Ian Buck

Graphics Lab
Stanford University

Aaron Lefohn

Institute for Data Analysis
and Visualization
University of California, Davis

John Owens

Institute for Data Analysis
and Visualization
University of California, Davis

Robert Strzodka

Caesar Institute
Bonn, Germany

GPGPU: General Purpose Computation on Graphics Processors
IEEE Visualization 2004 Tutorial

October, 2004

Abstract

In the last three years, commodity graphics processors (GPUs) have evolved from fixed-function graphics units into powerful data-parallel processors. These streaming processors are capable of sustaining computation rates of greater than ten times that of a single CPU. Researchers in the evolving field of general-purpose computation on graphics processors (GPGPU) have demonstrated mappings to these processors for a wide range of computationally intensive tasks. Examples include ray tracing, molecular dynamics, and surface processing. This tutorial provides a detailed introduction and overview of the GPGPU field to the visualization community. Attendees will gain an understanding of modern GPU architecture, the GPGPU programming model, and the techniques and tools required to apply GPUs to their own applications.

This tutorial will be of interest to the visualization community for several reasons. First, GPU acceleration of partial differential equation solvers, 2D and 3D image processing, and physical simulations directly affects the visualization community. Examples of this are the GPU-based interactive 3D segmentation algorithms published at IEEE Visualization last year. Second, until recently visualization has primarily focused on exploration of pre-captured data. The ability to perform GPGPU-based interactive simulation on a desktop PC, however, opens up a wealth of new visualization research challenges. Lastly, despite recent advances in GPU programming languages, GPGPU practitioners are predominantly graphics specialists. This tutorial presents the background, tools, and implementation details required for researchers in other fields to leverage the computational power of GPUs.

The tutorial speakers are experts in the field of general-purpose computation on GPUs and streaming architectures. They have presented papers, conference courses, and university courses on the topic at IEEE Visualization, SIGGRAPH, Graphics Hardware, Stanford, UCDavis, and elsewhere.

Organizer

Aaron Lefohn University of California, Davis

Name of Speakers

Ian Buck	Stanford University
Aaron Lefohn	University of California, Davis
John Owens	University of California, Davis
Robert Strzodka	Caesar Institute, Bonn, Germany

Length

Full day

Level

Intermediate

Speaker Biographies

Ian Buck is a fifth year Ph.D. candidate in Computer Science at the Stanford University Graphics Lab researching general-purpose computing models for GPUs. His research focuses on programming language design for graphics hardware as well as general computing applications which map to graphics hardware architectures. His latest work includes Brook, a general-purpose data-parallel programming language, and a complete compiler and runtime system for compiling to programmable graphics hardware. Ian has given numerous talks on general-purpose computation and is a speaker at the GPGPU course at ACM SIGGRAPH 2004. He received his B.S.E. in Computer Science from Princeton University in 1999 and is a recipient of Stanford School of Engineering and NVIDIA fellowships.

Aaron Lefohn is a Ph.D. student in computer science at the University of California at Davis and a part-time graphics software engineer at Pixar Animation Studios. His Ph.D. research focuses on general-purpose computation on graphics hardware and high-quality interactive rendering. Aaron's masters thesis research focused on the acceleration of level-set deformable surface models using graphics hardware. He has given talks on the subject of general-purpose computation on graphics hardware at SIGGRAPH, IEEE Visualization, MICCAI, IBM Watson Research, ATI, NVIDIA and Pixar. Aaron completed an M.S. in computer science at the University of Utah in 2003, an M.S. in theoretical chemistry from the University of Utah in 2001, and a B.A. in chemistry from Whitman College in 1997. Aaron is an NSF graduate fellow in computer science.

John Owens is an assistant professor of electrical and computer engineering at the University of California, Davis, where he leads projects in graphics hardware and sensor networks. He earned his Ph.D. in electrical engineering in 2003 from Stanford University, where he was an architect of the Imagine Stream Processor. His dissertation work concentrated on graphics on stream architectures such as Imagine. He also holds a M.S. in electrical engineering (Stanford University, 1997) and a B.S. in electrical engineering and computer sciences (University of California, Berkeley, 1995).

Robert Strzodka is a fifth year Ph.D. candidate in Numerical Mathematics at the University Duisburg-Essen. He was supported by a research fellowship of the German National Academic Foundation and received his diploma in Mathematics with distinction from University Bonn in 1999. His research focuses on error analysis and efficient implementation of PDE solvers on data-stream-based architectures including graphics cards, FPGAs and reconfigurable computing machines. In 2000 he was among the first to demonstrate the efficient use of graphics hardware for PDE based image processing. Currently he explores the enrichment of data stream processing with adaptive numerical concepts.

GPGPU Tutorial Schedule

Section 1: Introduction			
8:30	Introduction and Tutorial Overview	Lefohn	A
9:00	A Data-Parallel Genealogy: The GPU's family tree	Owens	B
Section 2: GPGPU Programming			
9:30	The Programming Model	Lefohn	C
10:00	Break		
10:30	GPGPU Programming Languages	Buck	D
Section 3: GPGPU Computational Primitives			
11:20	Mathematical Primitives	Strzodka	E
12:15	Lunch		
1:45	General Algorithmic Primitives	Owens	F
Section 4: "Getting Your Hands Dirty"			
2:15	Data Formatting and Addressing	Lefohn	G
2:45	Computation Tips and Tricks	Buck	H
3:15	Developer Tools	Strzodka	I
3:45	Break		
Section 5: Case Studies			
4:15	Level-Set Surface Deformation	Lefohn	J
4:35	Advanced Image Processing	Strzodka	K
3:55	Ray Tracing and Molecular Dynamics	Buck	L
Section 6: Conclusions			
5:15	The Future	Owens	M
5:30	Open Question and Answer	All	

GPGPU

VIS 2004 austin, texas October 10-15

Ian Buck
Graphics Lab
Stanford University

Aaron Lefohn
Institute for Data Analysis and Visualization
University of California, Davis

John Owens
Institute for Data Analysis and Visualization
University of California, Davis

Robert Strzodka
Caesar Institute
Bonn, Germany

Introduction and Overview

GPGPU

Aaron Lefohn
Institute for Data Analysis and Visualization
University of California, Davis

VIS 2004

Motivation

- Challenge Statement
  - GPGPU signifies the dawn of the data-parallel desktop computing age

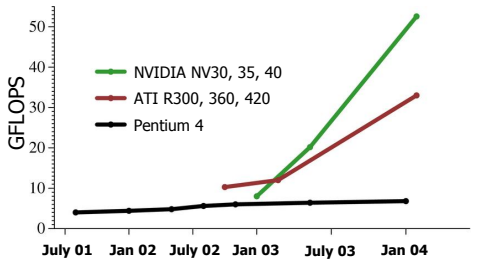
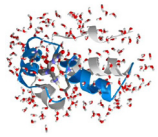
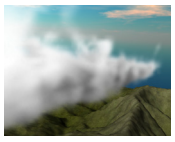


Figure courtesy of Ian Buck, Stanford University

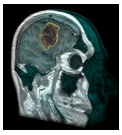
Motivation: GPU Compute Power



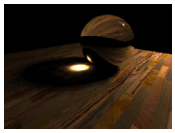
Molecular Dynamics (Buck)



Cloud Simulation (Harris)



Level-Set Surface Deformation (Lefohn)



Photon Mapping (Purcell)

Tutorial Introduction

- General-Purpose Computation on GPUs (GPGPU)
  - Modern GPUs can accelerate “appropriate“ algorithms >10x
  - Achieving this speedup currently requires a large amount of GPU-specific knowledge—We want to change this!
- Tutorial goal
  - Give visualization researchers/engineers the knowledge and tools to leverage the computational power of the GPU

Tutorial Topics

- What kinds of algorithms map well to GPUs?
- Why are GPUs faster than CPUs?
- GPGPU implementation details
  - Programming model
  - Basic building blocks
  - Nitty-gritty details
  - Real-world examples
- The future of GPGPU

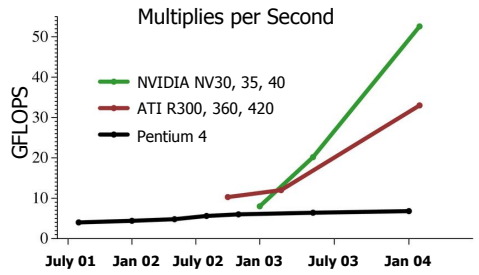
Motivation: Why GPGPU?

- Beginning of data-parallel desktop computing age
  - GPUs are the first **commodity, data-parallel** architecture
  - Advatages of data-parallelism
    - GPUs are >10x faster than CPU for appropriate problems
    - GPU performance increasing faster than CPU performance
  - Advantages of commodity
    - GPUs are inexpensive
    - GPUs are ubiquitous: Desktops, laptops, PDAs, cell phones
    - 1980‘s data-parallel architectures cost millions of dollars

Motivation: Why GPGPU Now?

- GPU feature set only recently matured
  - Programmability 2001
  - Read/write memory 2001
  - Floating point 2002
  - Conditional execution (?) 2004...
- GPU evolution driven by computer game market
- GPU power rapidly increasing

Motivation: GPU Compute Power



- Figure courtesy of Ian Buck

Brief History of GPGPU

See http://www.gpgpu.org for a more complete history

1990	Lengyel	Motion planning
1999	Hoff	Voronoi diagrams
2000	Peercy	Renderman with OpenGL
2001	Strzodka	2D PDE image processing
2002	Purcell / Carr Harris	Ray tracing Cellular automata
2003	Krueger / Boltz / Goodnight Lefohn Rottger / Krueger	Linear algebra 3D level-set solver Ray casting
2004	Govindaraju Buck / McCool	Database operations GPGPU languages

Brief History of GPGPU

- Where are we now?
  - Transitioning from "What can we do" to "What should we do"
  - Examples
    - Lefohn et al., Univ. of Utah technical report 2002
      - Unpublished brute force solution gave no speedup
    - Buck et al., Siggraph 2004, BrookGPU Language
      - Clarifies GPGPU programming model
    - Fatahalian et al., Graphics Hardware 2004
      - Describes why matrix-matrix multiplication can never be fast on current GPUs

Motivation: Why GPGPU Vis Tutorial?

- Visualization community can benefit from GPGPU
  - 2D and 3D image processing
    - Segmentation, registration, tone mapping, …
  - Simulation
  - New rendering algorithms
  - Interactive "Visulation"
  - Familiarity with graphics programming makes transition easier
- Harnessing power of GPU is hard
  - Data-parallel algorithm mappings
  - Mapping from graphics primitives to compute primitives
  - Many performance pitfalls

Tutorial Prerequisites

- We assume
  - Basic knowledge of interactive graphics and graphics hardware
  - Basic knowledge of vertex and fragment shaders
- Target audience
  - Researchers interested in GPGPU
  - Engineers interested in GPU acceleration of their applications
  - Attendees wishing a survey of this exciting new field

Tutorial Speakers (Alphabetical)

- Ian Buck
  - Ph.D. student, Pat Hanrahan
  - Stanford University
- Aaron Lefohn
  - Ph.D. student, John Owens
  - University of California, Davis
  - Graphics software engineer, Pixar Animation Studios
- John Owens
  - Assistant professor, Electrical and Computer Engineering
  - University of California, Davis
  - Ph.D., Bill Dally and Pat Hanrahan, Stanford University
- Robert Strzodka
  - Staff researcher, Caesar Institute, Bonn, Germany
  - Ph.D., Martin Rumpf, University of Duisburg

Tutorial Schedule Overview

- Morning
  - Introduction
  - GPU/data-parallel architecture overview
  - GPGPU programming model and languages
  - Computational building blocks
- Afternoon
  - “Getting your hands dirty: Making it work”
  - Case studies
  - The future
  - Q&A

Tutorial Schedule

- Section 1: Introduction
  - 8:30 Introduction and Tutorial Overview — Lefohn
    Motivation, introduction, and overview of the day
  - 9:00 A Data-Parallel Genealogy: The GPU’s family tree — Owens
    Data-parallel architectures, stream processing, GPUs
- Section 2: GPGPU Programming
  - 9:30 The Programming Model — Lefohn
    Building computational primitives out of OpenGL calls
  - 10:00 Break
  - 10:30 GPGPU Programming Languages — Buck
    Beyond OpenGL: Data-parallel GPU languages

Tutorial Schedule

- Section 3: GPGPU Computational Primitives
  - 11:20 Mathematical Primitives — Strzodka
    Linear algebra, PDEs, FEMs
  - 12:15 Lunch
  - 1:45 General Algorithmic Primitives — Owens
    Sorting, Searching

Tutorial Schedule

- Section 4: "Getting Your Hands Dirty"
  - 2:15 Data Formatting and Addressing — Lefohn
    GPU memory model and data structures
  - 2:45 Computation Tips and Tricks — Buck
    Performance tips, working around GPU limitations
  - 3:15 Developer Tools — Strzodka
    Compiling, debugging, profiling
- 3:45 - 4:15 Break

Tutorial Schedule

- Section 5: Case Studies
  - 4:15 Level-Set Surface Deformation — Lefohn
    Computation and visualization of dynamic, sparse PDEs
  - 4:35 Advanced Image Processing — Strzodka
    Registration, segmentation, and skeletons
  - 3:55 Ray Tracing and Molecular Dynamics — Buck
    Particle simulations and light transport

Tutorial Schedule

- Section 6: Conclusions
  - 5:15 The Future — Owens
    The future of commodity data-parallel computing
  - 5:30 Open Question and Answer — All
    Q & A

Coming Next…

- "A Data-Parallel Genealogy: The GPU's Family Tree"
  - John Owens
  - Introduction to data-parallel and streaming architectures
  - The bigger picture of GPGPU

A Data-Parallel Genealogy: The GPU Family Tree

GPGPU

John Owens
Department of Electrical and Computer Engineering
Institute for Data Analysis and Visualization
University of California, Davis

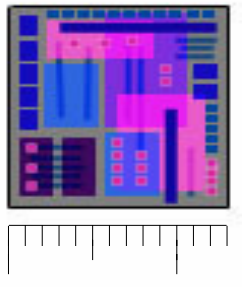
VIS 2004

Outline

- Moore's Law brings opportunity
  - Gains in performance …
  - … and capabilities.
  - What has 20+ years of development brought us?
- How can we use those transistors?
  - Microprocessors?
  - Stream processors
    - Today's commodity stream processor: the GPU

The past: 1987

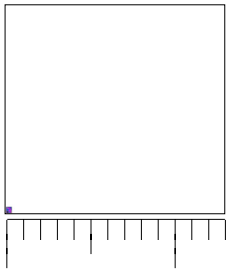
20 MIPS CPU
1987



[courtesy Anant Agarwal]

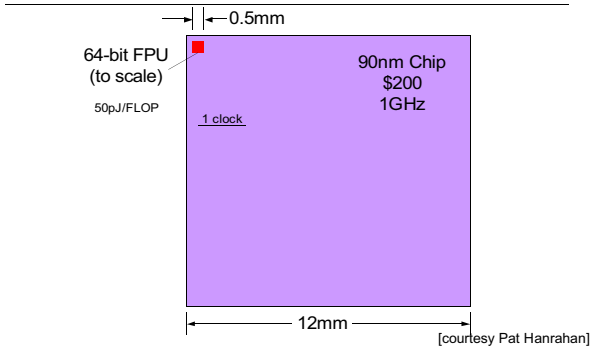
The future: 2007

1 Billion Transistors
2007



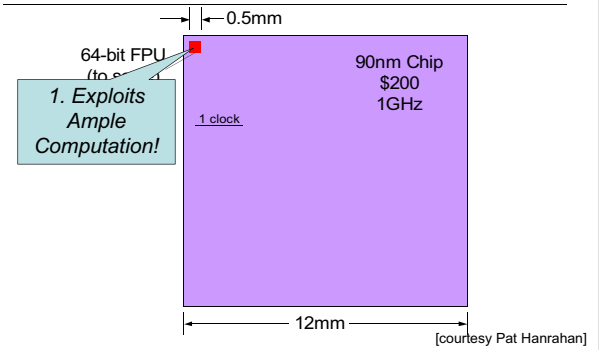
[courtesy Anant Agarwal]

Today's VLSI Capability



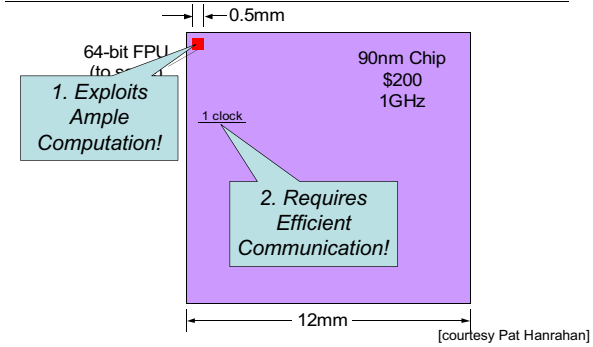
[courtesy Pat Hanrahan]

Today's VLSI Capability



[courtesy Pat Hanrahan]

Today's VLSI Capability



[courtesy Pat Hanrahan]

SGI Historicals (Depth Buffered)

Year	Product	Zbuf rate	Yr rate	ZTri rate	Yr rate
1984	Iris 2000	100K	-	0.8K	-
1988	GTX	40M	4.5	135K	3.6
1992	RealityEngine	380M	1.8	2M	2.0
1996	InfiniteReality	1000M	1.3	12M	1.6
			2.2		2.2

… yearly growth well above Moore's Law!

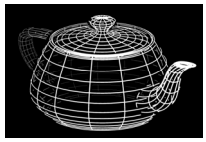
NVIDIA Historicals

Season	Product	Fill rate	Yr rate	Tri rate	Yr rate
2H97	Riva 128	20M	-	3M	-
1H98	Riva ZX	31M	2.4	3M	1.0
2H98	Riva TNT	50M	2.6	6M	4.0
1H99	TNT2	75M	2.3	9M	2.3
2H99	GeForce	120M	2.6	15M	2.8
1H00	GeForce2	200M	2.6	25M	2.8
2H00	NV16	250M	1.6	31M	1.5
1H01	NV20	500M	4.0	30M	1.0
			2.5		2.2

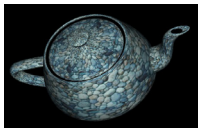
… yearly growth well above Moore's Law!

GPU History: Features


< 1982


1982-87


1987-92


1992-2000


2000-

Outline

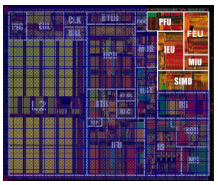
- Moore's Law brings opportunity
  - Gains in performance …
  - … and capabilities.
  - What has 20+ years of development brought us?
- How can we use those transistors?
  - Microprocessors?
  - Stream processors
    - Today's commodity stream processor: the GPU

Characteristics of Our Applications

- Lots of arithmetic
- Lots of parallelism
- Multiple stages
- Simple control
- Latency-tolerant / deep pipelines

Microprocessors: A Solution?

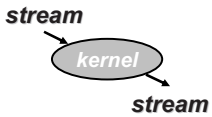
- Microprocessors address a different application space
  - Scalar programming model with no native data parallelism
    - Excel at control-heavy tasks
    - Not so good at data-heavy, regular applications
  - Few arithmetic units – little area
  - Optimized for *low latency* not *high bandwidth*

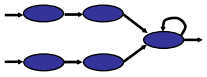


Pentium III – 28.1M T

Stream Programming Abstraction

- Let's think about our problem in a new way
- Streams
  - Collection of data records
  - All data is expressed in streams
- Kernels
  - Inputs/outputs are streams
  - Perform computation on streams
  - Can be chained together



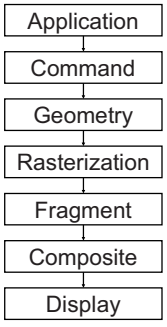


Why Streams?

- Ample computation by exposing parallelism
  - Streams expose data parallelism
    - Multiple stream elements can be processed in parallel
  - Pipeline (task) parallelism
    - Multiple tasks can be processed in parallel
- Efficient communication
  - Producer-consumer locality
  - Predictable memory access pattern
    - Optimize for throughput of all elements, not latency of one
    - Processing many elements at once allows latency hiding
  - High arithmetic intensity

Graphics Apps are Stream Apps

- Lots of arithmetic
- Lots of parallelism
- Multiple stages
- Feed forward pipelines
- Latency-tolerant / deep pipelines

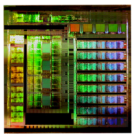


Taxonomy of Streaming Processors

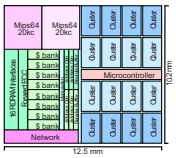
- In common:
  - Exploit parallelism for high computation rate
    - Each stage processes many items in parallel (d.p.)
    - Several stages can run at the same time (t.p.)
  - Efficient communication
    - Minimize memory traffic
    - Optimized memory system
- What's different?
  - Mapping of processors to tasks in graphics pipeline

Stream Processors

- Fewer compute units than tasks
  - "Time multiplexed" organization
  - Each stage fully programmable
- Stanford Imagine
  - 32b stream processor for image and signal processing (2001)
- Stanford Merrimac
  - 64b stream processor for scientific computing (2004)
  - Core of Stanford Streaming Supercomputer
- Challenge:
  - Efficiently mapping all tasks to one processor - no specialization

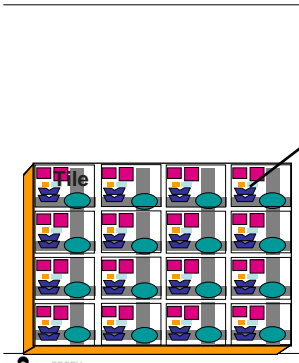


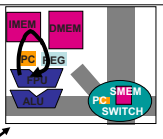
[Stanford Imagine - 2001]



[Stanford Merrimac - 2004]

MIT RAW: Tiled Processor Architecture

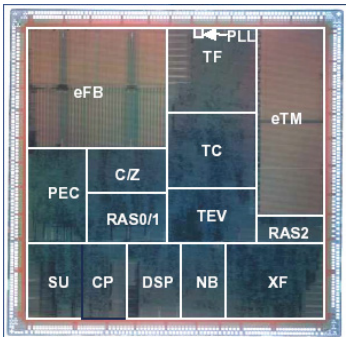




- More processors than tasks
  - MIT RAW, IBM Cell
- Each tile is programmable
  - Streams connect tiles
- "Task parallel" organization
- Lots of ALUs and registers
- Short, programmable wires
- Challenge: Software support

[courtesy Anant Agarwal]

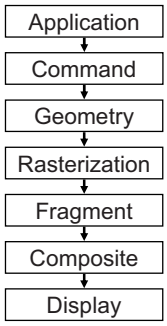
GPU: Special-Purpose Graphics Hardware



- Task-parallel organization
- Each module hardwired to specific task - huge performance advantage!
- Provides ample computation resources
- Efficient communication patterns
- Dominant graphics architecture

[ATI Flipper – 51M T]

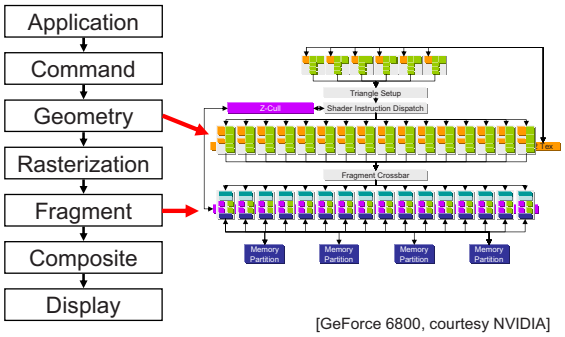
Today's Graphics Pipeline



- Graphics is well suited to:
  - The stream programming model
  - Stream hardware organizations
    - Especially GPUs!
- What if we could apply these techniques to more general-purpose problems?
- On appropriate problems, hardware with
  - ample computation and
  - efficient communication

  should excel!
- What's missing?

The Programmable Pipeline



[GeForce 6800, courtesy NVIDIA]

Conclusions

- Adding programmability to GPUs is exciting!
  - GPUs have great performance
    - Computation & communication
  - Programmability allows them to address many interesting problems
- Many challenges remain …
  - Algorithms, programming models, architecture, languages, tools …
- Next speaker:
  - Aaron Lefohn
  - "The GPGPU Programming Model"

The GPGPU Programming Model

Aaron Lefohn

Institute for Data Analysis and Visualization
University of California, Davis

Overview

- Data-parallel programming basics
- The GPU as a data-parallel computer
- "Hello World" GPGPU Example
- Emulating missing functionality
- Conclusions

Data-Parallel Programming Basics

- Serial programming of loops
  - Instructions and data access are intermixed

```
forEach data element, i
  executeLoopBody(i)
```

- Data-parallel programming
  - Separate specification of data and instructions

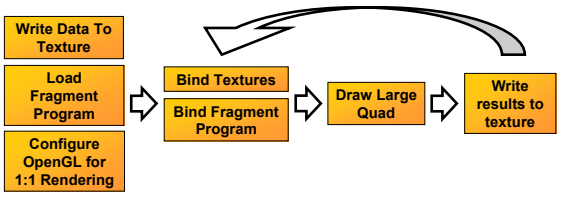
```
dataStream   = specifyAllData()
kernel       = specifyLoopBody()
forEach( dataStream, kernel )
```


Explicit Data-Parallelism

- Why write explicitly data-parallel programs?
  - Instructions stay fixed while data streams past
  - Independent computations on each data element
    - "forEach" call is parallelizable
    - Hide cost of memory access with parallelism

GPU as a Data-Parallel Computer

- Data specification → Textures
- Kernel specification → Fragment program
- `forEach` execution → Draw single large quad



GPU Computational Primitives

- Operations available in kernel
  - Read-only memory (input streams) — Texture sampler
  - Random access read-only (gather) — Texture sampler
  - Per-data-element interpolants — Varying registers
  - Temporary storage (no saved state) — Local registers
  - Read-only constants — Constant registers
  - Write-only memory (result streams) — Render-to-texture
  - Floating-point ALUops

GPU Computational Primitives

- What's missing?
  - No stack
  - No heap
  - No integer or bitwise operations
  - No scatter (`a[i] = b`)
  - No reduction operations (max, min, sum)
  - Data-dependent conditionals
- Why missing?
  - Parallelism, parallelism, parallelism
  - Lack of demand from games
  - Early in GPU evolution as general data-parallel processor

GPU Computational Primitives

- Handling missing features
  - We'll explain how to emulate
    - Scatter
    - Global (reduction) operations
    - Conditionals

“Hello World“ GPGPU Example

- 3 x 3 Image processing convolution
- CPU version

```
image = loadImage( WIDTH, HEIGHT );
blurImage = allocZeros( WIDTH, HEIGHT );

for (x=0; x < WIDTH; x++)
  for (y=0; y < HEIGHT; y++)
    for (i=-1; i <= 1; i++)
       for (j=-1; j <= 1; j++)
          float w = computeWeight(i,j);
          blurImage[x][y] += w * image[x+i, y+j];
```


“Hello World“ GPGPU Example

- GPU Version
  1) Load **`image`** into texture

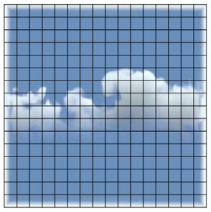


Figure courtesy of Mark Harris

  2) Create **`blurImage`** texture to hold result

“Hello World“ GPGPU Example

- GPU Version
  3) Load fragment program (kernel)
     Example shown in Cg

```
float4 blurKernel( uniform samplerRECT image,
                   float2      winPos : WPOS,
                   out float4  blurImage )
{
   blurImage = float4(0,0,0,0);

   for (i=-1; i <= 1; i++) {
     for (j=-1; j <= 1; j++) {
       float2 texCoord = winPos + float2(i,j);
       float  w        = computeWeight(i,j);
       blurImage += w * texRECT( image, texCoord );
     }
   }
}
```


“Hello World“ GPGPU Example

- GPU Version
  4) Configure OpenGL to draw 1:1
     No projection or rescaling

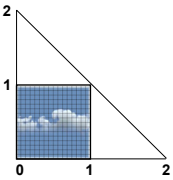
```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluOrtho2D(0, 1, 0, 1);
glViewport(0, 0, WIDTH, HEIGHT );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
```

  5) Bind **`image`** and **`blurKernel`** (texture and fragment program)
  6) Bind **`blurImage`** as render target

“Hello World“ GPGPU Example

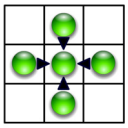
- GPU Version
  7) Execute kernel on each stream element
     Draw quad of size [`WIDTH` x `HEIGHT`]

```
glBegin( GL_TRIANGLES );
   glVertex2f(0, 0);
   glVertex2f(2, 0);
   glVertex2f(0, 2);
glEnd();
```



“Hello World“ GPGPU Example

- What happened?
  - `blurKernel` executed on each element of `image`
    - Rendering replaced outer two loops of CPU version
  - `blurKernel` performed *gather* operation at each element



Gather

  - Result (`blurImage`) was written to framebuffer / texture

“Hello World“ GPGPU Example

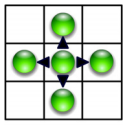
- Get the source code for GPGPU examples
  - http://www.gpgpu.org/developer/
  - http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html
    - gpgpu_fluid
    - gpgpu_disease
    - gpu_particles
  - http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/HW_Image_Processing.html

Emulating Scatter

- Scatter

```
i = foo();
a[i] = bar();
```

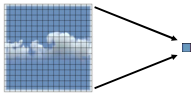
- Solution 1
  - Transform scatter algorithm into gather algorithm
  - See Ian’s “Tips and Tricks“ for more details
- Solution 2
  - Do actual scatter
  - Vertex processor can scatter points
    - Render points instead of large triangle
    - Render-to-texture with vertex-texture-reads (PS 3.0)
    - Or render-to-vertex-array
  - Problem
    - Drawing a point for each data element is slow



Scatter

Emulating Reduction Operations

- Reductions
  - Operations that require all data elements
  - max, min, sum, norm, etc.



Cloud figure courtesy of Mark Harris

- Solution
  - Perform repeated gathers until only single data value left
  - `log( WIDTH )` gather operations (assuming `WIDTH == HEIGHT`)
- Problem
  - Extra passes can be costly

Conditionals In a Data-Parallel World

- Data-parallel execution and conditionals are at odds
  - Ideal data-parallel model assumes all data elements processed identically
  - Conditional execution breaks this assumption
- Solutions
  1) Statically resolve conditional with substreams
  2) Occlusion query
  3) Early z-culling
  4) Better hardware?

Conditionals in a Data-Parallel World

- Solutions
  1) Can decision be made before fragment processor?
     Static branch resolution with substreams

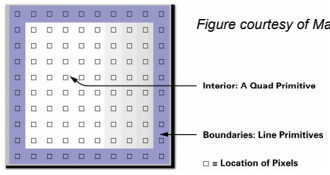


Figure courtesy of Mark Harris

Trick "discovered" by Lefohn, Harris, and Goodnight Simultaneously in 2003

Conditionals in a Data-Parallel World

- Solution
  2) Occlusion query
- Idea
  - Occlusion query reports the number of fragments that passed the depth test
  - Useful when number of loop iterations is data-dependent
  - Kernel kills fragments that do not need further processing
  - CPU continues to issue render until all fragments are killed

Conditionals in a Data-Parallel World

- Solution
  3) Early depth cull
- Idea
  - Modern GPUs can kill fragments before kernel execution
  - Kernel sets z-value to control whether or not execution occurs
- Problem
  - Conditionals must be block-coherent
  - More about this in Ian's "Tips and Tricks" talk

Conditionals in a Data-Parallel World

- Solution
  4) Better hardware?
  - MIMD hardware
    - NVIDIA GeForce 6-series (NV4x) vertex processors
  - SIMD-with-conditional-support
    - NVIDIA GeForce 6-series (NV4x) fragment processor
    - Uniform branches are a win if more than ~5 instructions
    - Varying branches must be coherent across hundreds of pixels
      - Only 10s x 10s pixels—Very useful in many cases
- Problem
  - Do we want more SIMD processors or fewer MIMD processors?

Conditionals in a Data-Parallel World

- Conclusions
  - Conditionals are tough with today's GPUs
    - Best case is when conditional can be staticaly resolved and removed from computational kernel
  - Future GPUs will most likely fully support conditionals
    - Solution must not interfere with parallelism
    - MIMD?
    - SIMD with "Conditional Streams?"
      - Kapasi et al., Micro 33, 2000

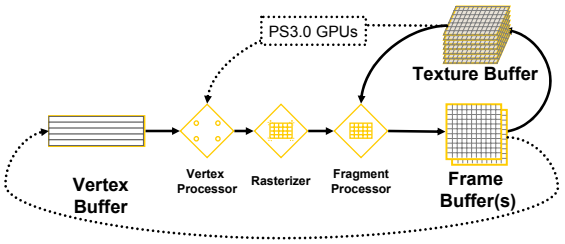
Conclusions

- GPGPU computational basics
  - Textures → Storage for data streams
  - Fragment program → Computational kernel
  - Render pass → `forEach` loop over data stream
- Coming next...
  - Ian Buck
  - Data-parallel languages for GPGPU programming
    - Express data-parallel programs more elegantly than `glBegin()...glEnd()`

References

- ATI Developer web site, http://www.ati.com/developer/
- GPGPU Developer web site, http://www.gpgpu.org/developer
- N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware," Graphics Hardware 2003
- M. Harris, W. Baxter, T. Scheuermann, A. Lastra, "Simulation of Cloud Dynamics on Graphics Hardware," Graphics Hardware 2003
- Hillis et al., "Data Parallel Algorithms," Comm. ACM, 29(12), December 1986
- Kapasi et al., "Efficient Conditional Operations for Data-parallel Architectures," In Proc. of the 33rd Ann. Int'l Symp. on Microarchitecture, pages 159--170, 2000
- A. Lefohn, J. Kniss, C. Hansen, R. Whitaker, "A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets," IEEE Transactions on Visualization and Computer Graphics 2004
- NVIDIA Developer web site, http://developer.nvidia.com/page/home

Modern Graphics Pipeline



High level languages

GPGPU

Ian Buck

Graphics Lab
Stanford University

GPGPU Languages

- Why do want them?
  - Make programming GPUs easier!
    - Don't need to know OpenGL, DirectX, or ATI/NV extensions
    - Simplify common operations
    - Focus on the algorithm, not on the implementation
- Sh
  - University of Waterloo
  - http://libsh.sourceforge.net
  - http://www.cgl.uwaterloo.ca
- Brook
  - Stanford University
  - http://brook.sourceforge.net
  - http://graphics.stanford.edu/projects/brookgpu

Sh Features

- Implemented as C++ library
  - Use C++ modularity, type, and scope constructs
  - Use C++ to metaprogram shaders and kernels
  - Use C++ to sequence stream operations
- Operations can run on
  - GPU in JIT compiled mode
  - CPU in immediate mode
  - CPU in JIT compiled mode
- Can be used
  - To define shaders
  - To define stream kernels
- No glue code
  - To set up a parameter, just declare it and use it
  - To set up a texture, just declare it and use it
- Memory management
  - Automatically uses pbuffers and/or uberbuffers
  - Arrays simulated with textures Textures can encapsulate interpretation code
  - Programs can encapsulate texture data
- Program manipulation
  - Introspection
  - Uniform/varying conversion
  - Program specialization
  - Composition & concatenation
  - Interface adaptation
- Free and Open Source
- http://libsh.sourceforge.net

Sh Fragment Shader

```
fsh = SH_BEGIN_PROGRAM("gpu:fragment") {
  ShInputNormal3f nv;        // normal (VCS)
  ShInputVector3f lv;        // light-vector (VCS)
  ShInputVector3f vv;        // view vector (VCS)
  ShInputColor3f ec;         // irradiance
  ShInputTexCoord2f u;       // texture coordinate

  ShOutputColor3f fc;        // fragment color

  vv = normalize(vv);
  lv = normalize(lv);
  nv = normalize(nv);
  ShVector3f hv = normalize(lv + vv);
  fc = kd(u) * ec;
  fc += ks(u) * pow(pos(hv|nv), spec_exp);
} SH_END;
```

Streams and Channels

- `ShChannel<element_type>`
  - Sequence of elements of given type
- `ShStream`
  - Sequence of channels
  - Combine channels with &:

```
ShStream s = a & b & c;
```

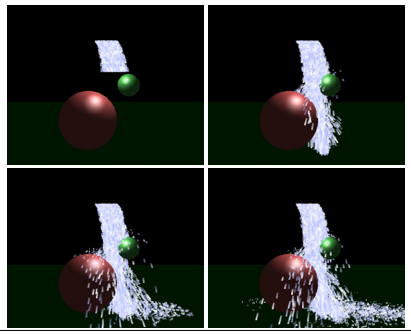
  - *Refers* to channels, does *not* copy
  - Single channel also a stream
- Apply programs to streams with <<

```
ShStream t = (x & y & z);
s = p << t;
(a & b & c) = p << (x & y & z);
```


Stream Processing: Particles

```
// SETUP (define particle state update kernel)
p = SH_BEGIN_PROGRAM("gpu:stream") {
  ShInOutPoint3f Ph, Pt;
  ShInOutVector3f V;
  ShInputVector3f A;
  ShInputAttrib1f delta;
  Pt = Ph;
  A = cond(abs(Ph(1)) < 0.05,
    ShVector3f(0.,0.,0.), A);
  V += A * delta;
  V = cond((V|V) < 1.,
    ShVector3f(0., 0., 0.), V);
  Ph += (V + 0.5*A)*delta;
  ShAttrib1f mu(0.1), eps(0.3);
  for (i = 0; i < num_spheres; i++) {
    ShPoint3f C = spheres[i].center;
    ShAttrib1f r = spheres[i].radius;
    ShVector3f PhC = Ph - C;
    ShVector3f N = normalize(PhC);
    ShPoint3f S = C + N*r;
    ShAttrib1f collide =
      ((PhC|PhC) < r*r)*((V|N) < 0);
    Ph = cond(collide,
      Ph - 2.0*((Ph - S)|N)*N, Ph);
    ShVector3f Vn = (V|N)*N;
    ShVector3f Vt = V - Vn;
    V = cond(collide,
      (1.0 - mu)*Vt - eps*Vn, V);
  }
  ShAttrib1f under = Ph(1) < 0.;
  Ph = cond(under,
    Ph * ShAttrib3f(1.,0.,1.), Ph);
  ShVector3f Vn =
    V * ShAttrib3f(0.,1.,0.);
  ShVector3f Vt = V - Vn;
  V = cond(under,
    (1.0 - mu)*Vt - eps*Vn, V);
  Ph(1) = cond(min(under,(V|V)<0.1),
    ShPoint1f(0.), Ph(1));
  ShVector3f dt = Pt - Ph;
  Pt = cond((dt|dt) < 0.02, Pt +
    ShVector3f(0.0, 0.02, 0.0), Pt);
} SH_END;

// define state stream
ShStream state =
  (pos & pos_tail & vel);
// curry p with state and parameters
ShProgram update =
  p << state << gravity << delta;

...

// IN INNER LOOP
// execute state update (input to update is compiled in)
state = update;
```


Stream Processing: Particles



Brook

- Stream programming model
  - GPU = streaming coprocessor
- C with stream extensions
- Cross platform
  - ATI & NVIDIA
  - OpenGL & DirectX
  - Windows & Linux

Streams

- Collection of records requiring similar computation
  - particle positions, voxels, FEM cell, …

```
Ray r<200>;
float3 velocityfield<100,100,100>;
```

- Similar to arrays, but…
  - index operations disallowed: `position[i]`
  - read/write stream operators

```
streamRead (r, r_ptr);
streamWrite (velocityfield, v_ptr);
```


Kernels

- Functions applied to streams
  - similar to for_all construct
  - no dependencies between stream elements

```
kernel void foo (float a<>, float b<>,
                 out float result<>) {
  result = a + b;
}

float a<100>;
float b<100>;
float c<100>;

foo(a,b,c);
```

```
for (i=0; i<100; i++)
    c[i] = a[i]+b[i];
```


Kernels

- Kernel arguments
  - input/output streams

```
kernel void foo (float a<>,
                 float b<>,
                 out float result<>) {
   result = a + b;
}
```


Kernels

- Kernel arguments
  - input/output streams
  - gather streams

```
kernel void foo (..., float array[] ) {
      a = array[i];
}
```

Kernels

- Kernel arguments
  - input/output streams
  - gather streams
  - iterator streams

```
kernel void foo (..., iter float n<> ) {
      a = n + b;
}
```


Kernels

- Kernel arguments
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

```
kernel void foo (..., float c ) {
      a = c + b;
}
```


Kernels

- Ray triangle intersection

```
kernel void krnIntersectTriangle(Ray ray<>, Triangle tris[],
                                 RayState oldraystate<>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<>) {
  float idx, det, inv_det;
  float3 edge1, edge2, pvec, tvec, qvec;
  if(oldraystate.state.y > 0) {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0f/det;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot( tvec, pvec ) * inv_det;
    qvec = cross( tvec, edge1 );
    candidatehit.data.z = dot( ray.d, qvec ) * inv_det;
    candidatehit.data.x = dot( edge2, qvec ) * inv_det;
    candidatehit.data.w = idx;
  } else {
    candidatehit.data = float4(0,0,0,-1);
  }
}
```


Reductions

- Compute single value from a stream
  - associative operations only

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}

float a<100>;
float r;

sum(a,r);
```

```
r = a[0];
for (int i=1; i<100; i++)
  r += a[i];
```

Reductions

- Multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}

float a<20>;
float r<5>;

sum(a,r);
```

```
for (int i=0; i<5; i++)
  r[i] = a[i*4];
  for (int j=1; j<4; j++)
    r[i] += a[i*4 + j];
```


Stream Repeat & Stride

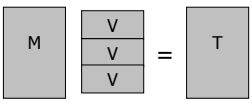
- Kernel arguments of different shape
  - resolved by repeat and stride

```
kernel void foo (float a<>, float b<>,
                 out float result<>);

float a<20>;
float b<5>;
float c<10>;

foo(a,b,c);
```

```
foo(a[0],  b[0], c[0])
foo(a[2],  b[0], c[1])
foo(a[4],  b[1], c[2])
foo(a[6],  b[1], c[3])
foo(a[8],  b[2], c[4])
foo(a[10], b[2], c[5])
foo(a[12], b[3], c[6])
foo(a[14], b[3], c[7])
foo(a[16], b[4], c[8])
foo(a[18], b[4], c[9])
```

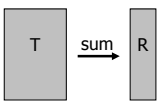

Matrix Vector Multiply

```
kernel void mul (float a<>, float b<>,
                 out float result<>) {
   result = a*b;
}

reduce void sum (float a<>,
                 reduce float result<>) {
   result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```

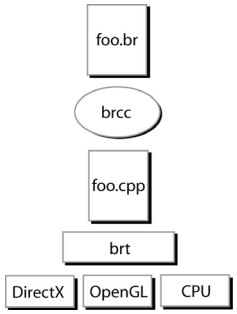


Matrix Vector Multiply

```
kernel void mul (float a<>, float b<>,
                 out float result<>) {
   result = a*b;
}

reduce void sum (float a<>,
                 reduce float result<>) {
   result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



System Outline



brcc

Source to source compiler

- Generate CG & HLSL code
- CGC and FXC for shader assembly
- Virtualization

brt

Brook run-time library

- Stream texture management
- Kernel shader execution

Running Brook

- Compiling .br files

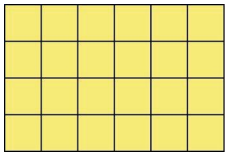
```
Brook CG Compiler
Version: 0.2  Built: Apr 24 2004, 18:11:59
brcc [-hvndktyAN] [-o prefix] [-w workspace] [-p shader ] foo.br
    -h              help (print this message)
    -v              verbose (print intermediate generated code)
    -n              no codegen (just parse and reemit the input)
    -d              debug (print cTool internal state)
    -k              keep generated fragment program (in foo.cg)
    -t              disable kernel call type checking
    -y              emit code for ATI 4-output hardware
    -A              enable address virtualization (experimental)
    -N              deny support for kernels calling other kernels
    -o prefix       prefix prepended to all output files
    -w workspace    workspace size (16 - 2048, default 1024)
    -p shader       cpu / ps20 / fp30 / cpumt (can specify multiple)
    -f compiler     favor a particular compiler (cgc / fxc / default)
```


Eliminating GPU Limitations

Treating texture as memory

- Limited texture size and dimension
- Compiler inserts address translation code

```
float matrix<8096,10,30,5>;
```



Eliminating GPU Limitations

Extending kernel outputs

- duplicate kernels, let **cgc** or **fxc** do dead code elimination
- better solution:

"Efficient Partitioning of Fragment Shaders for Multiple-Output Hardware"
Tim Foley, Mike Houston, and Pat Hanrahan

"Mio: Fast Multipass Partitioning via Priority-Based Instruction Scheduling"
Andrew T. Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone, and John D. Owens

Running Brook

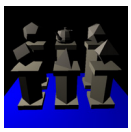
- BRT_RUNTIME selects platform
  - CPU Backend:
    BRT_RUNTIME = cpu
  - OpenGL ARB Backend:
    BRT_RUNTIME = arb
  - DirectX9 Backend:
    BRT_RUNTIME = dx9

Runtime

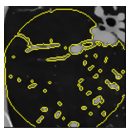
- Accessing stream data for graphics aps
  - Brook runtime api available in C++ code
  - autogenerated .hpp files for brook code

```
brook::initialize( "dx9", (void*)device );

// Create streams
fluidStream0 = stream::create<float4>( kFluidSize, kFluidSize );
normalStream = stream::create<float3>( kFluidSize, kFluidSize );

// Get a handle to the texture being used by
// the normal stream as a backing store
normalTexture = (IDirect3DTexture9*)
                normalStream->getIndexedFieldRenderData(0);

// Call the simulation kernel
simulationKernel( fluidStream0, fluidStream0, controlConstant,
                  fluidStream1 );
```


Applications

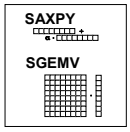


Ray-tracer



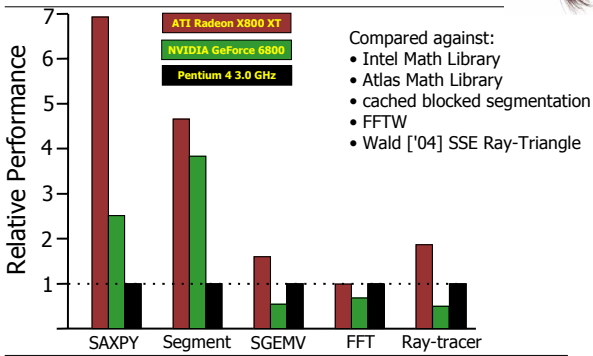
Segmentation

FFT edge detect



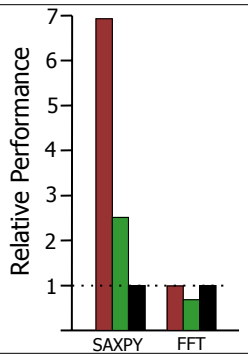
Linear algebra

Performance



Compared against:
- Intel Math Library
- Atlas Math Library
- cached blocked segmentation
- FFTW
- Wald ['04] SSE Ray-Triangle

Understanding Performance



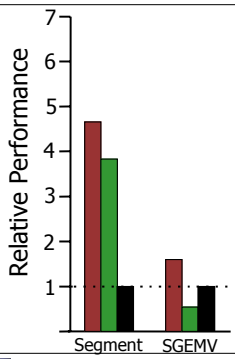
GPU wins when…

- limited data reuse
  - ✓ SAXPY
  - × FFT

Pentium 4 3.0 GHz
44 GB/sec peak cache bandwidth

NVIDIA GeForce 6800 Ultra
36 GB/sec peak memory bandwidth

Understanding Performance

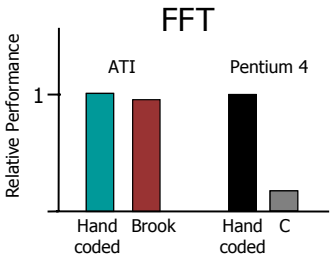


GPU wins when…

- Arithmetic intensity
  - ✓ Segment
    3.7 ops per word
  - × SGEMV
    1/3 ops per word

Efficiency

Brook version within 80% of hand-coded GPU version



Brook for GPUs

- Release v0.3 available on Sourceforge
- Project Page
  - http://graphics.stanford.edu/projects/brook
- Source
  - http://www.sourceforge.net/projects/brook
- Over 6K downloads!

Brook for GPUs: Stream Computing on Graphics Hardware
SIGGRAPH 2004
Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan

Fly-fishing fly images from The English Fly Fishing Shop

Mathematical Primitives

GPGPU

Robert Strzodka
caesar research center
Bonn, Germany

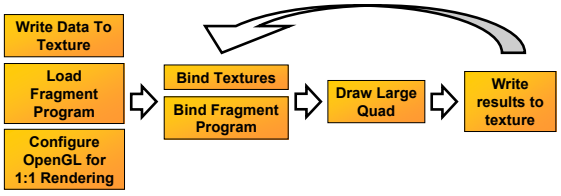
VIS 2004

Overview

- Mathematical GPU Functionality
- Partial Differential Equations (PDEs)
  - Examples
  - Time and Space Discretization
  - Local and Global Operations
  - Matrix Vector Product
  - Gather – Scatter
- Advanced Topics
  - Discretization Grids
  - Discretization Schemes
  - Quantization

GPU as a Data-Parallel Computer

- Data specification → Textures
- Kernel specification → Fragment program
- General execution → Draw single large quad



Fragment Processor Functionality as seen from a High Level Language

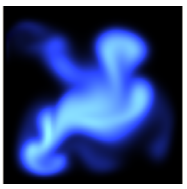
- Float data types:
  - 16-bit & 32-bit (NVIDIA), 24-bit (ATI)
- Vectors, structs and arrays:
  - float4, float vec[6] , float3x4, float arr[10][20], struct {}
- Arithmetic and logic operators:
  - +, -, *, /; &&, ||, !
- Trignonometric, exponential functions:
  - sin, asin, exp, log, pow, …
- User defined functions
  - max3(float a, float b, float c) { return max(a,max(b,c)); }
- Conditional statements by predication, unrollable loops:
  - if, for, while, dynamic branching in PS3
- Arbitrary texture positions can be accessed

Overview

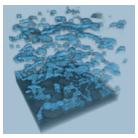
- Mathematical GPU Functionality
- Partial Differential Equations (PDEs)
  - Examples
  - Time and Space Discretization
  - Local and Global Operations
  - Matrix Vector Product
  - Gather – Scatter
- Advanced Topics
  - Discretization Grids
  - Discretization Schemes
  - Quantization

PDE Examples

fluid dynamics

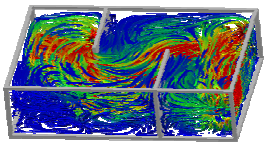


cloud dynamics

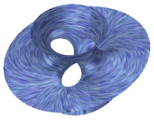


boiling

PDE Examples



visualization



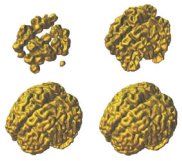
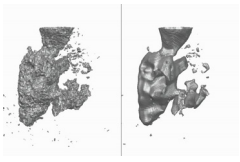


image processing



A Common PDE Model

We seek a function $u(x,t):(\Omega,\Re^+)\rightarrow\Re^m,\Omega\subseteq\Re^d$ which satisfies

PDE	$\partial_t u+F[u,u]=0$	in $\Re^+\times\Omega$
initial value	$u(0)=u_0$	in Ω
boundary	$\partial_v u=b_N$ or $u=b_D$	on $\Re^+\times\partial\Omega$

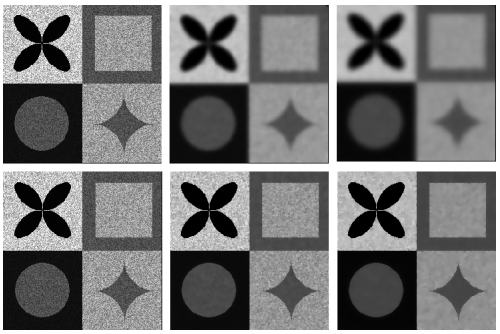
We distiguish between linear ***u*** and non-linear ***v*** dependencies in ***F[u,v]***

linear diffusion equation: $F[u,v]=-\Delta v:=-\sum_{k=1}^{d}\frac{\partial^2 v}{\partial x_k^2}$

non-linear diffusion equation: $F[u,v]=-\mathrm{div}\left(g(\|\nabla_\sigma u\|)\nabla v\right)$

Theory and numerical analysis of PDEs are extensive subjects. There are many other PDE types, discretization methods and in general approaches than presented in the following slides.

Denoising by a linear and a non-linear diffusion process



linear diffusion

non-linear diffusion

Diffusion Example - PDE

Initial image $u_0 : \Omega \to [0,1]$, unknown $u : (\Omega, \Re^+) \to \Re$

$$\partial_t u - \mathrm{div}(G(\nabla u_\sigma)\nabla u) = 0 \quad \text{in } \Re^+ \times \Omega$$

$$u(0) = u_0 \quad \text{in } \Omega$$

$$\partial_\nu u = 0 \quad \text{on } \Re^+ \times \partial\Omega$$

- linear

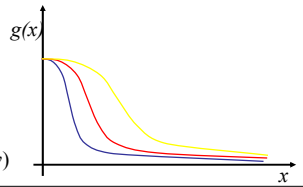
$G(v) := 1$

- isotropic non-linear

$G(v) := g(\| v \|)$ scalar

- anisotropic

$G(v) := B^T(v) \begin{pmatrix} g_1(\|v\|) & 0 \\ 0 & g_2(\|v\|) \end{pmatrix} B(v)$



Time Discretization

$$\partial_t u + F[u,u] = 0 \;\rightarrow\; \frac{u^{n+1} - u^n}{\tau^n} + F[u^?, u^?] = 0$$

Explicit $\quad u^{n+1} = u^n - \tau^n F[u^n, u^n]$

New solution can be computed directly from the previous one.
Strong restrictions on the time step width.

Implicit $\quad u^{n+1} + \tau^n F[u^{n+1}, u^{n+1}] = u^n$

A non-linear solver must be used to obtain the solution.
The time step width is not restricted.

Semi-implicit $\quad u^{n+1} + \tau^n F[u^n, u^{n+1}] = u^n$

A linear equation solver suffices to obtain the new solution.
Time step restrictions depend on the problem, often none.

In any case we obtain a discrete sequence of results $\{u^n\}_{n \in \mathrm{N}}$.

Space Discretization

□ Transition □

Continuous $\qquad$ Discrete

Ω $\quad u^n,\ \nabla u^n,\ F[u^n, v^n]$ $\qquad$ Ω_h $\quad U^n,\ \nabla_h U^n,\ F_h[U^n, V^n]$

The discrete function U^n can be represented by a vector $(\overline{U}^n_\alpha)_{\alpha \in \Omega_h}$ which contains the values at the grid nodes Ω_h.

Linear operations translate into matrix-vector products

$$\Delta u^n \;\rightarrow\; A \cdot \overline{U}^n$$

Non-linear operations become multi-dimensional functions on the vector components $\quad F[u^n] \;\rightarrow\; F_h\left((\overline{U}^n_\alpha)_\alpha\right)$

Diffusion Example - Discretization

continuous model $\partial_t u - \mathrm{div}(G(\nabla u_\sigma)\nabla u) = 0$

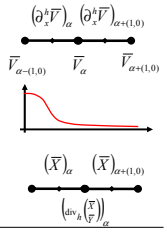
time disc. (semi-implicit) $u^{n+1} - \tau^n \mathrm{div}(G(\nabla u_\sigma^n)\nabla u^{n+1}) = u^n$

space disc. (Finite Differences) $\overline{U}^{n+1} - \tau^n \mathrm{div}_h(G(\nabla_h \overline{U}_\sigma^n)\nabla_h \overline{U}^{n+1}) = \overline{U}^n$

$$(\nabla_h \overline{V})_\alpha := \frac{1}{h}\begin{pmatrix} \overline{V}_\alpha - \overline{V}_{\alpha-(1,0)} \\ \overline{V}_\alpha - \overline{V}_{\alpha-(0,1)} \end{pmatrix}$$

$$\begin{pmatrix} \overline{X} \\ \overline{Y} \end{pmatrix} := G(\nabla_h \overline{U}_\sigma^n)(\nabla_h \overline{V})$$

$$\left(\mathrm{div}_h \begin{pmatrix} \overline{X} \\ \overline{Y} \end{pmatrix}\right)_\alpha := \frac{1}{h}\left(\overline{X}_{\alpha+(1,0)} - \overline{X}_\alpha + \overline{Y}_{\alpha+(0,1)} - \overline{Y}_\alpha\right)$$



Diffusion Example – Linear Algebra

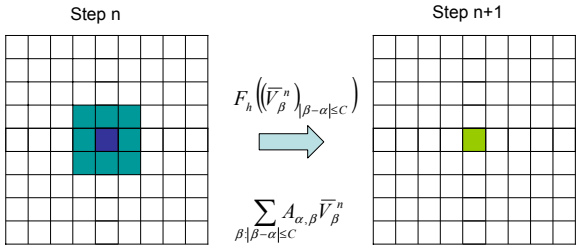
$$\overline{U}^{n+1} - \tau^n \mathrm{div}_h(G(\nabla_h \overline{U}_\sigma^n)\nabla_h \overline{U}^{n+1}) = \overline{U}^n$$

$$\overline{U}^{n+1} - \tau^n L[\nabla_h \overline{U}^n]\cdot \overline{U}^{n+1} = \overline{U}^n$$

$$A[\nabla_h \overline{U}^n]\cdot \overline{U}^{n+1} = \overline{U}^n, \quad A[\nabla_h \overline{U}^n] := \mathbf{1} - \tau^n L[\nabla_h \overline{U}^n]$$

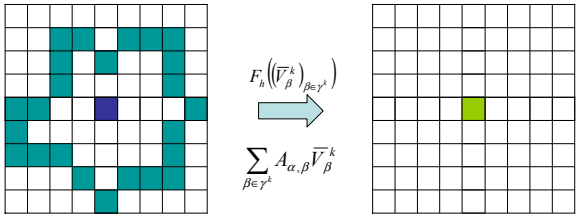
- Typical situation in semi-implicit schemes
  - Matrix *A* depends non-linearly on explicit data
  - Linear equation system must be solved
- Solvers on GPUs have similar requirements as on parallel computers
  - Parallel processing of matrix entries
  - No direct write-read-modify cycles
  - Examples: Jacobi solver, conjugate gradient, block-SOR

Local Gather Operation



$$\left(\overline{V}_\beta^n\right)_{\beta-\alpha\le C} \qquad \overline{V}_\alpha^{n+1}$$

Global Gather Operation



$$\left(\overline{V}_\beta^k\right)_{\beta\in\gamma^k} \qquad \overline{V}_\alpha^k$$

Matrix Vector Product on GPUs

- Pure matrix vector product is bandwidth bound on GPU
  - Number of processing elements against bandwidth increases
  - Try to occupy the many processing elements with computations
- Three possibilities for a matrix vector product $A \cdot V$ if A depends on some data and must be computed itself
  - On-the-fly: compute entries of A for each $A \cdot V$ appliction
    - Lowest memory requirement
    - Good for simple entries or seldom use of A
  - Partial assembly: apply A on-the-fly with some precomputed results
    - Allows to balance computation and bandwidth requirements
    - Good choice of precomputed results requires also little memory
  - Full assembly: precompute all entries of A, use these in $A \cdot V$
    - Good if other computations hide bandwidth problem in $A \cdot V$
    - Otherwise try to use partial assembly

Diffusion Example – Matrix Vector Product

$$A[\nabla_h \overline{U}^n] \cdot \overline{V} := \left(\mathbf{1} - \tau^n L[\nabla_h \overline{U}^n]\right) \cdot \overline{V} = \overline{V} - \tau^n \operatorname{div}_h \left(G(\nabla_h \overline{U}^n_\sigma) \nabla_h \overline{V}\right)$$

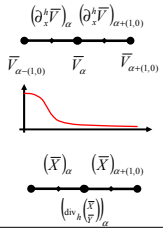
Partial Assembly: Precompute $G(\nabla_h \overline{U}^n_\sigma)$

Perform $A[\nabla_h \overline{U}^n] \cdot \overline{V}$ on-the-fly

$$\left(\nabla_h \overline{V}\right)_\alpha := \frac{1}{h} \begin{pmatrix} \overline{V}_\alpha - \overline{V}_{\alpha-(1,0)} \\ \overline{V}_\alpha - \overline{V}_{\alpha-(0,1)} \end{pmatrix}$$

$$\begin{pmatrix} \overline{X} \\ \overline{Y} \end{pmatrix} := G(\nabla_h \overline{U}^n_\sigma)\left(\nabla_h \overline{V}\right)$$

$$\left(\operatorname{div}_h \begin{pmatrix} \overline{X} \\ \overline{Y} \end{pmatrix}\right)_\alpha := \frac{1}{h}\left(\overline{X}_{\alpha+(1,0)} - \overline{X}_\alpha + \overline{Y}_{\alpha+(0,1)} - \overline{Y}_\alpha\right)$$



Matrix Assembly

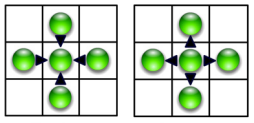
- Avoid assembling dense matrices
  - A dense matrix for a 1000x1000 grid has 1T=1000G entries
  - In practice most matrices are sparse
- Local gather operations produce band matrices
  - Each band (corresponding to a stencil position) requires one vector for storage, e.g. 9 vectors for a general 3x3 stencil
  - Stencil values are often symmetric, separable or have other intrinsic structure, which reduces the storage requirements
- Exploit patterns in global gather operations
  - For simple entries store only their geometry
  - Consider the gather - scatter interchangability

Gather - Scatter

Interaction types between node values (vector components)

$$A \cdot \overline{V} = (A_{\alpha,\cdot} \cdot \overline{V})_\alpha$$

$$A_{\alpha,\cdot} := (A_{\alpha,\beta})_\beta$$



$$A \cdot \overline{V} = \sum_\gamma T_\gamma \left(\widetilde{A}^\gamma \bullet \overline{V}\right)$$

$$\widetilde{A}^\gamma := (A_{\alpha-\gamma,\alpha})_\alpha$$

$$\left(\overline{X} \bullet \overline{Y}\right)_\alpha := \overline{X}_\alpha \bullet \overline{Y}_\alpha$$

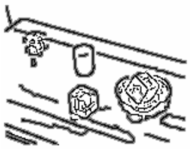
$$\left(T_\gamma(\overline{X})\right)_\alpha := \overline{X}_{\alpha+\gamma}$$

Both types are interchangeable in matrix vector products

- Easy conversion if gather and scatter positions are static
- Dynamic gather is ok for GPU, dynamic scatter rather slow

Object recoginition by the Generalized Hough Transform

original image



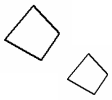
edge image

first detected cube

second detected cube

Generalized Hough Transform

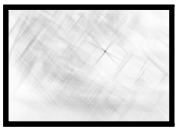
Generate poses of different perspective and scale



Store pose k as a list of offset vectors $l^{k,i}$

Draw image with offsets

normalize the result



$$\overline{C}^k(x,y) = \sum_{i=1}^{|l^k|} I^{\text{edge}}(x + l_x^{k,i}, y + l_y^{k,i})$$

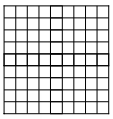
$$C^k(x,y) = \overline{C}^k(x,y) / |l^k|$$

Overview

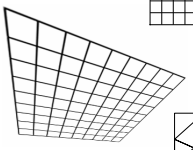
- Mathematical GPU Functionality
- Partial Differential Equations (PDEs)
  - Examples
  - Time and Space Discretization
  - Local and Global Operations
  - Matrix Vector Product
  - Gather – Scatter
- Advanced Topics
  - Discretization Grids
  - Discretization Schemes
  - Quantization

Discretization Grids on GPUs

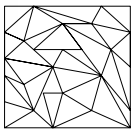
- An equidistant grid
  - Easy to implement
  - One texture holds all values



- Deformed tensor grid
  - Parallel dynamic updates
  - One texture for values second for deformation

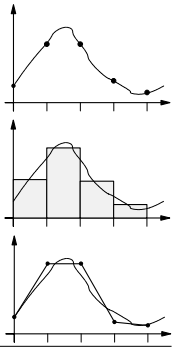


- Unstructured grid
  - Good performance for static, poor for dynamic grid topology
  - Several indirections are needed



Discretization Schemes

- Finite Differences
  - Interpolative approach: simple and fast
  - Usually interaction with direct neighbors
- Finite Volumes
  - Volumetric approach: mass conservation
  - Good at discontinuities, less for smooth data
  - Interaction over element boundaries
- Finite Elements
  - Approximative approach: error minimization
  - Good handling of deformed, unstructured grids
  - Interaction of basis functions (all neighbors)



Quantization

Roundoff examples for the **float s23e8** format

additive roundoff	a= 1 + 0.00000004	$=_{fl}$ 1
multiplicative roundoff	b= 1.0002 * 0.9998	$=_{fl}$ 1
cancellation c=a,b	$(c-1) * 10^8$	$=_{fl}$ 0

Cancellation promotes the small error 0.00000004
to the absolute error 4 and an infinite relative error.

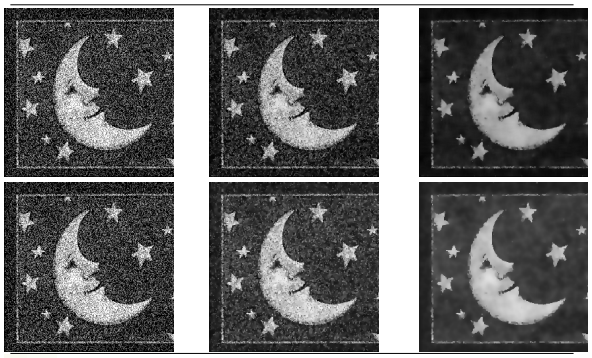
Order of operations can be crucial:

$1 + 0.00000004 - 1 =_{fl} 0$

$1 - 1 + 0.00000004 =_{fl} 0.00000004$

Cancellation cannot be avoided automatically, so watch out!

Uncontrolled and Controlled Roundoff Effects on Diffusion in 8 bit



General Algorithm Primitives

GPGPU

John Owens
Department of Electrical and Computer Engineering
Institute for Data Analysis and Visualization
University of California, Davis

VIS 2004

Topics

- Two fundamental algorithms!
- Sorting
  - Sorting networks
- Search
  - Binary search
  - Nearest neighbor search
- Raytracing on GPUs: [Purcell 2002]
- Photon mapping on GPUs: [Purcell 2003]

Assumptions

- Data organized into 1D arrays
- Rendering pass == screen aligned quad
  - Not using vertex shaders
- PS 2.0 GPU
  - No data dependent branching at fragment level

Sorting

- Given an unordered list of elements, produce list ordered by key value
  - Fundamental kernel: *compare and swap*
- GPUs constrained programming environment limits viable algorithms
  - *Oblivious* sort (does not rely on data values)
  - Bitonic merge sort [Batcher 68]
  - Periodic balanced sorting networks [Dowd 89]

Bitonic Merge Sort Overview

- Repeatedly build bitonic lists and then sort them
  - Bitonic list is two monotonic lists concatenated together, one increasing and one decreasing.
    - List A: (3, 4, 7, 8) monotonically increasing
    - List B: (6, 5, 2, 1) monotonically decreasing
    - List AB: (3, 4, 7, 8, 6, 5, 2, 1) bitonic
  - Bitonic lists can be easily sorted into monotonic lists
  - Strategy: Divide and conquer

Bitonic Merge Sort

3
7
4
8
6
2
1
5

8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists: (3,7) (4,8) (6,2) (1,5)

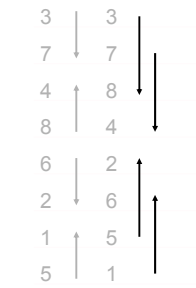
Bitonic Merge Sort (1/3)

Sort the bitonic lists (Step 1 of 1)

Bitonic Merge Sort (1/3 done)

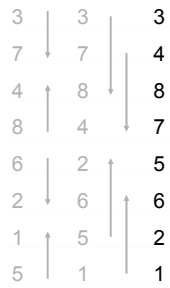
4x monotonic lists: (3,7) (8,4) (2,6) (5,1)
2x bitonic lists: (3,7,8,4) (2,6,5,1)

Bitonic Merge Sort (2/3)



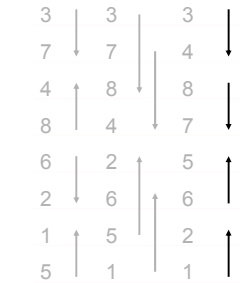
Sort the bitonic lists (Step 1 of 2)

Bitonic Merge Sort (2/3)



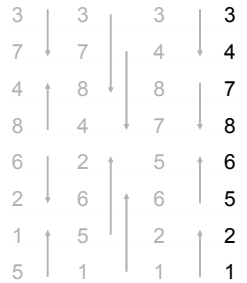
Sort the bitonic lists (Step 2 of 2)

Bitonic Merge Sort (2/3)



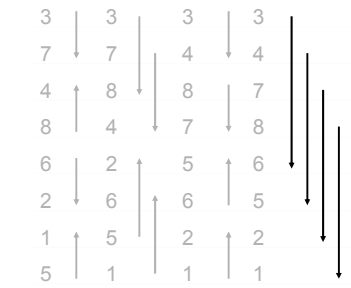
Sort the bitonic lists (Step 2 of 2)

Bitonic Merge Sort (2/3 done)



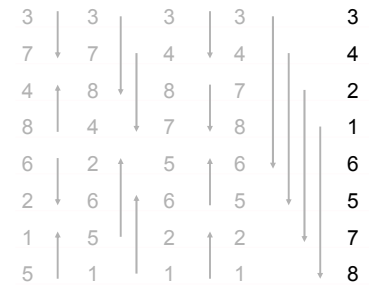
2x monotonic lists:
(3,4,7,8) (6,5,2,1)
1x bitonic list:
(3,4,7,8, 6,5,2,1)

Bitonic Merge Sort (3/3)



Sort the bitonic list (Step 1 of 3)

Bitonic Merge Sort (3/3)

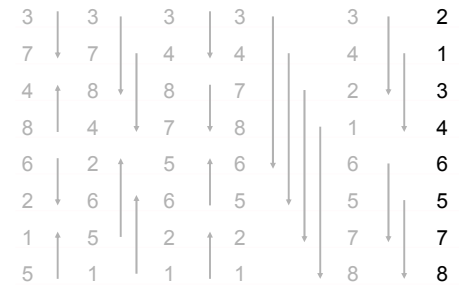


Sort the bitonic list (Step 2 of 3)

Bitonic Merge Sort (3/3)

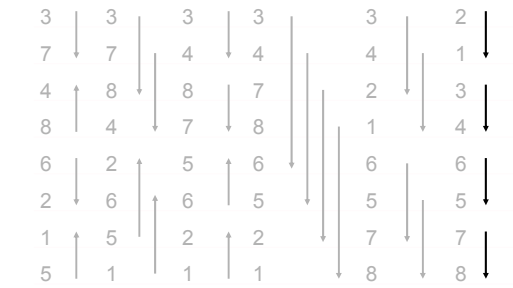
Sort the bitonic list (Step 2 of 3)

Bitonic Merge Sort (3/3)



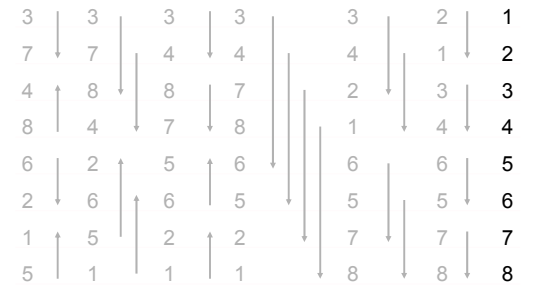
Sort the bitonic list (Step 3 of 3)

Bitonic Merge Sort (3/3)



Sort the bitonic list (Step 3 of 3)

Bitonic Merge Sort (Complete)



Done!

Bitonic Merge Sort Summary

- Separate rendering pass for each set of swaps
  - $O(\log^2 n)$ passes
  - Each pass performs n compare/swaps
    - Exploits parallelism
    - Each swap is oblivious
  - Total compare/swaps: $O(n \log^2 n)$
    - Limitations of GPU cost us factor of log n over best CPU-based sorting algorithms

Searching

Types of Search

- Search for specific element
  - Binary search
- Search for nearest element(s)
  - k-nearest neighbor search
- Both searches require ordered data

Binary Search

- Find a specific element in an ordered list
- Implement just like CPU algorithm
  - Assuming hardware supports long enough shaders
  - Finds the first element of a given value ***v***
    - If ***v*** does not exist, find next smallest element > ***v***
- Search algorithm is sequential, but many searches can be executed in parallel
  - Number of pixels drawn determines number of searches executed in parallel
    - 1 pixel == 1 search

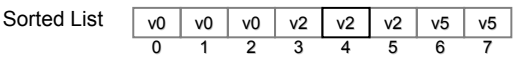
Binary Search

- Search for v0

Initialize 4

Search starts at center of sorted array

v2 >= v0 so search left half of sub-array



Binary Search

- Search for v0

Initialize 4

Step 1 2

v0 >= v0 so search left half of sub-array

Binary Search

- Search for v0

Initialize	4
Step 1	2
Step 2	1

v0 >= v0 so search left half of sub-array

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0

Initialize	4
Step 1	2
Step 2	1
Step 3	0

At this point, we either have found v0 or are 1 element too far left

One last step to resolve

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0

Initialize	4
Step 1	2
Step 2	1
Step 3	0
Step 4	0

Done!

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0 and v2

Initialize	4	4

Search starts at center of sorted array

Both searches proceed to the left half of the array

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0 and v2

Initialize	4	4
Step 1	2	2

The search for v0 continues as before

The search for v2 overshot, so go back to the right

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0 and v2

Initialize	4	4
Step 1	2	2
Step 2	1	3

We've found the proper v2, but are still looking for v0

Both searches continue

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0 and v2

Initialize	4	4
Step 1	2	2
Step 2	1	3
Step 3	0	2

Now, we've found the proper v0, but overshot v2

The cleanup step takes care of this

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search

- Search for v0 and v2

Initialize	4	4
Step 1	2	2
Step 2	1	3
Step 3	0	2
Step 4	0	3

Done! Both v0 and v2 are located properly

Sorted List

v0	v0	v0	v2	v2	v2	v5	v5
0	1	2	3	4	5	6	7

Binary Search Summary

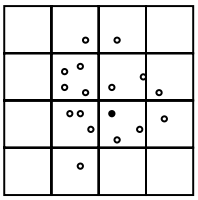
- Single rendering pass
  - Each pixel drawn performs independent search
- O(log n) steps

Nearest Neighbor Search

Nearest Neighbor Search

- Given a sample point *p*, find the *k* points nearest *p* within a data set
- On the CPU, this is easily done with a heap or priority queue
  - Can add or reject neighbors as search progresses
  - Don't know how to build one efficiently on GPU
- kNN-grid
  - Can only add neighbors…

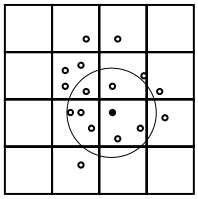
kNN-grid Algorithm



- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

kNN-grid Algorithm

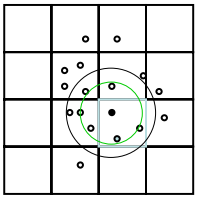


- Candidate neighbors must be within max search radius
- Visit voxels in order of distance to sample point

• sample point
◦ candidate neighbor
• neighbors found

Want 4 neighbors

kNN-grid Algorithm

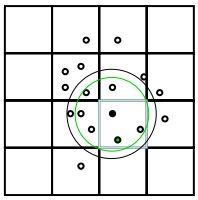


- If current number of neighbors found is less than the number requested, grow search radius

• sample point
◦ candidate neighbor
• neighbors found

Want 4 neighbors

kNN-grid Algorithm

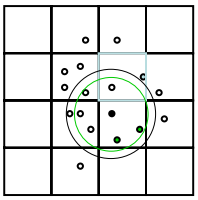


- If current number of neighbors found is less than the number requested, grow search radius

• sample point
◦ candidate neighbor
• neighbors found

Want 4 neighbors

kNN-grid Algorithm

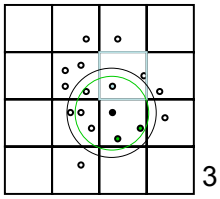


- Don't add neighbors outside maximum search radius
- Don't grow search radius when neighbor is outside maximum radius

• sample point
◦ candidate neighbor
• neighbors found

Want 4 neighbors

kNN-grid Algorithm

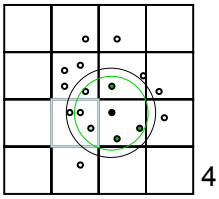


- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

kNN-grid Algorithm

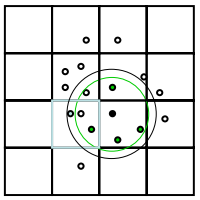


- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

kNN-grid Algorithm

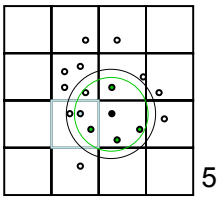


- Don't expand search radius if enough neighbors already found

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

kNN-grid Algorithm

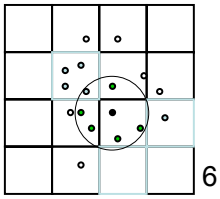


- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

kNN-grid Algorithm

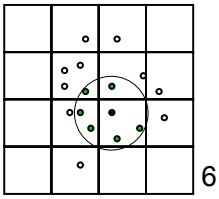


- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

- Visit all other voxels accessible within determined search radius
- Add neighbors within search radius

kNN-grid Summary



- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors

- Finds all neighbors within a sphere centered about sample point
- May locate more than requested *k*-nearest neighbors

References

- Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. ACM Transactions on Graphics, 21(3):703–712, July 2002.
- Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In Graphics Hardware 2003, pages 41–50, July 2003.

GPGPU Memory Model

Aaron Lefohn
Institute for Data Analysis and Visualization
University of California, Davis

Overview

- GPU Memory Model
- GPU-Based Data Structures
- Pbuffer Survival Guide

CPU Memory Model

- Random Memory Access at Any Program Point
  - Read/write to registers
  - Read/write to local (stack) memory
  - Read/write to global (heap) memory
  - Read/write to disk

GPU Memory Model

- Much more restricted memory access
  - GPU Kernels
    - Read/write to registers
    - No local stack memory
    - No disk access
    - Read-only global memory access

```
v = a[i];
```

    - Write to global memory at end of pass
      - Pre-computed memory addresses (no scatter)

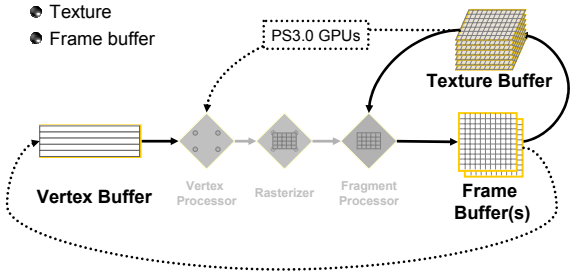
```
i = foo(a);
a[i] = bar(b);
```

      - Write location set by fragment position

```
a[fragPos] = bar(b);
```

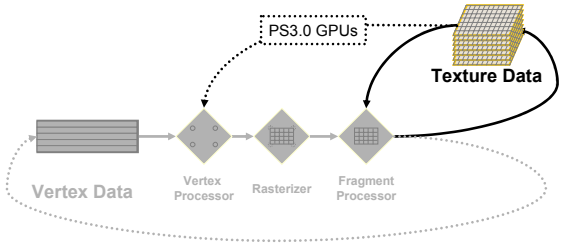
GPU Memory Model

- Where is GPU Data Stored?
  - Vertex buffer
  - Texture
  - Frame buffer



Render-to-Texture

- Idea
  - Write rendering result to texture memory
  - Enables GPU-based computational iterations



Render-to-Texture

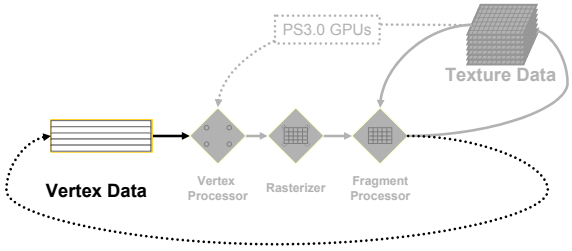
- OpenGL Support
  - Save up to 16, 32-bit floating values per pixel
    Multiple Render Targets (MRTs) on ATI and NVIDIA

1. Copy-to-texture
   glCopyTexSubImage

2. Render-to-texture
   - WGL_ARB_render_texture
     Pbuffers: Current state of the art
   - GL_EXT_render_target
     Proposed extension
   - Superbuffers
     Proposed extension

Render-to-Vertex-Array

- Idea
  - Write rendering results to vertex array
  - Allows GPU to loop back to beginning of pipeline

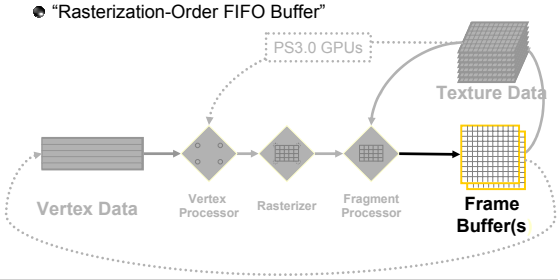


Render-To-Vertex-Array

- OpenGL Support
  - Copy-to-vertex-array
    - GL_EXT_pixel_buffer_object
    - NVIDIA and ATI
  - Render-to-vertex-array
    - Superbuffers
- Semantics Still Under Development…

Fbuffer: Capturing Fragments

- Idea
  - Save all fragment values instead of one per pixel
  - "Rasterization-Order FIFO Buffer"



Fbuffer: Capturing Fragments

- Details
  - Designed for multi-pass rendering with transparent geometry
  - Mark and Proudfoot, Graphics Hardware 2001
    http://graphics.stanford.edu/projects/shading/pubs/hwws2001-fbuffer/
  - New possibilities for GPGPU
    - Varying number of results per pixel
    - RTT and RTVA with an fbuffer
- OpenGL Support
  - ATI Radeon 9800 and newer ATI GPUs
  - Not yet exposed to user (ask for it!)

Overview

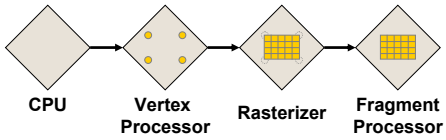
- GPU Memory Model
- GPU-Based Data Structures
- Pbuffer Survival Guide

GPU-Based Data Structures

- Building Blocks
  - GPU memory addresses
    - Address Generation
    - Address Use
    - Pointers
  - Multi-dimensional arrays
  - Sparse representations

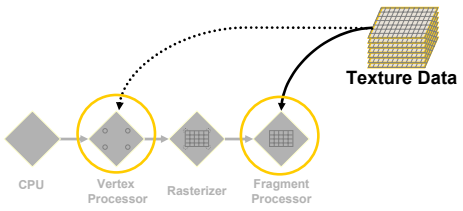
GPU Memory Addresses

- Where Are GPU Addresses Generated?
  - CPU — Vertex stream or textures
  - Vertex processor — Input stream, ALU ops or textures
  - Rasterizer — Interpolation
  - Fragment processor — Input stream, ALU ops or textures



GPU Memory Addresses

- Where Are Addresses Used?
  - Vertex textures (PS3.0 GPUs)
  - Fragment textures



GPU Memory Addresses

- Floating-Point Addressing
  - Normalized addresses [0,1]
    - GL_TEXTURE_1D, _2D, _3D, _CUBE
  - Non-Normalized addresses [0,N]
    - GL_TEXTURE_RECTANGLE
  - Warning: Floating-point can leave unaddressable texels

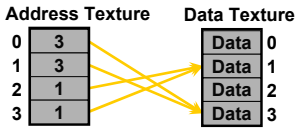
NVIDIA FP32:	16,777,217	Counting numbers
ATI 24-bit float:	131,073	Counting numbers
NVIDIA FP16:	2,049	Counting numbers

Courtesy of Ian Buck

GPU Memory Addresses

- Pointers
  - Store addresses in texture
  - Dependent texture read

```
float2 addr = tex2D( addrTex, texCoord );
float2 data = tex2D( dataTex, addr );
```

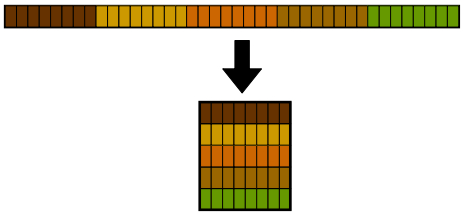


GPU-Based Data Structures

- Building Blocks
  - GPU memory addresses
    - Address Generation
    - Address Use
    - Pointers
  - Multi-dimensional arrays and structs
  - Sparse representations

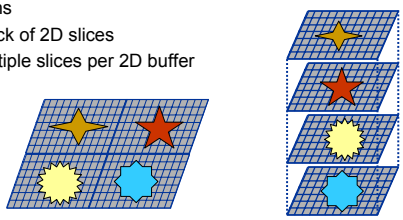
GPU Arrays

- Large 1D Arrays
  - Current GPUs limit 1D array sizes to 2048 or 4096
  - Pack into 2D memory
  - 1D-to-2D address translation



GPU Arrays

- 3D Arrays
  - Problem
    - GPUs do not have 3D frame buffers
    - No RTT to slice of 3D texture with pbuffers
  - Solutions
    1. Stack of 2D slices
    2. Multiple slices per 2D buffer



GPU Arrays

- Problems With 3D Arrays for GPGPU
  - Cannot read stack of 2D slices as 3D texture
  - Must know which slices are needed in advance
  - Visualization of 3D data difficult
- Solutions
  - Flat 3D textures
  - Need render-to-slice-of-3D-texture
    - GL_EXT_render_target and Superbuffers
  - Volume rendering of slice-based 3D data
    - Course 28, “Real-Time Volume Graphics”, Siggraph 2004

GPU Arrays

- Higher Dimensional Arrays
  - Pack into 2D buffers
  - N-D to 2D address translation
  - Same problems as 3D arrays if data does not fit in a single 2D texture

GPU Structures

- Store each member in a different “array” (texture)
  - Update structs with Multiple-Render Targets (MRTs)

```
struct Foo {
   float4 a;
   float4 b;
};
Foo foo[N];
```

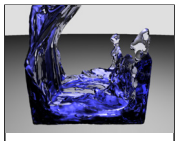
```
float4 Foo_a[N];
float4 Foo_b[N];
```


GPU-Based Data Structures

- Building Blocks
  - GPU memory addresses
    - Address Generation
    - Address Use
    - Pointers
  - Multi-dimensional arrays
  - Sparse representations

Sparse Data Structures

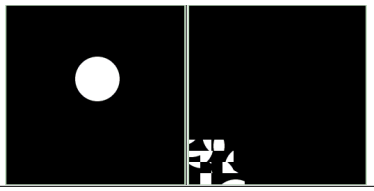
- Why Sparse Data Structures?
  - Reduce memory pressure
  - Reduce computational workload
- Examples
  - Sparse matrices
    - Krueger et al., Siggraph 2003
    - Bolz et al., Siggraph 2003
  - Deformable implicit surfaces (sparse volumes/PDEs)
    - Lefohn et al., IEEE Visualization 2003



Premoze et al. Eurographics 2003

Sparse Data Structures

- Basic Idea
  - Pack "active" data elements into GPU memory
  - For more information
    - Linear algebra section in this course : Static structures
    - Level-set case study in this course : Dynamic structures



GPU Data Structures

- Conclusions
  - Fundamental GPU memory primitive is a fixed-size 2D array
  - GPGPU needs more general memory model
  - Building and modifying complex GPU-based data structures is an open research topic…

Overview

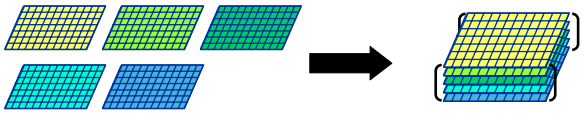
- GPU Memory Model
- GPU-Based Data Structures
- Pbuffer Survival Guide

Pbuffer Survival Guide

- Pbuffers Give us Render-To-Texture
  - Designed to create an environment map or two
  - Never intended to be used for GPGPU (100s of pbuffers)
  - Problem
    - Each pbuffer has its own OpenGL render context
    - Each pbuffer may have depth and/or stencil buffer
    - Changing OpenGL contexts is slow
  - Solution
    - Many optimizations to avoid this bottleneck…

Pbuffer Survival Guide

1. Use Multi-Surface Pbuffers
   - Each RGBA surface is its own render-texture
     - Front, Back, AuxN (N = 0,1,2,…)
   - Greatly reduces context switches
   - Technically illegal, but “blessed” by ATI and NVIDIA



5 Pbuffers
1 RGBA Surface Each

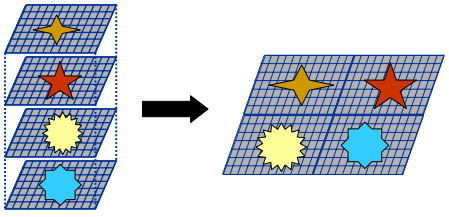
1 Pbuffer
5 RGBA Surfaces

Pbuffer Survival Guide

1. Using Multi-Surface Pbuffers
   a) Allocate multi-surface pbuffer (Front/Back/AUX buffers)
   b) Set render target to back buffer
      ```
      glDrawBuffer(GL_BACK)
      ```
   c) Bind front buffer as texture
      ```
      wglBindTexImageARB(pbuffer, WGL_FRONT_ARB)
      ```
   d) Render
   e) Switch buffers
      ```
      wglReleaseTexImageARB(pbuffer, WGL_FRONT_ARB)
      glDrawBuffer(GL_FRONT)
      wglBindTexImageARB(pbuffer, WGL_BACK_ARB)
      ```


Pbuffer Survival Guide

2. Pack 2D domains into large buffer
   - “Flat 3D textures”
   - Be careful of read-modify-write hazard



3D Volume

Flattened Volume

Conclusions

- GPU Memory Model Evolving
  - Writable GPU memory forms loop-back in an otherwise feed-forward streaming pipeline
  - Memory model will continue to evolve as GPUs become more general stream processors
- GPGPU Data Structures
  - Basic memory primitive is limited-size, 2D texture
  - Use address translation to fit all array dimensions into 2D
- Render-To-Texture
  - Use pbuffers with care and eagerly adopt their successor

Selected References

- J. Boltz, I. Farmer, E. Grinspun, P. Schoder, "Spare Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," SIGGRAPH 2003
- N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware," Graphics Hardware 2003
- M. Harris, W. Baxter, T. Scheuermann, A. Lastra, "Simulation of Cloud Dynamics on Graphics Hardware," Graphics Hardware *2003*
- H. Igehy, M. Eldridge, K. Proudfoot, "Prefetching in a Texture Cache Architecture," Graphics Hardware *1998*
- J. Krueger, R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," SIGGRAPH 2003
- A. Lefohn, J. Kniss, C. Hansen, R. Whitaker, "A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets," IEEE Transactions on Visualization and Computer Graphics 2004

Selected References

- A. Lefohn, J. Kniss, C. Hansen, R. Whitaker, "Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware," IEEE Visualization 2003
- W. Mark, K. Proudfoot, "The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering," Graphics Hardware 2001
- T. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware," Graphics Hardware 2003
- A. Sherbondy, M. Houston, S. Napel, "Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware," IEEE Visualization 2003

OpenGL References

- GL_EXT_pixel_buffer_object
  http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_pixel_buffer_object.txt
- GL_EXT_render_target,
  http://www.opengl.org/resources/features/GL_EXT_render_target.txt
- OpenGL Extension Registry
  http://oss.sgi.com/projects/ogl-sample/registry/
- Superbuffers
  http://www.ati.com/developer/gdc/SuperBuffers.pdf
- WGL_ARB_render_texture
  http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_render_texture.txt
  http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt

Acknowledgements

- Nick Triantos, Craig Kolb, Cass Everitt, Chris Seitz at NVIDIA
- Mark Segal, Rob Mace, Arcot Preetham, Evan Hart at ATI
- Brian Budge, Ph.D. student at UCDavis and NVIDIA intern
- The other GPGPU IEEE Visualization 2004 course presenters

- John Owens, Ph.D. advisor, Univ. of California Davis
- Ross Whitaker, M.S. advisor, SCI Institute, Univ. of Utah

- National Science Foundation Graduate Fellowship
- Pixar Animation Studios, summer internships
- ALIENWARE Interchangeable mobile GPUs

Computing Strategies and Tricks

Ian Buck
Graphics Lab
Stanford University

VIS 2004

Strategies & Tricks:

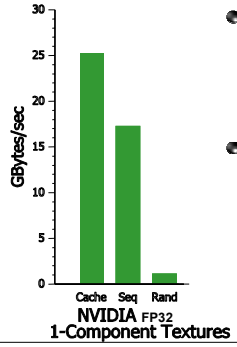
DirectX or OpenGL?

- DirectX
  - + Render to Texture
    - SetRenderTarget()
  - + Write once run anywhere
  - + Debugging tools
  - – Short programs
    - Only 512 instr limit
  - – Readback is slow!
    - ~50 MB/sec
- OpenGL
  - + 0 to N texture addressing
    - GL_TEXTURE_RECTANGLE_EXT
  - + Vendor Features
  - + Readback is fast
  - – Render-to-Texture not finalized
    - SuperBuffers
    - GL_EXT_render_target
  - – Specialized float formats for ATI and NV

Strategies & Tricks:

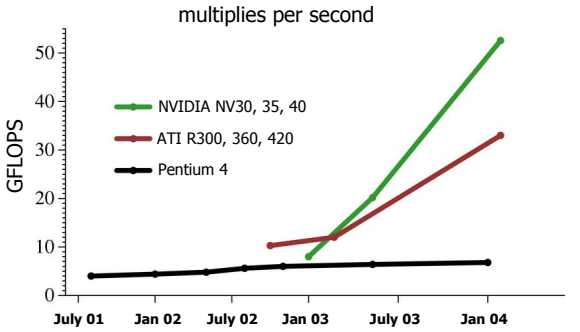
Understanding Performance

Locality, Locality, Locality

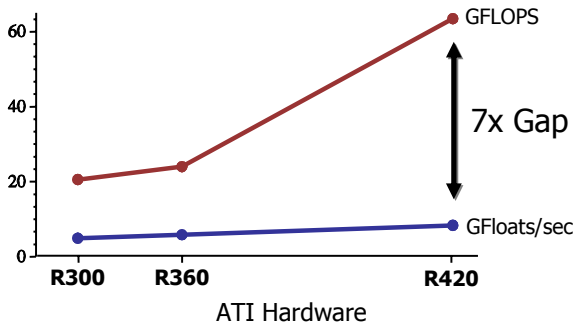


- CPU
  - Pentium 4 3.0 GHz
  - 44 GB/sec peak Cache
  - 6 GB/sec peak Seq
- Output Reuse
  - CPU can cache outputs
  - GPU must write outputs to memory

Compute Performace



Bandwidth Gap



Compute vs. Bandwidth

Arithmetic Intensity =
Compute-to-Bandwidth ratio

Graphics Pipeline

- Vextex
  - BW: 1 vertex = 32 bytes;
  - OP: 100-500 f32-ops / vertex
- Fragment
  - BW: 1 fragment = 10 bytes
  - OP: 300-1000 i8-ops/fragment

Considering Readback

- GPUs need to download and readback results
  - Time to complete = download + compute + readback
  - Not a problem on CPU
- Readback
  - Getting a lot better!
    - > 600 MB/sec NVIDIA OpenGL
    - GL_UNSIGNED_BYTE: BGRA
    - Floating Point: RGBA

Strategies & Tricks:

Understanding Floating Point

Floating Point Precision

s	exponent	mantissa

sign * 1.mantissa * $2^{(exponent+bias)}$

- NVIDIA FP32
  - s23e8
- ATI 24-bit float
  - s16e7
- NVIDIA FP16
  - s10e5

Floating Point Precision

- Common Bug
  - Pack large 1D array in 2D texture
  - Compute 1D address in shader
  - Convert 1D address into 2D
- FP precision will leave unaddressable texels!

Largest Counting Number

NVIDIA FP32:	16,777,217
ATI 24-bit float:	131,073
NVIDIA FP16:	2,049

Strategies & Tricks:

Implementing Scatter

```
a[i] = p
```

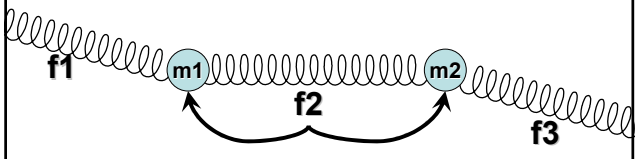

Scatter Techniques

- Problem: a[i] = p
  - Indirect write
  - Can't set the x,y of fragment in pixel shader
  - Often want to do: a[i] += p

Scatter Techniques

- Solution 1: Convert to Gather

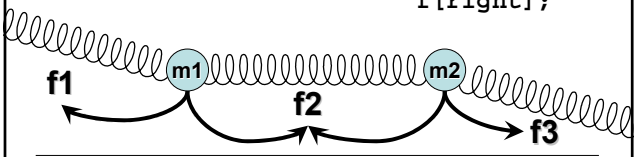
```
for each spring
    f = computed force
    mass_force[left]  += f;
    mass_force[right] -= f;
```



Scatter Techniques

- Solution 1: Convert to Gather

```
for each spring
    f = computed force
for each mass
    mass_force = f[left] -
                 f[right];
```



Scatter Techniques

- Solution 2: Address Sorting
  - Sort & Search
    - Shader outputs destination address and data
    - Bitonic sort based on address
    - Run binary search shader over destination buffer
      - Each fragment searches for source data

Scatter Techniques

- Solution 3: Vertex processor
  - Render points
    - Use vertex shader to set destination
    - or just read back the data and re-issue
  - Vertex Textures
    - Render data and address to texture
    - Issue points, set point x,y in vertex shader using address texture
    - Requires texld instruction in vertex program

Strategies & Tricks:

Conditionals

Conditionals

- Problem:

```
if (a) b = f();
else   b = g();
```

- Limited fragment shader conditional support

Conditionals

- Solution 1: Predication
  - Execute both f and g

```
if (a) b = f();
else   b = g();
```

- Use LRP instruction
  - LRP b, a, f, g b = a ? f : g
  - Executes all conditional code

Conditionals

- Solution 1: Predication
- Use DP4 instruction
  - DP4 b.x, a, f
  - Executes all conditional code

```
a = (0, 1, 0, 0)
f = (x, y, z, w)
```

```
if (a.x) b = x;
else if (a.y) b = y;
else if (a.z) b = z;
else if (a.w) b = w;
```


Conditionals

- Solution 2: Using early Z-kill
  - Set Z buffer to a
    - Clear Z to 1.0f
    - Render quad at z=0.3
    - Evaluate conditional and kill to set Z

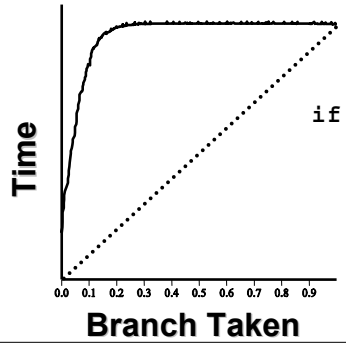
```
if (a) b = f();
else   b = g();
```

Conditionals

- Solution 2: Using early Z-kill
  - Set Z buffer to a
  - Z-test can prevent shader execution
    - glEnable(GL_DEPTH_TEST)
  - Good only if locality in conditional

```
if (a) b = f();
else   b = g();
```

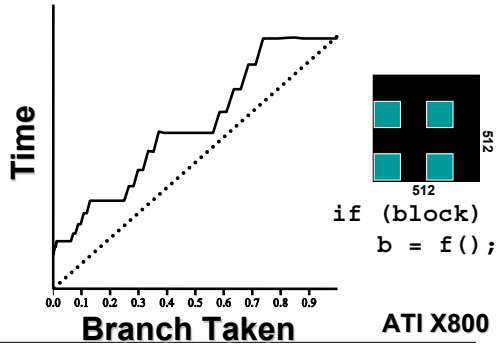
Conditionals




```
if (rand)
   b = f();
```

ATI X800

Conditionals




```
if (block)
   b = f();
```

ATI X800

Conditionals

- Solution 2: Using early Z-kill
  - Very Sensitive!

ATI:
- **Z out in shader**
- **alpha test enabled**
- **texkill is shader**

NV3X & NV4X:
- **Changing depth test direction invalidates for remainder of frame**

NV3X:
- **Alpha test, alpha-to-coverage, user clip planes,**
- **Pixel kill in shader (KIL), shader alters Z**

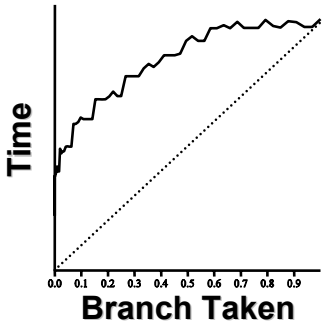
NV4X:
- **Writing stencil while rejecting based on stencil**
- **Changing stencil func/ref/mask invalidates for remainder of frame**

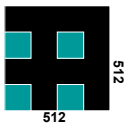
Conditionals

- Solution 3: Conditional Instructions
  - Available with NV_fragment_program2

```
MOVC CC, R0;
IF GT.x;
MOV R0, R1; # executes if R0.x > 0
ELSE;
MOV R0, R2; # executes if R0.x <= 0
ENDIF;
```

Conditionals






```
if (block)
   b = f();
```

GeForce 6800

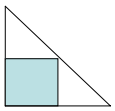
Strategies & Tricks:

Optimizing Execution

Optimizing Execution

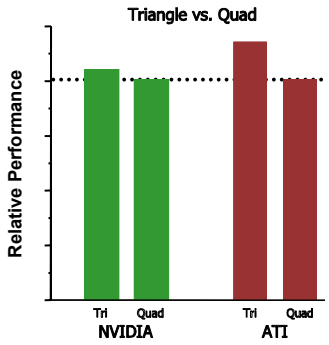
- Two methods for GPGPU shader execution

```
glBegin(GL_QUADS);
glVertex2f(left,  bottom);
glVertex2f(right, bottom);
glVertex2f(right, top);
glVertex2f(left,  top);
glEnd();
```



```
glViewport(0,0,width,height)
glBegin(GL_TRIANGLE);
glVertex2f(  0,       0);
glVertex2f(width*2, 0);
glVertex2f(  0,       height*2);
glEnd();
```


Optimizing Execution



Strategies & Tricks:

Multiple Outputs

Multiple Outputs

- Software solution
  - Let driver, cgc, or fxc do dead code elimination

```
kernel void foo (float3 a<>,
                 float3 b<>, …,
                 out float3 x<>,
                 out float3 y<>)
```

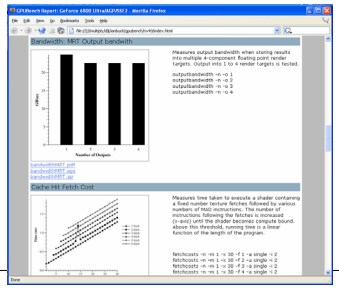
  - Works well if shader is separable

```
kernel void foo1(float3 a<>,
                 float3 b<>, …,
                 out float3 x<>)
```

```
kernel void foo2(float3 a<>,
                 float3 b<>, …,
                 out float3 y<>)
```


GPUBench

- http://graphics.stanford.edu/projects/gpubench
- http://sourceforge.net/projects/gpubench



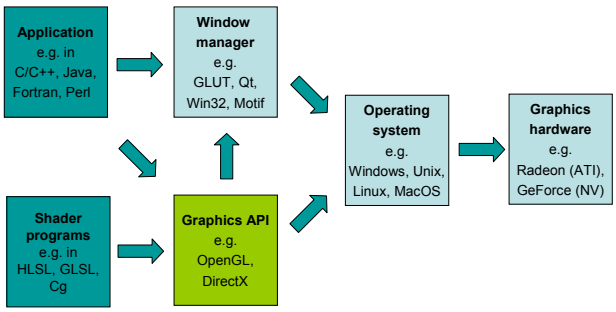
Developer Tools

GPGPU

Robert Strzodka
caesar research center
Bonn, Germany

VIS 2004

Choices in GPU Programming

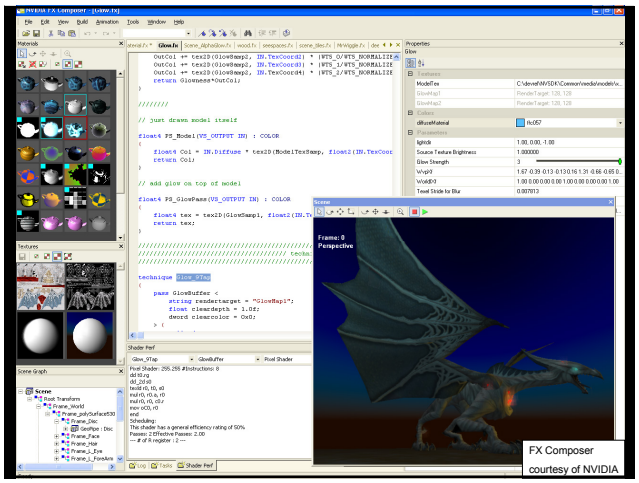


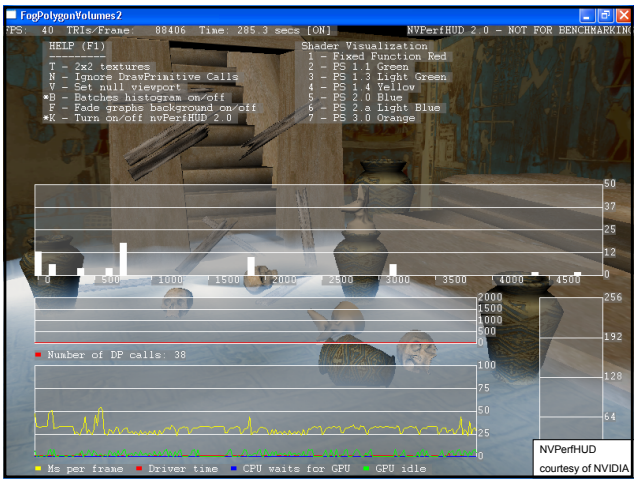
DirectX Choices

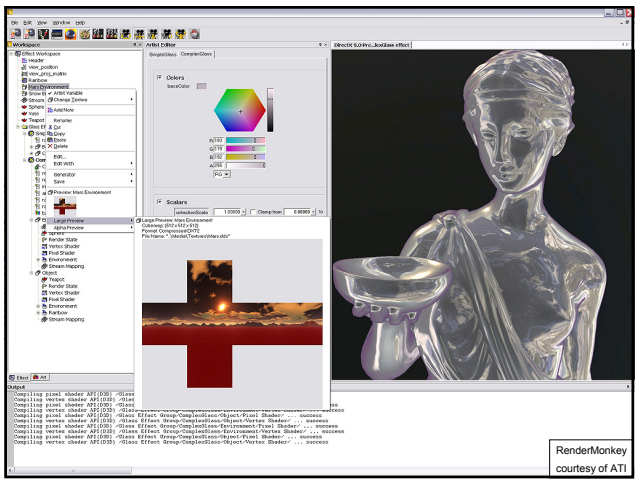
- Operating System
  - Windows
- Window manager
  - Win32, Qt
- Programming languages
  - C/C++/C#, Basic, Delphi, Perl, .NET Framework
- Shader languages
  - HLSL, Cg, DirectX pixel- and vertex-shader
- Stream languages
  - CgFX, Brook

DirectX Tools

- FX Composer, NVPerfHUD (NVIDIA)
  - HLSL shader IDE and performance analysis, real-time statistics
  - www.developer.nvidia.com/page/tools.html
- RenderMonkey (ATI)
  - HLSL, GLSL shader IDE and performance analysis
  - www.ati.com/developer/tools.html
- EffectEdit (Microsoft)
  - Interactive HLSL renderer
  - msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/TutorialsAndSamples/Samples/EffectEdit.asp
- ShaderWorks (Mad Software)
  - HLSL shader IDE
  - www.shaderworks.com







OpenGL Choices

- Operating System
  - Linux/Unix, Windows, MacOS, OS/2, BeOS
- Window manager
  - GLUT, Qt, Motif
- Programming languages
  - C/C++, Java, Fortran, Ada, Python, Perl, Pike
- Shader languages
  - GLSL, Cg, OpenGL fragment- and vertex-shader
- Stream languages
  - Brook, Sh

OpenGL Tools

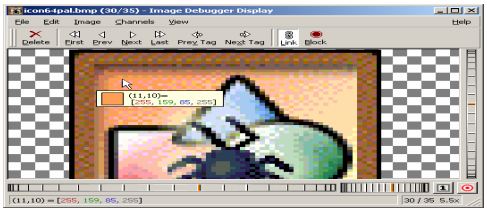
- NVShaderPerf (NVIDIA)
  - HLSL, OpenGL fragment shader performance analysis
  - www.developer.nvidia.com/page/tools.html
- RenderMonkey (ATI)
  - HLSL, GLSL shader IDE and performance analysis
  - www.ati.com/developer/tools.html
- Babelshader (D. Horn)
  - Translator: DirectX pixelshader to OpenGL fragment shader
  - www.graphics.stanford.edu/~danielrh/babelshader.html
- OpenGL Panther Tools (Apple)
  - OpenGL vertex and fragment shader IDE, profiling tools
  - developer.apple.com/opengl/panther.html
- OpenGL Shader Designer (Typhoon Labs)
  - GLSL shader IDE
  - www.typhoonlabs.com

Shader Debugger

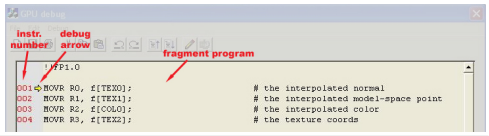
- Visual debugging with the shader IDEs (Windows)
  - FX Composer (DX), RenderMonkey (DX&GL), EffectEdit (DX), ShaderWorks (DX), OpenGL Panther (GL)
- Shader Debugger Tool (Microsoft)
  - HLSL debugger extension for Visual Studio IDE
  - msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/tools/shaderdebugger.asp
- Imdebug – The Image Debugger (B. Baxter)
  - Analysis of images output by shaders, easy integration
  - www.cs.unc.edu/~baxter/projects/imdebug/
- Shadesmith (T. Purcell, P. Sen)
  - Interactive OpenGL fragment shader debugger
  - graphics.stanford.edu/projects/shadesmith/

Shader Debugger

imdebug



Shadesmith



images courtesy of Tim Purcell

Many More Tools

- Plug-ins for various content creator programs
- Texture conversion and compression
- Normal maps
- Mesh optimization

- Available online for free
  - www.msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/Tools/Tools.asp
  - www.developer.nvidia.com/page/tools.html
  - www.ati.com/developer/tools.html

How To Get Started?

- Visit the GPGPU base: papers, code, news, links
  - **www.gpgpu.org**
- Get the 'Hello GPGPU' example and experiment
  - www.gpgpu.org/developer/
- Get a SDK with plenty of examples
  - www.developer.nvidia.com/object/sdk_home.html
  - www.ati.com/developer/radeonSDK.html
- For discussion go to
  - www. gpgpu.org/forums
  - www.shadertech.com

Interactive Level-Set Deformation On the GPU

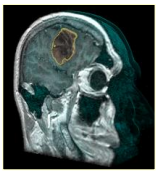
GPGPU

Aaron Lefohn
Institute for Data Analysis and Visualization
University of California, Davis

VIS 2004

Problem Statement

- Goal
  - Interactive system for deformable surface manipulation
  - Level-sets
- Challenges
  - Deformation is slow
  - Deformation is hard to control
- Solution
  - Accelerate level-set computation with GPU
  - Visualize computation in real-time



Collaborators

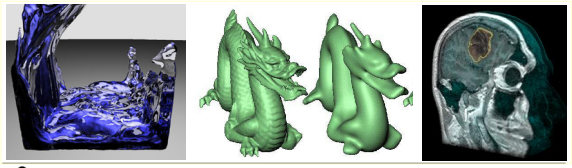
University of Utah
- Joe Kniss
- Joshua Cates
- Charles Hansen
- Ross Whitaker

Overview

- Why?
  - Motivation and previous work
- How?
  - Streaming level-set algorithm
  - Real-time visualization
  - Segmentation application
- Does it work?
  - Demonstration
  - User study

Deformable Surfaces

- Applications of Level-Sets
  - Fluid simulation
  - Surface reconstruction for 3D scanning
  - Surface processing
  - Image / Volume segmentation

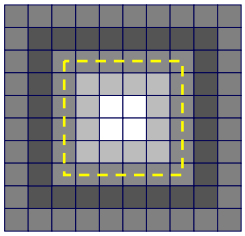


Level-Set Method

- Implicit surface

$$\mathbf{S}_t = \{\mathbf{x} | \phi(\mathbf{x}, t) = k\}$$

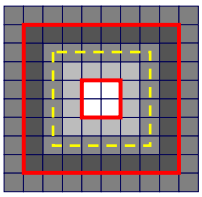
- Distance transform
  - ϕ denotes inside/outside



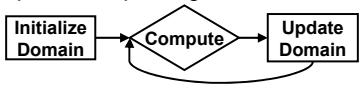
- Surface motion
  - $\phi(\mathbf{x}, t + \triangle t) = \phi(\mathbf{x}, t) + \triangle t F |\nabla \phi|$
  - F = Signed speed in direction of normal

CPU Level-Set Acceleration

- Narrow-Band/Sparse-Grid
  - Compute PDE *only* near the surface
    - *Adalsteinson et al. 1995*
    - *Whitaker et al. 1998*
    - *Peng et al. 1999*



- Time-dependent, sparse-grid solver



GPU Level-Set Acceleration

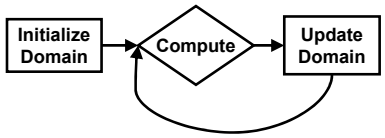
- Strzodka et al. 2001
  - 2D level-set solver on NVIDIA GeForce 2
  - No narrow-band optimization
- Lefohn et al. 2002
  - Brute force 3D implementation on ATI Radeon 8500
  - No faster than CPU, but ~10x more computations
  - No narrow-band optimization

Overview

- Why?
  - Motivation and previous work
- How?
  - Streaming narrow-band
  - Real-time visualization
  - Segmentation application
- Does it work?
  - Demonstration
  - User study

GPU Narrow-Band Solver

- Sparse Volume Computation
  - CPU algorithm: Traverse list of active voxels
  - GPU algorithm: Compute all active voxels in parallel



- Data structures change after each PDE time step

Algorithm Goals

GPU Narrow-Band Solver

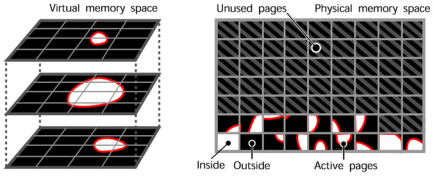
- Goals
  1. Leverage GPU parallelism
  2. Perform sparse computation
  3. Minimize GPU memory usage
  4. Fast update of sparse data structures
  5. Interactive visualization

Algorithm Solutions

- Pack Active Voxels Into 2D Texture
  - Increase parallelism, reduce computation and memory use
- Efficient GPU-to-CPU Message Passing
  - Fast update of packed data structure
- On-The-Fly Decompression Volume Rendering
  - Interactive visualization without increasing memory use

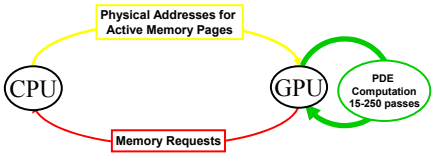
A Dynamic, Sparse GPU Data Structure

- Multi-Dimensional Virtual Memory
  - 3D virtual memory
  - 2D physical memory
  - 16 x 16 pixel pages



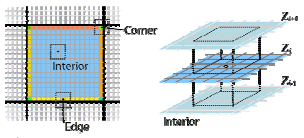
A Dynamic, Sparse GPU Data Structure

- GPU: Computes PDE
  - Level-set computation (2D physical memory)
  - Issues memory requests
- CPU: Manages memory
  - Memory manager
  - Page table (3D virtual memory)



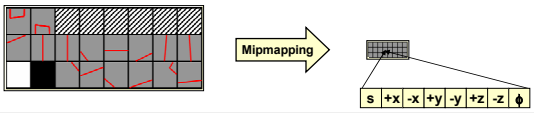
A Dynamic, Sparse GPU Data Structure

- Problem
  - Neighbor lookups across page boundaries
  - Branching slow on GPU
- Solution
  - Substreams
    - Create homogeneous data streams
    - Resolve conditionals with geometry
    - Lefohn 2003, Goodnight 2003, Harris 2003
    - Optimizes cache and pre-fetch performance
      - Kapasi et al., Micro 33, 2000



GPU-to-CPU Message Passing

- Problem: Active Voxel Set is Time-Dependent
  - GPU memory request mechanism
  - Low bandwidth GPU-to-CPU communication
- Solution
  - Compress GPU memory request
  - Use GPU computation to save GPU-to-CPU bandwidth

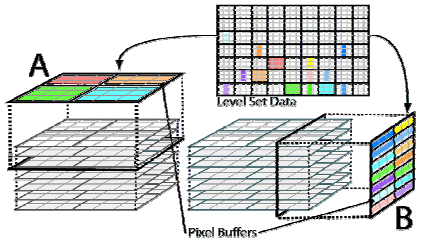


Overview

- Why?
  - Motivation and previous work
- How?
  - Streaming level-set algorithm
  - Real-time visualization (with Joe Kniss)
  - Segmentation application
- Does it work?
  - Live demonstration
  - User study

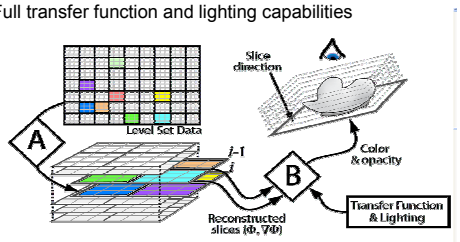
Direct Volume Rendering of Level Set

- Reconstruct 2D Slice of Virtual Memory Space
  - On-the-fly decompression on GPU
  - Use 2D geometry and texture coordinates



Direct Volume Rendering of Level Set

- Deferred Filtering: Volume Rendering Compressed Data
  - 2D slice-based rendering: *No data duplication*
  - Tri-linear interpolation
  - Full transfer function and lighting capabilities



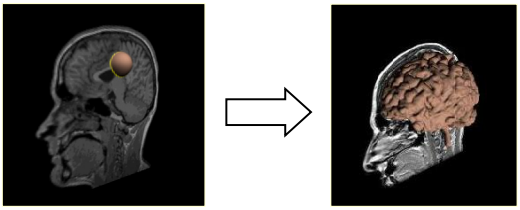
Overview

- Why?
  - Motivation and previous work
- How?
  - Streaming level-set algorithm
  - Real-time visualization
  - Segmentation application
- Does it work?
  - Demonstration
  - User study

Application

Level-Set Segmentation Application

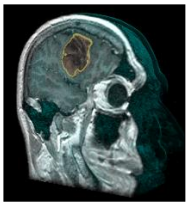
- Idea: Segment Surface from 3D Image
  - Begin with “seed” surface
  - Deform surface into target segmentation



Results

Demo

- Segmentation of MRI volumes
  - 128^3 scalar volume
- Hardware Details
  - ATI Radeon 9800 Pro
  - 2.4 GHz Intel Pentium 4
  - 1 GB of RAM



Results

GPU Narrow-Band Performance

- Performance
  - 10x – 15x faster than optimized CPU version (Insight Toolkit)
  - Linear dependence on size of narrow band
- Bottlenecks
  - Fragment processor (~80%)
  - Conservative time step
    - Need for global accumulation register (min, max, sum, etc.)

Overview

- Why?
  - Motivation and previous work
- How?
  - Streaming level-set algorithm
  - Real-time visualization
  - Segmentation application
- Does it work?
  - Demonstration
  - User study (with Josh Cates)

Evaluation User Study

- Goal
  - Can a user quickly find parameter settings to create an accurate, precise 3D segmentation?
  - Relative to hand contouring

User Study Results

- Efficiency
  - 6 ± 3 minutes per segmentation (vs multiple hours)
  - Solver idle 90% - 95% of time
- Precision
  - Intersubject similarity significantly better
  - 94.04% ± 0.04% vs. 82.65% ± 0.07%
- Accuracy
  - Within error bounds of expert hand segmentations
  - Compares well with other semi-automatic techniques
    - Kaus et al., Radiology, 2001

Summary

- Interactive Level-Set System
  - 10x – 15x speedup over optimized CPU implementation
  - Intuitive parameter tuning
  - User study evaluation
- Algorithm Developments
  - Multi-dimensional virtual memory
  - Substreams
  - GPU-to-CPU Message passing
  - Volume rendering packed data

Future Directions

- Other Level-Set Applications
  - Surface processing, surface reconstruction, physical simulation
- Integrate GPGPU Code Into Open Source Software
  - The Insight Toolkit (www.itk.org)?
- “Interactive Visulation”
  - GPGPU allows for simultaneous visualization and simulation
  - What problems can be solved with “interactive visulation?”
  - What is the user interface for a visulation?

Acknowledgements

- Joe Kniss – Volume rendering
- Josh Cates – Tumor user study
- Gordon Kindlmann – "Teem" raster-data toolkit
- Milan Ikits – "GLEW" OpenGL extension wrangler

- Ross Whitaker, Charles Hansen, Steven Parker and John Owens
- ATI: Evan Hart, Mark Segal, Jeff Royle, and Jason Mitchell
- Brigham and Women's Hospital

- National Science Foundation Graduate Fellowship
- Office of Naval Research grant #N000140110033
- National Science Foundation grant #ACI008915 and #CCR0092065
- ALIENWARE Interchangeable mobile GPUs

Questions?

For More Information

Google "Lefohn level set"
http://graphics.cs.ucdavis.edu/~lefohn/

Journal Papers Based on this Work

Lefohn, Kniss, Hansen, Whitaker, "**A Streaming Narrow Band Algorithm: Interactive Computation and Visualization of Level Sets,"** IEEE Transactions on Visualization and Computer Graphics, 10 (40), Jul / Aug, pp. 422-433, 2004

Cates, Lefohn, Whitaker, "**GIST: An Interactive, GPU-Based Level-Set Segmentation Tool for 3D Medical Images**," Medical Image Analysis, to appear 2004

Advanced Image Processing

GPGPU

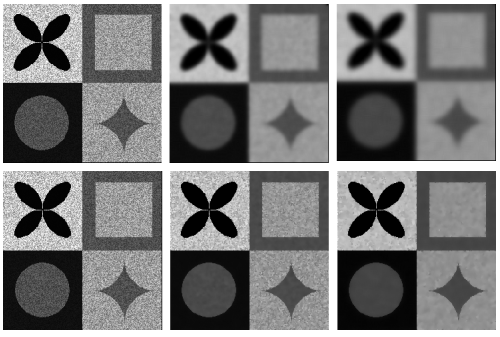
Robert Strzodka
caesar research center
Bonn, Germany

VIS 2004

Overview

- **Classic Image Processing**
  - Denoising
  - Segmentation
  - Registration
- **Computer Vision**
  - Object recognition
  - Object classification
  - Motion estimation

Denoising by a linear and a non-linear diffusion process



linear diffusion

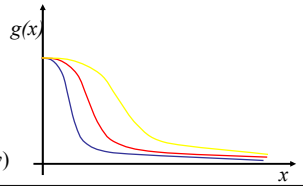
non-linear diffusion

Diffusion PDE

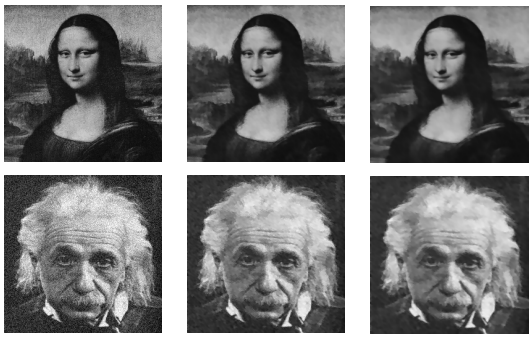
Initial image $u_0 : \Omega \to [0,1]$

$$\partial_t u - \operatorname{div}\left(G(\nabla u_\sigma)\nabla u\right) = 0 \qquad \text{in } \Re^+ \times \Omega$$
$$u(0) = u_0 \qquad \text{in } \Omega$$
$$\partial_\nu u = 0 \qquad \text{on } \Re^+ \times \partial\Omega$$

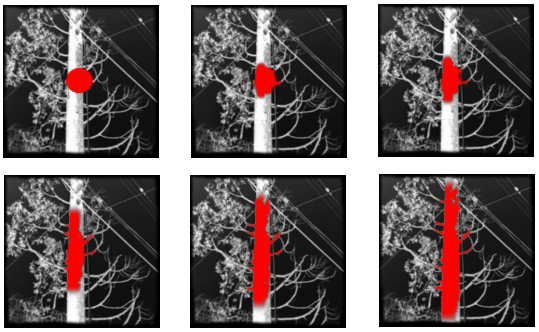
- linear
  $G(v) := 1$
- isotropic non-linear
  $G(v) := g(\| v \|)$ scalar
- anisotropic
  $G(v) := B^T(v)\begin{pmatrix} g_1(\|v\|) & 0 \\ 0 & g_2(\|v\|) \end{pmatrix} B(v)$



Denoising by anisotropic diffusion



Segmentation by the level-set method



Level-set equation

Initial image and level-set function $p, \varphi_0 : \Omega \to [0,1]$

$$\partial_t \varphi + f^\sigma[p,\varphi] \cdot \nabla\varphi = 0 \qquad \text{in } \Re^+ \times \Omega$$

$$\varphi(0) = \varphi_0 \qquad \text{in } \Omega$$

$$\partial_\nu \varphi = 0 \qquad \text{on } \Re^+ \times \partial\Omega$$

The level-set is driven by different forces $f^\sigma[p,\varphi] :=$

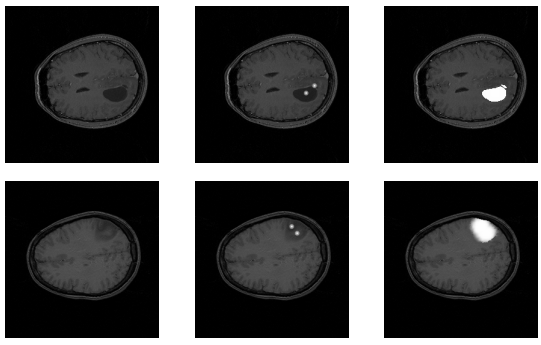
$$f_g^\sigma[p]\frac{\nabla\varphi}{\|\nabla\varphi\|} \quad + \quad f_\kappa[\varphi]\frac{\nabla\varphi}{\|\nabla\varphi\|} \quad + \quad f_V$$

image based forces dependent on $p, \nabla p$

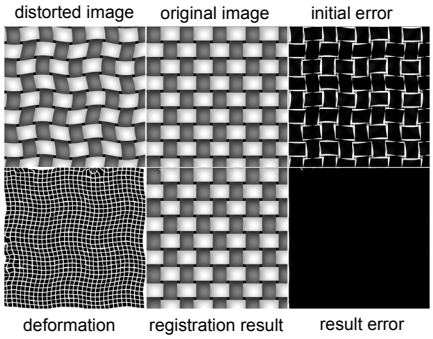
internal forces dependent on the form of level-sets, e.g. curvature $\kappa[\varphi]$

external forces, e.g. an advection field from a simulation

Segmentation by the level-set method



Registration by a reguralized gradient flow



Cascaded gradient flow on a multi-scale hierarchy

Input images $T, R : \Omega \to [0,1]$

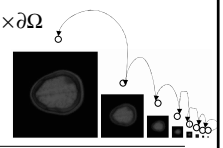
Energy measure $E[u] = \frac{1}{2}\int_{\Omega} |T \circ (\mathbf{1}+u) - R|^2$

Gradient regularization $A(\sigma) = \mathbf{1} - \frac{\sigma^2}{2}\Delta$

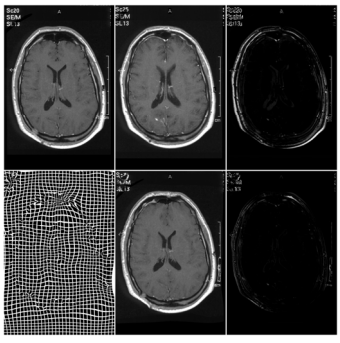
$$\partial_t u + A(\sigma)^{-1}\,\mathrm{grad}_{L^2} E[u] = 0 \quad \text{in } \Re^+ \times \Omega$$
$$u(0) = 0 \quad \text{in } \Omega$$
$$\partial_\nu u = 0 \quad \text{on } \Re^+ \times \partial\Omega$$

Multi-scale on multi-grid regularization

$$T_{\varepsilon^i} = S(\varepsilon^i)T, \quad R_{\varepsilon^i} = S(\varepsilon^i)R$$
$$E_{\varepsilon^i}[u] = \frac{1}{2}\int_{\Omega} |T_{\varepsilon^i} \circ (\mathbf{1}+u) - R_{\varepsilon^i}|^2$$



Registration by a reguralized gradient flow

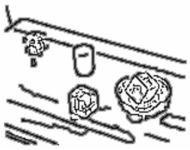


Overview

- **Classic Image Processing**
  - Denoising
  - Segmentation
  - Registration
- **Computer Vision**
  - Object recognition
  - Object classification
  - Motion estimation

Object recoginition by the Generalized Hough Transform

original image



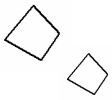
edge image

first detected cube

second detected cube

Generalized Hough Transform

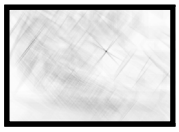
Generate poses of different perspective and scale



Store pose k as a list of offset vectors $l^{k,i}$

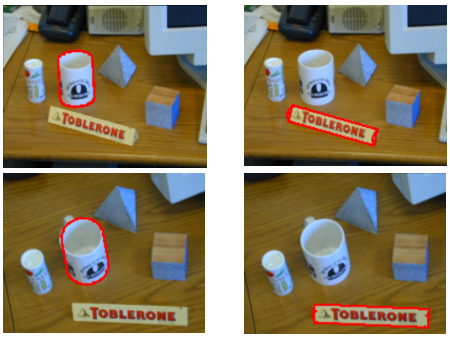
Draw image with offsets

normalize the result



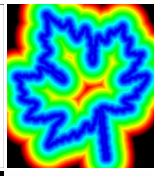
$$\overline{C}^k(x,y) = \sum_{i=1}^{|l^k|} I^{\text{edge}}(x + l_x^{k,i}, y + l_y^{k,i}) \qquad C^k(x,y) = \overline{C}^k(x,y) / |l^k|$$

Object recoginition by the Generalized Hough Transform



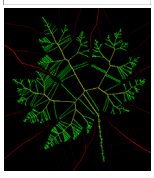
Object classification by skeletons

original boundary



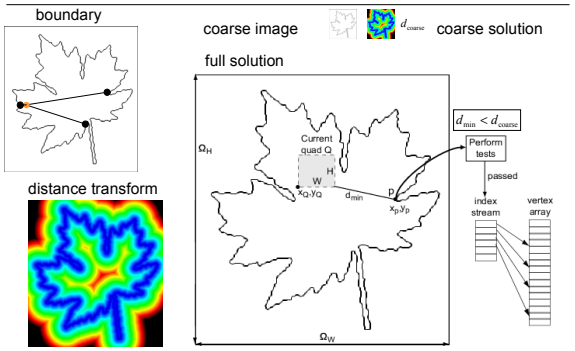
distance transform

fine skeleton

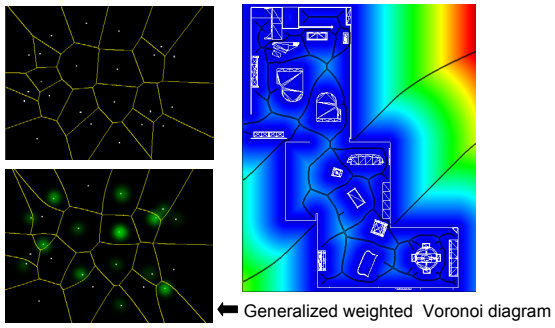


trimmed skeleton

Adaptive Generalized Distance Transform

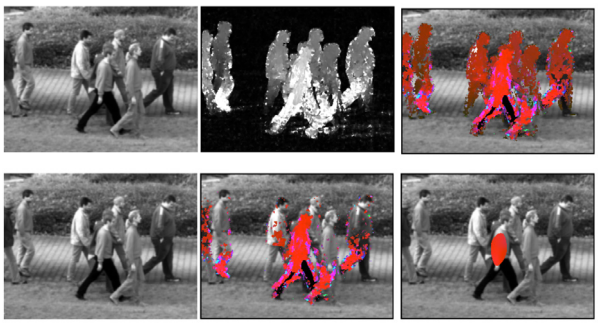


Distance Transforms and Voronoi Diagrams



Generalized weighted Voronoi diagram

Motion estimation by an eigenvector analysis of the spatio-temporal tensor



Motion estimates as weighted least square minimizers

Input image sequence $u(\xi): \Xi \to [0,1], \quad \Xi := (\Omega, \Re^+), \ \xi := (x,t)$

Brightness change constraint equation, one equation two unknowns

$$0 = \frac{du}{dt} = (\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial t}) \cdot (\frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, 1)^T =: d^T p$$

Local continuity assumption of the flow p gives the minimization problem

$$\int_\Xi w(\xi - \xi') \left(d^T(\xi') p(\xi')\right)^T \left(d^T(\xi') p(\xi')\right) d\xi' \approx p^T(\xi) J(\xi) p(\xi) \to \min$$

$$J(\xi) := \int_\Xi w(\xi - \xi') d(\xi') d^T(\xi') \, d\xi', \quad w: \Xi \to [0,1] \text{ weight function}$$

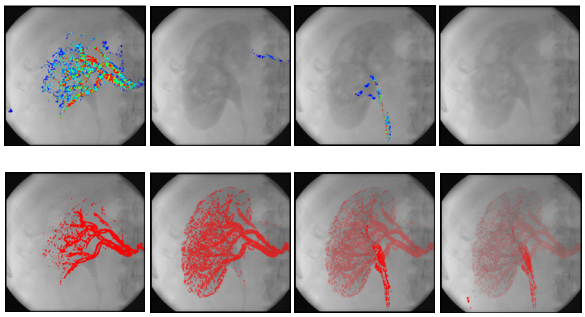
The diagonalization of the symmetric spatio-temporal 3x3 tensor

$J = V \Lambda V^T$, V eigenvector basis, Λ diagonal eigenvalue matrix

gives a motion estimation as the solution to the minimization problem

$p = (\frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, 1)^T = \frac{(v_x, v_y, v_t)^T}{v_t}$, v eigenvector to smallest eigenvalue

Motion estimation by an eigenvector analysis of the spatio-temporal tensor



References

Classic Image Processing

M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings ICIP'01*, volume 3, pages 1103–1106, 2001a.

M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84. Springer, 2001b.

M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings VIIP'01*, pages 193–202, 2001c.

R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 2004. to appear.

Computer Vision

R. Strzodka, I. Ihrke, and M. Magnor. A graphics hardware implementation of the generalized hough transform for fast object recognition, scale, and 3d pose detection. In *International Conference on Image Analysis and Processing (ICIAP 2003)*, pages 188–193, 2003.

R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym'04*, 2004.

R. Strzodka and C. Garbe. Real-time motion estimation and visualization on graphics cards. In *Proceedings Visualization '04*, 2004.

Homepage

http://www.numerik.math.uni-duisburg.de/people/strzodka/strzodka.htm

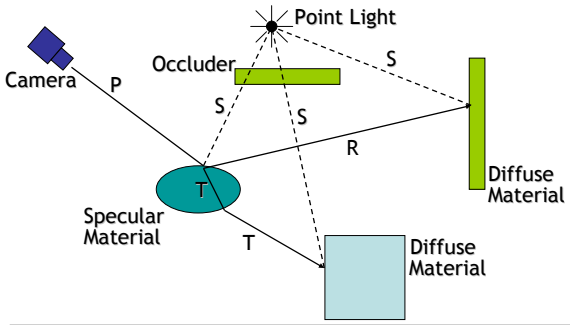
Ray Tracing on GPUs

GPGPU

Ian Buck
Graphics Lab
Stanford University

VIS 2004

Ray Tracing

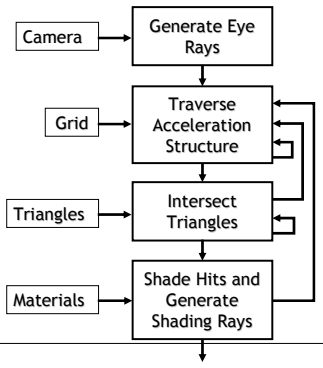


Ray Tracing slides courtesy of Tim Purcell

Implementation Options

- GPU as a ray-triangle intersection engine [Carr et al. 2002]
  - Rays and geometry streamed to GPU
  - Intersection calculation results read back
  - Acceleration structure traversal done on host CPU
- GPU as a ray tracing engine [Purcell et al. 2002]
  - Scene geometry and acceleration structure stored on GPU
  - GPU performs ray generation, acceleration structure traversal, intersection, and shading
  - Host provides camera info

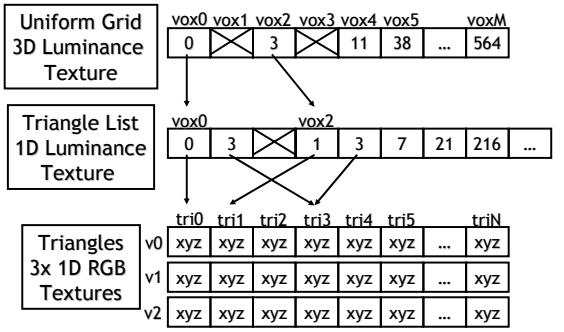
Streaming Ray Tracer



Techniques Used

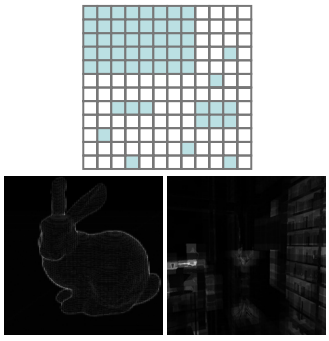
- Data structure navigation
  - Texture memory stores data structures
  - Dependent texture fetches walk through data
- Flow control
  - Kernel binding based on occlusion query results
  - Efficient selective execution of kernels using early-z occlusion culling
  - Difficulty in flow control disappearing with newest graphics cards
    - PS 3.0

Texture Memory Organization

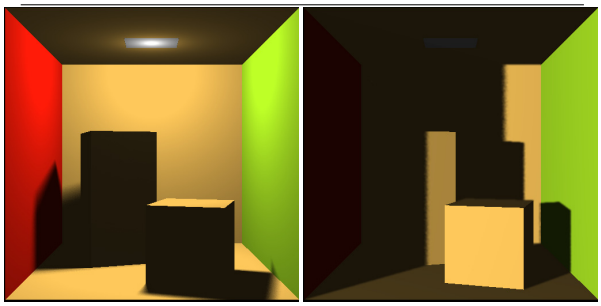


Efficient Selective Execution

- Rendering giant screen filling quad not ideal
  - Not all pixels need to process every rendering pass
- Use early fragment kill
  - Computation mask
  - Controllable early-Z occlusion culling
- Trade computation for bandwidth

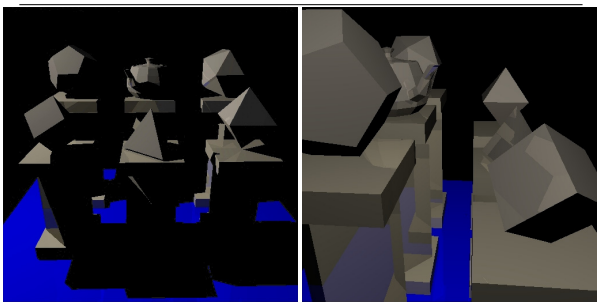


Cornell Box – Ray Traced Shadows



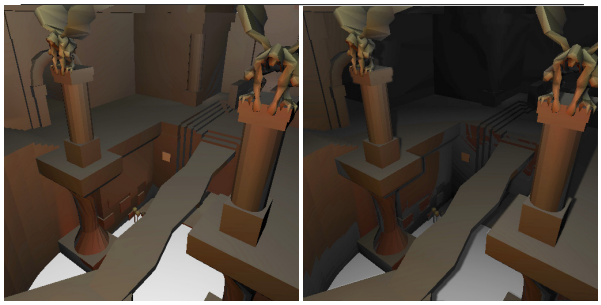
Rendered using a Radeon 9700 Pro

Teapotahedron



Rendered using a Radeon 9700 Pro

Quake 3 – Ray Traced Shadows



Rendered using a Radeon 9700 Pro

Performance Results

- Radeon 9700 Pro
  - 100M ray-triangle intersections/s
  - 300K to 4.0M rays/s
  - Between 3 – 12 fps @ 256x256 pixels
- CPU implementation
  - 20M intersections/s P3 800 MHz [Wald et al. 2001]
  - 800K to 7.1M ray/s 2.5 GHz P4 [Wald et al. 2003]
    - With simple shading: 1.8M to 2.3M rays/s

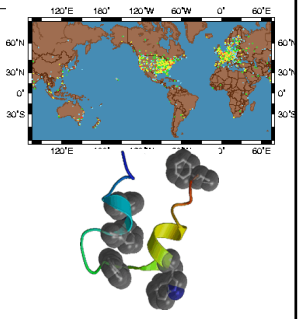
Molecular Dynamics on Graphics Hardware

GPGPU

Ian Buck
Graphics Lab
Stanford University

Folding@home: Vijay Pande

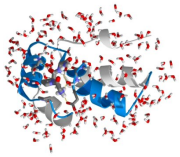
What does Folding@Home do?
Folding@Home is a distributed computing project which studies protein folding, misfolding, aggregation, and related diseases. We use novel computational methods and large scale distributed computing, to simulate timescales thousands to millions of times longer than previously achieved. This has allowed us to simulate folding for the first time, and to now direct our approach to examine folding related disease.



Results from Folding@Home simulations of villin

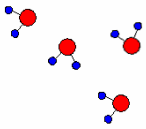
GROMACS: Erik Lindahl

- GROMACS provides *extremely high performance* compared to all other programs.
- Lot of algorithmic optimizations:
  - Own software routines to calculate the inverse square root.
  - Inner loops optimized to remove all conditionals.
  - Loops use SSE and 3DNow! multimedia instructions for x86 processors
  - For Power PC G4 and later processors: Altivec instructions provided
- normally 3-10 times faster than any other program.



Nonbonded forces

- Accounts for 80% of the runtime in C/Fortran code
- Most common form:



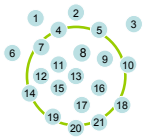
$$V_{nb} = \sum_{i,j} \left[\frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}} + \left(\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^{6}} \right) \right]$$

Electrostatics

Lennard-Jones

Using cutoffs & neighbor lists

- Neighbor list constructed every 10 steps.
- In practice: 10,000-100,000 atoms, with 100-200 neighbors in each list



Neighbor list for atom 13 =
{ 8, 9, 11, 12, 15, 16, 17}

What we do in the inner loop?

```
For each i atom {
    fetch atom i data
    i_force = 0;
    For each j atom in our neigborlist {
        fetch atom j data
        Calculate vectorial distance; dr = r_i-r_j
        Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
        Calculate potential and vectorial force
        Subtract the force from the j atom force
        i_force += force;
    }
    Store i_force;
}
```

Inner loop

What we do in the inner loop?

```
For each i atom {
    fetch atom i data
    i_force = 0;
    For each j atom in our neigborlist {
        fetch atom j data
        Calculate vectorial distance; dr = r_i-r_j
        Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
        Calculate potential and vectorial force
        Subtract the force from the j atom force
        i_force += force;
    }
    Store i_force;
}
```

Reading from memory

Writing to memory

What we do in the inner loop?

```
For each i atom {
    fetch atom i data
    i_force = 0;
    For each j atom in our neigborlist {
        fetch atom j data
        Calculate vectorial distance; dr = r_i-r_j
        Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
        Calculate potential and vectorial force
        Subtract the force from the j atom force
        i_force += force;
    }
    Store i_force;
}
```

Texture Fetch

Reading from memory

Texture Fetch

Writing to memory

What we do in the inner loop?

```
~~For each i atom {~~  For each fragment {
    fetch atom i data
    i_force = 0;
    For each j atom in our neigborlist {
        fetch atom j data
        Calculate vectorial distance; dr = r_i-r_j
        Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
        Calculate potential and vectorial force
        Subtract the force from the j atom force
        i_force += force;
    }
    Store i_force;
}
```

Texture Fetch

Reading from memory

Texture Fetch

Writing to memory

Output Color

What we do in the inner loop?

~~For each i atom~~ { **For each fragment {**
fetch atom i data ← **Texture Fetch**
i_force = 0; — Reading from memory
For each j atom in our neigborlist {
fetch atom j data ← **Texture Fetch**
Calculate vectorial distance; **dr = r_i-r_j**
Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
Calculate potential and vectorial force
Subtract the force from the j atom force ← **SCATTER**
i_force += force; — Writing to memory
}
Store i_force; ← **Output Color**
}

What we do in the inner loop?

~~For each i atom~~ { **For each fragment {**
fetch atom i data ← **Texture Fetch**
i_force = 0; — Reading from memory
For each j atom in our neigborlist {
fetch atom j data ← **Texture Fetch**
Calculate vectorial distance; **dr = r_i-r_j**
Calculate r^2=dx*dx+dy*dy+dz*dz, and 1/r=1/sqrt(r^2)
Calculate potential and vectorial force
~~Subtract the force from the j atom force~~ ← ~~SCATTER~~
i_force += force; — Writing to memory
}
Store i_force; ← **Output Color**
}

Perform each force computation twice

Inner loop

C Version

```
jnr = jjnr[k];
j3 = 3*jnr;
jx = pos[j3];
jy = pos[j3+1];
jz = pos[j3+2];
dx = ix - jx;
dy = iy - jy;
dz = iz - jz;
rsq = dx*dx+dy*dy+dz*dz;
rinv = 1.0/sqrt(rsq);
rinvsq  = rinv*rinv;
rinvsix = rinvsq*rinvsq*rinvsq;
tjA = ntiA+2*type[jnr];
vnb6 = rinvsix*nbfp[tjA];
vnb12 = rinvsix*rinvsix*nbfp[tjA+1];
vnbtot  = vnbtot + vnb12-vnb6;
qq  = iqA*charge[jnr];
vcoul = qq*rinv;
fs = (twelve*vnb12-
    six*vnb6+vcoul)*rinvsq;
vctot = vctot + vcoul;
tx = dx*fs;
ty = dy*fs;
tz = dz*fs;
fix = fix + tx;
fiy = fiy + ty;
fiz = fiz + tz;
```

Cg Version

```
jnr = f1tex1D (jjnr, k);

j = f3tex1D(pos, jnr);


d = i - j;


rsq = dot(d, d);
rinv = rsqrt(rsq);
rinvsq = rinv*rinv;
rinvsix = rinvsq*rinvsq*rinvsq;
tjA = ntiA+2*f1tex1D(type, jnr);
vnb6  = rinvsix*f1tex1D(nbfp, tjA);
vnb12 = rinvsix*rinvsix*f1tex1D(nbfp, tjA+1);
vnbtot = vnbot+vnb12-vnb6;
qq  = iqA * f1tex1D(charge, jnr);
vcoul = qq*rinv;
fs = (twelve*vnb12-
    six*vnb6+vcoul)*rinvsq;
vctot = vctot + vcoul;
t   = d * fs;


fi  = t;
```


Challenges

- Scalar inner loop code
  - Solution: Perform 4 force calc per loop iteration
- Duplicate force calculations
  - Bad: 2x computation than CPU
  - Good: Much less bandwidth!!!
    - Don't have to output partial forces
  - Overall bandwidth much more expensive than compute on GPUs
- Inner loop unrolling
  - 20 interactions before instruction limit

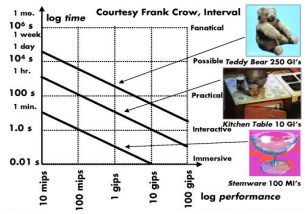
The Future: What's Next for GPUs?

GPGPU

John Owens
Department of Electrical and Computer Engineering
Institute for Data Analysis and Visualization
University of California, Davis

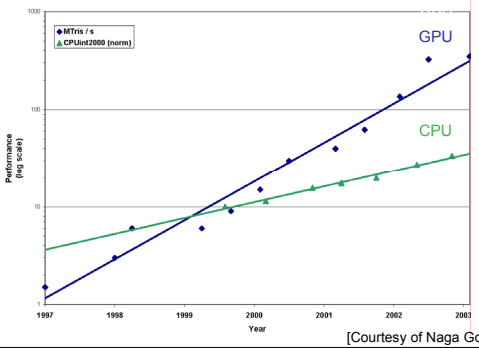
VIS 2004

Off-line to on-line to real-time …



[Courtesy of Crow/Hanrahan/Akeley]

Motivation: Computational Power



[Courtesy of Naga Govindaraju]

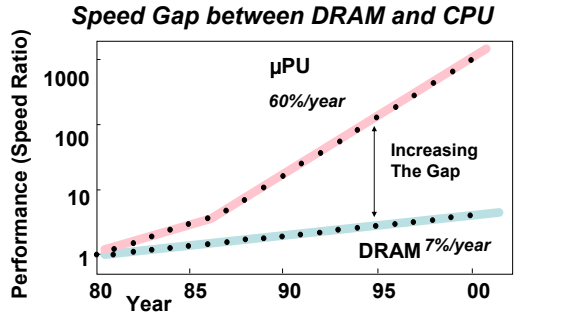
Semiconductor Scaling Rates

- From: *Digital Systems Engineering*, Dally and Poulton

Parameter	Current Value	Yearly Factor	Years to Double (Half)
Moore's Law (grids on a die)**	1 B	1.49	1.75
Gate Delay	150 ps	0.87	(5)
Capability (grids / gate delay)		1.71	1.3
Device-length wire delay		1.00	
Die-length wire delay / gate delay		1.71	1.3
Pins per package	750	1.11	7
Aggregate off-chip bandwidth		1.28	3

**** Ignores multi-layer metal, 8-layers in 2001**

DRAM "Memory Wall"

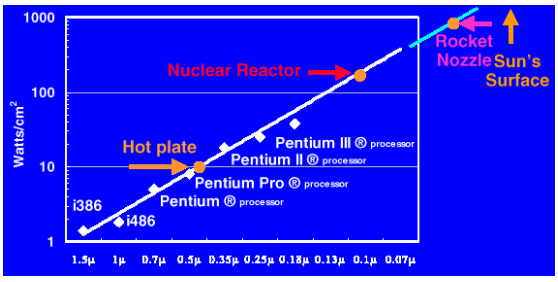


[Courtesy of Mark Horowitz, from Junji Ogawa 1998 presentation]

Hardware Considerations

- "Memory wall"
  - Continued migration of functionality onto GPU
    - Physics & simulation
    - Higher-level graphics functionality
- Size of design teams
  - Intel design teams increase in size 40% / generation
  - Validation for increasingly complex designs
- Power …

Power Considerations

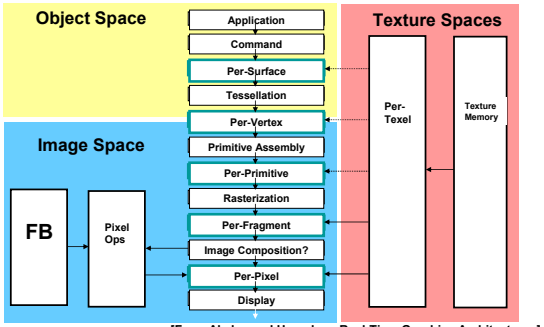


[Courtesy Bob Colwell]

Architecture/Microarchitecture

- Current programming model:
  - MIMD for vertex processing
  - SIMD for fragment processing
- Can we share units between the stages?
- To what will the instruction sets converge?
- Are these the only stages that will be programmable?
- How will the CPU interact with the GPU?
- How can we extend to multiple GPUs and multiple CPUs?

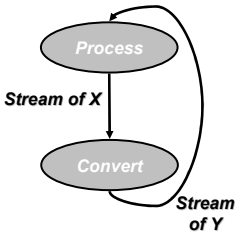
Generalized Graphics Pipeline?



[From Akeley and Hanrahan, Real-Time Graphics Architectures]

Future GPUs?

- Programmable stages operate on primitives ("process")
  - Fragment, vertex programs
- Hardwired or programmable stages "convert" one kind of primitive to another
  - Rasterization, composite
- Could define own pipelines!
  - Reyes, raytracing …



Algorithms

- Much to be done!
  - New/optimized stream algorithms
  - New features of graphics hardware
- Move from kernels to applications
  - Scientific computation
  - Simulation (game physics?) + visualization
  - What will be first "killer app" on GPUs?
- Ask for new features …
- … but don't lose what gives the GPU high performance

Tools and Programming Models

- CPU programmers have it easy!
  - Straightforward programming model
  - Many languages
  - Great compilers
  - Optimization tools
  - Debuggers
  - Profiling and performance tools
- GPU: Long way to go
  - Vendors working hard to provide these (but targeted primarily at games)
  - Active academic research
    - Brook is a great start, but domain specific languages and other design philosophies are needed
    - People who need these tools should help design them!

What *should* we map to GPUs?

- Problems with *high compute requirements*
- Problems with *regular structure*
- Problems with *predictable communication needs*
- Problems that require *interaction with the graphics system*
- Enormous opportunity at frontiers of applications, software, and hardware!

Backup slides

International Technology Roadmap '01