Historical Perspective and Further Reading

Graphics Pipeline Evolution

3D graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then to PC accelerators in the mid- to late 1990s. During this period, three major transitions occurred:

- Performance-leading graphics subsystems declined in price from $50,000 to $200.
- 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.
- Native hardware capabilities evolved from wireframe (polygon outlines) to flat shaded (constant color) filled polygons, to smooth shaded (interpolated color) filled polygons, to full-scene anti-aliasing with texture mapping and rudimentary multitexturing.

Fixed-Function Graphics Pipelines

Throughout this period, graphics hardware was configurable, but not programmable by the application developer. With each generation, incremental improvements were offered. But developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The NVIDIA GeForce 3, described by Lindholm, et al. [2001], took the first step toward true general shader programmability. It exposed to the application developer what had been the private internal instruction set of the floating-point vertex engine. This coincided with the release of Microsoft's DirectX 8 and OpenGL's vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating point capability to the pixel fragment stage, and made texture available at the vertex stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel fragment processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. This was part of a general trend toward unifying the functionality of the different stages, at least as far as the application programmer was concerned. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs and separate hardware dedicated to the vertex and to the fragment processing. The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

Evolution of Programmable Real-Time Graphics

During the last 30 years, graphics architecture has evolved from a simple pipeline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery that appears three-dimensional. Concurrently, many of the calculations involved became far more sophisticated and user programmable.

In these graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the position of triangle vertexes or generating pixel colors. This data independence is a key difference between GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have 1 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 and have evolved to be programmable. Vertex programs map the position of triangle vertices on to the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. For all three types of graphics shaders, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

Between these programmable graphics pipeline stages are dozens of fixed-function stages which perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," 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; these algorithms provide excellent bandwidth utilization 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. Whereas CPU die area is dominated by cache memory, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (since latency can be readily hidden by a high thread count); indeed, bandwidth is typically many times higher than a CPU, exceeding 100 GB/second in some cases. The far-higher number of fine-grained lightweight threads effectively exploits the rich parallelism available.

Beginning with NVIDIA's GeForce 8800 GPU in 2006, the three programmable graphics stages are mapped 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. Since different rendering algorithms present wildly different loads among the three programmable stages, this unification provides processor load balancing.

Unified Graphics and Computing Processors

By the DirectX 10 generation, the functionality of vertex and pixel fragment shaders was to be made identical to the programmer, and in fact 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 chose to pursue a processor design with higher operating frequency than standard-cell methodologies had allowed to deliver the desired operation throughput as area-efficiently as possible. High-clock-speed design requires substantially more engineering effort, and this favored designing one processor, 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) to get the benefits of one processor design.

GPGPU: an Intermediate Step

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 complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations. For example, to run many simultaneous instances of a pixel shader, a triangle had to be issued to the GPU (with clipping to a rectangle shape if that's what was desired). Shaders did not have the means to perform arbitrary 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 framebuffer operation stage to write (or blend, if desired) the result to a two-dimensional framebuffer. 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 framebuffer, then use that framebuffer as a texture map as input to the pixel fragment shader of the next stage of the computation. 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.

GPU Computing

While developing the Tesla architecture for the GeForce 8800, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU as 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, 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. This processor was designed to take advantage of the common case of groups of threads executing the same code path. NVIDIA added memory load and store instructions with integer byte addressing to support the requirements of compiled C programs. It introduced the thread block (cooperative thread array), grid of thread blocks, and barrier synchronization to dispatch and manage highly parallel computing work. Atomic memory operations were added. NVIDIA 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.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. 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 appeared, the market had room for a range of offerings. 3dfx introduced multiboard scaling with the original SLI (Scan Line Interleave) on their Voodoo2, which held the performance crown for its time (1998). 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 & GeForce 2 MX). At present, for a given architecture generation, four or five separate GPU 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, NVIDIA continued the multi-GPU SLI concept in 2004, starting with GeForce 6800—providing multi-GPU scalability transparently to the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

CPUs are scaling to higher transistor counts by increasing the number of constant-performance cores on a die, rather than increasing the performance of a single core. At this writing the industry is transitioning from dual-core to quad-core, with eight-core not far behind. Programmers are forced to find fourfold to eightfold task parallelism to fully utilize these processors, and applications using task parallelism must be rewritten frequently to target each successive doubling

of core count. In contrast, the highly multithreaded GPU encourages the use of many-fold data parallelism and thread parallelism, which readily scales to thousands of parallel threads on many processors. The GPU scalable 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 processors. As shown in Section A.3, a CUDA programmer explicitly states both fine-grained and coarse-grained parallelism in a thread program by decomposing the problem into grids of thread blocks—the same program will run efficiently on GPUs or CPUs of any size in current and future generations as well.

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. Examples include n-body simulation, molecular modeling, computational finance, and oil and gas exploration data processing. Although many of these use single precision floating-point arithmetic, some problems require double precision. The recent arrival of double precision floating point in GPUs enables an even broader range of applications to benefit from GPU acceleration.

For a comprehensive list and examples of current developments in applications that are accelerated by GPUs, visit CUDAZone: www.nvidia.com/CUDA.

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 a new field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations.

Further Reading

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. SIGGRAPH 1988* (August), 239–46.

Akeley, K. [1993]. "RealityEngine Graphics." *Proc. SIGGRAPH 1993* (August), 109–16.

Blelloch, G. B. [1990]. "Prefix Sums and Their Applications." In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Francisco.

Blythe, D. [2006]. "The Direct3D 10 System," *ACM Trans. Graphics*, Vol. 25, no. 3 (July), 724–34.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahlian, M. Houston, and P. Hanrahan [2004]. "Brook for GPUs: Stream Computing on Graphics Hardware." *Proc. SIGGRAPH 2004*, 777–86, August. http://doi.acm.org/10.1145/1186562.1015800

Elder, G. [2002] "Radeon 9700." Eurographics/SIGGRAPH Workshop on Graphics Hardware, Hot3D Session, www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt

Fernando, R. and M. J. Kilgard [2003]. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, Reading, MA.

Fernando, R. ed. [2004]. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley, Reading, MA. http://developer.nvidia.com/object/gpu_gems_home.html.

Foley, J., A. van Dam, S. Feiner, and J. Hughes [1995]. *Computer Graphics: Principles and Practice, second edition in C*, Addison-Wesley, Reading, MA.

Hillis, W. D. and G. L. Steele [1986]. "Data parallel algorithms." *Commun. ACM* 29, 12 (Dec.), 1170–83. http://doi.acm.org/10.1145/7902.7903.

IEEE Std 754-2008 [2008]. *IEEE Standard for Floating-Point Arithmetic*. ISBN 978-0-7381-5752-8, STD95802, http://ieeexplore.ieee.org/servlet/opac?punumber=4610933 (Aug. 29).

Industrial Light and Magic [2003]. *OpenEXR*, www.openexr.com.

Intel Corporation [2007]. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November. Order Number: 248966-016. Also: www3.intel.com/design/processor/manuals/248966.pdf.

Kessenich, J. [2006]. *The OpenGL Shading Language, Language Version 1.20, Sept. 2006*. www.opengl.org/documentation/specs/.

Kirk, D. and D. Voorhies [1990]. "The Rendering Architecture of the DN10000VS." *Proc. SIGGRAPH 1990* (August), 299–307.

Lindholm E., M. J. Kilgard, and H. Moreton [2001]. "A User-Programmable Vertex Engine." *Proc. SIGGRAPH 2001* (August), 149–58.

Lindholm E., J. Nickolls, S. Oberman, and J. Montrym [2008]. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." *IEEE Micro,* Vol. 28, no. 2 (March–April), 39–55.

Microsoft Corporation. Microsoft DirectX Specification, http://msdn.microsoft.com/directx/

Microsoft Corporation. [2003]. Microsoft DirectX 9 Programmable Graphics Pipeline, Microsoft Press, Redmond, WA.

Montrym, J., D. Baum, D. Dignam, and C. Migdal [1997]. "InfiniteReality: A Real-Time Graphics System." *Proc. SIGGRAPH 1997* (August), 293–301 .

Montrym, J. and H. Moreton [2005]. "The GeForce 6800," *IEEE Micro*, Vol. 25, no. 2 (March–April), 41–51.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits," *Electronics*, Vol. 38, no. 8 (April 19).

Nguyen, H., ed. [2008]. *GPU Gems 3*, Addison-Wesley, Reading, MA.

Nickolls, J., I. Buck, M. Garland, and K. Skadron [2008]. "Scalable Parallel Programming with CUDA," *ACM Queue*, Vol. 6, no. 2 (March–April) 40–53.

NVIDIA [2007]. CUDA Zone. www.nvidia.com/CUDA.

NVIDIA [2007]. *CUDA Programming Guide 1.1*. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

NVIDIA [2007]. *PTX: Parallel Thread Execution ISA version 1.1*. www.nvidia.com/object/io_1195170102263.html.

Nyland, L., M. Harris, and J. Prins [2007]. "Fast N-Body Simulation with CUDA." In *GPU Gems 3*, H. Nguyen (Ed.), Addison-Wesley, Reading, MA.

Oberman, S. F. and M. Y. Siu [2005]. "A High-Performance Area-Efficient Multifunction Interpolator," *Proc. Seventeenth IEEE Symp. Computer Arithmetic*, 272–79.

Patterson, D. A. and J. L. Hennessy [2004]. *Computer Organization and Design: The Hardware/Software Interface*, third edition, Morgan Kaufmann Publishers, San Francisco.

Pharr, M. ed. [2005]. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA.

Satish, N., M. Harris, and M. Garland [2008]. "Designing Efficient Sorting Algorithms for Manycore GPUs," NVIDIA Technical Report NVR-2008-001.

Segal, M. and K. Akeley [2006]. *The OpenGL Graphics System: A Specification, Version 2.1, Dec. 1, 2006*. www.opengl.org/documentation/specs/.

Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens [2007]. "Scan Primitives for GPU Computing." In *Proc. of Graphics Hardware 2007* (August), 97–106.

Volkov, V. and J. Demmel [2008]. "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Technical Report No. UCB/EECS-2008-49, 1–11. www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," In *Proc. Supercomputing 2007*, November.