Beyond Programmable Shading: Fundamentals

SIGGRAPH 2008
5/1/2008

Course Organizers:
Aaron Lefohn, Intel
Mike Houston, AMD

Course Speakers:
Aaron Lefohn   Intel
Mike Houston   AMD
Chas. Boyd     Microsoft
Kayvon Fatahalian Stanford University
Tom Forsyth    Intel
David Luebke   NVIDIA
John Owens     University of California, Davis

Speaker Contact Info:

Chas. Boyd, Microsoft
1 Microsoft Way
Bldg 84/1438
Redmond, WA 98052-6399
chasb@microsoft.com

Kayvon Fatahalian
Gates Building Rm 381
353 Serra Mall
Stanford University
Stanford, CA 94305
kayvonf@graphics.stanford.edu

Tom Forsyth
6841 NE 137th Street
Kirkland WA 98034
tomf@radgametools.com

Mike Houston, AMD
4555 Great America Parkway
Suite 501
Santa Clara, CA 95054
Michael.Houston@amd.com

Aaron Lefohn, Intel
2700 156th Ave NE, Suite 300
Bellevue, WA 98007
aaron.lefohn@intel.com

David Luebke
1912 Lynehburg Dr.
Charlottesville, VA 22903
dlebke@nvidia.com

John Owens
Electrical and Computer Engineering
University of California, Davis
One Shields Avenue
Davis, CA 95616
jowens@ece.ucdavis.edu
Beyond Programmable Shading: Fundamentals

Course Description:
This first course in a series gives an introduction to parallel programming architectures and environments for interactive graphics. There are strong indications that the future of interactive graphics involves a programming model more flexible than today’s OpenGL/Direct3D pipelines. As such, graphics developers need to have a basic understanding of how to combine emerging parallel programming techniques with the traditional interactive rendering pipeline. This course gives an introduction to several parallel graphics architectures, programming environments, and an introduction to the new types of graphics algorithms that will be possible.

Intended Audience:
We are targeting researchers and engineers interested in investigating advanced graphics techniques using parallel programming techniques on many-core GPU and CPU architectures, as well as graphics and game developers interested in integrating these techniques into their applications.

Prerequisites:
Attendees are expected to have experience with a modern graphics API (OpenGL or Direct3D), including basic experience with shaders, textures, and framebuffers and/or background with parallel programming languages. Some background with parallel programming on CPUs or GPUs is useful but not required as an overview of will be provided in the course.
Level of difficulty: Advanced

Special presentation requirements:
Several speakers will bring their own demo machines for use in the course.

Speakers:
- Aaron Lefohn, Intel
- Mike Houston, AMD
- Chas Boyd, Microsoft
- Kayvon Fatahalian, Stanford
- Tom Forsyth, Intel
- David Luebke, NVIDIA
- John Owens, UC Davis
“Beyond Programmable Shading: Fundamentals”

- Introduction:
  - Why and how is interactive graphics programming changing? Lefohn: 10 min
    - Very high throughput parallel hardware increases flexibility and complexity of algorithms that will run at > 30 fps
    - The transition from “programmable shading” to “programmable graphics”
    - What does all of this programmability mean for graphics?

- Parallel Architectures for Graphics:
  - Overview of graphics architectures: Fatahalian: 15 min
  - NVIDIA architecture: Luebke: 20 min
  - AMD/ATI architecture: Houston: 20 min
  - Intel architecture: Forsyth: 20 min

- Parallel Programming Models Overview: Owens: 20 min
  - What to look for in the coming talks

- 15 minute break

- Parallel Programming for Interactive Graphics:
  - Brook+ / CAL: Mike Houston: 20 min
    - The Brook+ computing platform
    - Combining Brook+ and DX/OGL together for graphics
    - Example graphics algorithms enabled by Brook+
  - CUDA: David Luebke: 20 min
    - The CUDA GPU Computing platform
    - Combining CUDA + DX/OGL together for graphics
    - Example graphics algorithms enabled by CUDA
  - Future Direct3D: Chas Boyd: 20 min
    - Traditional DX graphics pipeline
    - General GPU computation in Direct3D
    - Combining GPU-Compute + DX together for graphics
  - TBA: A new programming model: TBA: 20 min
  - Intel: Aaron Lefohn: (20 min)
    - Content to be announced at SIGGRAPH

- Wrap-Up, Q&A: All speakers: 5+ minutes
Speaker Biographies

Aaron Lefohn, Ph.D.

Aaron Lefohn is a Senior Graphics Architect at Intel on the Larrabee project. Previously, he designed parallel programming models for graphics as a Principal Engineer at Neoptica, a computer graphics startup that was acquired by Intel in October 2007. Aaron's Ph.D. in Computer Science from the University of California Davis focused on data structure abstractions for graphics processors and data-parallel algorithms for rendering. From 2003 - 2006, he was a researcher and graphics software engineer at Pixar Animation Studios, focusing on interactive rendering tools for artists and GPU acceleration of RenderMan. Aaron was formerly a theoretical chemist and was an NSF graduate fellow in computer science.

Aaron Lefohn  
2700 156th Ave NE, Suite 300  
Bellevue, WA  98007  
425-881-4891  
aaron.lefohn@intel.com

Mike Houston, Ph.D.

Mike Houston is a System Architect in the Advanced Technology Development group at AMD in Santa Clara working in architecture design and programming models for parallel architectures. He received his Ph.D. in Computer Science from Stanford University in 2008 focusing on research in programming models, algorithms, and runtime systems for parallel architectures including GPUs, Cell, multi-core, and clusters. His dissertation includes the Sequoia runtime system, a system for programming hierarchical memory machines. He received his BS in Computer Science from UCSD in 2001 and is a recipient of the Intel Graduate Fellowship.

Mike Houston  
4555 Great America Parkway  
Suite 501  
Santa Clara, CA 95054  
408-572-6010  
Michael.Houston@amd.com
Chas Boyd

Chas. is a software architect at Microsoft. Chas. joined the Direct3D team in 1995 and has contributed to releases since DirectX 3. Over that time he has worked closely with hardware and software developers to drive the adoption of features like programmable hardware shaders and float pixel processing. He has developed and demonstrated initial hardware-accelerated versions of techniques like hardware soft skinning, and hemispheric lighting with ambient occlusion. He is currently working on the design of future DirectX releases and related components.

Chas. Boyd
1 Microsoft Way
Bldg 84/1438
Redmond WA 98052-6399
425-922-7859
chasb@microsoft.com

Kayvon Fatahalian

Kayvon Fatahalian is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming systems for commodity parallel architectures and computer graphics/animation systems for the interactive and film domains. His thesis research seeks to enable execution of more flexible rendering pipelines on future GPUs and multi-core PCs.

Gates Building Rm 381
353 Serra Mall
Stanford University
Stanford, CA 94301
kayvonf@graphics.stanford.edu

Tom Forsyth

Tom Forsyth has been rendering Cobra MkIIIs on everything he’s ever used. In rough chronological order he has worked on the ZX Spectrum, Atari ST, 386, Virge, Voodoo, 32X, Saturn, Pentium1, Permedia2, Permedia3, Dreamcast, Xbox1, PS2, 360, PS3 and now Larrabee. Past jobs include writing utilities for Microprose, curved-surface libraries for Sega, DirectX drivers for 3Dlabs, three shipped games for Muckyfoot Productions, and Granny3D and Pixomatic for RAD Game Tools. He is currently working for Intel as a software and hardware architect on the Larrabee project.
David Luebke, Ph.D.

David Luebke is a Research Scientist at NVIDIA Corporation, which he joined after eight years on the faculty of the University of Virginia. He has a Ph.D. in Computer Science from the University of North Carolina and a B.S. in Chemistry from the Colorado College. Luebke's research interests are GPU computing and realistic real-time computer graphics. Recent projects include advanced reflectance and illumination models for real-time rendering, image-based acquisition of real-world environments, temperature-aware graphics architecture, and scientific computation on GPUs. Past projects include leading the book "Level of Detail for 3D Graphics" and the Virtual Monticello museum exhibit at the New Orleans Museum of Art.

David Luebke
1912 Lynchburg Dr.
Charlottesville, VA 22903
434-409-1892
dlebke@nvidia.com

John Owens, Ph.D.

John Owens is an assistant professor of electrical and computer engineering at the University of California, Davis. His research interests are in commodity parallel hardware and programming models, including GPU computing. At UC Davis, he received the Department of Energy Early Career Principal Investigator Award and an NVIDIA Teaching Fellowship. John earned his Ph.D. in electrical engineering in 2003 from Stanford University and his B.S. in electrical engineering and computer sciences in 1995 from the University of California, Berkeley.

John Owens
Electrical and Computer Engineering
University of California, Davis
One Shields Avenue
Davis, CA 95616
530-754-4289
jowens@ece.ucdavis.edu
http://www.ece.ucdavis.edu/~jowens/
Disclaimer about these Course Notes

• The material in this course is bleeding edge
  - Unfortunately, that means we can’t share most of the details with you until SIGGRAPH 2008
  - Most talks are missing from the submitted notes
  - The talks that are included will change substantially

• To address this inconvenience
  - We will post all course notes/slides on a permanent web page, available the first day of SIGGRAPH 2008
  - We have included in the notes a number of related recently published articles that provide key background material for the course
Future interactive rendering techniques will be an inseparable mix of data- and task-parallel algorithms and graphics pipelines.
How do we write new interactive 3D rendering algorithms?
Fixed-Function Graphics Pipeline

- Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...

- Examples
  - Shadow volumes
  - Hidden line removal
  - ...

Beyond Programmable Shading: Fundamentals
Programmable Shading

- Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders

- Examples
  - Parallax mapping
  - Shadow-mapped spot light
  - ...

Beyond Programmable Shading: Fundamentals
Beyond Programmable Shading

• Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders
  - Plus: Writing data- and task-parallel algorithms
    • Analyze results of rendering pipeline
    • Create data structures used in rendering pipeline

  - Examples
    • Dynamic summed area table
    • Dynamic quadtree adaptive shadow map
    • Dynamic ambient occlusion
    • ...

Beyond Programmable Shading: Fundamentals
“Fast Summed-Area Table Generation and its Applications,”
Hensley et al., Eurographics 2005

“Resolution Matched Shadow Maps,”
Lefohn et al., ACM Transactions on Graphics 2007

“Dynamic Ambient Occlusion and Indirect Lighting,” Bunnell, GPU Gems II, 2005
Beyond Programmable Shading

• Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders
  - Plus: Writing data- and task-parallel algorithms
    • Analyze results of rendering pipeline
    • Create data structures used in rendering pipeline
  - Plus: Extending, modifying, or creating graphics pipelines

- Examples
  • PlayStation 3 developers creating hybrid Cell/GPU graphics pipelines
    - See afternoon talk from Jon Olick (Id Software)

• Active area of research
Why “Beyond Programmable Shading?”

Short answer:
- The parallel processors in your desktop machine or game console are now flexible and powerful enough to execute both
  - User-defined parallel programs and
  - Graphics pipelines
- ...All within 1/30th of a second
The Point Is...

• Interactive graphics programming is changing

• This course gives you:
  - Introduction to the HW causing/enabling this change
  - Programming tools used to explore this new world
  - A little bit about what developers/researchers can do with these new capabilities
  - And the afternoon course...
This Afternoon Course

• “Beyond Programmable Shading: In Action”
  - Case studies from game developers, academics, and industry
This Afternoon’s Course

• “Beyond Programmable Shading: In Action”
  - Show-casing new interactive rendering algorithms that result in more realistic imagery than is possible using only the pre-defined DX/OpenGL graphics pipeline by
    - Combining task-, data-, and/or graphics pipeline parallelism,
    - Analyzing intermediate data produced by graphics pipeline,
    - Building and using complex data structures every frame, or
    - Modifying/extending the graphics pipelines
Speakers (in order of appearance)

- Aaron Lefohn, Intel
- Kayvon Fatahalian, Stanford
- Dave Luebke, NVIDIA
- Mike Houston, AMD
- Tom Forsyth, Intel
- John Owens, UC Davis
- Chas Boyd, Microsoft
- TBA, TBA
### Schedule

- **Intro**
  - Time: 8:30 - 8:40
  - Speaker: Lefohn

- **GPU Architectures**
  - Overview
    - Time: 8:40 - 8:55
    - Speaker: Fatahalian
  - NVIDIA
    - Time: 8:55 - 9:15
    - Speaker: Luebke
  - AMD
    - Time: 9:15 - 9:35
    - Speaker: Houston
  - Intel
    - Time: 9:35 - 9:55
    - Speaker: Forsyth

- **GPU Programming Models**
  - Overview
    - Time: 9:55 - 10:15
    - Speaker: Owens
  - Break
    - Time: 10:15 - 10:30

  - Brook+
    - Time: 10:30 - 10:50
    - Speaker: Houston
  - CUDA
    - Time: 10:50 - 11:10
    - Speaker: Luebke
  - DirectX
    - Time: 11:10 - 11:30
    - Speaker: Boyd
  - TBA
    - Time: 11:30 - 11:50
    - Speaker: TBA
  - Intel
    - Time: 11:50 - 12:10
    - Speaker: Lefohn

- **Q & A**
  - Time: 12:10 - 12:15+
Overview: Making Sense of GPU Architectures

Kayvon Fatahalian
Stanford University
GPUs are high throughput multi-core processors.
High throughput execution

Lots of compute resources
Multi-core
SIMD execution

Efficiently utilize resources
Multi-threading
A thread of execution

- A sequence of instructions executing within a processor context

Program counter

Registers

Memory mappings

Processing Core

Execution Context

ALU

Instruction Decode
Multi-core increases throughput

- Replicate resources and execute in parallel
SIMD processing

- Share instruction stream control logic across ALUs
Example: 4 cores, 4-wide SIMD

Core 1
- Exec Context
- Decode
- ALU
- ALU
- ALU

Core 2
- Exec Context
- Decode
- ALU
- ALU
- ALU

Core 3
- Exec Context
- Decode
- ALU
- ALU
- ALU

Core 4
- Exec Context
- Decode
- ALU
- ALU
- ALU
Thread stalls reduce throughput

- Instruction (pipeline) dependencies: a few cycles
- Off-chip memory access: 100’s – 1000’s of cycles
Multi-threading

- Each core maintains more thread execution contexts than it can simultaneously execute.
- Upon thread stall, core chooses another thread to execute.

Diagram:
- Processing Core
  - Instruction Decode
  - Exec Cxt
  - ALU

SIGGRAPH 2008
Multi-threading

• Latency hiding ability $\sim$ ratio of thread contexts to threads executable in a clock (T)

• On a GPU, T $\sim$ 10 to 100
Example:

• Running a fragment shader on a GPU core
Questions to ask about GPUs!

- How does architecture organize itself into multi-core, multi-threaded, and SIMD processing?
- How do kernels/shaders map to SIMD execution and multiple threads?
- How are instructions streams shared across kernels/threads?
A gamer wanders through a virtual world rendered in near-cinematic detail. Seconds later, the screen fills with a 3D explosion, the result of unseen enemies hiding in physically accurate shadows. Disappointed, the user exits the game and returns to a computer desktop that exhibits the stylish 3D look-and-feel of a modern window manager. Both of these visual experiences require hundreds of gigaflops of computing performance, a demand met by the GPU (graphics processing unit) present in every consumer PC.
The modern GPU is a versatile processor that constitutes an extreme but compelling point in the growing space of multicore parallel computing architectures. These platforms, which include GPUs, the STI Cell Broadband Engine, the Sun UltraSPARC T2, and, increasingly, multicore x86 systems from Intel and AMD, differentiate themselves from traditional CPU designs by prioritizing high-throughput processing of many parallel operations over the low-latency execution of a single task.

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

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

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

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

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

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

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

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

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

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

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

PO (pixel output). PO updates the output image with contributions from the fragments, accounting for surface visibility. In this example, \(p1\) is nearer to the camera than \(p0\). As a result \(p0\) is occluded by \(p1\).
the virtual camera, as well as values computed via interpolation of the source primitive’s vertex parameters.

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

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

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

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

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

CHARACTERISTICS AND CHALLENGES
Graphics pipeline execution is characterized by the following key properties.

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

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

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

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

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

**TABLE 1: Tale of the Tape: Throughput Architectures**

<table>
<thead>
<tr>
<th>Type</th>
<th>Processor</th>
<th>Cores/Chip</th>
<th>ALUs/Core</th>
<th>SIMD width</th>
<th>MaxT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs</td>
<td>AMD Radeon HD 2900</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>NVIDIA GeForce 8800</td>
<td>16</td>
<td>8</td>
<td>32</td>
<td>96</td>
</tr>
<tr>
<td>CPUs</td>
<td>Intel Core 2 Quad</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>STI Cell BE</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Sun UltraSPARC T2</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

1SSE processing only, does not account for x86 FPU.
2Stream processing (SPE) cores only, does not account for PPU cores.
332-bit, floating point (all ALUs are multiply-add except the Intel Core 2 Quad)
4The ratio of core thread contexts to simultaneously executable threads. We use the ratio T (rather than the total number of per-core thread contexts) to describe the extent to which processor cores automatically hide thread stalls via hardware multithreading.
ing an instruction across multiple threads with identical PCs. In either SIMD implementation, the complexity of processing an instruction stream and the cost of circuits and structures to control ALUs are amortized across multiple ALUs. The result is both power- and area-efficient chip execution.

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

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

GPUs execute groups of shader invocations in parallel to take advantage of SIMD processing. Dynamic per-entity control flow is implemented by executing all control paths taken by the shader invocations. SIMD operations that do not apply to all invocations, such as those within shader code conditional or loop blocks, are partially nullified using write-masks. In this implementation, when shader control flow diverges, fewer SIMD ALUs do useful work. Thus, on a chip with width-8 SIMD processing, worst-case behavior yields performance equaling 1/8 the chip's peak rate. Fortunately, shader workloads exhibit sufficient levels of instruction stream sharing to justify wide SIMD implementations. Additionally, GPU ISAs contain special instructions that make it possible for shader compilers to transform per-entity control flow into efficient sequences of SIMD operations.

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

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

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

Note that in the absence of stalls, the throughput of single- and multithreaded processors is equivalent. Multithreading does not increase the number of processing resources on a chip. Rather, it is a strategy that interleaves execution of multiple threads in order to use existing resources more efficiently (improve throughput). On average, a multithreaded core operating at its peak rate runs each thread $1/T$ of the time.
Large-scale multithreading requires execution contexts to be compact in order to fit many contexts within on-chip memories. The number of thread contexts supported by a GPU core is shader-program dependent and typically limited by the size of on-chip storage. GPUs require compiled shader binaries to declare input and output entity sizes, as well as bounds on temporary storage and scratch registers required for execution. At runtime, GPUs use these bounds to partition unspillable on-chip storage (including data registers) dynamically among execution contexts. Thus, GPUs support many thread contexts (up to an architecture-specific bound) and, correspondingly, provide maximal latency-hiding ability when shaders use fewer resources. When shaders require large amounts of storage, the number of execution contexts provided by a GPU drops. (The accompanying sidebar details an example of the efficient execution of a fragment shader on a GPU core.)

**FIXED-FUNCTION PROCESSING RESOURCES**

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

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

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

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

**THE MEMORY SYSTEM**

Parallel-processing resources place extreme load on a GPU’s memory system, which services memory requests from both fixed-function and programmable components. GPU memory systems are architected to deliver high-bandwidth, rather than low-latency, data access. These requests include a mixture of fine-granularity and bulk prefetch operations and may even require realtime guarantees (such as display scan out).

Recall that a GPU’s programmable cores tolerate large memory latencies via hardware multithreading and that interstage stream data accesses can be prefetched. As a result, GPU memory systems are architected to deliver high-bandwidth, rather than low-latency, data access. High throughput is obtained through the use of wide
memory buses and specialized GDDR (graphics double data rate) memories that operate most efficiently when memory access granularities are large. Thus, GPU memory controllers must buffer, reorder, and then coalesce large numbers of memory requests to synthesize large operations that make efficient use of the memory system. As an example, the ATI HD 2700XT memory controller manipulates thousands of outstanding requests to deliver 105 GB per second of bandwidth from GDDR3 memories attached to a 512-bit bus.

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

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

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

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

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

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

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

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

GPU hardware can take responsibility for sophisticated scheduling decisions because semantics and invariants of
Running a Fragment Shader on a GPU Core

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

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

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

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

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

As memory latencies on real GPUs involve hundreds of cycles, modern GPUs must contain support for significantly more threads to sustain high utilization. If we extend our simple GPU to a more realistic size of eight processing cores and provision each core with storage for 16 execution contexts, then simultaneous processing of 4,096 fragments is needed to approach peak processing rates. Clearly, GPU performance relies heavily on the abundance of parallel shading work.
the graphics pipeline are known a priori. Hardware implementation enables fine-granularity logic that is informed by precise knowledge of both the graphics pipeline and the underlying GPU implementation. As a result, GPUs are highly efficient at using all available resources. The drawback of this approach is that GPUs execute only those computations for which these invariants and structures are known.

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

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

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

**GPU AND CPU CONVERGENCE**

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

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

**LOVE IT, HATE IT? LET US KNOW**

feedback@acmqueue.com or www.acmqueue.com/forums

**KAYVON FATAHALIAN** is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming systems for commodity parallel architectures and computer graphics/animation systems for the interactive and film domains. His thesis research seeks to enable execution of more flexible rendering pipelines on future GPUs and multicore PCs. He will soon be looking for a job.

**MIKE HOUSTON** is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming models, algorithms, and runtime systems for parallel architectures including GPUs, Cell, multicore CPUs, and clusters. His dissertation includes the Sequoia runtime system, a system for programming hierarchical memory machines. He received his B.S. in computer science from UCSD in 2001 and is a recipient of the Intel Graduate Fellowship.

© 2008 ACM 1542-7730/08/0300 $5.00
FUTURE GRAPHICS ARCHITECTURES

WILLIAM MARK, INTEL AND UNIVERSITY OF TEXAS, AUSTIN
Graphics architectures are in the midst of a major transition. In the past, these were specialized architectures designed to support a single rendering algorithm: the standard Z buffer. Realtime 3D graphics has now advanced to the point where the Z-buffer algorithm has serious shortcomings for generating the next generation of higher-quality visual effects demanded by games and other interactive 3D applications. There is also a desire to use the high computational capability of graphics architectures to support collision detection, approximate physics simulations, scene management, and simple artificial intelligence. In response to these forces, graphics architectures are evolving toward a general-purpose parallel-programming...
model that will support a variety of image-synthesis algorithms, as well as nongraphics tasks.

This architectural transformation presents both opportunities and challenges. For hardware designers, the primary challenge is to balance the demand for greater programmability with the need to continue delivering high performance on traditional image-synthesis algorithms. Software developers have an opportunity to escape from the constraints of hardware-dictated image-synthesis algorithms so that almost any desired algorithm can be implemented, even those that have nothing to do with graphics. With this opportunity, however, comes the challenge of writing efficient, high-performance parallel software to run on the new graphics architectures. Writing such software is substantially more difficult than writing the single-threaded software that most developers are accustomed to, and it requires that programmers address challenges such as algorithm parallelization, load balancing, synchronization, and management of data locality.

The transformation of graphics hardware from a specialized architecture to a flexible high-throughput parallel architecture will have an impact far beyond the domain of computer graphics. For a variety of technical and business reasons, graphics architectures are likely to evolve into the dominant high-throughput “manycore” architectures of the future.

This article begins by describing the high-level forces that drive the evolution of realtime graphics systems, then moves on to some of the detailed technical trends in realtime graphics algorithms that are emerging in response to these high-level forces. Finally, it considers how future graphics architectures are expected to evolve to accommodate these changes in graphics algorithms and discusses the challenges that these architectures will present for software developers.

APPLICATIONS DRIVE EVOLUTION OF GRAPHICS ARCHITECTURES

To understand what form future graphics architectures are likely to take, we need to examine the forces that are driving the evolution of these architectures. As with any engineered artifact, graphics architectures are designed to deliver the maximum benefit to the end user within the fundamental technology constraints that determine what is affordable at a particular point in time. As VLSI (very large-scale integration) fabrication technology advances, the boundary of what is affordable changes, so that each generation of graphics architecture can provide additional capabilities at the same cost as the previous generation. Thus, the key high-level question is: What do we want these new capabilities to be?

Roughly speaking, graphics hardware is used for three purposes: 3D graphics, particularly entertainment applications (i.e., games); 2D desktop display, which used to be strictly 2D but now uses 3D capabilities for compositing desktops such as those found in Microsoft’s Vista and Apple’s Mac OS X; and video playback (i.e., decompression and display of streaming video and DVDs).

Although for most users desktop display and video playback are more important than 3D graphics, this article focuses on the needs of 3D graphics because these applications, with their significant demands for performance and functionality, have been the strongest force driving the evolution of graphics architectures.

Designing a graphics system for future 3D entertainment applications is particularly tricky because at a technical level the goals are ill defined. It is currently not possible to compute an image of the ideal quality at real-time frame rates, as evidenced by the fact that the images in computer-generated movies are of higher quality than those in computer games. Thus, designers must make approximations to the ideal computation. There are an enormous variety of possible approximations to choose...
from, each of which introduces a different kind of visual artifact in the image, and each of which uses different algorithms that may in turn run best on different architectures. In essence, the system design problem is reduced to the ill-specified problem of which system (software and hardware) produces the best-quality game images for a specific cost. Figure 1 illustrates this problem. In practice, there are also other constraints, such as backward compatibility and a desire to build systems that facilitate content creation.

As VLSI technology advances with time, the system designer is provided with more transistors. If we assume that the frame rate is fixed at 60 Hz, the additional computational capability provided by these transistors can be used in three fundamental ways: increasing the screen resolution; increasing the scene detail (polygon count or material shader complexity); and changing the overall approximations, by changing the basic rendering algorithm or specific components of it.

Looking back at the past six years, we can see these forces at work. Games have adopted programmable shaders that allow sophisticated modeling of materials and multipass techniques that approximate shadows, reflections, and other effects. Graphics architectures have enabled these changes through the addition of programmable vertex and fragment units, as well as more flexibility in how data moves between stages in the graphics pipeline.

Current graphics processors use the programming model illustrated in figure 2a. This model supports the traditional Z-buffer algorithm and is organized around a predefined pipeline structure that is only partially reconfigurable by the application. The predefined pipeline structure employs specialized hardware for the Z-buffer algorithm (in particular for polygon rasterization and Z-buffer read-modify-write operations), as well as for other operations such as the thread scheduling needed by the programmable stages.

Many of the individual pipeline stages are programmable (to support programmable material shading computations in particular), with all of the programmable stages multiplexed onto a single set of homogeneous programmable hardware processors. The programs executing within these pipeline stages, however, are heavily restricted in how they can communicate with each other and in how—if at all—they can access the global shared memory. This programming model provides high performance for the computations it is designed to support, but makes it difficult to support other computations efficiently.

It is important to realize that modern game applications fundamentally require programmability in the graphics hardware. This is because the real world contains an enormous variety of materials (wood, metal, glass, skin, fur, ...), and the only reasonable way to specify the

---

**Evolution of Graphics Programming Models**

*a. today’s graphics programming model*

- vertex program
- geometry program
- rasterizer
- fragment program
- output merger (ROP)
- 2 D video decode

*b. future graphics programming model*

- flexible multicore architecture
- specialized ISA extensions
- rasterizer
- texture unit
- 2 D video decode

---

more queue: www.acmqueue.com
interactions of these materials with light is to use a different program for each material.

This situation is very different from that found for other high-performance tasks, such as video decode, which does not inherently require programmable hardware; one could design fixed-function hardware sufficient to support the standard video formats without any programmability at all. As a practical matter most video-decode hardware does include some programmable units, but this is an implementation choice, not a fundamental requirement. This need for programmability by 3D graphics applications makes graphics architectures uniquely well positioned to evolve into more general high-throughput parallel computer architectures that handle tasks beyond graphics.

LIMITS OF THE TRADITIONAL Z-BUFFER GRAPHICS PIPELINE

The Z-buffer graphics pipeline with programmable shading that is used as the basis of today’s graphics architectures makes certain fundamental approximations and assumptions that impose a practical upper limit on the image quality. For example, a Z buffer cannot efficiently determine if two arbitrarily chosen points are visible from each other, as is needed for many advanced visual effects. A ray tracer, on the other hand, can efficiently make this determination. For this reason, computer-generated movies use rendering techniques such as ray-tracing algorithms and the Reyes (renders everything you ever saw) algorithm that are more sophisticated than the standard Z-buffer graphics pipeline.

Over the past few years, it has become clear that the next frontier for improved visual quality in real-time 3D graphics will involve modeling lighting and complex illumination effects more realistically (but not necessarily photo-realistically) so as to produce images that are closer in quality to those of computer-generated movies. These effects include hard-edged shadows (from small lights), soft-edged shadows (from large lights), reflections from water, and approximations to more complex effects such as diffuse lighting interactions that dominate most interior environments. There is also a desire to model effects such as motion blur and to use higher-quality anti-aliasing techniques. Most of these effects are challenging to produce with the traditional Z-buffer graphics pipeline.

Modern game engines (e.g., Unreal Engine 3, CryEngine 2) have begun to support some of these effects using today’s graphics hardware, but with significant limitations. For example, Unreal Engine 3 uses four different shadow algorithms, because no one algorithm provides an acceptable combination of performance and image quality in all situations. This problem is a result of limitations on the visibility queries that are supported by the traditional Z-buffer pipeline. Furthermore, it is common for different effects such as shadows and partial transparency to be mutually incompatible (e.g., partially transparent objects cast shadows as if they were fully opaque objects). This lack of algorithmic robustness and generality is a problem for both game-engine programmers and for the artists who create the game content. These limitations can also be viewed as violations of important principles of good system design such as abstraction (a capability should work for all relevant cases) and orthogonality (different capabilities should not interact in unexpected ways).

The underlying problem is that the traditional Z-buffer graphics pipeline was designed to compute visibility (i.e., the first surface hit) for regularly spaced rays originating at a single point (see figure 3a), but effects such as hard-edged shadows, soft-edged shadows, reflections, and diffuse lighting interactions all require more general visibility computations. In particular, reflections and diffuse lighting interactions require the ability to compute visible surfaces efficiently along rays with a variety of origins and directions (figure 3d). These types of visibility queries cannot be performed efficiently with the traditional graphics pipeline, but VLSI technology now provides enough transistors to support more sophisticated real-time visibility algorithms that can perform these queries efficiently. These transistors, however, must be organized into an architecture that can efficiently support the more sophisticated visibility algorithms.

Since the Z-buffer graphics pipeline is ill suited for producing the desired effects, the natural solution is to design graphics systems around more powerful visibility algorithms. Figure 3 provides an overview of some of these algorithms. I believe that these more powerful visibility algorithms will be gradually adopted over the next few years in response to the inadequacies of the standard Z buffer, although there is substantial debate in the graphics community as to how rapidly this change will occur. In particular, algorithms such as ray tracing...
are likely to be adopted much more rapidly in realtime graphics than they were in movie rendering, because realtime graphics does not permit the hand-tweaking of lighting for every shot that is common in movie rendering.

THE ARGUMENT FOR GENERAL-PURPOSE GRAPHICS HARDWARE

Given the desire to support more powerful visibility algorithms, graphics architects could take several approaches. Should the new visibility techniques be implemented in some kind of specialized hardware (like today's Z-buffer visibility computations), or should they be implemented in software on a flexible parallel architecture? I believe that a flexible parallel architecture is the best choice, because it supports the following software capabilities:

Mixing visibility techniques. Flexible hardware supports multiple visibility algorithms, ranging from the traditional Z buffer to ray tracing and beam tracing. Each application can choose the best algorithm(s) for its needs. The more sophisticated of these visibility algorithms require the ability to build and traverse irregular data structures such as KD-trees, which demands a more flexible parallel programming model than that used by today's GPUs.

Application-tailored approximations. Rendering images at realtime frame rates requires making mathematical approximations (e.g., for particular lighting effects), but the variety of possible approximations is enormous. Often, different approximations use very different overall rendering algorithms and have very different performance characteristics. Since the best approximation and algorithm vary from application to application and sometimes even within an application, an architecture that allows the application to choose its approximations can provide far greater efficiency for the overall rendering task than an architecture that lacks this flexibility.

Integration of rendering with scene management. Traditionally, realtime graphics systems have used one set of data structures to represent the persistent state of the scene (e.g., object positions, velocities, and groupings) and a different set of data structures to compute visibility. The two sets of data structures are on opposite sides of an intervening API such as DirectX or OpenGL. For every frame, all of the visible geometry is transferred across this API. In a Z-buffer system this approach works because it is relatively straightforward to determine which geometry might be visible. In a ray-tracing system, however, this approach does not work very well, and it is desirable to integrate the two sets of data structures more tightly, with both residing on the graphics processor (figure 4). It is also desirable to change the traditional layering of APIs so that the game engine takes over most of the low-level rendering tasks currently handled by graphics hardware (figure 5). A highly programmable architecture makes it much easier to do this integration while still preserving flexibility for the application to maintain the persistent...
data structures in the most efficient manner. It also allows scene management computations to be performed on the high-performance graphics hardware, eliminating a bottleneck on the CPU.

**Support for game physics and AI.** A flexible parallel architecture can easily support computations such as collision detection, fluid dynamics simulations (e.g., for explosions), and artificial intelligence for game play. It also allows these computations to be tightly integrated with the rendering computation.

**Rapid innovation.** Software can be changed more rapidly than hardware, so a flexible parallel architecture that uses software to express its graphics algorithms enables more rapid innovation than traditional designs.

The best choice for the system as a whole is to use flexible parallel hardware that permits software to use aggressive algorithmic specialization and optimization, rather than to use specialized parallel hardware that mandates a particular algorithm.

**PROGRAMMING MODEL**
When I say that future graphics architectures are likely to support an extremely flexible parallel programming model, what do I mean? There is considerable debate within the graphics hardware community as to the specific programming model that graphics architectures should adopt in the near future. I expect that in the short term each of the major graphics hardware companies will take a somewhat different path. There are a variety of reasons for this diversity: different emphasis placed on adding new capabilities versus improving performance of the old programming models; fundamental philosophical differences in tackling the parallel programming problem; and the desire by some companies to evolve existing designs incrementally.

In the longer term (five years or so), the programming models will probably converge, but there is not yet a consensus on what such a converged programming model would look like. This section presents some of the key issues that today’s graphics architects face, as well as thoughts on what a converged future programming model could look like and the challenges that it
will present for programmers. Most of the programming challenges discussed here will be applicable to all future graphics architectures, even those that are somewhat different from the one I am expecting.

END OF THE HARDWARE-DEFINED PIPELINE

Graphics processors will evolve toward a programming model similar to that illustrated in figure 2b. User-written software specifies the overall structure of the computation, expressed in an extremely flexible parallel programming model similar to that used to program today’s multicore CPUs. The user-written software may optionally use specialized hardware to accelerate specific tasks such as texture mapping. The specialized hardware may be accessed via a combination of instructions in the ISA (instruction set architecture), special memory-mapped registers, and special inter-processor messages.

The latest generation of GPUs (graphics processing units) from NVIDIA and AMD have already taken a significant step toward this future graphics programming model by supporting a separate programming model for nongraphics computations that is more flexible than the programming model used for graphics. This second programming model is an assembly-level parallel-programming model with some capabilities for fine-grained synchronization and data sharing across hardware threads. NVIDIA calls its model PTX (Parallel Thread Execution), and AMD's is known as CTM (Close to Metal). Note that NVIDIA’s C-like CUDA language (see “Scalable Parallel Programming with CUDA” in this issue) is a layer on top of the assembly-level PTX. It is important to realize, however, that PTX and CTM have some significant limitations compared with traditional general-purpose parallel programming models. PTX and CTM are still fairly restrictive, especially in their memory and concurrency models.

These limitations become obvious when comparing PTX and CTM with the programming models supported by other single-chip highly parallel processors, such as Sun’s Niagara server chips. I believe that the programming model of future graphics architectures will be substantially more flexible than PTX and CTM.

TASK PARALLELISM AND MULTITHREADING

The parallelism supported by current GPUs primarily takes the form of data parallelism—that is, the GPU operates simultaneously on many data elements (such as vertices or pixels or elements in an array). In contrast, task parallelism is not supported well, except for the specific case of concurrent processing of pixels and vertices. Since better support for task parallelism is necessary to support user-defined rendering pipelines efficiently, I expect that future GPUs will support task parallelism much more aggressively. In particular, multiple tasks will be able to execute asynchronously from each other and from the CPU, and will be able to communicate and synchronize with each other. These changes will require a substantially more sophisticated software runtime environment than the one used for today’s GPUs and will introduce significant complexity into the hardware/software interactions for thread management.

As with today’s GPUs and Sun’s Niagara processor, each core will use hardware multithreading, possibly augmented by additional software multithreading along the lines of that used by programmers of the Cell architecture. This multithreading serves two purposes:

• First, it allows the core to remain fully utilized even if each individual instruction has a pipeline latency of several cycles—the core just executes an instruction from another thread.

• Second, it allows the core to remain fully utilized even if one or more of the threads on the core stalls because of an off-chip DRAM access such as those that occur when fetching data from a texture. Programmers will face the challenge of exposing parallelism for multiple cores and for multiple threads on each core. This challenge is already starting to appear with programming models such as NVIDIA’s CUDA.

SIMD EXECUTION WITHIN EACH CORE

An important concern in the design of graphics hardware is obtaining the maximum possible performance using a fixed number of transistors on a chip. If one instruction cache/fetch/decode unit can be shared among several arithmetic units, the die area and power requirements of the hardware are reduced, as compared with a design that has one instruction unit per arithmetic unit. That
is, a SIMD (single instruction, multiple data) execution model increases efficiency as long as most of the elements in the SIMD vectors are kept active most of the time. A SIMD execution model also provides a simple form of fine-grained synchronization that helps to ensure that memory accesses have good locality.

Current graphics hardware uses a SIMD execution model, although it is sometimes hidden from the programmer behind a scalar programming interface as in NVIDIA's hardware. One area of ongoing debate and change is likely to be in the underlying hardware SIMD width; there is a tension between the efficiency gained for regular computations as SIMD width increases and the efficiency gained for irregular computations as SIMD width decreases. NVIDIA GPUs (GeForce 8000 and 9000 series) have an effective SIMD width of 32, but the trend has been for the SIMD width of GPUs to decrease to improve the efficiency of algorithms with irregular control flow.

There is also debate about how to expose the SIMD execution model. It can be directly exposed to the programmer with register-SIMD instructions, as is done with x86 SSE instructions, or it may be nominally hidden from the programmer behind a scalar programming model, as is the case with NVIDIA's GeForce 9000 series. If the SIMD execution model is hidden, the conversion from the scalar programming model to the SIMD hardware may be performed by either the hardware (as in the GeForce 9000 series) or a compiler or some combination of the two. Regardless of which strategy is used, programmers who are concerned with performance will need to be aware of the underlying SIMD execution model and width.

SMALL AMOUNTS OF LOCAL STORAGE
One of the most important differences between GPUs and CPUs is that GPUs devote a greater fraction of their transistors to arithmetic units, whereas CPUs devote a greater fraction of their transistors to cache. This difference is one of the primary reasons that the peak performance of a GPU is much higher than that of a CPU.

I expect that this difference will continue in the future. The impact on programmers will be significant: although the overall programming model of future GPUs will become much closer to that of today's CPUs, programmers will need to manage data locality much more carefully on future GPUs than they do on today's CPUs.

This problem is made even more challenging by multithreading; if there are $N$ threads on each core, the amount of local storage per thread per core is effectively $1/N$ of the core's total local storage. This issue can be mitigated if the $N$ threads on a core are sharing a working set, but to do this the programmer must think of the $N$ threads as being closely coupled to each other. Similarly, programmers will have to think about how to share a working set across threads on different cores.

These considerations are already becoming apparent with CUDA. The constraints are likely to be frustrating to programmers who are accustomed to the large caches of CPUs, but they need to realize that extra local storage would come at the cost of fewer ALUs (arithmetic logic units), and they will need to work closely with hardware designers to determine the optimum balance between cache and ALUs.

CACHE-COHERENT SHARED MEMORY
The most important aspect of any parallel architecture is its overall memory and communication model. To illustrate the importance of this aspect of the design, consider four (of many) possible alternatives (of course, hybrids and enhancements of these models are possible):

- A message-passing architecture, in which each processor core has its own memory space and all communication occurs through explicit message passing. Most large-scale supercomputers (those with 100-plus processors) use this model.
- An architecture such as the Sony/Toshiba/IBM Cell with a noncached, noncoherent shared memory. In such an architecture, all transfers of data between a core's small private memory and the global memory must be orchestrated through explicit memory-transfer commands.
- An architecture such as NVIDIA's GeForce 8800 with what amounts to a minimally cached, noncoherent shared memory, with support for load/store to this memory.
- An architecture such as modern multicore CPUs, with cached, coherent shared memory. In such architectures, hardware mechanisms manage transfer of data between cache and main memory and ensure that data in caches of different processors remains consistent.

There is considerable debate within the graphics architecture community as to which memory and communication model would be best for future architectures, and
in the near term different hardware vendors are taking different approaches. Software programmers should think carefully about these issues so that they are prepared to influence the debate.

Which approach is most likely to dominate in the medium to long term? I have previously argued that the trend in rendering algorithms is toward those that build and traverse irregular data structures. These irregular data structures allow algorithms to adapt to the scene geometry and the current viewpoint. Explicitly managing all data locality for these algorithms is painful, especially if multiple cores share a read/write data structure. In my experience, it is easier to develop these algorithms on a cache-coherent architecture, even if achieving optimal performance often still requires thinking very carefully about the communication and memory-access patterns of the performance-critical kernels.

For these and other reasons too detailed to discuss here, I believe that future graphics architectures will efficiently support a cache-coherent memory model, and that any architecture lacking these capabilities will be a second choice at best for programmers who are developing innovative rendering techniques. Sun’s Niagara architecture provides a good preview of the kind of memory and threading model that I anticipate for future GPUs. I also expect, however, that cache-coherent graphics architectures will include a variety of mechanisms that provide the programmer with explicit control over communication and memory access, such as streaming loads that bypass the cache.

FINE-GRAINED SPECIALIZATION
The desire to support greater algorithmic diversity will drive future graphics architectures toward greater flexibility and generality, but specialization will still be used where it provides a sufficiently large benefit for the majority of applications. Most of this specialization will be at a fine granularity, used to accelerate specific operations, in contrast to the coarse, monolithic granularity used to dictate the overall structure of the algorithms executed on the hardware in the past.

In particular, I expect the following specialization will continue to exist for graphics architectures:

**Texture hardware.** Texture addressing and filtering operations use low-precision (typically 16-bit) values that are decompressed on the fly from a compressed representation stored in memory. The amount of data accessed is large and requires multithreading to deal effectively with cache misses. These operations are a significant fraction of the overall rendering cost and benefit enormously from specialized hardware.

**Specialized floating-point operations.** Rendering makes heavy use of floating-point square-root and reciprocal operations. Current graphics hardware provides high-performance instructions for these operations, as well as other operations used for shading such as swizzling and trigonometric functions. Future graphics hardware will need to do the same.

**Video playback and desktop compositing.** Video playback and 2D and 2.5D desktop window operations benefit significantly from specialized hardware. Specialization of these operations is especially important for power efficiency. I anticipate that much of this hardware will follow the traditional coarse-grained monolithic fixed-function model and thus will not be useful for user-written 3D graphics programs.

Current graphics hardware also includes specialized hardware to assist with triangle rasterization, but I expect that this task will be taken over by software within a few years. The reason is that rasterization is gradually becoming a smaller fraction of total rendering costs, so the penalty for implementing it in software is decreasing. This trend will accelerate as more sophisticated visibility algorithms supplement or replace the Z buffer.

As graphics software switches to more powerful visibility algorithms such as ray tracing, it may become clear that certain operations represent a sufficiently large portion of the total computation cost that hardware acceleration would be justified. For example, future architectures could include specialized instructions to accelerate the data-structure traversal operations used by ray tracing.

THE CHALLENGE FOR GRAPHICS ARCHITECTS
At a high level, the key challenge facing future graphics architectures is to strike the best balance between the desire to provide high performance on existing graphics algorithms and the desire to provide the flexibility needed to support new algorithms with high perfor-
mance, including non-graphics algorithms and the next generation of more capable and sophisticated graphics algorithms. I believe that the opportunity for improved visual quality and robustness provided by more sophisticated graphics algorithms will cause the transition to more flexible architectures to happen relatively rapidly, an opinion that remains a matter of debate within the graphics architecture community.

THE FUTURE OF GRAPHICS ARCHITECTURES
In the past, graphics architectures defined the algorithms used for rendering and their performance. In the future, graphics architectures will cease to define the rendering algorithms and will simply set the performance and power efficiency limits within which software developers may do whatever they want.

For the programmer, future graphics architectures are likely to be very similar to today’s multicore CPU architectures, but with greater SIMD instruction widths and the availability of specialized instructions and processing units for some operations. Like today’s Niagara processor, however, the amount of cache per processor core will be relatively small. To achieve peak performance, programmers will have to think more carefully about memory-access patterns and data-structure sizes than they have been accustomed to with the large caches of modern CPUs.

Future graphics architectures will enable a golden age of innovation in graphics; I expect that over the next few years we will see the development of a variety of new rendering algorithms that are more efficient and more capable than the ones used in the past. For computer games, these architectures will allow game logic, physics simulation, and AI to be more tightly integrated with rendering than before. For data-visualization applications, these architectures will allow tight integration of domain-specific data analysis with the rendering computations used to display the results of this analysis. The general-purpose nature of these architectures combined with the low cost enabled by their high-volume market will also cause them to become the preferred platform for almost all high-performance floating-point computations.

ACKNOWLEDGMENTS AND FURTHER READING
Don Fussell, Kurt Akeley, Matt Pharr, Pat Hanrahan, Mark Horowitz, Stephen Junkins, and several graphics hardware architects contributed directly and indirectly to the ideas in this article through many fun and productive discussions. More details about many of the ideas discussed in this article can be found in another article I wrote with Don Fussell in 2005.4 The tendency of graphics hardware to become increasingly general until the temptation emerges to incorporate new specialized units has existed for a long time and was described in 1968 as the “wheel of reincarnation” by Myer and Sutherland.5 The fundamental need for programmability in realtime graphics hardware, however, is much more important now than it was then.

REFERENCES

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

BILL MARK leads Intel’s advanced graphics research lab. He is on leave from the University of Texas at Austin, where until January 2008 he led a research group that investigated future graphics algorithms and architectures. In 2001-2002 he was the technical leader of the team at NVIDIA that co-designed (with Microsoft) the Cg language for programmable graphics hardware and developed the first release of the NVIDIA Cg compiler. His research interests focus on systems and hardware architectures for realtime computer graphics and on the opportunity to extend these systems to support more general parallel computation and a broader range of graphics algorithms, including interactive ray tracing.

© 2008 ACM 1542-7730/08/0300 $5.00
What is a programming model?

**Specification model** (in domain of the application)

**Programming model**

**Computational model** (representation of computation)

**Cost model** (how computation maps to hardware)

- **Is a programming model a language?**
  - Programming models allow you to express ideas in particular ways
  - Languages allow you to put those ideas into practice
Writing Parallel Programs

• **Identify** concurrency in problem
  - Do this in your head

• **Expose** the concurrency when writing the code to solve the problem
  - Choose a programming model and language that allow you to express this concurrency

• **Exploit** the concurrency in the problem
  - Choose a language and hardware that together allow you to take advantage of the concurrency
The Graphics Pipeline (simplified)

Vertex processing

Fragment processing

These two stages can run in parallel.

Within each of these stages, different data elements can run in parallel.
Threads / Units of Execution

- Complex applications can be broken down into individual “units of execution” (UEs) or “threads”
  - “Thread” is a loaded word but we’ll use it today
  - Programmer’s first task is to identify the concurrency between threads

A graphics “vertex processing” stage processes many vertices.
The Graphics Pipeline (simplified)

- **Vertex processing**
  - How do we express just one stage of this program?
  - How do we express parallelism within a stage? How do we express parallelism between stages?
  - Is the parallelism within a stage all controlled in lockstep, or is it more flexible?
- **Fragment processing**
  - Do we think of this as parallel tasks or as a pipeline?
Kinds of Parallelism

- Vertex processing
- Fragment processing
Task Parallelism (1)

We may be able to process multiple stages at the same time. Here these are structured as a pipeline, but don’t have to be.

We can have different hardware or cores work on different pipelines at the same time (task-parallel hardware) or timeslice on one piece of hardware (time-multiplexed).
Task Parallelism (2)

Vertex processing

Fragment processing

Graphics

Shadow

AI

Physics

Beyond Programmable Shading: Fundamentals
We may be able to process multiple data elements at the same time.

Are these elements all running the exact same code? The same program, but taking different directions? Different programs?
Algorithm Structure Design Space

Start

Organize By Flow of Data

Organize By Tasks

Organize By Data

Regular?

Irregular?

Linear?

Recursive?

Pipeline

Event-Based Coordination

Task Parallelism

Divide and Conquer

Geometric Decomposition / Data Parallel

Recursive Data

Beyond Programmable Shading: Fundamentals

• Courtesy Tim Mattson, Intel
**Terminology: SIMD, SPMD, MIMD**

- **Models for exploiting data parallelism**
  - Many items can be processed in parallel
  - MD = “multiple data”

- **Are multiple threads processed ...**
  - in lockstep? [SIMD: Single Instruction, Multiple Data]
    - GPU model: early fragment programs
  - by the same program, but not in lockstep? [SPMD: Single Program, Multiple Data]
    - GPU example: CUDA
  - by different programs? [MIMD: Multiple Instruction, Multiple Data]
    - GPU example: vertex programs
Terminology: Streaming

- In a streaming programming model,
  - Data structure: streams (list of data items)
  - Algorithm: kernel (operates on streams)
- Streams have *implicit* parallelism
- Limited visibility, $O(1)$ storage
- Explicit data access pattern
- GPU example:
  - Vertex streams
  - Brook (which had other features as well)
Terminology: Traditional Coprocessor

- **Is the parallel processor controlled by another processor ...**
  - ... that is responsible for calling tasks ...
  - ... or allocating/deallocating memory ...
  - This is the traditional coprocessor model

- **... or is the parallel processor responsible for its own global control?**
  - Allocates its own memory, calls its own kernels
  - Can submit work to itself

- **Coprocessors are adding features that are blurring these lines**
Bibliography

• Mattson/Sanders/Massingill, *Patterns for Parallel Programming*, Addison Wesley 2004

• Buck et al., “Brook for GPUs: Stream Computing on Graphics Hardware”, Siggraph 2004

• Lindholm et al., “A User-Programmable Vertex Engine”, Siggraph 2001
Data-Parallel Computing

CHAS. BOYD, MICROSOFT
Users always care about performance. Although often it’s just a matter of making sure the software is doing only what it should, there are many cases where it is vital to get down to the metal and leverage the fundamental characteristics of the processor.

Until recently, performance improvement was not difficult. Processors just kept getting faster. Waiting a year for the customer’s hardware to be upgraded was a valid optimization strategy. Nowadays, however, individual processors don’t get much faster; systems just get more of them.

Much comment has been made on coding paradigms to target multiple-processor cores, but the data-parallel paradigm is a newer approach that may just turn out to be easier to code to, and easier for processor manufacturers to implement.

This article provides a high-level description of data-parallel computing and some practical information on how and where to use it. It also covers data-parallel programming environments, paying particular attention to those based on programmable graphics processors.

A BIT OF BACKGROUND

Although the rate of processor-performance growth seems almost magical, it is gated by fundamental laws of physics. For the entire decade of the ’90s, these laws enabled processors to grow exponentially in performance as a result of improvements in gates-per-die, clock speed, and instruction-level parallelism. Beginning in 2003, though, the laws of physics (power and heat) put an end to growth in clock speed. Then the silicon area requirements
for increasingly sophisticated ILP (instruction-level parallelism) schemes (branch prediction, speculative execution, etc.) became prohibitive. Today the only remaining basis for performance improvement is gate count.

Recognizing this, manufacturers have restructured to stop pushing clock rate and focus on gate count. Forecasts project that gates-per-die can double every two years for the next six to eight years at least. What do you do with all those gates? You make more cores. The number of cores per die will therefore double every two years, resulting in four times today’s core counts (up to 32 cores) by 2012.

Customers will appreciate that growth rate, but they will benefit only if software becomes capable of scaling across all those new cores. This is the challenge that performance software faces in the next five to ten years. For the next decade, the limiting factor in software performance will be the ability of software developers to restructure code to scale at a rate that keeps up with the rate of core-count growth.

PARALLEL PROGRAMMING

Parallel programming is difficult. We deprecate the use of GOTO statements in most languages, but parallel execution is like having them randomly sprinkled throughout the code during execution. The assumptions about order of execution that programmers have made since their early education no longer apply.

The single-threaded von Neumann model is comprehensible because it is deterministic. Parallel code is subject to errors such as deadlock and livelock, race conditions, etc. that can be extremely subtle and difficult to identify, often because the bug is nonrepeatable. These issues are so severe that despite decades of effort and dozens of different approaches, none has really gained significant adoption or even agreement that it is the best solution to the problem.

An equally subtle challenge is performance scaling. Amdahl’s law states that the maximum speedup attainable by parallelism is the reciprocal of the proportion of code that is not parallelizable. If 10 percent of a given code base is not parallel, even on an infinite number of processors it cannot attain more than a tenfold speedup.

Although this is a useful guideline, determining how much of the code ends up running in parallel fashion is very difficult. Serialization can arise unexpectedly as a result of contention for a shared resource or requirements to access too many distant memory locations.

The traditional methods of parallel programming (thread control via locks, message-passing interface, etc.) often have limited scaling ability because these mechanisms can require serialization phases that actually increase with core count. If each core has to synchronize with a single core, that produces a linear growth in serial code, but if each core has to synchronize with all other cores, there can be a combinatoric increase in serializa-

The single-threaded von Neumann model is comprehensible because it is deterministic. Parallel code is subject to errors such as deadlock and livelock, race conditions, etc. that can be extremely subtle and difficult to identify, often because the bug is nonrepeatable. These issues are so severe that despite decades of effort and dozens of different approaches, none has really gained significant adoption or even agreement that it is the best solution to the problem.

An equally subtle challenge is performance scaling. Amdahl’s law states that the maximum speedup attainable by parallelism is the reciprocal of the proportion of code that is not parallelizable. If 10 percent of a given code base is not parallel, even on an infinite number of processors it cannot attain more than a tenfold speedup.

Although this is a useful guideline, determining how much of the code ends up running in parallel fashion is very difficult. Serialization can arise unexpectedly as a result of contention for a shared resource or requirements to access too many distant memory locations.

The traditional methods of parallel programming (thread control via locks, message-passing interface, etc.) often have limited scaling ability because these mechanisms can require serialization phases that actually increase with core count. If each core has to synchronize with a single core, that produces a linear growth in serial code, but if each core has to synchronize with all other cores, there can be a combinatoric increase in serialization.

After all, any code that serializes is four times slower on a four-core machine, but 40 times slower on a 40-core machine.

Another issue with performance scaling is more fundamental. A common approach in multicore parallel programming for games is to start with a top-down breakdown. Relatively isolated subsystems are assigned to separate cores, but what happens once the number of subsystems in the code base is reached? Since restructuring code at this level can be pervasive, it often requires a major rewrite to break out subsystems at the next finer level, and again for each hardware generation.

For all these reasons, transitioning a major code base to parallel paradigms is time-consuming. Getting all the subtle effects of nondeterminism down to an acceptable level can take years. It is likely that by that time, core-count growth will have already exceeded the level of parallelism that the new code structure can scale to. Unfortunately, the rate of core-count growth may be outstripping our ability to adapt to it.

Thus, the time has come to look for a new paradigm—ideally one that scales with core count but without requiring restructuring of the application architecture every time a new core count is targeted. After all, it’s not about choosing a paradigm that operates well at a fixed core count; it’s about choosing one that continues to scale with an increasing number of cores without requiring code changes. We need to identify a finer level of granularity for parallelism.
DATA-PARALLEL PROGRAMMING

Given the difficulty of finding enough subsystem tasks to assign to dozens of cores—the only elements of which there are a comparable number are data elements—the data-parallel approach is simply to assign an individual data element to a separate logical core for processing. Instead of breaking code down by subsystems, we look for fine-grained inner loops within each subsystem and parallelize those.

For some tasks, there may be thousands to millions of data elements, enabling assignment to thousands of cores. (Although this may turn out to be a limitation in the future, it should enable code to scale for another decade or so.) For example, a modern GPU can support hundreds of ALUs (arithmetic logic units) with hundreds of threads per ALU for nearly 10,000 data elements on the die at once.

The history of data-parallel processors began with the efforts to create wider and wider vector machines. Much of the early work on both hardware and data-parallel algorithms was pioneered at companies such as MasPar, Tera, and Cray.

Today, a variety of fine-grained or data-parallel programming environments are available. Many of these have achieved recent visibility by supporting GPUs. They can be categorized as follows:

Older languages (C*, MPL, Co-Array Fortran, Cilk, etc.). Several languages have been developed for fine-grained parallel programming and vector processing. Many add only a very small difference in syntax from well-known languages. Few of them support a variety of platforms and they may not be available commercially or be supported long term as far as updates, documentation, and materials.

Newer languages (XMT-C, CUDA, CAL, etc.). These languages are being developed by the hardware company involved and therefore are well supported. They are also very close to current C++ programming models syntactically; however, this can cause problems because the language then provides no explicit representation of the unique aspects of data-parallel programming or the processor hardware. Although this can reduce the changes required for an initial port, the resulting code hides the parallel behavior, making it harder to comprehend, debug, and optimize. Simplifying the initial port of serial code through syntax is not that useful to begin with, since for best performance it is often an entire algorithm that must be replaced with a data-parallel version. Further, in the interest of simplicity, these APIs may not expose the full features of the graphics-specific silicon, which implies an underutilized silicon area.

Array-based languages (RapidMind, Acceleware, Microsoft Accelerator, Ct, etc.). These languages are based on array data types and specific intrinsics that operate on them. Algorithms converted to these languages often result in code that is shorter, clearer, and very likely faster than before. The challenge of restructuring design concepts into array paradigms, however, remains a barrier to adoption of these languages because of the high level of abstraction at which it must be done.

Graphics APIs (OpenGL, Direct3D). Recent research in GPGPU (general-purpose computing on graphics processing units) has found that while the initial ramp-up of using graphics APIs can be difficult, they do provide a direct mapping to hardware that enables very specific optimizations, as well as access to hardware features that other approaches may not allow. For example, work by Naga Govindaraju¹ and Jens Krüger² relies on access to fixed-function triangle interpolators and blending units that the newer languages mentioned here often do not expose. Further, there is good commercial support and a large and experienced community of developers already using them.

GPUS AS DATA-PARALLEL MACHINES

The GPU is the second-most-heavily used processor in a typical PC. It has evolved rapidly over the past decade to reach performance levels that can exceed the CPU by
a large factor, at least on appropriate workloads. GPU evolution has been driven by 3D rendering, an embarrassingly data-parallel problem, which makes the GPU an excellent target for data-parallel code. As a result of this significantly different workload design point (processing model, I/O patterns, and locality of reference), the GPU has a substantially different processor architecture and memory subsystem design, typically featuring a broader SIMD (single instruction, multiple data) width and a higher-latency, higher-bandwidth streaming memory system. The processing model exposed via a graphics API is a task-serial pipeline made up of a few data-parallel stages that use no interthread communication mechanisms at all. While separate stages appear for processing vertices or pixels, the actual architecture is somewhat simpler.

As shown in figure 1, a modern DirectX10-class GPU has a single array of processors that perform the computational work of each stage in conjunction with specialized hardware. After polygon-vertex processing, a specialized hardware interpolator unit is used to turn each polygon into pixels for the pixel-processing stage. This unit can be thought of as an address generator. At the end of the pipeline, another specialized unit blends completed pixels into the image buffer. This hardware is often useful in accumulating results into a destination array. Further, all processing stages have access to a dedicated texture-sampling unit that performs linearly interpolated reads on 1D, 2D, or 3D source arrays in a variety of data-element formats.

Shaped by these special workload requirements, the modern GPU has:

- Ten times the GFLOPS of CPU chips for similar price and power consumption
- Thousands of threads distributed over hundreds of single-precision floating-point ALUs
- A dedicated streaming-memory system with 10 times the memory bandwidth of a CPU
- Dedicated memory capacity similar to the CPU system memory capacity
- Specialized cores for filtering, blending, rasterizing, and video processing

A GPU’s memory subsystem is designed for higher I/O latency to achieve increased throughput. It assumes only very limited data reuse (locality in read/write access), featuring small input and output caches designed more as FIFO (first in, first out) buffers than as mechanisms to avoid round-trips to memory.

Recent research has looked into applying these processors to other algorithms beyond 3D rendering. There have been applications that have shown significant benefits over CPU code. In general, those that most closely match the original design workload of 3D graphics (such as image processing) and can find a way to leverage either the tenfold compute advantage or the tenfold bandwidth advantage have done well. (Much of this work is cataloged on the Web at http://www.gpgpu.org.)

This research has identified interesting algorithms. For example, compacting an array of variable-length records is a task that has a data-parallel implementation on the parallel prefix sum or scan. The prefix-sum algorithm computes the sum of all previous array elements (i.e.,
the first output element in a row \( r \) is \( r_0 \), while the second
is \( o_2 = r_0 + r_1 \), and the \( n \)th output element is \( o_n = r_0 + r_1 + \ldots + r_n \). Using this, a list of record sizes can be accu-
mulated to compute the absolute addresses where each
record element is to be written. Then the writes can occur
completely in parallel. Note that if the writes are done
in order, the memory-access pattern is still completely
sequential.4

**MAKING CODE DATA-PARALLEL**

Before starting to write your code, check for tasks that
are known data-parallel cases. Often you can find library
routines already available for accelerating common tasks
using data-parallel hardware. Most data-parallel program-
ing environments include such libraries as a convenient
way for users to begin adopting their technology.

If you need to write custom data-parallel code, the
process is similar to a localized optimization effort. You
can adopt data-parallel programming incrementally, since
you can identify and optimize the key inner loops one at
a time, without perturbing the larger-scale structure of the
code base. Here are the basic steps for converting code to
the data-parallel model:

1. Identify a key task that looks data-parallel.
2. Identify a data-parallel algorithm for this task.
3. Select a data-parallel programming environment.
4. Implement code.
5. Evaluate performance scaling rate.
6. Go to step 1.

**STEP 1: IDENTIFY A KEY TASK THAT
LOOKS DATA-PARALLEL**

Look for a segment of code that doesn’t rely greatly on
cross communication between data elements, or con-
versely, a set of data elements that can be processed
without requiring too much knowledge of each other.
Look for data-access patterns that can be regularized, as
opposed to arbitrary/random (such as linear arrays versus
sparse-tree data structures).

While searching for candidates to parallelize, you
can evaluate performance potential via Amdahl’s law:
just comment out this candidate task (simulate infinite
parallelism) and check to see total performance change. If
there isn’t a significant improvement, going through the
effort of parallelizing won’t pay off.

**STEP 2: IDENTIFY A DATA-PARALLEL
ALGORITHM FOR THIS TASK**

Often a good place to look is in the history books (math)
or in routines developed by Tera/Cray for its vector
processors. For example, bitonic sorts were identified as
interesting before computers were developed, but fell out
of favor during the rise of current cache-based machines.
Other examples are radix sorts, and prefix sum (scan)
operations used for packing sparse data.

**STEP 3: SELECT A DATA-PARALLEL
PROGRAMMING ENVIRONMENT**

Many data-parallel programming environments are avail-
able today. Many of the criteria to use in evaluation are
the same as for any development environment. The areas
to look for are:

- **Abstraction level.** Do you need a library, a set of data-
  abstraction utilities, or a language?
- **Syntax clarity.** Are limitations of the implementation
  explicit in the syntax or hidden by it?
- **Maintainability.** Would the resulting code complexity be
  manageable?
- **Support.** Are there user groups or support services?
- **Availability.** How broadly distributed is the environment
  or any hardware that it requires?
- **Compatibility.** Is the environment compatible with a
  broad range of systems or only a specific subset?
- **Lifespan.** Is the environment compatible with future
  hardware, even from the same vendor?
- **Documentation.** Do the docs make sense? Are the
  samples useful?
- **Cost.** How much will it cost users of your product to get
  any required hardware or software?

**STEP 4: IMPLEMENT CODE**

Code it up, at least at the pseudocode level. If implemen-
tation turns out to require more than one or two places
where interthread communication is required, then this
may not be a sufficiently data-parallel algorithm. In that
case, it may be necessary to look for another algorithm
(step 2) or another task to parallelize (step 1).

**STEP 5: EVALUATE PERFORMANCE SCALING**

Performance at a given core count is interesting but not
the key point. (If you are going to check that, be sure to
compare using a realistic “before” case.) A more impor-
tant metric to check is how the new code scales with
increasing core count. If there is no sign of a performance
plateau, the system will have some scaling headroom.
After all, absolute performance relative to a single core is
not as relevant as how it scales with core-count growth
over time.
In summary:

- **Understand the paradigm.** What are data-parallel computation and the streaming-memory model?
- **Understand your code.** Which portions operate at which level of granularity?
- **Understand the environment.** How does it help solve the problem?

**GPU PERFORMANCE HINTS**

If targeting a GPU, are there operations that can leverage the existing graphics-related hardware? Are your data types small enough? GPUs are designed to operate on small data elements so media data (image/video pixels or audio samples) is a good fit. Or when sorting on the GPU, working with key-index pairs separately is often a win. Then the actual movement of data records can be done on the CPU, or on the GPU as a separate pass.

GPUs are optimized for work with 1D, 2D, or 3D arrays of similar data elements. Array operations are often faster using GPU hardware because it can transparently optimize them for spatially coherent access.

When reading such arrays, the GPU can easily linearly interpolate regular array data. This effectively enables a floating-point (fuzzy) array index. Many mathematical algorithms use either a simple linear interpolation of array elements or slightly higher-order schemes that can be implemented as a few linear interpolations. GPU hardware has a significant proportion of silicon allocated to optimizing the performance of these operations.

Algorithms that involve an accumulation or summation of values into a set of results (instead of just a write/copy) can leverage yet another large chunk of special silicon on GPUs: the blender is designed for efficiently compositing or accumulating values into an array. Some matrix math algorithms and reduction operations have shown benefits here.

**REGISTER PRESSURE**

Some architectures (such as GPUs) are flexible in that they can assign variable numbers of threads to a core based on how many registers each thread uses. This enables more threads to be used when fewer temporary registers are needed, but reduces the threads available (and the parallelism) for algorithms that need more registers. The key is to break tasks into simpler steps that can be executed across even more parallel threads. This is the essence of data-parallel programming.

For example, a standard 8x8 image DCT (discrete cosine transform) algorithm operates on transposed data for its second half. The transpose can takes dozens of registers to execute in place, but breaking it into two passes so that the transpose happens in the intervening I/O results in only a handful of registers needed for each half. This approach improved performance from far slower than a CPU to three times that of a highly optimized SSE assembly routine.

**HINTS FOR REDUCTIONS**

Reductions are common operations: find the total, average, min, max, or histogram of a set of data. The computations are easily data-parallel, but the output write is an example of cross-thread communication that must be managed carefully.

Initial implementations allocated a single shared location for all the threads to write into, but execution was completely serialized by write contention to that location. Allocating multiple copies of the reduction destination and then reducing these down in a separate step was found to be much faster. The key is to allocate enough intermediate locations to cover the number of cores (hundreds) and, therefore, performance level that you want to scale to.

**PROGRAMMING THE MEMORY SUBSYSTEM**

The data-parallel paradigm extends to the memory subsystem as well. A full data-parallel machine is able not only to process individual data elements separately, but also to read and write those elements in parallel. This characteristic of the memory subsystem is as important to performance as the execution model. For example, I/O ports are a shared resource, and performance is improved if multiple threads are not contending for the same one.

Data structures manipulated imply memory-access patterns. We have seen cases where switching from pointer-based data structures such as linked lists or sparse trees to data-parallel-friendly ones (regular arrays, grids, packed...
streams, etc.) allows code to become compute-bound instead of memory-bound (which can be as much as 10 times faster on GPUs). This is because memory is typically organized into pages, and there is some overhead in switching between pages. Grouping data elements and threads so that many results can be read from (or written to) the same page helps with performance.

Many types of trees and other sparse-data structures have data-parallel-friendly array-based implementations. Although using these structures is quite conventional, their implementations are nonintuitive to developers trained on pointer-based schemes.

The most important characteristic of the GPU memory subsystem is the cache architecture. Unlike a CPU, the GPU has hardly any read/write cache. It is assumed that so much data will be streaming through the processor that it will overflow just about any cache. As a result, the only caches present are separate read-through and write-through buffers that smooth out the data flow. Therefore, it is critical to select algorithms that do not rely on reuse of data at scales larger than the few local registers available. For example, histogram computation requires more read/write storage to contain the histogram bins than typical register allocation supports. Upcoming GPU architectures are beginning to add read/write caches so that more algorithms will work, including reasonably sized histograms, but since these caches are still 10 to 100 times smaller than those on the CPU, this will remain a key criterion when choosing an algorithm.

**GPU AS DATA-PARALLEL HARDWARE**

GPU systems are cheap and widely available, and many programmers (such as game developers) have identified key approaches to programming them efficiently.

First, it can be important to leverage all the silicon on the die. Applications that don’t light up the graphics-specific gates are already at a disadvantage compared with a CPU. For example, Govindaraju’s sort implementations show significant benefits from using the blending hardware.

Another way to ensure programming efficiency is to keep the data elements small. This extra hardware is assuming graphics data types that are optimal when they are 16 or fewer bytes in size, and ideally four bytes. If you can make your data look like what a GPU usually processes, you will get large benefits.

Unfortunately, the GPU’s high-speed memory system (10 times faster throughput than the CPU front side bus) is typically connected to the CPU by a link that is 10 times slower than CPU memory. Minimizing data and control traffic through this link is vital to GPU performance in low-latency scenarios. The secret is to keep data in the GPU’s memory as long as possible, bringing it back to the CPU only for persistent storage. Sometimes this may involve executing a small non-data-parallel task on the GPU because the cost of sending the required data across to the CPU, synchronizing it, and sending it back may be even greater.

**GPU GENERALITY**

With shorter design cycles, GPUs have been evolving more rapidly than CPUs. This evolution has typically been in the direction of increased generality. Now we are seeing GPU generality growing beyond the needs of basic rendering to more general applications. For example, in the past year new GPU environments have become available that expose features that the graphics APIs do not. Some now support sharing of data among threads and more flexible memory-access options.

This enables entirely new classes of algorithms on GPUs. Most obviously, more general approaches to 3D processing are becoming feasible, including manipulation of acceleration data structures for ray tracing, radiosity, or collision detection. Other obvious applications are in media processing (photo, video, and audio data) where the data types are similar to those of 3D rendering. Other
domains using similar data types are seismic and medical analysis.

**FUTURE HARDWARE EVOLUTION: CPU/GPU CONVERGENCE?**

Processor features such as instruction formats will likely converge as a result of pressure for a consistent programming model. GPUs may migrate to narrower SIMD widths to increase performance on branching code, while CPUs move to broader SIMD width to improve instruction efficiency.

The fact remains, however, that some tasks can be executed more efficiently using data-parallel algorithms. Since efficiency is so critical in this era of constrained power consumption, a two-point design that enables the optimal mapping of tasks to each processor model may persist for some time to come.

Further, if the hardware continues to lead the software, it is likely that systems will have more cores than the application can deal with at a given point in time, so providing a choice of processor types increases the chance of more of them being used.

Conceivably, a data-parallel system could support the entire feature set of a modern serial CPU core, including a rich set of interthread communications and synchronization mechanisms. The presence of such features, however, may not matter in the longer term because the more such traditional synchronization features are used, the worse performance will scale to high core counts. The fastest apps are not those that port their existing single-threaded or even dual-threaded code across, but those that switch to a different parallel algorithm that scales better because it relies less on general synchronization capabilities.

Figure 2 shows a list of algorithms that have been implemented using data-parallel paradigms with varying degrees of success. They are sorted roughly in order of how well they match the data-parallel model.

Data-parallel processors are becoming more broadly available, especially now that consumer GPUs support data-parallel programming environments. This paradigm shift presents a new opportunity for programmers who adapt in time.

The data-parallel industry is evolving without much guidance from software developers. The first to arrive will have the best chance to drive and shape upcoming data-parallel hardware architectures and development environments to meet the needs of their particular application space.

When programmed effectively, GPUs can be faster than current PC CPUs. The time has come to take advantage of this new processor type by making sure each task in your code base is assigned to the processor and memory model that is optimal for that task.

**REFERENCES**


**SUGGESTED FURTHER READING**

**GPU Gems 2:**
http://developer.nvidia.com/object/gpu_gems_2_home.html

**GPU Gems 3:**

**Glift data structures:**
http://graphics.cs.ucdavis.edu/~lefohn/work/glift/

**Rapidmind:**
http://www.rapidmind.net/index.php

**Intel Ct:**
http://www.intel.com/research/platform/terascale/
TeraScale_whitepaper.pdf

**Microsoft DirectX SDK:**

**Direct3D HLSL:**

**Nvidia CUDA SDK:**

**AMD Firestream SDK:**
http://ati.amd.com/technology/streamcomputing/
stream-computing.pdf

**Microsoft Research’s Accelerator:**
.aspx?type=technical%20report&id=1040&0sr=a

http://research.microsoft.com/research/downloads/
Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.
.aspx

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

**CHAS. BOYD** is a software architect at Microsoft. He joined the Direct3D team in 1995 and has contributed to releases since DirectX 3. During that time he has worked closely with hardware and software developers to drive the adoption of features such as programmable hardware shaders and float pixel processing into consumer graphics. Recently he has been investigating new processing architectures and applications for mass-market consumer systems.

© 2008 ACM 1542-7730/08/0300 $5.00

Object-Relational Mappers
The End of Transactions
ORM in Dynamic Languages
LINQ and Entity Framework

Coming Soon in Queue
Note: These slides will be updated for the final course presentation with updated material
A little history...
Graphics Processors: An Incredible Ride

GPU Evolution - Last 20 Years

- Wire Frame Rendering (Early 80's)
- Solid Shading
- Depth Buffering
- Vertex Lighting (Late 80's)
- Programmable Shading (2002-2005)
- 32-bit data parallel computing (2006 -)

Beyond Programmable Shading: Fundamentals
### A quick look back...

<table>
<thead>
<tr>
<th>Year</th>
<th>1998</th>
<th>1999</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor Size</td>
<td>350 nm</td>
<td>250 nm</td>
<td>180 nm</td>
<td>150 nm</td>
<td>150 nm</td>
<td>130 nm</td>
<td>90 nm</td>
<td>90 nm</td>
<td>80 nm</td>
<td></td>
</tr>
<tr>
<td>Transistor Count</td>
<td>5 million</td>
<td>13 million</td>
<td>30 million</td>
<td>60 million</td>
<td>110 million</td>
<td>110 million</td>
<td>160 million</td>
<td>321 million</td>
<td>384 million</td>
<td>700 million</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>75 MHz</td>
<td>100 MHz</td>
<td>183 MHz</td>
<td>275 MHz</td>
<td>325 MHz</td>
<td>412 MHz</td>
<td>550 MHz</td>
<td>625 MHz</td>
<td>650 MHz</td>
<td>740 MHz</td>
</tr>
<tr>
<td>Rendering Pipelines / Shader Processors</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/sec)</td>
<td>0.6</td>
<td>1.6</td>
<td>5.9</td>
<td>8.8</td>
<td>19.8</td>
<td>23.4</td>
<td>37.8</td>
<td>44.8</td>
<td>64</td>
<td>106.0</td>
</tr>
<tr>
<td>Pixel Clocking Rate (Mpixels/sec)</td>
<td>75</td>
<td>200</td>
<td>366</td>
<td>1100</td>
<td>2600</td>
<td>3300</td>
<td>8800</td>
<td>10000</td>
<td>31200</td>
<td>47360</td>
</tr>
<tr>
<td>Vertex Processing (Mvertices/sec)</td>
<td>4</td>
<td>8</td>
<td>30</td>
<td>69</td>
<td>325</td>
<td>412</td>
<td>825</td>
<td>1250</td>
<td>1300</td>
<td>11840</td>
</tr>
<tr>
<td>System Bus Bandwidth (GB/sec)</td>
<td>0.53</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>2.11</td>
<td>2.11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DirectX Version and Shader Model Support</td>
<td>DirectX 5</td>
<td>DirectX 6</td>
<td>DirectX 7</td>
<td>DirectX 8 (SM1.4)</td>
<td>DirectX 9.0 (SM2.0)</td>
<td>DirectX 9.0 (SM2.0)</td>
<td>DirectX 9.0 (SM3.0)</td>
<td>DirectX 9.0 (SM3.0)</td>
<td>DirectX 10.0 (SM4.0)</td>
<td></td>
</tr>
<tr>
<td>Features</td>
<td>Hardware Triangle Setup</td>
<td>128-bit memory interface, AGP 4X</td>
<td>Hardware transform &amp; lighting, DDR memory support</td>
<td>Programmable Shaders, Higher Order Surfaces</td>
<td>Floating Point Processing, 256-bit memory interface, AGP 8X</td>
<td>Unlimited shadertex instructions, 256MB DDR2 memory support</td>
<td>PCI Express x16, GDDR3 memory support</td>
<td>Dynamic shader flow control, 128-bit precision, HDR output</td>
<td>Unified Shader Architecture, 512-bit memory interface</td>
<td></td>
</tr>
</tbody>
</table>

**Beyond Programmable Shading: Fundamentals**

**1.7x / year**

**1.3x / year**

**1.6x / year**

**1.8x / year**

**2.0x / year**

**2.4x / year**

**1.3x / year**
What can we do on a GPU?

Game physics
Image processing
Video processing
Computer vision
AI
Financial modeling
CFD
Medical imaging
...

Beyond Programmable Shading: Fundamentals
ATI RV670 Feature Highlights

~75 GB/s memory bandwidth
- 256b GDDR4 interface

Targeted for handling thousands of simultaneous lightweight threads

320 (64x5) stream processors
- 256 (64x4) basic units (FMAC, ADD/SUB, etc.)
  - ~1/2 TFlops peak
- 64 enhanced transcendental units (adds COS, LOG, EXP, RSQ, etc.)
- Support for INT/UINT in all units (ADD/SUB, AND, XOR, NOT, OR, etc.)
- 64-bit double precision FP support
  - 1/5 rate peak (~100GFlops)
Stream Computing SDK
Stream Computing SDK

- Compilers
  - Brook+
  - ...

- Libraries
  - ACML
  - Cobra
  - RNG
  - Crypto

- 3rd Party Tools
  - Rapidmind
  - Other ISVs

- AMD Runtime

- Compute Abstraction Layer (CAL)
  - AMD Multicore CPUs
  - AMD Stream Processors

- Graphics API
  - DirectX
  - OpenGL

Beyond Programmable Shading: Fundamentals
Software-Hardware Interface

Developer Ecosystem
- Libraries (ACML, COBRA, etc.)
- Tools / dev environment

Compiled high level languages
- AMD will provide various implementations
- Developers free to create their own

Device independent / portable assembly
- Assembly spec provided

Device specific ISA
- Via device specific extensions to CAL and/or HAL
- ISA spec provided
Compute Abstraction Layer
Compute Abstraction Layer (CAL) goals

Exposé relevant parts of the GPU as they really are
- Command Processor
- Data Parallel Processor(s)
- Memory Controller

Hide all other graphics-specific features

Provide direct communication to device

Eliminate driver implemented procedural API
- Push policy decisions back to application
- Remove constraints imposed by graphics APIs
CAL Highlights

Memory managed
- Don’t have to manually maintain offsets, etc
- Asynchronous DMA: CPU GPU, GPU GPU, GPU CPU
- Multiple GPUs can share the same “system” memory

Core CAL API is device agnostic
Enables multi-device optimizations
- e.g. Multiple GPUs working together concurrently
- Multiple GPUs show up as multiple CAL devices

Extensions to CAL provide opportunities for
device specific optimization
CAL Memory System

CPU

CPU Memory (system / remote)

GPU

GPU Memory (local)

GPU

GPU Memory (local)

CPU Memory (system / remote)
static int gDevice = 0;

int main( int argc, char** argv )
{
    CALresult res = CAL_RESULT_OK;

    // open a cal device and create a context
    res = calInit();

    CALuint numDevices = 0;
    res = calDeviceGetCount( &numDevices );
    CHECK_ERROR(r, "There was an error enumerating devices.");

    CALdeviceinfo info;
    res = calDeviceGetInfo( &info, 0 );
    CHECK_ERROR(r, "There was an error getting device info.");

    CALdevice device = 0;
    res = calDeviceOpen( &device, 0 );
    CHECK_ERROR(r, "There was an error opening the device.");

    CALcontext ctx = 0;
    res = calCtxCreate( &ctx, device );
    CHECK_ERROR(r, "There was an error creating the context.");
// load module
CALmodule module;
res = calModuleLoadFile(&module, ctx, filename);
CHECK_ERROR(res, "There was an error loading the program module.");

NOTE: Modules can be created “online” via CAL compiler interface plugins, or “offline” via external tools (compilers, etc)

Load pre-compiled module from file
// allocate input and output resources and map them into the context
CALresource constRes;
res = calResAllocRemote1D(&constRes, &device, 1, 16, CAL_FORMAT_FLOAT4, 0, 0);
CHECK_ERROR(res, "There was an error allocating the constant resource.\n");

CALmem constMem;
res = calCtxGetMem(&constMem, ctx, constRes);
CHECK_ERROR(res, "There was an error getting memory from the constant resource.\n");

CALresource outputRes;
res = calResAllocRemote2D(&outputRes, &device, 1,
    BufferWidth, BufferHeight,
    CAL_FORMAT_FLOAT4, 0, 0);
CHECK_ERROR(res, "There was an error allocating the output resource.\n");

CALmem outputMem;
res = calCtxGetMem(&outputMem, ctx, outputRes);
CHECK_ERROR(res, "There was an error getting memory from the output resource.\n");

Allocate system (CPU) resource for output buffer
Allocate system (CPU) resource for constants
Get handle to actual memory
// clear the resources to known values
float* fdata;
int* idata;
CALuint pitch;

// set constant values
res = calMemMap( (CALvoid**)&idata, &pitch, ctx, constMem, 0 );
idata[0] = InputValue;
idata[1] = InputValue;
idata[2] = InputValue;
idata[3] = InputValue;
res = calMemUnmap( ctx, constMem );

res = calMemMap( (CALvoid**)&fdata, &pitch, ctx, outputMem, 0 );
for (int i = 0; i < BufferHeight; i++)
{
    float* tmp = &fdata[i * pitch * 4];
    for (int j = 0; j < 4 * BufferWidth; j++)
    {
        tmp[j] = OutputValue;
    }
}
res = calMemUnmap( ctx, outputMem );
// setup the program's inputs and outputs
CALname constName;
res = calModuleGetName( &constName, ctx, module, "cb0" );
CHECK_ERROR( res, "There was an error finding the constant buffer.
" );

res = calCtxSetMem( ctx, constName, constMem );
CHECK_ERROR( res, "There was an error setting the constant buffer memory.
" );

CALname outName;
res = calModuleGetName( &outName, ctx, module, "o0" );
CHECK_ERROR( res, "There was an error finding the program output.
" );

res = calCtxSetMem( ctx, outName, outputMem );
CHECK_ERROR( res, "There was an error setting the program output.
" );
// get the program entry point
CALfunc func;
res = calModuleGetEntry(&func, ctx, module, "main");
CHECK_ERROR(res, "There was an error finding the program entry point.");

// set computational domain
CALdomain rect;
rect.x = 0;
rect.y = 0;
rect.width = BufferWidth;
rect.height = BufferHeight;

// run the program, wait for completion
CALevent event;
res = calCtxRunProgram(&event, ctx, func, &rect);
CHECK_ERROR(res, "There was an error running the program.");

// wait for function to finish
while (calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING);
// cleanup and exit
calCtxSetMem( ctx, constName, 0 );
calCtxSetMem( ctx, outName, 0 );
calModuleUnload( ctx, module );
calCtxReleaseMem( ctx, constMem );
calResFree( constRes );
calCtxReleaseMem( ctx, outputMem );
calResFree( outputRes );
calCtxDestroy( ctx );
calDeviceClose( device );
Using multiple GPUs

Application Thread

calclCompile(...)
calclLink(...)

GPU Compute Thread 0

calDeviceOpen(&dev0, 0)
calCtxCreate(&ctx0, dev0)

Setup program, inputs, outputs, constants

calCtxRunProgram(&ctx0, ...)
calCtxIsEventDone(ctx0, ...)

No

Done?

Yes

calCtxDestroy(ctx0)
calDeviceClose(dev0)

GPU Compute Thread 1

calDeviceOpen(&dev1, 1)
calCtxCreate(&ctx1, dev1)

Setup program, inputs, outputs, constants

calCtxRunProgram(&ctx1, ...)
calCtxIsEventDone(ctx1, ...)

No

Done?

Yes

calCtxDestroy(ctx1)
calDeviceClose(dev1)
/ Program to perform warp on an image (bilinear warp done using automatic texture filtering)
OneOutput_PS WarpImage( float2 vpos: VPOS )
{
   // compute scaled texcoord
   float2 pos = (vpos + 0.5) / gImageSize;

   float2 wtexel = tex2D( gWarpMap, pos ).rg;
   float2 wpos = (wtexel + 0.5) / gMapSize;

   float4 texel = tex2D( gInput, wpos );

   OneOutput_PS outC;
   outC.Col0 = texel;
   return outC;
}
Beyond Programmable Shading: Fundamentals

AMD IL

il_ps_2_0
dcldef_x(*)_y(*)_z(*)_w(*) r0
def c2, 0.5, 0.0, 0.0, 0.0
dclpi_x(*)_y(*)_z(-)_w(-)_center_centered vWinCoord0
dclpt_type(2d)_coordmode(normalized)_stage(0)
dclpt_type(2d)_coordmode(normalized)_stage(1)
add r0.xy__, c2.x, vWinCoord0
rcp_zeroop(infinity) r0.__z__, c0.x
rcp_zeroop(infinity) r0.__w, c0.y
mul r0.xy__, r0, r0.zwzw
texld_stage(1)_shadowmode(never) r0, r0
add r0.xy__, r0, c2.x
rcp_zeroop(infinity) r0.__z__, c1.x
rcp_zeroop(infinity) r0.__w, c1.y
mul r0.xy__, r0, r0.zwzw
texld_stage(0)_shadowmode(never) r35, r0
colorclamp oC0, r35
end
; ------------------ PS Disassembly -----------------------

00 ALU: ADDR(32) CNT(9)
  0 x: ADD R123.x, R0.y, -0.5
  y: ADD R123.y, R0.x, -0.5
  t: RECIP_IEEE R127.w, C0.x

1 z: ADD R123.z, PV(0).x, 0.5
  w: ADD R123.w, PV(0).y, 0.5
  t: RECIP_IEEE R122.z, C0.y

2 x: MUL R0.x, PV(1).w, R127.w
  y: MUL R0.y, PV(1).z, PS(1).x
  t: RECIP_IEEE R0.w, C1.x

01 TEX: ADDR(64) CNT(1)
  3 SAMPLE R0.xy__, R0.xyxx, t1, s1 WHOLE QUAD

02 ALU: ADDR(41) CNT(5)
  4 x: ADD R123.x, R0.y, 0.5
  y: ADD R123.y, R0.x, 0.5
  t: RECIP_IEEE R122.z, C1.y

5 x: MUL R0.x, PV(4).y, R0.w
  y: MUL R0.y, PV(4).x, PS(4).x

03 TEX: ADDR(66) CNT(1) VALID_PIX
  6 SAMPLE R0, R0.xyxx, t0, s0

04 ALU: ADDR(46) CNT(5)
  7 x: MOV R0.x, R0.x
  y: MOV R0.y, R0.y
  z: MOV R0.z, R0.z
  w: MOV R0.w, R0.w
  t: MOV R1.x, R1.x FOGMERGE

05 EXP_DONE: PIX0, R0
END_OF_PROGRAM

Beyond Programmable Shading: Fundamentals
Brook+
What is Brook+?

Brook is an extension to the C-language for stream programming originally developed by Stanford University.

Brook+ is an implementation by AMD of the Brook GPU spec on AMD's compute abstraction layer with some enhancements.
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}

int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float input_c[10][10];
    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, input_c);
    ...
}
```c
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}

kernel void sum(float a[], float b[], out float c[])
{
    int idx = indexof(c);
    c = a[idx] + b[idx];
}
kernel void sum(float a<>, float b<>, out float c[])
{
    int idx = indexof(c);
    c[idx] = a + b;
}

+    +    +    +    +    +    +    +

=    =    =    =    =    =    =    =

```

**Standard Streams** - implicit and predictable access pattern

**Gather Streams** - dynamic read access pattern

**Scatter Stream** - dynamic write access pattern
Brook+ Compiler

Converts Brook+ files into C++ code. Kernels, written in C, are compiled to AMD’s IL code for the GPU or C code for the CPU.
Beyond Programmable Shading: Fundamentals

IL code is executed on the GPU. The backend is written in CAL.
Brook+ Features

Brook+ is an extension to the Brook for GPUs source code.

Features of Brook for GPUs relevant to modern graphics hardware are maintained.

Kernels are compiled to AMD’s IL

Runtime uses CAL to execute on AMD GPUs
  • CAL runtime generates ASIC specific ISA dynamically

Original CPU backend also included
  • Currently used mainly for debugging
  • Optimizations currently underway
Case Study: Folding@Home

Folding@Home client using Brook+
Currently 145 TFLOPS on 2100 GPU clients (R5XX/R6XX)
Avg. 60 GFLOPS per GPU client (R5XX)
R6XX beta client available now

Compared to:
Avg. 1 GFLOPS per CPU client
On par with PS3, but GPU version running more complicated core
Brook+ vs. Brook

Double precision
Integer support
Scatter (mem-export)
Asynchronous CPU->GPU transfers (GPU->CPU still synchronous)
Linux, Vista, XP
  • 32 & 64-bit

Extension Mechanism
  • Allow ASIC specific features to be exposed without ‘sullying’ core language
What’s coming in Brook, 1/2

Libraries
- Basic math (libm)
- Random number generators
- Extended primitives
  - Scan, compact, reduce

Inline HLSL, IL, ISA
- Allows heavier tuning
- Map to esoteric hardware features

Performance tuning
- Remove copies from user pointer to PCIe, direct DMA from/to pointer
  - Runtime tuning
Multi-core support
- Initial OpenMP support is in
  - Good scaling, but runtime and code-gen not optimized

Future-future-future stuff:
- Transparent multi-GPU
  - “Crossfire” mode for Brook
  - Lots of technical challenges
  - There will be scalability issues
- Optimizing stream compiler
  - Optimize stream graph, not just kernels
  - Kernel fusion
  - Compiler controlled data movement to and from GPU
- Support AC/Fusion devices
Questions?

AMD Stream Computing SDK
http://ati.amd.com/technology/streamcomputing/

Folding@Home
http://folding.stanford.edu/

Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2006 Advanced Micro Devices, Inc. All rights reserved.
Scalable Parallel Programming

JOHN NICKOLLS, IAN BUCK, AND MICHAEL GARLAND, NVIDIA, KEVIN SKADRON, UNIVERSITY OF VIRGINIA
The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore’s law. The challenge is to develop mainstream application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

According to conventional wisdom, parallel programming is difficult. Early experience with the CUDA\textsuperscript{1,2} scalable parallel programming model and C language, however, shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the new Tesla unified graphics and computing architecture (described in the GPU sidebar) run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA
Scalable Parallel PROGRAMMING with CUDA

model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs. CUDA provides three key abstractions—a hierarchy of thread groups, shared memories, and barrier synchronization—that provide a clear parallel structure to conventional C code for one thread of the hierarchy.

Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into

UNIFIED GRAPHICS AND COMPUTING GPUS

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU (graphics processing unit) has evolved into a highly parallel, multithreaded, manycore processor. It is designed to efficiently support the graphics shader programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels at fine-grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently.

The tremendous raw performance of modern GPUs has led researchers to explore mapping more general non-graphics computations onto them. These GPGPU (general-purpose computation on GPUs) systems have produced some impressive results, but the limitations and difficulties of doing this via graphics APIs are legend. This desire to use the GPU as a more general parallel computing device motivated NVIDIA to develop a new unified graphics and computing GPU architecture and the CUDA programming model.

GPU COMPUTING ARCHITECTURE

Introduced by NVIDIA in November 2006, the Tesla unified graphics and computing architecture significantly extends the GPU beyond graphics—its massively multithreaded processor array becomes a highly efficient unified platform for both graphics and general-purpose parallel computing applications. By scaling the number of processors and memory partitions, the Tesla architecture spans a wide market range—from the high-performance enthusiast GeForce 8800 GPU and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs. Its computing features enable straightforward programming of the GPU cores in C with CUDA. Wide availability in laptops, desktops, workstations, and servers, coupled with C programmability and CUDA software, make the Tesla architecture the first ubiquitous supercomputing platform.

The Tesla architecture is built around a scalable array of multithreaded SMs (streaming multiprocessors). Current GPU implementations range from 768 to 12,288 concurrently executing threads. Transparent scaling across this wide range of available parallelism is a key design goal of both the GPU architecture and the CUDA programming model. Figure A shows a GPU with 14 SMs—a total of 112 SP (streaming processor) cores—interconnected with four external DRAM partitions. When a CUDA program on the host CPU invokes a kernel grid, the CWD (compute work distribution) unit enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

An SM consists of eight scalar SP cores, two SFUs (special function units) for transcendentals, an MT IU (multithreaded instruction unit), and on-chip shared memory. The SM creates, manages, and executes up to 768 concurrent threads in hardware with zero scheduling overhead. It can execute as many as eight CUDA thread blocks concurrently, limited by thread and memory resources. The SM implements the CUDA __syncthreads() barrier synchronization intrinsic with a single instruction. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing a new thread to be created to compute each vertex, pixel, and data point.

To manage hundreds of threads running several different programs, the Tesla SM employs a new architecture we call
finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

THE CUDA PARADIGM
CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel kernels, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A grid is a set of thread blocks that may each be executed independently and thus may execute in parallel.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks...
making up the grid. Each thread is given a unique thread ID number threadIdx within its thread block, numbered 0, 1, 2, \ldots, blockDim-1, and each thread block is given a unique block ID number blockIdx within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.

As a very simple example of parallel programming, suppose that we are given two vectors $x$ and $y$ of $n$ floating-point numbers each and that we wish to compute the result of $y \leftarrow ax + y$, for some scalar value $a$. This is the

SIMT warp start together at the same program address but are otherwise free to branch and execute independently. Each SM manages a pool of 24 warps of 32 threads per warp, a total of 768 threads.

Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. As a result, the Tesla-architecture GPUs are dramatically more efficient and flexible on branching code than previous-generation GPUs, as their 32-thread warps are much narrower than the SIMD (single-instruction multiple-data) width of prior GPUs.

SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

A thread’s variables typically reside in live registers. The 16KB SM shared memory has very low access latency and high bandwidth similar to an L1 cache; it holds CUDA per-block __shared__ variables for the active thread blocks. The SM provides load/store instructions to access CUDA __device__ variables in GPU external DRAM. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. Because global memory latency can be hundreds of processor clocks, CUDA programs copy data to shared memory when it must be accessed multiple times by a thread block. Tesla load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimizations. The large thread count in each SM, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM. The latest Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

CUDA applications perform well on Tesla-architecture GPUs because CUDA’s parallelism, synchronization, shared memories, and hierarchy of thread groups map efficiently to features of the GPU architecture, and because CUDA expresses application parallelism well.

REFERENCES
so-called saxpy kernel defined by the BLAS (basic linear algebra subprograms) library. The code for performing this computation on both a serial processor and in parallel using CUDA is shown in figure 1.

The __global__ declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function-call syntax

```
kernella{dimGrid, dimBlock}<<<... parameter list ...>>;  
```

where dimGrid and dimBlock are three-element vectors of type dim3 that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to 1.

In the example, we launch a grid that assigns one thread to each element of the vectors and puts 256 threads in each block. Each thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. The serial and parallel versions of this code are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling, and termination are handled for the programmer by the underlying system. Indeed, a Tesla-architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a barrier by calling the __syncthreads() intrinsic. This guarantees that no thread participating in the barrier can proceed until all participating threads have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by participating threads before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Since threads in a block may share local memory and synchronize via barriers, they will reside on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. This virtualizes the processing elements and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. This also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may coordinate their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example.

This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary
Scalable Parallel Programming with CUDA

number of cores, as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock.

An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently given sufficient hardware resources. Dependent grids execute sequentially, with an implicit inter-kernel barrier between them, thus guaranteeing that all blocks of the first grid will complete before any block of the second dependent grid is launched.

Threads may access data from multiple memory spaces during their execution. Each thread has a private local memory. CUDA uses this memory for thread-private variables that do not fit in the thread’s registers, as well as for stack frames and register spilling. Each thread block has a shared memory visible to all threads of the block that has the same lifetime as the block. Finally, all threads have access to the same global memory. Programs declare variables in shared and global memory with the __shared__ and __device__ type qualifiers. On a Tesla-architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can, therefore, provide for high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure 2 diagrams the nested levels of threads, thread blocks, and grids of thread blocks. It shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as cudaMalloc() and cudaMemcpy(). Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use cudaMemcpy() to copy data between the allocated space and the host system memory.

The CUDA programming model is similar in style to the familiar SPMD (single-program multiple-data) model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. CUDA, however, is more flexible than most real-
izations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads.

Figure 3 shows an example of a SPMD-like CUDA code sequence. It first instantiates `kernelF` on a 2D grid of $3 \times 2$ blocks where each 2D thread block consists of $5 \times 3$ threads. It then instantiates `kernelG` on a 1D grid of four 1D thread blocks with six threads each. Because `kernelG` depends on the results of `kernelF`, they are separated by an inter-kernel synchronization barrier.

The concurrent threads of a thread block express fine-grained data and thread parallelism. The independent thread blocks of a grid express coarse-grained data parallelism. Independent grids express coarse-grained task parallelism. A kernel is simply C code for one thread of the hierarchy.

**Restrictions**

When developing CUDA programs, it is important to understand the ways in which the CUDA model is restricted, largely for reasons of efficiency. Threads and thread blocks may be created only by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla architecture implements hardware management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread-block level, but blockwide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between thread blocks within the same kernel grid is not allowed—they must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any
order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks. However, multiple thread blocks can coordinate their work using atomic operations on global memory (e.g., to manage a data structure).

Recursive function calls are not allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects. Of course, problems large enough to need a GPU performance boost amortize the overhead better than small problems.

RELATED WORK

Although the first CUDA implementation targets NVIDIA GPUs, the CUDA abstractions are general and useful for programming multicore CPUs and scalable parallel systems. Coarse-grained thread blocks map naturally to separate processor cores, while fine-grained threads map to multiple-thread contexts, vector operations, and pipelined loops in each core. Stratton et al. have developed a prototype source-to-source translation framework that compiles CUDA programs for multicore CPUs by mapping a thread block to loops within a single CPU thread. They found that CUDA kernels compiled in this way perform and scale well.4

CUDA uses parallel kernels similar to recent GPGPU programming models, but differs by providing flexible thread creation, thread blocks, shared memory, global memory, and explicit synchronization. Streaming languages apply parallel kernels to data records from a stream. Applying a stream kernel to one record is analogous to executing a single CUDA kernel thread, but stream programs do not allow dependencies among kernel threads, and kernels communicate only via FIFO (first-in, first-out) streams. Brook for GPUs differentiates between FIFO input/output streams and random-access gather streams, and it supports parallel reductions. Brook is a good fit for earlier-generation GPUs with random access texture units and raster pixel operation units.5

Pthreads and Java provide fork-join parallelism but are not particularly convenient for data-parallel applications. OpenMP targets shared memory architectures with parallel execution constructs, including “parallel for” and teams of coarse-grained threads. Intel’s C++ Threading Building Blocks provide similar features for multicore CPUs. MPI targets distributed memory systems and uses message passing rather than shared memory.

CUDA APPLICATION EXPERIENCE

The CUDA programming model extends the C language with a small number of additional parallel abstractions. Programmers who are comfortable developing in C can quickly begin writing CUDA programs.

In the relatively short period since the introduction of CUDA, a number of real-world parallel application codes have been developed using the CUDA model. These include FHD-spiral MRI reconstruction,6 molecular dynamics,7 and n-body astrophysics simulation.8 Running on Tesla-architecture GPUs, these applications were able to achieve substantial speedups over alternative implementations running on serial CPUs: the MRI reconstruction was 263 times faster; the molecular dynamics code was 10–100 times faster; and the n-body simulation was 50–250 times faster. These large speedups are a result of the highly parallel nature of the Tesla architecture and its high memory bandwidth.

Compressed Sparse Row (CSR) Matrix

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>row 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Av[7] = \{3, 1, 2, 4, 1, 1\}

Aj[7] = \{0, 2, 1, 2, 3, 0, 3\}

Ap[5] = \{0, 2, 2, 5, 7\}

FIG 4
EXAMPLE: SPARSE MATRIX-VECTOR PRODUCT

A variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. SpMV (sparse matrix-vector multiplication) is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss here, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient method straightforward.

A sparse $n \times n$ matrix is one in which the number of nonzero entries $m$ is only a small fraction of the total. Sparse matrix representations seek to store only the nonzero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only $m=O(n)$ nonzero elements, this represents a substantial savings in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the CSR (compressed sparse row) representation. The $m$ nonzero elements of the matrix $A$ are stored in row-major order in an array $Av$. A second array $Aj$ records the corresponding column index for each entry of $Av$. Finally, an array $Ap$ of $n+1$

.$$float \text{multiply}_\text{row}(unsigned \ int \ \text{rowsize},$$
$$\quad unsigned \ int \ *Aj, \ // \ column \ indices \ for \ row$$
$$\quad float *Av, \quad // \ non-zero \ entries \ for \ row$$
$$\quad float *x) \quad // \ the \ RHS \ vector$$

$$\{$$
$$\quad float \ sum = 0;$$

$$\quad for(unsigned \ int \ \text{column}=0; \ \text{column}<\text{rowsize}; \ ++\text{column})$$
$$\quad \quad sum += Av[\text{column}] * x[Aj[\text{column}]];$$

$$\quad return \ sum;$$

$$\}$$

$$\text{FIG 5}$$

.$$void \text{csrmul}_\text{serial}(unsigned \ int \ *Ap, unsigned \ int \ *Aj,$$
$$\quad float *Av, unsigned \ int \ \text{num\_rows},$$
$$\quad float *x, \ float *y)$$

$$\{$$
$$\quad for(unsigned \ int \ \text{row}=0; \ \text{row}<\text{num\_rows}; \ ++\text{row})$$
$$\quad \quad unsigned \ int \ \text{row\_begin} = Ap[\text{row}];$$
$$\quad unsigned \ int \ \text{row\_end} \quad = Ap[\text{row}+1];$$

$$\quad y[\text{row}] = \text{multiply}_\text{row}(\text{row\_end-row\_begin}, \ Aj+\text{row\_begin},$$
$$\quad \quad Av+\text{row\_begin}, \ x);$$

$$\}$$

$$\text{FIG 6}$$

.$$\_\_\text{global}\_\_$$

.$$void \text{csrmul}_\text{kernel}(unsigned \ int \ *Ap, unsigned \ int \ *Aj,$$
$$\quad float *Av, unsigned \ int \ \text{num\_rows},$$
$$\quad float *x, \ float *y)$$

$$\{$$
$$\quad unsigned \ int \ \text{row} = \text{blockIdx.x*blockDim.x + threadIdx.x};$$

$$\quad if( \ \text{row}<\text{num\_rows} )$$
$$\quad \quad unsigned \ int \ \text{row\_begin} = Ap[\text{row}];$$
$$\quad unsigned \ int \ \text{row\_end} \quad = Ap[\text{row}+1];$$

$$\quad y[\text{row}] = \text{multiply}_\text{row}(\text{row\_end-row\_begin}, \ Aj+\text{row\_begin},$$
$$\quad \quad Av+\text{row\_begin}, \ x);$$

$$\}$$

$$\text{FIG 7}$$
elements records the extent of each row in the previous arrays; the entries for row $i$ in $A_j$ and $A_v$ extend from index $A_p[i]$ up to, but not including, index $A_p[i+1]$. This implies that $A_p[0]$ will always be 0 and $A_p[n]$ will always be the number of nonzero elements in the matrix. Figure 4 shows an example of the CSR representation of a simple matrix.

Given a matrix $A$ in CSR form, we can compute a single row of the product $y = Ax$ using the `multiply_row()` procedure shown in figure 5.

Computing the full product is then simply a matter of looping over all rows and computing the result for that row using `multiply_row()`, as shown in figure 6.

This algorithm can be translated into a parallel CUDA kernel quite easily. We simply spread the loop in `csrmul_serial()` over many parallel threads. Each thread will compute exactly one row of the output vector $y$. Figure 7 shows the code for this kernel. Note that it looks extremely similar to the serial loop used in the `csrmul_serial()` procedure. There are really only two points of difference. First, the row index is computed from the block and thread indices assigned to each thread. Second, we have a conditional that evaluates a row product only if the row index is within the bounds of the matrix (this is necessary since the number of rows $n$ need not be a multiple of the block size used in launching the kernel).

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like the code in figure 8.

The pattern that we see here is a common one. The original serial algorithm is a loop whose iterations are independent of each other. Such loops can be parallelized quite easily by simply assigning one or more iterations of the loop to each parallel thread. The programming model provided by CUDA makes expressing this type of parallelism particularly straightforward.

This general strategy of decomposing computations into blocks of independent work, and more specifically breaking up independent loop iterations, is not unique to CUDA. This is a common approach used in one form or another by various parallel programming systems, including OpenMP and Intel’s Threading Building Blocks.

```c
unsigned int blocksize = 128;  // or any size up to 512
unsigned int nblocks  = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

FIG 8

```c
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                    float *Av, unsigned int num_rows,
                    const float *x, float *y)
{
    // Cache the rows of x[] corresponding to this block.
    __shared__ float cache[blocksize];

    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end   = block_begin + blockDim.x;
    unsigned int row             = block_begin + threadIdx.x;

    // Fetch and cache our window of x[].
    if( row<num_rows)  cache[threadIdx.x] = x[row];
    __syncthreads();

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end    = Ap[row+1];
        float sum = 0, x_j;

        for(unsigned int col=row_begin; col<row_end; ++col)
        {
            unsigned int j = Aj[col];

            // Fetch x_j from our cache when possible
            if( j>=block_begin && j<block_end )
            {
                x_j = cache[j-block_begin];
            }
            else
            {
                x_j = x[j];
            }

            sum += Av[col] * x_j;
        }

        y[row] = sum;
    }
}
```

FIG 9
CACHING IN SHARED MEMORY

The SpMV algorithms outlined here are fairly simplistic. We can make a number of optimizations in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking.\(^\text{10}\) The parallel kernels can also be reimplemented in terms of data-parallel scan operations.\(^\text{11}\)

One of the important architectural features exposed by CUDA is the presence of the per-block shared memory, a small on-chip memory with very low latency. Taking advantage of this memory can deliver substantial performance improvements. One common way of doing this is to use shared memory as a software-managed cache to hold frequently reused data, shown in figure 9.

In the context of sparse matrix multiplication, we observe that several rows of \(A\) may use a particular array element \(x[i]\). In many common cases, and particularly when the matrix has been reordered, the rows using \(x[i]\) will be rows near row \(i\). We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows \(i\) through \(j\) will load \(x[i]\) through \(x[j]\) into its shared memory. We will unroll the \texttt{multiply\_row()} loop and fetch elements of \(x\) from the cache whenever possible. The resulting code is shown in figure 9. Shared memory can also be used to make other optimizations, such as fetching \(A[p[row+1]]\) from an adjacent thread rather than refetching it from memory.

Because the Tesla architecture provides an explicitly managed on-chip shared memory rather than an implicitly active hardware cache, it is fairly common to add this sort of optimization. Although this can impose some additional development burden on the programmer, it is relatively minor, and the potential performance benefits can be substantial. In the example shown in figure 9, even this fairly simple use of shared memory returns a roughly 20 percent performance improvement on representative matrices derived from 3D surface meshes. The availability of an explicitly managed memory in lieu of an implicit cache also has the advantage that caching and prefetching policies can be specifically tailored to the application needs.

EXAMPLE: PARALLEL REDUCTION

Suppose that we are given a sequence of \(N\) integers that must be combined in some fashion (e.g., a sum). This occurs in a variety of algorithms, linear algebra being a common example. On a serial processor, we would write a simple loop with a single accumulator variable to construct the sum of all elements in sequence. On a parallel machine, using a single accumulator variable would create a global serialization point and lead to very poor performance. A well-known solution to this problem is the so-called parallel reduction algorithm. Each parallel thread sums a fixed-length subsequence of the input. We then collect these partial sums together, by summing

```c
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
 // Each block loads its elements into shared memory, padding
 // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements. See attached figure.
    for(int s=blockDim.x/2; s>0; s=s/2)
        { 
            if(tid < s) x[tid] += x[tid + s];
            __syncthreads();
        }

    // Thread 0 now holds the sum of all input values
    // to this block. Have it add that sum to the running total
    if( tid == 0 ) atomicAdd(total, x[tid]);
}
```

FIG 10
pairs of partial sums in parallel. Each step of this pair-wise summation cuts the number of partial sums in half and ultimately produces the final sum after \( \log_2 N \) steps. Note that this implicitly builds a tree structure over the initial partial sums.

In the example shown in figure 10, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length one). At the end of the reduction, we want thread 0 to hold the sum of all elements initially loaded by the threads of its block. We can achieve this in parallel by summing values in a tree-like pattern. The loop in this kernel implicitly builds a summation tree over the input elements. The action of this loop for the simple case of a block of eight threads is illustrated in figure 11. The steps of the loop are shown as successive levels of the diagram and edges indicate from where partial sums are being read.

At the end of this loop, thread 0 holds the sum of all the values loaded by this block. If we want the final value of the location pointed to by total to contain the total of all elements in the array, we must combine the partial sums of all the blocks in the grid. One strategy would be to have each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until we had reduced the sequence to a single value. A more attractive alternative supported by the Tesla architecture is to use \texttt{atomicAdd()}, an efficient atomic read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

Parallel reduction is an essential primitive for parallel programming and highlights the importance of per-block shared memory and low-cost barriers in making cooperation among threads efficient. This degree of data shuffling among threads would be prohibitively expensive if done in off-chip global memory.

**THE DEMOCRATIZATION OF PARALLEL PROGRAMMING**

CUDA is a model for parallel programming that provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications. In fact, CUDA is an excellent programming environment for teaching parallel programming. The University of Virginia has used it as just a short, three-week module in an undergraduate computer architecture course, and students were able to write a correct k-means clustering program after just three lectures. The University of Illinois has successfully taught a semester-long parallel programming course using CUDA to a mix of computer science and non-computer science majors, with students obtaining impressive speedups on a variety of real applications, including the previously mentioned MRI reconstruction example.

CUDA is supported on NVIDIA GPUs with the Tesla unified graphics and computing architecture of the GeForce 8-series, recent Quadro, Tesla, and future GPUs.
The programming paradigm provided by CUDA has allowed developers to harness the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of 100 times or more on a variety of sophisticated applications.

The CUDA abstractions, however, are general and provide an excellent programming environment for multicore CPU chips. A prototype source-to-source translation framework developed at the University of Illinois compiles CUDA programs for multicore CPUs by mapping a parallel thread block to loops within a single physical thread. CUDA kernels compiled in this way exhibit excellent performance and scalability.

Although CUDA was released less than a year ago, it is already the target of massive development activity—there are tens of thousands of CUDA developers. The combination of massive speedups, an intuitive programming environment, and affordable, ubiquitous hardware is rare in today’s market. In short, CUDA represents a democratization of parallel programming.

REFERENCES
4. See reference 3.
12. See Reference 3.

Links to the latest version of the CUDA development tools, documentation, code samples, and user discussion forums can be found at: http://www.nvidia.com/CUDA.

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

JOHN NICKOLLS is director of architecture at NVIDIA for GPU computing. He was previously with Broadcom, Silicon Spice, Sun Microsystems, and was a cofounder of MasPar Computer. His interests include parallel processing systems, languages, and architectures. He has a B.S. in electrical engineering and computer science from the University of Illinois, and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

IAN BUCK works for NVIDIA as the GPU-Compute software manager. He completed his Ph.D. at the Stanford Graphics Lab in 2004. His thesis was titled “Stream Computing on Graphics Hardware,” researching programming models and computing strategies for using graphics hardware as a general-purpose computing platform. His work included developing the Brook software tool chain for abstracting the GPU as a general-purpose streaming coprocessor.

MICHAEL GARLAND is a research scientist with NVIDIA Research. Prior to joining NVIDIA, he was an assistant professor in the department of computer science at the University of Illinois at Urbana-Champaign. He received Ph.D. and B.S. degrees from Carnegie Mellon University. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models.

KEVIN SKADRON is an associate professor in the department of computer science at the University of Virginia and is currently on sabbatical with NVIDIA Research. He received his Ph.D. from Princeton University and B.S. from Rice University. His research interests include power- and temperature-aware design, and manycore architecture and programming models. He is a senior member of the ACM.

© 2008 ACM 1542-7730/08/0300 $5.00
Programmable Graphics—
The Future of Interactive Rendering

Matt Pharr, Aaron Lefohn, Craig Kolb, Paul Lalonde, Tim Foley, and Geoff Berry
Neoptica Technical Report, March 2007
Overview

Recent innovations in computer hardware architecture—the arrival of multi-core CPUs, the generalization of graphics processing units (GPUs), and the imminent increase in bandwidth available between CPU and GPU cores—make a new era of interactive graphics possible. As a result of these changes, game consoles, PCs and laptops will have the potential to provide unprecedented levels of visual richness, realism, and immersiveness, making interactive graphics a compelling killer app for these modern computer systems. However, current graphics programming models and APIs, which were conceived of and developed for the previous generation of GPU-only rendering pipelines, severely hamper the type and quality of imagery that can be produced on these systems. Fulfilling the promise of programmable graphics—the new era of cooperatively using the CPU, GPU, and complex, dynamic data structures to efficiently synthesize images—requires new programming models, tools, and rendering systems that are designed to take full advantage of these new parallel heterogeneous architectures.

Neoptica is developing the next-generation interactive graphics programming model for these architectures, as well as new graphics techniques, algorithms, and rendering engines that showcase the unprecedented visual quality that they make possible.

Introduction

Computer system architecture is amidst a revolution. The single-processor computer is being supplanted by parallel heterogeneous systems comprised of processors supporting multiple styles of computation. CPU architects are no longer able to improve computational performance of the traditional heart of the computer system, the CPU, by increasing the clock speed of a single processor; instead, they are now providing a rapidly-increasing number of parallel coarse-grained cores, currently capable of delivering approximately 90 GFLOPS. Simultaneously, graphics processing units have evolved to be efficient fine-grained data-parallel coprocessors that deliver much greater raw floating-point horsepower than today’s multi-core CPUs; the latest graphics processors from NVIDIA and AMD provide on the order of 400 GFLOPS of peak performance via hundreds of computational units working in parallel. In addition, although CPUs and GPUs have traditionally been separated by low-bandwidth and high-latency communication pathways (e.g. AGP and PCI-Express), rapidly-improving interconnect technology (e.g. AMD Torrenza and Intel Geneseo) and the promise of integrating CPUs and GPUs on a single chip (e.g. AMD Fusion) allow CPUs and GPUs to share data much more efficiently, thereby enabling graphics applications to intermix computation styles to optimally use the system’s computational resources.

The success of these new heterogeneous parallel architectures hinges upon consumer applications taking full advantage of their computational power. In order for this to happen, programmers must be presented with intuitive and efficient parallel programming models for these systems. However, decades of work on parallel programming solutions have shown that low-level primitives such as mutexes, semaphores, threads, and message passing are not amenable to creating reliable, complex software systems. Furthermore, existing higher-level parallel programming abstractions have not proven widely successful; these models typically limit developers to a single type of parallelism (i.e., exclusively data-parallel or exclusively task-
parallel), which unnecessarily constrains developer flexibility and makes poor use of the mixed computational resources in heterogeneous systems. Without a higher-level, easy-to-use parallel programming model that allows developers to take full advantage of the entire system, the new parallel architectures may not deliver compelling benefit to users, thus reducing consumer demand for new PCs.

Interactive 3-D computer graphics is now the most computationally demanding consumer application. The economic force of the computer gaming industry and its appetite for computational power have driven the rapid development of current GPUs. In addition, the GPU programming model represents perhaps the only widely-adopted parallel programming model to date. Unfortunately, this model assumes a GPU-only, unidirectional, fixed graphics pipeline. Creating a new programming model for interactive graphics that fully exposes the computational and communication abilities of these new architectures is necessary to enable a revolution in the quality and efficiency of interactive graphics and to provide a killer app for these new platforms.

Neoptica is developing the next-generation interactive graphics programming model for heterogeneous parallel architectures, as well as a broad suite of new graphics techniques, algorithms, and renderers that showcase the unprecedented visual quality possible with these systems. Neoptica’s solution makes possible the new era of programmable graphics: parallel CPU and GPU tasks cooperatively executing graphics algorithms while sharing complex, dynamic data structures. With Neoptica’s technology, graphics programmers are able to:

- treat all processors in the system as first-class participants in graphics computation;
- easily express concurrent computations that are deadlock-free, composable, and intuitive to debug;
- design custom graphics software pipelines, rather than being limited to the single pipeline exposed by current GPUs and graphics APIs; and
- design rendering algorithms that use dynamic, complex user-defined data structures for sparse and adaptive computations.

By enabling graphics programmers to fully leverage these new architectures and freeing them from the constraints of the predefined, one-way graphics pipeline, Neoptica’s system spurs the next generation of graphics algorithm innovation, with potential impact far greater than that of the programmable shading revolution of the last five years.

**Trends in Interactive Graphics**

The last five years have seen significant innovation in interactive graphics software and hardware. GPUs have progressed from being configurable fixed-function processors to highly-programmable data-parallel coprocessors, while CPUs have evolved from single-core to task-parallel multi-core processors. These changes have brought about three stages of interactive graphics programming:

*Programmable Graphics—The Future of Interactive Rendering*
• **Fixed function**: the GPU was configurable, but not programmable. Certain specialized states could be set to achieve simple visual effects (e.g. bump mapping) using multi-pass rendering techniques. The life-span of this stage was short due to its lack of flexibility and relatively high bandwidth demands.

• **Programmable shading**: the vertex and fragment processing stages of the GPU rendering pipeline could be programmed using small data-parallel programs called shaders. Using shaders, procedural techniques such as vertex skinning, complex texturing techniques, and advanced lighting models could be implemented efficiently on the GPU. This approach spurred a great deal of graphics algorithm innovation. However, existing graphics APIs and programming models limit developers to a fixed rendering pipeline and a small set of predefined data structures. Implementing custom rendering techniques that exploited more complex data structures was possible only with heroic developer effort, greatly increased code complexity, and high development costs.

• **Programmable graphics**: today, developers are on the threshold of being able to define custom interactive graphics pipelines, using a heterogeneous mix of task- and data-parallel computations to define the renderer. This approach enables complex data structures and adaptive algorithms for techniques such as dynamic ambient occlusion, displacement mapping, complex volumetric effects, and real-time global illumination that were previously only possible in offline rendering. However, current graphics programming models and tools are preventing the widespread transition to this era.

The promise of programmable graphics illustrates the fact that GPU programmability has implications for computer graphics far beyond simple programmable shaders. User-defined data structures and algorithms bring tremendous flexibility, efficiency, and image quality improvements to interactive rendering. Indeed, programmable graphics can be seen as completing the circle of GPGPU (general purpose computing on GPUs). Much of the recent innovation in using data structures and algorithms on GPUs has been driven by the application of GPUs to computational problems outside of graphics. In the era of programmable graphics, techniques developed for GPGPU are applied to the computational problems of advanced interactive graphics. By giving graphics programmers the ability to define their own rendering pipelines with custom data structures, programmable graphics brings far greater flexibility to interactive graphics programmers than is afforded even to users of today’s offline rendering systems.

A renderer’s ability to efficiently build and use dynamic, complex data structures relies on a mix of task- and data-parallel computation. The GPU’s data-parallel computation model, where the same operation is performed on a large number of data elements using many hardware-managed threads, is ideal for using data structures and for generating large amounts of new data. In contrast, the task-parallel compute model used by CPU-like processors provides an ideal environment in which to build data structures, perform global data analysis, and perform other more irregular computations. While it is possible to use only one processor type for all rendering computations, heterogeneous renderers that leverage the strengths of each use available hardware resources much more efficiently, and make interactive many techniques that would otherwise be limited to use in offline rendering alone.

The transition to programmable graphics is hampered by current graphics programming models and tools. Seemingly simple operations such as sharing data between the CPU and GPU, building pointer-based data structures on one processor for use on the other, and using complex application data in graphics computation currently require esoteric expertise. The specialized knowledge required severely limits the
number of developers who are able to creatively explore the capabilities of hardware systems, which has historically been the key driver of advancing the state of the art in interactive graphics.

The New Era Of Programmable Graphics

Neoptica has built a new system that moves beyond current GPU-only graphics APIs like OpenGL and Direct3D and presents a new programming model designed for programmable graphics. The system enables graphics programmers to build their own heterogeneous rendering pipelines and algorithms, making efficient use of all CPU and GPU computational resources for interactive rendering.

With Neoptica’s technology and a mixture of heterogeneous processing styles available for graphics, software developers have the opportunity to reinvent interactive graphics. Many rendering algorithms are currently intractable for interactive rendering with GPUs alone because they require sophisticated per-frame analysis and dynamic data structures. The advent of programmable graphics makes many of these approaches possible in real-time. New opportunities from the era of programmable graphics include:

• Feedback loops between GPU and CPU cores: with the ability to perform many round-trip per-pixel communications per frame, users can implement per-frame, global scene analyses that guide adaptive geometry, shading, and lighting calculations to substantially reduce unnecessary GPU computation.

• Complex user-defined data structures that are built and used during rendering: these data structures enable demand-driven adaptive algorithms that deliver higher-quality images more efficiently than today’s brute-force, one-way graphics pipeline.

• Custom, heterogeneous rendering pipelines that span all processor resources: for example, neither a pure ray-tracing approach nor a pure rasterization approach is the most efficient way to render complex visual effects like shadows, reflections, and global lighting effects; heterogeneous systems and programmable graphics will make it possible to easily select the most appropriate algorithm for various parts of the graphics rendering computation.

During the past year, Neoptica has built a suite of high-level programming tools that enable programmable graphics by making it easy for developers to write applications that perform sophisticated graphics computation across multiple CPUs and GPUs, while insulating them from the difficult problems of parallel programming. The system:

• uses a C-derived language for coordinating rendering tasks while using languages such as Cg and HLSL for GPU programming and C and C++ for CPU programming, integrating seamlessly with existing development practices and environments and providing for easy adoption;

• treats all processors in the system as first-class participants in graphics computation and enables users to easily share data structures between processors;

• presents a deadlock-free, composable parallel programming abstraction that embraces both data-parallel and task-parallel workloads;
provides intuitive, source-level debugging and integrated performance measurement tools.

This system has enabled the rapid development of new programmable graphics rendering algorithms and pipelines. Developers are able to design custom rendering algorithms and systems that deliver imagery that is impossible using the traditional hardware rendering pipeline, and deliver 10x to 50x speedups of existing GPU-only approaches.

Summary

We are at the threshold of a new era of interactive computer graphics. No longer limited to today’s brute-force, unidirectional rendering pipeline, developers will soon be able to design adaptive, demand-driven renderers that efficiently and easily leverage all processors in new heterogeneous parallel systems. New rendering algorithms that tightly couple the distinct capabilities of the CPU and the GPU will generate far richer and more realistic imagery, use processor resources more efficiently, and scale to hundreds of both CPU and GPU cores. Neoptica’s technology ushers in this new era of interactive graphics and makes it accessible to a large number of developers.