INVITED PAPER

Rise of the Graphics Processor

Programmable graphics processors can be used for applications such as image and signal processing, linear algebra, engineering analysis, physical simulation, database management, financial services, and molecular biology.

By David Blythe, *Member IEEE*

ABSTRACT | The modern graphics processing unit (GPU) is the result of 40 years of evolution of hardware to accelerate graphics processing operations. It represents the convergence of support for multiple market segments: computer-aided design, medical imaging, digital content creation, document and presentation applications, and entertainment applications. The exceptional performance characteristics of the GPU make it an attractive target for other application domains. We examine some of this evolution, look at the structure of a modern GPU, and discuss how graphics processing exploits this structure and how nongraphical applications can take advantage of this capability. We discuss some of the technical and market issues around broader adoption of this technology.

KEYWORDS | Computer architecture; computer graphics; parallel processing

I. INTRODUCTION

Over the past 40 years, dedicated graphics processors have made their way from research labs and flight simulators to commercial workstations and medical devices and later to personal computers and entertainment consoles. The most recent wave has been to cell phones and automobiles. As the number of transistors in the devices has begun to exceed those found in CPUs, attention has focused on applying the processing power to computationally intensive problems beyond traditional graphics rendering.

In this paper, we look at the evolution of the architecture and programming model for these devices. We discuss how the architectures are effective at solving the graphics rendering problem, how they can be exploited for other types of problems, and what enhancements may be necessary to broaden their applicability without compromising their effectiveness.

Manuscript received July 2, 2007; revised October 31, 2007.
The author is with Microsoft Corp., Redmond, WA 98052-6399 USA (e-mail: dblythe@microsoft.com).
Digital Object Identifier: 10.1109/JPROC.2008.917718

Fig. 1. *A synthesized image of a scene composed of lighted, shaded objects.*

A. Graphics Processing

Graphics processors are employed to accelerate a variety of tasks ranging from drawing the text and graphics in an internet web browser to more sophisticated synthesis of three-dimensional (3-D) imagery in computer games. We will briefly describe the nature of processing necessary for the 3-D image synthesis fundamental to many of the application areas. Other applications of graphics processing use a subset of this 3-D processing capability. For brevity, we use a simplified description of a contemporary image synthesis pipeline that provides enough detail to inform a discussion about the processing characteristics of graphics accelerators. More detailed descriptions of the image synthesis process can be found in Haines [1] and Montrym [2].

An image, such as shown in Fig. 1, is synthesized from a model consisting of geometric shape and appearance descriptions (color, surface texture, etc.) for each object in the scene, and environment descriptions such as lighting, atmospheric properties, vantage point, etc. The result of the synthesis is an image represented as a two-dimensional

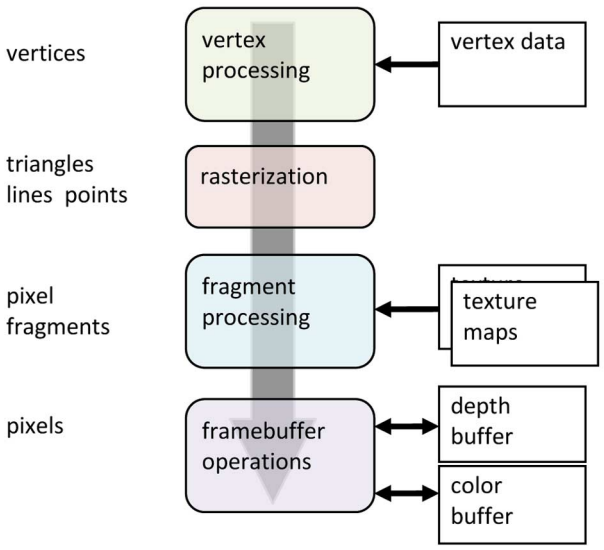

Fig. 2. ***Three-dimensional processing pipeline.***

(2-D) rectangular array of picture elements *(pixels)*, where each pixel represents a discrete color sample of the image.

To synthesize the image, each object in the scene is rendered using a four-step sequence (or four-stage pipeline): geometry processing, rasterization, fragment processing, and frame buffer processing, as shown in Fig. 2.

Geometry processing transforms a 3-D polygonal (triangle) representation of the object's surface through various coordinate spaces to ultimately produce a 2-D projection of the object triangles. The transformations operate on the vertices of the incoming triangle representation and apply operations such as translations and rotations to position the object and to compute additional parameters that are used during later shading calculations. Only the vertices need be projected and the results reconnected with straight lines to reproduce the projected boundary, since the surface of projection is assumed to be planar. For a nonplanar projection such as dome or head-mounted displays, all points on each triangle must be projected.

Rasterization converts each resulting 2-D triangle to a collection of *pixel fragments* corresponding to a discrete sampling of the triangle over a uniform grid (e.g., at the center of each grid cell). During rasterization, a parametric description of the triangle is also computed allowing additional parameters (such as color) to be interpolated at each sample point from values associated with vertices of the original triangle. These interpolated values become part of the pixel fragment.

Each pixel fragment is subsequently processed to compute a final color value. The computation of this color value (termed *shading*) can range from simply using the interpolated color value to computing complex equations incorporating the geometric and appearance descriptions of the object and the environment description. These computations include *texture mapping* operations that use the parametric coordinates of the fragment to sample an associated image (texture map) to generate a color value or other shading parameter. A single fragment may use several texture maps to compute surface material properties and lighting properties and combine these together to produce a final color for the pixel fragment (as in Fig. 1).

The resulting shaded pixel fragments are written to a color buffer (traditionally called a *framebuffer*) that holds a single fragment for each pixel in the final image. As part of this operation, *depth buffering* is used to resolve fragment visibility, that is, to determine whether one fragment at a particular pixel location is closer to the viewer than another. This is done by computing a depth value corresponding to the distance to the viewer for each fragment (during rasterization) and tracking the current version of this value for each fragment in the color buffer. As new fragments are written to the color buffer, their depth values are compared against the current depth value and fragments with a lesser depth value (closer to the viewer) replace the current fragment color and depth value.

II. EVOLUTION

A brief survey of the evolution of graphics hardware is instructive in understanding the current state of graphics processing.

A. 1960s—Origins

The earliest applications driving the development of computer graphics were computer-aided design (CAD) [3] and flight simulation [4]. Following close behind were entertainment applications such as computer games [5] and content creation for film [6]. Early graphics processing focused on controlling an analog *vector display* (e.g., an oscilloscope) to stroke a line or *wireframe* representation of an object by tracing the object's shape with the electron beam [3]. Displays supporting animated drawing required periodic refresh by redrawing the object, whereas a static figure could be drawn once on a long-persistence storage display.

Concurrently, research led to development of algorithms to project a 3-D object representation on to a 2-D plane and simulate linear perspective [7], as well as methods for doing hidden-line [7], [8] and hidden-surface [9], [10] elimination.

By the mid-1960s, the first graphics terminals with autonomous display processing and hardware-computed (accelerated) line-drawing commands were introduced commercially [11]. These are likely the earliest ancestors of the modern graphics processor.

B. 1970s—Uncovering Fundamentals

With the introduction of semiconductor memory, vector display techniques gave way to *raster* techniques using a discretely sampled (pixel) representation of an

image [12], [13]. This was combined with a television-like display device that scanned an electron beam from left to right and top to bottom to display the image—the modern-day computer monitor. While the raster representation was costly in terms of storage, it simplified the transition from wireframe images to solid images. The cost of semiconductor memory limited both the spatial and color resolution of early raster displays, for example, $640 \times 480 \times 8$ bits [14]. Today it is common to see devices that can produce HDTV resolution images at 1920×1080 pixels and 24 bits per pixel (approximately 6 million bytes).

The use of raster representations also inspired research into a variety of areas, including the underlying signal processing theory behind the representation and generation of raster images [15], new image synthesis algorithms such as texture mapping for complex color shading [16], environment mapping for reflections [17], bump mapping for surface roughness [18], and various lighting models for diffuse and specular illumination [17], [19]. There was also considerable interest in algorithms for rapid generation of images, including those amenable to dedicated acceleration hardware, in the pursuit of interactive generation of complex images.

During this time period, a unification of traditional text processing and graphics processing was led by Xerox PARC. Early text display systems used raster techniques in the display terminals but stored only the text data and converted the lines of text to a raster image on-the-fly as part of the display process. A high-resolution monochrome image (606×808) could be represented with less than 100 Kbytes of memory [20], allowing the intermixing of text and graphics. This *bitmap* display is an early example of a transition from fixed-function support (dedicated text support in the terminal) to a more flexible system. This is one of the central patterns in the evolution of graphics systems: early use of fixed-function hardware acceleration to produce a cost-effective solution, followed by replacement with a more flexible interface to broaden applicability and improve quality. To avoid an unmanageable explosion of operating "modes," the interface frequently evolves to a programmable implementation.

An important development from this evolution of text processing was the ability to display the output from multiple applications simultaneously in different regions of the display, termed *windows* [21], [22]. This was the beginning of the graphics equivalent of CPU multiprogramming where multiple applications could not only time-share the CPU and space-share system memory but also time-share the graphics processor and space-share the display output. At the time this capability was rather unique, as many application areas (simulation, mechanical design, etc.) used a single, dedicated application at a time.

Another division in graphics processing is *interactive* versus *noninteractive* processing. The former places stringent constraints on processing time to produce a new images at regular intervals (e.g., 60 times per second), whereas the latter is free to use more sophisticated and time-consuming algorithms (ranging from seconds to hours) to produce higher quality images. Traditionally, noninteractive (or *offline*) processing has used complex, CPU-based algorithms, whereas interactive processing systems (such as the Evans and Sutherland Picture System [23]) used a combination of simple CPU-based algorithms and fixed algorithms with dedicated hardware acceleration. Flexible hardware acceleration capable of interactive execution of a broader set of evolving algorithms remained a distant goal.

C. 1980s—Hardware Acceleration

By the early 1980s, raster systems had largely replaced vector systems and commercial companies were combining microcomputers with raster display systems to produce "smart" terminals and workstations targeted at CAD and word-processing applications. The graphics requirements of interactive word-processing system were largely met by the CPU, though there were cases where some operations had the right mixture of commonality, simplicity, and frequency that dedicated hardware acceleration was built. A simple example of this is the bit-block transfer (BITBLT) operations for copying a rectangular region from one area to another [20], [24]. This can be used to accelerate screen updates for operations such as panning or scrolling the image on the display. They can be further generalized to allow a logical operation (AND, OR, NOT, XOR, etc.) to be computed between the source and destination pixel to produce the final pixel. These are part of the ancestry of the framebuffer operations in the modern processing pipeline.

The economics of the personal computer enabled rapid capture of the market for modestly demanding document-processing applications. This was partially facilitated by *add-in* 2-D raster graphics cards, such as IBM's Color Graphics Adapter [25], that allowed greater configurability for end users. This is a familiar recurring pattern in the evolution of graphics hardware, where technology improvements allow low-cost hardware to meet the performance and quality requirements of formerly high-end applications, moving the market to lower cost platforms (see Fig. 3). The add-in technology allowed personal computers to incrementally provide the capabilities for the "low end" of other applications areas such as electronic CAD (predominantly 2-D graphics). The add-in approach provided both configuration flexibility and the ability to upgrade the graphics capability independent of the rest of the system. The success of PC add-in graphics cards gave rise to the term "video card" to describe them.

Other applications, such as mechanical design, required more intense computations. These included matrix and vector operations involving four-element floating-point vectors for vertex processing, additional floating-point operations for rasterization initialization, and a substantial amount of fixed-point operations for fragment

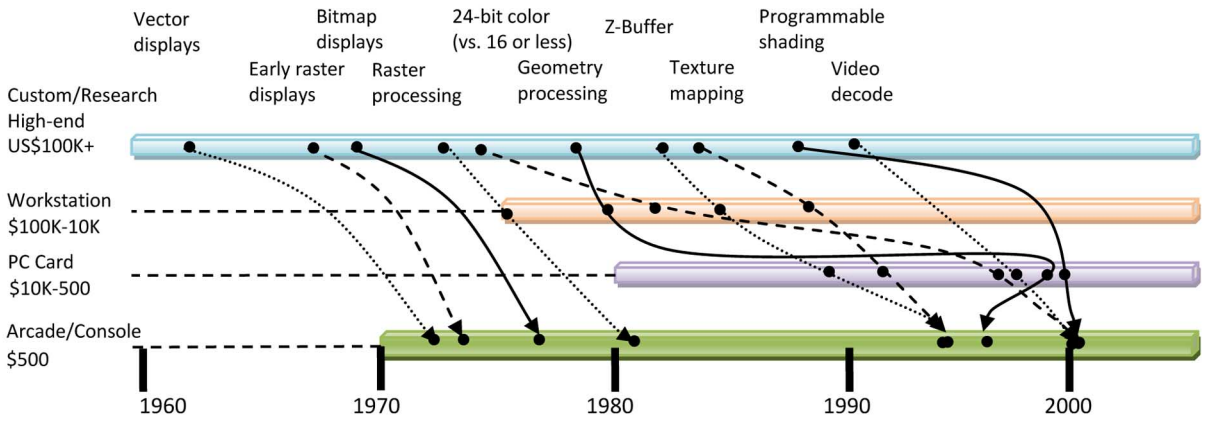

Fig. 3. ***Graphics accelerator evolutionary timeline (approximate). Accelerator implementations are divided into four segments defined by price with independent timelines. Over time, many capabilities (but not all) were introduced in the most expensive segment first and migrated to the other segments, but not necessarily in the introduction order depending on market demands. Accelerated geometry (vertex) processing and 24-bit color were not essential in the more price-sensitive markets, whereas texture mapping was an early requirement for games. By the early 2000s, the source of most features was the PC card segment (flowing up and down).***

processing. To satisfy these requirements, hardware acceleration targeted at these types of computations was incorporated using dedicated logic for each part of the pipeline [26], [27]. These accelerators provided a fixed pipeline of operations. The geometry operations applied a positioning/orienting transform and a projection transform. The rasterization operation generated a set of fragments, and a simple fragment processor generated a color and depth value. Later in the 1980s, this pipeline underwent substantial enhancement in which algorithm development and empirical experience led to more robust and faster implementations with greater functionality: simplified polygonal representation (triangles), depth buffering, simple lighting models, point-sampled geometry, and texture mapping [28], [29].

Both the workstation and PC systems continued to develop more powerful virtualization solutions that allowed the display screen to be space-shared through window systems [30], [31] and graphical user interfaces, and for the acceleration hardware to be time-shared using hardware virtualization techniques [32].

Concurrently, specialized systems (without the space/time sharing requirement) continued to be developed and enhanced for dedicated simulation applications (e.g., GE Compu-Scene IV [4] and Evans and Sutherland CT-5 [33]). These systems had additional requirements around image realism and led to substantial work on accelerated implementation of texture mapping, anti-aliasing, and hidden surface elimination.

New application areas arose around scientific and medical visualization and digital content creation (DCC) for film. These applications used a mixture of workstation (e.g., from Apollo, Sun Microsystems, or Silicon Graphics) and dedicated systems (e.g., Quantel's Harry for film editing [34]) depending on the details of the applications.

During the early part of the 1980s, arcade and home entertainment consoles transitioned to raster-based graphics systems such as the Atari 2600 or the Nintendo NES and SNES. These systems were often *sprite* based, using raster acceleration to copy small precomputed 2-D images (e.g., scene objects) to the display.

During the 1980s, alternative rendering technologies, such as ray-tracing [35] and REYES [36], were explored, demonstrating visual quality benefits in reflections, shadows, and reduction of jagged-edge artifacts over what was becoming the de facto 3-D rasterization pipeline. However, by the end of the 1980s, the 3-D rasterization pipeline had become the pragmatic choice for hardware acceleration for interactive systems, providing hardware vendors with flexible tradeoffs in cost, performance, and quality.

D. 1990s—Standardization, Consolidation

In the early 1990s, workstation accelerators expanded from the requirements of computer-aided design/manufacturing applications to incorporate many of the features found in specialized simulation systems. This resulted in a transition to commercial off-the-shelf (COTS) systems for simulation for all but the most specialized programs. At roughly the same time, dedicated game consoles moved from traditional 2-D raster pipelines to include (crude) texture-mapping 3-D pipelines (such as the Sony PlayStation and Nintendo 64), and 3-D acceleration add-in cards became available for personal computers.

Also during this time, there was renewed effort to standardize the processing pipeline to allow portable applications to be written. This standardization was expressed in terms of the logical pipeline expressed by the OpenGL application programming interface (API) [37] and later by the Direct3D API [38]. These matched similar efforts around 2-D drawing APIs (X Window System 1

[30], GDI [39], SVG [40]), and page or document description languages (PostScript [41], PDF [42]) that had occurred during the late 1980s and early 1990s. These efforts proved more successful than earlier standards such as the Core Graphics System [43] and PHIGS [44].

Two new application segments became prominent in the 1990s. The first was real-time video playback and editing. Originally video was supported by dedicated systems, but in the 1990s workstations started to encroach on that market using the high-performance texturing and framebuffer systems to operate on video frames [45]. Around this time, the concept of unifying image processing and fragment processing took hold. In the latter part of the decade, PC add-in cards also began to include support for decoding compressed MPEG2 video (subsuming this functionality from dedicated decoding cards). Even more significantly, the Internet and the popularity of the World Wide Web increased the demand for displaying text and images and helped raise the bar for color and spatial resolution for low-cost PC systems, for example, moving from 8 bits/pixel to 16 or 24 bits/pixel and from 640×480 to 800×600 or 1024×768 display sizes.

Later in the 1990s, just as workstation technology eroded the dedicated simulator market earlier in the decade, the increasing capabilities of the personal computer eroded the market for workstations for CAD and content-creation applications.

E. 2000s—Programmability, Ubiquity

A large part of the 1990s saw cost reductions in hardware acceleration, with a small number of enhancements to the fixed-function pipeline (multiple textures per fragment, additional texture mapping algorithms). By the end of the decade, PC add-in accelerators, such as NVIDIA's GeForce 256 [46] or ATI's Radeon 7200 [47], incorporated acceleration for fixed-function geometry processing, rasterization, texture-mapped fragment processing, and depth-buffered pixel processing. These accelerators could do a credible job of supporting large portions of the major application domains: CAD, medical imaging, visual simulation, entertainment content creation, and document processing. Game consoles made use of the same technology that was available in personal computers (Nintendo 64, Sega Dreamcast). Around this time, the term *graphics processing unit* (GPU) arose to refer to this hardware used for graphics acceleration.

In the early 2000s, the ongoing process of adding more capability to the logical pipeline took a new turn. Traditionally, there has always been demand for new capabilities in the pipeline. At this time, the demand largely came from the entertainment space with a desire to produce more realistic or more stylized images, with increased flexibility allowing greater expression. In the late 1990s, this was partially achieved using "multipass" techniques in which an object is drawn multiple times (passes) with the same geometric transformations but with different shading in each pass. The results are combined together, for example, summing them using framebuffer operations to produce the final result [48]. As a simple example, the use of multiple textures to affect the surface of an object could be simulated with a single texture system by drawing the object once for each texture to be applied and then combining the results. Multipass methods are powerful but can be cumbersome to implement and consume a lot of extra bandwidth as intermediate pass results are written to and read from memory. In addition to multipass, the logical pipeline continued to be extended with additional vertex and fragment processing operations, requiring what were becoming untenably complex mode settings to configure the sequence of operations applied to a vertex or pixel fragment [49].

The alternative (or complementary) mechanism to fulfill the demand for flexibility was answered by providing application-developer–accessible programmability in some of the pipeline stages to describe the sequencing of operations. This enabled programmers to create custom geometry and fragment processing programs, allowing more sophisticated animation, surface shading, and illumination effects. We should note that, in the past, most graphics accelerators were implemented using programmable technology using a combination of custom and off-the-shelf processing technologies. However, the specific implementation of the programmability varied from platform to platform and was used by the accelerator vendor solely to create firmware that implemented the fixed-function pipeline. Only in rare cases (for example, the Adage Ikonas [50]) would the underlying programmability be exposed to the *application programmer* [50]–[54]. However, by 2001, the popularity of game applications combined with market consolidation, reducing the number of hardware manufacturers and providing a favorable environment for hardware manufacturers to expose programmability [55].

This new programmability allowed the programmer to create a small custom program (similar to a subroutine) that is invoked on each vertex and another program that is invoked on each pixel fragment. These programs, referred to as "shaders," have access to a fixed set of inputs (for example, the vertex coordinates) and produce a set of outputs for the next stage of the pipeline as shown in Fig. 4. Programmable shading technology is rooted in CPU-based film-rendering systems where there is a need to allow programmers (and artists) to customize the processing for different scene objects without having to continually modify the rendering system [56]. Shaders work as an extension mechanism for flexibly augmenting the behavior of the rendering pipeline. To make this technology successful for hardware accelerators, hardware vendors had to support a machine-independent shader representation so that portable applications could be created.

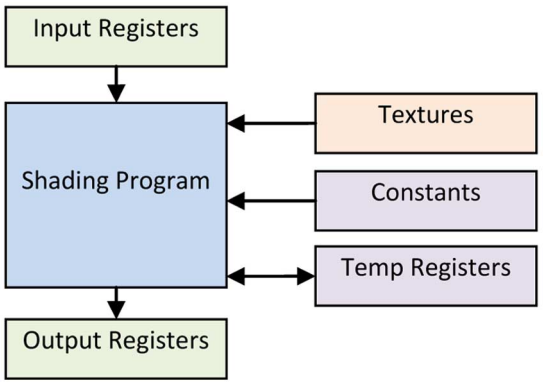

Fig. 4. *Abstract shading processor operates on a single input and produces a single output. During execution, the program has access to a small number of on-chip scratch registers and constant parameters and to larger off-chip texture maps.*

The first half of the decade has seen an increase in the raw capabilities of vertex and pixel shading programs. These capabilities included an increase in range and precision of data types, longer shading programs, dynamic flow control, and additional resources (e.g., larger numbers of textures that can be applied to a pixel fragment) [57]. This increasing sophistication also led to concurrent evolution of programming language support in the form of *shading languages* that provide programming constructs similar to CPU programming languages like "C." These traditional constructs are augmented with specialized support for graphics constructs such as vertices and fragments and the interfaces to the rest of the processing pipeline (HLSL [58], Cg [59], GLSL [60]).

These improvements in general have occurred in parallel with steady performance increases. The technology has also pushed downward from workstations, personal computers, and game consoles to set-top boxes, portable digital assistants, and mobile phones.

III. THE MODERN GPU

Today a moderately priced ($200) PC add-in card is capable of supporting a wide range of graphics applications from simple document processing and Web browser graphics to complex mechanical CAD, DVD video playback, and 3-D games with rapidly approaching cinematic realism. The same fundamental technology is also in use for dedicated systems, such as entertainment consoles, medical imaging stations, and high-end flight simulators. Furthermore, the same technology has been scaled down to support low-cost and low-power devices such as mobile phones. To a large extent, all of these applications make use of the same or a subset of the same processing components used to implement the logical 3-D processing pipeline.

This logical 3-D rasterization pipeline has retained a similar structure over the last 20 years. In some ways, this has also been out of necessity to provide consistency to application programmers, with this same application portability (or conceptual portability) allowing the technology to migrate down to other devices (e.g., cell phones). Many of the characteristics of the physical implementations in today's systems have survived intact from systems 20 years ago or more. Modern GPU design is structured around four ideas:

- exploit parallelism;
- organize for coherence;
- hide memory latency;
- judiciously mixed programmable and fixed-function elements.

The ideas are in turn combined with a programming model that provides a simple and efficient match to these features.

A. Exploiting Parallelism

Fundamental to graphics processing is the idea of parallel processing. That is, primitives, vertices, pixel fragments, and pixels are largely independent, and a collection of any one of these entities can therefore be processed in parallel. Parallelism can be exploited at each stage of the pipeline. For example, the three vertices of a triangle can be processed in parallel, two triangles can be rasterized in parallel, a set of pixel fragments from rasterizing a triangle can be shaded in parallel, and the depth buffer comparisons of the shaded fragments can also be processed in parallel. Furthermore, using pipelining all of these operations can also proceed concurrently, for example, processing the vertices of the next triangle while the previous triangle is being rasterized.

There are also a very large number of these entities to process: a scene might be composed of 1 million triangles averaging 25 pixel fragments per triangle. That corresponds to 3 million vertices and 25 million pixel fragments that can be processed in parallel. This *data parallelism* has remained an attractive target from the earliest hardware accelerators. Combining this with processing vertices, pixel fragments, and whole pixels in a pipelined fashion exploits additional *task parallelism*.

One constraint that complicates the use of parallelism is that the logical pipeline requires that primitives be processed in the order they are submitted to the graphics pipeline, or rather, the net result must be the same as one where primitives are rendered in the order they are submitted. This adds some necessary determinism to the result and allows *order-dependent* algorithms, such as the "painter's algorithm," in which each new object is painted on top of the previous object, to produce the correct image. This constraint means that if two overlapping triangles T_1 and T_2 are submitted in the order T_1 followed by T_2, then where the pixels overlap, the pixels of T_1 must be written to the framebuffer before the pixels of T_2. Applications also interleave state-mutating commands, such as switching texture maps, with primitive-drawing commands, and

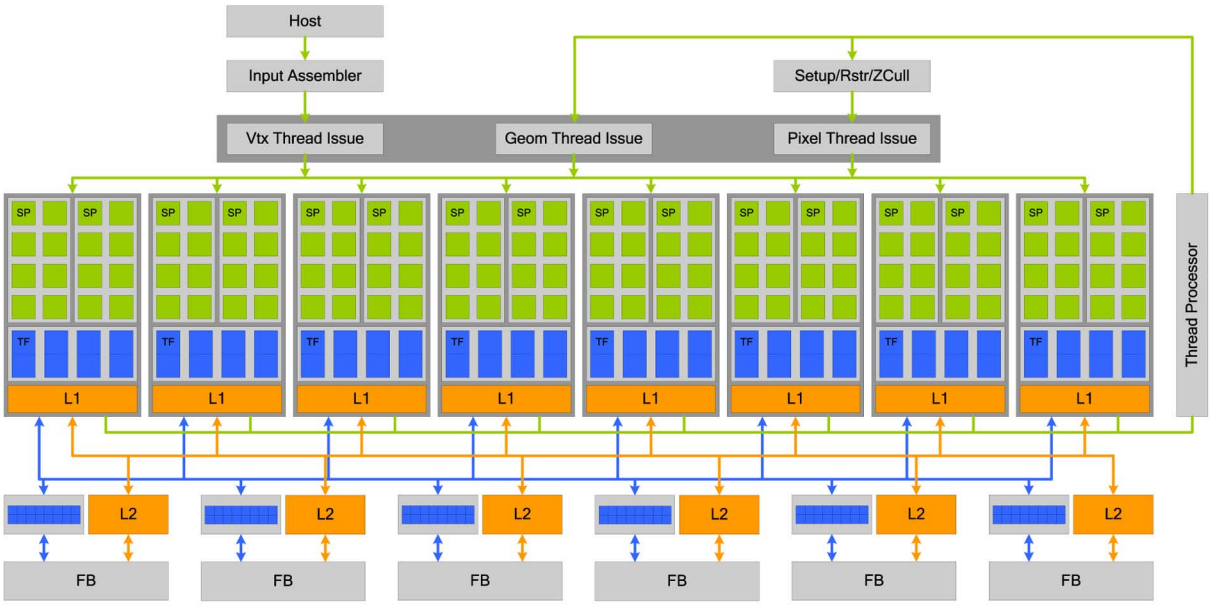

Fig. 5. ***Internal structure of a modern GPU with eight 16-wide SIMD processing units combined with six 64-bit memory interfaces (courtesy of NVIDIA).***

they too must be processed in the order they are submitted. This allows the processing result to be well defined for any mixture of rendering commands.

Fig. 5 shows a block diagram of a commercially available NVIDIA GeForce 8800GTX GPU [61]. At the heart of the system is a parallel processing unit that operates on n entities at a time (vertices, pixel fragments, etc.). This unit is further replicated m times to allow m sets of entities to be processed in parallel. A processing unit can process multiple vertices or pixel fragments in parallel (data parallelism) with different processing units concurrently operating on vertices and fragments (task parallelism).

This implementation represents a transition from previous generations, where separate processing units were customized for and dedicated to either vertex or pixel processing, to a single *unified processor* approach. Unified processing allows for more sophisticated scheduling of processing units to tasks where the number of units assigned to vertex or pixel processing can vary with the workload. For example, a scene with a small number of objects (vertices) that project to a large screen area (fragments) will likely see performance benefits by assigning more processors to fragment processing than vertex processing.

Each processing unit is similar to a simple, traditional, *in-order* (instruction issue) processor supporting arithmetic/logical operations, memory loads (but not stores[1]), and flow control operations. The arithmetic/logical operations use conventional computer arithmetic implementations with an emphasis on floating-point arithmetic. Significant floating-point processing is required for vertex and pixel processing. Akeley and Jermoluk describe the computations required to do modest vertex processing resulting in 465 operations per four-sided polygon [28]. This corresponds to gigaflop/second processing requirements to support today's more than 100 million/s polygon rates. Aggregating across the multiple processing units, current high-end GPUs ($500) are capable of approximately 350–475 Gflops of programmable processing power (rapidly approaching 1 Tflop), using 128–320 floating-point units operating at 0.75–1.3 GHz[2] [61], [62]. Supporting these processing rates also requires huge amounts of data to be read from memory.

B. Exploiting Coherence

A processor operating on n entities at a time typically uses a *single-instruction multiple-data* (SIMD) design, meaning that a single stream of instructions is used to control n computational units (where n is in the range of 8 to 32). Achieving maximum efficiency requires processing n entities with the identical instruction stream. With programmable vertex and fragment processing, maximum efficiency is achieved by executing groups of vertices or groups of pixel fragments that have the identical shader program where the groups are usually greater than the SIMD width n. We call this grouping of like processing *computational coherence*.

[1]Current processors do support store instructions, but their implementations introduce nondeterminism and severe performance degradation. They are not currently exposed as part of the logical graphics pipeline.

[2]Assuming the floating-point unit performs a simultaneous multiply and add operation.

However, computational coherence can be more challenging when the shader programs are allowed to make use of conditional flow control (branching) constructs. Consider the case of a program for processing a pixel fragment that uses the result of a load operation to select between two algorithms to shade the pixel fragment. This means that the algorithm chosen may vary for each pixel in a group being processed in parallel. If processors are given a static assignment to fragments, then the SIMD processing structure necessitates that a fraction of the SIMD-width n processors can execute one of the algorithms at a time, and that fragments using the other algorithm must wait until the fragments using the first algorithm are processed. For each group of n fragments executed on the SIMD processor, the total execution time is $t1 + t2$ and the efficiency compared to the non-branching case is $\max(t1, t2)/(t1 + t2)$. This problem is exacerbated as the SIMD width increases.

SIMD processors can perform some optimizations; for example, detecting that all of the branches go in one direction or another, achieving the optimal efficiency. Ultimately, efficiency is left in the hands of the application programmers since they are free to make choices regarding when to use dynamic branching. It is possible to use other hardware implementation strategies, for example, dynamically reassigning pixel fragments to processors to group like-branching fragments. However, these schemes add a great deal of complexity to the implementation, making scheduling and maintaining the drawing-order constraint more difficult.

Coherence is also important for maximizing the performance of the memory system. The 3 million vertex/25 million fragment scene described above translates to 90 million vertices/s and 750 million fragments/s when processed at an update rate of 30 frames/s. Consider only the 750 M/s fragment rate: assuming that each fragment reads four texture maps that each require reading 32 bytes, this results in a read rate of 24 GB/s. Combining that with the depth buffer comparisons and updates, and writing the resulting image, may (conservatively) add another 3 GB/s of memory bandwidth. In practice, high-end GPUs support several times that rate using multiple memory devices and very wide busses to support rates in excess of 100 GB/s. For example, the configuration in Fig. 5 uses 12 memory devices, each with a maximum data rate of 7.2 GB/s, grouped in pairs to create 64-bit-wide data channels.

However, simply attaching multiple memory devices is not sufficient to guarantee large bandwidths can be realized. This is largely due to the nature of dynamic random-access memory devices. They are organized as a large 2-D matrix of individual cells (e.g., 4096 cells/row). A memory transaction operates on a single row in the matrix and transfers a small contiguous block of data, typically 32 bytes (called column access granularity). However, there can be wasted time while switching between rows in the matrix or switching between writes and reads [63]. These penalties can be minimized by issuing a minimum number of transfers from the same row, typically two transfers (called row access granularity).

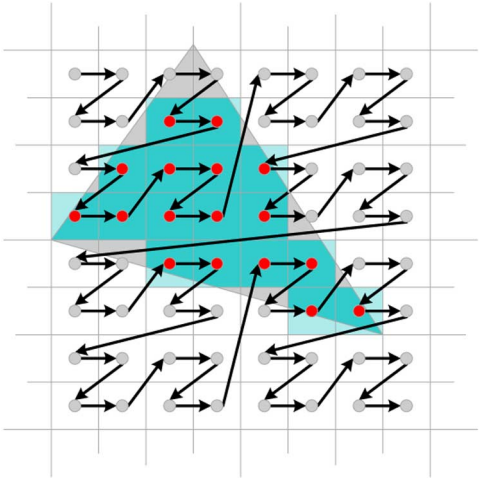

Fig. 6. *Coherence-preserving rasterization pattern.*

Thus, to maintain maximum memory system efficiency and achieve full bandwidth, reads and writes must transfer multiple contiguous 32-byte blocks of data from a row before switching to another row address. The computation must also use the entire block of data in the computation; otherwise the effective efficiency will drop (unused data are wasted). To support these constraints, texture images, vertex data, and output images are carefully organized, particularly to map 2-D spatial access patterns to coherent linear memory addresses. This careful organization is carried through to other parts of the pipeline, such as rasterization, to maintain fragment coherence, as shown in Fig. 6 [2].

Without this careful organization, only a small fraction of the potential memory bandwidth would be utilized and the processor would frequently become idle, waiting for the data necessary to complete the computations.

Coherent memory access alone is not sufficient to guarantee good processor utilization. Processor designers have to both achieve high bandwidth and manage the latency or response time of the memory system.

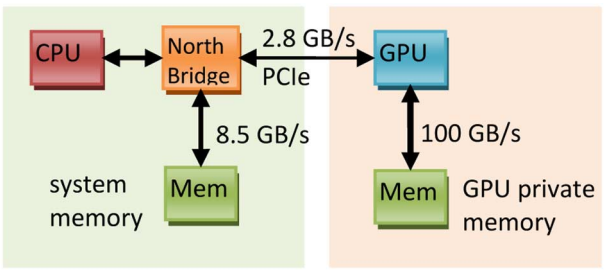

Fig. 7. *Relative bandwidths in a personal computer using PCI Express interconnect to a high-end GPU add-in card.*

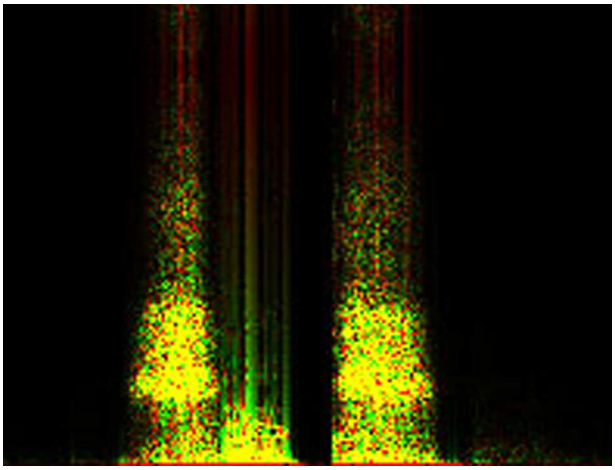

Fig. 8. *Spectrogram using FFT.*

C. Hiding Memory Latency

In a conventional in-order processor, execution will *stall* when the result from a memory load operation is needed in a subsequent instruction (e.g., source operand of an arithmetic instruction). This stall will last until the result of the load operation is available. CPUs employ large, high-speed, on-chip caches to reduce the latency of operations to external memory with the assumption that there is both *locality* and *reuse* in the data, such that the data being referenced multiple times will likely be present in the cache. The sheer volume of data referenced while rendering a scene (tens or hundreds of megabytes of texture data) greatly reduces the effectiveness of a cache and there is minimal reuse of the data. GPUs do employ caches, but they are substantially smaller and are used to aggregate SIMD memory requests to optimize memory bandwidth. Apparent memory latency is managed using a different strategy.

There is so much data parallelism available that the processor can switch to executing another vertex or pixel fragment while waiting for a load operation to complete. To accomplish this, there must be storage for all of the execution state for an additional pixel fragment (for an SIMD processor, the state is actually for n fragments) so that the processor can switch to this other fragment when necessary. To be practical, the processor must be able to switch between these pixel fragments (their stored state) with zero or near zero cost. This is similar to the concept of hyperthreading technology in CPUs (e.g., Sun's Niagara processor [64]). The technique, more generally referred to as *multithreading*, has been utilized in several earlier multiprocessor system designs [65].

To further complicate matters, the latency of a memory read operation can be very large relative to the time it takes to execute individual instructions, and a second fragment is likely to also issue a load operation of its own before the read from the first fragment has completed. Thus a larger pool of pixel or vertex "threads" must be available for execution to cover the latency of a single read operation. For example, if the ratio of memory and instruction latencies is ten to one, then a pool of ten pixel threads must be available to span the latency of a single read operation. In practice, these ratios can easily exceed 100 to 1.

One implementation of this technique might use a set of k pixel threads (where each thread is SIMD-width n fragments wide) arranged in a ring and, as each instruction is issued, move to the next thread in the ring. This means that a program of length x requires kx instruction cycles to complete the processing of a given pixel fragment, but it will complete k pixel fragments in total. That is, the latency to process any pixel fragment may be quite large, but the throughput (pixel fragments/unit time) is optimal. Note that this technique has a multiplicative effect on the *effective width* of an SIMD array, widening it from n to kn. This in turn has implications on computational coherence.

In practice, GPU implementations use more sophisticated variations on this idea, but the use of multiple threads is an essential mechanism for maintaining efficiency in the presence of large numbers of memory read operations. These include *data-dependent* reads in which the address of a later read operation is dependent on the result of an earlier memory read operation and cannot be scheduled until the earlier read has completed.

D. Fixed-Function Versus Programmable Elements

While there has been a steady increase in the programmable processing power of GPUs, a significant portion of an implementation continues to make use of fixed-function hardware. The reasons are that a fixed-function implementation is more cost effective and power efficient, and there has not been a compelling application need to convert it to a programmable element. Two examples of this are texture filtering and rasterization.

Texture filtering implements a small set of filtering algorithms: point sampling, bilinear filtering, trilinear mipmap filtering, and anisotropic filtering, as well as texture image decompression and color format conversion operations. These operations benefit hugely from customized implementations taking advantage of constrained data types (32-bit floating-point processing would be overkill). Application programmers increasingly see benefit to implementing their own customized filtering algorithms. However, since the percentage of processing using these algorithms is small, it is more cost effective to slightly increase the programmable processing power for these instances and leave the remainder using the fixed-function processing. At some point in the future, this balance may change in favor of moving all texture filtering to the programmable processors. Rasterization, on the other hand, serves a much more specialized purpose, and there has been little interest in making it more programmable. This allows the implementation to remain small and efficient.

E. CPU–GPU Connection

The GPU executes independently from the CPU but is controlled by the CPU. Application programs running on the CPU use graphics API, runtime, and driver software components to communicate with the GPU. Most of the communication involves placing commands or data in memory buffers and transmitting them to the GPU. Graphical data that are accessed frequently (vertices, textures, output images) by the GPU are often placed in a high-bandwidth memory attached directly to the GPU, with the CPU being used to set the initial state for these objects. Even with the dedicated GPU memory, the CPU sends a great deal of data to the GPU on behalf of the application. Modern PCs use the PCI Express (PCIe) bus [66] to connect the CPU and add-in graphics card. PCI Express is a scalable bus, divided into serial, bidirectional *lanes* of 2.5 Gbits/s.[3] A 16-lane bus is capable of a theoretical bandwidth of 4 GBytes/s in each direction, but current implementations achieve closer to 2.8 GB/s and often provide even less bandwidth in the direction from the device to the CPU.

Entertainment consoles or other dedicated devices use their own interconnect strategy. In the case of consoles, this may offer considerably higher bandwidth than is available through PCI Express. In some implementations, the GPU may be integrated into the memory controller chip (e.g., north bridge) and CPU may share the same memory rather than using dedicated memory for the GPU. These integrated GPUs are a popular low-cost alternative to add-in cards. These options provide interesting cost and performance tradeoffs and also affect some of the processing strategies used by the application developer. However, the basic GPU architecture concepts remain unaffected.

F. Scalability

One of the benefits of the parallel nature of graphics processing is that the hardware implementations can scale to different problem sizes. For example, if the target application processes fewer triangles and pixel fragments, then the number of processors and amount of memory bandwidth can be reduced proportionately. GPU implementations contain some amount of fixed resources (e.g., for the display controllers, CPU interconnect, memory controller, etc.) and a larger proportion of replicated elements for processing. This scaling manifests itself in different versions or stock keeping units (SKUs) for the PC, ranging from high-end (enthusiast) implementations to low-end (entry) systems. The parallel nature of the problem also allows additional scaling up, by distributing work either from within a frame (a single scene) or from multiple frames across multiple instances of an implementation (e.g., AMD Crossfire, NVIDIA SLI).[4] This may be implemented transparently to the application or explicitly programmed by the application developer. However, to achieve the benefits of scalability, the application programmer is left with the challenge of building applications that can scale across the performance range. This is typically achieved by changing the richness of the content and varying the resolution of the output image.

Implementations may be further scaled down for embedded or low-cost consumer devices, but more often, these implementations use a slightly older (simpler) processing pipeline with a design optimized for specific platform constraints such as power consumption. However, the basic architecture ideas remain intact across a large range of implementations.

IV. NEW APPLICATIONS

The combination of raw performance and programmability has made the GPU an attractive vehicle for applications beyond the normal document/presentation graphics, gaming/simulation, and CAD/DCC realms. The target is applications that exhibit significant data parallelism, that is, where there are a large number of data elements that can be operated on independently.

Development of nongraphical applications on graphics accelerators also has a rich history with research projects dating back to more than a decade using the multipass methods with texture mapping and framebuffer operations. However, the widespread availability of low-cost add-in accelerator cards combined with greater programmability through shaders has precipitated intense interest amongst researchers. This class of research is referred to as general-purpose programming on GPUs (GPGPU). Owens *et al.* survey the history and state of the art in GPGPU in [67]. We provide a brief overview of the principles, some of the current applications, and the limitations.

A. Data Parallel Processing

Rasterizing a polygon and processing pixel fragments can be viewed as an operation on a domain where the domain shape corresponds to the shape of the rasterized geometry. For example, drawing a rectangle of size $n \times m$ pixels corresponds to operating on a 2-D domain of $n \times m$ domain points. A local operation on each of the domain points amounts to executing a pixel shading program at each of the corresponding fragments. The trick is that the domain data are stored as a 2-D texture map and the program must read the corresponding domain point from memory before operating on it. Since the fragment program has access to the entire texture map, it can also perform region operations by reading a neighborhood of domain points. The result of the domain operations is written to the output image, which can be either used in a subsequent computation (reusing the output image as a texture map) or used in subsequent operations on the CPU, e.g., writing the results to a file.

[3]PCI express version 1.1 is 2.5 Gb/s per lane; the recently announced version 2.0 is 5.0 Gb/s per lane.

[4]See http://www.ati.amd.com/technology/crossfire/CrossFireWhitePaperweb2.pdf.

Research in parallel programming has produced a set of basic operators for data parallel processing. Parallel algorithms are constructed from these operations. The point-wise processing algorithm corresponds to the *map* operator that applies a function to each element in a domain. Other operators include reduce, gather, scatter, scan, select, sort, and search [67].

GPU architecture is specifically tailored to rendering tasks with no underlying intent to support more general parallel programming. This makes mapping some of the data parallel operators onto the GPU a challenge.

The reduce operator produces a single output value from the input domain, for example, summing the values in the domain. It is typically implemented as a multipass sequence of partial reductions, for example, reducing the size of the domain by 1/2 or 1/4 in each pass by summing two or four adjacent values in a shader and outputting the result.

The gather operation performs an indirect memory read $(\mathrm{v} = \mathrm{mem}[\mathrm{x}])$ and maps well to reads from texture maps. Conversely, scatter performs an indirect memory write $(\mathrm{mem}[\mathrm{x}] = \mathrm{v})$ and does not map well as the output of a fragment shader goes to a fixed location in the output image. Scatter can be implemented using vertex processing to control the destination address by adjusting the 2-D image coordinates of the vertex. Alternatively, programs may transform the computation to an equivalent one using gather when it is possible.

Other parallel operations (scan, select, sort, and search) require an, often complex, multipass sequence of operations to efficiently implement them. There are many subtleties to providing optimal implementations, often requiring detailed understanding of a specific GPU design. Similar to dynamic branching, it is also important how these operations are used. For example, gather and scatter operations can quickly erode memory system efficiency if they are directed to widely varying addresses.

Graphics applications typically manipulate simple data types such as four-element vectors representing colors or geometric coordinates. These data types are usually organized into 1-, 2-, or 3-D arrays, which can be efficiently manipulated on a GPU. More general programming typically involves more complex data types and irregular data structures such as linked lists, hash tables, and tree structures. Implementing these data structures in shader programs can be cumbersome, especially while trying to maintain computation and memory coherence. To alleviate these difficulties, efforts such as Glift [68] have been made to create reusable GPU data structure libraries.

B. Parallel Languages

As GPUs are designed for running graphics applications, most of the development for programming languages and APIs has been targeted at writing graphics applications. This makes nongraphics programming more challenging as the programmer must master idioms from the graphics APIs and languages, such as drawing triangles to create a set of domain points and trigger fragment processing across that domain. Shading programs must be written to process the domain points, using texture mapping operations to read data associated with each domain point and writing the computed result as a color value.

To simplify this programming task and hide the underlying graphics idioms, several programming languages and runtimes have been created. These range from systems from graphics vendors that expose low-level details of the underlying graphics hardware implementation (CTM [69], CUDA [70]) to research and commercial higher-level languages and systems intended to simplify development of data parallel programs. These systems cover a spectrum of approaches including more traditional array processing style languages (Accelerator [71], PeakStream [72]) to newer *stream processing* languages [73] (derived from parallel processing research in specialized stream processing hardware [74]). In many cases, the language combines the parts of the code that execute on the CPU and the parts that execute on the GPU in a single program. This differs from many of the graphics APIs (OpenGL, Direct3D) that deliberately make the boundary between the CPU and GPU explicit.

One advantage of higher level languages is that they preserve high-level information that can *potentially* be used by the underlying runtime to manage execution and memory coherence. In contrast, lower level systems leave that largely up to the programmer, requiring the programmer to learn various architectural details to approach peak efficiencies. Low-level systems may allow programmers to achieve better performance at the cost of portability, but they also may allow access to newer, more efficient processing constructs that are not currently available in the graphics APIs. One example of this is additional support for explicit scatter operations and sharing on-chip memory available in NVIDIA's CUDA system. These may prove beneficial, though also at the cost of portability—at least initially.

C. Application Areas

The significant interest in harnessing the processing power at the GPU has led to investigations into a large number of application areas. We describe a small set of these areas. Owens *et al.* [67] provides a more comprehensive summary.

D. Image and Signal Processing

One of the more obvious application areas is general image and signal processing on 1-, 2-, or 3-D data. Graphics applications already make use of image-processing operations such as sampling and filtering in texture mapping operations. Extensions for fixed-function imaging operations, such as histogram and convolution, were added as extensions to the OpenGL architecture in the late 1990s [37], but they have effectively been subsumed by the programmable pipeline. A number of projects have

implemented convolution with various size kernels, fast Fourier transforms (FFTs), segmentation, and histograms [75], [76], [77]. Such signal-processing methods are often critical in analysis applications. For example, GPU-accelerated backprojection accelerates reconstruction from sensor data sufficiently fast to allow interactive scanning and visualization of tomograms. The reconstruction process involves taking the multiple 2-D projections of 3-D data captured by the sensors and applying an approximate inverse Radon transform [78]. In geophysical analysis, wavelet compression can significantly reduce (100x) the size of raw seismic sensor data. Headwave has demonstrated a 200x speedup over the CPU for their GPU-based wavelet compression/decompression algorithm, allowing interactive manipulation of much larger data sets [79].

E. Linear Algebra

A broad area that has shown promising results is linear algebra. This includes both dense and sparse matrix operations such as multiplying a vector by a matrix or solving a system of equations. Multiplication can be viewed as a series of dot products, where each dot product is a reduction operation. For dense matrices, great care must be taken to preserve memory coherence or the processors will be memory starved [80], [81]. Sparse matrices contain a large number of zero elements (e.g., a banded matrix), which, if treated as a dense matrix, results in many wasted calculations. Sparse representations require more complex data structures to represent the nonzero elements, typically using a separate index array to map a sparse element back to the matrix. To ensure good computational coherence, sparse computations may be sorted on the CPU to process vectors with the same number of nonzero elements together in groups [82]. With access to efficient scan primitives, the sparse computations can be done efficiently without CPU intervention [83].

F. Engineering Analysis and Physical Simulation

Computationally intensive physical simulations (fluids, weather, crash analysis, etc.) have traditionally relied on high-performance computing (HPC) systems (supercomputers). The processing power of GPUs is attractive in this area for two reasons. One is the opportunity to provide lower cost solutions for solving these research and industrial problems, perhaps using the cost differential to solve even larger problems. Secondly, physical simulations are becoming an increasingly important way of adding new types of behaviors to entertainment applications (e.g., better character animations; more complex, but not necessarily realistic, interactions with the physical world).

There are many types of problems that fit in this simulation category, but two important classes of problems are solving ordinary and partial differential equations. Ordinary differential equations are functions of a single variable containing one or more derivatives with respect to that variable. A simple example is Newton's law of acceleration (force $=$ mass $\times$ acceleration). In computer graphics, it is common to model objects as collections of particles (smoke, fluids [84]) and to solve this equation for each of the particles using an explicit integration method [85]. Such a method computes the new value (e.g., position at a new time step) using the computed values from previous time steps. If the particles are independent and subject to global forces, then each particle can be computed independently and in parallel. If the particles also exert local forces on one another then information about neighboring particles must also be collected (from the previous time step) and used in the computation. Several GPU-based simulators that model the physics and render the results have been built, showing speed improvements of 100 or more over CPU-based algorithms [86], [87].

Partial differential equations are functions of multiple independent variables and their partial derivatives. They are frequently found in physical models for processes that are distributed in space or space and time, such as propagation of sound or fluid flow. A spatial domain is sampled using finite differences (or finite elements) on regular or irregular grids. The grid of finite elements forms a system of (linear or nonlinear) equations that are solved using a method optimized around the characteristics of the equations. For example, a conjugate gradient solver is an iterative solver for a symmetric positive definite (SPD) sparse matrix and a Jacobi solver is an iterative method for a linear system with a diagonally dominant matrix. These solvers make heavy use of the sparse linear algebra methods described earlier using optimized representations and algorithms to exploit the particular sparse pattern. GPU implementations have been applied to simulation problems such as fluid flow [88] and reaction-diffusion [89].

G. Database Management

Another interesting area is data management problems including database operations such as joins and sorting on large amounts of data. The large memory bandwidth makes the GPU very attractive, but maintaining coherence in sorting algorithms is challenging. This has led to a resurgence of interest in sorting networks such as the bitonic sort [90], which can be mapped relatively efficiently onto a GPU [91]. GPUs are particularly effective in performing *out of core* sorting, that is, a sort where the number of records is too large to fit in system memory and must be read from disk. The GPUTeraSort [92] implements a CPU–GPU hybrid system in which the CPU manages disk transfers while the GPU performs the bandwidth-intensive sorting task on large blocks of keys. The result is a system that is more cost-effective at sorting a large collection of records than a similarly priced system using only CPUs and no GPU, as measured by the Penny Sort benchmark [93].

In addition to bulk data processing, the graphics processing ability of GPUs can also be leveraged for specialized database queries such as spatial queries used in mapping or geospatial processing.

H. Financial Services

The financial services industry has produced increasingly complex financial instruments requiring greater computational power to evaluate and price. Two examples are the use of Monte Carlo simulations to evaluate credit derivatives and evaluation of Black–Scholes models for option pricing. More sophisticated credit derivatives that do not have a closed-form solution can use a Monte Carlo (nondeterministic) simulation to perform numerical integration over a large number of sample points. PeakStream has demonstrated speed increases of a factor of 16 over a CPU when using a high-end GPU to evaluate the differential equation and integrate the result [94].

The Black–Scholes model prices a put or call option for a stock assuming that the stock prices follow a geometric Brownian motion with constant volatility. The resulting partial differential equation is evaluated over a range of input parameters (stock price, strike price, interest rate, expiration time, volatility) where the pricing equation can be evaluated in parallel for each set of input parameters. Executing these equations on a GPU [95] for a large number of inputs (15 million), the GPU approaches a factor of 30 faster than a CPU using PeakStream's software [94] and 197 times faster using CUDA [97] (measured on different GPU types).

I. Molecular Biology

There have been several projects with promising results in the area of molecular biology, in particular, with the analysis of proteins. Molecular dynamics simulations are used to simulate protein folding in order to better understand diseases such as cancer and cystic fibrosis. Stanford University's *folding@home* project has created a distributed computational grid using volunteer personal computers [96]. The client program executes during idle periods on the PC performing simulations. Normally, PC clients execute the simulation entirely on the CPU, but a version of the molecular dynamics program *gromacs* has been written for the GPU using the Brook programming language [73]. This implementation performs as much as a factor of 20–30 faster than the CPU client alone [98].

Another example is protein sequence analysis using mathematical models based on hidden Markov models (HMMs). A model for a known program sequence (or set of sequences) with a particular function on homology is created. A parallel algorithm is used to search a large database of proteins by computing a probability of each search candidate being in the same family as the model sequences. The probability calculation is complex, and many HMMs may be used to model a single sequence, so a large amount of processing is required. The GPU version of the algorithm [99] uses data-parallel processing to simultaneously evaluate a group of candidates from the database. This makes effective use of the raw processing capabilities and provides an order of magnitude or better speedup compared to a highly tuned sequential version on a fast CPU.

J. Results

Many of these new applications have required a significant programming effort to map the algorithms onto the graphics processor architecture. Some of these efforts were on earlier, less-general graphics processors, so the task is becoming somewhat less complex over time, but it is far from simple to achieve the full performance of the processors. Nevertheless, some of these applications are generating significant commercial interest, particularly in the area of technical computing. This is encouraging the graphics processor vendors to experiment with product variations targeted at these markets.

Programmer productivity will continue to be the limiting factor in developing new applications. Some productivity improvements may come from adding further generalizations to the GPU, but the challenge will continue to be in providing a programming model (language and environment) that allows the programmer to realize a significant percentage of the raw processing power without painstakingly tuning the data layout and algorithms to specific GPU architectures.

K. Limits of Parallelism

In our discussions thus far, we have focused principally on data parallelism and, to a lesser extent, on task parallelism. It is important to note that often, only some parts of an algorithm can be computed in parallel and the remaining parts must be computed sequentially due to some interdependence in the data. This has the result of limiting the benefits of parallel processing. The benefit is often expressed as *speedup* of a parallelized implementation of an algorithm relative to the nonparallelized algorithm. Amdahl's law defines this as $1/[(1-P)+P/S]$, where P is the proportion of the algorithm that is parallelized with speedup S [100]. For a perfectly parallelizable algorithm, $P=1$ and the total improvement is S. In parallel systems, S is usually related to the number of processors (or parallel computing elements) N.

A special case of Amdahl's law can be expressed as $1/[F+(1-F)/N]$, where F is the fraction of the algorithm that cannot be parallelized. An important consequence of this formula is that as the number of computing elements increases, the speedup approaches $1/F$. This means that algorithms with even small sequential parts will have limited speedup even with large numbers of processors. This is shown in Fig. 9, where speedup versus number of processors is plotted for several different values of F. With 10% serial processing, the maximum speedup reaches a factor of ten, and the effectiveness of adding more processors greatly diminishes after approximately 90 processors. Even with only 1% sequential processing, that maximum speedup reaches a factor of 100, with increases in speedup rapidly diminishing after 300 processors. To mitigate this effect, general-purpose parallel systems include fast sequential (scalar) processors, but they remain the limiting factor in improving performance. GPU programmable

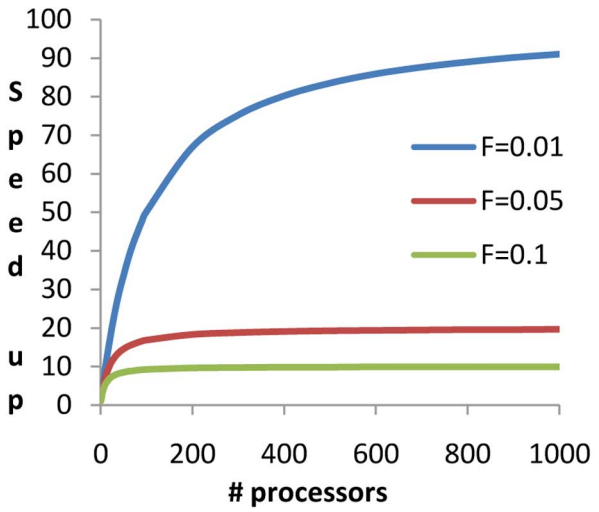

Fig. 9. *Speedup versus number of processors for applications with various fractions of nonparallelizable code.*

processing units are tuned for parallel processing. Sequential algorithms using a single slice of a single GPU SIMD processing unit will make inefficient use of the available processing resources. Targeting a more general mix of parallel and sequential code may lead GPU designers to add fast scalar processing capability.

V. THE FUTURE

Despite years of research, deployment of highly parallel processing systems has, thus far, been a commercial failure and is currently limited to the high-performance computing segment. The graphics accelerator appears to be an exception, utilizing data-parallel techniques to construct systems that scale from a 1/2 Tflop of floating-point processing in consoles and PCs to fractions of that in handheld devices, all using the same programming model. Some of this success comes from focusing solely on the graphics domain and careful exposure of a programming model that preserves the parallelism. Another part comes from broad commercial demand for graphics processing. The topic on many people's minds is whether this technology can be successfully applied *commercially* to other application spaces.

A. Market Forces

Today the GPU represents a device capable of supporting a great deal of the (consolidated) market of graphics- and media-processing applications. Tuning capabilities to serve multiple markets has led to a pervasiveness measured by an installed base of hundreds of millions of GPUs and growth of several hundred million per year on the PC alone [101]. We have grouped these markets into five major segments: graphical user interface/document/presentation (including internet browsing), CAD/DCC, medical imaging, games/simulation, and multimedia. Fig. 10 shows a visualization of these markets and the market addressed by current GPUs. First, we note that GPUs do not fully address any of these markets, as there may be more specialized or esoteric requirements for some applications in those areas. This gap between capabilities and requirements represents potential opportunity. As is the case for any product development, the market economics of addressing these opportunities are considered carefully in new products. Some enhancements are valuable for multiple segments, e.g., increasing screen resolution. However, for features that are targeted to specific segments, the segments that provide the greatest return for additional investment are currently games and multimedia.

Another important aspect of the market forces, particularly for commodity markets, is the significance of cost. The more utilitarian parts of the markets, such as documents and presentation or even high-definition (HD) media playback, can be adequately served with lower cost GPUs. This skews the installed base for high-volume markets such as consumer and business devices towards low-cost parts that have just enough performance to run the target applications. In the PC market, this translates to a large installed base of integrated GPUs that have a modest fraction (one-tenth) of the performance of their high-end peers. In practice, the PC GPU market is divided into multiple segments (e.g., enthusiast, performance, mainstream, and value) with more demanding segments having smaller sales volumes (albeit with higher sales margins) [101].

New applications represent a potentially disruptive force in the market economics—that is, by finding the next "killer app" that translates to broad market demand for additional GPUs or more capable GPUs. This could be in the form of a new fixed-function appliance or a new application running on a general-purpose platform such as a PC or cell phone. Three-dimensional games in the mid-1990s and DVD playback in the late 1990s are two examples of "killer apps" that pushed demand for substantial new capabilities in PC and console graphics accelerators.

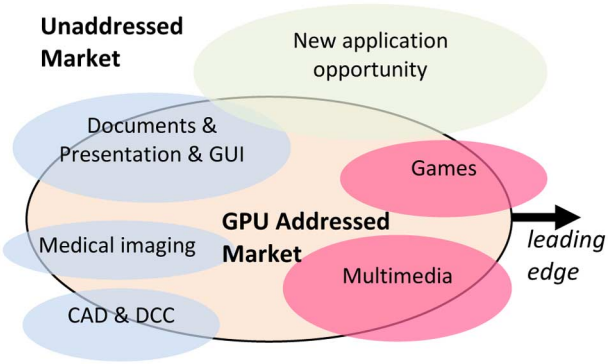

Fig. 10. *Graphics acceleration markets.*

There are both low- and high-risk strategies for developing such applications. The low-risk strategy is to focus on applications that can be addressed with current GPU capabilities and look for synergies with features being added for other market segments. An advantage of this strategy is that it can be pursued by application developers independently of GPU manufacturers. A downside of the strategy is that the "killer app" may require features not currently available. This leads to higher risk strategies involving additional engineering and silicon cost to add features specific to new markets. This strategy requires participation from the GPU manufacturers as well as the application developers, despite which, the "killer app" may still be hindered by poor market acceptance.

B. Technical Challenges

Graphics accelerators are well positioned to absorb a broader data-parallel workload in the future. Beyond the challenge of effectively programming these systems, several potential obstacles lay in the path: power consumption, reliability, security, and availability.

The number of transistors on a GPU chip has followed the exponential integration curve of Moore's law [102], resulting in devices that will soon surpass 1 billion transistors in a 400 mm^2 die. Clock speed has also increased aggressively, though peak GPU clock speeds are roughly 1/3 to 1/2 of those of CPUs (ranging from 1 to 1.5 GHz). These increases have also resulted in increasing power dissipation, and GPU devices are hindered by the same power-consumption ceilings encountered by CPU manufacturers over the last several years. The ceiling for power consumption for a chip remains at approximately 150 W, and there are practical limits on total power consumption for a consumer-oriented system, dictated by the power available from a standard wall outlet. The net result is that GPU manufacturers must look to a variety of technologies to not only increase absolute performance but also improve efficiency (performance per unit of power) through architectural improvements. A significant architectural dilemma is that fixed-function capability is usually more efficient than more general-purpose functionality, so increasing generality may come at an increasingly higher total cost.

A related issue around increasing complexity of devices concerns system reliability. Increasing clock speed and levels of integration increases the likelihood of transmission errors on interconnects or transient errors in logic and memory components. For graphics-related applications the results range from unnoticeable (a pixel is transiently shaded with the wrong value) to a major annoyance (the software encounters an unrecoverable failure and either the application or the entire system must be restarted). Recently, attention has focused on more seamless application or system recovery after an error has been detected, with only modest effort at improving ability to transparently correct errors. In this respect, GPUs lag behind CPUs, where greater resources are devoted to detecting and correcting low-level errors before they are seen by software. Large-scale success on a broader set of computational tasks requires that GPUs provide the same levels of computation reliability as CPUs.

A further challenge that accompanies broader GPU usage is ensuring that not only are results trustable from correctness perspective but also the GPU device does not introduce new vectors for security attacks. This is already a complicated problem for open systems such as consumer and enterprise PCs. Auxiliary devices such as disk controllers, network interfaces, and graphics accelerators already have read/write access to various parts of the system, including application memory, and the associated controlling software (drivers) already serve as an attack vector on systems. Allowing multiple applications to directly execute code on the GPU requires that, at a minimum, the GPU support hardware access protections similar to those on CPUs used by the operating system to isolate executing applications from one another and from system software. These changes are expected to appear over the next several years [103].

C. Hybrid Systems

One possible evolution is to extend or further generalize GPU functionality and move more workloads onto it. Another option is to take parts of the GPU architecture and merge it with a CPU. The key architectural characteristics of current GPUs are: high memory bandwidth, use of SIMD (or vector) arithmetic, and latency-hiding mechanisms on memory accesses. Many CPUs already contain vector units as part of their architectures. The change required is to extend the width of these units and add enough latency tolerance to support the large external memory access times. At the same time, the memory bandwidth to the CPU must be increased to sustain the computation rates for the wider vector units (100 GB/s GPU versus 12 GB/s CPU bandwidth).[5] The economic issues around building such a CPU are perhaps even more significant than the technical challenges, as these additions increase the cost of the CPU and other parts of the system, e.g., wider memory busses and additional memory chips. CPU and system vendors (and users) will demand compelling applications using these capabilities before making these investments.

There are still other risks with hybrid architecture. The graphics processing task is still comparatively constrained, and GPU vendors provide extra fixed-function logic to ensure that the logical pipeline is free of bottlenecks. Great attention is placed on organizing data structures such as texture maps for memory efficiency. Much of this is hidden from graphics application developers through the use of graphics API and runtimes that abstract the logical pipeline and hide the implementation details. If it is necessary for

[5]12.8 GB/s assuming dual-channel (128-bit) DDR2-800 memory. In the future, dual-channel DDR3-1333 will be capable of 21.3 GB/s.

developers of new applications to master these implementation details to achieve significant performance, broad commercial success may remain elusive in the same way it has for other parallel processing efforts.

While Intel has alluded to working on such a hybrid CPU–GPU using multiple ×86 processing cores [104], [105], a more evolutionary approach is to integrate existing CPU and GPU designs onto a single die or system on a chip (SOC). This is a natural process of reducing cost through higher levels of integration (multiple chips into a single chip), but it also affords an opportunity to improve performance and capability by exploiting the reduction in communication latency and tightly coupling the operation of the constituent parts. SOCs combining CPUs and graphics and media acceleration are already commonplace in handheld devices and are in the PC product plans of AMD (Fusion [106]) and Intel (Nehalem [109]).

VI. CONCLUSION

Graphics processors have undergone a tremendous evolution over the last 40+ years, in terms of expansion of capabilities and increases in raw performance. The popularity of graphical user interfaces, presentation graphics, and entertainment applications has made graphics processing ubiquitous through an enormous installed base of personal computers and cell phones.

Over the last several years, the addition of programmability has made graphics processors an intriguing platform for other types of data-parallel computations. It is still too early to tell how commercially successful new GPU-based applications will be. It is similarly difficult to suggest an effective combination of architecture changes to capture compelling new applications while retaining the essential performance benefits of GPUs. GPU vendors are aggressively pushing forward, releasing products, tailored for computation, that target the enterprise and HPC markets [107], [108]. The customizations include reducing the physical footprint and even removing the display capability.

However, GPU designers are not alone, as CPU vendors are—out of necessity—vigorously embracing parallel processing. This includes adding multiple processing units as multi- or many-core CPUs and other architectural changes, such as transactional memory, to allow applications to exploit parallel processing [110]. Undoubtedly, there will be a transfer of ideas between CPU and GPU architects. Ultimately, we may end up with a new type of device that has roots in both current CPU and GPU designs. However, architectural virtuosity alone will not be sufficient: we need a programming model and tools that allow programmers to be productive while capturing a significant amount of the raw performance available from these processors. Broad success will require that the programming environments be viable beyond experts in research labs, to typical programmers developing real-world applications.

Graphics accelerators will also continue evolving to address the needs of the core graphics-processing market. The quest for visual realism will certainly fuel interest in incorporating ray-tracing and other rendering paradigms into the traditional pipeline. These changes are more likely to be evolutionary additions to the rasterization pipeline rather than a revolutionary switch to a new pipeline. In 2006, application-programmability was added to the per-primitive processing step (immediately before rasterization) in the Direct3D logical 3-D pipeline [57], and it is conceivable that other programmable blocks could be added in the future. Innovative developers, in market segments such as game applications, will also encourage looking beyond traditional graphics processing for more sophisticated physical simulations, artificial intelligence, and other computationally intensive parts of the applications. With so many possibilities and so many opportunities, an interesting path lies ahead for graphics processor development. ■

Acknowledgment

The author is grateful to H. Moreton (NVIDIA), K. Fatahalian (Stanford University), K. Akeley (Microsoft Research), C. Peeper, M. Lacey (Microsoft), and the anonymous reviewers for careful review and insightful comments on this paper. The author also wishes to thank S. Drone for the images in Figs. 1 and 8.

REFERENCES

[1] E. Haines, "An introductory tour of interactive rendering," *IEEE Computer Graphics and Applications*, vol. 26, no. 1, pp. 76–87, Jan./Feb. 2006.

[2] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, Mar. 2005.

[3] I. Sutherland, "SketchPad: A man-machine graphical communication system," in *Proc. AFIPS Sprint Joint Comput. Conf.*, 1963, vol. 23, pp. 329–346.

[4] R. Bunker and R. Economy, "Evolution of GE CIG systems," Simul. Contr. Syst. Dept., General Electric, Daytona Beach, FL, 1989, Tech. Rep.

[5] J. M. Graetz, "The origin of spacewar!" in *Creative Computing*. Cambridge, MA: MIT Press, 2001.

[6] A. M. Noll, "The digital computer as a creative medium," *IEEE Spectrum*, vol. 4, pp. 89–95, Oct. 1967.

[7] L. G. Roberts, "Machine perception of three-dimensional solids," MIT Lincoln Lab., TR315, May 1963.

[8] A. Appel, "The notion of quantitative invisibility and the machine rendering of solids," in *Proc. ACM Nat. Conf.*, 1967, pp. 387–393.

[9] W. J. Bouknight, "An improved procedure for generation of half-tone computer graphics representations," Coordinated Science Lab., Univ. of Illinois, R-432, Sep. 1969.

[10] J. Warnock, "A hidden-surface algorithm for computer-generated halftone pictures," Computer Science Dept., Univ. of Utah, TR 4-15, Jun. 1969.

[11] IBM. (1964). *Preliminary user's guide for IBM 1130/2250 model 4 graphics system*. [Online]. Available: http://www.bitsavers.org/pdf/ibm/2250/Z22-1848-0_2250prelimUG.pdf

[12] A. M. Noll, "Scanned-display computer graphics," *CACM*, vol. 14, no. 3, pp. 143–150, Mar. 1971.

[13] J. T. Kajiya, I. E. Sutherland, and E. C. Cheadle, "A random-access video frame buffer," in *Proc. Conf. Comput. Graph., Pattern Recognit., Data Structure*, May 1975, pp. 1–6.

[14] R. Shoup, "SuperPaint: An early frame buffer graphics system," *IEEE Ann. History Comput.*, vol. 23, no. 2, pp. 32–37, Apr.–Jun. 2001.

[15] F. Crow, "The aliasing problem in computer-generated shaded images," *CACM*, vol. 20, no. 11, pp. 799–805, Nov. 1977.

[16] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, Univ. of Utah, Salt Lake City, 1974.

[17] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *CACM*, vol. 19, no. 10, pp. 542–547, Oct. 1976.

[18] J. F. Blinn, "Simulation of wrinkled surfaces," in *Proc. SIGGRAPH '78*, Aug. 1978, pp. 286–292.

[19] B. T. Phong, "Illumination for computer generated pictures," *CACM*, vol. 18, no. 6, pp. 311–317, Jun. 1975.

[20] C. P. Thacker, R. M. McGeight, B. W. Lampson, R. F. Sproull, and R. R. Boggs, "Alto: A personal computer," in *Computer Structures, Readings and Examples*, D. Siewiorek, G. Bell, and A. Newell, Eds., 2nd ed. New York: McGraw-Hill, 1979.

[21] D. Engelbart and W. English, "A research center for augmenting human intellect," in *ACM SIGGRAPH Video Rev.*, 1994, p. 106. (reprinted from 1968).

[22] A. Goldberg and D. Robson, "A metaphor for user interface design," in *Proc. 12th Hawaii Int. Conf. Syst. Sci.*, 1979, pp. 148–157.

[23] Evans and Sutherland. (1974). *Picture system technical brochure*. [Online]. Available: http://www.rchive. computerhistory.org/resources/text/Evans_Sutherland/EvansSutherland.3D.1974.102646288.pdf

[24] R. Pike, B. N. Locanthi, and J. Reiser, "Hardware/software trade-offs for bitmap graphics on the blit," *Software Practice Exper.*, vol. 15, no. 2, pp. 131–151, 1985.

[25] J. Sanchez and M. Canton, *The PC Graphics Handbook*. Boca Raton, FL: CRC Press, 2003, ch. 1, pp. 6–17.

[26] J. Clark, "The geometry engine: A VLSI geometry system for graphics," *Comput. Graph.*, vol. 16, no. 3, pp. 127–133, 1982.

[27] J. Torborg, "A parallel processor architecture for graphics arithmetic operations," in *Proc. SIGGRAPH'87*, Jul. 1987, vol. 21, no. 4, pp. 197–204.

[28] K. Akeley and T. Jermoluk, "High-performance polygon rendering," in *Proc. ACM SIGGRAPH Conf.*, 1988, pp. 239–246.

[29] P. Haeberli and K. Akeley, "The accumulation buffer: Hardware support for high-quality rendering," in *Proc. ACM SIGGRAPH Conf.*, 1990, pp. 309–318.

[30] R. W. Scheifler and J. Gettys, "The X window system," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, Apr. 1986.

[31] R. Pike, "The Blit: A multiplexed graphics terminal," *AT&T Bell Labs Tech. J.*, vol. 63, no. 8, pp. 1607–1631, Oct. 1984.

[32] D. Voorhies, D. Kirk, and O. Lathrop, "Virtual graphics," in *Proc. SIGGRAPH '88*, 1988, pp. 247–253.

[33] R. Schumacker, "A new visual system architecture," in *Proc. 2nd I/ITSEC*, Salt Lake City, UT, Nov. 1980, pp. 94–101.

[34] R. Thorton, "Painting a brighter picture," *Inst. Elect. Eng. Rev.*, vol. 36, no. 10, pp. 379–382, Nov. 1990.

[35] A. Glassner, Ed., *An Introduction to Ray Tracing*. London, U.K.: Academic, 1989.

[36] R. Cook, L. Carpenter, and E. Catmull, "The Reyes image rendering architecture," in *Proc. SIGGRAPH '87*, Jul. 1987, pp. 95–102.

[37] M. Segal and K. Akeley, *The OpenGL graphics system: A specification*, Silicon Graphics, Inc., 1992–2006.

[38] Microsoft. (2006). *Direct3D 10 Reference*. [Online]. Available: http://www.msdn.microsoft.com/directx

[39] F. Yuan, *Windows Graphics Programming: Win32 GDI and DirectDraw*. Englewood Cliffs, NJ: Prentice-Hall, Dec. 2000.

[40] W3C. (2003). *Scalable vector graphics (SVG) 1.1 specification*. [Online]. Available: http://www.w3.org/TR/SVG/

[41] *PostScript Language Reference*. Reading, MA: Addison-Wesley, 1999.

[42] Adobe Systems. (2006). *PDF Reference*, 6th ed. [Online]. Available: http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference.pdfversion 1.7

[43] J. C. Michener and J. D. Foley, "Some major issues in the design of the core graphics system," *ACM Comput. Surv.*, vol. 10, no. 4, pp. 445–463, Dec. 1978.

[44] Computer Graphics—Programmer's Hierarchical Interactive Graphics System (PHIGS) (Part 1: Functional Description), ISO/IEC 9592-1:1989, American National Standards Institute, 1989.

[45] R. Braham, "The digital backlot," *IEEE Spectrum*, vol. 32, pp. 51–63, Jul. 1995.

[46] NVIDIA. (1999). *GeForce 256*. [Online]. Available: http://www.nvidia.com/page/geforce256.html

[47] AMD. (2000). *Radeon 7200*. [Online]. Available: http://www.ati.amd.com/products/radeon7200/

[48] M. Peercy, M. Olano, J. Airey, and J. Ungar, "Interactive multi-pass programmable shading," in *Proc. SIGGRAPH 2000*, 2000, pp. 425–432.

[49] NVIDIA. (1999). *Register combiners OpenGL extension specification.* [Online]. Available: http://www.oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt

[50] N. England, "A graphics system architecture for interactive application-specific display functions," *IEEE Comput. Graph. Appl.*, vol. 6, pp. 60–70, Jan. 1986.

[51] G. Bishop, "Gary's Ikonas assembler, version 2: Differences between Gia2 and C," Univ. of North Carolina—Chapel Hill, Computer Science Tech. Rep. TR82-010, 1982.

[52] A. Levinthal and T. Porter, "Chap—A SIMD graphics processor," in *Proc. SIGGRAPH 84*, Jul. 1984, vol. 18, no. 3, pp. 77–82.

[53] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Gold-feather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories," in *Proc. ACM SIGGRAPH '89*, Jul. 1989, vol. 23, no. 3, pp. 79–88.

[54] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-speed rendering using image composition," in *Proc. ACM SIGGRAPH '92*, Jul. 1992, vol. 26, no. 2, pp. 231–240.

[55] E. Lindholm, M. Kilgard, and H. Moreton, "A user-programmable vertex engine," in *Proc. of SIGGRAPH '01*, Aug. 2001, pp. 149–158.

[56] P. Hanrahan and J. Lawson, "A language for shading and lighting calculations," in *Proc. ACM SIGGRAPH '90*, Aug. 1990, vol. 24, no. 4, pp. 289–298.

[57] D. Blythe, "The Direct3D 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, Aug. 2006.

[58] Microsoft Corp. (2002). *High-level shader language: In DirectX 9.0 graphics*. [Online]. Available: http://www.msdn.microsoft.com/directx/

[59] W. R. Mark, R. S. Glanville, K. Akeley, and M. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, Jul. 2003.

[60] J. Kessenich, D. Baldwin, and R. Rost. (2004). *The OpenGL shading language version 1.10.59*. [Online]. Available: http://www.opengl.org/documentation/oglsl.html

[61] NVIDIA. (2007). *GeForce 8800 GPU architecture overview*. [Online]. Available: http://www.nvidia.com/object/IO_37100.html

[62] AMD. (2007). *ATI Radeon HD 2900 series—GPU specifications*. [Online]. Available: http://www.ati.amd.com/products/Radeonhd2900/specs.html

[63] V. Echevarria. (2005). "How memory architectures affect system performance," *EETIMES Online*. [Online]. Available: http://www.us.design-reuse.com/articles/article10029.html

[64] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar.–Apr. 2005.

[65] K. Kurihara, D. Chaiken, and A. Agarwal, "Latency tolerance through multithreading in large-scale multiprocessors," in *Proc. Int. Symp. Shared Memory Multiprocess.*, Apr. 1991, pp. 91–101.

[66] PCI SIG. (2007). *PCI Express Base 2.0 specification*. [Online]. Available: http://www.pcisig.com/specifications/pciexpress/

[67] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[68] A. Lefohn, J. Knis, R. Strzodka, S. Sengupta, and J. Owens, "Glift: Generic, efficient, random-access GPU data structures," *ACM Trans. Graph.*, vol. 25, no. 1, pp. 60–99, Jan. 2006.

[69] M. Segal and M. Peercy, "A performance-oriented data parallel virtual machine for GPUs," in *SIGGRAPH 2006 Sketch*, 2006.

[70] NVIDIA. (2007). *CUDA Documentation*. [Online]. Available: http://www.developer.nvidia.com/cuda/

[71] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data-parallelism to program GPUs for general purpose uses," in *Proc. 12th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, Oct. 2006, pp. 325–335.

[72] PeakStream. (2007). *Technology Overview*. [Online]. Available: www.peakstreaminc.com/product/overview/

[73] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on

graphics hardware," in *Proc. ACM SIGGRAPH 2004*, Aug. 2004, pp. 777–786.

[74] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in *Proc. Int. Conf. Comput. Design*, Sep. 2002, pp. 282–288.

[75] D. Horn. (2006). *libgpufft*. [Online]. Available: http://www.sourceforge.net/projects/gpufft/

[76] A. Griesser, "Real-time GPU-based foreground-background segmentation," Computer Vision Lab, ETH Zurich, Switzerland, Tech. Rep. BIWI-TR-269, Aug. 2005.

[77] T. McReynolds and D. Blythe, *Advanced Graphics Programming Using OpenGL*. San Francisco, CA: Morgan Kaufmann, 2005, ch. 12, pp. 225–226.

[78] F. Xu and K. Mueller, "Ultra-fast 3D filtered backprojection on commodity graphics hardware," in *Proc. IEEE Int. Symp. Biomed. Imag.*, Apr. 2004, vol. 1, pp. 571–574.

[79] Headwave Inc. (2007). *Technology Overview*. [Online]. Available: http://www.headwave.com

[80] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proc. Graph. Hardware 2004*, Aug. 2004, pp. 133–138.

[81] V. Volkov. (2007). *120 Gflops in matrix-matrix multiply using DirectX 9.0*. [Online]. Available: http://www.cs.berkeley.edu/~volkov/sgemm/index.html

[82] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *Proc. ACM SIGGRAPH 2003*, Jul. 2003, vol. 22, no. 3, pp. 908–916.

[83] S. Sengupta, M. Harris, Y. Zhang, and J. Owens, "Scan primitives for GPU computing," in *Proc. Graph. Hardware 2007*, Aug. 2007, pp. 97–106.

[84] W. Reeves, "Particle systems—A technique for modeling a class of fuzzy objects," in *Proc. ACM SIGGRAPH '83*, Jul. 1983, vol. 17, no. 3, pp. 359–375.

[85] J. Butcher, *Numerical Methods for Ordinary Differential Equations*, 2nd Ed. Chichester, U.K.: Wiley, 2003.

[86] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proc. Graph. Hardware 2004*, Aug. 2004, pp. 123–132.

[87] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3D flows," *IEEE Trans. Vis. Comput. Graphics*, vol. 11, no. 6, pp. 744–756, 2005.

[88] Y. Liu, X. Liu, and E. Wu, "Real-time 3D fluid simulation on GPU with complex obstacles," in *Proc. Pacific Graph. 2004*, Oct. 2004, pp. 247–256.

[89] A. Sanderson, M. Meyer, R. Kirby, and C. Johnson, "A framework for exploring numerical solutions of advection-reaction-diffusion equations using a GPU-based approach," *Comput. Vis. Sci., 2007*.

[90] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Comput. Conf.*, 1968, vol. 32, pp. 307–314.

[91] I. Buck and T. Purcell, "A toolkit for computation on GPUs," in *GPU Gems*, R. Fernando, Ed. Boston, MA: Addison-Wesley, 2004, ch. 37, pp. 627–630.

[92] N. Govindaraju, J. Gray, and D. Manocha, "GPUTeraSort: High performance graphics coprocessor sorting for large database management," in *Proc. ACM SIGMOD Conf.*, Jun. 2006, pp. 325–336.

[93] J. Gray. (2006). *Indy penny sort benchmark results*. [Online]. Available: http://www.research.microsoft.com/barc/SortBenchmark/

[94] PeakStream. (2007). *High performance modeling of derivative prices using the PeakStream platform*. [Online]. Available: http://www.peakstreaminc.com/reference/peakstream_finance_technote.pdf

[95] C. Kolb and M. Pharr, "Option pricing on the GPU," in *GPU Gems 2*, M. Pharr, Ed. Boston, MA: Addison-Wesley, 2005, ch. 45, pp. 719–731.

[96] Stanford University. (2007). *Folding@Home Project*. [Online]. Available: http://www.folding.stanford.edu/

[97] I. Buck. (2006, Nov. 13). "GeForce 8800 & NVIDIA CUDA: A new architecture for computing on the GPU," in *Proc. Supercomput. 2006 Workshop: General Purpose GPU Comput. Practice Exper.*, Tampa, FL. [Online]. Available: www.gpgpu.org/sc2006/workshop/presentations/Buck_NVIDIA_Cuda.pdf

[98] Stanford Univ. (2007). *Folding@Home on ATI GPU's: A major step forward*. [Online]. Available: http://www.folding.stanford.edu/FAQ-ATI.html

[99] D. R. Horn, D. M. Houston, and P. Hanrahan, "ClawHMMER: A streaming HMMer-search implementation," in *Proc. 2005 ACM/IEEE Conf. Supercomput.*, Nov. 2005, p. 11.

[100] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proc. AFIPS Conf.*, 1967, vol. 30, pp. 483–485.

[101] J. Peddie. (2007, Mar.). "4th Quarter 2006: PC-based graphics shipments market activity," Market Watch. [Online]. Available: http://www.jonpeddie.com/MarketWatch.shtml

[102] G. Moore. (1965). "Cramming more components onto integrated circuits," Electron. Mag. [Online]. Available: http://www.intel.com/technology/mooreslaw/index.htm

[103] S. Pronovost, H. Moreton, and T. Kelley. (2006, May). *Windows display driver model (WDDM) v2 and beyond* [Online]. Available: http://www.download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/PRI103_WH06.ppt

[104] J. Stokes. (2007, Jun.). *Clearing up the confusion over Intel's Larrabee, Part II*. [Online]. Available:http://www.arstechnica.com/news.ars/post/20070604-clearing-up-the-confusion-over-intels-larrabee-part-ii.html

[105] P. Otellini, "Extreme to mainstream," in *Proc. IDF Fall 2006*, San Francisco, CA. [Online]. Available: http://www.download.intel.com/pressroom/kits/events/idffall_2007/KeynoteOtellini.pdf

[106] P. Hester and B. Drebin. (2007, Jul.). *2007 Technology Analyst Day*. [Online]. Available:http://www.amd.com/us-en/assets/content_type/DownloadableAssets/July_2007_AMD_2Analyst_Day_Phil_Hester-Bob_Drebin.pdf

[107] AMD. (2006). *Enterprise stream processing*. [Online]. Available: http://www.ati.amd.com/products/streamprocessor/index.html

[108] NVIDIA. (2006, May). NVIDIA Tesla: CPU computing technical brief. [Online]. Available: http://www.nvidia.com/docs/IO/43395/Compute_Tech_Brief_v1-0-0_final.pdf

[109] Intel, "45 nm product press briefing," in *Proc. IDF Fall 2007*, San Francisco, CA. [Online]. Available: http://www.intel.com/pressroom/kits/events/idffall_2007/BriefingSmith45nm.pdf

[110] A. Adl-Tabatabai, C. Kozyrakis, and B. Saha. (2006, Dec.). "Unlocking concurrency: Multicore programming with transactional memory," ACM Queue. [Online]. *4(10)*, pp. 24–33. Available: http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=444

ABOUT THE AUTHOR

David Blythe (Member, IEEE) received the B.Sc. and M.S. degrees from the University of Toronto, Toronto, ON, Canada, in 1983 and 1985, respectively.

He is a Software Architect with the Desktop and Graphics Technology Group, Windows Product Division, Microsoft Corp., Redmond, WA. He is responsible for graphics architecture associated with hardware acceleration, driver subsystem, low-level graphics and media APIs, and the window system. Prior to joining Microsoft in 2003, he was a Cofounder of BroadOn Communications Corp., a startup producing consumer 3-D graphics and networking devices and a network content distribution system. From 1991 to 2000, he was with the high-end 3-D graphics group at Silicon Graphics, where he contributed to the design of several graphics accelerators and the OpenGL graphics API. During his time at SGI, he participated in the roles of Member of Technical Staff, Principal Engineer, and Chief Engineer. He regularly participates as an author, reviewer, and lecturer in technical conferences. In 2002, he was a Cofounder and Editor of the OpenGL ES 3-D graphics standard for embedded systems and coauthored a book on 3-D graphics programming. His current interests included graphics architecture, parallel processing, and design of large software systems.

Mr. Blythe is a member of ACM.