Programming GPUs

Outline

- History
- Contemporary GPUs
  - Programming model
  - Implementation
- GPGPU: non-graphical computation using GPUs

Caveat

- This is not a programming tutorial
- You will gain some understanding of the architectural issues that underlie programmable GPUs
GPU programmability is not new

Essentially all GPUs are (and have been) programmable
The questions are:
- Who gets to program them?
- And when?

Example systems from SGI’s early history:
- Geometry Engine (1981)
- GE4 (1987)
- VGXT Image Engine (1991)
- Dynamic code generation

Geometry engine (1981)

Four 32-bit floating-point units
- Jim-Clark format
- 2’s-complement e and m

Programmed using John Hennessy’s SLIM
(Stanford Language for the Implementation of Microcode)

~500-term PLA, hard-wired at fabrication time

Programmed by Jim Clark and Marc Hannah
SGI GE4 (1987)

Weitek 3332 floating-point ALU
~4k ~64-bit lines of microcode in SRAM
Programmed in low-level assembly code
No run-time code changes were made (though they could have been)
Programmed by several SGI employees, including Kurt

Wire-wrapped GE4 prototype

VGXT image engine (1991)

Tiny code space to implement
- Texture filter and application
- Frame buffer operations

Code overlays loaded on-the-fly
- Code sequences generated prior to run-time
- Fields (e.g., blend function) filled in at run time
Dynamic code generation

SUN graphics card
  ■ Bresenham lines

Abrash talk ...

Application programmability is not new

GPUs have a long history of application programmability

Example systems:
  ■ Ikonas graphics system (1978)
  ■ Trancept TAAC-1 (1987)
  ■ Multi-pass vector processing (2000)
  ■ Register combiners (2001)
Ikonas graphics system (1978)

Multi-board system
- 32-bit integer processor (16x16 multiply)
- 16-bit graphics processor (16 pixel parallel render)
- Hardware Z-buffer
- ...

Trancept TAAC-1 (1987)

C-language microcode compiler
200-bit microcode word
Sold to Sun Microsystems in 1987

“Customers got a C language microcode compiler and were strongly encouraged (!) to develop their own code” - Nick England, www.virhistory.com/trancept/trancept.html
Multi-pass vector processing (2000)

Treat OpenGL as a very long instruction word

Compute vector style

- Apply inst. to all pixels

Build up final image in many passes

Peercy, Olano, Airey, and Ungar, Interactive Multi-Pass Programmable Shading, SIGGRAPH 2000

(Figure adapted from the SIGGRAPH paper)

Register combinators (2001)

ARB_texture_env_combine

- “New texture environment function
  COMBINE_ARB allows texture combiner operations, including REPLACE, MODULATE, ADD, ...
  , INTERPOLATE.”

ARB_texture_env_combine

- “Adds the capability to use the texture color from other texture units as sources to the
  COMBINE_ARB environment function.”

ARB_texture_env_dot3

- Adds a dot-product operation to the texture combiner operations.
Summary of history

General-purpose processor with attached units
  - Could not compete in performance/cost ratio
  - "Heathkit" marketing

Multi-pass vector processing
  - Low performance
  - Wasteful use of bandwidth (against the trends)

Register combiners
  - Rube Goldberg design quality
  - Hopelessly increasing complexity

Programmable pipeline stages
  - Not accessible to application programmers
  - Reason (at SGI) was
    - Unable to keep the programming model consistent from product to product, and
    - Unable to use software to hide the inconsistencies

Contemporary GPU Programming Model

Application-programmable pipeline stages
Programming model (Direct3D 10 pipeline)

Traditional graphics pipeline (except location of geometry shader)
Three application-programmable stages operate on independent vertexes, primitives, and pixel fragments
All stages share same programming model, and (most) resources
Single shared memory, but memory objects remain distinct

Direct3D 10 programmable core

Complete instruction set
- Floating-point operations
- Integer operations (new)
- Branching operations
- Unlimited code length

Vector instructions
Lots of temporary state
Independent operation
- Each vertex, or
- Each primitive, or
- Each pixel fragment

From previous stage

To next stage
Vector instructions

Vector vs. scalar

- 4-component vector arithmetic is typical of gfx
- But as shaders get longer, an increasing fraction of their code is scalar

Some GPUs support dual-issue split vectors:

Programming interface

Assembly code

- Low level and error prone
- Not available in Direct3D 10

C-like shading language

- Cg: layered above assembly language
- HLSL, OpenGL Shading Language: intrinsic

Fixed-function stages

- Lost when a application-defined shader is used
- Not included in Direct3D 10
- Were to be omitted from OpenGL 2.0

PARAM mat[4] = { state.matrix.modelview }; 
DP4 result.x, vertex, mat[0]; 
DP4 result.y, vertex, mat[1]; 
DP4 result.z, vertex, mat[2]; 
DP4 result.w, vertex, mat[3]; 

void transform(float4 position : POSITION, 
out float4 oPosition : POSITION, 
uniform float4x4 modelView) 
{ 
oPosition = mul(modelView, position); 
}

Vertex transformation 
(Cg)
Parallel coding complexity is minimized

Complexities are handled by the implementation:

- Dependencies
  - Ordering
  - Sorting
- Scalability
  - Computation
  - Bandwidth
  - Load balancing (much more implementation work)

All threads are independent. Message passing is

- Not required
- Not allowed

Correctness constraints include:

- Cannot modify textures directly from shaders
- Cannot bind a target as both source and sink
  - Texture and render target
  - Vertex buffer and stream output
- Cannot “union” different data formats
Performance considerations

Programmer can largely ignore:

- Memory latency
- Floating-point cost (same as integer)
- Special function cost
- Texture filtering cost

Programmer should pay attention to:

- Divergent branching
- Number of registers used
- Size of storage formats (bandwidth)
- Re-specification of Z values (defeats Z cull)

Contemporary GPU Implementation
SIMD control

SIMD = Single Instruction, Multiple Data
One or more SIMD blocks; each block includes many data paths controlled by a single sequencer

SIMD benefits:
- Reduced circuitry
- Synchronization of events
  - Reduced instruction-fetch bandwidth
  - Better texture cache utilization

SIMD liabilities:
- Branch divergence reduces performance
- Branch convergence is complicated to implement

CS448 Lecture 9
Kurt Akeley, Pat Hanrahan, Spring 2007
Multiple threads hide memory latency

Scheduler maintains a queue of ready-to-run threads
- When a thread blocks, another is started

Blocking determination may be
- Simple (any attempted memory access), or
- Advanced (block only on data dependencies)

Thread context storage is significant - product of
- Number of processors,
- Threads per processor, and
- State per thread (registers, pc, ...)

Fixed-size memory pool may be insufficient for large thread contexts
- Large state/thread → too few threads/processor
- Local registers may not be “free”

Latency hiding consumes memory

Latency hiding mechanisms consume memory
- Cache (CPU, ~texture)
- Fragment fifo (texture pre-fetching)
- Shader context (threaded execution)

GPUs devote less die area to cache than CPUs
- But the real question is: what area is devoted to latency-hiding memory?
- Hint: ask the invited experts

From lecture 7
GeForce 8800 thread scheduling

Threads are scheduled in SIMD blocks
- 4 threads per quad (2x2 fragment unit)
- 16 threads per “warp” (NVIDIA GeForce 8800)

Thread scheduling is critical and complicated
- David Kirk assessment of 8800 load balancing: “3x more difficult overall”
- Sort-last-fragment, with load balancing for
  - Vertex processors
  - Geometry (primitive) processors
  - Pixel fragment processors

Reference: Mike Shebanow, ECE 498 AL: Programming Massively Parallel Computers, Lecture 12

Thread scheduling complexity

Two parts to scheduling decisions:
- What class of thread should be run?
  - Vertex, primitive, fragment
- When should a thread (group) be blocked?
  - When a memory access is attempted?
  - When a data dependency is reached?

Data-dependent blocking is worthwhile, because it conserves thread-context memory space
- Has been an ATI advantage over NVIDIA
- Did this contribute to 8800 scheduling difficulty?
Abstraction distance

Actual instruction set may not match “advertised” one
- Was true when assembly language was supported
  - Complex post-assembly run-time optimization
  - Layered languages (like Cg) -> 2-layer optimization
- Even easier with Direct3D 10
- Example: use scalar processor to emulate vector operations

Large abstraction distance wasn’t practical in the ’90s
- CPUs are much faster now
- Software technology has come a long way

Why isn’t Output Merger programmable?

OM is highly optimized
- Memory access is the GPU limiter in many cases
OM has data hazards
- All programmable interfaces are memory read-only

Input Assembler
  - Vertexes
  - Sampler
  - Constant
  - Texture

Vertex Shader
  - Sampler
  - Constant
  - Texture

Geometry Shader
  - Sampler
  - Constant
  - Texture

Rasterizer
  - Memory

Pixel Shader
  - Sampler
  - Constant
  - Texture

Output Merger
  - Target

Application programmable
GPGPU

Performance motivates interest in GPGPU

7x performance advantage to GPUs today:

- 47 GFLOPS - Intel core2 extreme e6800
- 345 GFLOPS - NVIDIA GeForce 8800 GTX
- 350 GFLOPS - ATI X1900
Rate-of-change of performance is the key

CPU and GPU rates differ:
- GPU: ~2x CAGR
- CPU: <1.5x CAGR, recently much less

Will this 10x/decade disparity persist in the multi-core era?

Additional GPGPU motivations

Visible success stories
- Folding@home gets 20-40x speedup over PCs
- Matrix multiply achieves 120 GFLOPS on ATI X1900

Ubiquity of GPU hardware
- Every PC has a GPU of some kind
- This is a first for an attached hardware accelerator

Evolving GPU architecture
- Rich instruction set
- Highly parallel
- Improving GPU→CPU data transfer rates
- ...
GPUs are high-performance multiprocessors

GeForce 8800 block diagram (NVIDIA Corporation 2006)

Scatter, Gather, and Reduction

Frame buffer mindset: fragment associated with a specific region of memory

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather</td>
<td>Texture fetch</td>
<td>Texture fetch</td>
</tr>
<tr>
<td>Scatter</td>
<td>--</td>
<td>Uses fragment sort</td>
</tr>
<tr>
<td>Reduction</td>
<td>--</td>
<td>Occlusion query</td>
</tr>
</tbody>
</table>
GPGPU concerns

Application specificity. Requires:
- High arithmetic intensity (compute / bandwidth)
- High branching coherence
- Ability to decompose to 1000s of threads

Computation quality
- No memory error detection/correction
- No double precision
- Incomplete (but improving) IEEE single precision

Possible end of divergence of CPU and GPU performance
Abysmal history of attached processor success

How will CPUs and GPUs coexist?

Possible outcomes:
- CPU instruction set additions
- Heterogeneous multi-core
  - Shared memory
  - Differentiated memory
- Generalized GPUs

What do/will we mean by GPU?
- Anything graphics related?
- Massively multi-threaded to hide memory latency?
Summary

Long history of GPU programmability, but current situation (app-programmable pipeline stages) is new

Current GPUs are easily programmed (for graphics) and give programmers amazing power

Current GPUs are high-performance multiprocessors

Exciting things are happening in computer architecture, and GPUs will play an important role

Suggested readings


Real-Time Graphics Architecture

Lecture 9: Programming GPUs

Kurt Akeley
Pat Hanrahan

http://graphics.stanford.edu/cs448-07-spring/