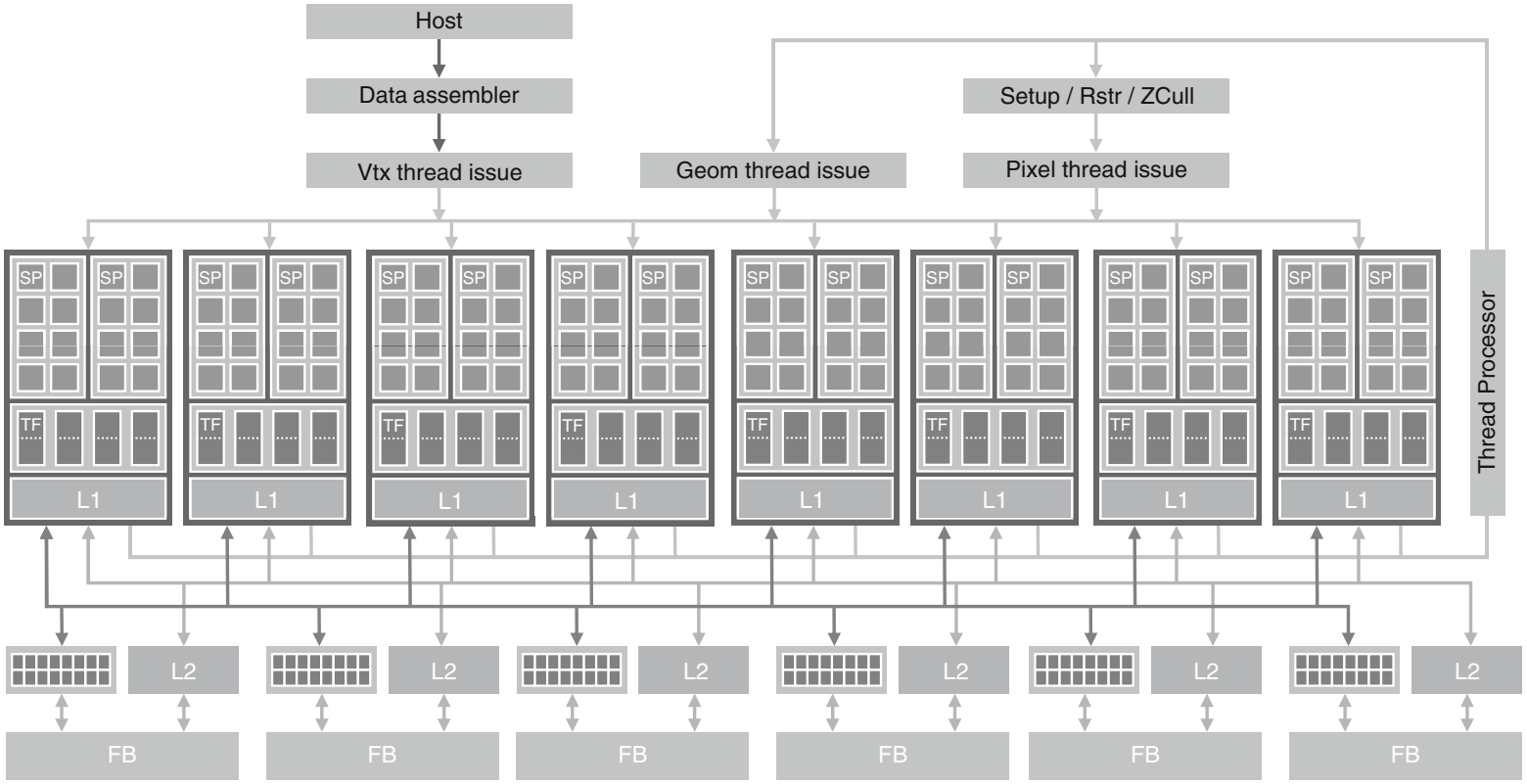

FIGURE 2.5

Unified programmable processor array of the GeForce 8800 GTX graphics pipeline.

2.1.4 GPGPU: An Intermediate Step

While the GPU hardware designs evolved toward more unified processors, they increasingly resembled high-performance parallel computers. As DirectX 9-capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and began to explore the use of GPUs to solve compute-intensive science and engineering problems; however, DirectX 9 GPUs had been designed only to match the features required by the graphics APIs. To access the computational resources, a programmer had to cast his or her problem into native graphics operations so the computation could be launched through OpenGL or DirectX API calls. To run many simultaneous instances of a compute function, for example, the computation had to be written as a pixel shader. The collection of input data had to be stored in texture images and issued to the GPU by submitting triangles (with clipping to a rectangle shape if that was what was desired). The output had to be cast as a set of pixels generated from the raster operations.

The fact that the GPU processor array and frame buffer memory interface were designed to process graphics data proved too restrictive for general numeric applications. In particular, the output data of the shader programs are single pixels whose memory locations have been predetermined; thus, the graphics processor array is designed with very restricted memory reading and writing capability. Figure 2.6 illustrates the limited

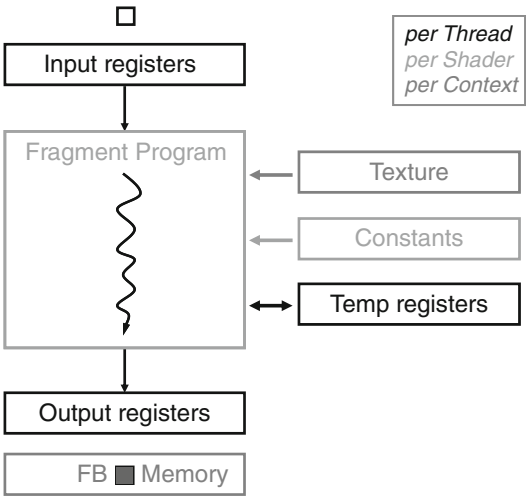

FIGURE 2.6

The restricted input and output capabilities of a shader programming model.

memory access capability of early programmable shader processor arrays; shader programmers needed to use texture to access arbitrary memory locations for their input data. More importantly, shaders did not have the means to perform writes with calculated memory addresses, referred to as *scatter operations*, to memory. The only way to write a result to memory was to emit it as a pixel color value, and configure the frame buffer operation stage to write (or blend, if desired) the result to a two-dimensional frame buffer.

Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel frame buffer, then use that frame buffer as a texture map input to the pixel fragment shader of the next stage of the computation. There was also no user-defined data types; most data had to be stored in one-, two-, or four-component vector arrays. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU," for general-purpose computing on GPUs.

2.2 GPU COMPUTING

While developing the Tesla™ GPU architecture, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU like a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX™ 10 generation of graphics, NVIDIA had already begun work on a high-efficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. The designers of the Tesla GPU architecture took another step. The shader processors became fully programmable processors with large instruction memory, instruction cache, and instruction sequencing control logic. The cost of these additional hardware resources was reduced by having multiple shader processors to share their instruction cache and instruction sequencing control logic. This design style works well with graphics applications because the same shader program needs to be applied to a massive number of vertices or pixels. NVIDIA added memory load and store instructions with random byte addressing capability to support the requirements of compiled C programs. To nongraphics application programmers, the Tesla GPU architecture introduced a more generic parallel programming model with a hierarchy of parallel threads, barrier synchronization, and atomic

operations to dispatch and manage highly parallel computing work. NVIDIA also developed the CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications. Programmers no longer need to use the graphics API to access the GPU parallel computing capabilities. The G80 chip was based on the Tesla architecture and was used in the GeForce 8800 GTX, which was followed later by G92 and GT200.

2.2.1 Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. In the early days, workstation graphics systems gave customers a choice in pixel horsepower by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s, PC graphics scaling was almost nonexistent. There was one option: the VGA controller. As 3D-capable accelerators began to appear, there was room in the market for a range of offerings. In 1998, 3dfx introduced multiboard scaling with their original Scan Line Interleave (SLI) on their Voodoo2, which held the performance crown for its time. Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high-performance) and Vanta (low-cost), first by speed binning and packaging, then with separate chip designs (GeForce 2 GTS and GeForce 2 MX). At present, for a given architecture generation, four or five separate chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx in 2001, NVIDIA continued the multi-GPU SLI concept; for example, the GeForce 6800 provides multi-GPU scalability transparently to both the programmer and the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

By switching to the multicore trajectory, CPUs are scaling to higher transistor counts by increasing the number of nearly-constant-performance cores on a die rather than simply increasing the performance of a single core. At this writing, the industry is transitioning from quad-core to hex- and oct-core CPUs. Programmers are forced to find four- to eight-fold parallelism to fully utilize these processors. Many of them resort to coarse-grained parallelism strategies where different tasks of an application are performed in parallel. Such applications must be rewritten often to have more parallel tasks for each successive doubling of core count. In contrast, the highly multithreaded GPUs encourage the use of massive, fine-grained

data parallelism in CUDA. Efficient threading support in GPUs allows applications to expose a much larger amount of parallelism than available hardware execution resources with little or no penalty. Each doubling of GPU core count provides more hardware execution resources that exploit more of the exposed parallelism for higher performance; that is, the GPU parallel programming model for graphics and parallel computing is designed for transparent and portable scalability. A graphics program or CUDA program is written once and runs on a GPU with any number of processor cores.

2.2.2 Recent Developments

Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these programs run the application tens or hundreds of times faster than multicore CPUs are capable of running them. With the introduction of tools such as MCUDA [Stratton 2008], the parallel threads of a CUDA program can also run efficiently on a multicore CPU, although at a lower speed than on GPUs due to lower levels of floating-point execution resources. Examples of these applications include *n*-body simulation, molecular modeling, computational finance, and oil/gas reservoir simulation. Although many of these use single-precision floating-point arithmetic, some problems require double precision. The arrival of double-precision floating point in GPUs enabled an even broader range of applications to benefit from GPU acceleration.

For an exhaustive list and examples of current developments in applications that are accelerated by GPUs, visit CUDA Zone at http://www.nvidia.com/CUDA. For resources in developing research applications, see CUDA Research at http://www.cuda-research.org.

2.3 FUTURE TRENDS

Naturally, the number of processor cores will continue to increase in proportion to increases in available transistors as silicon processes improve. In addition, GPUs will continue to enjoy vigorous architectural evolution. Despite their demonstrated high performance on data parallel applications, GPU core processors are still of relatively simple design. More aggressive techniques will be introduced with each successive architecture to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is still a young field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations. Chapter 12 provides more details of such future trends.